Resurrecting Anti-virtualization and Anti-debugging: Unhooking your Hooks

Theodoros Apostolopoulos[a], Vasilios Katos[b], Kim-Kwang Raymond Choo[c], Constantinos Patsakis[a,d,]

[a] *University Piraeus, 80 Karaoli & Dimitriou str, 18534 Piraeus, Greece*
[b] *Bournemouth University, Poole House P323, Talbot Campus, Fern Barrow, Poole, Dorset, BH12 5BB, UK*
[c] *University of Texas San Antonio, 1 UTSA Cir, San Antonio, Texas 78249, USA*
[d] *Athena Research Center, Artemidos 6, Marousi 15125, Greece*

Abstract

Dynamic malware analysis involves the debugging of the associated binary files and the monitoring of changes in sandboxed environments. This allows the investigator to manipulate the code execution path and environment to develop an understanding of the malware's internal workings, aims and modus operandi. However, the latest state of the art malware may incorporate anti-virtual environment (VM) and anti-debugging countermeasures (i.e. to determine whether the malware is being executed in a VM or using a debugger prior to payload execution). We argue that for the malware to be effective, it will need to support an array of anti-detection and evasion mechanisms. In essence, from the malware's perspective, it needs to adopt a "defence in depth" paradigm to achieve its underlying business logic functionality. Beyond the malicious uses, software vendors to preserve the intellectual property rights of their products often resort to similar methods to deter competitors from gaining intelligence from the binaries or prevent customers from using their products in unauthorised hardware.

In this work, we illustrate how Windows architecture impedes the work of debuggers when they analyse with armoured binaries. The debugger and the malware have the same privileges, so the attacker may manipulate the

Email addresses: organ6667@gmail.com (Theodoros Apostolopoulos), vkatos@bournemouth.ac.uk (Vasilios Katos), raymond.choo@fulbrightmail.org (Kim-Kwang Raymond Choo), kpatsak@unipi.gr (Constantinos Patsakis)

address space that the debugger operates and, e.g. bypass detection. We showcase this by presenting a new framework (ANTI), which automates the procedure of integrating anti-debugging and anti-VM in the binary. Specifically, ANTI introduces an anti-hooking method targeting Windows binaries, where hooks applied by state of the art debuggers are removed and injects its code in other processes. This significantly compounds the challenge of binary analysis. Our extensive evaluation also demonstrates that ANTI successfully circumvents detection from state-of-the-art detection methods. Therefore, ANTI illustrates that current tools for dynamic analysis have serious implementation gaps that allow for binaries to bypass them. More alarmingly, ANTI shows how one can use well-known methods to "resurrect" old attacks.

Keywords: Malware, Windows hooking, dynamic analysis, anti-debugging, anti-virtualization

---

1. Introduction

Malicious application (malware) underpins many criminal activities, particularly financially-motivated criminal activities such as ransomware as well as advanced persistent threats (APTs). In recent times, the number and sophistication of malicious application (malware) are increasing significantly, and the cybercrime economy was estimated to be worth 1.5 trillion dollars [1].

Similarly, malware analysis is continuously evolving to mitigate detection techniques employed by malware designers/authors. For example, the use of obfuscators and packers is currently a norm in malware development [2, 3, 4, 5, 6], and this compounds the challenge of static analysis. An alternative approach is dynamic analysis, where malware is executed in a monitored and controlled environment (sandboxing) or in debug mode. The latter provides the capabilities to step into the code while it is being executed, dump the memory, or even alter the execution of the process. From a software engineering perspective, the malware can be considered to champion best practices for robust system development, as it is challenging to operate in adverse conditions and "hostile" environments.

This paper focuses on establishing the extent and capabilities of dynamic analysis evasion and particularly on how malware can render a debugger incapable of tracing the malware's execution flows, by presenting a proof of

concept to release the debugger hook attempts. Simultaneously, the same method can be used for copyright protection as it can be used to prevent competitors from reversing a binary or allow customers to execute a binary on unauthorised devices. Nevertheless, we argue that the implications for malware detection are more severe; therefore, we will mostly focus on this aspect.

Most existing monitoring systems, security solutions or malware analysis tools rely heavily on the modification of user-mode code in memory to manipulate code execution. This is typically performed by the interception of function calls, known as API hooking. The latter has a two-faceted usage, benign and malicious. For instance, if hooking is performed by a malware analyst, it would allow the analyst to study several features and aspects of the software (e.g. What are the API calls performed? What resources are requested and used?). API hooking can also be used by an adversary to create a keylogger by hooking the keyboard and intercept all keyboard events across all applications.

Virtual machines (VMs), containers, and sandboxes can also be used to facilitate malware analysis, for example by allowing analysts to execute a collected binary in an isolated environment that is easy to set up, replicate and then dispose of it.

To avoid detection, execution environment awareness is often built into malware. In doing so, such malware can determine the presence of hooks, etc., and adapt and possibly completely change their execution on-the-fly, thus preventing their real-time, dynamic analysis. This can be performed by monitoring "environmental artifacts" [7]. These environmental artifacts range from unique characteristics of the environment they are executed in (e.g. hardware identifiers, uptime, usernames, and number of CPUs) to the identification of sensor changes (e.g. accelerometers), and from timing discrepancies to the existence of "pills" [8, 9]. In other words, techniques such as anti-debug and anti-VM can be abused by malware to bypass dynamic malware analysis and detection [10, 3, 11].

1.1. Main contributions

Due to the large and increasing number of collected malware samples, automated analysis of binaries is a common practice. Moreover, the use of evasion methods from malware is a common practice. Therefore, to determine whether current security measures can counter such methods efficiently,

we need to assess whether they are detected and mitigated by state of the art malware analysis tools.

In this paper, we develop a tool (hereafter referred to as ANTI), which implements anti-debugging, anti-VM and process injection methods. ANTI automates the addition of this functionality in existing executables and combines it with a powerful unhooking technique that can bypass debugger hiding tools. Specifically, we use known anti-debug and anti-VM techniques that are based on Windows API functions and reinforce them via user-land unhooking functionality and process migration. Even though these methods are well documented in the literature, we demonstrate that existing tools for dynamic analysis fail to detect or prevent our approach. This deficiency can be attributed to two key factors, the Windows architecture and the lack of provisions for auxiliary entry points from the debuggers. The former forces the analyst and the malware to "fight" with the same privileges, which clearly facilitates the adversary, as she has the time to study the defence strategies and mechanisms to devise one that bypasses or even disables them.

The **practical implications** of our research is that given the reliance on automated dynamic analysis in such tools, many malware in the wild can potentially circumvent these existing security mechanisms by using similar methods and avoid detection. Our approach could also be used by software vendors to safeguard their intellectual property rights. As highlighted in the existing literature [12, 13], emulation techniques have been exploited to infringe Intellectual Property (IP) rights. Hence, by using ANTI software vendors may prevent the analysis of their binaries from a competitor or prevent customers from executing the binary in unauthorised devices or emulated proprietary hardware (also proposed by Jang et al. [14]). In fact, the latter method is used by Samsung as a security enhancement in its mobile devices.

1.2. Organization of this work

The remainder of this paper is organized as follows. The next section reviews existing literature, focusing on anti-debug and anti-VM approaches and their countermeasures. We also discuss several Windows-specific internal workings that are necessary to explain how ANTI works. Then, we present our adversary model and in Section 4, we present ANTI, its scope, its architecture, features and the underpinning methods. Section 5 presents the evaluation of ANTI, in terms of its effectiveness in evading detection by state-of-the-art analysis tools. In section 6, we discuss our findings along

with the ethical aspects of our work. Finally, Section 7 summarizes our contributions and discusses ideas for future work.

2. Background and Related Work

In the following subsections, we briefly introduce the terminology and relevant concepts.

2.1. Windows Internal Workings

In the context of cybersecurity, it is essential to define policies of whom is allowed to perform specific actions, which can be defined by, e.g. white/blacklisting roles, users or actions depending on the needs of the system and the organization. Therefore, in operating systems, we have to define which CPU instructions are allowed to be executed or which resources can be used and by whom and when. We conceptualize some nested rings which would enable different access levels to the ones that can use them. These access levels are numbered from 0 (highest privilege) to 3 (lowest privilege). Typically, in ring 0, we have the kernel and system drivers. In ring 1, we have device controllers, etc. In ring 2, we have database management, device drivers, etc. Finally, in ring 3 we have the so-called "User mode", where actual applications are executed.

Windows, like most modern x86 kernels, use only two privilege levels, namely, rings 0 and 3. Therefore, we have the "*kernel mode*" (where one's code has complete and unrestricted access to the underlying hardware) and the "*user mode*" (where the executed code does not have direct access to hardware or referenced memory).

Due to their architecture, two of the most basic dynamically linked libraries are `user32.dll` and `ntdll.dll`. Both libraries serve as proxies between user-mode and kernel-mode. However, the latter allows direct communication with the Windows' kernel using system calls. Therefore, most of the other Windows libraries would eventually end up calling APIs from `ntdll.dll` [15]. A simplified overview of the Windows architecture is presented in Figure 1.

