Irreversible Applications for Windows NT Systems

Rahul Sankalana Gunawardhana[1][a] and Kavinga Yapa Abeywardena[2][b]
[1]*Faculty of Graduate Studies and Research, Sri Lanka Institute of Information Technology, Colombo, Sri Lanka*
[2]*Faculty of Computing, Sri Lanka Institute of Information Technology, Colombo, Sri Lanka*

Keywords: Anti-reverse Engineering, Anti-cheat, Third Party Monitoring, Anti-debug, Windows NT.

Abstract: Anti-reversing or anti-debugging mechanisms refer to the implementations put in place in an application that tries to hinder or completely halt the process of debugging and disassembly. The paper discusses the possibility of a monitoring system that would prevent any debugger from debugging a given process in a Windows NT environment. This project aims to facilitate a similar concept present in that of anti-cheat monitoring programs in online games for commercial products and applications. In contrast, an anti-cheat product monitors the game's memory pages for direct or indirect modifications either via internal (within the process) mechanisms such as hooks and DLL injections or external mechanisms such as Read Process Memory (RPM), Write Process Memory (WPM), named pipes, sockets. In many other scenarios, the anti-debug program would monitor a selected process for attempts of debug or disassembly.

1 INTRODUCTION

De-compilation and debugging of commercial software or software in general that are not released under the GNU GPL/Open-Source Category prohibits any person from altering the code at any given level of execution. Thus, this is difficult to achieve considering the technological advancements at present. Free and Open-Source debugging and de-compilation tools have led to widespread knowledge in the reverse engineering area making the protection of source codes at commercial level almost impossible. For this reason, experts have utilized many techniques over the years to mitigate the issue of de-compilation and debugging using various techniques. Some are provided by the API of an OS itself while some get developed when observing the internal changes present when a process of debugging or decompilation starts. The latter is effective for the majority of cases since these changes are undocumented and therefore bypasses for these mechanisms rarely exist.

Most modern techniques of reverse engineering rely mainly on the operating system and its provided API functions to prevent debug and disassembly. However, this is inadequate and can be bypassed easily using simple patches.

For the Windows environment, the papers (Canzanese, 2012) and (Marpaung, 2017) discuss the possibilities of mitigating the issue of disassembly and debugging through API (Application Programming Interface) functions of Windows and internal Windows structures accessed by those functions. However, these could be easily mitigated or bypassed using suitable mechanisms as stated in (Canzanese, 2012) since most of them are documented and are available publicly.

Exploits or vulnerabilities within applications are concerned as the main medium of entry into gaining unauthorized control over a system. Unlike social-engineering which relies on the weaknesses of human beings, these take a more technical aspect as an attacker would have to identify the vulnerabilities and come up with ways to exploit them without triggering any detection mechanisms implemented in the system. The way an exploit is developed for a particular application or software is first by analyzing it for its weaknesses. These weaknesses can be in the form of stack or buffer overflows, dangling pointers, weak references and many other scenarios. The finding of these types of vulnerabilities were made

[a] https://orcid.org/0000-0001-6072-8544
[b] https://orcid.org/0000-0001-7089-1476

Gunawardhana, R. and Abeywardena, K.
Irreversible Applications for Windows NT Systems.
DOI: 10.5220/0011051700003283
In *Proceedings of the 19th International Conference on Security and Cryptography (SECRYPT 2022)*, pages 479-484
ISBN: 978-989-758-590-6; ISSN: 2184-7711
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

possible due to the ease of reverse engineering an application. However, if there exists any possibility to make that process extremely complex or even impossible, then the surfacing of such exploits would be rendered extremely unlikely.

The idea for this mainly comes from online gaming and their anti-cheat systems. Anti-cheats are third party software present in a computer that monitors and detects arbitrary modification to the process's memory, execution flow or its file components. These detect anomalies affecting the games memory by having a constant memory scan and a look up for different programs by using techniques such as signature detection, page hooks, mini filters, stack walking etc.

2 BACKGROUND AND RELATED WORK

2.1 What Is an Anti-cheat?

In this discussion, the main focus would be regarding client-sided anti-cheats in which the general concept of this paper was derived from.

2.1.1 Server-sided

These types of anti-cheats are implemented at the server. The purpose of them is to verify the data that is being sent to the server and to perform analysis on player statistics (Peter Laurens∗, 2007) in order to determine whether they are cheating or not. This is not the only type of server-sided anti-cheat that is but currently, the most advanced type of implementation that can happen at a server and this generally happens using rules. One advantage of using this type of anti-cheat implementation is that the typical bypassing methods relevant to client-sided anti-cheats are useless since an attacker has less access to a server than they would to a physical computer.

