Cheating in Modern Computer Games: Methodologies and Prevention

FYP Final Report

Ronald Chan (3035374503)
4th April, 2021

Supervisor: Dr. Chim T.W.
The University of Hong Kong
Department of Computer Science
Abstract

In the day and age where competitive multiplayer online games are becoming increasingly popular, and billions of dollars are generated through game and in-game purchases together with eSports competitions and tournaments, it is of no surprise that the issue of cheating in online video games is an equally fast-growing problem.

This research project aims to shed light on the lesser known facet of cheating in gaming, by first classifying common game cheats into categories, in terms of their functionality, abilities, and development processes.

The project will then investigate the measures put in place by anti-cheat developers hoping to mitigate the issue, how they work, and how cheat developers keep improving to find new ways to bypass the existing defense measures. A Proof-of-Concept kernel-mode arbitrary Read/Write Driver will also be provided as a deliverable of this project.

Lastly, the project will end with an outlook on the future landscape and possibilities of cheating and anti-cheating.

Acknowledgements

I would like to express my gratitude and thanks to Dr. Chim for his guidance and advice on this project, and to my peers who have given me encouragement and pointers in completing this research.
Contents

Abstract and Acknowledgements 2

1. Background 5
   1.1 The Modern Gaming Industry
   1.2 The Video Game Cheating Industry

2. Objectives 6

3. Existing Research on Topic 7

4. Methodology 8
   4.1.1 Sources of information and steps in conducting project
   4.1.2 Foreseeable difficulties and limitations
   4.2.1 Classification of different existing types of cheats 9
   4.2.2 Common anti-cheat measures and bypasses
   4.2.3 The future of game cheating and anti-cheating 10

5. Section 1 - Classification of existing types of cheats 11
   5.1 Classification of existing types of cheats
      In-Game Exploits
      Network Manipulation Cheats 13
      Memory Manipulation Cheats 17

6. Section 2 - Common anti-cheat measures and bypasses 22
   6.1 Common Cheat Detection Methods
      Signature Scanning
      Driver Blacklisting
      Kernel Object Monitoring 25
   6.2 EventSyncDriver - An up-to-date kernel driver arbitrary R/W bypass module 32
      Obtaining one-time arbitrary execution of kernel code
      Kernel memory allocation, and hooking an existing kernel driver
      Calling the kernel hook from usermode, and restoring the hooked function


Execution and synchronization of Read/Write commands
given from usermode

7. **Section 3 - The future of game cheating and anti-cheating**  42
   7.1 Virtualization Technologies
   7.2 Machine Learning

8. **Conclusion**  45

9. **References**  46
1. Background

1.1 The Gaming Industry

In the past decade, due to the increase in the popularity and ease-of-access of the internet, the computer video games industry has transformed from mainly accommodating for single-player games, into focusing mainly on the production of multiplayer online games. In fact, according to Steam’s statistics on top games by current player count in September, 2020 [1], the top 5 games are all multiplayer online games, which shows how the video game industry has evolved into having a major focus in online multiplayer gameplay.

The incorporation of online multiplayer gaming has also brought in new commercial elements such as eSports, and the Free-to-play business model into the gaming market, which together contributes to the rapidly growing market of video gaming - the size of the global video gaming industry is estimated to be worth about 159 billion USD in 2020 [2], displaying how the video gaming industry is undoubtedly a massive and profitable one.

1.2 The Video Game Cheating Industry

Many online multiplayer video games share a common element - the emphasis in competition between players. Whether it is a competition for resources (such as in Real-Time-Strategy games like Starcraft 2), a competition to fight for winning a match (such as in round-based shooter games like Counterstrike:Global Offensive), or a literal fight for survival in battle royale games such as PlayerUnknown’s BattleGrounds, players are frequently put together in multiplayer online games to compete with each other, which is a crucial element in making the game engaging and addictive to players.

However, as with other types of games (football, basketball, horse-racing, etc.), with the desire to win against other players, or even the attractiveness of potential tangible rewards (such as reward money from winning eSports competitions), some players will
search for ways to cheat in online video games in order to gain an advantage over the others. Targeting the profitability of such a market opportunity, some programmers turn into cheat developers who develop dedicated pieces of software and sell to players at a price to gain an unfair advantage over others. The prevalence of cheating software in modern online video gaming is a major issue that game developers and legitimate players have to deal with daily.

2. Objectives
With the rapid growth of the competitive online gaming market, and inevitably together with the game cheating market, we believe that much of the information related to this topic is not adequately investigated and documented in the academic perspective, as much of the technical information about game cheating is scattered in different underground cheat forums and communities.

Furthermore, due to the inherent nature of “Security through obscurity” in anti-cheat development (Since a public disclosure of anti-cheating techniques is more than certain to allow for cheater developers to pin-point and bypass the exact techniques used in the future), there is even less incentive for anti-cheat developers to publicly release information on the current game cheating and anti-cheating environment.