2.2. Portable Executable File Format

The standard executable format type in Windows is the so-called the Portable Executable (PE) file format which is used in both x86 and x64 architectures. Common extensions of such files include `.exe`, `.dll`, and `.sys`.

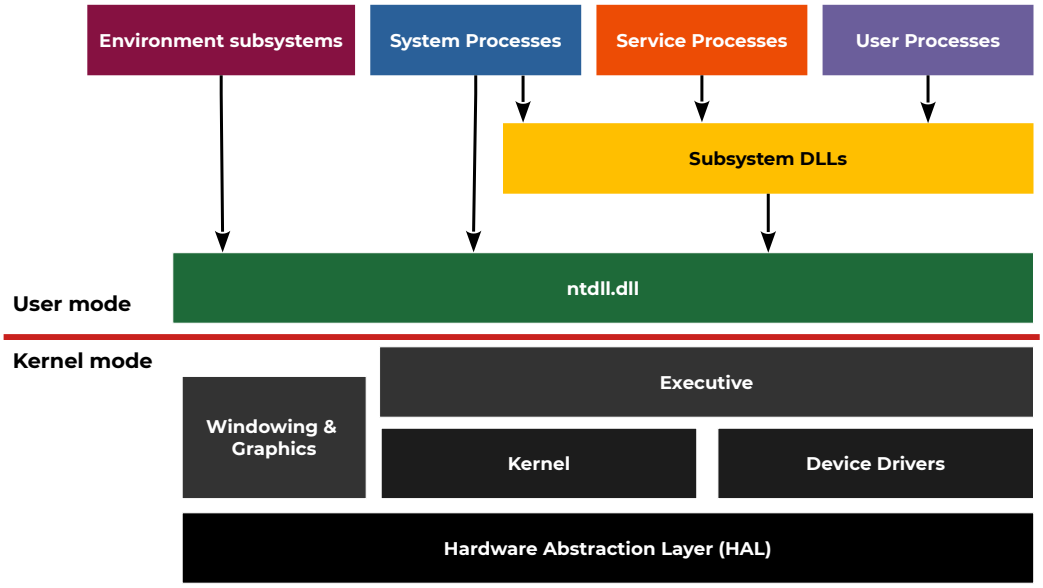

Figure 1: Overview of Windows architecture.

In principle, a PE file follows a structured format (see Figure 2), which allows Windows to execute the code it carries by providing the necessary information in an easy-to-use format to the dynamic linker. In fact, PE files are mapped in memory similarly to the way they are stored in media. In this regard, the loader uses the memory-mapped file mechanism and maps the corresponding parts of the file into the allocated virtual address space for the binary.

The basic structure of a PE file can be split into two substructures, one containing the header (all technical information about the file needed to map the file in memory and execute it), and one containing the different sections of the file which are the actual code and all the information needed to be stored in memory.

The `DOS` header consists of the first 64 bytes of a PE file, which allows DOS to recognize it as a valid executable file. The first two bytes (`e_magic`) are set to $0x54AD$, which is the ASCII representation of "MZ". Anti-malware software often use this signature when scanning memory to locate PE executables. The final and most important field of this header is located at offset $0x3c$ (`e_lfanew`), a 4-byte offset where the actual PE file header is located from the beginning of the file. The only two important fields are `e_magic` and `e_lfanew`. Most compilers and linkers will ignore all other fields of the

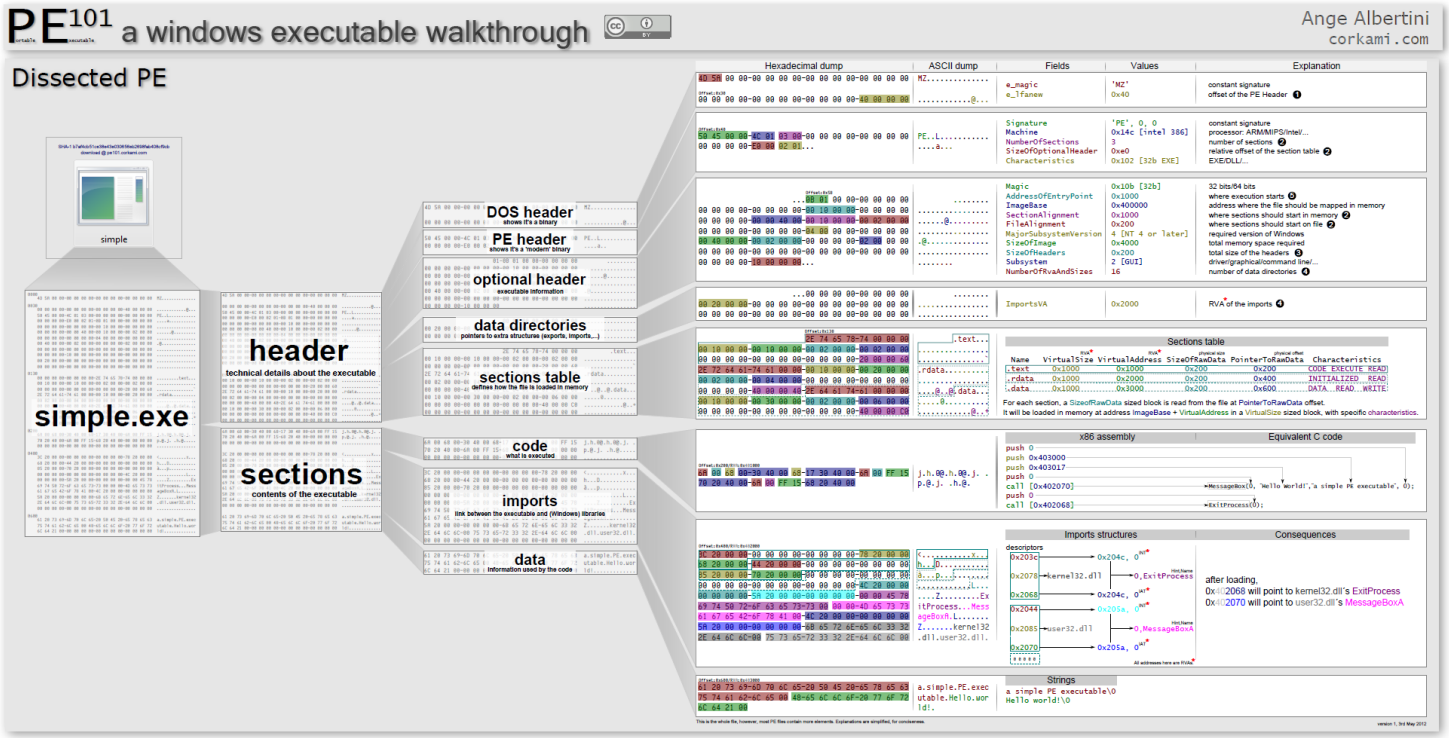

Figure 2: Overview of PE format. Source: Ange Albertini `https://github.com/corkami/`

DOS structure. Malware may also alter its standard DOS structure in an attempt to fool static analyzers [16, 17].

The `PE` header contains information such as "PE" signature, processor for which the code is written for (e.g. ARM, Intel and MIPS), how many sections the executable has, relative offset of the section table, and file types (e.g. EXE and DLL).

The `Optional` header contains information on how the executable should be mapped and executed. Such information includes the reference architecture (32/64 bits), where execution starts (`AddressOfEntryPoint`), address where the file should be mapped in memory (`ImageBase`), size of headers, and memory required.

Then, we have the data structures which are pointers for other structures such as import and exports of the executable. For instance, the Import Address Table (IAT) is defined here, and it is used as a lookup table when the application is calling a function in a different module. As the name suggests, the Sections table contains information required for the sections of the PE file that are followed.

Windows allows processes to have multiple threads of execution, and each has different storage, called Thread Local Storage (TLS). This information is stored in the `.tls` section, also described in the Sections table. What is important about this section is that it is loaded and initialized before the entry point is run. The latter is important as most analysts would start their investigation by setting a breakpoint at the entry point of the executable.

Another important aspect related to PE executables, when loaded in memory, is the Process Environment Block (`PEB`) structure. This is populated in user address space at runtime. The `PEB` contains various pieces of information about the process such as `BeingDebugged` (a flag that indicates that the process is debugged), `SessionId` (associated Terminal Services session identifier) and `ProcessParameters` (information about the command line and the path of the executable which the process is related to). It can be accessed either via Windows API or directly from memory. More precisely, its address can be located at offset $0x30$ (for 32-bit systems) from the `fs` segment register. The `fs` segment register is a special register of Windows for the x32 architecture and is used as a pointer to data structures and information of an active thread. What is important about the `PEB` structure is that among others it contains a pointer to PEB's Loader Data structure at offset $0x0c$. This is where we can traverse the Loader Module structures through the doubly linked lists to obtain useful information about the base

address of each loaded DLL, their entry point, their names, etc.

For more details regarding the PE format, the interested reader may refer to [18].

2.3. Hooking

Hooking can be defined as the interception of specific functions or system calls to monitor and/or alter its execution [19]. As previously discussed, hooking can be undertaken in the event that one does not have the source code, but there is a need to determine which API calls, subroutines are called from a given process. By being able to monitor these calls, one can "detour" these calls to alter the calls or the returned results, modifying in this regard the code execution in a binary, or even the entire operating system (OS). As such, hooking is widely used in malware analysis [20, 21, 22], software testing [23, 24, 25], troubleshooting of application misconfigurations, and manipulation of binaries to perform their analysis.

