An analysis into kernel level anti cheat system in online games

By

Boros Octav

A thesis submitted to

University of Birmingham

For the degree of

MASTERS OF SCIENCE

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

September 2020

Acknowledgements

The project would not have been possible without a few people who need mentioning. The author's supervisor, Tom Chothia, whose in depth approach has pushed the author to a more decisive and articulated mindset. Author's parents for their ongoing support in dealing with everyday challenges as well as author's friends without whom this project would have been out of reach.

Abstract

Online game applications have seen an immense increase with regards to their market capitalization. Ever present in proliferating industries are adversaries whose main aim is unlawful real-life monetary reward. As such, this paper dissects the functionality of an inhouse built kernel level anti-cheat system, Vanguard from Riot Games Inc. Its role is to deter cheaters from abusing game logic for illicit advantages. This is deployed to Riot's recently released first person shooter, Valorant, which we base our case study upon.

Measures for testing are deployed along with functionality reverse engineering, network traffic decryption and possible bypass scenarios from the perspective of the adversary. Kernel space methods are explored along with their limitations. Low level testing methods ensue prior to development of cheat logic. In the case of a working cheat test, a discussion on detection and lack of this mechanism is entailed along with their limitations. A static analysis further reinforces our findings.

Contents

1 Introduction

1.1 Problem Domain……...7
1.2 Aims of study……7
1.3 Approach……7
1.4 Contributions……..8

2 Background

2.1 Evolution……9

2.2 Literature review……9

2.3 Limitations……..10

2.4 Tools……...10

3 Code injection

3.1 Static and Dynamic linking……12

3.2 DLL injection……13

3.3 Implementation……..13

3.4 Testing……...14

3.5 Manual Mapping of a DLL……...15

4 Api tests

4.1 Windows Api……17

4.2 Tests……..17

5 Kernel space

5.1 Kernel Debugging……20

5.2 Kernel driver……23

6 Memory Analysis

6.1 Memory editing and scanners……25

6.2 Memory dump……25

7 Anti-cheat communications

7.1 Networking……..30

7.2 Frequency analysis……..32

8 Other tests

8.1 SeDebugPrivilege……33

8.2 ProtectedProcessesLight……33

9 Pixel scanning cheats

9.1 Background……35

9.2 Implementation……...35

9.3 Discussion……...36

10 Static analysis

10.1 Debugging and PE header……38

10.2 Dependencies……38

11 Summary and conclusion……40

List of references……..41

Appendix……..44

Chapter 1
Introduction

1.1 Problem Domain

Online game applications have seen an immense increase in the revenue generated in the last decade, reaching a staggering value of $159.3 billion a year and roughly 2.6 billion players **[**1**]**. While these applications are well intended to satisfy the entertainment of their users, unethical behaviour often finds use cases in economically proliferating industries. As such, the age old cat and mouse race pictured first by malware developers and Anti-virus engineers, has taken root in the online gaming industry.

In this context, skilled programmers act as cheat developers, while security engineers are creating detection mechanisms for punishment of those not abiding by terms of service. This situation presents a unique challenge security wise, as portrayed by Black Hat Conferences [2][3]. Many private companies have developed as a response to this situation, with the most notable ones to mention – those developing kernel level systems: BattlEye, EasyAntiCheat and others.

Our case study will focus on what we believe to be the state of art kernel anti cheat Vanguard, released along with VALORANT FPS by RIOT Games. This system, although new to the market, has already faced backlash from potential users for their privacy intrusive behaviour [4], with some users even going as far as calling this kernel driver a “malware” or “rootkit”. While there is some similarity with either of the terms, mainly because of launching before system boot, much like a rootkit, we believe that anti cheat developers are indeed fighting an ever losing battle and thus face the need to resort to very intrusive mechanisms to ensure presence of fair play in game.