2.1.2 Client-sided

This concept is the main focus area on this paper. There are mainly two types of client-side anti-cheats as kernel and user-mode but these will not be discussed in detail due to the fact that it generally refers to the privilege level that the anti-cheat is operating on. Client-side anti-cheats are basically programs that monitor the system state and the game state at the client's end. By system state it means the enumeration of system processes, handles, loaded modules and memory. More advanced systems also perform integrity checks on both system and game files in both live and stored memory.

To summarize, a well-established anti-cheat consists of several modules that work together. In kernel level anti-cheats these usually consist of a Windows service, driver and a detection module that gets mapped into the client process(game) at runtime. However, in the user-mode only the driver is absent. The use of having a driver is that it blocks user-mode attempts at gaining access to the targeted process using functions such as OpenProcess, WriteProcessMemory, SetWindowsHookEx etc. These are achieved using minifilters and preventive callbacks (ObRegisterCallbacks) at the kernel level. To achieve this, the anti-cheat drivers, services and the detection modules need to get mapped into the client (game) file before the client execution starts. This could be achieved using TLS Callbacks. One main advantage of these types of callbacks is that they happen before the OEP (Original entry point) is called. Usually these anti-cheats have a 'heartbeat'. This is in the case that someone simply terminates the anti-cheat process while the client is still running. Having a heartbeat prevents the execution of the client when the anti-cheat is not present. The heartbeat module has to be in both the anti-cheat and the client and has to have a regular challenge-response type communication either via named pipes, sockets, mapped files etc, to determine whether or not both of them are running

A new layer of complexity can be introduced into reverse engineering an application if this concept of anti-cheats can be established within the general software development. This would not only mean that the reverse engineer would have to know about the anti-debug and anti-disassembly mechanisms present within the binary itself but should be able to circumvent a monitoring system that is capable of detecting debuggers via different methods.

3 METHODOLOGY

3.1 Initial Prototype

The initial prototype was developed in order to assess the effectiveness of the anti-debug mechanisms currently present. Even though these were not explicitly used in the final monitoring system, these will be used by the self-defense mechanisms present within both the monitoring system and the mapped DLLs that was made afterwards.

The formatting and the general organization of the prototype version was not considered to be important

as it was primarily done in order to demonstrate and analyze the effectiveness of the current anti-debug mechanisms. Credits to referenced authors and source code are given in the source code itself. There are several routines that can be considered as main components in this iteration of the implementation which will be discussed below. The prototype was developed to be able to be used as either a TLS callback or injected as a DLL.

3.1.1 Program Execution Flow

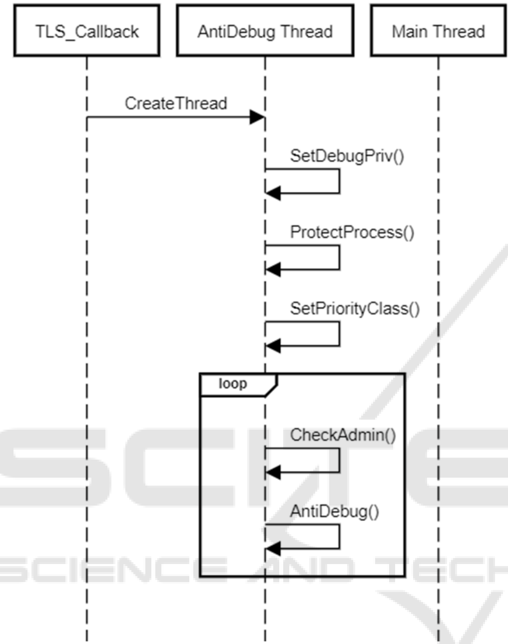

Figure 1: Prototype Execution Flow.

