Research Summary of Anti-debugging Technology

Bowen Zhang*
Capital Normal University Information Engineering College, Beijing, China

*Corresponding author e-mail: zhangbowen@ict.ac.cn

Abstract. Software protection technology has always been the most powerful weapon against malicious reverse engineering, and anti-debugging technology is the top priority among them. This article will give a detailed introduction to the anti-debugging technology, classify it according to the implementation principle, discuss the different execution strategies in each category. Finally, summarize the anti-debugging technology from the three perspectives of complexity, resistance and universality, and analyze their respective advantages and disadvantages.

Keywords: Software Protection, Anti-debug Technology, Reverse Engineering

1. Introduction
Software protection technology refers to the technology that protects the core code and confidential data of the software from being stolen and used in a malicious environment, and the software can operate normally within the scope of authorization. Software protection technology is mainly designed for anti-reverse engineering, including: code obfuscation, packing technology, authorization code verification, anti-debugging technology, etc. Anti-debugging technology plays an extremely important role in preventing software dynamic analysis.

The first part of this article mainly introduces anti-debugging technology and its concepts. Section 2 to section 4 introduce the anti-debugging technology in detail according to the category. Section 5 compares various anti-debugging techniques according to different dimensions, and finally is a review and summary of the entire article.

2. Overview
Anti-debugging technology is mainly used in the field of anti-reverse engineering to detect whether the debugger exists or make the debugger invalid. The purpose is to prevent reverse engineering personnel from cracking the software during the life cycle of the software operation, or to increase the difficulty of reverse engineering as much as possible and extend the analysis time.

According to the principle of anti-debugging technology, anti-debugging technology can be divided into static anti-debugging and dynamic anti-debugging. Static anti-debugging acts on the entire program, and the system state is checked before the program runs to determine whether the debugger exists. Static anti-debugging technology is highly dependent on the operating system, and most methods only take effect for a specific system. Dynamic anti-debugging focuses on observing whether the execution process and running state of some codes are normal during the running of the program. It is a kind of anti-debugging technology based on indirect inference.
The static anti-debugging technology has a simple principle and low implementation difficulty, but it is easy to be cracked. The dynamic anti-debugging technology is judged according to the running state of the program, which is more concealed and more difficult to crack, but compared to the static anti-debugging technology, it is more complicated to implement.

3. Static Anti-Debugging Technology

Static anti-debugging technology refers to some technologies that use the application program interface provided by the operating system or some information during program runtime to directly detect whether a debugger is attached to the process [1]. Static anti-debugging technology is often detected before the core code of the program runs, and it acts on the entire program.

3.1. Function Interface

Using API interface is the most common and direct static anti-debugging technique. The Windows operating system has many API interfaces that can be used to detect whether a debugger exists [12].

The IsDebuggerPresent()[15] function can determine whether the current process is being debugged by a user-mode debugger. If the current process is being debugged, this function returns a non-zero value, otherwise it returns zero.

The CheckRemoteDebuggerPresent()[25] function is similar to the IsDebuggerPresent() function, which detects whether the specified process is in the debugging state. This function has two parameters. hProcess represents the handle of the process to be detected, and pbDebuggerPresent points to a bool type variable. When the process represented by hProcess is in the debugging state, the function sets this variable to true, otherwise it is false.

The function of NtQueryInformationProcess()[13] is to read information about a specific process. Run this function when ProcessHandle is set to the handle of the current process, and ProcessInformationClasss is set to ProcessDebugPort (7). If the return value is non-zero, it means that the current process is being debugged by the debugger running in user mode.

3.2. Manual Inspection

Although using APIs is the most direct way to detect debuggers, these APIs will also be used by people who study anti-anti-debugging. These functions can be invalidated or return errors through API HOOK [10] or Rootkit technology. Even some APIs, such as NtQueryInformationProcess(), no longer provide support in subsequent versions of the Windows operating system. Another more common practice is to write code by hand to directly read the PEB [11], achieving the same purpose as the API [1,4].

PEB exists in the address space of each process, and it contains all user mode parameters related to this process [14]. Many attributes in this structure can be used as the basis for judging the debugger [21]. In Microsoft's official documents, most of the fields of the PEB structure are reserved. Among the currently published fields, the BeingDebugged field can be used as a basis for judgment. When the program is in the context of the debugger, the value of BeingDebugged is 1, otherwise it is 0. In addition to the BeingDebugged field, ProcessHeap and NTGlobalFlag can also expose debugger information, and the latter two belong to reserved fields in the PEB structure. In Windows 32-bit operating system, you can get the PEB structure through fs: [0x30h]. The ProcessHeap field is located at offset 0x18h of the PEB structure, indicating the address of the first heap allocated by the loader for the process. The header of the first heap has an attribute indicating whether the current heap is created by the debugger; NTGlobalFlag is located at the offset 0x68h of the PEB structure If NTGlobalFlag is set to 0x70, it means that it is currently in the debugging state.