1.2 Aims of study

The aim of the study is to aid developers of games in building more robust defence mechanisms, with the scope of prevention being the preferred method rather than detection. We consider the approach RIOT has taken to be the most carefully crafted, implementing certain mechanisms already put in place by the operating system the game is available on: Windows 7/8/10 x64. By testing, implementing and making use of reverse engineering tools, we wish to shed light on latest methods used by anti-cheat systems to deter cheat developers and cheat users.

1.3 Approach

In this paper we will develop and test few existing cheat archetypes, as well as report on the specific countermeasures taken for deterring a casual cheat user. As a rule of thumb, since our test case uses a kernel mode anti cheat, most operations that we would like to test would need to have an interface in the form of a driver. This has been proven to be the most difficult with regards to the scope of this project and thus more explanation on our choice of analysis will follow in chapter 2. As such, the author will take into account ethical considerations of the aforementioned tests. Prior to any cheat development we will provide a high-level description of how it is supposed to work, what API’s are needed and discussion of the

results. This should effectively allow us to gauge whether a cheat can be created prior to bypassing the anti-cheat itself. Furthermore, we will make use of kernel debugging tools to identify functionality of the kernel driver, along with a static analysis, anti-debugging techniques and a breakdown of available attack surface vectors.

1.4 Contributions

Our main contributions consist of the tests we have created in order to validate specific functionality of anti-cheat. The working pixel triggerbot is our own original work, along with the discussion of countermeasures and reasoning of not applying said detection mechanisms. Also, the overview of the attack surface vector and method of obtaining game process memory dump are of great value, as the author has not seen these methods used anywhere within scope of cheating in online games. Lastly, we believe the tests we have implemented are somewhat universal, they can be used to test functionality of rootkit like software drivers or anti cheat systems with a kernel component.

Chapter 2

Background

This chapter will look at current consensus regarding cheating in games as well as explore the sparse literature available. We discuss the first steps taken with regards to anti cheat systems with the aim of providing an overview of evolution. We also consider the limitations to prevention of cheating and provide a scope for future research.

2.1 Evolution

With regards to the history of anti-cheat systems this can be dated perhaps to one of the first multiplayer online games, Ultima Online. Back then, most game logic was handled by the server side where bugs, exploits and potential hacks were not trivial to discover. Manfred explores his experience of 20 years of cheating in online games, iterating that in Ultima days, monitoring network traffic packets and understanding their structure could give the possibility of man-in-the-middle-attack of crafting own packets which were easily accepted by the server[22]. The attack might seem trivial but it was not easy to find. He went as far as demonstrating how in game houses could be deleted or created via the aforementioned method and sold on markets like eBay for a large amount of real money further illustrating the "underworld" of cheating.

As gaming industry gained more traction however, the game of cat and mouse has become more pronounced than ever: anti cheat systems seemed to always be on the tail-end of cheaters. In this context, Valve was among the first company to build an inhouse client-side software system aimed to deter the casual cheat customer with little to no deep understanding of behind the scene structure. Consequently, private solutions(EAC,Battleye) have emerged as response to the niche created, described by major online gaming companies such as Bohemia Interactive with their DayZ product. The previously mentioned solutions have shifted their software to ring 0 level, implementing a kernel level anti-cheat system seen as the first "offensive" against cheat developers.

Ultimately, in June 2020, Riot Games has released their inhouse kernel anti cheat system, namely Vanguard along with their first FPS(first person shooter) type of game. Before this release, Riot engineers have also written an article illustrating their anti-cheat development journey on a high-level [32].

2.2 Literature review

As previously mentioned, the literature is sparse and we attribute this to the nature of the defence system: prevention is aimed at increasing difficulty of reverse engineering functionality as well as the grey line computer scientists must walk should they wish to test the defences put in place with no possibility of requesting symbols. We believe this to be the case because in online forums people always use a nickname with the purpose of evading potential legal actions that could arise. An example of such event could be the most well known case of Glider bot in World of Warcraft[33]. The author of said hack effectively disregarded any possible backfire and in the end got tracked and charged with $6 million in damages.