This project aims to provide academic insight and documentation into the complex and lesser-known domain of game cheating in modern online video games, by laying out the classification of cheats in terms of how they function to obtain an advantage to players, together with a brief overview of common anti-cheat techniques used by game developers. The above will then be followed by an example of how a cheat developer would step-by-step develop a bypass method to overcome the measures put up by the anti-cheat, this will be in the form of a proof-of-concept arbitrary memory read/write windows kernel driver. Lastly, the project will end with an outlook on the future
developments and possibilities in game cheating and anti-game cheating, mainly touching on topics such as use of artificial intelligence and virtualization.

3. **Existing Research on Topic**

Perhaps due to the proprietary nature of both cheat development and anti-cheat development, formal academic research on the technicalities of game cheating is very scarce. For one to obtain information on the methodologies and techniques to actually develop a cheat or an anti-cheat for a game, is it often required to spend a substantial amount of time and effort to assemble many different pieces of information scattered across cheat forum posts/ blog articles. One fairly recent study on the technicalities of game cheating would be *Comparative Study of Anti-cheat Methods in Video Games* by Lehtonen in March 2020 [3], where the author discusses heavily on the practices that game developers should be adopting to better protect their games from cheats. The study gives valuable insight into game cheats from the perspective of an anti-cheat developer, and focuses mainly on the anti-cheating side of the “cheats versus anti-cheats” battle.

In turn, this study also hopes to provide insight into the game cheating world, however focusing more on the perspective of cheat developers - what technical methodologies they use, what issues and difficulties they encounter the most, how they methodically overcome those difficulties step by step, and how they try to always keep one step ahead of anti-cheat developers in the never ending tug-of-war of game cheating.
4. Methodology

4.1.1 Sources of information and steps in conducting project

The project will be conducted by obtaining information from 3 major sources:

1. Actual commercial cheats sold by cheat developers from game cheating forums
2. Documentation and techniques published publicly regarding game cheating on forums, and blog posts.
3. Formal academic literature regarding game cheating

Publicly available sample codes of game cheats will be analysed to give an overview of how they achieve their functionality in this project, and information about game cheats from internet forums and blogposts will be reviewed carefully to filter the valid and useful information out from the spam and false information that may be mixed within due to the anarchic nature of the public internet.

4.1.2 Foreseeable difficulties and limitations

Two difficulties and limitations in conducting this research can be foreseen.

Firstly, the validity of the methods in cheating and anti-cheating reviewed in this report may change substantially in the future. Due to the constant competition between cheat developers and anti-cheat developers in this field, it is very likely for cheating techniques or anti-cheating techniques discussed in this project that are valid during the writing of this project to be outdated in the near future, since it will be quickly patched or have new ways to be bypassed.

Secondly, the gathering and processing of cheating information from public forums and underground cheating forums has to be carefully filtered and organized. Much of the existing valid information on game cheating are scattered in different posts and comments of internet forums, where some are valid and some are misleading, which will have to be carefully screened and selected in this project.
4.2.1 Classification of different existing types of cheats

This section will classify existing video game cheats into 3 main categories in terms of how they exploit the game to gain an advantage for the cheating player, namely:

1. In-Game Exploits
2. Memory Manipulation Cheats
3. Network Manipulation Cheats

Through exploring commercial cheating product listings in underground cheating communities, and reviewing publicly available game cheating documentation from forums, this section aims to review and classify existing cheat types on the market into easily distinguishable categories, and explain the basic concepts of how each of the cheat types differ from each other in terms of 1.) the methods they use to achieve exploitation of the game; 2.) Difficulty in development and maintenance of the cheats of the category; 3.) Difficulty in patching and removing the category of exploits by the game developers.

4.2.2 Common anti-cheat measures and bypasses

This section will aim to provide better understanding on how the cheating behaviours in the previous section are commonly detected and reprimanded, and how cheat developers adapt to these cheat prevention methods to create bypasses for the existing anti-cheat methods.

A proof-of-concept windows kernel driver capable of arbitrary memory read/write through anti-cheat protection will also be included to demonstrate how a cheat developer would think and methodically implement bypass code to overcome the anti-cheat.
4.2.3 The future of game cheating and anti-cheating

Lastly, this study will conclude with possible future directions that cheat developers and anti-cheat developers may be heading towards, in order to raise their game to defeat each other.

This section aims to investigate how future developments in virtualization and AI technology can affect the landscape of game cheating, in particular, academic studies on using machine learning to detect game cheating will be reviewed, and proof of concept projects in cheat forums on using virtualization to bypass anti-cheat will be analysed.

Topics covered in this section will include:

- Virtualization technology
  - Direct Memory Access Cheating
  - Cloud Gaming

- Machine learning
  - Machine learning and cheating detection
  - Image Recognition and cheating
5. **Section 1- Classification of existing types of cheats**

Section 1 of the report summarizes the types of game cheats found in this research into 3 main categories based on the 1.) method of exploitation, 2.) the capabilities of the cheat, 3.) the level of difficulty of developing such a cheat, and 4.) the difficulty for game developers to detect and patch such an exploit.

**In-Game Exploits**

Bugs and unintended functionalities exist in almost every video game, and sometimes, these bugs can allow players to obtain an unfair advantage over other players in a way that the game developers did not intend it to be.