In general, hooks can be categorized into user and kernel-level hooks. More precisely, the two primary user-level hooking methods are IAT and inline hooking. In the former, one attempts to exploit the fact that an executable is not aware of the memory addresses associated with the libraries it depends on are loaded. Therefore, in IAT hooking, one usually injects a DLL containing the hooking code in the target process. The DLL can then rewrite the IAT entries and redirect them to handlers that it manages. Inline hooking is a more straightforward approach, as practically one overwrites the first bytes of the function (s)he wants to monitor by placing a jump that redirects the execution to a custom function. Clearly, if there are recursive calls to the function that is being investigated, this may create bottlenecks. Clearly, due to their nature, user-level hooks can be easily removed as applications can modify any memory page in their private allocated address space.

We refer readers interested in other methods for kernel-level hooking, such as hooking the System Service Descriptor and the Interrupt Descriptor Table, to existing literature such as [26, 27, 19, 28].

To provide malware protection, anti-malware software relies on API hooking. In this regard, anti-malware software registers the API calls and monitors the parameters of each process. In other words, they apply a mixture of user and kernel-level hooks. Clearly, hooking every possible API function would incur significant overheads affecting the performance. Therefore, anti-malware software hooks target specific userspace API functions, which

are known to be used by malware. This significantly reduces the associated overheads. To achieve this, anti-malware software usually places a hook to all newly created processes before the initial execution of the underlying main module. In this way, they divert the execution flow to their monitoring library, which is appended to the address space of the new process.

While one would assume the prevalence of kernel hooks in Windows (due to their low-level access), anti-malware software and debuggers heavily depend on user-level access. The reason is that Windows has special mechanisms to protect the integrity of the Windows kernel. One such mechanism is PatchGuard, which is designed to prevent modification to the Windows kernel and critical kernel data structures. Therefore, kernel-level hooks from anti-malware software may result in severe usability issues after system updates. This is also the reason to use third-party software for kernel-level hooking, with the most common one being Detours[1] from Microsoft. Finally, to provide low-level input/output, Windows provides File System Minifilter drivers[2] that must be digitally signed.

2.4. Anti-debugging

Debuggers are key tools for malware analysis, enabling the study of the software dynamically fashion, through a step-by-step execution of the code to examine its internals and impact. Debuggers may also manipulate the execution environment by altering, for instance, memory, registers, values of variables, configurations, among others. Furthermore, they support the disassembling of the binary code, tracing of system calls, capturing of exceptions, and so forth [29]. Therefore, they allow the analyst to inspect the binary code in greater detail, compared to static analysis which can be blinded by the use of packers, obfuscators etc. This is because debuggers provide the analyst to manipulate the low-level runtime behaviours of the malware being investigated. Thus, debugging facilitates the analyst in understanding in-depth the behaviour of malware, the mechanisms it uses, its capabilities, and assess its possible impact.

However, malware authors have been known to use anti-debugging techniques to obstruct malware analysis. Anti-debugging is the implementation

[1] `https://github.com/microsoft/detours`

[2] `https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-concepts`

of one or more techniques to hinder attempts at reverse engineering or debugging a target binary [30]. Once the malware realizes that it is being executed inside a debugger, it will attempt to deviate from its normal code execution path or interrupt/crash the debugger. This prevents analysis from being performed, introducing an additional time and cost for the analyst when reverse-engineering the binary.

2.5. Anti-VM

To study malware and its behaviour, a common approach is to execute the malware in a virtualized or sandboxed execution environment. This allows an analyst to observe the behaviour of malware by monitoring the system function calls and filesystem changes it performs, or even investigate memory snapshots during its execution. When the analysis is completed, the environment can be deleted, typically without affecting the host machine [31]. Therefore, an analyst has a fast, disposable, and secure way to examine malicious samples.

Not surprisingly, malware authors have started designing malware to detect whether it is being executed in a VM or sandbox. Such malware will then adjust their behaviour, usually by not executing the malicious activity or stalling its execution, when it detects that it is being executed in a VM or sandbox. In recent years, one observed trend is for malware to focus only on the detection of a sandboxed environment rather than generic virtualized environment due to the popularity of production systems running in virtualized cloud environments [32].

To facilitate the management and communication to the VM manager (VMM), virtualized systems leave some artifacts in guest OS. These artifacts include among others processes, registry keys and values, loaded and exported DLL's, network artifacts (e.g. specific MAC addresses and adapter name), file system artifacts (e.g. `system32\vboxtray.exe`), special directories that are created (e.g. `%PROGRAMFILES%\VMWare`), virtual devices (e.g. `\\.\HGFS`), and hardware label. Such artifacts also facilitate the malware's detection of a VM. For a brief overview of several such methods and their efficacy, the interested reader may refer to [10, 33, 34, 35]. Further evasion techniques are discussed in [36, 37] and more recently in a technical report from Checkpoint research [38].

2.6. Countermeasures

To counter these anti-analysis methods, several methods have been proposed in the literature.

MalGene [39] tries to abstract and not stick to the implementation details as many variations of the code may bypass the checks. The concept is to try to look at the end result by using data flow analysis and data mining techniques on the system calls. Using them, MalGene tries to determine whether the collected information from the binary could be used from an evasion technique.

Identifying misconfigurations, as well as understanding differences in execution environments and how one would locate them, are also crucial countermeasures. This observation also aligns with the proposed approaches, Apate in [29] and VM Cloak in [40]. Apate attempts to hide the existence of the debugger and sets breakpoint not only at the entry point but also in the TLS callbacks. This allows one to identify one of the 79 attack vectors it supports. TitanHide[3] hides debuggers in radically modified Windows installations by installing SSDT hooks. To allow such functionality, the machine must be running with disabled PatchGuard. In VM Cloak, the researchers try to identify CPU specifications that VMs fail to follow, and under-specifications of processor's instructions. They then monitor each malware command to detect potential pills. Similarly, approaches such as the one of [41] attempt to harden the sandbox environment to prevent its discovery from the malware.

One could also attempt to dynamically analyze the malware without introducing any in-guest monitoring component into the malware execution environment [42, 43, 44, 45, 46]. This approach prevents the malware from fingerprinting the analysis environment as the analysis system does not try to be indistinguishable from a real host, but is actually a host. These systems operate in a host environment and have a special service that takes a malware sample, copies it locally and executes it, without leaving any trace of its intervention. Then, changes on the system are monitored and compared with the execution in other environments which can imply the existence of a new evasion technique.

Finally, if there is kernel-level access, approaches such as Mac-A-Mal [47] may efficiently detect evasion mechanisms. The corresponding system calls can be captured and dealt with appropriately to prevent the malware from

[3] `https://github.com/mrexodia/TitanHide`

identifying the existence of a virtual environment or a debugger.

3. Adversary model

In our approach, we consider two adversaries. Adopting the model definition of Do et al. [48], we consider that both adversaries have similar assumptions and capabilities but eventually converge in terms of goals. In essence, these adversaries can be associated by duality as elaborated below.

First, we assume that the adversary, Malory, has a binary, an x32 PE executable, which is used in an attack vector to infect and eventually compromise a particular target. From a cyber kill chain perspective, ANTI is introduced during the weaponization phase as the binary is processed by Malory to deliver an armoured binary, denoted as $\mathbb{B}$. We also make the assumption of a secure target adhering to the principle of *least privilege*, so $\mathbb{B}$ would only be executed in user level. In this case, this security assumption would work on Malory's advantage as the victim would not have in this case complete access to the binary's context. However, in the case of the target being a security analyst with dynamic analysis capabilities, Malory's goal is to prolong the analysis time or even completely evade detection mechanisms. Therefore, Malory expects that one of her potential victims might not trust her and submit $\mathbb{B}$ for analysis to a service like VirusTotal[4] which uses automated analysis systems like Cuckoo[5] to determine whether the submitted binary is malicious. Similarly, Malory expects that an analyst might try to reverse engineer $\mathbb{B}$ and her goal is to dynamically alter the behaviour of $\mathbb{B}$ once such potential analysis behaviour is detected, e.g. execution in debug mode or within a VM. Hence, once executed, $\mathbb{B}$ initially has typical user-level access and the capabilities of the corresponding user in the execution environment.

With regards to the second adversary, we assume that organisation $\mathbb{O}$ produced a binary $\mathbb{B}$ containing some Intellectual Property and is making it available to another organization $\mathbb{M}$. The goal of adversary $\mathbb{M}$ in this case is to analyse the binary to bypass potential DRM controls, run it on non-authorised hardware or to extract any industrial intelligence. From a security standpoint, the adversary trusts the binary and may attempt to run the code with any privileges, perform static or dynamic analysis. With regards

[4] `https://www.virustotal.com`
[5] `https://cuckoosandbox.org/`

to the former, the binary may already have further obfuscation techniques, rendering static analysis impractical (comparable to a computationally secure model). This would force the adversary to follow the dynamic analysis route. In this case this adversary's goal and approach would be equivalent to that of Malory's.

