Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies

Rodrigo Rubira Branco, Gabriel Negreira Barbosa, Pedro Drimel Neto
{rbranco,gbarbosa,pdrimel} *NOSPAM* qualys.com
Qualys – Vulnerability & Malware Research Labs (VMRL)

Version 1.0

Abstract

Malware is widely acknowledged as a growing threat with hundreds of thousands of new samples reported each week. Analysis of these malware samples has to deal with this significant quantity but also with the defensive capabilities built into malware; Malware authors use a range of evasion techniques to harden their creations against accurate analysis. The evasion techniques aim to disrupt attempts of disassembly, debugging or analyse in a virtualized environment.

This talk catalogs the common evasion techniques malware authors employ, applying over 50 different static detections, combined with a few dynamic ones for completeness. We validate our catalog by running these detections against a database of 4 million samples (the system is constantly running and the numbers will be updated for the presentation), enabling us to present an analysis on the real state of evasion techniques in use by malware today. The resulting data will help security companies and researchers around the world to focus their attention on making their tools and processes more efficient to rapidly avoid the malware authors' countermeasures.

This first of its kind, comprehensive catalog of countermeasures was compiled by the paper's authors by researching each of the known techniques employed by malware, and in the process new detections were proposed and developed. The underlying malware sample database has an open architecture that allows researchers not only to see the results of the analysis, but also to develop and plug-in new analysis capabilities. The system will be made available in beta at Black Hat, with the purpose of serving as a basis for innovative community research.

1. Introduction

Besides the common sentences among researchers and industry regarding the amount of new samples every day (near to the hundred thousand daily), still the analysis efforts focus on automating a specific task or automate the analysis of only one sample. Researchers around the globe have many challenges to contribute in combating new malware, since they either lacks the access to the samples or access to the computing power to process them (or both). This limits the amount of contributions coming from the academia and from individual contributors.
The situation created an industry full of incomplete results and opinions. Analysis comprising just a few thousands of malware samples are not a basement for decisions, but still they are the majority of the cases.
This works analyzed millions of malwares focusing in their protection mechanisms. We divided the protection mechanisms in 4 different categories:

- Anti-Debugging: Techniques to compromise debuggers and/or the debugging process
- Anti-Disassembly: Techniques to compromise disassemblers and/or the disassembling process
- Obfuscation: Techniques to make the signatures creation more difficult and the disassembled code harder to be analyzed by a professional
- Anti-VM: Techniques to detect and/or

compromise virtual machines

Techniques that are not being currently being detected in the malware samples are also explained: we are constantly updating the system.

The paper is organized as follows. Section 1.1 discusses our methodology, the automated analysis system and some other choices made for this research. Section 2 provides the results of our analysis, while the rest of the paper discusses the technical details of the implementations themselves. Section 3 enumerates and details each of the anti-debugging techniques. Section 4 discusses disassembly concepts and anti-disassembly and obfuscation techniques. Section 5 discusses anti-VM techniques. Section 6 illustrates new techniques and advancements proposed by this work. Section 7 comprises the downloading links for getting updated versions of this paper and for downloading the developed examples to validate each of the detection anti-reverse engineering mechanisms. Section 8 concludes and provides future directions. Section 9 has some acknowledges. Finally, in Section 10, the references used in this work.

1.1. Methodology

The analysis performed in this work relied in a total of 72 cores and a 100GB of RAM distributed in 9 different machines.
We analyzed a bit more than 4 million samples (4,030,945). Packed samples were not analyzed individually: all packed samples using the same packer have been considered as one single unique sample.
All our samples were 3.9MB or less in size (performance reasons). The only exception was the Flame malware due to its importance.
We used mostly static techniques, but included a few dynamic ones for completeness: some techniques cannot be detect using only a static approach.
The automated malware analysis system, called Dissect || PE, relies in plugins. Each application that reads a malware and produces an output is considered a plugin. There are:

- Dynamic plugins: plugins that run inside a Windows VM;
- Static plugins: plugins that run outside of the VM

It was developed a plugin that is a framework for disassembly-related analysis:

- Facilitates the development of disassembly analysis code;
- Speeds up the disassembly process for plugins;
- Calls-back the plugins for specific instruction types;
- Disassembly once, analyze all;
- Care must be taken to detect disassembly attacks themselves.

For this work, we disassembled and analyzed only PE sections explicitly marked as executable or where the entry point is located.

The anti-reverse engineering techniques were detected in the malware samples through plugins. Before its deploy, each plugin was tested against 883 PE files looking for bugs and for the quality of the detection coverage itself.

2. Executive Summary

For this research, we analyzed 4.030.945 malware samples in our lab. As depicted in Chart 1, 34,79% were packed, and the top packer families are shown in Chart 2.

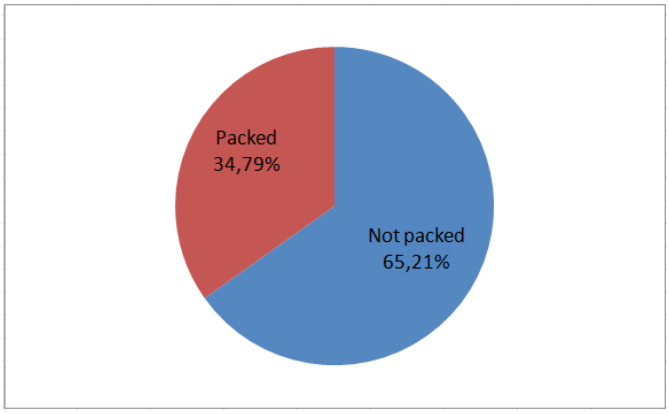


Chart 1 – Packer Statistics

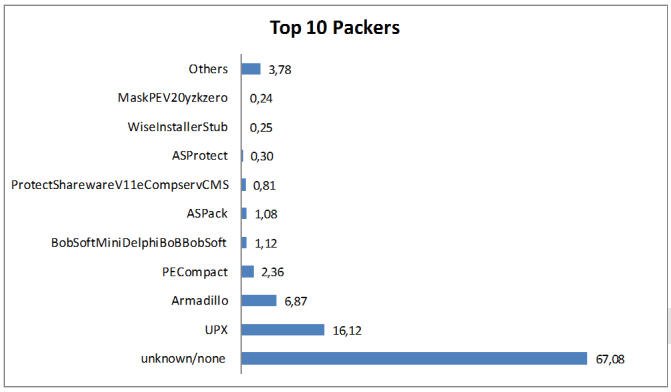


Chart 2 – Top Packer Families

Looking for anti-reverse engineering techniques in the top packer families, we had different results for the same packer family because of different versions. Being so, we detailed the techniques found in each version in Table 1.

1	**UPX**
UPXV200V290MarkusOberhumerLaszloMolnarJohnReiser	
	Anti-VM (SLDT)
	Anti-VM (IN)
	Push Pop Math
	Instruction Counting
	PEB NtGlobalFlag
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
UPXv20MarkusLaszloReiser	
	Anti-VM (SLDT)
	Anti-VM (IN)
	Push Pop Math
	Instruction Counting
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
	SS register
UPX290LZMAMarkusOberhumerLaszloMolnarJohnReiser	
	Anti-VM (IN)
	Push Pop Math
	Instruction Counting
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
	SS register
UPX20030XMarkusOberhumerLaszloMolnarJohnReiser	
	Anti-VM (IN)
	Push Pop Math
	Instruction Counting
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
UPX293300LZMAMarkusOberhumerLaszloMolnarJohnReiser	
	Anti-VM (IN)
	Instruction Counting
	PEB NtGlobalFlag
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
UPXProtectorv10x2	
	Nothing
2	**Armadillo**
Armadillov171	
	Instruction Counting
	Instruction Substitution (push – ret)
Armadillov1xxv2xx	
	Nothing
3	**PECompact**
	Anti-VM (STR)
	Anti-VM (SLDT)
	Anti-VM (IN)
	Push Pop Math
	PEB NtGlobalFlag
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
	SoftICE – Interrupt 1
	Software Breakpoint Detection
	SS register
4	**BobSoftMiniDelphiBoBBobSoft**
	Anti-VM (STR)
	Anti-VM (SLDT)
	Anti-VM (IN)
	Push Pop Math
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
	SoftICE – Interrupt 1
	SS register
5	**ASPack**
ASPackv212AlexeySolodovnikov ASProtectV2XDLLAlexeySolodo	
	Anti-VM (IN)
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
	SS register
ASPackv10803AlexeySolodovnikov	
	Anti-VM (IN)
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
ASPackv21AlexeySolodovnikov	
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
	SS register
6	**ProtectSharewareV11eCompservCMS**
	Anti-VM (SLDT)

	Anti-VM (IN)
	Instruction Counting
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
	Instruction Substitution (push – ret)
7	**ASProtect13321RegisteredAlexeySolodovnikov ASProtectv12**
	Anti-VM (STR)
	Anti-VM (SLDT)
	Anti-VM (IN)
	Push Pop Math
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
	SoftICE – Interrupt 1
	Software Breakpoint Detection
	SS register
8	**WiseInstallerStub**
	Nothing
9	**MaskPEV20yzkzero**
	Anti-VM (SLDT)
	Anti-VM (IN)
	Push Pop Math
	PEB's BeingDebugged (Stealth IsDebuggerPresent)
	SS register

Table 1 – Packers Anti-Reverse Engineering

The top packer families for malware samples targeting brazilian banks were also analyzed. As shown in Chart 3, we found that 50,49% were packed, and the top packer families are depicted in Chart 4.