For example, a video game can have an in-game item called an Orange and an Apple, which are supposed to restore 10 health and 100 health respectively when eaten. However, due to a coding mistake, Oranges are now unintentionally restoring 1000 health to players when eaten instead. This creates a situation where most players will prefer to eat Oranges instead of Apples since they restore more health, which is an unintended in-game bug, and when exploited, becomes an in-game exploit.

Exploiting such bugs in a game usually requires no external/additional software for the cheating player, and often requires the player to perform specific sequential in-game actions in order to trigger the bug.

The capabilities of such in-game exploits can range from trivial things such as allowing a player to have a little bit more health compared to other players in a battle, or being able to access locations on a game map that is usually inaccessible - to something game-breaking such as using certain in-game character abilities in...
specific situations can crash and shut down the entire game server [4], ruining the gaming experience for all players in the session.

However, due to the wide accessibility of such in-game exploits, it is often reported quickly by players to developers, who then usually patches the more important or game-breaking exploits in a short timeframe, thus the longevity of in-game exploits are usually very short, especially if their capabilities affects players’ game experience a lot, since it will get more publicity, and in turn more quickly the developers will be able to pinpoint and hot-fix the exploit.
Network Manipulation Cheats

Since most online multiplayer video games utilizes a client-server model to deliver information between players’ computers and game servers, this introduces the possibility for unauthorized access and tampering of in-game information (such as enemy player’s health, location, etc.) during the transit of such data in the network.

![Diagram](image)

Figure 1: Transfer of data from game server to player

Figure 1 illustrates the typical process of how a game server sends information about a player’s health data as an example, inside a data packet, to the player’s computer through the internet, in order to be first saved in memory in the computer, and then processed by the game process into information for the player to view on screen.

As it can be observed from the illustration, the data packet containing game data must first pass through the public internet, then pass through the player’s home network.
network, before finally arriving into the player’s computer and then get stored and processed in the memory space of the computer. This gives the opportunity for one to view, or even manipulate the data packet before it arrives at the player’s computer. By packet manipulation, one can trick the game client into performing operations with false data, giving the player unfair advantages.

Figure 2 illustrates how by manipulating the game data packet containing the playerhealth variable (originally set as 10) when it enters the player’s home network (with a packet manipulation software such as WireShark, or by using another computer as a network proxy), one can change the value of the playerhealth variable contained in the data packet to become any arbitrary value (e.g. 99999).

Next, the packet will be accepted by the player’s computer as a normal game data packet, and the game process will read in the value 99999 from the manipulated game data packet and save it in computer memory as the player’s health, thus effectively tricking the game to recognize the player’s health as 99999 - an absurd arbitrary value, instead of the original 10.
Network manipulation cheats, once developed, usually require minimal maintenance effort from the cheat developer’s perspective since for most game updates, their networking protocols and game data packet structure rarely change from patch to patch, which gives networking manipulation cheats an advantage in terms of ease of maintenance.

Despite the seemingly endless possibilities achievable by packet manipulation cheats, the popularity of it in reality is not very high. This is due to the fact that:

1. Nowadays most if not all games apply end-to-end encryption between game process and game server to protect their data packets from being manipulated, making it incredibly difficult to break the encryption and do the manipulation.
2. Often game clients and game servers have validity checks in their networking protocols such that when game data packets are manipulated in transit, it will be detected and properly discarded (e.g. via a Checksum).

Network manipulation cheats do appear every now and then, for example in the popular hardcore realism first-person shooter game Escape From Tarkov, a packet-based radar cheat, which exploited the lack of encryption in the game’s networking system, allowed players to obtain the exact location of enemies in a match in real-time through intercepting game data inside network packets (This is a huge advantage for the cheating player in a highly realistic and competitive shooter game such as Escape from Tarkov) [5]. Despite the cheat being published in a public cheating forum, it took the game developers months to actually update their networking protocols and patch out the networking manipulation exploit [6]. This shows how network manipulation cheats are still relevant in today’s cheating landscape whenever game developers are careless about their choice in game networking design and security.
Memory Manipulation Cheats

Referring back to Figure 1 in the previous section, wary readers may observe that there is another process which all data must go through from game server to game client, which is the process of storing the game data in the player computer’s memory, and then reading the game data from the computer’s memory by the game process to display the information/ do calculations with the data. This presents another vector for cheat developers to exploit and manipulate game data, which is by manipulating game data stored in the computer’s memory space - which is the basis of memory manipulation cheats.

Memory manipulation cheats can be sub-categorised into two types, namely Internal Cheats and External Cheats.

Internal Cheats

Memory Manipulation Cheats are classified as “Internal” if they run inside the context of the game process, and reside inside the memory of the game process.

Figure 3 shows the typical virtual memory layout of a windows user-mode process, any thread that is owned by the process is able to manipulate the memory inside the stack and the heap, and use any code inside the process image and any Dynamically linked libraries (DLLs) linked with the process.
Most internal cheats make use of a custom DLL file containing all the cheating code and functionality, which gets loaded into the game process’s virtual memory space via a DLL injector. The code inside the DLL is then allowed execution via launching another thread in the process, or hijacking an existing thread.