In the world of academics, an interesting research debates the difference between client-side and server-side protection mechanisms with regards to cheating [34]. We note here that the specified research is a high-level representation and a collection of methods used in both the

client and the server to which a ranking system of efficiency is attributed, rather than a black box environment type of analysis.

Moreover, we identify one of the most unique applications of blockchain technology: a means of detecting client-side inconsistencies by implementation of server logic atop smart contracts in multiplayer online games [35]. We believe this to be an interesting topic for further research as their tests have proven to achieve small and acceptable overhead with regards to smart contract optimisation. A critical note should be taken given scalability issues cryptocurrencies such as Ethereum have faced, which is based on same underlying technology.

Lastly, one of the most intriguing methods of deterring cheat developers is by use of a specially crafted tamper resistant hardware [36]. As such, the authors investigate the main methods of cheat implementation and detection in order to construct potential functionality the hardware should have in order to successfully achieve its objective. We note here that this is the only method we see fit for completely removing possibility of cheating without damaging performance of game. The other possibility which is currently not feasible, would be to move all game logic on the server side and implement an interactive streaming protocol for players with the obvious caveat of latency.

2.3 Limitations and ethics consideration

At the start of our project, we decided to emphasize the limitations of our tests in order not to infringe on either Computer Misuse Act or cause any damage to RIOT Games Inc. As such, all cheat tests performed shall be restricted as much as possible to the practice range menu, which effectively puts the user in PVE setting, i.e vs bots. The only scenario in which we will defy the aforementioned setting is the live testing of our developed cheat with means to assess the detection mechanism put in place. The reason for this resides in the fact that we have not found any publicly available information regarding cheats being detected in both in PVE and in multiplayer and we aim for robustness of results.

This latter type of testing shall be done under unrated matches so as to limit the disturbance potentially caused to other players. Additionally, since our cheat has been well optimised to work with little overhead, we will aim to not abuse the illicit functionality and introduce some artificial human error such that our overall performance in game doesn’t exceed expectations of an average skilled player. All optimisation of cheat shall be performed during practice range.

Nevertheless, with regards to Networking chapter, we shall not employ any method that would cause any damage to RIOT servers such as denial of service.

2.4 Tools

With respect to tools we used during this project, one of the first that stands out is the Windows 10 x64 Professional Operating system up to date patch, as this was the one available on our machines at the time. We use mainly Visual basic C++ for most of our tests with exception being python 3 for pixel scanning cheats. Dependency walker is used for static analysis along with PE studio. For process related tasks such as identification of networking services or inspection of module handles we employ Process Hacker and Process

explorer. WinDbg, x64dbg, Ida are among the debuggers we have used throughout our project along with memory scanning software Cheat Engine. Extreme injector is also used with regards to DLL injection method. Wireshark is used for networking purposes along with the embedded windows firewalling tool netsh.

Chapter 3

Code injection

This chapter provides an introduction to static and dynamic linking libraries as well as implement and test 2 injection methods. Results are analysed and conclusions on whether a cheat can be injected via this method will be drawn.

3.1 Static and Dynamic library linking

In order to understand the process being followed in next chapter, we will look at life cycle of a program, from source code to execution. As such, the source code is written in a text file, regardless of language. From here, it is then thrown into a compiler which produces object code a target machine can understand and execute. Consequently, the machine runs the compiled code by loading it into memory.

In most scenarios, these compiled codes will make use of other programs or libraries to achieve their purpose and this method is effectively split into 2 types. The first which we will not investigate further because it does not apply to our test case, is the static linking. This is the last process executed by the compiler, before producing the executable image. Essentially, library modules are being copied and their references are merged into the program code such that the final executable image will be able to load into memory, along with all its libraries. The reason we will not investigate this further is because we are dealing with a commercially packed executable image, and besides anti debugging routines being present, there is no way of fully retrieving the source code of this target application[5] for the purpose of patching and recompilation.