4. Methodology

We now present our approach, ANTI, with a proof of concept available on GitHub[6] under the MIT license. ANTI's main purpose is the automated integration of known anti-debug and anti-VM techniques in a PE executable. At the time of writing, the anti-debug part is heavily based on approaches described in [49]. However, it can easily be extended to integrate other methods. ANTI is designed for x32 PE executables and evaluated on Windows 10 machines. In this section, we will describe the functionality of ANTI and the ways it fortifies malware.

A simple overview of how ANTI works is illustrated in Figure 3. Initially, one would take an x32 PE executable and pass it to ANTI. ANTI will reformulate the PE header and append the armouring functions at the end of the file. Once the file is executed, ANTI will try to determine whether it has been hooked and try to remove the hooks on DLLs that it supports. Currently, it supports only `ntdll.dll`. Once the hooks are removed, ANTI will try to detect whether it is being debugged or executed in a VM, and once detected, ANTI would terminate the debuggee process or migrate to another process.

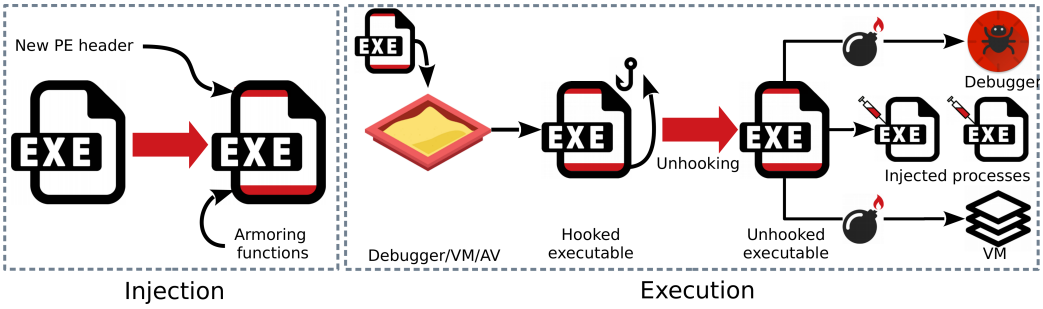

Figure 3: An overview of how ANTI works.

[6] `https://github.com/nihilboy/anti`

4.1. Overview

To armour a binary, ANTI performs two steps. First, it appends a new section at the end of the executable which contains all the necessary functions to perform the checks for debuggers and VM. While this provides some functionality, it cannot be executed timely. Therefore, the next step is to scan the PE to identify the appropriate entry points and amend them accordingly, so that the checks of the appended section will be performed before any other action of the executable.

The above are realised by ANTI with two key functions. The first key function is to create a new empty section to be appended at the end of the executable. This functionality is similar to the conventional "backdooring" of binaries that malware infectors (also known as file or virus infectors) use. While this version of ANTI does not yet support different methods like code-cave jumping or appending at the end of the text section, they will be included along with other stealth methods to ANTI in the near future. This function firstly pre-processes the executable's PE Header to obtain information necessary for the addition of a new executable section with a user-supplied name, before inserting the required information in the PE Header fields (`NumberOfSections`, `SizeOfImage`, `Characteristics`, etc).

The second function is responsible for the actual insertion of the inline assembly code. ANTI adds to the binary the ability to execute from dynamically created TLS Callbacks. First, it creates a TLS directory entry by specifying its size (`sizeof(IMAGE_TLS_DIRECTORY)`) in the Size field and updates its Virtual Address (`TLS Directory RVA`) field to point 4 bytes after the start of our newly created section. At this location, we manually build the TLS directory structure by adding the necessary fields (see also [50]). For TLS Callbacks, Windows Loader reads Data Directories → TLS Directory. If VirtualAddress is not zero, then the loader reads Address of Callbacks Array, executes the first callback in the list and continues by reading the next element of Address of Callbacks Array. In other words, we can dynamically add and remove new TLS callbacks from another TLS callback. Moreover, it is possible to register or 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 needs the data. Thus, we can change the TLS table while in TLS callback itself; those added will normally execute since they are not cached by the Windows loader.

TLS callback is a powerful anti-debugging technique and a good place to perform a debugger presence check since the Callback function will be

called before the executable reaches the main module entry point. As ANTI makes the executable start execution from dynamically TLS Callbacks, the `AddressOfEntryPoint` field in the Optional Header is overwritten with a random value within the text section's range since it is no longer needed. The executable jumps to the original Entry Point after the unhooking/anti-debug/anti-VM checks have been performed. Finally, the checksum of the new executable is computed and corrected in the respective `Checksum` field of the Optional Header. In case of debugging or virtualized environment being detected, ANTI calls `SwitchDesktop` and `NtTerminateProcess` to terminate the debuggee process. ANTI also has the ability to attempt to migrate to a remote process once it detects a debugger. Migration works using a standard process injection technique by creating a remote thread in the target process specified by its PID given by the user. To be able to migrate, the debugger must run as administrator or have the debug privilege enabled.

4.2. Unhooking

ANTI manages to bypass security solutions due to its unhooking capabilities. It works, firstly, by mapping a view of `ntdll.dll` from the disk using the standard sequence of Windows functions: `CreateFileW()`, `CreatFileMappingW()`, `MapViewOfFile()`. This has the advantage of acquiring a clean unhooked view of `ntdll.dll`.

A clean version of `ntdll.dll` can also be mapped in process address space, more stealthily by loading the `ntdll.dll` from the `KnownDlls` global section. This technique creates less noise as it does not interact with the filesystem, and can also prevent fake/instrumented `ntdll.dll` copies of system DLL's being loaded from an applications folder.

Furthermore, after the unhooking process takes place, we can apply hooks on all binary functions. The hook handler is responsible for performing anti-debug/anti-VM checks each time a function is called. This way, malware analysis checks can be active even after the execution passes the OEP; thus, a manual approach is unavoidable.

Note that `ntdll.dll` is a library that has no import, and all its exported functions are basically wrappers for system calls; thus, it can directly communicate with the Windows kernel through system calls. Additionally, `ntdll.dll` contains Nt* counterparts of many functions exported by `Win32API`, such as `kernel32.dll`, `user32.dll`, and `gdi32.dll`. These will eventually call their corresponding Nt* function to implement a system

call. For instance, `CreateFile` will call `NtCreateFile`. We also posit the importance of unhooking `ntdll.dll`, as it allows the interaction with the NativeAPI at the lowest user-mode level.

The next step in our unhooking strategy is to parse the PE headers, allocate space, and finally copy the DLL header and sections in our newly allocated region of memory. Then, we implement a relocation procedure where the delta of relocation is computed by subtracting the image base of the copied `ntdll` located in the PE's optional header from the base address of the initial library. Finally, we compare the two images, namely: the image from the disk and the already loaded image. Should modifications be found, they are overwritten with the clean unhooked code.

Continuing this process for every loaded DLL, a binary that acquires the aforementioned mechanism can easily detect and remove all user-mode hooks. Monitoring can be easily bypassed for many standard security/analysis solutions, since many antivirus engines, as well as instrumentation frameworks, use DLLs injected to the binary's process at runtime. For example, Frida injects its own DLL, namely `frida-agent-32.dll`, to perform the instrumentation. Bitdefender uses its own DLL, named `atcuf32.dll`, which is injected pretty early in the loading process, that contains hooks in functions of many modules to perform its monitoring.

4.3. Anti-Debugging

To armour a binary against a debugger we consider two key strategies, one static and two dynamic. In the static, we check for flags that denote the presence of a debugger. The dynamic come in two flavours. First we try to unhook the binary from possible hooks that have been made on a key windows library. Then, we log timestamps during execution to identify lags that can be attributed to debuggers.

More precisely, ANTI is designed to integrate existing anti-debug methods such as those reported in [49]. It also appends to a binary known flag checks to infer the presence of a debugger. These checks include examining if the `BeingDebugged` byte-field located at offset $0x02$ in Process Environment Block is set or calling the equivalent `IsDebuggerPresent()` function. This method is probably the most well-known and easy to bypass. Another common method is the use of PEB's `NtGlobalFlag` value at offset $0x68$. Normally, when a process is not being debugged, the NtGlobalFlag field contains the value 0x0. On the contrary, if a program was launched from

a debugger, the following flags are set by the Windows heap manager in `PEB!NtGlobalFlags`:

- `FLG_HEAP_ENABLE_TAIL_CHECK` ($0x10$),
- `FLG_HEAP_ENABLE_FREE_CHECK` ($0x20$),
- `FLG_HEAP_VALIDATE_PARAMETERS` ($0x40$),

when a debugger creates a process. Therefore, checking for a combination of the aforementioned flags ($0x70$) can help to detect the presence of a debugger. At offset $0x18$ in PEB we can find `ProcessHeap` field, which is a pointer to the heap base. This is the first heap allocated by the loader, which can also reveal if the process is running in debug mode. Specifically, the values in fields `Flags` and `ForceFlags` at offsets $0x40$ and $0x44$, inside the heap's structure header can be compared to `HEAP_GROWABLE` (2) and 0 respectively, to detect a debugger. The ntdll `RtlQueryProcessHeapInformation()` function can also be used to read the heap flags of the current process. The techniques described so far are based on simple process memory checks of the fields holding standard values. These can be easily bypassed by patching the aforementioned fields inside the debugger prior to their execution.

