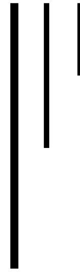

Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Προηγμένα Συστήματα Πληροφορικής»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Αυτοματοποιημένη θωράκιση των κακόβουλων προγραμμάτων μέσω της εφαρμογής επιλεγμένων anti-debugging και αντι-vm τεχνικών.** **Automated armoring of PE malwares through the implementation of selected anti-debugging and anti-vm techniques**
Ονοματεπώνυμο Φοιτητή	**Θεόδωρος Αποστολόπουλος**
Πατρώνυμο	**Μηνάς Αποστολόπουλος**
Αριθμός Μητρώου	**ΜΠΣΠ15006**
Επιβλέπων	**Κωνσταντίνος Πατσάκης, Επίκουρος Καθηγητής**

Ημερομηνία Παράδοσης **Ιούνιος 2018**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Κωνσταντίνος Πατσάκης Επίκουρος Καθηγητής	Χρήστος Δουληγέρης Καθηγητής	Ευθύμιος Αλέπης Επίκουρος Καθηγητής

Table of Contents

Abstract 5
1. Introduction 6
2. PE Overview and Basic Concepts 7
2.1 Relative Virtual Addressing 7
2.2 DOS Header 8
2.3 PE Header 8
2.4 File Header 8
2.5 Optional Header 9
2.6 Section Headers 10
2.7 PEB Structure 10
2.8 Structure of the tool 10
3 Anti-debugging techniques 14
3.1 Flags within the PEB structure & Manual Checks 14
3.1.1 Check PEB.BeingDebugged flag /kernel32.IsDebuggerPresent(). 14
3.1.2 Check PEB.NtGlobalFlag 15
3.1.3 Check Heap Flags 16
3.1.4 Anti-Step-Over 16
3.1.5 Thread Local Storage Callbacks 17
3.1.6 SS Register 18
3.1.7 Interrupt 0x2d 18
3.1.8 RDTSTC – as anti-step 19
3.1.9 Selectors 19
3.2 API Calls 20
3.2.1 CheckRemoteDebuggerPresent(). 20
3.2.2 NtQueryInformationProcess() 20
3.2.3 NtSetInformationThread() 21
3.2.4 RtlQueryProcessDebugInformation() 22
3.2.5 RtlQueryProcessHeapInformation() 22
3.2.6 Self-debugging with CreateProcess() 23
3.2.7 SwitchDesktop() 23
3.2.8 OutputDebugString() 24
3.2.9 NtQueryObject 24
3.2.10 BlockInput 26
4 Anti-VM Techniques 26
4.1 Red Pill 27
4.2 CPUID – Hypervisor presence 27
4.3 CPUID – Hypervisor Vendor 28

4.4 Number of Processors 28
4.5 Virtual Devices 28
5 Conclusion 29
References 30
List of techniques 32

Abstract

Debuggers are tools traditionally used by programmers to find errors (called “bugs”) in code. However, in the field of malware analysis, debuggers are an essential tool used to reverse-engineer malware binaries, helping analysts to understand the purpose and functionality of malware when static analysis isn’t enough. Because of their significance, many malware authors try to prevent analysts from using them. By employing various techniques in the code (known as “anti-debugging”), malware can successfully delay analysts and prolong its “life”. Moreover, malware analysis relies heavily on the use of virtualization and emulation technology to run samples in an isolated environment, for functionality and safety. However, virtual machines and emulators always create traces, so called artifacts, which malware can use to detect the execution environment.

The goal of this paper to present selected anti-debugging and anti-vm techniques and include them in a tool that can automatically append them to the basic functionality of a malware Windows binary in order to armor it.

Περίληψη

Τα προγράμματα απασφαλματοποίησης είναι εργαλεία τα οποία χρησιμοποιούνται συνήθως από προγραμματιστές έτσι ώστε να εντοπίσουν σφάλματα (τα λεγόμενα “bugs”) στον κώδικα. Ωστόσο, στον τομέα της ανάλυσης κακόβουλου λογισμικού, αποτελούν βασικό εργαλείο στην διαδικασία αντίστροφης μηχανικής ενός κακόβουλου εκτελέσιμου, βοηθώντας τους αναλυτές να κατανοήσουν τον σκοπό και την λειτουργικότητά του, όταν η στατική ανάλυση δεν είναι αρκετή. Εξαιτίας της σημαντικότητάς τους οι δημιουργοί κακόβουλων προγραμμάτων προσπαθούν να εμποδίσουν τη χρήση τους από τους αναλυτές. Χρησιμοποιώντας ποίκιλες τεχνικές (γνωστές ως “anti-debugging”), ένα κακόβουλο πρόγραμμα μπορεί να καθυστερήσει τους αναλυτές και να παρατείνει το χρόνο «ζωής» του. Επιπρόσθετα, η ανάλυση κακόβουλου λογισμικού βασίζεται σε μεγάλο βαθμό στη χρήση τεχνολογιών εικονικοποίησης και εξομοίωσης έτσι ώστε να εξετάσει δείγματα του κακόβουλου προγράμματος, σε απομονωμένο περιβάλλον, για λόγους λειτουργικότητας και ασφάλειας. Ωστόσο οι τεχνολογίες αυτές, αναπόφευκτα, δημιουργούν ίχνη, τα οποία ένα κακόβουλο πρόγραμμα μπορεί να ανιχνεύσει

Στόχος της παρούσας εργασίας είναι να παρουσιάσει επιλεγμένες anti-debugging και anti-vm τεχνικές οι οποίες μέσω ενός εργαλείου θα ενσωματώνονται αυτόματα σε ένα κακόβουλο εκτελέσιμο Windows με σκοπό την θωράκισή του.

1 Introduction

Debuggers are vital tools for malware analysis since they enable detailed analysis of malware's behaviors. This involves the step by step execution of a malware to examine its current state as well as the ability to make changes to memory space, registers, variable values, configurations and more. Furthermore, it enables the disassembling of the binary code, tracing of system calls, capturing of exceptions etc. [40]. In general, they allow the inspection of code in greater detail than static analysis and give full control over the malware's low-level runtime behaviors. Therefore; debugging eases the burden of understanding of malware's behavior, mechanisms and capabilities. But debugging, as most of the techniques in the field of malware analysis are a doubled-edged sword. Malware authors know that malware analysts use debuggers to figure out how malware operates, and the authors use anti-debugging techniques, in an attempt to slow down the analyst as much as possible and thwart the incident response. Anti-debugging is the implementation of one or more techniques within computer code that hinders attempts at reverse engineering or debugging a target binary [10]. By the time malware realizes that it is executed inside a debugger, it may divert from its normal code execution path or modify the code to cause a crash, thus preventing the process of analysis from being carried out and adding time and cost to analyst's efforts. Generally, the main objective of anti-debugging is to prevent the process of reverse engineering by detecting the existence of a debugger and behave differently when a debugger is attached to the current process or by interrupting/crashing the debugger. In fact, there is no flawless solution to deter the experienced malware analyst. However, making the analysis procedure as exhausting and painful as possible increases the time and skills needed for a complete analysis of the hidden procedures inside the malware.

Malware analysis also applies technologies of virtualization [21], [22], [23] and emulation [24], [25], [26] to quarantine malware in an area where it can be studied and dissected in order to analyze its behavior at runtime. On the contrary, malware authors write malwares that can recognize when they are running inside a virtual machine/sandbox. If they can accomplish that objective, the malware will avoid taking any malicious actions until it reaches a specific target machine, and as a result escape the analysis mechanism and hide its true malicious nature. Virtual machines detection based mostly on execution artifacts with predicted behavior [41]. These artifacts may include additional operating system files and processes necessary for the virtualization to work, supplementary CPU features, hardware parameters, timing attacks (code inside virtual machine is expected to execute slower than on host) etc.

A variety of anti-debugging, anti-virtualization, and anti-emulation techniques [17], [12], [28], [31], [32], [33] exists that in many times can give malware the ability to detect the presence of a VM or emulator and alter its behavior to hide itself and escape analysis. According to [2] malware authors are using more and more anti-debugging and anti-VM techniques to thwart the analysis, and also state that the use of anti-debugging and anti-VM techniques in malware might increase over years while the use of these evasive techniques help malware to evade many antivirus products. In [11] it is showed that 39.9% and 2.7% of 6,222 malware samples use anti-debugging and anti-virtualization techniques respectively. Also, in [12] researchers found that 43.21% of 4 million samples are armed with anti-debugging techniques and 81.4% armed anti-VM behaviors. Furthermore in [13] it is demonstrated the detection of 5,835 malware samples (out of 110,005) that exhibit evasive behaviors and finally in [15] there is a detection of evasion behavior in 25.6% of 1,686 malicious binaries. All these studies show that anti-virtualization and anti-debugging techniques have become the most popular methods of evading malware analysis.

The rest of the paper is organized as follows: Section 2 presents briefly the structure and basic concepts of the PE file format, basic concepts of Windows insides related to the mechanisms of the tool

as well as the structure of the tool. Section 3 analyses the anti-debugging techniques used by the program, Section 4 analyses the anti-VM techniques used and finally Section 5 concludes.

2 PE Overview and Basic Concepts

In order to understand the content of this paper better, one must be familiar with the Portable Executable file format. Portable executable file format is the Windows standard executable format type used in x86 and x64 architectures. The term "portable" refers to format's scalability within numerous environments of operating system software architecture. The Windows loader needs to handle the wrapped executable code so the structures contained in the PE file format actually maintain all the necessary information to achieve this. Predecessor to the PE file format was the COFF format used in Windows NT. This standard is created by Microsoft and while some structures are partially documented it is available at [5]. PE files have extensions like .exe, .dll, .sys (driver files) and others

The way PE files are laid out in memory is actually very similar to the executable file on disk. The loader uses the memory-mapped file mechanism to map the appropriate pieces of the file into the virtual address space. To use a construction analogy, a PE file is like a prefabricated home [4].

2.1 Relative Virtual Addressing

Windows makes heavy use of RVAs in order to represent addresses in PE headers and achieve position independence as Windows may load an executable into any location in memory. Absolute memory addresses cannot be used because an executable normally will be loaded into an unknown address in memory. The concept of RVA gives the ability to determine virtual addresses of data in memory as an offset relative to a specific section in the file. The compiler generates an RVA and then the Windows loader converts that RVA into an actual virtual address at runtime because the loader knows where the executable will be loaded into [8].