Since all the cheating code resides in the virtual memory space of the game process, the cheating code is able to read or write any of the game data that reside in the stack/heap or directly call and use game functions that reside in the process image.

This provides a few benefits in terms of development for Internal Cheats:
1. Access to game data in memory is direct and has minimal overhead, allowing for minimal performance impact on the game or system when running the cheat.

2. Access to game functions can allow for easier development of cheats by calling the original game functions to obtain functionality or data required, saving the time needed to reverse engineer and replicate game functions from zero in the cheat code.

However, in terms of ease of maintenance, internal cheats has its own drawbacks, mainly:

1. When game updates occur, the structures and memory locations of in-game objects and functions frequently change, which requires internal cheats to be updated too to adapt to the game structure very often.

External Cheats

External Cheats are classified as “External” if the cheat code and functionality is executed by a separate process, and runs outside of the context and memory region of the game process.

Since memory manipulation cheats in essence only requires being able to read and/or write memory to the target game process to function as a cheat, its code does not necessarily have to reside inside the target game process’s virtual memory context.
Figure 4: Typical Architecture of an external cheat

Figure 4 illustrates the typical architecture of an external cheat, where the virtual memory of the game process itself is not injected of any cheat code, and all reading and writing to the game data in the stack or heap is performed by an external process which has read/write permissions to the game process memory. Game data is usually read to the external cheat process, where calculations and operations are performed to carry out cheating functions, then written back to the game process from the external cheat process if necessary.

In terms of maintenance, external and internal cheats share the same issue of having to be frequently updated to adapt to new game versions.
In terms of development, external cheats compared to internal cheats have the drawbacks of:

1. Increased overhead, since memory access is indirect, and requires a separate process on a separate thread to read and write the memory in the game process, which some developers may detest.
2. Inability to call built-in game functions, since external cheats do not reside in the game process’s memory context, they are unable to directly call in-game functions, which may prove to be inconvenient for some cheat developers.

However in terms of development, external cheats compared to internal cheats also have the benefits of:

1. Easier development process when bypassing anti-cheat measures. Since external cheats are a completely separate process from the game process, anti-cheats cannot as easily just scan the game process for the cheat module residing in memory, and is required to check all processes running in the system at the time instead, which adds a barrier against detection by anti-cheat software.
6. **Section 2 - Common anti-cheat measures and bypasses**

This section will be split into two sub-sections. In the first part, we will go through some of the common anti-cheating measures deployed by anti-cheat developers, with the idea and reasoning behind their implementations. In the second part, a Proof-Of-Concept Kernel Arbitrary Read/Write driver will be presented as a means to bypass kernel anti-cheat detection routines.

6.1 **Common Cheat Detection Methods**

**Signature Scanning**

One of the commonly used methods for detecting known cheating programs on a computer by an anti-cheat, is through signature/pattern scanning of running processes on a machine.

By identifying features such as 1.) a commonly used string in cheating programs such as “Aimbot”, or 2.) a unique function signature of a cheating program. By scanning for such signatures in the running processes of a machine (including the game process itself), an anti-cheat can identify known cheating software with varying degrees of certainty - At its best, signature scanning can directly confirm a cheating program is being used on the user machine, and at its worst, it can at least generate a flag for the user’s machine for potential of cheating, and notify the anti-cheat to proceed with further, more intense checks on the machine.

**Driver Blacklisting**

To understand the importance and capabilities of drivers in Windows, and why both Microsoft and Anti-cheat software developers go on great lengths to prevent potentially malicious drivers from being launched on a computer, we must first take a look at the differences between user mode and kernel mode in Windows.

In normal user mode processes on Windows, each of the processes’ memory spaces are isolated, and they do not have direct access to hardware nor memory addresses. When
user mode processes have to do anything that requires memory access or hardware access, it does so via the well-documented Windows API, which can be easily monitored.

In Windows kernel mode however, any executing code has complete access to the hardware, memory pages and addresses. In other words, code in kernel mode usually can do anything it wants, which makes it difficult even for a kernel anti-cheat to monitor and detect the existence of a kernel cheat module in the system, if the cheat module has gone through steps to hide itself, similar to how malware rootkits and bootkits hide themselves from anti-virus software.

Ever since the release of the 64 bit version of Windows Vista, Microsoft implemented the Kernel Mode Code Signing Policy [7], which required all kernel device drivers to be signed by one of the major Certification Authorities before they are allowed to launch.

With the release of Windows 10 build 1607, Microsoft took another bigger step to thwart malicious drivers, from that point on, all new Windows 10 drivers must be signed by one of the two accepted CAs by Microsoft, and then submitted directly to Microsoft to undergo further testing and then obtain a Microsoft signature before it can be ran by a regular Windows 10 installation.
As it can be seen, with the restrictions put up by Microsoft over the years, it is no longer possible to just create a random self-signed piece of malicious driver code, and expect Windows to accept it and launch it nowadays. However, even with the strict requirements for a kernel driver to be signed by Microsoft, one inevitable loophole still exists which allows hackers (and cheat developers of course) to inject arbitrary code into the kernel and execute it - by using vulnerabilities in signed kernel drivers.