3.2 Final System

This system utilizes the main focus point of this discussion which is the concept of third party monitoring similar to that of anti-cheats present in online gaming leagues. Although the implementation specifics were detoured from the original concept idea which was taken from the BattlEye anti-cheat system, the base idea remains the same. The BattlEye system consisted of mainly 4 components which were at both the server side and the client side. Even though having a server side is the much more viable and fool-proof option, in this particular implementation the server side was not taken into consideration. The main components of this system are as follows,

1. MNS-Console
2. MNS-DLL
3. MNS-Driver

MNS-Console is a console application that is utilized as the third-party monitoring application. MNS-DLL component is the DLL that will get mapped into the process that is being protected by this system. MNS-Driver is the kernel level implementation that is responsible for stripping handle access and providing all the kernel level detection mechanisms. It is important to understand that all of these components are interconnected and use inter process communications (IPC) in order to determine that they are still running. This is where the heartbeat implementation comes into place. This ensures that no component is executing alone at a time but instead all are executing at all times. The main functionality of the heartbeat is to ensure the overall cohesion of the monitoring system.

https://github.com/sank20144/MNS-System.

3.2.1 Program Execution Flow

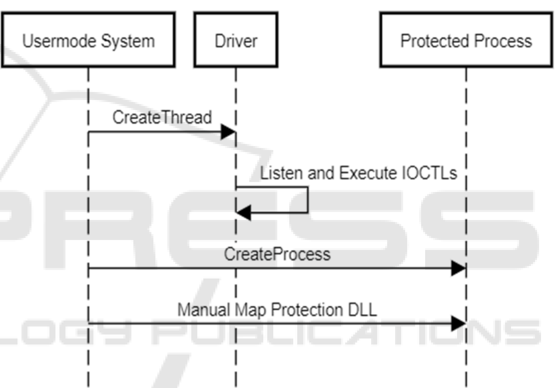

Figure 2: MNS Execution Flow.

3.2.2 Defects in the System

One of the main defects in the current implementation is the fact that it is difficult to determine whether a thread has been completely suspended. The system mainly relies on several threads in order to implement the desired functionality and each thread is hidden using an undocumented method, but it does not mean that it is not possible for an attacker to discover and access the particular threads. There are several restrictions in place to stop such an attempt such as the callbacks registered using the driver but it is possible to mitigate them using several methods as well. The operating system suspends certain threads periodically with regards to its scheduling algorithm. Any documented method of obtaining the state of a thread would produce false positives in this matter since it cannot determine whether the thread was suspended by the operating system or by an attacker.

A fix would be to monitor the threads for termination and not suspended state but it is possible to just set the thread to be suspended indefinitely without terminating it. A possible solution for this would be to check the suspended time on each thread and determine whether they have been suspended for an unusual amount of time, but the implementation of this solution would have to be specific since suspension times can vary due to various reasons from the operating system as well. In the current implementation, the only thread that would produce a detection vector when suspended would be the heartbeat thread since both the driver and the user mode application operates upon it. Another defect is present in the driver. Both the user mode application and the protected process utilize API functions and subroutines from the driver to enable their self-defence mechanisms but the driver does not implement such features. Due to this fact it is possible to simply unregister the callbacks registered by the driver if an attacker is able to map a driver of their own. The only method preventing rogue mapping of drivers is checking for test signing policy through the user mode application. However, it is possible to manual map drivers into kernel space without a valid certificate using methods such as through vulnerable drivers. The implemented heartbeat system is vague and could be improved upon. This is due to the fact that only the heartbeat thread is responsible for carrying out the communication between the driver. This means that the heartbeat thread could be terminated or simply suspended in order to bypass the entire implementation. A solution for this would be to create a system thread of its own through the driver and synchronize the heartbeat process through them. Another major weakness would be the fact that the monitoring system does not perform any integrity checks on itself and the files. An attacker would be able to patch system routines and monitoring system files in order to circumvent most of the detection mechanisms. The files include system files as well. Obtaining hash values for all system modules in each distribution of Windows would be a tedious and long task that would not be fruitful when achieving the ultimate goal of developing this system, which is to introduce a new method of approach to anti-debugging and anti-disassembly.

4 TESTING THE SYSTEM

In these scenarios, we will attempt to cloak the x32dbg debugger using various methods and attempt to debug a protected application. The effectiveness of the monitoring application will be decided on the following criteria.

1. Detection
2. Traceback
3. Termination – Not carried out

Detection refers to the ability of the monitoring application to detect any debugging attempts done to the protected application. Detection criteria will be triggered if Hooks, Suspicious handles to the process, presence of a dormant (installed but not running) debugger within the system, injected modules, manually mapped modules are discovered.

