POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

A Study of Evasive Behaviors in Commercial Packers

Laurea Magistrale in Computer Science Engineering - Ingegneria Informatica

Author: Giorgio Coccia

Advisor: Polino Mario

Co-advisor: Carminati Michele, Zanero Stefano

Academic year: 2021-2022

1. Abstract

The goal of this research is to show that packers give a new layer of protection against dynamic analysis to the original file, in addition to compression and obfuscation. This research is based on the measurement of a program that was packed with 20 different packers. The instrumentation tool utilized can identify 45+ distinct strategies, allowing anti-debugging and anti-virtual machine methods to be detected. The study will not only show that there is a correlation between anti-debugging and packers, but it will also offer proof of the tactics used by each packer.

2. Introduction

The Cyber Defense and topic related to the Cyber Security have never been crucial as it is in these days. The Russian war against Ukraine [1] highlighted the importance of being Cyber protected for companies, nations, and for a single human. Every day we can hear of a group of hackers leaking information, dropping down and owning organizations, or declaring war on the heads of the society[2]. In these settings, discovering and repressing malicious attacks can be decisive for the protection of your system and information. Malware and virus are constantly looking for unknown methods to avoid antivirus. In the race between the malicious programs and defensive tools, a relevant spot is detected by packers. A packer [12] [4] is a tool that guarantees obfuscation and compression to a program. This means that it protects the content of the file, making it difficult to read, and shrink it, making it smaller and portable.

Despite the original benign intent of the packers, they became a valuable resource for the virus. The obfuscation feature is exploited to hide the malicious content of a program from antivirus, while the compression helped in spreading small and effective trojans. While obfuscation and polymorphism are great features against static analysis, this research is gonna focus on the possibility that packers are moving towards a new state of protection, adding features that protect also the file from dynamic analysis. We are gonna detect the evasive behaviors of a packed file, which means anti virtual machine technologies, anti-debugging techniques, or for example different ways to slow down the analysis. The innovative aspect is that this research exploits a new way of analyzing the sample, the instrumentation [6]. While using a debugger would take time to discover and reverse the whole code, the instrumentation makes it possible to detect when a specific function is executed and apply routines that may avoid the technique and collect all the logs in the meanwhile. [11][9].

3. Implementation

The frameworks used for this research were tools used to analyze malware in previous papers. Starting from the original version, they were updated with features that suited better the scope of this research, the packers. The framework is composed of Brioscia, a Client- Server structure written in python that permits to detonate of malware in a safe environment, collecting and parsing the results. The most relevant changes were done in the parser, a piece of code built from scratch that permitted distinct new techniques while producing an excel to summarize the results for each packer. The newly added technique are indicators that may evidence the presence of a debugger and are shown in the Table 1. The first part of the table is dedicated to driver names that may be opened by the packed sample for anti-debugging purposes [3]. Following, there is **nkvwovuotd.exe** which 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 "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\filename" where it may be stored the NtGlobalFlag value (a flag that is set to 1 while debugging a program). Then there's the **cng** library, which is used to encrypt files, and **computer name fields**, which may be examined in the program to check the names of virtual machines. The last one is **Session Manager** , which is part of the path "HKLM\System\CurrentControlSet\Control\Session Manager" another way to check the presence of the NTFlag [10].

The last changes applied are methods to obtain better results, like ignoring multiple calls of the same function, or ignoring the logs which are equal to the log data of the starting program. In this way, it is feasible to study only logs that

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"

Table 1: Table containg the blacklist used during the parsing procedure.

are generated by a line of code inserted by the packer. The second tool utilized is the Intel Pin tool Brioscia, already used in the papers [11][9]. This is an instrumentation tool written in CPP and contains an advanced pool of 45+ techniques and various useful features against anti-analysis techniques. The tool itself needed an update to the last SDK of Visual Studio, but the most relevant changes were introduced by new techniques that were found in papers and anti-debugging articles [10][5]. In particular, some of the new hooks added were implemented to verify evasive techniques based on the processes management. These types of techniques are based on studying the father of the running process. When a debugger starts a program, it becomes the father of the process: if the father of the sample is Explorer.exe, it is an expected behaviour because generated by double-clicking in the system files. While, if the father is a debugger or some other process, the packed file could decide to terminate the program because there is a high probability of being debugged. The hooks needed to discover this techniques are calls like the function **Process32Next()** or **Create32SnapshotTool()**, both of them providing information about a process. Another technique added which is specific to mew11, it's a technique that exploits the memory permission management. Some parts of the memory of the process are labeled as Read_only and a debugger can't add any interrupt. One of these parts of memory is the headers, and one of the techniques is implemented by moving the code in the headers part, preventing the debugger to add any kind of line of code (Figure 1).
Following there is the hooked function **Block-**