Vulnerabilities and bugs exist in all the time code, and signed kernel drivers are of no exception. Some vulnerabilities in signed, approved drivers allow attackers to execute arbitrary pieces of kernel code, which basically allows attackers to run any code in the kernel despite not having a valid, signed kernel driver. Even if the vulnerabilities are known to the developers of the signed driver and to Microsoft, it is very unlikely that the driver’s certificate will be revoked, since Microsoft’s policy of trusting in deprecated
software make it such that they will only revoke a certificate if the certificate itself has vulnerability, not the software. Thus, it is very often that vulnerable drivers can be used as a proxy to exploit the Windows kernel system, without any response and repercussions from Microsoft.

Many cheat developers make use of a method called “Manual Mapping”, which makes use of a known, vulnerable signed driver, to manually map an arbitrary piece of kernel driver code into the kernel memory space, and obtain a one-time execution of the code. Common “Manual Mappers” include kdmapper [8], which exploits the vulnerable Intel Network Adapter Diagnostic Driver (iqvw64e.sys) installed together with some Intel network drivers; and drvmap [9], which exploits the vulnerable Capcom game driver (capcom.sys) installed with Capcom multiplayer games.

This ultimately leads us to the idea of Driver Blacklisting employed by many kernel anti-cheat developers, which tries to monitor what kernel drivers are loaded in a Windows system, and prevents the video game from launching/ take reprimanding actions against the player (such as banning the player account) who has vulnerable drivers installed in their system, due to the suspicion of potential cheating modules being loaded together with the vulnerable kernel driver module.

**Kernel Object Monitoring**

Even with the previously discussed Driver Blacklisting actions implemented by anti-cheat developers, it is often the case that the loading of vulnerable drivers and kernel cheat modules cannot be prevented by solely driver blacklisting, due to the fact that: 1.) There are too many signed but vulnerable kernel drivers that exist in the world, which some may even be unknown to the public. 2.) Some drivers may be known to be vulnerable, but are commonly required by legitimate software to function on the user computer, which anti-cheats cannot ban/reprimand the player for having the vulnerable driver running on the system solely.
Thus, anti-cheat developers commonly employ another strategy - Kernel Object Monitoring, to more accurately reach conclusions that the user has the potential to be running kernel cheating modules on their computers.

Kernel Object Monitoring monitors the existence and creation of commonly used kernel space objects by cheat programs such as threads, kernel caches, and device objects. By identifying these kernel objects being used in a suspicious or malicious way, kernel anti-cheat programs can more definitively reach a conclusion of whether a user is running kernel cheat programs on the system.

Two of the commonly monitored kernel objects by commercial anti-cheats are as follows:

1. **Device Objects**
   
   In Windows driver programming, all properly coded drivers that are to be loaded by Windows via official routines have to create a Driver Object, and most drivers will further create something called a Device Object in its initialization via the IoCreateDevice() routine, in order for it to be able to interact with much of the documented kernel functions and inter-process communication methods.

   If a vulnerable signed driver is being hijacked and used to load a kernel cheat driver, the kernel cheat driver may create a Device Object itself in the kernel space in order for it to function and interact with usermode programs (such as a game process) - this gives an opportunity for anti-cheat programs to identify the potential of a cheat driver being intentionally and maliciously injected into the user’s system.
The kernel anti-cheat program can iterate through all the registered Device Objects on the system, and check whether each of the Device Objects are backed by a valid driver image. Regular device objects will be backed by signed and legitimately loaded Driver Objects, with a valid image path (As can be seen in Figure 6).

Meanwhile, Device Objects created by hijacked vulnerable drivers are not backed by any driver images since they are just allocated an arbitrary kernel memory pool, and placed there. As we can see from Figure 7, the device object BadDriver is created through vulnerable driver hijacking, thus it is not backed by any valid
driver object, and therefore can be flagged by anti-cheats as potential cheating behaviour.

![BadDriver device object viewed in WinObjEx tool, and its DriverObject reference being non-existent](image)

**Figure 7:** BadDriver device object viewed in WinObjEx tool, and its DriverObject reference being non-existent

2. **System Threads**

Threads are essential for commands to be executed, and kernel cheat modules are of no exception, whether they are launched as a legitimate driver module, or launched via hijacking a signed driver and manually mapped into kernel memory space.
The Windows Kernel API provides the function `PsCreateSystemThread()` for drivers to launch a special system thread, which creates a thread that has no user-mode context, and runs only in kernel mode [10]. Normal device drivers call this function when they initialize, or when they have to handle Asynchronous Input/Output requests coming into the driver’s dispatch routine.

Kernel cheat modules that are manually mapped via hijacking a signed kernel driver will still have to obtain continuous execution in order to continuously Read/Write data to and from the game process, thus they make use of the `PsCreateSystemThread()` function to launch their own system thread, and achieve this purpose.