When parsing the raw binary, we need to convert RVAs into file offsets because the executable is not loaded in memory but instead what it is parsed is the static content inside the file as it is on disk. In order to calculate the raw offset in disk from a given RVA we need to find the PE section that belongs to. We determine that a given RVA belongs to a specific section by looping through the section headers, and check if that RVA is inside that section. The base RVA of a section is where the section start is loaded in memory so by subtracting from RVA we get the actual offset. Now if we add the file offset of the section’s beginning on disk we can get the actual file offset of the given RVA. The following equations summarize the above:

file_offset = section_raw_offset + (RVA - section_base_RVA).

By looping through the array of section headers we can find which section contains the RVA by examining:

if section_base_rva <= rva < (section_base_rva + section_virtual_size)

The following definitions are related to addresses in the PE format. Addresses are either physical or virtual (in-memory), and either relative or absolute.

Physical address: A physical address is the offset of a certain byte in a file as it is written on disk. Physical addresses are necessary to access parts of the PE file that must be read from disk.

Base address: The base address is the address of the first byte when the file is loaded in memory. PE files specify a preferred base address in a field called ImageBase (see Optional Header below). If the image file cannot be loaded at the preferred address into process space, another base address is applied, which is known as rebasing.

RVA: Relative virtual addresses (RVA) are used while an image file is loaded in memory. They are relative to the base address of the image file or to another RVA. RVAs are a way to specify addresses in

memory independently from the base address. This makes it possible to rebase the file without having to re-calculate all in-memory addresses in the file.

VA: Virtual address (VA) is an absolute in-memory addresses. Although the PE/COFF specification defines a VA this way, it uses the term also for addresses that are actually relative to the image base.

Entry point: The entry point is a RVA to the starting address for EXE files, or to the initialization function for device drivers for DLL files (see AddressOfEntryPoint)

2.2 DOS Header

It is important to understand that the PE header is not the first item loaded into memory when a program is executed. A Microsoft DOS portion of the executable runs first to determine if a compatible version of Microsoft Windows is being used. At the start of every PE file we find DOS header, an MS-DOS executable ("stub") which is responsible to flag a PE file as a valid MS-DOS executable. Its structure can be found as IMAGE_DOS_HEADER in WINNT.H. The MS-DOS header is the same MS-DOS header since version 2 of the MS-DOS operating system. The main reason for keeping the same structure intact at the beginning of the PE file format is so that, when you attempt to load a file created under Windows version 3.1 or earlier, or MS DOS version 2.0 or later, the operating system can read the file and understand that it is not compatible. In other words, when you attempt to run a Windows NT executable on MS-DOS version 6.0, you get this message: "This program cannot be run in DOS mode." If the MS-DOS header was not included as the first part of the PE file format, the operating system would simply fail the attempt to load the file and offer something completely useless, such as: "The name specified is not recognized as an internal or external command, operable program or batch file."[42].

The MS-DOS header fills the first 64 bytes of the PE file. The only two fields that matter are the first field e_magic and the last field e_lfanew. The first field, e_magic, (contains magic bytes) is used to check for MS-DOS-compatibility. These magic bytes contain the value 0x54AD, which represents the ASCII characters MZ. Anti-virus products usually use this classical signature to locate PE executable in memory. The final field, e_lfanew, is a 4-byte offset into the file where the actual PE file header and is located at offset 0x3C (60-64) from the beginning of the file. By indexing the e_lfanew field of the MS-DOS header we take an offset in the file, so we can add the file's memory-mapped base address to determine the actual memory-mapped address [42]. All other field of the DOS structure are actually useless, compilers and linkers may ignore them. Malware actually can malform its standard format due to the fact that many fields in the PE headers are ignored [6][7].

2.3 PE Header

The PE File Header is placed after the MS-DOS Stub. While the PE header usually follows the DOS header it might not be the case as that isn’t defined by the specification as a strict rule in respect to the file layout so by moving the location of the PE header can lead to several specific malformation cases which have the potential of affecting security and reverse engineering tools [7]. The information in the PE file is basically high-level guidelines used by Windows loader or the executable to determine how to load the file. All substructures of the PE File Header, normally, are consecutively arranged, thus located at a fixed offset from the beginning of the PE signature. The PE File Header consists of the PE signature, a 32-bit-signature with the magic number 0x00004550 (this signature is "PE\0\0"), the (COFF) File Header, the Optional Header, and the Section Table.

2.4 File Header

The PE File header is called IMAGE_FILE_HEADER (WINNT.H) and contains limited high-level information about the executable such as the type of machine the binary it can be executed (**Machine**), the number of sections in it (**NumberOfSections**), the time it was linked, whether it is an executable or a DLL etc.

2.5 Optional Header

Immediatly following the file header is the IMAGE_OPTIONAL_HEADER (WINNT.H), that contains information the loader needs. Most important fields are:

AddressOfEntryPoint: This is a 32-bit-value an offset to where the execution starts a.k.a the codes's entry point. This is for example the address of a DLL's LibMain() or the program's startup code which will in turn call main() or a driver's DriverEntry().

ImageBase: This is a 32-bit-value which indicates the preferred (linear) load address of the entire binary, including all headers. The linker will relocate the file to this is the address. The preferred load address cannot be used if another image has already been loaded to that address or the memory in question has been used for other purposes (stack, malloc(), uninitialized data, whatever). In these cases, the image must be relocated and loaded to some other address.

SectionAlignment: The field is a 32-bit-value that holds information about the alignment of the PE-file's sections in RAM (when the image has been loaded). SectionAlignment is 4096 in most cases.

FileAlignment: The field is a 32-bit-value that holds information about the alignment of the PE-file's sections as it is on disk. Usually FileAlignment is 512.

SizeOfImage: This is a 32-bit-value giving the amount of memory the image will need, in bytes ('SizeOfImage'). It is the sum of all headers' and sections' lengths if aligned to 'SectionAlignment'. It is a hint to the loader how many pages it will need in order to load the image.

DllCharacteristics: The DLL characteristics field of the executable image are also really important for a malware analyst

IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE (0x0040) - The DLL can be relocated at load time this means we have ASLR enabled.

IMAGE_DLLCHARACTERISTICS_NX_COMPAT (0x0100) - The image has data execution prevention (DEP) enabled.

IMAGE_DLLCHARACTERISTICS_NO_SEH (0x0400) - The image does not use structured exception handling (SEH). This is important because no handlers can be called in this image and exception handling has been used extensively as an attack vector.

Data directories: The data directory acts phonebook that gives us the locations of maybe the most important components of the executable in the file. It is an array of IMAGE_DATA_DIRECTORY structures that are located at the end of the optional header structure. Currently the specification defines 16 indexes of data directories. However, the number of data directories is completely up to the compiler/linker/malicious actor so the NumberOfRvaAndSizes field of the IMAGE_OPTIONAL_HEADER is important to know how many to parse.

At each index is a IMAGE_DATA_DIRECTORY structurethat has the following form:

- VirtualAddress - The offset of the table
- Size - The size of the table, in bytes.

The VirtualAddress field is an offset into a specified section in memory. Each data directory entry specifies the size and relative virtual address of the directory. To locate a particular directory, we have to determine the relative address from the data directory array in the optional header. Then use the virtual address to determine which section the directory is in. Once we determine which section contains the directory, the section header for that section is then used to find the exact file offset.

2.6 Section Headers

Last in PE Header and before the raw data for the image's sections lies the section table. This structure is called IMAGE_SECTION_HEADER and contains necessary information about each section in the image. The sections in the image are sorted by their starting address (RVAs) and their number is determined by NumberOfSections field in the file header (IMAGE_FILE_HEADER) structure. We will discuss some of the important entries below.

Name: This is 8-byte, null-padded UTF-8 string. There is no terminating null character if the string is exactly eight characters long. For longer names, this member contains a forward slash (/) followed by an ASCII representation of a decimal number that is an offset into the string table. Executable images do not use a string table and do not support section names longer than eight characters.

VirtualSize: The total size of the section when loaded into memory, in bytes. If this value is greater than the SizeOfRawData member, the section is filled with zeroes. This field is valid only for executable images and should be set to 0 for object files.

VirtualAddress: The address of the first byte of the section when loaded into memory, relative to the image base. For object files, this is the address of the first byte before relocation is applied.

SizeOfRawData: The size of the initialized data on disk, in bytes. This value must be a multiple of the FileAlignment member of the IMAGE_OPTIONAL_HEADER structure. If this value is less than the VirtualSize member, the remainder of the section is filled with zeroes. If the section contains only uninitialized data, the member is zero.

PointerToRawData: A file pointer to the first page within the PE file. This value must be a multiple of the FileAlignment member of the IMAGE_OPTIONAL_HEADER structure. If a section contains only uninitialized data, set this member is zero.

Characteristics: The characteristics of the image. For example, the value IMAGE_SCN_MEM_EXECUTE (0x20000000) shows the section can be executed as code.

2.7 The PEB Structure

The PEB is a complex data structure which maintains in its fields important data about the current process, while many of these fields may point to other structures lower in the PEB. Every process has its own PEB and the Windows Kernel will also have access to the PEB of every user-mode process, so it can keep track of certain data stored within it. The PEB structure comes from the Windows Kernel (although is accessible in user-mode as well). PEB has been abused for malicious purposes in the past, but Microsoft has made many changes over the recent years to help prevent this [9]. The PEB is the user-mode portion of Windows process control structures. We will see in the following that parsing this structure can give us access to the Windows API functions and also that many anti-debugging tricks are based upon specific values of the fields inside PEB.

2.8 Structure of the tool

Basically, to add the anti-debugging code the program adds an executable section to the malware on disk, with a name (for that section) provided by the user. The technique used to add a section is similar to one of the many techniques viruses used in traditional PE infection-backdooring. The tool first accepts two arguments, the target executable and the name of the new section to add. The first function AddSection() first reads the file and check its DOS signature for a PE validity, and x86 compatibility (doesn't support x64 executables). It then checks if the name of section we want to add already exists to continue. The next step is to copy the name of the section in the Name field of the newly created section header and add all the required information. Specifically:

1.The total size of the section when loaded into memory.

Misc.VirtualSize = size of new section, aligned by Optional Header's SectionAlignment

2. The address of the first byte of the section relative to the image base when the section is loaded into memory will be equal to previous section's end.