Algorithm 1 Piece of code added to check jmp in the headers section. I is the single instruction line while C is the set of all instructions executed in the code.

```
1: Instruction
2: for $I - In - C$ do
3:     if $I.operand == jmp$ then
4:         if $0x400000 < I.memoryLocation < 0x401000$ then
5:             Jmp Headers Found
6:         end if
7:     end if
8: end for
```

Input(). This function is used for blocking every input of the keyboard and the mouse. It becomes difficult to debug the application in this way since it is impossible to type on the keyboard or pick tools in the Debugger GUI. It is implemented simply by hooking the function and printing the result. Another function added is **GetVersion()**, this function is used to ensure that the descriptor table layout matches the operating system platform, discovering if a system is being emulated. The hook to **OpenProcess()** has been used for multiple purposes, first to check when the program is looking for the information of a specific process, second for checking if the program tried to open csrss.exe. 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 an error, and demonstrate that the file is probably being debugged. The last technique added is a check on the Guard Page exception code. This technique regards the EXCEPTION_GUARD_PAGE (0x80000001) [10] 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. **DebugActiveProcessStop()** is the last function added. 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 DebugActiveProcessStop() uses the identical procedure, but instead of launching a process, it tries to stop one that is already being debugged.

4. Results

The major goal of this study is to show that at least half of the packers chosen to use one evasive technique or more. Because some of the functions involved may be employed for different purposes, we split the outcomes into certain and uncertain techniques during our research. Despite this categorization, we found that the main goal was met, with 12 out of 20 packers using at least one technique, regardless of whether or not included uncertain techniques. The results obtained are parsed in categories (Table 2), each focusing on a different component of the running program in order to detect a debugger The anti-debugging category is a set of functions and methods that are recognized to be used for anti-debugging purposes and are planned for checking the debugger presence. Same story for anti virtual machine category, but reference to the capability of discovering a virtual machine. File and Registry categories contain both anti-VM and anti-debugging techniques, with the difference that the methods are applied by looking into registries or file descriptors. The last ones are stalling and timing, where the difference is that the evasive behavior can be found by waiting or measuring the execution time of instructions. In particular, timing consists in using functions that measure the time of execution of a portion of code and compare it with the timing obtained without a debugger. When a program is being debugged the execution time is delayed because it adds routing e new instructions. Stalling is referencing the attempt of delaying and enlarging the timing of the execution, for escaping the execution under a sandbox, and slowing down the analysis of a debugger. A sandbox is a virtual machine used to automatically analyze and run a program for a limited amount of time. As it can be seen, in each category we have multiple hits and a big variety of different evasive techniques.

Packers implementing Anti Dynamic Analysis ●

	Alternate	mpress2	pecompact	obsidium	enigmap	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	enigmav	yp	themida1	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 2: 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.

In the table 2 we can see the number of techniques implemented for each category. We can evidence that the packers not involving evasion are **Upx, Mpress, Aspack, Pecompact, Rlp, kkrunchy, petite, AlternateExe**, most of them are programs that derive from Upx, one of the oldest and most used compression packer. We demonstrated that Upx can be debugged without finding any problem in the complete version of the thesis. Because of this, the result may propagate in the other packers that used upx as a starting point and updated only the obfuscation routines, without adding any anti-debugging method. Because of a large number of functions and methods, it is impossible to describe every result in this paper as it is done in the original thesis, but we can provide the list of techniques seen at least one time in the research (Tab. 3) and the logs of at least one category (Tab 4).
In the Anti debugging functions (4) we can find relevant results, like 7 occurrences of IsDebuggerPresent or CheckRemoteDebuggerPresent, a unique result proofing the anti-debugging intentions without any doubts. But this technique is just the beginning, following to it some well known and recognizable techniques for checking debugging flags (NtQueryInformationProcess [7], NtGetContextThread) or functions exploiting exceptions (CloseHandles(),Interrupts). A special mention goes to **FindWindow**, used in this case by **Themida and Yoda's protector** to check the presence of Procmon [8] and the shell process.

Section	Technique
Antidebug	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 ●
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) ●
File	NtCreateFile(\\??\\global\\procmondebuglogger) ●
	NtCreateFile(\\??\\ntice), ●
	NtCreateFile(\\??\\sice) ●
	NtCreateFile(\\??\\siwvidstart), ●
	NtCreateFile(\\??\\spcommand) ●
	NtCreateFile(\\??\\syser), ●
	NtCreateFile(\\??\\syserboot) ●
	NtCreateFile(\\??\\syserdbgmsg) ●
	NtCreateFile(\\??\\global\\procmondebuglogger) ●
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) ●
Stalling	NtDelayExecution() ●
	SetTimer() ●
	Sleep/SleepEx() ●
	waitForSingleObject/Ex() ●
Timing	GetLocalTime ◐
	GetTickCount ◐
	GetTickCount64 ○
	Instruction: RDTSC/D ●
	QueryPerformanceCounter ○
	timeGetTime ◐

Table 3: Collection of all the techniques found divided by categories.
○False Positive, ◐Uncertain Technique,●Certain technique.