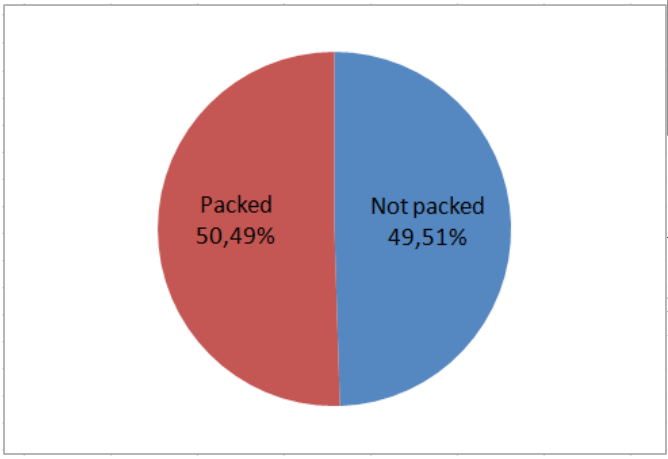


Chart 3 – Packer Statistics of Samples Targeting Brazilian Banks

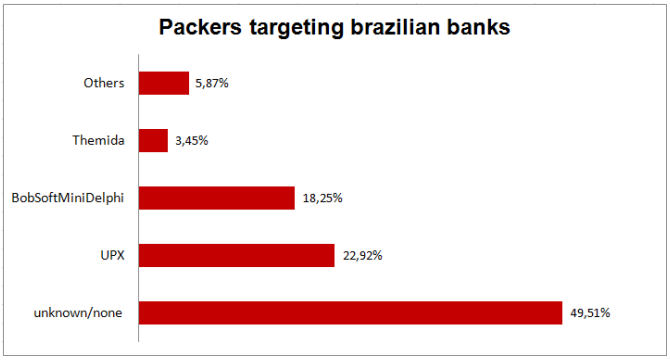


Chart 4 – Packer Families of Malware Samples Targeting Brazilian Banks

From this point on, and according to the proposed methodology in which each packer was analyzed once, the following numbers are related to the not packed samples. Additionally, in the next statistics, malware analysis algorithms that produce evidences were not considered.

Chart 5 shows that 88,96% of the samples had at least one anti-reverse engineering technique detected.

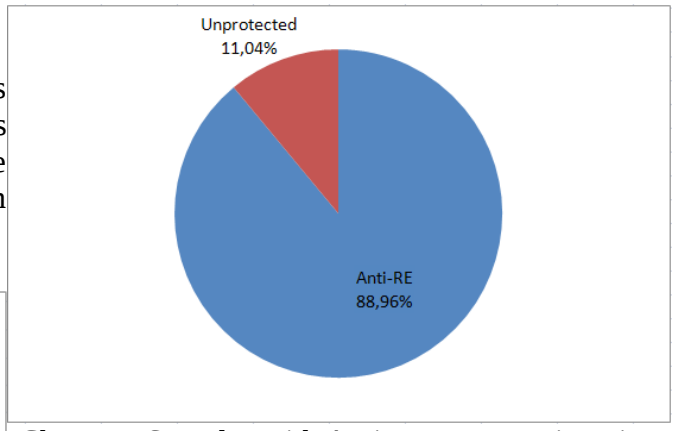


Chart 5 – Samples with Anti-Reverse Engineering

As shown in Chart 6, 6,42% of the analyzed samples have implemented at least one protection mechanism in each of the four categories (named as fully armored samples in this work).

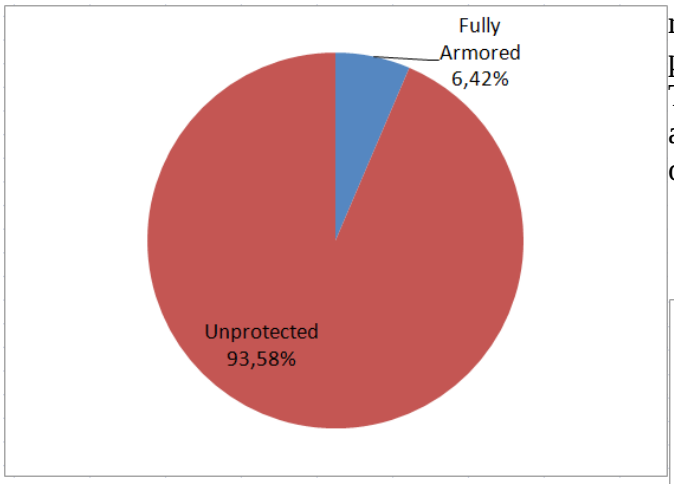


Chart 6 – Fully Armored Samples

For a sample to be considered as part of a category, at least one technique of such a category have to be detected. The prevalence of each considered anti-reversing engineering categories in the analyzed samples are detailed in Chart 7.

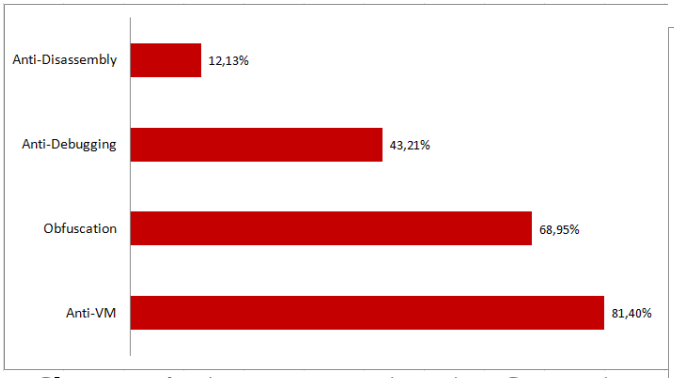
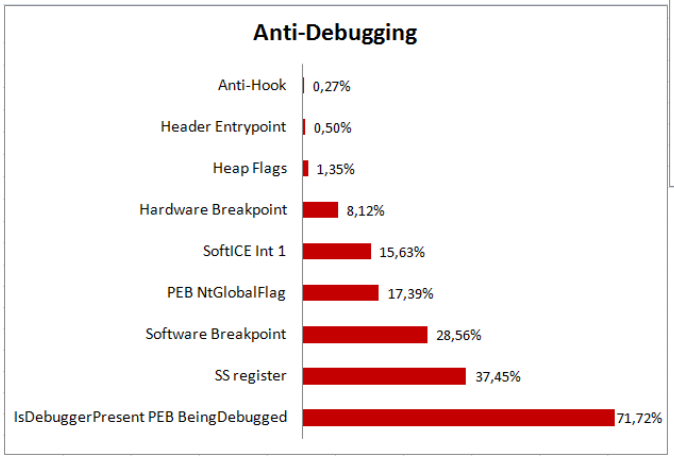


Chart 7 – Anti-Reverse Engineering Categories

So, anti-VM and obfuscation categories are considerably more prevalent in the samples with, respectively, 81,40% and 68,95%.

The considered anti-debugging techniques in all of the statistics in this work relied on the techniques depicted in Chart 8. Additionally, the percentage of each considered anti-debugging technique regarding the total samples in this category is also present in Chart 8.



The same information are present in charts 9, 10 and 11, but for, respectively, anti-disassembly, obfuscation and anti-VM categories.

Chart 8 – Anti-Debugging Techniques

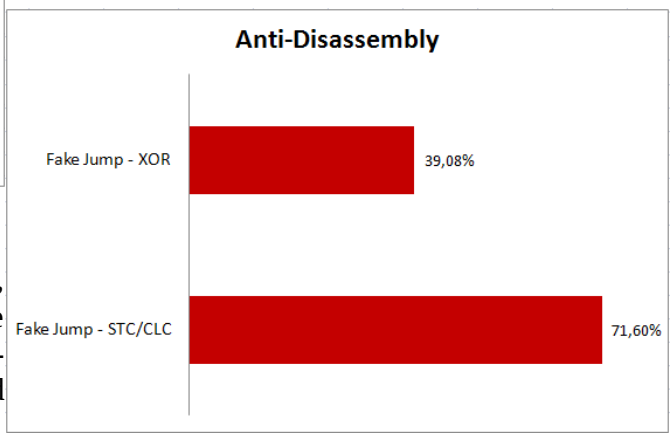


Chart 9 – Anti-Disassembly Techniques

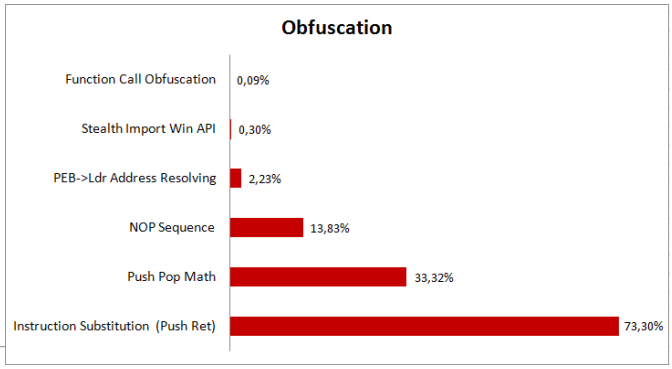


Chart 10 – Obfuscation Techniques

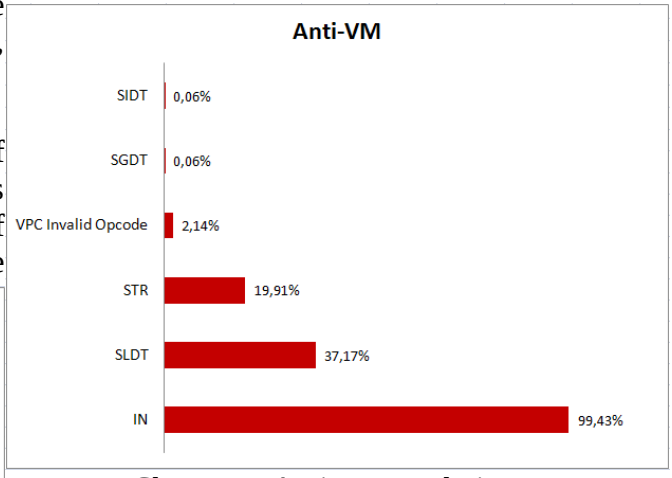


Chart 11 – Anti-VM Techniques

3. Anti-Debugging Techniques

Some anti-debugging techniques are described in the next sections.
Techniques currently covered by detection plugins will have an additional information: the algorithm used to detect such a technique.