VirtualAddress = previous section's Misc.VirtualSize, aligned by OptionalHeader SectionAlignment

3.The size of new section on disk

SizeOfRawData = size of new section, aligned by Optional Header's FileAlignment

4. The file pointer to the first page of the section must be equal to the end of the previous section on disk

PointerToRawData = previous section's SizeOfRawData, aligned by Optional Header's FileAlignment

5. The Characteristics of the new Section to make it executable writable readable

Characteristics = 0xE00000E0; // 0xE00000E0 = IMAGE_SCN_MEM_WRITE | IMAGE_SCN_CNT_CODE | IMAGE_SCN_CNT_UNINITIALIZED_DATA | IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ

6. The SizeOfImage field in the Optional Header , which indicates the size as the image is loaded in memory has changed

SizeOfImage = VirtualAddress + Misc.VirtualSize

7.The number of sections in the FileHeader also has changed

NumberOfSections +=1

8. Finally the changes are written to the file

C++

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

fig.26 Section Header structure

Next, is the function that adds the inline assembly code that also contains the anti-techniques. We create 2 labels in assembly. One represents the start of the opcode section and one is the end. The __asm code snippets between is where the actual code that is to be written resides. So, by subtracting the end from the start address we get the offset to the code we want to write without copying the other opcodes, thus corrupting the integrity of the infected anti-reverse code in the binary. We jump over the middle code, so we don't execute it in the target binary itself by using the jmp instruction (jmp to label over).

```
DWORD start(0), end(0);

__asm
{
    mov eax, loc1
    mov[start], eax
    jmp over //we jump over the second __asm,so we dont execute it in the infector itself
    loc1 :
}
```

```
__asm {
over:
mov eax, loc2
mov[end], eax
loc2 :
}
```

fig.27 asm labels

We also manually create an entry in the Optional Header's TLS Directory (index 9) by specifying it's size (sizeof(IMAGE_TLS_DIRECTORY)) in the Size field and update its Virtual Address (TLS Directory RVA) field to point 4 bytes after the start of our newly created section.

```
///////////////We Update TLS Directory, directory number 9
nt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].Size = sizeof(IMAGE_TLS_DIRECTORY);
nt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].VirtualAddress = last->VirtualAddress + 4; //it starts 4 bytes after the start of last section
```

fig.28 Create TLS Directory Entry

Finally, manually, again we create the tls directory itself that is located in our new section. The IMAGE_TLS_DIRECTORY has the following prototype:

```
struct _IMAGE_TLS_DIRECTORY {
0x00  DWORD       StartAddressOfRawData;
0x04  DWORD       EndAddressOfRawData;
0x08  LPDWORD     AddressOfIndex;
0x0c  PIMAGE_TLS_CALLBACK *AddressOfCallBacks;
0x10  DWORD       SizeOfZeroFill;
0x14  DWORD       Characteristics;
};
```

The only important fields are (1) AddressOfIndex which is the location to receive the TLS index, which the loader assigns at runtime. This location is in an ordinary data section, so it can be given a symbolic name that is accessible to the program. (2) AddressOfCallbacks which is a pointer to an array of TLS callback functions. The array is null-terminated, so if no callback function is supported, this field points to 4 or 8 bytes set to zero. The steps that the tool follows to create the tls structure are:

1. Write 4 null bytes at the start of the section
2. After the 4 null bytes we place the IMAGE_TLS_DIRECTORY structure (size 24 bytes)
3. AddressOfIndex = (ImageBase + section address) + 24 (sizeof(IMAGE_TLS_DIRECTORY)) + 4 {null bytes)
4. AddressOfCallbacks = (ImageBase + section address) + 24 (sizeof(IMAGE_TLS_DIRECTORY)) + 4 (null bytes) + 4 (AddressOfIndex)
5. Finally, we update the address to where AddressOfCallbacks points to be 40 bytes after the start of the section, this is where the inline assembly code begins. This is the first TlsCallback, we also leave space (4 null bytes) for one more that will be added dynamically.

```
////////////Set a pointer to last sections start and write four null bytes
SetFilePointer(file, last->PointerToRawData, NULL, FILE_BEGIN);
char index[4] = { 0x00, 0x00, 0x00, 0x00 };
WriteFile(file, index, sizeof(index), &dw, 0);

//////////////Set file pointer after the four null bytes
SetFilePointer(file, last->PointerToRawData + 4, NULL, FILE_BEGIN);
//////////// where we create the TLS_DIRECTORY
IMAGE_TLS_DIRECTORY* pTLS = new IMAGE_TLS_DIRECTORY();

/////////////the AddressOfIndex inside the tls directory points exactly after the directory which is 28 bytes, 4*0x00 + 24 for the tls directory
pTLS->AddressOfIndex = nt->OptionalHeader.ImageBase + last->VirtualAddress + 28;
////////the AddressOfCallBacks points to the 1st callback it is located at: 0x00*4 + 24(tlsdirectory) + 4(addressofindex) = 32
pTLS->AddressOfCallBacks = nt->OptionalHeader.ImageBase + last->VirtualAddress + 32;
WriteFile(file, (PVOID)pTLS, sizeof(IMAGE_TLS_DIRECTORY), &dw, 0);

////////
SetFilePointer(file, last->PointerToRawData + 28, NULL, FILE_BEGIN);
DWORD dwOffsetOfIndexDWORD = nt->OptionalHeader.ImageBase + last->VirtualAddress;
WriteFile(file, &dwOffsetOfIndexDWORD, sizeof(DWORD), &dw, 0);

////////////
SetFilePointer(file, last->PointerToRawData + 32, NULL, FILE_BEGIN);
DWORD dwOffsetOfCodeDWORD = nt->OptionalHeader.ImageBase + last->VirtualAddress + 16 + sizeof(IMAGE_TLS_DIRECTORY);
WriteFile(file, &dwOffsetOfCodeDWORD, sizeof(DWORD), &dw, 0);

//////////////make room for another callback + 4 null bytes to end it
SetFilePointer(file, last->PointerToRawData + 36, NULL, FILE_BEGIN);
char index2[12] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
WriteFile(file, index, sizeof(index), &dw, 0);
```

fig.29 Create TLS Structure manually

Finally, many anti-debugging techniques rely on functions exported by kernel32.dll, ntdll.dll and user32.dll so the tool must resolve their addresses memory. It first reads the base address of kernel32.dll or ntdll.dll, (that are always loaded) from PEB, and walk their export table to search for needed functions by comparing their names. There are many techniques to do that here we use the following:

1. Get a pointer to the PEB

mov eax, fs: [30h]

2. PEB_LDR_DATA structure is a structure that contains information about all of the loaded modules in the current process. Is located from a pointer at offset 0x0c in the Win32 Process Environment Block (PEB).

mov eax, [eax + 0x0c]

3. Get PEB->Ldr.InMemoryOrderModuleList.Flink(1st entry). The InMemoryOrderLinks contains Flink/Blink to the next/previous loaded module.

mov eax, [eax + 0x14]

second entry is ntdll.dll

mov eax, [eax]

third entry is kernel32.dll

mov eax, [eax]

4. Get the 3rd entry's base address(kernel32.dll)

mov eax, [eax + 0x10]

mov ebx, eax; base address of kernel32 in ebx

5. Get its PE Header

mov eax, [ebx + 0x3c]; PE header VMA

6. Go to its export table

mov edi, [ebx + eax + 0x78]; Export table relative offset, PE + 0x78 (i.e., offset 120 bytes) is the relative address (relative to DLL base address) of the export table

add edi, ebx; Export table VMA

7. Get Number of names to use it as a counter to walk its functions

mov ecx, [edi + 0x18]; NumberOfNames

8. Get Address of names

mov edx, [edi + 0x20]; Names table relative offset

add edx, ebx; Names table VMA

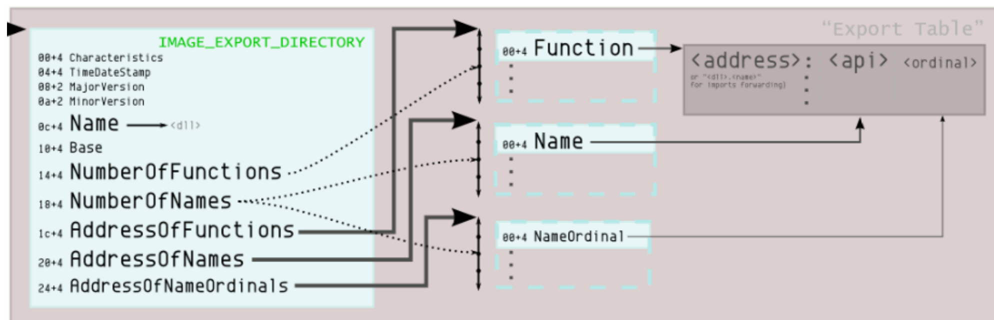

fig.31 Export Directory

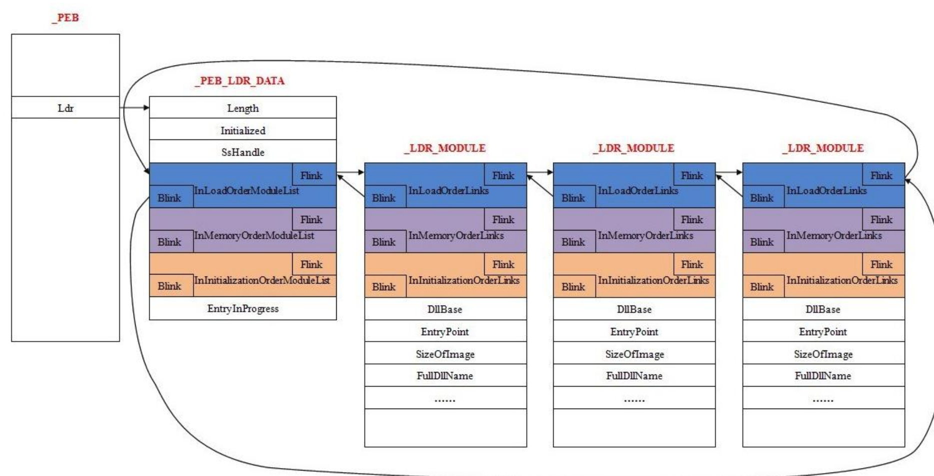

fig.30 Walking the PEB

3 Anti-debugging techniques

3.1 Flags within the PEB structure & Manual Checks

3.1.1 Check PEB.BeingDebugged flag /kernel32.IsDebuggerPresent().

The most popular native Windows function in kernel32.dll that can be used to check if a process is being debugged is the IsDebuggerPresent() function which returns TRUE if a debugger is detected. The problem with such method is that the function is easily traceable and by changing the return value to 0, such protection will be bypassed. Also, this is first anti-debugging method that most new reverse engineers discover is the Windows API, so it is probably one of the first things a malware analyst will observe. Since a call to this function is easy to detect, a similar method is to replace it by the method used inside this function in order to bypass the API call and directly access the details of the running PEB. A Windows PEB structure is maintained by the OS for each running process. As stated before (see PEB section above), it contains all user-mode parameters associated with a process. These parameters include the process's environment data, which itself includes environment variables, the loaded modules list, addresses in memory, and debugger status [1]. What the IsDebuggerPresent() function really does inside is to check the BeingDebugged byte of the PEB structure. The BeingDebugged flag in PEB at offset 0x2 is set when the current process is under debugging. Thus, this flag can indicate the existence of debugger. When operating in Windows 32-bit systems, PEB location can be referenced from

fs segment register at offset 0x30, where fs:[0x00] is the address of TIB (Win32 Thread Information Block). So, the check can be made using this code:

mov eax, fs:[30h] ; PEB
cmp byte ptr [eax+2],0 ; BeingDebugged flag
jne being_debugged