3.3. Using-trace Detection

From installation to use, the debugger will leave information in many places of the operating system, such as the registry, file system, window name, etc [2]. Anti-debugging can be achieved by detecting these traces.
Find or list the names of all top-level windows by calling FindWindow() and EnumWindows() functions, and filter whether there is a debugger window opened [7]. Call the FindFirstFile() function to find a specific file in the file system to determine whether the debugger is installed.

3.4. Parent Process Detection
By default, the parent process of the executable file run by double-clicking the mouse or by the command line is explorer.exe. There are many implementation methods based on parent process detection, such as directly detect whether the parent process is a debugger process or explorer.exe [3, 9]. Or check whether certain fields in the STARTUPINFO structure are zero [16].

Explorer.exe uses ShellExcute() to start a program. The ShellExcute() function will set unused values zero in the STARTUPINFO structure. When the debugger starts a program, the structure is generally ignored. Detecting whether a process has SeDebugPrivilege permission can also determine whether it is currently being debugged. If a program is started by the debugger, the process will inherit the SeDebugPrivilege permission. You can judge whether a program has SeDebugPrivilege permission by opening the csrss.exe program.

4. Dynamic Anti-Debugging Technology
The static anti-debugging detection method is simple and widely used, and can easily be bypassed by the debugger forged data structures. The dynamic anti-debugging technology acts on the program during the running period, which is more concealed and more difficult to prevent, compared with the static anti-debugging technology.

4.1. Breakpoint Detection
In the process of software debugging, debuggers often use breakpoints to stop the program at a certain instruction or statement. The existence of breakpoints greatly facilitates the debugger to analyze the execution flow of the program, but it also leaves important evidence to prove the existence of the debugger [5]. According to the nature of breakpoints, breakpoint detection is divided into software breakpoint detection and hardware breakpoint detection.

In the x86 system, INT 3 (0xCC) is an instruction with interrupt number 3, which is only one byte in length and is often used as a software interrupt instruction. The principle of the breakpoint is: when the user breaks a certain instruction, the debugger will replace the first byte of the instruction with 0xCC, when the program runs to the breakpoint, an exception will be triggered, and the operating system will execute No. 3 interrupt service subroutine. The operating system will determine whether there is a debugger currently, and if so, the operating system will hand the exception to the debugger for processing. When the program wants to continue running, the debugger will replace the INT 3 instruction with the original instruction [8].

A very direct method of anti-debugging is to scan the code segment during program execution to see if any instructions are modified to 0xCC. Or the same can be achieved by comparing the checksum. Calculate the checksum of part of the code while the program is running. If the newly generated check value does not match the original value, the program should be terminated immediately.

In addition to software breakpoints, the operating system provides debugging registers to support hardware breakpoints. Among them, DR0-DR3 are used to set hardware breakpoints, the Dr4 and Dr5 registers are reserved by the system, and the Dr6 and Dr7 registers store debugging information. When the hardware breakpoint is not set, the values of DR0, DR1, DR2, and DR3 are all zero. You can check whether the values of these 4 registers are zero at the same time to determine whether the program has hardware breakpoints.

4.2. Time Interval Detection
The anti-debugging technology based on clock detection infers whether it is in the debug state by calculating the running time. Because the existence of single step debugging and breakpoints will
prolong the running time of the program, if the running time of the program exceeds the preset threshold, then there is reason to think that there is a debugger.

There are generally two ways to calculate the running time, using the API provided by the system, such as GetTickCount, QueryPerformanceCounter or rdtsc assembly instructions [17]. The GetTickCount() function returns the number of milliseconds that have passed since the last restart to the execution of the function. Put the GetTickCount() function at the beginning and end of a program. The difference between the results of the two function calls is the running time of this operation. The principle of the rdtsc instruction is similar, but the precision and accuracy are higher than the system API.

4.3. Exception Detection
Whether the process can correctly handle exceptions can also be used as a means of detecting the debugger. In the absence of a debugger, when an exception is triggered during program execution, the operating system will receive the exception and call the callback function registered in the SHE [18]. However, when there is a debugger, the debugger will be superior to the operating system to obtain exception handling authority. If the debugger cannot correctly return the exception processing result to the debugged process, then this abnormal failure can be detected by the internal mechanism of the process.