ANTI also implements API-based anti-debug methods based on functions exported by ntdll.dll after it has reinforced them by removing any detected hooks. One such technique is the call to `NtSetInformationThread()` with `ThreadInformationClass` class value equal to $0x11$ (`ThreadHideFromDebugger`) on the current thread. Upon calling the function, the debugger will no longer receive events from the thread, and if that happens to be the main thread the process is terminated [49]. `NtQueryInformationProcess` is another anti-debug method implemented in ANTI that can be called in various ways to reveal the presence of a debugger. For instance, it can retrieve information about the port number of the debugger by using the value $0x07$ (`ProcessDebugPort`) of `ProcessInformationClass` class on the current process. If the returned value is not zero, then it implies the presence of a user-mode debugger. The same approach can be used to retrieve the debug object handle if we pass as input `ProcessDebugObjectHandle` ($0x1e$) on the current process. Additionally, we call `NtQueryInformationProcess` with the undocumented flag `ProcessDebugFlags` ($0x1f$), which returns zero if a debugger is present.

Timing checks can also exploit the debugger's single-stepping functionality, which causes a delay compared to normal execution that can be measured

and used to stop the debugging session. This is accomplished in ANTI by using the RDTSC instruction and the `NtQueryPerformanceCounter` Native API function. The RDTSC (Read Time Stamp Counter) instruction returns the current value of the timestamp counter from the processor, while `NtQueryPerformanceCounter` retrieves the current value of the performance counter. Both return high-resolution timestamps that can be used for time-interval measurements, although the latter is suggested by Microsoft as being more reliable.

4.4. Anti-VM

ANTI implements several techniques to detect a virtualized environment, such as calling `NtCreateFile()` function to determine whether the device name maches commonly used names by VMs. Similarly, accessing \\.\\HGFS or \\.\\vmci (VMware drivers) will return `INVALID_HANDLE_VALUE (0xffffffff)` in `eax` in a host, which differs from the value that would be returned inside VMWare.

Another approach to detecting execution within a virtual machine is to examine the machine's number of processors. This can be accomplished either by examining the Process Environment's Block field `NumberOfProcessors` at offset $0x64$ or by using the Windows `NtGetSystemInfo()` function end examining its corresponding return value in the `SYSTEM_INFO` structure. Should ANTI determine that the number of processors is below three, it interrupts the execution.

Additionally, we integrate a usual timing anti-sandbox technique that is based on the `NtGetTickCount()` function. The output of this function represents the milliseconds that the system has been alive, up to 49.7 days approximately. This value informs the decision-making process of malware. For example, a typical sandboxed malware analysis process involves starting up the VM, copying the malicious binary, and executing it either inside a debugger or through monitoring systems. Therefore, if the malware has a timing check handle, then it can wait an extended period of time to infer whether it is running inside a VM.

`CPUID` is an instruction for the x86 architecture, which allows executables to retrieve various information about the CPU (e.g. vendor string and model number, size of internal caches, and list of CPU features supported) that can also be used to facilitate VM detection. Depending on the input values that were initially used in the `eax` register, `CPUID` returns different results. For, example when called with the `eax` register being $0x40000000$, `CPUID` returns

the vendor ID string in `ebx`, `ecx` and `edx` registers. However, when this is performed inside a VM, it reveals the corresponding virtual machine vendor. For instance, in VMware, it returns `VMawareVMaware`. Additionally, when executed with input 1 in `eax`, it returns the processor's features. Thus, we can infer the existence of VM since the 31st bit value in `ecx` will be 0 when it is in the host, unlike guest where the same bit will be equal to 1.

5. Evaluation

Our evaluation methodology aims to assess whether ANTI manages to bypass state of the art analysis tools and what is the additional overhead that it introduces. To this end, we first perform some statistical tests to quantify, e.g. how many extra bytes are appended, what is the processing overhead etc. Then, we test the armoured binaries in a virtual machine to verify that they can detect the execution within a virtual environment. Finally, we examine the armoured binaries against state of the art debuggers fully equipped with plugins and standalone tools to determine whether ANTI manages to trick them and bypass their methods.

We evaluated ANTI by conducting experiments on all major Windows debuggers, a number of anti-anti plugins (plugins that counter anti-debug/VM methods) and stand-alone detectors. All evaluations were performed on a fully updated clean install of Windows 10 system, 64-bit operating system (version 1803) as host, and on an up-to-date VMWare Workstation 12.5.2 running snapshot containing Windows 10 Enterprise Evaluation (version 1709), 32 and 64-bit as guests. We evaluated ANTI with the debuggers, debugger plugins, and standalone tools listed in Tables 1, 2, and 3, respectively.

Experiments were conducted on various checksum verified PE executables randomly chosen (e.g. Putty) and simple executables specifically developed for evaluation purpose. The evaluation was performed in all steps of our approach to assess and demonstrate its efficacy. In essence, the evaluation consisted of the following steps. First, we executed the executables within a

[7] `https://github.com/brock7/xdbg`
[8] `https://https://github.com/stonedreamforest/NaiHeQiao`
[9] `https://github.com/x64dbg/ScyllaHide`
[10] `https://steel.isi.edu/Projects/apate/`
[11] `https://github.com/secrary/makin`

Debugger	Version
IDA pro debugger	7, 5
Immunity Debugger	v1.85
OllyDebugger	v1.10, v2
CheatEngine	v6.8.1
x64dbg	Build Aprl 5 2018
Windbg	10
Obsidian debugger	v0.11
Microsoft Visual Studio Debugger	v15.4.0

Table 1: Debuggers and their versions that were used in the evaluation.

VM and then through a debugger to determine whether they have inherent mechanisms preventing their execution. All executables that contained such mechanisms were pruned from the test dataset. Then, we used ANTI with different configurations to inject the proposed analysis of evasion controls. For instance, we disabled the anti-debugging or anti-hooking methods on the armoured binary.

5.1. Statistical properties

To evaluate the impact of using ANTI on a binary, we performed several tests and compared the original and armoured binaries. To this end, we calculated the average bytes difference in virtual memory, the time to start overhead, and the average entropy of file differences on a dataset consisting of 40 executables. One can observe from the values reported in Table 4 that ANTI's overhead is minimal, in terms of time and entropy. However, the memory footprint is as expected considerably higher, although acceptable in practice.

5.2. Anti-VM evaluation

For the anti-VM evaluation, we first executed each of the binaries both inside VMware and on the host to verify that they have not previously implemented anti-VM functionality. All binaries were executed normally. Then, we executed ANTI against each binary and re-executed them. When inside the VM, the execution stopped by switching desktops as a proof of concept; execution on the host was normal as expected by the program.

Debugger Plugins	Version
PhantOm (OllyDbg)	v1.85
StrongOD (OllyDbg)	v0.4.8.892
OllyAdvanced (OllyDbg)	v1.27
SharpOD (x64dbg)	0.6
aadp (OllyDbg,Immunity)	v0.2
HideDebugger (OllyDbg)	v1.2.4
Ida Stealth (Ida pro debugger)	v1.3.3
OllyExt (OllyDbg)	v1.8
Stealth64 (OllyDbg)	v1.3
HideOD (OllyDbg)	v0.181
xdbg[7] (x64dbg, cheatengine)	n/a
NaiHeQiao[8] (x64dbg)	v2017-3-15
ScyllaHide[9] (x64dbg, IDA, Olly, standalone)	v2019-05-24
Apate[10] (WinDbg)	n/a

Table 2: Debugger plugins and their versions (where available) that were used in the evaluation.

Standalone tools	Version
makin[11]	n/a
ApiMonitor	v2

Table 3: Standalone tools and their versions (where available) that were used in the evaluation.

Property (Δ)	Value (mean)	Standard Deviation
Memory footprint	1,22 MB	0.49
Time overhead	2.21 ms	1.27
File entropy	-5.71 bits	125.84
Added code file ratio	1.32%	0,31

Table 4: Statistical properties. The difference refers to value of the patched binary minus the corresponding value of the original binary.

Since malware analysis from Cuckoo[12], a widely used tool for malware

[12] `https://cuckoosandbox.org/`

Property	Value
File size overhead	10240 bytes
Contains TLS callbacks	True (dynamically added)
Added section entropy	7.293
Entry Point	randomized
Checksum	corrected
ASLR	Disabled

Table 5: Static analysis properties. The values refer to elements of the patched binary that have changed.

```
mov dword ptr ss:[ebp-204],eax     [ebp-204]:ZwQueryInformationProcess
xor ebx,ebx
push ebx
mov ebx,esp
push 0
push 4
push ebx
push 7
push FFFFFFFF
call dword ptr ss:[ebp-204]        [ebp-204]:ZwQueryInformationProcess
pop ebx
inc ebx
cmp ebx,0
je putty.4C3AEE
```

Listing 1: Dissasembly of NtQueryInformationProcess anti-debug method.

analysis, is performed inside a VM, we opted to test the implications that a binary armoured with ANTI could introduce. In our evaluation, we used the 2.0.6 version of Cuckoo. As in the previous case, we created an executable armoured with ANTI to shut down a machine when it detects virtualization. The latter opens a door opportunity for the adversary. Practically, the VM is shut down once ANTI detects virtualisation. However, because of not proper communication between the host and the guest, Cuckoo enters an infinite loop. The latter is a crucial issue since the main Cuckoo process must be restarted to operate again properly. This is a manual process which is not foreseen in an automated malware analysis environment. Therefore, ANTI bypasses automatic analysis with Cuckoo creating further issues for the analysis of other binaries, if we consider it in a batch process. Moreover,

it should be noted that no anti-VM or anti-debug signatures where flagged in the static analysis of the binaries armoured by ANTI.