Because of its simplicity there is a plethora of anti-anti-debug plugins that can disable this technique so, it is mostly used obfuscated by inserting junk code/garbage bytes, or other techniques as an extra anti-disassembly trick. fig.1 shows an example of PEB.BeingDebugged technique using speculative execution and fig.2 an example using code transposition.

```
/////////////////////PEB being debugged with speculative execution

__asm
{
    xor ebx, ebx
    call spec_exec
    spec_exec:
    push ebp
    mov ebp, esp
    mov eax, 0xffffffff
    _emit 0xc7; xbegin:
    _emit 0xf8
    _emit 0
    if_xbegin_neg:
    cmp eax, 0xffffffff ; -1
    jnz if_xbegin_not_neg

    mov     eax, fs:[30h]; PEB
    mov     al, [eax+68h] ; BeingDebugged
    mov     bl, al ; al=1 -> debug, al=0 ->no debug
    ; end of the speculative execution

    _emit 0x0f ; xend:
    _emit 0x01
    _emit 0xd5
    jmp end_if_xbegin

    if_xbegin_not_neg:
    mfence

    end_if_xbegin:
    ; epilogue
    cmp bl, 0
    jne being_debugged
    mov esp, ebp
    pop ebp
}
```

fig.1 BeingDebugged /w spec execution

```
/////////// PEB.BeingDebugged flag with code transposition applied.
__asm{
    jmp step1
    step3:
    test eax, eax
    jmp step4
    step2 :
    movzx eax, byte ptr[eax + 2]
    jmp step3
    step4 :
    jne being_debugged
    jmp continue_execution
    step1 :
    mov eax, fs : [30h]
    jmp step2
    }
```

fig.2 BeingDebugged /w code transposition

3.1.2 Check PEB.NtGlobalFlag

Processes, when started under a debugger, in general run slightly differently than those started without a debugger attached. In particular, debugged processes create memory heaps differently than those not being debugged. The information that is stored within the PEB NtGlobalFlag at offset 0x68 for Windows 32-bit informs the kernel how to create heap structures. The default value is always 0 and doesn't change when a debugger is attached to the process. There are several methods in which the NtGlobalFlag can be changed to detect the presence of a debugger. The NtGlobalFlag contains many flags which affect the running of a process. The most common flags which are set with NtGlobalFlag when a debugger creates a process, is the heap checking flags:

FLG_HEAP_ENABLE_TAIL_CHECK (0x10),

FLG_HEAP_ENABLE_FREE_CHECK (0x20), and

FLG_HEAP_VALIDATE_PARAMETERS (0x40).

Thus, a way to detect the presence of a debugger is to check if the bits corresponding to 0x70 (the sum of the above flags) are set in NtGlobalFlag.

```
//////////////////////Check PEB.NtGlobalFlag
    __asm {
        mov eax, fs:[30h]; Process Environment Block
        mov al, [eax + 68h]; NtGlobalFlag
        and al, 70h
        cmp al, 70h
        je being_debugged
    }
```

fig.3 NtGlobalFlag

These three flags are usually set for the process that is created by the debugger, but for the process to which the debugger attaches (attach), these flags are not set. There are three exceptions:

1 - additional flags can be set for all processes, through the system registry.

[HKEY_LOCAL_MACHINE \ System \ CurrentControlSet \ Control \ Session Manager]

"GlobalFlag" = "0x00000000"

2 - The second is through the registry for a specific program

[HKEY_LOCAL_MACHINE \ SOFTWARE \ Microsoft \ Windows NT \ CurrentVersion \ Image File Execution Options \ <file name>]

"GlobalFlag" = "0x00000000"

3 - All flags can be controlled by loading the configuration structure (Load Configuration Structure).

3.1.3 Check Heap Flags

The PEB structure contains the pointer to the process heap – the _HEAP structure-, known simply as ProcessHeap, that is set to the location of a process's first heap allocated by the loader. ProcessHeap is located at 0x18 in the PEB structure, for Win32 Systems, but also can be retrieved by the GetProcessHeap(). This first heap contains a header with fields used to tell the kernel whether the heap was created within a debugger. These are known as the ForceFlags and Flags fields [13]. Due to the flags set in NtGlobalFlag, heaps that are created will have several flags turned on, and that this behavior can be observed inside ntdll!RtlCreateHeap(). Typically, the initial heap created for the process (PEB.ProcessHeap) will have its Flags and ForceFlags fields set to 0x02 (HEAP_GROWABLE) and 0x0 respectively. However, when a process is being debugged, these flags are usually set to 0x40000062 (depending on the NtGlobalFlag) and 0x40000060. By default, the following additional heap flags are set when a heap is created on a debugged process [14]:

- HEAP_TAIL_CHECKING_ENABLED (0x20)
- HEAP_FREE_CHECKING_ENABLED (0x40)

The values in those fields are affected by the presence of a debugger, but also depend on the version of Windows. The location of those fields also depends on the version of Windows since the _HEAP structure is undocumented.

```
/////////////////////heap FLAGS
call getversion  ;GetVersion API call
cmp al, 6
cmc
sbb ebx, ebx
and ebx, 34h
mov eax, fs:[30h]; Process Environment Block
mov eax, [eax + 18h]; get process heap base
mov eax, [eax + ebx + 0ch]; Flags; not HEAP_SKIP_VALIDATION_CHECKS
bswap eax
and al, 0efh
cmp eax, 62000040h; HEAP_GROWABLE + HEAP_TAIL_CHECKING_ENABLED + HEAP_VALIDATE_PARAMETERS_ENABLED + HEAP_FREE_CHECKING_ENABLED  reversed by bswap
je being_debugged

///////////////////////////heap FORCE FLAGS
call getversion
cmp al, 6
cmc
sbb ebx, ebx
and ebx, 34h
mov eax, fs:[30h]; Process Environment Block
mov eax, [eax + 18h]; get process heap base
cmp[eax + ebx + 10h], 40000060h; ForceFlags = HEAP_TAIL_CHECKING_ENABLED + HEAP_FREE_CHECKING_ENABLED + HEAP_VALIDATE_PARAMETERS_ENABLED
je being_debugged
```

fig.4 Heap Flags

3.1.4 Anti-Step-Over

Most debuggers support stepping over certain instructions, such as "call" and "rep" sequences. In such cases, a software breakpoint is often placed in the instruction stream, and then the process is allowed to

resume execution. The debugger normally receives control again when the software breakpoint is reached. However, in the case of the "rep" sequence, the debugger must check that the instruction following the rep prefix is indeed an instruction to which the rep applies legally. Some debuggers assume that any rep prefix precedes a string instruction. This introduces a vulnerability when the instruction following the rep prefix is another instruction entirely. Specifically, the problem occurs if that instruction removes the software breakpoint that would be placed in the stream if the instruction were stepped over. In that case, when the instruction is stepped over, and the software breakpoint is removed by the instruction, execution resumes under complete control of the process and never returns to the debugger [13]. The code below will detect a breakpoint that is placed at anti_step_loc1. It works by copying the value at anti_step_loc1 over the "90h" at anti_step_loc11+1. The value is then compared at anti_step_loc2.

```
////////////////////////////////////antistep-over
__asm{

    xor     ecx, ecx
    inc     ecx
    call    anti_step_1
    anti_step_1:
    pop     esi
    add     esi, 9
    lea     edi, [esi + 1]
    rep     movsb
    anti_step_1_loc1:
    mov     dl, 0x90
    xor     eax, eax
    anti_step_1_loc2:
    cmp     dl, 0xcc
    je being_debugged
    setz    al
}
```

fig.5 Anti-Step-Over

3.1.5 Thread Local Storage Callbacks

Threads are like individual execution mechanisms that run inside of a process. They mainly used for parallel execution in order to accomplish multiple actions concurrently. Usually threads inside the process context share the same memory, but they can also maintain their own private local storage. This storage is similar to a stack but is only accessible to a specific thread. There is a certain chunk of memory that will be reserved for this thread, and variables can be stored in it. This way, only this one thread has access. Windows also allows for threads to define their own initialization and deconstruction routines. All of this is captured inside the TLS data directory entry. The TLS data directory entry is an RVA that points to a IMAGE_TLS_DIRECTORY32 or IMAGE_TLS_DIRECTORY 64 depending on the CPU architecture (defined in WINNT.H) [43]. TLS callback is a very powerful anti-debugging technique and good place to perform debugger presence check since the Callback function will be called before the executable reach the main module entry point. To use the TLS directory, we must create an entry in the PE file format's Optional Header (see above at Structure of the tool). The TLS structure, *IMAGE_TLS_DIRECTORY*, pointed to by the TLS directory entry has a small number of fields. The one of special interest is the one pointing to a list of callbacks, *AddressOfCallBacks.* Windows Loader reads Data Directories -> TLS Directory, if VirtualAddress isn't zero, the loader reads Address of Callbacks Array and executes the first callback in the list and continues by reading the next element of Address of Callbacks Array. This means that we can add/remove new TLS callbacks from another TLS callback dynamically. It is possible to register/unregister new TLS callbacks on the fly even after the file has been loaded, since the Windows Loader re-reads PE-header and the section where the callbacks are stored every time it need the data. It is possible to change tls table while in tls callback itself; those added will execute normally (table is not