Furthermore, the other type of linking libraries to our source code is Dynamic linking. This is of particular interest for our test case because it can be done at runtime, that is right after the executable has been loaded into memory. The main difference is that libraries are not effectively written as part of the executable, but rather their names appear at the start of this image. Thus, in the life cycle of our program, we will have a much smaller executable image. At runtime, the libraries will be dynamically linked, placing both the executable and libraries in memory. As a side note, dynamic linking allows for a single copy of the module imported to be used by multiple applications, reducing overhead. It is also really important to mention that from the developer perspective, dynamic linking provides a great advantage: if some platform related library gets an update, the developer hardly needs to do anything because it uses the copy of library already present in target machine memory. This is not true for static linking as the developer will need to update their library and recompile with the updated version.

In the next subchapter, we will introduce a classical approach to executing arbitrary code in a target application via Dynamic linking libraries.

3.2 DLL injection

DLL injection is one of the most popular methods to insert arbitrary code into a target process at runtime such that one can run this arbitrary code without knowledge of source code. Traditional injection methods make use of the windows PE loader, namely LoadLibrary. This

api has few functions which will be described in the manual mapping of a DLL section later, but most importantly, processes call LoadLibrary() to explicitly link to a DLL[6].

3.3 Implementation

In order to create a DLL, one has to research general architecture for how code is being handled by a generic process injector. The following code snippet details our approach which we used in a previous assignment during the course's taught sessions.

```
DWORD WINAPI OnDllAttach(PVOID base)
{
#ifdef _DEBUG
    AllocConsole();
    freopen_s((FILE**)stdin, "CONIN$", "r", stdin);
    freopen_s((FILE**)stdout, "CONOUT$", "w", stdout);
    freopen_s((FILE**)stdout, "CONOUT$", "w", stderr);

    SetConsoleTitleA("PROJECT");
#endif

    while (!(GetAsyncKeyState(VK_DELETE) & 0x8000))
    {


        uint64_t* valorant = (uint64_t*)GetModuleHandle(NULL);
        printf("Address of valorant client is %p\n", (uint64_t)(valorant));

        Sleep(2000);

    }
    FreeLibraryAndExitThread(static_cast<HMODULE>(base), 0);

}
```

Figure 1 OnDLLAttach function

First, we see the OnDllAttach WINAPI function, which as the name suggest is the routine that is performed at the start of our injected module. We have defined a debug console, printing out the base address of Valorant client, via GetModuleHandle api. The while GetAsyncKeyState is just a way to exit the injected process via a button, for ease in debugging. The last api call, FreeLibraryAndExitThread as Microsoft documentation points out [7], the reference counter of the dll is decremented and the thread which was created during injection is exited.

```
VOID WINAPI OnDLLDetach()
{
#ifdef _DEBUG
    fclose((FILE*)stdin);
    fclose((FILE*)stdout);

    HWND hw_ConsoleHwnd = GetConsoleWindow();
    FreeConsole();
    PostMessageW(hw_ConsoleHwnd, WM_CLOSE, 0, 0);
#endif
}

BOOL WINAPI DllMain(HMODULE hModule, DWORD dwReason, LPVOID lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        DisableThreadLibraryCalls(hModule);
        CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)OnDllAttach, hModule, NULL, NULL);
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        OnDLLDetach();

    }
    return TRUE;

}
```

Figure 2 OnDLLDetach and DLLmain functions

Secondly, we have our OnDLLDetach Win api function which effectively kills the console spawned upon attach, as a routine for detaching the injected dll. Furthermore, the dll main where our calls to the aforementioned functions are being done along with the thread creation in which the cheat logic will be executed. Notice that the main logic of our simple DLL is in the OnDllAttach function, and thus this is where potential implementation of a cheat logic should reside. Lastly, we've also disabled thread library calls, i.e DLL_PROCESS_ATTACH and DLL_PROCESS_DETACH, with the purpose of optimizing and stealth increase, as the

initialization code of our DLL will not be paged in memory during creation/termination of threads.