Anti-cheat developers realized this, and promptly implemented measures to detect such “rogue” system threads. Since the kernel cheat modules which are manually mapped are not backed by any valid driver images, if a system thread is created by them, the system thread’s start address will be located in kernel memory that is not backed by any valid driver images. Thus, anti-cheats use the following steps to identify the existence of such rogue threads:

1. Scan through all threads in the computer, and identifying all system threads via the `PsIsSystemThread()` function provided by the Windows Kernel API
2. Check the start address of each of the system threads, if they are not located in a valid address inside a valid, system registered driver module, flag the user’s computer for potential cheating behaviour.

Rogue system threads can be easily identified manually with tools such as Process Hacker 2 [11], which allows us to inspect information about a Windows system’s processes and threads in detail.

In Figure 8 below, we can see a normal system thread with ID 56 being recognized with its start address located in a valid driver module’s memory space.
Figure 8: A normal kernel thread with its start address located within a valid driver module’s address space (ntoskrnl.exe in this case)

However, in Figure 9 below, we can identify that thread with ID 4092 has a thread start address of 0xffffa684527f8030 (which is kernel space memory in 64 bit windows), thus allowing us to identify it as a system thread. Next, we will notice the start address does not belong in any valid driver module, as seen in the start module line in the image. Thus, we can conclude that the system thread is indeed a rogue thread (which is extremely rare to occur in regular, healthy Windows systems), and we can then flag the system for potential cheating behaviour.
Figure 9: A rogue kernel thread with its start address located in kernel memory space, but not inside by any valid driver module’s address space
6.2 EventSyncDriver - An up-to-date kernel driver arbitrary R/W bypass module

In order to demonstrate how a cheat developer may attempt to craft a piece of cheating software capable of bypassing the kernel detection routines put in place by anti-cheat developers as described in section 6.1, in the following section, a Proof-of-concept Kernel Arbitrary Read/Write Driver module, EventSyncDriver, that is injected via vulnerable signed driver hijacking will be demonstrated, together with its development steps complete code samples.

The program is divided into 2 major modules:

1. The Kernel Driver module EventSyncDrv.sys, which is responsible for kernel hooking, the actual memory read/writing, and initialization of Windows Events for synchronization with the usermode control process.

2. The usermode control process umctrlproc.exe, which is responsible for thread creation, issuing commands and communicating with EventSyncDrv through a usermode API.

The complete project code of EventSyncDriver can be found on github at:

https://github.com/Y33Tcoder/EventSyncDriver

Obtaining one-time arbitrary execution of kernel code

The first step to crafting a continuously executing, kernel cheat driver is to obtain a chance to execute some arbitrary kernel code for once in the target system. In order to achieve this, we will use the kdmapper project, which allows us to run any windows kernel driver code once, through hijacking the vulnerable Intel iqvw64e.sys network driver.

Kernel memory allocation, and hooking an existing kernel driver

With a way to have one-time code execution in the kernel now, 2 additional requirements must be satisfied in order for a continuously running kernel cheat module to exist in a Windows System, and that is:

1. Memory Space in kernel to hold the kernel cheating module
2. A method to obtain continuous execution of the kernel cheating code without creating a System Thread, and without creating any device objects.

For requirement 1, kdmapper itself already provides us with the capability of doing so. When kdmapper.exe is used with a .sys driver image, kdmapper will first allocate a pool of Non-Paged Memory in the Windows Kernel, and then map whatever code is written inside the driver image into that allocated memory. Afterwards, kdmapper will execute whatever is inside the DriverEntry (main function of a driver module) once.

For requirement 2, we will opt to use a method called Windows kernel hooking in order to achieve continuous execution of our cheat module in kernel mode. In essence, what we will do is 1.) Search for a Windows kernel routine (preferably an obscure one that is seldomly used, to avoid causing system instability) which has a respective exported usermode-callable function. 2.) Overwrite the address of the function called by the kernel routine, causing execution to be redirected to our cheating module whenever the kernel routine is called by its usermode API.

The kernel routine we have chosen to hook in this PoC is the NtQueryCompositionSurfaceStatistics routine inside the dxgkrnl.sys DirectX kernel driver module installed by default with all Windows 10 versions.

The EventSyncDrv.sys driver will be responsible for setting up the kernel hook, and will be performing the majority of the memory read/write functionality of the PoC program.

![Code in EventSyncDrv for obtaining the function address of NtQueryCompositionSurfaceStatistics in kernel](image)

Figure 10: Code in EventSyncDrv for obtaining the function address of NtQueryCompositionSurfaceStatistics in kernel
The corresponding usermode function that calls NtQueryCompositionSurfaceStatistics in kernel is exported by win32u.dll, and will be triggered by the call_hook_test function in umctrlproc.exe.