cached by loader). In Fig. 32 a new callback is added in the AddressOfCallbacks immediately after the first callback that points to the code exactly after.

```
/////////////////////////////////Add TLS Dynamically
    __asm {
        call next
        next:
        pop eax          //get current instruction address is start + 5
        mov ebx, eax      //store address
        sub eax, 0x2d     // sub (24 + 5)h=29h to go to section start
        add eax, 0x24         //2nd callback address
        add ebx, 0xf          //
        mov [eax], ebx
        retn

    }
```

fig.32 Dynamically added TLS Callback

3.1.6 SS Register

There is a simple trick to detect single-stepping that has worked since the earliest of Intel CPUs. As per Wikipedia a trap flag permits operation of a processor in single-step mode. If such a flag is available, debuggers can use it to step through the execution of a computer program. If it's set, executing an instruction will raise SINGLE_STEP exception. When POP SS (SS is stack segment register) is executed, the CPU will prevent triggering of interrupts, as to avoid corruption of the stack, so when the Single-step flag is set , it triggers and interrupt in the CPU , but when a POP SS is executed it won't trigger interrupts before it has executed the next instruction after it , and thus the debugger will never get a single-step exception for the PUSHFD and won't know it has been executed. Debugger will not break push ss / pop ss and inevitably will stop on the following instruction. In other words, unsetting of the trap flag won't be possible after that, and if check is done here, debugger will be detected. Thus, debugger will not clean out the trap-flag and leave debugger vulnerable to detection.

```
__asm {

    push ss
    pop ss
    pushfd
    cmp byte ptr[esp + 1], 1
    jne being_debugged

}
```

fig.6 push/pop ss

3.1.7 Interrupt 0x2d

The interrupt 0x2D when it is executed, Windows uses the current EIP register value as the exception address, and then it increments by one the EIP register value. However, Windows also examines the value in the EAX register to determine how to adjust the exception address. If the EAX register has the value of 1, 3, or 4 on all versions of Windows, or the value 5 on Windows Vista and later, then Windows will increase by one the exception address. Finally, it issues an EXCEPTION_BREAKPOINT (0x80000003) exception if a debugger is present. The interrupt 0x2D behavior can cause trouble for debuggers. The problem is that some debuggers might use the EIP register value as the address from which to resume, while other debuggers might use the exception address as the address from which to resume. This can result in a single-byte instruction being skipped, or the execution of a completely different instruction because the first byte is missing. These behaviors can be used to infer the presence of the debugger [13].

```
//////////////////////////8.interrupt 0x2d
    __asm {
        xor eax, eax
        int 2dh
        inc eax
        je being_debugged
    }
```

fig.6 interrupt 0x2d

3.1.8 RDTSTC – as anti-step

RDTSC is an IA-32 instruction that stands for Read Time-Stamp Counter. Processors since the Pentium have had a counter attached to the processor that is incremented every clock cycle and reset to 0 when the processor is reset. RDTSC returns the count of the number of ticks since the last system reboot as a 64-bit value placed into EDX:EAX. When a debugger is present, and used to single-step through the code, there is a significant delay between the executions of the individual instructions, when compared to native execution [44]. The idea is to measure the time taken to execute a piece of code and comparing it against a maximum tolerated threshold of time. This delay can be measured using a set of time-related Win API functions or Intel instructions. This set includes the RDPMC instruction (however, this instruction requires that the PCE flag is set in the CR4 register, but this is not the default setting, getting exception PRIV_INSTRUCTION), the RDTSC instruction (however, this instruction requires that the TSD flag is clear in the CR4 register, but this is the default setting), and some Win32 API functions, the kernel32 GetLocalTime() function, the kernel32 GetSystemTime() function, the kernel32 QueryPerformanceCounter() function, the kernel32 GetTickCount() function, the ntoskrnl KiGetTickCount() function (exposed via the interrupt 0x2A interface on the 32-bit versions of Windows), and the winmm timeGetTime() function. The RDMSR instruction can also be used as a time source, but it cannot be used in user mode.

```
//////////////////////////////////RDTSTC
        __asm{
        rdtsc
        xchg esi, eax
        mov edi, edx
        rdtsc
        sub eax, esi
        sbb edx, edi
        jne being_debugged
        cmp eax, 500h
        jnbe being_debugged
        }
```

fig.7 RDTSTC

3.1.9 Selectors

A selector register indicates a specific block of memory from which one can read or write. The real memory address is looked up in an internal CPU table. For certain values FS and GS selectors will be affected by a single-step event. In the case of the GS selector it will be not restored to its default value on the 32-bit versions of Windows if it was set to a value from zero to three [13].

```
/////////Selectors
  __asm {
      push 3
      pop gs
      selectors_loc_1 :
      mov ax, gs
      cmp al, 3
      je selectors_loc_1
      push 3
      pop gs
      mov ax, gs
      cmp al, 3
      jne being_debugged

  }
```

fig.8 Selectors

3.2 API Calls

3.2.1 CheckRemoteDebuggerPresent()

This function is one of the most common and simple API-based anti-debugging tricks since as stated in the documentation it determines whether the specified (in our case the current) process is being debugged by a user-mode debugger. The function sets to 0xffffffff the value to which the pbDebuggerPresent argument points if a debugger is present (that is, attached to the current process). In other words it also checks the BeingDebugged field in the PEB structure to make the decision, so it is similar to IsDebuggerPresent() function except it can be used for a remote process. Internally CheckRemoteDebuggerPresent() calls the NTDLL export NtQueryInformationProcess() with the SYSTEM_INFORMATION_CLASS parameter set to 7 (ProcessDebugPort) [13].

```
mov ebx, fs:[30h] ;PEB
lea ebx, [ebx + 0x02] ;pbDebuggerPresent
push ebx
push 0xffffffff ;Current Process
call eax ;CheckRemoteDebuggerPresent()
cmp ebx, 0
jne being_debugged
```

fig.9 CheckRemoteDebuggerPresent()

3.2.2 NtQueryInformationProcess()

A large number of Windows API functions and structures are considered internal to the operating system and thus not documented well. Many of these functions have undergone extensive research and reverse engineering through the years to be able to understand how they operate and what can be achieved using them. One such half-documented API function is the NtQueryInformationProcess function which is used to retrieve information about a target process and resides in ntdll [45].

i. **ProcessDebugPort**

   The ntdll.NtQueryInformationProcess() function accepts a parameter which is the class of information to query. Most of the classes are not documented. However, one of the documented classes is the ProcessDebugPort (7). It is possible to query for the existence (not the value) of the port. The return value is 0xffffffff if the process is being debugged. Internally, the function queries for the non-zero state of the DebugPort field in the EPROCESS structure.

```
xor ebx, ebx
push ebx
mov ebx, esp
push 0
push 4
push ebx

push 7; ProcessDebugPort
push -1  ;Current Process
call eax ;NtQueryInformationProcess()
pop ebx
inc ebx
cmp ebx, 0
je being_debugged
```

fig.10 /w ProcessDebugPort

ii. Debug Objects – ProcessDebugObjectHandle Class

When a debugging session begins, a debug object is created, and a handle is associated with it. It is possible to query for the value of this handle, using the undocumented ProcessDebugObjectHandle (0x1e) class. Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger [13].

```
//////////////////////NtQueryInformationProcess: w/ ProcessDebugObjectHandle

    xor ebx, ebx
    push ebx
    mov ebx, esp
    push 0
    push 4 ;ProcessInformationLength
    push ebx
    push 1eh; ProcessDebugObjectHandle
    push -1  ;Current Process
    call eax ;NtQueryInformationProcess()
    pop ebx
    test ebx, ebx
    jne being_debugged
```

fig.11 /w ProcessDebugObjectHandle

iii. Debug Objects – ProcessDebugFlags Class

The undocumented ProcessDebugFlags (0x1f) class returns the inverse value of the NoDebugInherit bit in the EPROCESS structure. That is, the return value is zero if a debugger is present.

```
/////////////////NtQueryInformationProcess: w/ ProcessDebugFlags
  xor ebx, ebx
  push ebx
  mov ebx, esp
  push 0
  push 4 ;ProcessInformationLength
  push ebx
  push 1fh; ProcessDebugFlags
  push -1    ;Current Process
  call eax ;NtQueryInformationProcess()
  pop ebx
  test ebx, ebx
  je being_debugged
```

fig.11 /w ProcessDebugFlags

3.2.3 NtSetInformationThread()

According to MSDN, ntdll NtSetInformationThread() sets the priority of a thread. However, its ThreadInformationClass parameter has an undocumented value, ThreadHideFromDebugger (0x11), which prevents debugging events to be sent to the debugger [5]. This is a very powerful anti-debugging technique that can be used to difficult the debugging. When the function is called, the thread will continue to run but a debugger will no longer receive any events related to that thread. It is important to mention that the process will be terminated if it is called on the main thread. The reason why the

function exists is to avoid an unexpected interruption when an external process uses the ntdll RtlQueryProcessDebugInformation() function to query information about the debuggee. The ntdll RtlQueryProcessDebugInformation() function injects a thread into the debuggee in order to gather information about the process. If the injected thread is not hidden from the debugger then the debugger will gain control when the thread starts, and the debuggee will stop executing [13].

```
push 0
push 0
push 11h ;ThreadHideFromDebugger
push -2 ;GetCurrentThread()
call nt_set_information_thread
```

fig.12 NtSetInformationThread

3.2.4 RtlQueryProcessDebugInformation()

The ntdll RtlQueryProcessDebugInformation() function can be used to read certain fields from the process memory of the requested process, including the heap flags. The function does this for the heap flags by calling the ntdll RtlQueryProcessHeapInformation() function internally.

```
xor ebx, ebx
push ebx
push ebx
call rtl_query_create_debug_buffer
push eax
push 14h; PDI_HEAPS + PDI_HEAP_BLOCKS
xchg ebx, eax
push fs : [eax + 20h]; UniqueProcess
call rtl_query_process_debug_information
mov eax, [ebx + 38h]; HeapInformation
mov eax, [eax + 8]; Flags
bswap eax
; not HEAP_SKIP_VALIDATION_CHECKS
and al, 0efh
; GROWABLE
; +TAIL_CHECKING_ENABLED
; +FREE_CHECKING_ENABLED
; +VALIDATE_PARAMETERS_ENABLED
; reversed by bswap
cmp eax, 62000040h
je being_debugged
```