There are many ways to trigger exceptions, such as using INT 3, INT 2D instructions to trigger interrupt exceptions, using ICEBP instructions or setting the trap bit of the EFLAGS register to 1 to trigger single-step exception, and using the RaiseException() function to trigger multiple types of exceptions.

5. Advanced Anti-Debugging Technology
Static anti-debugging technology and dynamic anti-debugging technology are used to directly detect or look for evidence to prove the existence of the debugger, and another more aggressive approach is when the debugger is found to exist, the program tries to hide itself, or prevent the debugger from attaching or even Exploit the debugger vulnerability to make the debugger crash, unable to continue debugging the program. Next, several anti-debugging techniques that interfere with the debugger will be introduced.

5.1. Control Flow Manipulation
In simple terms, control flow manipulation is the use of some means to change or hide the execution thread, making it difficult for the debugger to find or attach to the execution thread. Specifically, it is divided into hidden thread, suspended thread, multi-threaded operation, self-debugging and LTS callback technologies [6].

Hidden thread refers to hiding the thread of execution, making it difficult for the debugger to find, thereby preventing debugging. You can use the NtSetInformationThread() function to set the HideThreadFromDebugger field of the RTHREAD kernel data structure to realize thread hiding.

Suspending a thread means that the debugger process calls SuspendThread() or NtSuspendThread() to suspend the debugger thread, but this method only works for debuggers running in user-mode [22].

Multi-threaded execution uses the CreateThread() function to place the core code part of the program for execution in other threads outside the debugger, making it difficult for the debugger to track.

The principle of self-debugging technology is that a process can only connect to one debugger at a time [23]. The software first copies itself as a debugger and attaches it to the process before running, so as to prevent the real debugger from debugging the program and achieve the purpose of anti-debugging. Use DbgUiDebugActiveProcess() or NtDebugActiveProcess() to implement this technique.

5.2. Debugger Vulnerability
There are vulnerabilities in any program, so do debuggers. Discovering and reasonably using debugger vulnerabilities may cause the debugger to crash or the program to get rid of the debugger's control. Some vulnerabilities only appear in specific debuggers, so this technique is not universal. Here are a few vulnerabilities in the OllyDBG debugger [19].

PE format vulnerability. The executable file format in the Windows operating system is the PE file format, which is composed of header information and each section block. The IMAGE_OPTIONAL_HEADER structure is a part of the PE header. The end of the structure is the DataDirectory array, which is composed of the IMAGE_DATA_DIRECTORY structure. NumberOfRvaAndSizes represents the number of elements in the array. Because the size of the DataDirectory array is determined by the IMAGE_NUMBEROF_DIRECTORY_ENTRIES field, its value is 16. Therefore, if the value of NumberOfRvaAndSizes is greater than 16, it will be ignored by the loader. OllyDBG also follows this standard, so setting the value of NumberOfRvaAndSizes to any number greater than 16 will make OllyDBG unable to load the executable file correctly and exit early.

OutputDebugString vulnerability. There is a format string vulnerability in OllyDBG1.1. If the %s parameter representing the string format is passed to the OutputDebugString() function, it will cause OllyDBG to crash [20].

6. Comparison
This section will summarize the anti-debugging techniques introduced above, and evaluate it from three aspects: Complexity, Resistance, and Pervasiveness [24].

| Classification                  | Tactic                  | Method                                                                                           | Complexity | Resistance | Pervasiveness |
|---------------------------------|-------------------------|-------------------------------------------------------------------------------------------------|------------|------------|---------------|
| Static anti-debugging technology| Function interface      | IsDebuggerPresent() CheckRemoteDebuggerPresent() NtQueryInformationProcess()                     | Low        | Low        | High          |
| Manual inspection               | Read the BeingDebugged, ProcessHeap, NTGlobalFlag fields of PEB                                    | Low        | Low        | High          |
| Using-trace detection           | Check the registry      | GetCurrentProcessId() + CreateToolhelp32Snapshot() + (Process32First()) + Process32Next()         | Medium     | Medium     | Low           |
| Parent process detection        | Scan INT 3 instruction  | Low                                                                                             | Low        | Low        | High          |
| Dynamic anti-debugging technology| Breakpoint detection    | GetTickCount(), QueryPerformanceCounter(), rdtsr instruction                                   | Low        | High       | Medium        |
| Interval detection              | Read DR registers       | NtQueryInformationProcess()                                                                 | Low        | High       | Medium        |
7. Conclusion
This article expounds the principle of anti-debugging technology, classifies anti-debugging technology, introduces the specific application of different types of anti-debugging technology in detail, and analyzes various anti-debugging technologies from three aspects: complexity, resistance and universality. From the comparison results, it can be seen that a single anti-debugging technology is difficult to achieve the desired effect. The relevant technical staff should comprehensively use multiple anti-debugging technologies and even combine anti-virtual machine, anti-tampering, code obfuscation and other anti-reverse methods to provide software comprehensive protection.