3.1. PEB NtGlobalFlag

NtGlobalFlag is a field of PEB at offset 0x68 [1]. The presence of such values is not a reliable debugger detection technique, but can be considered as an evidence: FLG_HEAP_ENABLE_TAIL_CHECK (0x10), FLG_HEAP_ENABLE_FREE_CHECK (0x20) and FLG_HEAP_VALIDATE_PARAMETERS (0x40). This might be used to detect the presence of a debugger.

Adopted Static Detection:

A MOV instruction (mov, movsx, movzx) copying PEB address (fs:[0x30]) somewhere (X) is looked for and X is saved for future use:

mov/movsx/movzx X,fs:[0x30]

Then, later in the same function, a CMP (cmp, cmpxchg) or a MOV (mov, movsx, movzx) instruction referencing the NtGlobalFlag ([X+0x68] in some of the operands) is looked for:

cmp/cmpxchg/mov/movsx/movzx op1,op2 → where [X+0x68] is a substring of op1 or op2

RET was used to consider the end of a function.

If this scenario happens, this anti-debugging technique is considered as detected.

3.2. IsDebuggerPresent

IsDebuggerPresent() is a kernel32 function that returns TRUE if a debugger is present [1]. Internally, it uses PEB's BeingDebugged Field. Such approaches can be used to detect the presence of a debugger.

Adopted Static Detection:

(1)

IsDebuggerPresent is looked for in IAT. If found, the technique is considered as detected.

(2)

A MOV instruction (mov, movsx, movzx) copying PEB address (fs:[0x30]) somewhere (X) is looked for and X is saved for future use:

mov/movsx/movzx X,fs:[0x30]

Then, later in the same function, another MOV instruction (mov, movsx, movzx) referencing the BeingDebugged field ([X+0x2] in some of the operands) is looked for:

mov/movsx/movzx op1,op2 → where “[X+0x2]” is a substring of op1 or op2

RET was used to consider the end of a function.

If this scenario happens, this anti-debugging technique is considered as detected.

3.3. CheckRemoteDebuggerPresent

CheckRemoteDebuggerPresent() is a kernel32 function that sets 0xffffffff in pbDebuggerPresent parameter if a debugger is present [1]. Internally, it uses NtQueryInformationProcess() with ProcessDebugPort as a ProcessInformationClass parameter. This function can be used to detect the presence of a debugger.

Adopted Static Detection:

(1)

CheckRemoteDebuggerPresent is looked for in IAT. If found, the technique is considered as detected.

(2)

NtQueryInformationProcess is looked for in IAT. If

found, the technique is considered as an evidence detected.

3.4. Heap flags

Process default heap (retrieved through GetProcessHeap() or PEB) has the following two fields of interest that are influenced by PEB->NtGlobalFlags: Flags, at offset 0x0c in the heap, and ForceFlags at offset 0x10 in the heap [1]. The following values for each of the fields are not a reliable approach to detect a debugger, but can be considered as an evidence:

- Flags: HEAP_GROWABLE (2), HEAP_TAIL_CHECKING_ENABLED (0x20), HEAP_FREE_CHECKING_ENABLED (0x40), HEAP_SKIP_VALIDATION_CHECKS (0x10000000) and HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000).
- ForceFlags: HEAP_TAIL_CHECKING_ENABLED (0x20), HEAP_FREE_CHECKING_ENABLED (0x40) and HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000).

Adopted Static Detection:

(1)

GetProcessHeap is looked for in IAT. If found, the technique is considered as an evidence detected.

(2)

An instruction referencing PEB (fs:[0x30]) is looked for. If found, the first operand (X) is saved for future use:

? X,? → The substring “fs:[0x30]” is looked for in all the operands. If found, the first operand (X) is saved

Then, later in the same function, any other instruction referencing the process default heap ([X+0x18] in some of the operands) is looked for:

? operands → where “[X+0x18]” is a substring of any of the operands

RET was used to consider the end of a function.

If this scenario happens, this anti-debugging technique is considered as detected.

3.5. NtQueryInformationProcess – ProcessDebugPort

Calling NtQueryInformationProcess() with ProcessDebugPort as a ProcessInformationClass parameter will set 0xffffffff in the ProcessInformation parameter if a process is being debugged [1]. Internally, such function queries for a non-zero state of EPROCESS->DebugPort. This function can be used to detect a debugger.

Adopted Static Detection:

NtQueryInformationProcess is looked for in IAT. If found, the technique is considered as evidence detected.

3.6. Debug Objects – ProcessDebugObjectHandle Class

A debug object is created and a handle is associated to it when a debugging session begins [1]. NtQueryInformationProcess() can be called with ProcessDebugObjectHandle as a ProcessInformationClass parameter to query for the debug object handle. This can be used to detect the presence of a debugger.

Adopted Static Detection:

NtQueryInformationProcess is looked for in IAT. If found, the technique is considered as an evidence detected.

3.7. Debug Objects – ProcessDebugFlags Class

NtQueryInformationProcess() can be called with ProcessDebugFlags as a ProcessInformationClass parameter to set the inverse of EPROCESS-

>NoDebugInherit bit in ProcessInformation parameter [1]. So, FALSE is set when a debugger is present. This can be used to detect a debugger.

Adopted Static Detection:

NtQueryInformationProcess is looked for in IAT. If found, the technique is considered as evidence detected.

3.8. NtQuerySystemInformation – SystemKernelDebuggerInformation

NtQuerySystemInformation() function of ntdll can be used with the undocumented SystemKernelDebuggerInformation as a SystemInformationClass parameter to detect the presence of a debugger [1]. The result, that is stored in the buffer pointed by SystemInformation parameter [23], has 2 bytes representing two flags, each one with 8 bits: KdDebuggerEnabled (least significant bye) and KdDebuggerNotPresent (most significant byte). KdDebuggerNotPresent is FALSE if a debugger is present.
It is possible to obfuscate such a function call by retrieving KdDebuggerNotPresent directly from KUSER_SHARED_DATA, at offset 0x7ffe02d4 for 2Gb user-space configurations. The value retrieved by the NtQuerySustemInformation() call does not come from this location. [2]
This can be used to the detect the presence of a kernel-mode debugger [22].

Adopted Static Detection:

NtQuerySystemInformation is looked for in IAT. If found, the technique is considered as evidence detected.

3.9. OpenProcess – SeDebugPrivilege

With SeDebugPrivilege privilege, a non-default privilege [5], a process can gain full control over the system process CSRSS.exe [1]. Additionally, such a privilege is passed to child processes. So, if a debugger acquires such a privielge, the debugged binary can have full control over CSRSS.exe also. [5] This technique has 2 steps:

1. Enumerate processes to get the process ID of CSRSS.exe. This can be achieve through CreateToolhelp32Snapshot()+ (Process32First())+Process32Next(). Another way could be using NtQuerySystemInformation() with SystemProcessInformation as a SystemInformationClass parameter. Alternatively, Windows XP introduced the ntdll CsrGetProcessId() which makes such a task easier and can also be used.
2. Open CSRSS.exe process with full access. If the operation succeeds, than it is an evidence of the presence of a debugger. This task can be achieved with OpenProcess() using PROCESS_ALL_ACCESS as a dwDesiredAccess parameter.

OllyDbg and WinDbg acquires SeDebugPrivilege privilege.
This technique might be used to indirectly detect the presence of some debuggers.

Adopted Static Detection:

The string “csrss.exe” is looked for in the binary in a case-insensitive way. If found, this technique is considered as evidence detected.

3.10. Alternative Desktop

Windows NT-based platforms supports multiple desktops, and it is possible to select a different active desktop, hiding the windows of the previously selected one with no obvious way to switch back to the old desktop [1]. This can be done calling CreateDesktop() followed by SwitchDesktop(). The dwDesiresAccess parameter of CreateDesktop() can be: DESKTOP_CREATEWINDOW | DESKTOP_WRITEOBJECTS | DESKTOP_SWITCHDESKTOP. This technique can be used to make the debugging process harder for an analyst.

Adopted Static Detection:

CreateDesktopA/CreateDesktopW are looked for in IAT. If found, and SwitchDesktop is also present,

the technique is considered as detected.

3.11. Self-Debugging

"Self-debugging is the act of running a copy of a process, and attach to it as a debugger." [1]. Since only one debugger can be attached to a process, such process could not be debugger by ordinary means (there are bypasses). It is possible to execute this technique creating a copy of the process to be debugged (CreateProcessA() with DEBUG_PROCESS as a dwCreationFlags parameter), and handling its debug events (WaitForDebugEvent() and ContinueDebugEvent()). This technique can be used to difficult a debugger to be attached to the process.

Adopted Static Detection:

CreateProcessA/CreateProcessW are looked for in IAT. If found, and both WaitForDebugEvent and ContinueDebugEvent are also present, the technique is considered as evidence detected.