fig.13 RtlQueryProcessDebugInformation

3.2.5 RtlQueryProcessHeapInformation()

The ntdll RtlQueryProcessHeapInformation() function can be used to read the heap flags from the process memory of the current process [19]. This function loads all heap blocks of the process into DebugBuffer according to the informations that we want and that could be specified throughout DebugInfoClassMask.

```
push 0
push 0
call rtl_query_create_debug_buffer
push eax
xchg ebx, eax
call rtl_query_process_heap_information
mov eax, [ebx + 38h]; HeapInformation
mov eax, [eax + 8]; Flags
bswap eax
; not HEAP_SKIP_VALIDATION_CHECKS
and al, 0efh
; GROWABLE
; +TAIL_CHECKING_ENABLED
; +FREE_CHECKING_ENABLED
; +VALIDATE_PARAMETERS_ENABLED
cmp eax, 62000040h
je being_debugged
```

fig.14 RtlQueryProcessHeapInformation

3.2.6 Self-debugging with CreateProcess()

Self-Debugging It does not mean a single process debugging itself, because that is not possible. Instead is a technique where the main process spawns a child process that debugs the process that created the child process a.k.a the parent process. It can prevent other debuggers from attaching to the target process since only one debugger can be attached to a process at a time, the second process becomes "undebuggable" by ordinary means. The first process does not even need to do anything debugger-related. There many variations of this technique as well as many advanced approaches [20].

```
__asm {
    //////////////Selfdebug
    xor ebx, ebx
    push start_up_info_struct

    call get_startup_info_A
    call get_command_line_A
    push  process_information_struct
    push start_up_info_struct
    push ebx
    push ebx
    push 1; DEBUG_PROCESS
    push ebx
    push ebx
    push ebx
    push eax
    push ebx
    call create_process_A
    mov ebx, debug_event_struct
    jmp create_process_loc2
    create_process_loc1 :
    push 10002h; DBG_CONTINUE
        push dword ptr[esi + 0ch]; dwThreadId
        push dword ptr[esi + 8]; dwProcessId
        call continue_debug_event
        create_process_loc2 :
    push - 1; INFINITE
        push ebx
        call wait_for_debug_event
        cmp byte ptr[ebx], 5
        ; EXIT_PROCESS_DEBUG_EVENT
        jne create_process_loc1


        create_process_loc3 : // sizeof(STARTUPINFO)//44h//db 44h dup(? )
    call create_process_loc3_back

        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)


        create_process_loc4:
    call create_process_loc4_back
        // sizeof(PROCESS_INFORMATION)
        //l4: db 10h dup(? )
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        create_process_loc5 :
        call create_process_loc5_back
        // sizeof(DEBUG_EVENT)
        //l5: db 60h dup(? )
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
        bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
```

fig.15 Self-Debug /w CreateProcess

3.2.7 SwitchDesktop()

It is possible to select a different active desktop through functions of WinAPI, which has the effect of hiding the windows of the previously active desktop, and with no obvious way to switch back to the old desktop (except by ctrl+alt+delete and killing the process from task manager). Further, the mouse and

keyboard events from the debuggee's desktop will not be delivered anymore to the debugger, because their source is no longer shared. This obviously makes debugging impossible [13]. Using CreateDesktopA and SwitchDesktop functions exported by user32.dll we can use this trick to crush the debugging session or the execution of a malware.

```
    xor eax, eax
    push eax
    ; DESKTOP_CREATEWINDOW
    ; +DESKTOP_WRITEOBJECTS
    ; +DESKTOP_SWITCHDESKTOP
    push 182h
    push eax
    push eax
    push eax
    mov eax, esi
    jmp switch_desktop_loc1
    switch_desktop_loc1_back :
    pop esi
    push esi

    call eax ;CreateDesktopA
    push eax
    mov eax, edi
    call eax ;SwitchDesktop



switch_desktop_loc1 :
call switch_desktop_loc1_back

bb('m') bb('y') bb('d') bb('e') bb('s')  bb('k')  bb('t')  bb('t')  bb('o')  bb('p')  bb(0x00)
```

fig.16 SwitchDesktop

3.2.8 OutputDebugString()

The OutputDebugString technique works by determining if OutputDebugString causes an error. It sends a string to the debugger for display. If OutputDebugString is called and there is a debugger attached, the call to OutputDebugString should succeed, and the value in GetLastError should not be changed. An error will only occur if there is no active debugger for the process to receive the string; therefore, we can conclude that if there is no error by calling GetLastError, after calling OutputDebugString, then there is a debugger present.

```
xor ebx, ebx
mov fs : [ebx + 34h], ebx; LastErrorValue
push ebx
push esp
call output_debug_string_A
cmp fs : [ebx + 34h], ebx; LastErrorValue
je being_debugged
```

fig.17 OutputDebugStringA

3.2.9 NtQueryObject

The NtQueryObject function, when called with the ObjectAllTypesInformation class, will return information about the host system and the current process. There is a wealth of information to be mined from this function, but we're most concerned with the information given about the DebugObjects in the environment. When a debugging session begins, a debug object is created, and a handle is associated with it. Using the ntdll NtQueryObject() function, it is possible to query for the list of existing objects, and check the number of handles associated with any debug object that exists.
A DebugObject entry is maintained in this list of objects, and most importantly, the number of objects of each type of object. The object and its related information can be expressed as

a OBJECT_INFORMATION_TYPE struct. However, calling the NtQueryObject function with the ObjectAllTypesInformation class actually returns a buffer that begins with a OBJECT_TYPE_INFORMATION struct [21]. Since this information comes from the kernel, there is no easy way for user-mode code to prevent this call from revealing the presence of the debugger [13].

```
__asm {

    xor ebx, ebx
    xor edx, edx
    jmp nt_query_object_loc2

    nt_query_object_loc1 :
    push 8000h; MEM_RELEASE
    push edx
    push esi
    call virtual_free

    nt_query_object_loc2:
    xor eax, eax
    mov ah, 10h; MEM_COMMIT
    add ebx, eax; 4kb increments
    push 4; PAGE_READWRITE
    push eax
    push ebx
    push edx
    call virtual_alloc
    push edx
    ; must calculate by brute - force
    push ebx
    push eax
    push 3; ObjectAllTypesInformation
    push edx
    xchg esi, eax
    call ntquery_object
    ; presumably STATUS_INFO_LENGTH_MISMATCH
    test eax, eax
    jl nt_query_object_loc1

    lodsd ;handle count
    xchg ecx, eax
    nt_query_object_loc3 :
    lodsd; string lengths
    movzx edx, ax; length
    lodsd; pointer to TypeName
    xchg esi, eax
    ; sizeof(L"DebugObject")
    ; avoids superstrings
    ; like "DebugObjective"
    cmp edx, 16h
    jne nt_query_object_loc4
    xchg ecx, edx
    mov edi, dword ptr[str_debugobject]

    repe cmpsb
    xchg ecx, edx
    jne nt_query_object_loc4
    cmp [eax+4], edx ;TotalNumberOfObjects
    jne being_debugged
    jmp continue_execution
    nt_query_object_loc4:
    lea esi, [esi + edx + 4]; skip null and align
    and esi, -4; round down to dword
    loop nt_query_object_loc3
    }


__asm {
nt_query_object_loc5:
call nt_query_object_loc5_back
bb('D')  bb(0) bb('e')  bb(0) bb('b')  bb(0) bb('u')  bb(0) bb(
```

fig.18 NtQueryObject

3.2.10 BlockInput

BlockInput as stated in msdn documentation [39] blocks keyboard and mouse input events from reaching applications (apart from the ctrl-alt-delete key sequence). This technique is effective due to the fact that only the thread that the thread that blocked input can successfully unblock input. This isn't really an anti-reverse-engineering technique, but more of a way to mess with with the debugging session. The effect remains until either the process exits, or the function is called again with the opposite parameter. It is a very effective way to disable debuggers. The call requires that the calling thread has the DESKTOP_JOURNALPLAYBACK (0x0020) privilege (which is set by default). On Windows Vista and later, it also requires that the process is running at a high integrity level (that is, a process requires elevation if it was running in a standard or lowrights user account), if elevation is enabled and either the "HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\System\EnableUIPI" registry value either does not exist (a default value of present and set is used in that case) or has a non-zero value (and which would require administrative privileges to change). The function will not allow the input to be blocked twice in a row, nor will it allow the input to be unblocked twice in a row. Thus, if the same request is made twice to the function, then the return value should be different. This fact can be used to detect the presence of a number of tools that intercept the call, because most of them simply return success, regardless of the input [13].

```
mov eax, [esp + 11]
push ebx; BlockInput
push user32_base; base address of user32.dll retrieved by LoadLibraryA
call get_proc_address; GetProcAddress address : ))
add esp, 11
    mov block_input, eax
    push 1
    call block_input
    xchg ebx, eax
    push 1
    call block_input
    xor ebx, eax
    je being_debugged
```

fig.19 BlockInput

4 Anti-VM Techniques

To extract malware intelligence, the most common use is to execute malwares inside a virtualized execution environment and try to comprehend how it behaves by observing for example all the system function calls. Once the analysis is complete, the environment can be destroyed, essentially without risk to the real environment that hosts it [15]. This tactic can implement a fast and secure way to examine malicious samples. For that reason, the most usual functionality a malware can add on its weaponry is to try to detect a virtual machine/sandbox. In case of detection of virtualization malicious samples usually adjust their behavior, usually by refusing to execute. This behavior makes analysis more complex, and possibly highly evasive. It is worth mentioning that anti-vm techniques are a rising trend in malware development

For the ease to manage and communicate to VMM, most of VM systems leave some artifacts in guest OS. Those artifacts make detection of VM possible. The artifacts include processes, registry keys and values, loaded and exported dll's, network artifacts like for example specific MAC addresses or adapter name, file system artifacts (system32\vboxtray.exe), special directories that are created (%PROGRAMFILES%\VMWare), virtual devices (\\.\HGFS), hardware label etc. The following are

techniques that have been implemented in the tool and in no case they form a complete list of the numerous anti-vm techniques that exist. (See List of Techniques at the end).