Figure 11: call_hook_test function in umctrlproc.exe, responsible for calling the hook inside kernel from usermode

The disassembly of the original NtQueryCompositionSurfaceStatistics routine, before any hooking, can be viewed with WinDbg kernel debugger as shown below:
After hooking the kernel routine, through the basichook:call_kernel_function function in EventSyncDrv.sys, the original first 12 bytes of the NtQueryCompositionSurfaceStatistics assembly code will be overwritten with a instruction to jump to the EventSyncDrv’s main function loop’s address instead, via a simple mov rax, xxx => jmp rax hook.
As we can see from the above, the first instruction of the `NtQueryCompositionSurfaceStatistics` routine is overwritten with a `mov, rax` to 0x0FFFFAF04541D5D90, which is the location of where the first instruction of our `EventSyncDrv` driver is mapped in the kernel non-paged memory. The second instruction is overwritten with a `jmp rax`, which tells the system to jump and start the execution of our main code in our kernel driver.
Calling the kernel hook from usermode, and restoring the hooked function

The steps to running the PoC project will be first to map the EventSyncDrv kernel driver first, via running the “kdmapper.exe EventSyncDrv.sys” command, which will map our driver code into kernel memory, and do the kernel routine hooking as described above.

Next, in order to obtain a continuous thread of execution, we will achieve this through the act of trapping a ordinary usermode thread inside kernel space indefinitely, which in turn allows us to obtain continuous kernel execution without the need of a Device Object, nor a system thread.

To perform usermode thread trapping, we simply create a new thread in the umctrlproc.exe process, and use it to call the usermode NtQueryCompositionSurfaceStatistics routine, which will then execute the code in the main kernel driver function. From this point on, as long as we keep the kernel driver in a while loop, the thread will never exit from kernel time, and we can do any required functionality such as memory Read/Write indefinitely.

In addition to the above, after the usermode thread has been successfully trapped inside kernel mode, the kernel driver code must also perform one additional critical action, which is to restore the original bytes of the hooked NtQueryCompositionSurfaceStatistics routine in kernel. This will effectively erase any evidence of the dxgkrnl.sys driver of ever being hooked, and prevents anti-cheats from flagging the system as potential cheating when checking for kernel API hooks.

```
//hook basichook:unhook_kernel_function() {
    DWORD function = reinterpret_cast<DWORD*>(get_system_module_export("C:\\SystemRoot\\System32\\dxgkrnl.sys", "NtQueryCompositionSurfaceStatistics"));
    if (!function) { return false; }
    write_to_read_only_memory(function, &original, sizeof(original)); //write back original bytes into kernel func's (NtQueryXXXX) base addr
    return true;
}
```

**Figure 14: unhooking the kernel hook with the basichook:unhook_kernel_function routine in EventSyncDrv**
Execution and synchronization of Read/Write commands given from usermode

Lastly, after we have obtained a continuously executing thread stuck inside kernel space, we must also need a continuous usermode thread and program capable of communication with the kernel driver, executing commands such as reading/writing memory of a certain process while synchronized.

To achieve this, the usermode part of this PoC, umctrlproc.exe, will create a usermode thread which will be responsible as the main thread for executing commands via an API with the kernel. The kernel driver will create 2 Windows Event objects with the IoCreateNotificationEvent() routine, which will be used as mutexes together with the usermode main thread to synchronize their memory read/write operations.

For example, if the umctrlproc.exe wishes to read the memory stored in offset base+0xABCDEF in the process notepad.exe,

1. umctrlproc.exe will first obtain the mutex lock, edit the RWbuf structure in its memory, change the “command” variable to the READ command code, change the “offset” variable to 0xABCDEF, and “struct_size” variable to something say, 4 for 4 bytes to be read.

2. EventSyncDrv will then be able to obtain the mutex lock, and check the RWbuf struct in umctrlproc.exe, and identify the target process’s name (notepad.exe in this case), together with the issued command (READ in this case), and the requested memory address offset to be read (0xABCDEF in this case).

3. EventSyncDrv will proceed to attach to notepad.exe, and read the memory stored in offset 0xABCDEF, and then write the memory read into the RWbuf buffer in umctrlproc. EventSyncDrv will then release the mutex lock.

4. umctrlproc will now obtain the mutex lock again, and will be able to obtain the read information retrieved by EventSyncDrv inside the RWbuf structure.
Figure 15: Step by step interaction between umctrlproc and EventSyncDrv illustrated

```
RWbuf struct
int command = COMMAND_RPM
int struct_size = 4
DWORD64 offset = 0xABCDEF
DWORD64 structure = [address to store read memory in umctrlproc]
```
In Figure 16 below, we can see the output of umctrlproc.exe in console with the default compilation of the project. First, notepad.exe is launched as the target process. Second, the EventSyncDrv.sys file is mapped into kernel with kdmapper.exe. Third, umctrlproc.exe is launched, and it will attempt to read the target usermode process’s (notepad.exe in this case) base address. As can be seen in Figure 16, the target process base address is found to be 0x7ff738380000. The umctrlproc.exe is then programmed to exit after 3 seconds by default.

Figure 16: Project default compilation output, with attempt to obtain base address of notepad.exe

We can further double check the validity of the base address of notepad.exe found by our project using Process Hacker 2, as we can see in Figure 17 below, Process Hacker
confirms that the base address of notepad.exe is indeed 0x7ff738380000, identical to what was found by our program.