3.12. RtlQueryProcessDebugInformation

RtlQueryProcessDebugInformation() is used to load some process information in DebugBuffer parameter including some heap information (heap flags is among them) [2][3][4]. This call can be made with PDI_HEAPS | PDI_HEAP_BLOCKS as a DebugInfoClassMask parameter. Internally, it uses RtlQueryProcessHeapInformation(), and this function can be used to develop a variation of this technique. The following heap flags value indicates that a process is being debugged: GROWABLE | TAIL_CHECKING_ENABLED | FREE_CHECKING_ENABLED | VALIDATE_PARAMETERS_ENABLED. This technique can be used to detect the presence of a debugger.

Adopted Static Detection:

RtlQueryProcessDebugInformation and RtlQueryProcessHeapInformation are looked for in IAT. If some ot them are found, technique is considered as evidence detected.

3.13. Hardware Breakpoints

When an exception occurs, Windows passes to the exception handler a context structure which have, among other information, the debug registers content [1]. If there is a debugger with hardware breakpoints being used and it passes the exception to the debuggee, then the debug registers can be analyzed looking for a debugger. This can be used to detect the presence of a debugger.

Adopted Static Detection:

A MOV instruction (mov, movsx, movzx) copying the ESP register to the SEH (fs:[0x0]) is looked for:

mov/movsx/movzx op1,esp → Where "fs:[0x0]" is a substring of op1

Then, later in the same function, another MOV instruction (mov, movsx, movzx) referencing the CONTEXT ([esp+0xc]) is looked for in the source operand and the destination one (X) is saved for future use:

mov/movsx/movzx X,op2 → where "[esp+0xc]" is a substring of op2

Then, later in the same function, instructions CMP (cmp, cmpxchg), MOV (mov, movsx, movzx) or OR, both with, in the source operand, having an offset of a debug register relative to the saved X (0x4, 0x8, 0xC 0x10) is looked for:
mov/mosx/mozx op1,op2
cmp/cmpxchg op1,op2
or op1,op2
→ All op2 having "[X+0x4]", "[X+0x8]", "[X+0xC]" or "[X+0x10]" substrings.

RET was used to consider the end of a function.

If this scenario happens, this anti-debugging technique is considered as detected.

3.14. OutputDebugString

The kernel32 OutputDebugString() has different behavior depending on the presence, or not, of debugger [1]. One of them is that kernel32

GetLastError() returns 0 if the debugger is present. This technique can be used to detect the presence of a debugger.

Adopted Static Detection:

OutputDebugStringA/OutputDebugStringW are looked for in IAT. If found, and GetLastError is also present, the technique is considered as detected.

3.15. BlockInput

BlockInput() function “Blocks keyboard and mouse input events from reaching applications.” [6]. This function can be used to difficult the access of an analyst to the debugger [1][5].

Adopted Static Detection:

BlockInput is looked for in IAT. If found, the technique is considered as evidence detected.

3.16. Parent Process

The parent process of an application executed by an user will usually be “Explorer.exe”, and it can be considered as a debugger evidence when such a characteristic does not happen [1]. The following functions can be used for this purpose:

- GetCurrentProcessId() + CreateToolhelp32Snapshot()+ (Process32First())+Process32Next().
- GetCurrentProcessId() + NtQuerySystemInformation() with SystemProcessInformation as a SystemInformationClass parameter.
- A simpler method: get Explorer.exe process ID (GetShellWindow() +GetWindowThreadProcessId()) and get the parent process ID (NtQueryInformationProcess() with ProcessBasicInformation as a ProcessInformationClass parameter).

This technique might be used to detect the presence of a debugger.

Adopted Static Detection:

(1)

GetCurrentProcessId and CreateToolhelp32Snapshot are looked for in IAT. If both are found, this technique is considered as evidence detected.

(2)

GetCurrentProcessID and NtQuerySystemInformation are looked for in IAT. If both are found, the technique is considered as evidence detected.

(3)

GetShellWindow, GetWindowThreadProcessId and NtQueryInformationProcess are looked for in IAT. If both are found, this technique is considered as evidence detected.

3.17. Device Names

Debuggers that uses kernel-mode drivers may use named devices to communicate with them [1]. So, if an open attempt in such devices succeeds, it does not necessarily means that a debugger is active, but that it is present. The implementation can use CreateFile() function with OPEN_EXISTING as a dwCreationDisposition parameter. Some device names:

- SoftICE: \\.\SICE, \\.\SIWVID, \\.\NTICE
- RegMon: \\.\FILEVXG, \\.\REGSYS
- FileMon: \\.\FILEVXG, \\.\FILEM
- \\.\TRW
- SoftICE extender: \\.\ICEEXT

This technique might be used to detect the presence of a debugger. The presence of a debugger does not necessarily means that the debugger is active.

Adopted Static Detection:

Device name strings (“\\.\SICE”, “\\.\SIWVID”, “\\.\NTICE”, “\\.\FILEVXG”, “\\.\REGSYS”, “\\.\FILEM”, “\\.\TRW”, “\\.\ICEEXT”) are looked

for in the binary itself in a case insensitive way. If found, this technique is considered as detected.

3.18. OllyDbg – OutputDebugString

OllyDbg is a debugger that have a format string vulnerability with the kernel32 OutputDebugString() function, leading to a crash or an arbitrary code execution [1][5]. The current final version (1.10) is still vulnerable. This can be used to break a debugging process with an affected version of OllyDbg.

Adopted Static Detection:

OutputDebugStringA/OutputDebugStringW are looked for in IAT. If found, this technique is considered as evidence detected.

3.19. FindWindow

FindWindow() function can be used to find opened debuggers using both parameters, lpClassName and lpWindowName [1]. Some parameters that can be used:

- lpClassName: OllyDbg: “OLLYDBG”; WinDbg: “WinDbgFrameClass”; MSLRH: “TESTDBG”, “kk1”, “Eew57”, “Shadow”.
- lpWindowName: MSLRH: “Import REConstructor v1.6 FINAL (C) 2001-2003 MackT/uCF”.

This can be used to detect the presence of a debugger.

Adopted Static Detection:

FindWindowA/FindWindowW are looked for in IAT. If found, this technique is considered as evidence detected.

3.20. SuspendThread

User-mode debuggers like OllyDbg and Turbo Debug can be disabled by calling kernel32 SuspendThread() (or the ntdll NtSuspendThread()) in its threads [1][2]. To find the threads, process enumeration and named window searching are two methods that can be used.

Adopted Static Detection:

SuspendThread and NtSuspendThread are looked for in IAT. If some of them found, this technique is considered as evidence detected.

3.21. SoftICE – Interrupt 1

Normally, the DPL of interrupt 1 is set to 0, meaning that a ring 3 attempt to execute int 1 (“0xcd01”) results in a CPU general protection fault (int “0x0d”) and in the end Windows raises an EXCEPTION_ACCESS_VIOLATION (0xc0000005) [1].
SoftICE hooks IDT entry of interrupt 1 and sets the DPL to 3, allowing it to single-step from user-mode code. The problem is that SoftICE does not identify and handle differently the situations that caused such an int 1, and always execute the default interrupt 1 handler.
So, a ring 3 attempt to execute int 1 results in the Windows raising EXCEPTION_SINGLE_STEP instead of EXCEPTION_ACCESS_VIOLATION (0x80000004). This characteristic can be used to detect if the SoftICE is running.

Adopted Static Detection:

INT1 instruction is looked for. Then, later in the same function, a CMP instruction with 0x80000004 in any of the operands is looked for:

…
int1
…
cmp operands → where any of the operands are 0x80000004
...

RET was used to consider the end of a function.

If this scenario happens, this anti-debugging technique is considered as detected.

3.22. SS register

While single-stepping through trap flag, debuggers typically try to clean such a flag when it is pushed in the stack [1][2][7].
When SS register is loaded (POP SS, for example), the interrupts are disabled until the end of the next instruction to avoid invalid stack troubles in some cases [8].
So, after the SS loading, the next instruction will be executed but the debugger will not break on it.
With the debugger unaware of the flags pushing (PUSHFD, for example), the trap flags will not be cleaned in the stack and its presence indicates a single-stepping thought trap flags debugging.

Adopted Static Detection:

A POP instruction with SS register as operand, or a MOV instruction (mov, movsx, movzx) having SS register as a destination operand are looked for:

pop ss

mov/movsx/movzx ss,? → It does not matter what is the second operand

Then, the next instruction is analyzed to check if the next mnemonic starts with the string “pushf”.

If this scenario happens, this anti-debugging technique is considered as detected.

3.23. UnhandledExceptionFilter

When an exception is generated and there was no exception handlers to processes it, a default handler exists to do such a job [1][5][7]. As part of the default handler procedures, kernel32 UnhandledExceptionFilter() is called. In such a function, NtQueryInformationProcess() is called with ProcessDebugPort as a ProcessInformationClass parameter to detect if the process that raised the exception is being debugged. If SetUnhandledExceptionFilter() was used and the process is not being debugged, the top-level exception filter set by such a function will be executed. Otherwise, if the process is being debugged, the debugger will be notified about the exception. [9]
This behavior can be used to detect the presence of a debugger by defining a top-level exception filter through SetUnhandledExceptionFilter() and then forcing an exception to occur. If the top-level exception filter gets executed, then the process is not being debugged.

Adopted Static Detection:

SetUnhandledExceptionFilter is looked for in IAT. If found, this technique is considered as evidence detected.

3.24. Guard Pages

An attempt to access an address within a guard page (page marked with PAGE_GUARD) results in a STATUS_GUARD_PAGE_VIOLATION (0x80000001) being raised by the system [1][10]. If a debugger is present, it might handle such an exception and allow the access. This behavior might be used to detect the presence of a debugger. An implementation of such a technique, as shown in [1], relies on writing 0xC3 (RET instruction) in a memory area and marking this page with PAGE_GUARD. If the RET gets executed, the debugger is detected; otherwise, a crafted exception handler is executed meaning that the debugger was not detected.