4.1 Red Pill

Maybe on of the oldest techniques introduced by Joanna Rutkowska, in 2004, is a technique known as Red Pill. This technique uses a single machine language instruction, SIDT ("Store Interrupt Descriptor Table") that can be run in user mode, and its purpose is to retrieve the location of the Interrupt Descriptor Table Register (IDTR) and store it in memory. Rutkowska observed that on VM guest machines, the IDT is typically located at 0xffXXXXXX, while on VirtualPC guests, it is located at 0xe8XXXXXX. For host operating systems, the IDT is located far lower in memory [46]. To handle both conditions, one checks to see if the IDTR is greater than 0xd0000000. Variations of this technique use other instructions notably the sgdt for Global Descriptor Table (GDT) and the sldt for Local Descriptor Table (LDT). These tables hold critical variables associated with the operating system and particular running processes, respectively. One can also use the str instruction that stores the segment selector from the Task Register (TR) [16]. However, these techniques don't work anymore on VMware 12 Workstation after experiments (they have not been tested in other virtualization software).

```
__asm {
        jmp sldt_loc1
        sldt_loc1_back:
        pop esi
        mov eax, [esi]
        sldt [esi]
        mov eax, [esi]
        cmp eax, 0xdead0000
        jne switch_desktop_1
        jmp continue_execution

    sldt_loc1:
    call sldt_loc1_back

    bb(0x37) bb(0x13) bb(0xad) bb(0xde) bb(0x90)
    }
```

fig.20 /w sldt instruction

```
__asm {
        jmp str_loc1
        str_loc1_back:
        pop esi
        mov eax, [esi]
        str [esi]

        cmp [esi], 0x40
        and [esi + 1], 0x00

        je switch_desktop_1
        jmp continue_execution

    str_loc1:
    call str_loc1_back

    bb(0x90) bb(0x90) bb(0x90) bb(0x90)
    }
```

fig.21 /w str instruction

4.2 CPUID – Hypervisor presence

The CPUID opcode is a processor supplementary instruction (its name derived from CPU IDentification) for the x86 architecture allowing software to discover details of the processor [38]. CPUID instruction is executed with EAX=1 as input, the return value describes the processors features. The 31st bit of ECX on a physical machine will be equal to 0. On a guest VM it will equal to 1.

```
///////////CPUID1
//This instruction is executed with EAX=1 as input, the return value describes the processors features.
//The 31st bit of ECX on a physical machine will be equal to 0. On a guest VM it will equal to 1.
__asm {
    xor eax, eax
    inc eax
    cpuid
    bt ecx, 0x1f
    jb switch_desktop_1
    jmp continue_execution
}
```

fig.22 cpuid 1

4.3 CPUID – Hypervisor Vendor

By calling CPUID with EAX=40000000 as input,1 the malware will get, as the return value, the virtualization vendor string in EAX, ECX, EDX.
For example:

- Microsoft: “Microsoft HV”
- VMware: “VMwareVMware”

```
////////////CPUID2
    //Hypervisor brand”: by calling CPUID with EAX=40000000 as input,1 the malware will get, as the return value,
    //the virtualization vendor string in EBX, ECX, EDX.
    __asm {
        xor eax, eax
        mov eax, 0x40000000
        cpuid
        cmp ebx, 0x61774d56 ////////////56 6d 57 61 Vmwa
        je switch_desktop_1
        jmp continue_execution
    }
```

fig.23 cpuid 2

4.4 Number of Processors

Malware authors use various techniques to identify a sandbox, but most methods involve analyzing the host’s hardware configuration. One of these techniques is the number of processors assigned to a vm. Malware can compare that value to detect virtualization. We can retrieve the number of processors using the kernel32!GetSystemInfo function

```
jmp get_system_info_loc1
get_system_info_loc1_back:
pop esi
push esi
call get_system_info
mov edx, [esi + 14h]; SYSTEM_INFO.dwNumberOfProcessors
cmp edx, 1
jbe switch_desktop_1
jmp continue_execution
get_system_info_loc1 : // sizeof(SYSTEM_INFO) db 24h dup(? )
call get_system_info_loc1_back

    bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
    bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
    bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
    bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90) bb(0x90)
    }
```

fig.24 checking number of processors

4.5 Virtual Devices

One way to detect if virtualization is active in the system is to try accessing their device drivers. The technique is simple and just involves calling kernel32!CreateFile() against well-known device names used by virtualization software. For example, trying to access \\.\HGFS or \\.\vmci when in host will return INVALID_HANDLE_VALUE (0xffffffff) in eax, which is different than the value returned inside VMWare.

```
xor eax, eax
push eax
push eax
push 3; OPEN_EXISTING
push eax
push 0x00000004; FILE_SHARE_READ
push 0x80000000; GENERIC_READ
jmp device_drivers_loc1
device_drivers_loc1_back:
pop esi
push esi
call create_file_A
cmp eax, -1; INVALID_HANDLE_VALUE
jne switch_desktop_1
jmp continue_execution

device_drivers_loc1 :
call device_drivers_loc1_back

bb(0x5c) bb(0x5c) bb(0x2e) bb(0x5c) bb(0x48) bb(0x47) bb(0x46) bb(0x53) bb(0x00)
        ;\\.\HGFS
```

fig.25 Virtual Device HGFS

5 Conclusion

In malware analysis field as well as in software protection the number of anti-reverse engineering techniques, others simple others advanced (e.g. TLB splitting), is probably a couple of hundreds. The main purpose is to manipulate the results of tools in order to fool and as result prevent or delay the analyst from dissecting and neutralizing the malware. The tool presented automates the implementation of known selected anti-debug and anti-vm techniques in order to armor a simple malware on the fly. In no way it acts as a targeted sophisticated solution. Because many of the techniques incorporated into the tool can be neutralized automatically either with anti-anti plugins or manually with hooking of Windows API functions, as a future work, the tool will implement an anti-hook engine.

References

[1] Practical Malware Analysis The Hands-On Guide to Dissecting Malicious Software by Michael Sikorski and Andrew Honig February 2012

[2] Chen P., Huygens C., Desmet L., Joosen W. (2016) Advanced or Not? A Comparative Study of the Use of Anti-debugging and Anti-VM Techniques in Generic and Targeted Malware. In: Hoepman JH., Katzenbeisser S. (eds) ICT Systems Security and Privacy Protection. SEC 2016. IFIP Advances in Information and Communication Technology, vol 471. Springer, Cham

[3] Towards Transparent Debugging Fengwei Zhang , Kevin Leach, Angelos Stavrou, and Haining Wang IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, VOL. 15, NO. 2, MARCH/APRIL 2018

[4] Peering Inside the PE: A Tour of the Win32 Portable Executable File Format

[5] http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[6] https://web.archive.org/web/20120121050913/http://www.phreedom.org:80/solar/code/tinype/

[7] https://media.blackhat.com/bh-us-11/Vuksan/BH_US_11_VuksanPericin_PECOFF_WP.pdf

[8] https://ntquery.wordpress.com/2014/03/29/anti-debug-ntsetinformationthread/

[9] http://blog.rewolf.pl/blog/?p=573

[10] Anti-Debugging – A Developers View

[11] X. Chen, J. Andersen, Z. Mao, et al. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In DSN, 2008.

[12] R. R. Branco, G. N. Barbosa, and P. D. Neto. (2012). "Scientific but not academical overview of malware anti-debugging, anti-disassembly and Anti-VM technologies,"

[13] Peter Ferrie – The "Ultimate" Anti-Debugging Reference

[14] The Art of Unpacking Mark Vincent Yason Malcode Analyst, X-Force Research & Development

[15] Attacks on Virtual Machine Emulators Peter Ferrie, Senior Principal Researcher, Symantec Advanced Threat Research

[16] https://sites.google.com/site/bletchleypark2/malware-analysis/malware-technique/anti-vm

[17] N. Falliere. (2010). Windows anti-debug reference [

[18] Intel - Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 2B: Instruction Set Reference

[18] Peter Ferrie – Anti-Unpacker Tricks

[19] https://evilcodecave.wordpress.com/2009/04/

[20] Tightly-Coupled Self-Debugging Software Protection

[21] https://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide

[22] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in Proc. 15th ACM Conf. Comput. Commun. Security, 2008, pp. 51–62.

[23] Z. Deng, X. Zhang, and D. Xu, "SPIDER: Stealthy binary program instrumentation and debugging via hardware virtualization," in Proc. Annu. Comput. Security Appl. Conf., 2013, pp. 289–298.

[24] A. Fattori, R. Paleari, L. Martignoni, and M. Monga, "Dynamic and transparent analysis of commodity production systems," in Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng., 2010, pp. 417–426.

[25] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin. (2012). V2E:Combining hardware virtualization and software emulation for transparent and extensible malware analysis. in Proc. 8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environ.

[26] (2009). Anubis. Analyzing Unknown Binaries

[27] N. A. Quynh and K. Suzaki. (2010). "Virt-ICE: Next-generation debugger for malware analysis," in Black Hat USA.

[28] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of Anti-Virtualization and AntiDebugging behavior in modern malware," in Proc. 38th Annu. IEEE Int. Conf. Dependable Syst. Netw., 2008, pp. 177–186.

[31] D. Quist and V. Val Smith. (2006). Detecting the presence of virtual machines using the local data table
[32] E. Bachaalany. (2005). Detect if your program is running inside a virtual machine
[33] T. Raffetseder, C. Kruegel, and E. Kirda, “Detecting system emulators,” in Information Security. Berlin, Germany: Springer, 2007.
[34] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, “Compatibility is not transparency: VMM detection myths and realities,” in Proc.11th USENIX Workshop Hot Topics Operating Syst., 2007. pp. 1–6.
[35] D. Kirat, G. Vigna, and C. Kruegel, “BareBox: Efficient malware analysis on Bare-metal,” in Proc. 27th Annu. Comput. Security Appl.Conf., 2011, pp. 403–412.
[36] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, “Down to the bare Metal: Using processor features for binary analysis,” in Proc. Annu. Comput. Security Appl. Conf., 2012, pp. 189–198.
[37] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, “Detecting Environment-Sensitive Malware,” in Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID), 2011.
[38] https://en.wikipedia.org/wiki/CPUID
[39] https://msdn.microsoft.com/en-us/library/windows/desktop/ms646290(v=vs.85).aspx
[40] Hiding Debuggers from Malware with Apate
[41] VMDE Virtual Machines Detection Enhanced N. Rin EP_X0FF
[42] https://blog.kowalczyk.info/articles/pefileformat.html
[43] https://github.com/deptofdefense/SalSA/wiki/PE-File-Format
[44] Anti-unpacker tricks – part two 2009-01-01 Peter Ferrie
[45] https://www.veracode.com/blog/2009/01/anti-debugging-series-part-iii
[46] On the Cutting Edge: Thwarting Virtual Machine Detection Tom Liston / Ed Skoudis, SANS

List of techniques

What follows is a list of techniques to thwart malware analysis of PE executable found in the wild.