5.3. Anti-debugging evaluation

For the anti-debugging evaluation, we executed the same clean executables inside the aforementioned debuggers and stand-alone detectors until they reached their entry point. All executions proceeded as intended. Next, we ran ANTI against all binaries, but we commented out the unhooking part of the code and kept only the anti-security methods. When inside the debugging environment, all executables crashed upon switching to the desktop environment. In the case of makin, the tool for revealing anti-debug attempts in executables, we noted that it stopped as it does not support TLS Callbacks. Finally, we executed the same binaries with ANTI anti-debugging plugins installed, and all checks turned on. We found that all debuggers remain hidden and as a result, binaries executed normally and reached their entry point. In Figure 6 the call to `NtSetInformationThread` is caught by ScyllaHide's hook handler when unhook functionality is off.

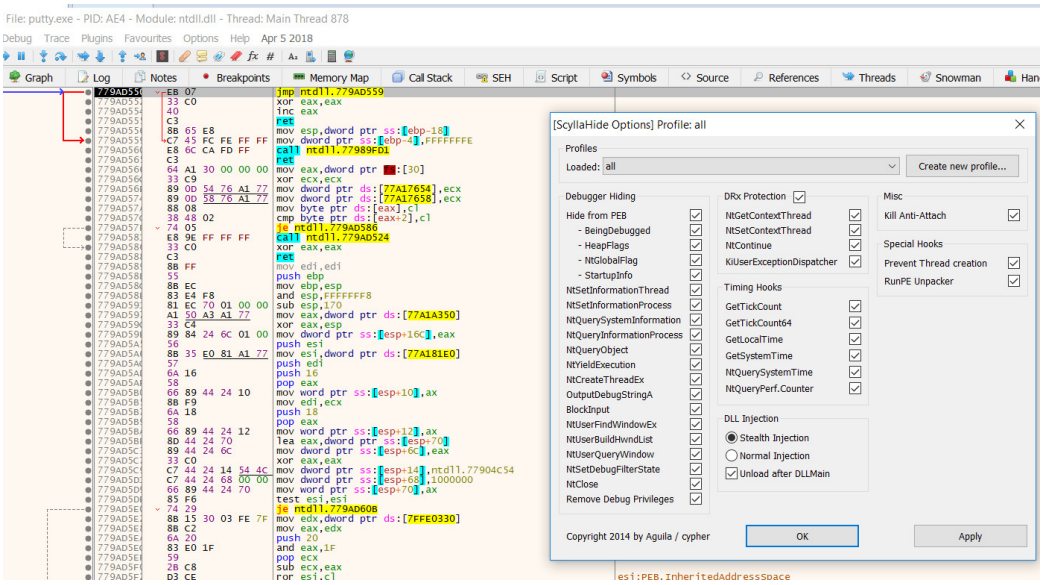

Figure 4: Bypassing fully activated ScyllaHide on x64dbg.

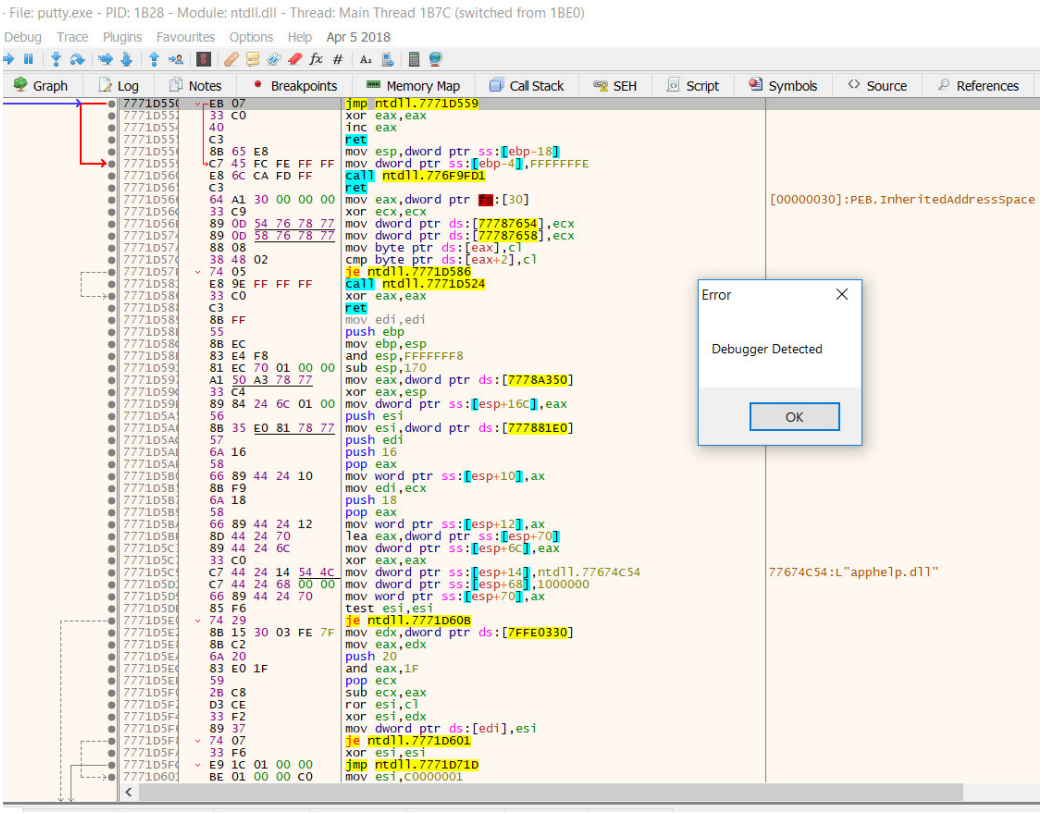

Figure 5: Disrupting debugger by prompting a Message Box.

5.4. Unhooking evaluation

For the final evaluation, we ran ANTI on all binaries with the unhook functionality turned on. The debuggers and plugins used in the evaluations failed, and the system has switched desktop and successfully hid the previously active desktop, thus validating our proof of concept. We can only return by killing the debugging process. No debugger remained hidden even with the respective plugins activated. Through debugging the binaries, we infer that all API based anti-debug methods that used `ntdll.dll` functions remain hidden. On the contrary, all methods based on specific flag checks of Process Environment Block failed (e.g.`IsDebuggerPresent`).

Listing 1 shows the `NtQueryInformationProcess()` method with the `ProcessDebugPort` value as the input, which is executed in the current process. This was executed normally and the jump to switch desktop was taken

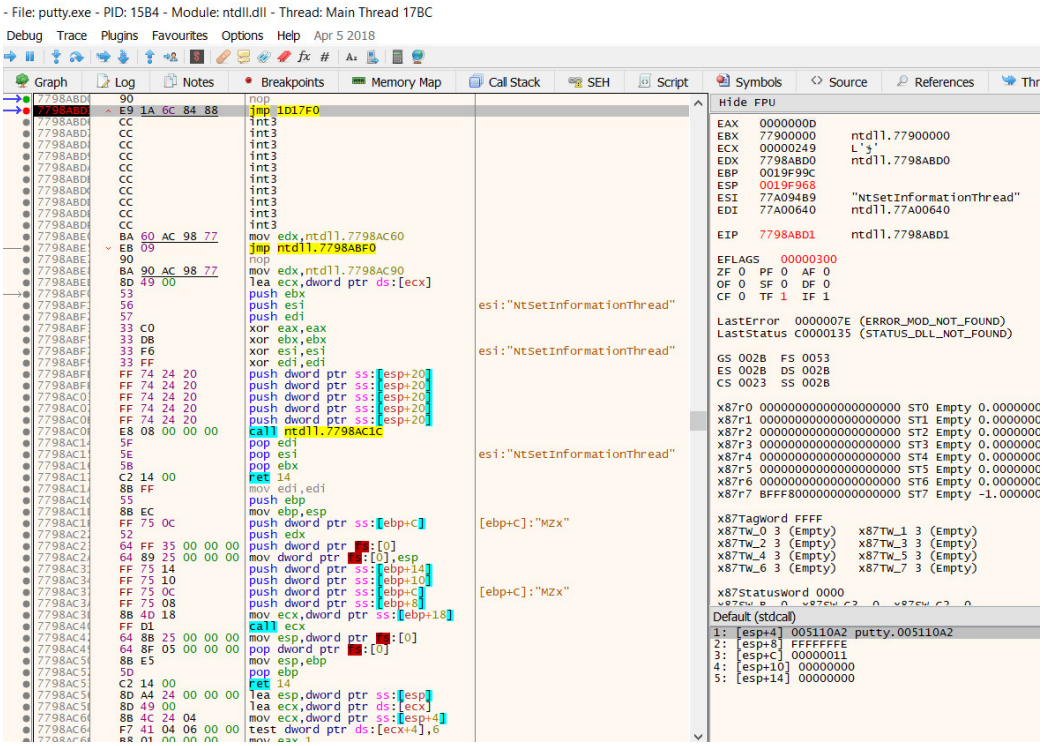

Figure 6: Jumping to ScyllaHide's hook handler.

when inside x64dbg under ScyllaHide fully activated. Figure 4 illustrates the `NtSetInformationThread()` method during single-stepping debugging session. Due to singe-stepping and the `NtSetInformationThread()` call on the main thread, the execution terminates before reaching the `SwitchDesktop()` implementation. In Figure 5, we changed the switching of desktops implementation with an error message box to demonstrate the validity of ANTI.