Adopted Static Detection:

VirtualAlloc/VirtualAllocEx and VirtualProtect/VirtualProtectEx are looked for in IAT. If found, this technique is considered as evidence detected.

3.25. Execution Timing

When a debugger is present, the time elapsed between subsequent instructions execution might be higher than without it [1][2][7]. The idea is to measure time elapsed between some instructions execution and based on such a value, infer the presence of a debugger. Some methods can be used to implement this technique (each method has its own characteristics):

- RDTSC instruction (it is a popular anti-

debugging technique [1] [2] [7] [11] but there are some issues to be aware of [8] [11] [12] [13])
- RDPMC instruction [2] [8]
- RDMSR instruction [2] [8]
- kernel32 GetTickCount() [14]
- winmm timeGetTime [1] [15]
- kernel32 GetLocalTime() [2] [16]
- kernel32 GetSystemTime() [2] [17]
- kernel32 QueryPerformanceCounter() [2] [7] [18].

Adopted Static Detection:

GetTickCount, timeGetTime, GetLocalTime, GetSystemTime and QueryPerformanceCounter are looked for in IAT. If some of them are found, this technique is considered as evidence detected.

3.26. Software Breakpoint Detection

Software breakpoint is a single-byte instruction (0xCC – INT 3) that stops the execution of the debugged process and passes control to the debugger [5]. The original byte is saved by the debugger before setting the breakpoint, this way the original instruction can be executed in the correct time. [19]
Code areas in memory are scanned for 0xCC byte that was not set by the code itself. To make such a check not so obvious, it is possible to use some operation in the compared by, such as [5]:

if(byte XOR 0x55 == 0x99) then breakpoint found

Note that 0xCC XOR 0x55 = 0x99.

Adopted Static Detection:

A CMP instruction (cmp, cmpxchg) with 0xCC in any of its operands is looked for. If found, this anti-debugging technique is considered as detected.

3.27. Thread Hiding

According to MSDN [20] [21], ntdll NtSetInformationThread() sets the priority of a thread [1][5][7]. However, its ThreadInformationClass parameter has an undocumented value, ThreadHideFromDebugger (0x11), which prevents debugging events to be sent to the debugger [5]. This can be used to difficult the debugging.

Adopted Static Detection:

NtSetInformationThread is looked for in IAT. If ound, this technique is considered as evidence detected.

3.28. NtSetDebugFilterState

The ntdll DbgSetDebugFilterState (or ntdll NtSetDebugFilterState) call succeeds in the presence of some debuggers [2]. This is a side-effect of the debugger's behaviour: the process SeDebugPrivilege privilege. SeDebugPrivilege is not a default privilege [5], so this technique might be used to indirectly detect the presence of some debuggers.

Adopted Static Detection:

DbgSetDebugFilterState and NtSetDebugFilterState are looked for in IAT. If some of them are found, this technique is considered as detected.

3.29. Instruction Counting

An exception handler is registered to deal with the EXCEPTION_SINGLE_STEP (0x80000004) exception [1].
Then, some hardware breakpoints are set in specific instructions. Debug registers cannot be accessed directly in user-mode [32], so a context structure is needed and the following procedures can be used to get it:
- Calling kernel32 GetThreadContext().
- Forcing an exception to occur and handling it, because the context structure is passed to the exception handler. This is more stealth than the previous procedure.

As instructions with hardware breakpoints are being reached, the previously registered exception handler is supposed to deal with the raised

exceptions. Such handler will simply count how many times it was reached and then can change the EIP to point to a new instruction and resume the execution.
Some debuggers do not deal correctly with hardware breakpoints that were set by them, and some of the raised EXCEPTION_SINGLE_STEP might not be handled by the previously set exception handler.
After all hardware breakpoints got reached and its exception handlers finished, the total counter used by them should have the number of hardware breakpoints initially set. If the value was different, it indicates the presence of a debugger.

3.30. Header Entrypoint

File sections that do not include the attribute IMAGE_SCN_MEM_WRITE (write) is read-only by default to a remote debugger [1].
Additionally, there is no section that describes the PE header, it will be also considered as read-only; there is an exception when the PE->SectionAlignment is less than 4kb, which causes it to be marked internally as both writable and executable [1].
Being so, if the debugger does detect such situation and does not set a write privilege in such a section, the debugger might allow the application to run freely.

Adopted Static Detection:

The entrypoint section is analyzed to check if it has the IMAGE_SCN_MEM_WRITE attribute. If it does not have, then this technique is considered as detected.

3.31. Self-Execution

This technique relies on a process to create another process of itself [1]. This way, the second process will not be debugged. Usually this trick is used with a mutex to prevent many copies of the process to be in execution.

Adopted Static Detection:

CreateProcessA/CreateProcessW, CreateMutex and WaitForSingleObject functions are looked for in IAT. If some of them are found, this technique is considered as evidence detected.

3.32. Hook Detection

Some hook techniques relies on overwriting the first instruction of the hooked function by a JMP instruction pointing to another place. [48]

Regarding Microsoft Detours, some characteristics exist that can be used as a signature, such as .detours section and the presence of detoured.dll. [49] [50]

Detecting the presence of a hook might detect some binary analysis procedures.

Adopted Static Detection:

(1)

A CMP instruction with 0xE9 in some of its operands is looked for. If found this technique is considered as evidence detected.

(2)

The string ".detour" is looked for in the binary with the exception of its sections. The string "detoured.dll" is also looked for in the binary, but with the exception of the imports. If some of them were found, the technique is considered as detected.

Section 3.33. DbgBreakPoint Overwrite

When a debugger attaches to a process, an exception is raised by DbgBreakPoint() function in NTDLL (called at attach time) [2]. Handling such an exception the debugger gains control of the debugee.

By marking the page(s) of DbgBreakPoint() as EXECUTE_READWRITE and overwriting it with, for example, a RET instruction, when a debugger attaches to the process the thread will exit immediately, thus, not breaking in.

4. Obfuscation and Anti-Disassembly Techniques

Both obfuscation and anti-disassembly techniques relies on a disassembly. Being so, they were put together in the same section.
Obfuscation is a kind of technique to make the disassembly result harder to be analyzed by a professional.
Anti-disassembly is a kind of technique to compromise disassemblers and/or the disassembling process.
Section 4.1 discusses some disassembly concepts. Section 4.2 and Section 4.3 describes, respectively, some obfuscation and anti-disassembly techniques.

4.1. Disassembly Concepts

It is possible to disassemble a binary with a static and a dynamic approach [39]. The former relies on executing the program and tracking instruction as they are being executed. The latter relies on analyzing the program bytes and finding instructions without executing it.
Static disassembling can be categorized in two main classes: linear sweep and recursive traversal.
Linear sweep approach starts from a given byte (for example, the first byte of the entry point) and from this point on analyzes byte after byte until a predefined end (for example, the end of the PE section). The main drawback of this approach is that data placed in the middle of code instructions may generate some noise, because they will be interpreted as code. An example of disassembler that uses linear sweep approach is objdump [26].
Recursive traversal is an approach that follows the program control flow instead of simply disassembling each byte. It is not vulnerable to the simple fact of data existing in the middle of code instructions, but it has another main drawback: it is not always possible to statically predict the exact program control flow. It may result in some parts not being disassembled and also the generation of some noise. The unreachable areas can be submitted to a linear sweep processing, and such a variation is called speculative disassembly. An example of disassembler that uses recursive traversal approach is IDA [40].

Anti-disassembly techniques are discussed in section 4.2 and obfuscation techniques in section 4.3.

4.2. Anti-Disassembly Techniques

Some anti-disassembly techniques are described in the next sections.
Techniques currently covered by detection plugins will have an additional information: the algorithm used to detect such a technique.

4.2.1. Garbage Bytes

This technique relies on adding additional bytes that will never be executed in run-time. [5] [38] This may break both linear sweep and recursive traversal approaches.

A liner sweep approach could interpret such bytes as being code-related bytes, breaking the alignment. As a result, such garbage bytes could be joined with valid bytes from next instructions generating wrong instructions instead of the correct ones. For example:

```
        jmp     .destination
        db      0x6a ; garbage byte technique
.destination:
        pop     eax
```

Such example generates the following disassembly by a objdump:

```
eb 01           jmp    0x401003
6a 58           push   0x58
```

Recursive traversal algorithms might also be compromised through garbage bytes if a situation in which the same set of bytes with more than one interpretation could be forced. In this case, the lack of alignment due to the interpretation of the garbage bytes as a valid code bytes might lead the disassembler to produce a wrong disassembly. For example, a Fake Conditional Jump implementation could be used for that:

```
        mov     eax,eax
        jz      .destination
```

```
        db      0x6a ; garbage byte technique
.destination:
        ; rest of the code
        pop     eax
```

Such example produces the following IDA output:

```
# IDA output
```

Adopted Static Detection:

(1)

A PUSH instruction immediately followed by a RET is looked for. If found, the technique is considered as evidence detected.

(2)

A XOR instruction with two equal operands is looked for. If found and is immediately followed by a JNZ instruction, the technique is considered as evidence detected. The same happens for STC instruction immediately followed by JNC or JAE and for CLC instruction immediately followed by JC or JB.

4.2.2. Program Control Flow Change

This technique relies on unconditionally forcing a program control flow change to occur, leaving an area with other anti-disassemble technique(s) unreachable in run-time. Disassemblers using linear sweep approach will disassemble such an area and the resulting assembly code may be compromised.