Acknowledgments
First of all, I am very grateful to my mentor, Professor Wu, who taught me many things and made me achievements in this field. Secondly, thank my alma mater for providing quality educational resources. Finally, I want to thank Dr. Tang. Without his encouragement and support, I would not have been able to finish this article.

References
[1] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, Jose Nazario. Towards an understanding of antivirtualization and anti-debugging behavior in modern malware. In IEEE International Conference on Dependable Systems and Networks With FTCS and DCC. IEEE, 2008, 177-186.
[2] Nguyen Anh Quynh, Kuniyasu Suzaki. Virt-ice: Next-generation debugger for malware analysis. Black Hat USA (2010).
[3] H. Mourad. Sleeping your way out of the sandbox. SANS Security Report, 2015.
[4] Michael Sikorski, Andrew Honig. Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software. No Starch Press, 2012.
[5] Michael Chourdakis. Toggle Hardware Data/Read/Execute Breakpoints Programmatically. Received October 2020 from https://www.codeproject.com/Articles/28071/Toggle-hardware-data-read-execute-breakpoints-prog.
[6] Hao Shi, Jelena Mirkovic. Hiding debuggers from malware with apate. In Proceedings of the Symposium on Applied Computing. ACM, 2017, 1703-1710.
[7] Tyler Shields. Anti-debugging—A developers view. Veracode Inc., USA (2010).
[8] Kulchytskyi Oleg. Anti-Debug Protection Techniques: Implementation and Neutralization. Retrieved October 2020 from https://www.codeproject.com/Articles/1090943/Anti-Debug-Protection-Techniques-Implementation-an.
[9] P. Ferrie. The Ultimate Anti-Debugging Reference. Retrieved October 2020 from https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf.
[10] Apriorit. Anti Debugging Protection Techniques. Retrieved October 2020 from https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-
use-before-releasing-software.

[11] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies, Black Hat (2012).

[12] Microsoft msdn. Debugging Functions. Retrieved October 2020 from https://docs.microsoft.com/zh-cn/windows/win32/debug/debugging-functions.

[13] Microsoft msdn. NtQueryInformationProcess function. Retrieved October 2020 from https://docs.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntqueryinformationprocess.

[14] Microsoft msdn. PEB structure. Retrieved October 2020 from https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb.

[15] Microsoft msdn. IsDebuggerPresent function. Retrieved October 2020 from https://docs.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent.

[16] Microsoft msdn. STARTUPINFOA structure. Retrieved October 2020 from https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/ns-processthreadsapi-startupinfoa.

[17] Microsoft. Acquiring High-Resolution Time Stamps. Retrieved October 2020 from https://docs.microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps.

[18] Microsoft. Structured Exception Handling. Retrieved October 2020 from https://docs.microsoft.com/en-us/windows/win32/debug/structured-exception-handling.

[19] Mark Vincent Yason. 2007. The Art of Unpacking. Retrieved October 2020 from https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf.

[20] Rami Siwail, Khairuddin Omar, Khairul Akram Zainol Ariffin. A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis. International Journal on Advanced Science, Engineering and Information Technology, 2018, 8(4-2):1662-1671.

[21] Lim, Charles, Sulistyan, Darryl Y, Suryadi, et al. Experiences in Instrumented Binary Analysis for Malware. Advanced Science Letters, 2015, 21(10):3333-3336(4).

[22] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Black Hat(2012).

[23] XPN.Windows Anti-Debug Techniques – OpenProcess Filtering. Retrieved October 2020 from https://blog.xpnsec.com/anti-debug-openprocess/.

[24] Afianian, A., Niksefat, S., Sadeghiyan, B., Baptiste, D. Malware Dynamic Analysis Evasion Techniques: A Survey. arXiv preprint arXiv:1811.01190 (2018).

[25] Microsoft msdn. CheckRemoteDebuggerPresent function. Retrieved October 2020 from https://docs.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-checkremotedebuggerpresent.