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 4: Anti debugging logs' results.

```
->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)
->Launched an exception with code 0xc0000005
```

Figure 1: Logs evidencing how yoda's protector is enumerating processes to find the descriptor of the current father's process.

5. False positives and Important findings

During our investigation, we came across some intriguing techniques in the logs that either confirmed the findings of the previous publications or revealed new techniques for specific packers. Yoda's Protector provides one of the most interesting outcomes. Beginning with the control of the parent process, this packer adds many strategies to the original code. We saw in the logs that the application listed all the processes until **pin.exe**, which is the father of the processes while performing instrumentation, and then terminated the execution with an exception (Fig. 1). The method is made up of the functions Create32Snapshot() and Process32Next(), and it has been defeated by interrupting the listing procedure. The function Process32Next(), in particular, allows a process to have the next element of a snapshot. If we constrict the method to return false when it is called the first time, the program can't' read all the list of processes and continued the execution. After this step we encountered new techniques like IsDebuggerPresent and BlockInput, demonstrating quite clearly how the packer is interested in adding a debugging protection.

The second interesting case is mew11, the only packer implementing the jmp headers technique. This type of technique has been described before and it is probably a singular technique implemented only by this packer. In the logs we found more than one jmp to the headers, especially at the beginning of the code, demonstrating that the intention was to prevent the debugger from analyzing the code at the beginning of its execution. The third relevant result is the anti virtual machine techniques found in several packers. These types of techniques look for indicators of a virtualized environment in multiple areas. The first and most important are the processes: we found that **Themida**,**Yoda's protector** and **Pelock** are looking for virtual box instances inside the processes. In the logs we found NtQuerySystemInformation() asking for information about virtual box processes like **VirtualBox.exe**,**VirtualBoxVm** and many others. We tried to run Themida while Procmon was running, and as well as in the previous case we found in the logs a call to NtQuerySystemInformation() asking for information about procmon64 processes. These may be used as a scan of processes for finding several apps used for the analysis. As for processes, some of the indicators are also searched in the file used, in particular, **Pelock** returned a large number of calls to the function NtQueryAttributeFile(), asking for the attributes of the file in input. This function is used to check the presence of two files used for the execution of the virtual box, **VBoxHook.dll** and **VBoxOGL.dll**. The last sources where we can look for the presence of a virtual environment are the quantity of memory of the environment, the network adapter, the name of the machine, or the hardware configuration. All of these cases are shown in our results in the thesis, but the most interesting one is a check performed by **Themida**. We found in the logs of this packers two calls for the In instruction (IN(0x564d5868, 0x00005658) and IN(0x68584d56, 0x77875856)). The In instruction copies the values of the I/O port specified inside a register, but some of the values of VBox and VMWare are by default known. In this case, Themida checks for the big-endian and little-endian Magic Numbers of VMWare. The last interesting finding reported is that the packers **Enigma Protector, Pelock, PCGuard, Enigma Virtual Machine, Themida, and AsProtect** are the only ones implementing a stalling technique. Despite using functions that are used for waiting during the execution of their samples, they demonstrated a stalling intent in repeating that function thousands of times. The final purpose was to slow the debugger or waste time finishing the timer of a sandbox.

6. Conclusion

The goal of this study was to find answers to three questions that are discussed in the next part:
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 packers. 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. 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 are able to debug a packer.

References

[1] Ansa. Ansa - massacro nel cuore di europa.

[2] Ansa. Ansa - russi rimuovete putin.

[3] Peter Ferrie. The "ultimate" anti-debugging reference. 04 2011.

[4] Francesco Mecca Martina Lindorfer Stefano Ortolani Davide Balzarotti Giovanni Vigna Hojjat Aghakhani, Fabio Gritti, Santa Barbara Universita degli Studi di Torino ‘ TU Wien Lastline Inc. Eurecom hojjat, degrigis, vigna, chris@cs.ucsb.edu francesco.mecca402@edu.unito.it mlindorfer@iseclab.org ortolani@lastline.com 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.

[5] Mi-Jung Choi Jong-Wouk Kim1, Jiwon Bang2. Defeating anti-debugging tech-

niques for malware analysis using a debugger. 11 2020.

[6] The linux programming interface. dnspy.

[7] Check Point Software Technologies LTD. Anti debugging tricks, 2020.

[8] Microsoft. Sysinternals.

[9] Mario Polino Michele Carminati-Andrea Continella Stefano Zanero Nicola, Galloro. A systematical and longitudinal study of evasive behaviors in windows malware. 12 2021.

[10] Microsoft Corporation Peter Ferrie, Senior Anti-Virus Researcher. Anti-unpacker tricks. 04 2011.

[11] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. Measuring and defeating anti-instrumentation-equipped malware. In *DIMVA*, 2017.

[12] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo 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.