An unconditional JMP is an example that can be used to implement this technique. [38] For example, the following JMP instruction jumps an unreachable area populated with Garbage Byte anti-disassembly technique, avoiding its execution. But objdump will disassemble such an area and the resulting output is compromised:

```
        jmp     .destination
        db      0x6a ; garbage byte technique
.destination:
        ; rest of the code
        pop     eax
        …
```

Resulting objdump output:

```
eb 01           jmp   0x401003
6a 58           push  0x58
```

Another example of implementation is the Instruction Substitution that uses a Push followed by RET to replace a conventional JMP.

It is also possible to use this technique to compromise recursive traversal algorithms by using indirection. An indirect jump, for example, is an approach that can be used for such a purpose. [39] [41] The previous example was modified to use an indirect jump:

```
        push    DWORD .destination
        jmp     DWORD [esp]
        db      0x6a ; garbage byte technique
.destination:
        pop     eax
```

```
##### IDA output ########
```

Adopted Static Detection:

A PUSH instruction immediately followed by a RET is looked for. If found, the technique is considered as evidence detected.

4.2.3. Fake Conditional Jumps

This technique, based on [5] and [38], relies on creating conditional jumps which conditions are always the same. For example:

```
(1)
...
xor     eax,eax
jz      .destination1 ; always true
…

(2)
...
xor     eax,eax
jnz     .destination2 ; always false
…
```

In the first example, the JZ instructions will be always true independently of the EAX content, the instructions before XOR and the instructions after JZ. The same happens for the second example, but the JNZ instruction will be always false.
Recursive traversal approach may disassemble areas that will never be executed, and such unreachable areas can be populated with other anti-disassembly techniques, such as Garbage Bytes, that creates two different interpretations for the same set of bytes.
Each disassembler has its own way to handle such a conflict, but most of them, trust its first interpretation [38]; IDA seems to be an example of this, because it first disassembles the false branch [38].

The following approaches are examples that can be used to implement this technique:

- xor x,x (XOR with two equal operands)
  - True branch: JZ
  - False branch: JNZ
- STC instruction
  - True branch: JC or JB
  - False branch: JNC or JAE
- CLC instruction
  - True branch: JNC or JAE
  - False branch: JC or JB

Adopted Static Detection:

A XOR instruction with two equal operands is looked for. If it is immediately followed by JNZ instruction, the technique is considered as detected. The same happens for STC instruction immediately followed by JNC or JAE and for CLC instruction immediately followed by JC or JB.

4.2.4. Call Trick

This technique relies on changing the default function's return address.[39] [41] In conjunction with other techniques such as Garbage Bytes, this trick may break all kind of disassemblers.
Recursive traversal disassemblers may disassemble the next instruction after the CALL, but the correct next instruction was actually changed by the called function. After the CALL and before the next executed instruction, other anti-disassembly techniques, such as Garbage Bytes, can be used.
Linear sweep is also affected because they do not interpret instructions and may also disassemble the next instruction after the call, getting vulnerable to other anti-disassembly techniques such as Garbage Bytes.
The following example, which also employs Garbage Bytes technique, may break for both, recursive traversal and linear sweep approaches:

```
        call    .function
        db      0x6a ; garbage byte
.correct_return:
        ; rest of the code
        pop     eax
        …

.function:
        push    DWORD .correct_return
        ret
```

The following output is produced by objdump:

```
401000:    e8 02 00 00 00    call   0x401007
401005:    6a 58             push   0x58
401007:    68 06 10 40 00    push   0x401006
40100c:    c3                ret
```

The following output is produced by IDA:

```
# IDA output
```

4.2.5. Flow Redirection to the Middle of an Instruction

This technique relies on redirecting the program flow to the middle of an instruction. [38] This might compromise both linear sweep and recursive traversal algorithms.

An implementation example could be hiding an instruction in the middle of another. So, the disassembler would show an instruction that is not executed in run-time instead of the correct instruction that resides in the middle of its bytes.
Linear sweep approaches could be bypassed because the instruction aligned to the rest of the

bytes are the wrong one. Recursive traversal algorithms could be affected by making the same set of bytes to have more than one interpretation; this can be achieved, for example, by using the Fake Conditional Jump technique.
The following example illustrates such a scenario with a code that affects both, linear sweep and recursive traversal approaches:

```
; Fake Conditional Jump
xor     eax,eax
jz      +4 ; jump to the ret

; 0xc3 = ret
mov     eax,0xc3abcdef
```

In such an example, the RET instruction does not directly appear in the disassembly outputs, but is executed in run-time, as shown in the objdump and IDA outputs below.

Output of objdump:

```
31 c0                   xor    eax,eax
74 04                   je     0x401008
b8 ef cd ab c3          mov    eax,0xc3abcdef
```

Output of IDA:

```
# IDA output
```

Another implementation example could be using this anti-disassembly technique to break the alignment and generate a set of wrong instruction instead of simply hiding one in the middle of another. The following example, that is based on [38], does this:

```
mov     ax,0x05eb
xor     eax,eax

; jump to “jmp 5” (0xeb 0xe5)
; last bytes of mov instruction is 0xeb 0xe5
; such “jmp 5” redirects the flow to the rest
; of the code
jz      -6 ;

db      0xe8 ; garbage byte
; rest of the code
xor     eax,eax
pop     eax
mov     eax,esp
push    ecx
```

Output of objdump:

```
66 b8 eb 05             mov    ax,0x5eb
31 c0                   xor    eax,eax
74 f9                   je     0x401001
e8 31 c0 58 89          call   0x8998d03e
e0 51                   loopne 0x401060
```

Output of IDA:

```
# IDA output
```

This technique could also be used to make recursive traversal algorithms to generate two different interpretations for the same set of bytes without using conditional jumps: jumping into itself [38]. Additionally, because it breaks the alignment, linear sweep algorithms may also be affected. The following example, based on [38], illustrates such a scenario:

```
; All bytes of the example:
; 0xeb 0xff 0xc0 0x48

; jmp -1 = 0xeb 0xff
; jumps to itself: 0xff
jmp     -1

; 0xff 0xc0 = inc eax
db      0xc0

; 0x48 = dec eax
db      0x48
```

Output of IDA:

```
# IDA output
```

Output of objdump:

```
eb ff           jmp    0x401001
c0              byte 0xc0
48              dec    eax
```

Adopted Static Detection:

(1)

A PUSH instruction immediately followed by a RET is looked for. If found, the technique is considered as evidence detected.

(2)

A XOR instruction with two equal operands is looked for. If found and is immediately followed by a JNZ instruction, the technique is considered as evidence detected. The same happens for STC instruction immediately followed by JNC or JAE and for CLC instruction immediately followed by JC or JB.

4.3. Obfuscation Techniques

Some obfuscation techniques are described in the next sections.
Techniques currently covered by detection plugins will have an additional information: the algorithm used to detect such a technique.

4.3.1. Push Pop Math

This technique can be used to obfuscate a value and relies in three steps [24]:

- Push a known immediate
- Pop such an immediate into a register
- Do some math on the register

At the end, the register will have the desired value, but such a value does not explicitly appear in the code itself.

Adopted Static Detection:

A PUSH instruction with an immediate operand is looked for:

push immediate

If found, the next instruction is compared against a POP: if it is true, the destination (X) is saved for future use:

pop X

Then, the next instruction is compared against AND, OR and XOR with the destination operand being the saved one (X) and the other one being an immediate:

and/or/xor X,immediate

If this scenario happens, the technique is considered as detected.

4.3.2. NOP Sequence

This type of dead-code insertion relies on adding a sequence of NOP instructions in the middle of the code [25]. This can make the disassembly analysis harder by reducing the legibility of the code and bypassing some signature-based algorithms.

Adopted Static Detection:

A sequence of 5 NOPs is looked for in the same function. RET was used to consider the end of a function.
If found, this technique is considered as detected.

4.3.3. Instruction Substitution

This technique relies on changing a instruction, or a set of them, by equivalent ones. [25] [45] It can be used to make the analysis process by a professional harder and also to bypass signatures. Some examples are:

- “xor eax,eax → jz” to replace a JMP
  - For example, “jmp .destination” can be replaced by “xor eax,eax → jz .destination”
- “push → pop” to replace a MOV
  - For example, “mov eax,0x1” can be replaced by “push 0x1 → pop eax”
- “sub” to replace a XOR
  - For example, “xor eax,eax” can be replaced by “sub eax,eax”

Another example, that will be discussed in more

details, is to replace a JMP by “push → ret”.

According to [8], RET “transfers program control to a return address located on the top of the stack” and, additionally, it pops such an address to EIP. So, if the stack gets manipulates to put in its top the desired address to transfer the program control flow to, RET and its variations, such as RETN and RETF, can be used as an obfuscated JMP.
The most known way to implement such a technique is the Push Ret: the address to redirect the flow to is pushed and then RET is called effectively changing the flow:

```
push    .destination
ret
```

Although Push Ret is the most known approach, there are other variations, for example:

```
mov     [esp],DWORD .destination
ret
```

RET is often used to return from a procedure. Being so, if the alternative jump variation seems like a given calling convention function prolog, it would be more stealth and more difficult to automatically detect. For example:

```
push    .destination
push    ebp
mov     ebp,esp
leave
ret
```

Adopted Static Detection:

PUSH instruction is looked for. If found and the next instruction is a RET, then the technique is considered as detected.