![Image of Process Hacker viewing the base address of notepad.exe](image)

**Figure 17: Base address of notepad.exe viewed by Process Hacker 2, compared with result found from umctrlproc.exe**

Therefore, we can conclude we have successfully built a kernel driver capable of doing arbitrary memory read/write, and potentially other kernel operations, which can communicate with a usermode control process to relay information, without creating any device objects or system threads. This builds a sample foundation for bypassing modern kernel anti-cheating software in the market nowadays.
7. Section 3 - The future of game cheating and anti-cheating

With the rapid developments in computational power and IT technology, both cheat developers and anti-cheat developers are venturing into new methods for upping their game in evasion and detection of cheats, respectively. In this section, the near-future possibilities of improving both game cheating and anti-cheating using Virtualization Technologies and Machine Learning will be discussed.

7.1 Virtualization Technologies

For Cheat developers

Virtualization allows for the insertion of a layer of abstraction between the Operating System and the Underlying Hardware. This allows for the possibility of any arbitrary memory reading/writing in the hypervisor of a virtual machine host, without the virtual machine guest ever realizing that its memory has been tampered with.

A proof of concept example of how this can be actually implemented can be found with h33p’s vmread project [12], which is a linux library which allows for reading/writing memory of a Windows 10 KVM machine hosted on the linux host.

For Anti-Cheat developers

Virtualization also brought us the possibility of large-scale cloud computing services, which in turn allowed the possibility of Cloud Gaming to be born. Cloud Gaming is a type of online gaming that runs the actual video game on remote servers and streams them directly to a user's device. The servers usually consist of multiple virtual machines, each delivering gaming service to one customer session. Examples of Cloud Gaming service providers nowadays include NVIDIA GeForce Now, and Google Stadia.

In cloud gaming, since the user/player has no direct access over the hardware of the server, and every modification/installation of software can be heavily monitored by the server side, installing cheating software on the cloud gaming virtual machine is almost impossible if restrictions has been properly put in place in the OS of the virtual machine.

Of course, cloud gaming is not without its drawbacks at its current state. Latency issues and inability for the service to provide a large geographical coverage for its customers
remains the top obstacle yet to be overcome by cloud gaming. However, if these obstacles can be solved in the future, we may be able to evolve into a new age of gaming without the plague of cheating software.

7.2 Machine Learning
For Cheat Developers
Advancements in machine learning, and subsequently computer vision, has allowed the possibility of making cheating bots that make use of the above technologies. For example, the FutureNNAimbot by Trombov [13] is a universal neural network aimbot designed to learn how to aim at enemy targets in all shooter games. The module will learn from the video feed of video games, takes screenshots, and tries to recognize enemy players or objects and move the mouse accordingly to aim at them. This gives the potential for cheats in the future to completely skip the requirement of having to illegally access memory of the game in order to gain non-human advantages, but instead rely on information legally given to the players (such as the video feed on screen) only to achieve non-human performances.

Figure 18: FNNAimbot in action in the game CSGO, where the aiming is completely controlled by the trained AI.
For Anti-Cheat developers

With adequate player data, and the advancements in Machine Learning and Big Data Analytics, it is now possible (and possibly already deployed) for anti-cheat developers to use player data to create machine learning models, and use AI to extract statistical anomalies from the data to identify players with high likelihood of cheating behaviour. For more detail on the relationship between Machine Learning and Anti-cheating, interested readers can delve deeper into the topic by looking into studies performed by other scholars, such as “Machine Learning to identify cheaters in online games”, a thesis in 2020 by Martin Willman [14], which studies in-depth about how the neural network anti-cheating module is implemented by Valve Anti-Cheat in the game CSGO.
8. **Conclusion**

To conclude, this project aims to deliver a review of the existing technologies and techniques employed by cheat developers and anti-cheat developers to compete against each other in the world of game cheating, followed with a proof-of-concept example of a kernel driver that is capable of arbitrary memory read/write through anti-cheat measures, which shows how despite rigorous anti-cheating attempts by game developers, cheat developers will methodically bypass each of those measures step-by-step to achieve their goals in cheating.

The battle between cheat developers and anti-cheat developers is a long and continuous one which cannot be expected to end in the short future.

Existing academic studies regarding this topic are rather scarce and it is hoped that with the completion of this project, it can become a piece of better updated and organized documentation on this lesser known field of software engineering.
9. References

  https://store.steampowered.com/stats/Steam-Game-and-Player-Statistics?l=english


  https://helda.helsinki.fi/bitstream/handle/10138/313587/Anti_cheat_for_video_games_final_07_03_2020.pdf?sequence=2


  https://www.gamerevolution.com/guides/649075-is-the-escape-from-tarkov-radar-hack-fixed


- [10] Microsoft. (April, 2018). PsCreateSystemThread function (wdm.h)
  https://processhacker.sourceforge.io/about.php
  https://github.com/h33p/vmread
  https://github.com/Trombov/FutureNNAimbot