6. Discussion

To address the above challenges and risks, as the first line of defence, the malware analyst would need to establish the existence of anti-detection techniques. This could be potentially achieved by reverse-engineering the packing and armouring process by identifying suitable Indicators of Compromise (IoCs). In the context of this domain problem, these would most likely be in the form of patterns described, for example as YARA rules. Another future work on malware analysis evasion is to assess the extent of the

unhooking techniques with obfuscator classes, fuzzing capabilities, return-oriented programming and the resulting complexity and overheads incur by these additional operations.

The aforementioned approaches may render existing static analysis tools ineffective. Therefore, by combining such methods and based on the results of using ANTI, we argue that we may need to fundamentally change the way we perform virtualization, debugging and hooking to analyse malware as many of our existing malware detection approaches fail to reach their goal. Especially for the case of debuggers, the main issue that is highlighted by our work is that they rely on user-mode hooks that are known to be malleable by any executable. The main reason is that they share the same address space that they are allowed to tamper with, creating a race condition of who will manage to do it first. Moreover, one needs to consider that while a malware analyst would try to investigate TLS callbacks, this is not the case of most dynamic analysis tools. The dynamic analysis tools are highly dependent on the entry point, omitting checks for other alternatives. Therefore, malware may evade many security checks. Finally, there is a need to provide more effort in virtualizing operating systems to impede the task of differentiating them from actual hosts.

We believe that the above is rather alarming since they imply that methods that were considered to be easily traced by modern tools are not handled appropriately and may allow binaries to pass below the radars of modern automated tools. We need to understand that having security mechanisms at the same execution level that the malware is executed ends up to an endless game to catch the dog's tail. The authors of malicious software will simply deactivate the mechanisms once their binary is executed as they have the same permissions to modify the address space. Inevitably, the security mechanisms have to be integrated deeper. Therefore, despite the Windows kernel's security measures, we should reconsider the case of having kernel-level tools that could provide an environment immune to user-level tampering of address space.

ANTI and its evaluation showcase that the aforementioned arguments are more than valid. However, the goal of ANTI is not be used maliciously; therefore, we opted not to further armour it with, e.g. encryption or section randomisation and the like. Moreover, to avoid possible exploitation on the wild, we have created a simple YARA rule (Listing 2) to locate it in binaries. The YARA rule is based on a pattern and fragments of strings used by ANTI to find and load specific API functions in memory used to load ntdll from

disk.

While ANTI is a proof of concept, it manages to efficiently bypass dynamic analysis through sandbox environments (e.g. Cuckoo, execution in virtual machines etc.) and debuggers. However, a simple static analysis, as with the YARA rule above, illustrates its limitations. In this regard, ANTI lacks the randomisation aspect of a full-fledged tool to bypass static checks. This could be resolved by splitting the appended section into other sections, adding a packer, exploiting code caving, etc. Finally, since ANTI was created as a proof of concept to showcase the deficiencies of the current methods through the use of old yet well-known techniques, one could further extend its arsenal by adding more recent anti-debug and anti-VM methods or by unhooking other libraries beyond ntdll.

7. Conclusions

In order for malware to efficiently evade dynamic analysis, it would need to support a range of diverse anti-detection capabilities. While key analysis approaches (i.e., sandboxes and debuggers) are effective tools for studying a piece of software in runtime, they suffer from a number of limitations due to the way they are developed and the way they operate. Specifically, we demonstrated that a given malware could circumvent the analysis tools using our proof of concept through the use of well-known methods. Furthermore, the ever-increasing rate of new malware and the trend of substantial code reuse in many malware families may result into increased risk, should an approach exists to incorporate the dynamic analysis evasion capabilities automatically. In a recent incident that targeted a number of Australian entities, for example, it was reported[13] that

> *"We found in analysing the code itself ... the attackers had reused a lot of the code that had been used by other people in the past," Duca said. "And one particular tool that was used was a tool that was actually used in the February 2019 attack against Parliament House."*

As shown in this paper, the proposed approach can be streamlined and automated to armour any software with no built-in integrity checks. These

[13] `https://www.canberratimes.com.au/story/6800107/china-says-not-behind-australia-cyber-hits/`, last accessed June 20, 2020

findings are concerning, particularly when the methods that we used are known and well-documented. For example, in the June 2020 advisory[14] released by the Australian Cyber Security Centre's (ACSC) investigation of a cyber campaign targeting Australian networks revealed that

> *The actor has been identified leveraging a number of initial access vectors, with the most prevalent being the exploitation of public-facing infrastructure primarily through the use of remote code execution vulnerability in unpatched versions of Telerik UI. Other vulnerabilities in public-facing infrastructure leveraged by the actor include exploitation of a deserialisation vulnerability in Microsoft Internet Information Services (IIS), a 2019 SharePoint vulnerability and the 2019 Citrix vulnerability. The actor has shown the capability to quickly leverage public exploit proof-of-concepts to target networks of interest and regularly conducts reconnaissance of target networks looking for vulnerable services, potentially maintaining a list of public-facing services to quickly target following future vulnerability releases. The actor has also shown an aptitude for identifying development, test and orphaned services that are not well known or maintained by victim organisations.*

In other words, the methods we used in this paper can also be used by cyber attackers to bypass state of the art tools, where many malware can potentially go undetected or escape the detailed analysis since the tools do not provide the needed functionality or do not have the necessary privileges to do so. We argue that the latter aspect is crucial as leaving the security mechanisms to have the same privileges as malware is a potential attack vector that can be exploited.

There are a number of potential future research directions. For example, we plan to assess further the malware detection mechanisms that are widely used, as well as investigating malware evasion methods and using machine learning to identify such malware evasion methods in a timely manner.

[14] `https://www.cyber.gov.au/threats/advisory-2020-008-copy-paste-compromises-tactics-techniques-and-procedures-used-target-multiple-australian-networks`, last accessed June 20, 2020

```
import "pe"

rule YARA_ANTI {
   meta:
      description = "YARA_ANTI"
   strings:
      $op0 = { 58 8B D8 83 E8 2D 83 C0 24 83 C3 0F 89 18 C3 }
      $op1 = { 81 3E 43 72 65 61 ?? ?? ?? ?? 81 7E 07 69 6C 65 57 }
      $op2 = { 81 3E 43 72 65 61 ?? ?? ?? ?? 81 7E 0E 69 6E 67 57 }
      $op3 = { 81 3E 4D 61 70 56 ?? ?? ?? ?? 81 7E 0A 69 6C 65 00 }
      $op4 = { 81 3E 56 69 72 74 ?? ?? ?? ?? 81 7E 09 6C 6F 63 00 }
      $op5 = { 81 3E 6D 65 6D 63 ?? ?? ?? ?? 81 7E 03 63 70 79 00 }
      $op6 = { 81 3E 6D 65 6D 63 ?? ?? ?? ?? 81 7E 03 63 6D 70 00 }
      $op7 = { 81 3E 56 69 72 74 ?? ?? ?? ?? 81 7E 0B 65 63 74 00 }
      $op8 = { 64 A1 30 00 00 00 8B 40 0C 8B 40 14 8B 00 8B 00 8B
               40 10 8B D8 8B 43 3C 8B 7C 03 78 03 FB 8B 4F 18
               8B 57 20 03 D3}

   condition:
      uint16(0) == 0x5a4d and ( all of ($op*) )
}
```

Listing 2: YARA rule to detect ANTI.

Acknowledgments

This work was supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the project *YAKSHA* (Grant Agreement no. 780498) *CyberSec4Europe* (`https://www.cybersec4europe.eu`) (Grant Agreement no. 830929), *LOCARD* (`https://locard.eu`) (Grant Agreement no. 832735), and *ECHO*, (`https://echonetwork.eu`) (Grant Agreement no 830943).

The content of this article does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

References

[1] N. Ismail, Global cybercrime economy generates over $1.5tn, according to new study, https://www.information-age.com/global-cybercrime-economy-generates-over-1-5tn-according-to-new-study-123471631/ (2018).

[2] B. Lau, V. Svajcer, Measuring virtual machine detection in malware using DSD tracer, Journal in Computer Virology 6 (3) (2010) 181–195.