4.3.4. Code Transposition

This technique relies on shuffling instructions so that the order they appear in the binary gets different from the order they were executed [25] [45].
The following two methods can be used for such a purpose:

- Shuffle the instructions and make them to be executed in the correct order by using program control flow changes. This can be achieved, for example, by using unconditional jumps and some Instruction Substitutions of it such as “xor eax,eax – jz” being used instead of a JMP instruction.
- Choose and reorder set of instructions that does not interfere in each other results. So, such a shuffling process will change the order of instructions in the binary and at the time does not change the program results

As an example, the following code is considered as the binary before the obfuscation process:

```
xor     eax,eax
inc     eax
push    ebx
...
```

The following code is an example of the original binary obfuscated with the program control flow changes approach:

```
        jmp     .first
.second:
        push    ebx
        jmp     .continuation
.first:
        xor     eax,eax
        inc     eax
        jmp     .second
.continuation:
        …
```

The following code is an example of the original binary obfuscated with the reordering approach:

```
        push    ebx

        ; inc depends on xor
        ; so such instruction order was not changed
        xor     eax,eax
        inc     eax
```

4.3.5. Register Reassignment

This technique relies on changing the registers used by a program or part of it [25][45].
For example, the following code shows a program before the obfuscation:

```
xor     eax,eax
inc     ebx
```

After a fictitious obfuscation which exchanges EAX by EBX and vice-versa, the following code will be generated:

```
xor     ebx,ebx
inc     eax
```

Although this technique does not make an analysis much more complicated, it can be used to bypass signatures.

4.3.6. Code Integration

This technique relies on disassembling a target program file, inserting the code to be obfuscated inside it [45][46]. In order to do that, the target program needs to be fixed. This way, the code to be obfuscated is hidden in the middle of the other program.

4.3.7. Fake Code Insertion

This is a variation of Garbage Bytes anti-disassembly technique. The idea is to insert instructions that will never be executed [38], making them to appear in the generated disassembly. This can, for example, confuse the professional that is analyzing the disassembly with lots of fake code and bypass signature-based algorithms.
The implementation is exactly the same as Garbage Bytes technique, but instead of adding garbage bytes, valid instructions are added.

```
jmp     .destination
push    0x12345678 ; fake code
inc     eax ; fake code
mov     esp,eax ; fake code
; more fake code here
.destination:
        …
```

Instead of using a simple JMP instruction, any other technique that can be used to redirect the program control flow, such as Fake Conditional Jump and Code Substitution, could be used. For example:

(1) Fake Conditional Jump example

```
        xor     eax,eax
        jnz     .fake_code
        jmp     .destination
.fake_code:
        push    0x12345678 ; fake code
        inc     eax ; fake code
        mov     esp,eax ; fake code
        ; more fake code here
.destination:
        ...
```

(2) Code Substitution example

```
        push    .destination
        ret
        push    0x12345678 ; fake code
        inc     eax ; fake code
        mov     esp,eax ; fake code
        ; more fake code here
.destination:
        ...
```

<u>Adopted Static Detection:</u>

(1)

A PUSH instruction immediately followed by a RET is looked for. If found, the technique is considered as evidence detected.

(2)

A XOR instruction with two equal operands is looked for. If found and is immediately followed by a JNZ instruction, the technique is considered as evidence detected. The same happens for STC instruction immediately followed by JNC or JAE and for CLC instruction immediately followed by JC or JB.

4.3.8. PEB->Ldr Address Resolving

PEB is a structure that contains process information. Among its fields, there is the Ldr, which points to a structure that contains information about the loaded modules for the process. [34]
It is possible to retrieve the PEB (fs:[0x30]) and access its Ldr field (0x0c). So, the loaded modules can be accessed and function addresses resolved. [34] [35] [36]

Adopted Static Detection:

A MOV instruction (mov, movsx, movzx) copying PEB address (fs:[0x30]) somewhere (X) is looked for and X is saved for future use:

mov/movsx/movzx X,op2 → Where “fs:[0x30]” is inside op2

Then, later in the same function, a MOV (mov, movsx, movzx) or a CMP (cmp, cmpxchg) instructions referencing the Ldr ([X+0x0c] in some of the operands) are looked for:

mov/movsx/movzx/cmp/cmpxchg op1,op2 → where [X+0xC] is a substring of op1 or op2

RET was used to consider the end of a function.

If this scenario happens, the technique is considered as detected.

4.3.9. Stealth Import of the Windows API

Regardless of the import table, ntdll.dll and kernel32.dll are automatically mapped into process address space [37]. It means that it is possible to access them even in an executable with no imports. Such DLLs can be access through SEH, because its first record normally points to either ntdll.dll or kernel32.dll.
To get the DLL address, the SEH could be walked until the first element, which 0x4 offset is the handler field. Then, it is possible to scan the memory looking for 'MZ' and, once found, check if it is in the correct place through 0x3C offset that is supposed to be “PE\0\0”: a handle to the module has been found. From this point on, the IMAGE_DATA_DIRECTORY entry of the DLL can be found using the 0x78 offset to get the RVA to the export directory, which, together with the previously found handle, results in the Export Directory Table address.

Adopted Static Detection:

(1)

A MOV instruction (mov, movsx, movzx) copying SEH address (fs:[0x0]) somewhere (X) is looked for and X is saved for future use:

mov/movsx/movzx X,op2 → Where “fs:[0x0]” is inside op2

Then, later in the same function, a MOV (mov, movsx, movzx) instruction referencing the PEB (fs:[0x30]) in the second operand is looked for and, if found, the algorithm is reseted.
Continuing with the next lines, a MOV instruction (mov, movsx, movzx) referencing the exception handler ([X+0x4]) in the second operand is looked for and, if found, the first operand (Y) is saved for future used:

mov/movsx/movzx Y,op2 → where “[X+0x4]” is a substring of op2

Later in the same function, a CMP instruction (cmp, cmpxchg) referencing Y in the first operand is looked for:

cmp/cmpxchg op1,? → Where Y is a substring of op1

Later in the same function, a MOV instruction (mov, movsx, movzx) with the “PE\0\0” offset relative to Y ([Y+0x3c]) in the second operand is looked for:

mov/movsx/movzx ?,op2 → Where [Y+0x3c] is a substring of op2

Later in the same function, instructions AND, OR, XOR, ADD or SUB CMP referencing the IMAGE_DATA_DIRECTORY offset (0x78) in

some of the operands is looked for

and/or/xor/add/sub ? → Where “0x78” is a substring in any of the operands

RET was used to consider the end of a function.

If this scenario happens, the technique is considered as detected.

(2)

If the IAT is empty, this technique is considered as evidence detected.

4.3.10. Function Call Obfuscation

LoadLibrary and GetProcAddress functions can be used to call any other. By only importing these two functions is possible to obfuscate function calls.

Adopted Static Detection:

If LoadLibraryA/LoadLibraryW/LoadLibraryExA/LoadLibraryExW and GetProcAddress are both found in IAT, this technique is considered as detected.

5. Anti-Virtual Machine

Some anti-virtual machine techniques are described in the next sections.
Techniques currently covered by detection plugins will have an additional information: the algorithm used to detect such a technique.

5.1. CPU Instructions Results Comparison

Some CPU instructions, due to their specific nature, have characteristic results when executed inside virtual machine solutions that can be used to infer its presence. [28]
The following instructions are examples that can be used for such a purpose:

- SIDT: Stores the Interrupt Descriptor Table Register (IDTR) content. [8] [27]. [28] [29] [30]
- SLDT: Stores the segment selector from the Local Descriptor Table Register (LDTR). [8] [27] [28] [30]
- SGDT: Stores the Global Descriptor Table Register (GDTR) content. [8] [27] [28] [30]
- STR: Stores the segment selector from the Task Register (TR). [8] [27] [28] [30]
- SMSW: Stores the machine status word into the destination operand . [8] [30] [42]

Adopted Static Detection:

Instructions SIDT, SLDT, SGDT and STR are looked for. If some of them are found, this technique is considered as detected.

5.2. VMWare – IN Instruction

I/O ports can be accessed through the privileged instructions IN and OUT: in normal cases [31] an attempt to run such instructions in user-mode will generate an exception. [28] [31]
VMWare [43] uses IN instruction in a special port (VX), that exists only inside its virtual machines, as an interface between virtual machines and VMWare software itself. So, such operation will not generate an exception if executed in user-mode inside a VMWare virtual machine. [28] [31]
This can be used to detect if an application is running inside a VMWare virtual machine.

Adopted Static Detection:

IN instruction is looked for. If it is found, this technique is considered as detected.

5.3. VirtualPC – Invalid Instruction

When an invalid instruction is executed, an exception is raised and it can be handled by the software using try/catch mechanism [31].
VirtualPC [44] relies on invalid instructions to interface between virtual machines and VirtualPC software itself. An example is the invalid instruction “0x0F 0x3F 0x07 0x0B”, which does not generate an exception inside a VirtualPC virtual machine.
This can be used to detect if an application is running inside a VirtualPC virtual machine.

Adopted Static Detection:

Starting at a byte that were not recognize as valid by the disassembler, the following four byte sequence are looked for:

0x0F 0x3F 0x07 0x0B

If this scenario happens, the technique is considered as detected.

6. New Techniques

The new techniques implemented by this work are described by the next sections.

6.1. Dynamic Approach

The static techniques in the previous section, which relied on function calls or function calls with specific parameters, are not reliably detected using only the static approach.
Being so, a dynamic approach was develop that puts a software breakpoint in the target functions. When such functions are reached, it is possible to more reliably detect the call and extract the parameters.