Suspicious handles are handles to the application created by untrusted processes. Untrusted processes refer to processes without a valid digital signature or forged signatures and processes containing (Microsoft, Debug Privilege, 2017) SE_DEBUG privileges. System processes are excluded from this detection vector since most of the system processes contain this privilege in latest Windows builds. Furthermore, registry values will be observed in order to determine whether certain analysis tools and processes are installed within a system. If a detection has occurred, the monitoring application would try to traceback to the process responsible. In the termination phase, the monitoring application would try to determine if the process obtained through the traceback is eligible to be terminated. The termination would occur in two separate ways by either terminating the process that triggered the detection or terminating the protected process itself. The latter method is used in case the monitoring application is unable to terminate the debugger. The monitoring program should only protect the protected application. It should not prevent debugging attempts done to other applications which are not protected by the monitoring system. Therefore, traceback and termination should not happen in the case of a debugger debugging another application which is not protected. The monitoring system is designed to traceback and log the triggered detection vector and the actual termination of the responsible process or tool is not carried out since it will be difficult to monitor the effectiveness of the system and log the required details. Therefore, a traceback is determined as a successful termination. One major issue during testing was the fact that since thread hiding is enabled it is difficult to attach a debugger to any application without it crashing since there are no visible threads in the process. A way to bypass this would be to byte patch the executable (timb3r, 2019). However, this would be impractical in most test scenarios therefore

the thread hiding feature was disabled during the testing phase for several of the tests that were carried out.

4.1 Test Cases

The following set of functions will be hooked to protect the debugger. All of the function hooks will be achieved using ScyllaHide. There are other plugins for x32dbg debugger that are used to hide the debugger but they all utilize at least one of these functions which are all available to be configured via ScyllaHide. Local hooks will be placed inside our predetermined application while global hooks will be attempted via SetWindowsHookEx. (fisherprice, 2020) VAD hiding will attempt to remove the debugging application from the current active process list. The following set of functions will be hooked individually and as groups at certain test cases. And finally, the debugger process will be hidden using VAD hiding and attempted to debug the protected application with the help of ScyllaHide and all of its functions.

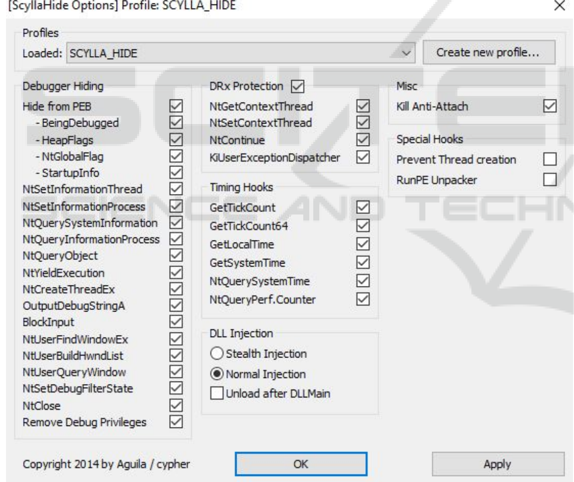

Figure 3: Hooked function list.

4.2 Test Results

The monitoring system and its capabilities were tested from a reverse engineer's perspective in order to determine its effectiveness and it yielded the expected outcome which was to detect, prevent or further complicate the process of debugging done to an application. The tests were carried out targeting several components of the system such as the user mode application and the protected process and the current design was able to prove that the system accomplishes its designated task in all of the testing environments. There were 20 test cases carried out trying to debug the protected process and 3 test cases trying to circumvent the monitoring system, this equates to 23 test cases carried out per test environment and a total of 115 test cases carried out across all the testing environments.

Out of the 115 test cases that were carried out only 1 test case produced negative results. This was the Test case 5.14. The test was to open the debugger using SE_DEBUG privileges and monitor for detection vectors triggered by the monitoring system. The monitoring system was unable to detect processes with SE_DEBUG privileges. However, it is safe to conclude that the monitoring system achieved its desired functionality across several builds of Windows operating system since 114 of the total 115 test cases were successful.