Anti-debugging

- IsDebuggerPresent
- CheckRemoteDebuggerPresent
- Process Environement Block (BeingDebugged)
- Process Environement Block (NtGlobalFlag)
- ProcessHeap (Flags)
- ProcessHeap (ForceFlags)
- NtQueryInformationProcess (ProcessDebugPort)
- NtQueryInformationProcess (ProcessDebugFlags)
- NtQueryInformationProcess (ProcessDebugObject)
- NtSetInformationThread (HideThreadFromDebugger)
- NtQueryObject (ObjectTypeInformation)
- NtQueryObject (ObjectAllTypesInformation)
- CloseHanlde (NtClose) Invalide Handle
- SetHandleInformation (Protected Handle)
- UnhandledExceptionFilter
- OutputDebugString (GetLastError())
- Hardware Breakpoints (SEH / GetThreadContext)
- Software Breakpoints (INT3 / 0xCC)
- Memory Breakpoints (PAGE_GUARD)
- Interrupt 0x2d
- Interrupt 1
- Parent Process (Explorer.exe)
- SeDebugPrivilege (Csrss.exe)
- NtYieldExecution / SwitchToThread
- TLS callbacks
- Process jobs
- Memory write watching

Anti-Dumping

- Erase PE header from memory
- SizeOfImage

Timing Attacks [Anti-Sandbox]

- RDTSC (with CPUID to force a VM Exit)
- RDTSC (Locky version with GetProcessHeap & CloseHandle)
- Sleep -> SleepEx -> NtDelayExecution
- Sleep (in a loop a small delay)
- Sleep and check if time was accelerated (GetTickCount)
- SetTimer (Standard Windows Timers)
- timeSetEvent (Multimedia Timers)
- WaitForSingleObject -> WaitForSingleObjectEx -> NtWaitForSingleObject
- WaitForMultipleObjects -> WaitForMultipleObjectsEx -> NtWaitForMultipleObjects (todo)
- IcmpSendEcho (CCleaner Malware)
- CreateWaitableTimer (todo)

- CreateTimerQueueTimer (todo)
- Big crypto loops (todo)

Human Interaction / Generic [Anti-Sandbox]

- Mouse movement
- Total Physical memory (GlobalMemoryStatusEx)
- Disk size using DeviceIoControl (IOCTL_DISK_GET_LENGTH_INFO)
- Disk size using GetDiskFreeSpaceEx (TotalNumberOfBytes)
- Mouse (Single click / Double click) (todo)
- DialogBox (todo)
- Scrolling (todo)
- Execution after reboot (todo)
- Count of processors (Win32/Tinba — Win32/Dyre)
- Sandbox known product IDs (todo)
- Color of background pixel (todo)
- Keyboard layout (Win32/Banload) (todo)

Anti-Virtualization / Full-System Emulation

- Registry key value artifacts
    - HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0 (Identifier) (VBOX)
    - HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0 (Identifier) (QEMU)
    - HARDWARE\Description\System (SystemBiosVersion) (VBOX)
    - HARDWARE\Description\System (SystemBiosVersion) (QEMU)
    - HARDWARE\Description\System (VideoBiosVersion) (VIRTUALBOX)
    - HARDWARE\Description\System (SystemBiosDate) (06/23/99)
    - HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0 (Identifier) (VMWARE)
    - HARDWARE\DEVICEMAP\Scsi\Scsi Port 1\Scsi Bus 0\Target Id 0\Logical Unit Id 0 (Identifier) (VMWARE)
    - HARDWARE\DEVICEMAP\Scsi\Scsi Port 2\Scsi Bus 0\Target Id 0\Logical Unit Id 0 (Identifier) (VMWARE)
    - SYSTEM\ControlSet001\Control\SystemInformation (SystemManufacturer) (VMWARE)
    - SYSTEM\ControlSet001\Control\SystemInformation (SystemProductName) (VMWARE)
- Registry Keys artifacts
    - HARDWARE\ACPI\DSDT\VBOX__ (VBOX)
    - HARDWARE\ACPI\FADT\VBOX__ (VBOX)
    - HARDWARE\ACPI\RSDT\VBOX__ (VBOX)
    - SOFTWARE\Oracle\VirtualBox Guest Additions (VBOX)
    - SYSTEM\ControlSet001\Services\VBoxGuest (VBOX)
    - SYSTEM\ControlSet001\Services\VBoxMouse (VBOX)
    - SYSTEM\ControlSet001\Services\VBoxService (VBOX)
    - SYSTEM\ControlSet001\Services\VBoxSF (VBOX)
    - SYSTEM\ControlSet001\Services\VBoxVideo (VBOX)
    - SOFTWARE\VMware, Inc.\VMware Tools (VMWARE)
    - SOFTWARE\Wine (WINE)
    - SOFTWARE\Microsoft\Virtual Machine\Guest\Parameters (HYPER-V)

- File system artifacts
    - «system32\drivers\VBoxMouse.sys»
    - «system32\drivers\VBoxGuest.sys»
    - «system32\drivers\VBoxSF.sys»
    - «system32\drivers\VBoxVideo.sys»
    - «system32\vboxdisp.dll»
    - «system32\vboxhook.dll»
    - «system32\vboxmrxnp.dll»
    - «system32\vboxogl.dll»
    - «system32\vboxoglarrayspu.dll»
    - «system32\vboxoglcrutil.dll»
    - «system32\vboxoglerrorspu.dll»
    - «system32\vboxoglfeedbackspu.dll»
    - «system32\vboxoglpackspu.dll»
    - «system32\vboxoglpassthroughspu.dll»
    - «system32\vboxservice.exe»
    - «system32\vboxtray.exe»
    - «system32\VBoxControl.exe»
    - «system32\drivers\vmmouse.sys»
    - «system32\drivers\vmhgfs.sys»
    - «system32\drivers\vm3dmp.sys»
    - «system32\drivers\vmci.sys»
    - «system32\drivers\vmhgfs.sys»
    - «system32\drivers\vmmemctl.sys»
    - «system32\drivers\vmmouse.sys»
    - «system32\drivers\vmrawdsk.sys»
    - «system32\drivers\vmusbmouse.sys»
- Directories artifacts
    - «%PROGRAMFILES%\oracle\virtualbox guest additions\»
    - «%PROGRAMFILES%\VMWare\»
- Memory artifacts
    - Interupt Descriptor Table (IDT) location
    - Local Descriptor Table (LDT) location
    - Global Descriptor Table (GDT) location
    - Task state segment trick with STR
- MAC Address
    - «\x08\x00\x27» (VBOX)
    - «\x00\x05\x69» (VMWARE)
    - «\x00\x0C\x29» (VMWARE)
    - «\x00\x1C\x14» (VMWARE)
    - «\x00\x50\x56» (VMWARE)
    - «\x00\x1C\x42» (Parallels)
    - «\x00\x16\x3E» (Xen)
- Virtual devices
    - «\\.\VBoxMiniRdrDN»

    - «\\.\VBoxGuest»
    - «\\.\pipe\VBoxMiniRdDN»
    - «\\.\VBoxTrayIPC»
    - «\\.\pipe\VBoxTrayIPC»)
    - «\\.\HGFS»
    - «\\.\vmci»
- Hardware Device information
    - SetupAPI SetupDiEnumDeviceInfo (GUID_DEVCLASS_DISKDRIVE)
        - QEMU
        - VMWare
        - VBOX
        - VIRTUAL HD
- System Firmware Tables
    - SMBIOS string checks (VirtualBox)
    - SMBIOS string checks (VMWare)
    - SMBIOS string checks (Qemu)
    - ACPI string checks (VirtualBox)
    - ACPI string checks (VMWare)
    - ACPI string checks (Qemu)
- Driver Services
    - VirtualBox
    - VMWare
- Adapter name
    - VMWare
- Windows Class
    - VBoxTrayToolWndClass
    - VBoxTrayToolWnd
- Network shares
    - VirtualBox Shared Folders
- Processes
    - vboxservice.exe (VBOX)
    - vboxtray.exe (VBOX)
    - vmtoolsd.exe(VMWARE)
    - vmwaretray.exe(VMWARE)
    - vmwareuser(VMWARE)
    - VGAuthService.exe (VMWARE)
    - vmacthlp.exe (VMWARE)
    - vmsrvc.exe(VirtualPC)
    - vmusrvc.exe(VirtualPC)
    - prl_cc.exe(Parallels)
    - prl_tools.exe(Parallels)
    - xenservice.exe(Citrix Xen)
    - qemu-ga.exe (QEMU)
- WMI
    - SELECT * FROM Win32_Bios (SerialNumber) (GENERIC)

    - SELECT * FROM Win32_PnPEntity (DeviceId) (VBOX)
    - SELECT * FROM Win32_NetworkAdapterConfiguration (MACAddress) (VBOX)
    - SELECT * FROM Win32_NTEventlogFile (VBOX)
    - SELECT * FROM Win32_Processor (NumberOfCores) (GENERIC)
    - SELECT * FROM Win32_LogicalDisk (Size) (GENERIC)
    - SELECT * FROM Win32_Computer (Model and Manufacturer) (GENERIC)
    - SELECT * FROM MSAcpi_ThermalZoneTemperature CurrentTemperature) (GENERIC)
- DLL Exports and Loaded DLLs
    - avghookx.dll (AVG)
    - avghooka.dll (AVG)
    - snxhk.dll (Avast)
    - kernel32.dll!wine_get_unix_file_nameWine (Wine)
    - sbiedll.dll (Sandboxie)
    - dbghelp.dll (MS debugging support routines)
    - api_log.dll (iDefense Labs)
    - dir_watch.dll (iDefense Labs)
    - pstorec.dll (SunBelt Sandbox)
    - vmcheck.dll (Virtual PC)
    - wpespy.dll (WPE Pro)
- CPU
    - Hypervisor presence using (EAX=0x1)
    - Hypervisor vendor using (EAX=0x40000000)
        - «KVMKVMKVM\0\0\0» (KVM)
            - «Microsoft Hv»(Microsoft Hyper-V or Windows Virtual PC)
            - «VMwareVMware»(VMware)
            - «XenVMMXenVMM»(Xen)
            - «prl hyperv «(Parallels) -«VBoxVBoxVBox»(VirtualBox)

Anti-Analysis

- Processes
    - OllyDBG / ImmunityDebugger / WinDbg / IDA Pro
    - SysInternals Suite Tools (Process Explorer / Process Monitor / Regmon / Filemon, TCPView, Autoruns)
    - Wireshark / Dumpcap
    - ProcessHacker / SysAnalyzer / HookExplorer / SysInspector
    - ImportREC / PETools / LordPE
    - JoeBox Sandbox