6.2. SSEXY Detection

SSEXY [33] is a tool developed by Jurriaan Bremer that, given a binary, obfuscates it converting many “conventional” assembly instructions to an SSE-based version. In this work, it was considered as an obfuscation technique. There were some troubles running SSEXY in the 883 executables used to test all plugins and techniques in this work because such a tool is still in an early development stage. So, it was developed some simple binaries for the specific purpose of testing the SSE obfuscation provided by SSEXY. At the end, together with the two demo binaries distributed with SSEXY, there were 9 cases to study the SSEXY obfuscation. The following pattern was identified in all the 9 cases:

```
66 0F 70 ?? ?? 66 0F DB ?? ?? ?? ?? ?? 66
0F DB ?? ?? ?? ?? ?? 66 0F EF
```

This pattern generated no false-positives when tested against the 883 executables and correctly detected SSEXY encryption in all the 9 cases. SSEXY was released in May/2012 and in more or less one month later a detection plugin was finished, tested and running in the Dissect || PE system.

Adopted Static Detection:

The following pattern is looked for in the binary:

```
66 0F 70 ?? ?? 66 0F DB ?? ?? ?? ?? ?? 66
0F DB ?? ?? ?? ?? ?? 66 0F EF
```

If found, the technique is considered as detected.

6.3. Flame Detection

The Flame malware made the news due to its rich capabilities and to the fact that it remained undetected for long time. Many researchers quickly noted the existence of embedded scripting language within the malware and pointed this as a new enhancement for malwares. We wrote a detection script to inspect all our samples for the presence of embedded scripting language, such as Lua [47].

7. Resources

The most updated version of this document can be found at: http://research.dissect.pe.
Additionally, examples for each of the attacking techniques discussed in this paper are available at: https://github.com/rrbranco/blackhat2012.

8. Conclusions and Future Directions

This research provides a guidance on protecting techniques used by malware, more specifically the anti-debugging, anti-disassembly, obfuscation and anti-VM ones. It also extrapolates the current standards in malware analysis providing the results against millions of samples.
We created examples for each of the techniques discussed in this paper, facilitating the development of the detection codes. Additionally, such codes are publicly available.
For validation purposes, this work explains how the

detections are being executed.

The research results can be expanded and hopefully we will publicly release more information, such as:

- More anti-reverse engineering techniques
- More statistics with more analyzed samples

9. Acknowledgement

Ronaldo Pinheiro de Lima – Joined our team a bit later in the research process, but gave amazing contributions.

Peter Ferrie – Great papers and feedback/discussions by email.

Jurriaan Bremer – SSEXY.

ReversingLabs for the TitaniumCore

10. References

[1] Peter Ferrie – Anti-Unpacker Tricks
[2] Peter Ferrie – The “Ultimate” Anti-Debugging Reference
[3] Peter Ferrie – Anti-Unpacker Tricks – Part Eight
[4] Evilcodecave's Weblog - RtlQueryProcessHeapInformation As Anti-Dbg Trick - http://evilcodecave.wordpress.com/2009/04 (Last access: 04/May/2012)
[5] Mark Vincent Yason – The Art Of Unpacking
[6] MSDN - BlockInput function - http://msdn.microsoft.com/en-us/library/windows/desktop/ms646290%28v=vs.85%29.aspx (Last Access: 05/May/2012)
[7] Nicolas Falliere – Windows Anti-Debug Reference - http://www.symantec.com/connect/articles/windows-anti-debug-reference (Last access: 24/June/2012)
[8] Intel - Intel® 64 and IA-32 Architectures Software Developer’s Manual - Volume 2B: Instruction Set Reference, N-Z - http://download.intel.com/design/processor/manuals/253667.pdf (Last Access: 24/June/2012)
[9] Matt Pietrek - A Crash Course on the Depths of Win32™ Structured Exception Handling - http://www.microsoft.com/msj/0197/exception/exception.aspx (Last Access: 24/June/2012)
[10] MSDN – Creating Guard Pages - http://msdn.microsoft.com/en-us/library/windows/desktop/aa366549%28v=vs.85%29.aspx (Last Access: 25/June/2012)
[11] Josh_Jackson - An Anti-Reverse Engineering Guide - http://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide (Last Access: 25/June/2012)
[12] Chuck Walbourn - Game Timing and Multicore Processors - http://msdn.microsoft.com/en-us/library/windows/desktop/ee417693%28v=vs.85%29.aspx (Last Access: 25/June/2012)
[13] Intel - Using the RDTSC Instruction for Performance Monitoring - http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf (Last Access: 25/June/2012)
[14] MSDN – GetTickCount function - http://msdn.microsoft.com/en-us/library/windows/desktop/ms724408%28v=vs.85%29.aspx (Last Access: 25/June/2012)
[15] MSDN - timeGetTime function - http://msdn.microsoft.com/en-us/library/windows/desktop/dd757629%28v=vs.85%29.aspx (Last Access: 25/June/2012)
[16] MSDN - GetLocalTime function - http://msdn.microsoft.com/en-us/library/windows/desktop/ms724338%28v=vs.85%29.aspx (LastAccess: 25/June/2012)
[17] MSDN - GetSystemTime function - http://msdn.microsoft.com/en-us/library/windows/desktop/ms724390%28v=vs.85%29.aspx (Last Access: 25/June/2012)
[18] MSDN - QueryPerformanceCounter function - http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904%28v=vs.85%29.aspx (Last Access: 25/June/2012)
[19] Justin Seitz - Gray Hat Python – Python Programming for Hackers and Reverse Engineers
[20] MSDN - NtSetInformationThread - http://msdn.microsoft.com/en-us/library/windows/hardware/ff557675%28v=vs.85%29.aspx (Last Access: 25/June/2012)
[21] MSDN - ZwSetInformationThread routine - http://msdn.microsoft.com/en-us/library/windows/hardware/ff567101%28v=vs.85%29.aspx (Last Access: 25/June/2012)
[22] Mark Stamp – Anti-Reversing Techniques -

http://www.cs.sjsu.edu/~stamp/CS286/pptSRE/SRE_anti-reversing.ppt (Last Access: 04/July/2012)
[23] MSDN - NtQuerySystemInformation function - http://msdn.microsoft.com/en-us/library/windows/desktop/ms724509%28v=vs.85%29.aspx (Last Access: 04/July/2012)
[24] Laspe Raber, Jason Raber - BlackHat 2008 - Deobfuscator: An Automated Approach to the Identification and Removal of Code Obfuscation
[25] Mihai Christodorescu and Somesh Jha - Proceedings of the 12th USENIX Security Symposium - Static Analysis of Executables to Detect Malicious Patterns
[26] objdump - http://www.gnu.org/software/binutils/ (Last Access: 12/July/2012)
[27] Intel - Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3A: System Programming Guide, Part 1 - http://download.intel.com/products/processor/manual/253668.pdf (Last Access: 12/July/2012)
[28] Stefan Bühlmann - Master Thesis – Detection of Virtual Machine Aware Malware
[29] Joanna Rutkowska - Red Pill
[30] John Scott Robin, Cynthia E. Irvine – Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor
[31] Elias Bachaalany - Detect if your program is running inside a Virtual Machine - http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual (Last Access: 12/July/2012)
[32] halfdead – Phrack Magazine - Volume 0x0c, Issue 0x41, Phile #0x08 of 0x0f - http://www.phrack.org/issues.html?issue=65&id=8 (Last Access: 12/July/2012)
[33] Jurriaan Bremer - SSEXY - https://github.com/jbremer/ssexy (Last Access: 12/July/2012)
[34] MSDN - PEB structure http://msdn.microsoft.com/en-us/library/windows/desktop/aa813706%28v=vs.85%29.aspx (Last Access: 12/July/2012)
[35] MSDN - PEB_LDR_DATA structure - http://msdn.microsoft.com/en-us/library/windows/desktop/aa813708%28v=vs.85%29.aspx (Last Access: 12/July/2012)
[36] Harmony Security – Blog - Retrieving Kernel32's Base Address - http://blog.harmonysecurity.com/2009/06/retrieving-kernel32s-base-address.html (Last Access: 12/July/2012)
[37] Alexey Lyashko - Stealth Import of Windows API - http://syprog.blogspot.com.br/2011/10/stealth-import-of-windows-api.html (Last Access: 12/July/2012)
[38] Nick Harbour – Advanced Software Armoring and Polymorphic Jung-Fu
[39] Christopher Kruegel, William Robertson, Fredrik Valeur and Giovanni Vigna – Proceedings of the 13th USENIX Security Symposium - Static Disassembly of Obfuscated Binaries
[40] IDA – http://www.hex-rays.com (Last Access: 12/July/2012)
[41] Cullen Linn and Saumya Debray - Obfuscation of Executable Code to Improve Resistance to Static Disassembly
[42] Boris Lau and Vanja Svajcer - EICAR 2008 EXTENDED VERSION - Measuring virtual machine detection in malware using DSD tracer
[43] VMWare – http://www.vmware.com (Last Access: 12/July/2012)
[44] VirtualPC - http://www.microsoft.com/windows/virtual-pc/ (Last Access: 12/July/2012)
[45] Ilsun You and Kangbin Yim – 2010 International Conference on Broadband, Wireless Computing, Communication and Applications - Malware Obfuscation Techniques: A Brief Survey
[46] Péter Ször and Peter Ferrie – VIRUS BULLETIN CONFERENCE, SEPTEMBER 2001 – Hunting for Metamorphic
[47] The Programming Language Lua - http://www.lua.org (Last Access: 12/July/2012)
[48] AlexAbramov - API Hooking with MS Detours - http://www.codeproject.com/Articles/30140/API-Hooking-with-MS-Detours (Last Access: 12/July/2012)
[49] Galen Hunt and Doug Brubacher – Microsoft Research - Detours: Binary Interception of Win32 Functions - http://research.microsoft.com/pubs/68568/huntusenixnt99.pdf (Last Access: 12/July/2012)
[50] coderrr - http://coderrr.wordpress.com/2008/08/27/how-to-get-rid-of-microsoft-detours-detoureddll/ (Last

Access: 12/July/2012)