5 CONCLUSION

This research was carried out in inspiration of introducing a concept from the online gaming community to the reverse engineering discipline. The main problem that leads to this research is the debugging and de compilation of software not published under the GNU/GPL license. In short, every product or source code not published under this license prohibits unauthenticated reversing and decompilation of the particular software product. There are mechanisms built by both the operating system and software developers that can be used to detect and mitigate these events but they are not very effective. This can be clearly seen in the prototype implementation carried out in this research. It implemented all the known anti-debugging mechanisms present into an application that implements them but it presented many drawbacks within the system that could be easily known and abused by an experienced reverse engineer. There were 29 current anti-debug mechanisms discussed in this paper and all of them posed a cat and mouse game between the developer and the reverse engineer. As a simple example in order to show how the same tools, functions and utilities used by developers in order to implement anti-debugging features are used by reverse engineers to easily circumvent we can take the famous Windows API function IsDebuggerPresent. This function can be utilized in many different ways by the developer in order to detect debuggers for example by simply calling the function through the API, this would leave artifacts in the IAT of the process about the existence and the use of this function. A reverse engineer would then be

able to either byte patch the function call by finding it in the executable. Another way the developer could use this would be by calling its underlying system functions, which would be to simply check the Process environment block in the current process in order to detect the BeingDebugged flag. This could then allow a reverse engineer to simply change the value within the process environment block. This could go on for several more steps as to patching previously known detection vectors and introducing new ones but the ultimate prevention mechanism currently in place would be to incorporate many different detection mechanisms together in hopes that one of them might slip through the patching process. This was exactly what was achieved in the initial prototype but to no avail and it is the main reason for introducing the concept of anti-cheating in online games in order to prevent debugging and disassembly. This was achieved to a certain degree in the implemented monitoring system. The wording is very specific when saying to a certain degree because there are many ways this could be improved in order to achieve better and more consistent results. The monitoring system discussed in this paper introduced another variable to the debugging equation. Not only does a reverse engineer have to deal with protection mechanisms present in the protected process but also the monitoring system in order to successfully debug an application. The monitoring system initiation method implemented in the paper basically initiates the monitoring system and its drivers whenever the protected process is executing and that is one method of accomplishing it. A much better implementation would be to have the monitoring system load up at boot time as in the anti-cheat concept used by Riot Games for their new title Valorant (Dev, 2020). It is the best cheat free title to date and its anti-cheat implementation mechanisms should be observed and taken into consideration when further developing this system. More detection vectors can be introduced into this system by closely observing how debugger routines are carried out.

The implementation of this concept was strictly limited to 32-bit Windows Operating systems. All the development was done in a 32-bit Windows Environment along with all the testing. The testing proves that it can be persistent through several generations and versions of the Windows NT Operating system. However, this concept could be implemented in any operating system following the same outline. Another major objective would be to understand and implement the server component. In order to do that, first the operations that could be carried out server side should be identified and separately discussed. It would mitigate most of the problems discussed during the implementation process regarding the user mode and kernel mode heartbeat process. This is merely a stepping stone into the goals that could be achieved by using this concept properly in reverse engineering and the ultimate goal would be to develop a monitoring system with solid self-defence and detection mechanisms that would detect, prevent and further complicate the process of reverse engineering protected commercial applications.

REFERENCES

al, M. K. (2010). Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering. *IEEE International Symposium on Object/Component/ Service-Oriented Real-Time Distributed Computing.*

Canzanese, J. M. (2012). *"A Survey of Reverse Engineering Tools for the 32-Bit Microsoft Windows Environment.* ACM.

Dev. (2020, 06 05). *Valorant Anti-Cheat: What, Why, And How*. (Riot Games) Retrieved from https://playvalorant.com/en-us/news/dev/valorant-anti-cheat-what-why-and-how/

fisherprice. (2020, 08 13). *VAD Hiding*. (UnKnoWnCheaTs) Retrieved from https://www.unknowncheats.me/forum/c-and-c-/411299-vad-hiding.html

Marpaung, M. S.-J. (2017). *Survey on malware evasion techniques.* Busan: Graduate School of General, Dongseo University.

Microsoft. (2017, 05 23). *Debug Privilege*. (Microsoft) Retrieved from https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debug-privilege

Microsoft. (2017, 12 21). *HAL Library Routines*. (Microsoft) Retrieved from https://docs.microsoft.com/en-us/previous-versions/windows/hardware/drivers/ff546644(v=vs.85)

Microsoft. (n.d.). *Processes, Threads, and Jobs in the Windows Operating System*. Retrieved from Microsoft Press Store: https://www.microsoftpressstore.com/articles/article.aspx?p=2233328

Peter Laurens, R. F. (2007). *A Novel Approach to the Detection of Cheating in Multiplayer Online Games.* ICECCS 2007.

timb3r. (2019, 12 27). *How to Find Hidden Threads - ThreadHideFromDebugger - AntiDebug Trick*. (Guided Hacking) Retrieved from https://guidedhacking.com/threads/how-to-find-hidden-threads-threadhidefromdebugger-antidebug-trick.14281/