[3] R. R. Branco, G. N. Barbosa, P. D. Neto, Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies, in: Blackhat USA, 2012.

[4] J. Calvet, F. L. Lévesque, J. M. Fernandez, J. Marion, E. Traourouder, F. Menet, Waveatlas: surfing through the landscape of current malware packers, in: Proceedings of Virus Bulletin Conference, 2015.

[5] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, P. G. Bringas, Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers, in: 2015 IEEE Symposium on Security and Privacy, IEEE, 2015, pp. 659–673.

[6] G. Suarez-Tangil, G. Stringhini, Eight years of rider measurement in the android malware ecosystem: Evolution and lessons learned, CoRR abs/1801.08115.

[7] A. Bulazel, B. Yener, A survey on automated dynamic malware analysis evasion and counter-evasion: PC, mobile, and web, in: Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium, ACM, ACM, New York, NY, USA, 2017, p. 2.

[8] L. Martignoni, R. Paleari, G. F. Roglia, D. Bruschi, Testing CPU emulators, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09, ACM, New York, NY, USA, 2009, pp. 261–272.

[9] H. Shi, A. Alwabel, J. Mirkovic, Cardinal pill testing of system virtual machines, in: 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, 2014, pp. 271–285.

[10] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, J. Nazario, Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware, in: 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), IEEE, IEEE, 2008, pp. 177–186.

[11] P. Chen, C. Huygens, L. Desmet, W. Joosen, Advanced or not? A comparative study of the use of anti-debugging and anti-vm techniques in generic and targeted malware, in: J.-H. Hoepman, S. Katzenbeisser (Eds.), ICT Systems Security and Privacy Protection, Springer International Publishing, Cham, 2016, pp. 323–336.

[12] J. Conley, E. Andros, P. Chinai, E. Lipkowitz, D. Perez, Use of a game over: Emulation and the video game industry, a white paper, Northwestern Journal of Technology and Intellectual Property 2 (2) (2004) 1.

[13] C. Collberg, The case for dynamic digital asset protection techniques (2011).

[14] D. Jang, Y. Jeong, S. Lee, M. Park, K. Kwak, D. Kim, B. B. Kang, Rethinking anti-emulation techniques for large-scale software deployment, Computers & Security 83 (2019) 182 – 200.

[15] P. Yosifovich, D. A. Solomon, A. Ionescu, Windows Internals, Part 1: System architecture, processes, threads, memory management, and more, 7th Edition, Microsoft Press, 2017.

[16] R. Mario Vuksan, Tomislav Pericin, Constant insecurity: Things you didn't know about (pe) portable executable file format, `https://media.blackhat.com/bh-us-11/Vuksan/BH_US_11_VuksanPericin_PECOFF_WP.pdf` (BlackHat USA 2011, Las Vegas).

[17] Solar Eclipse, Tiny pe - creating the smallest possible pe executable, `https://web.archive.org/web/20120121050913/http://www.phreedom.org:80/solar/code/tinype/` (2006).

[18] Microsoft, Pe format, `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format` (2019).

[19] J. Lopez, L. Babun, H. Aksu, A. S. Uluagac, A survey on function and system call hooking approaches, Journal of Hardware and Systems Security 1 (2) (2017) 114–136.

[20] M. Egele, T. Scholte, E. Kirda, C. Kruegel, A survey on automated dynamic malware-analysis techniques and tools, ACM computing surveys (CSUR) 44 (2) (2012) 6.

[21] Y.-s. Jeong, H.-t. Lee, S.-j. Cho, S. Han, M. Park, A kernel-based monitoring approach for analyzing malicious behavior on android, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, ACM, New York, NY, USA, 2014, pp. 1737–1738.

[22] N. Totosis, C. Patsakis, Android hooking revisited, in: IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC), IEEE, 2018.

[23] M. Musuvathi, S. Qadeer, T. Ball, M. Musuvathi, S. Qadeer, T. Ball, Chess: A systematic testing tool for concurrent software, Tech. Rep. MSR-TR-2007-149, Microsoft Research (2007).

[24] T. Kim, N. Zeldovich, Practical and effective sandboxing for non-root users, in: Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), USENIX, San Jose, CA, 2013, pp. 139–144.

[25] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, T. Kim, Cab-fuzz: Practical concolic testing techniques for COTS operating systems, in: 2017 USENIX Annual Technical Conference (USENIX ATC 17), USENIX Association, Santa Clara, CA, 2017, pp. 689–701.

[26] H. Greg, J. Butler, Rootkits - Subverting the Windows Kernel, Addison-Wesley, 2005.

[27] S. Kim, J. Park, K. Lee, I. You, K. Yim, A brief survey on rootkit techniques in malicious codes, Journal of Internet Services and Information Security 2 (3/4) (2012) 134–147.

[28] E. Rudd, A. Rozsa, M. Gunther, T. Boult, A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world

solutions, IEEE Communications Surveys & Tutorials 19 (2) (2017) 1145–1172.

[29] H. Shi, J. Mirkovic, Hiding debuggers from malware with apate, in: Proceedings of the Symposium on Applied Computing, SAC '17, ACM, New York, NY, USA, 2017, pp. 1703–1710.

[30] T. Shields, Anti-debugging - a developers view, technical report, veracode (2009).

[31] P. Ferrie, Attacks on virtual machine emulators (2011).
URL \url{http://pferrie.host22.com/papers/attacks.pdf}

[32] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes, et al., Sandprint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, Springer International Publishing, Cham, 2016, pp. 165–187.

[33] A. Issa, Anti-virtual machines and emulations, Journal in Computer Virology 8 (4) (2012) 141–149.

[34] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, S. Ioannidis, Rage against the virtual machine: Hindering dynamic analysis of android malware, in: Proceedings of the Seventh European Workshop on System Security, EuroSec '14, ACM, New York, NY, USA, 2014, pp. 5:1–5:6.

[35] J. Uitto, S. Rauti, S. Laurén, V. Leppänen, A survey on anti-honeypot and anti-introspection methods, in: World Conference on Information Systems and Technologies, Springer, 2017, pp. 125–134.

[36] C. S. Veerappan, P. L. K. Keong, Z. Tang, F. Tan, Taxonomy on malware evasion countermeasures techniques, in: 2018 IEEE 4th World Forum on Internet of Things (WF-IoT), IEEE, 2018, pp. 558–563.

[37] A. Afianian, S. Niksefat, B. Sadeghiyan, D. Baptiste, Malware dynamic analysis evasion techniques: A survey, arXiv preprint arXiv:1811.01190.

[38] Checkpoint Research, Evasion techniques, `https://evasions.checkpoint.com/` (2020).

[39] D. Kirat, G. Vigna, Malgene: Automatic extraction of malware analysis evasion signature, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, ACM, New York, NY, USA, 2015, pp. 769–780.

[40] H. Shi, J. Mirkovic, A. Alwabel, Handling anti-virtual machine techniques in malicious software, ACM Transactions on Privacy and Security (TOPS) 21 (1) (2017) 2:1–2:31.

[41] Y. Leguesse, M. Vella, J. Ellul, Androneo: Hardening android malware sandboxes by predicting evasion heuristics, in: IFIP International Conference on Information Security Theory and Practice, Springer, Springer International Publishing, Cham, 2017, pp. 140–152.

[42] D. Kirat, G. Vigna, C. Kruegel, Barebox: efficient malware analysis on bare-metal, in: Proceedings of the 27th Annual Computer Security Applications Conference, ACM, ACM, New York, NY, USA, 2011, pp. 403–412.

[43] L. Guan, S. Jia, B. Chen, F. Zhang, B. Luo, J. Lin, P. Liu, X. Xing, L. Xia, Supporting transparent snapshot for bare-metal malware analysis on mobile devices, in: Proceedings of the 33rd Annual Computer Security Applications Conference, ACM, ACM, New York, NY, USA, 2017, pp. 339–349.

[44] D. Kirat, G. Vigna, C. Kruegel, Barecloud: Bare-metal analysis-based evasive malware detection, in: USENIX Security Symposium, USENIX Association, Berkeley, CA, USA, 2014, pp. 287–301.

[45] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, G. Vigna, Baredroid: Large-scale analysis of android apps on real devices, in: Proceedings of the 31st Annual Computer Security Applications Conference, ACM, ACM, New York, NY, USA, 2015, pp. 71–80.

[46] X. Deng, J. Mirkovic, Malware analysis through high-level behavior, in: 11th USENIX Workshop on Cyber Security Experimentation and Test (CSET 18), USENIX Association, Baltimore, MD, 2018.

[47] D.-P. Pham, D.-L. Vu, F. Massacci, Mac-a-mal: macos malware analysis framework resistant to anti evasion techniques, Journal of Computer Virology and Hacking Techniques (2019) 1–9.

[48] Q. Do, B. Martini, K.-K. R. Choo, The role of the adversary model in applied security research, Computers & Security 81 (2019) 156–181.

[49] P. Ferrie, The "ultimate" anti-debugging reference, `http://pferrie.host22.com/papers/antidebug.pdf` (2011).

[50] Microsoft, Pe format - the .tls section, `https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format#the-tls-section` (2018).