One may ask why we have not developed a standalone cheat logic for the DLL injection, and the answer to that resides in the next chapter where we test our implementation on the game process.

3.4 Testing

During information gathering phase of our project, we have decided that the best approach would be to test the offset.dll injection via Extreme Injector with different settings in order to assess some functionality of the kernel anti cheat. As such, the next table shows results of attempting dll injection into the game client process, "Valorant-Win64-Shipping.exe".

Process name	Injection mode	Result	Comments	Date
Valorant-Win64-Shipping.exe	Standard Injection	Failed	Unable to find kernel32.dll in specified process	29/06/2020
Valorant-Win64-Shipping.exe	Thread Hijacking	Failed	Unable to find kernel32.dll in specified process	29/06/2020
Valorant-Win64-Shipping.exe	LdrLoadDll Stub	Failed	Unable to find ntdll.dll in the specified process	29/06/2020
Valorant-Win64-Shipping.exe	LdrpLoadDll Stub	Failed	Unable to find ntdll.dll in the specified process	29/06/2020
Valorant-Win64-Shipping.exe	Manual Map	Failed	Injection method returned null	29/06/2020

Table 1. Extreme injector DLL injection results

From the table above, we can infer that there is some type of protection the anti-cheat is doing. All injection modes have been tested against CSGO.exe and notepad.exe processes for robustness of results. Since the latter have injected our dll aimlessly, we turn our attention to error messages received upon injection trials. All of them but manual mapping have returned an unable to find module in the specified process, which suggests that module handles are being stripped by the kernel driver. We've investigated this further with other process explorer type of software, namely Process Hacker and Process Explorer.

In Process Explorer, as previously stated, an error message is shown where module handles should show: "Access is denied", regardless of whether the software is run with or without administrator privilege. However, upon repeating the same routine to launch the software and check properties of process, but this time with Process Hacker – we find all relevant game module handles. The explanation we found later was that Process Hacker installs a kernel driver for some of its functionality, namely "KProcessHacker.sys", with a digitally signed certificate. This essentially resolves our hypothesis that module handles of game are inaccessible from user mode space.

Since Manual Map mode of injection has returned null, we've decided to implement the manual mapping of a dll injection ourselves, with the idea in mind that certain api calls, such as LoadLibrary(), the windows PE loader, might get detected and instantly blocked. The reason for our train of thought was that we wanted to pinpoint precisely which function piece of manual mapping mode has been blocked, so next subchapter will handle that.

3.5 Manual Mapping of a DLL

The process of manually mapping a dll can be summed up in a sentence: manually perform tasks that the Loadlibrary api is doing, upon loading a module inside a process instance. To get a better view of what that means, have a look at the Figure 1.

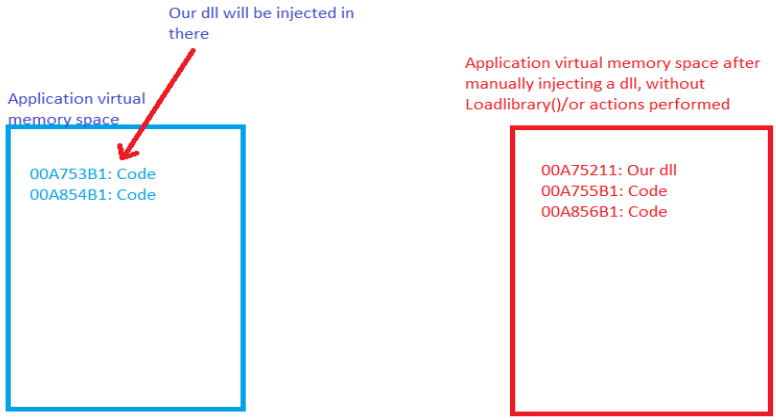

Figure 3. DLL raw manual injection

We can easily distinguish the Application virtual memory space on the left before injection and on the right after injection. Thus, if Loadlibrary() is not called, the pointers resolved by the application at launch time will no longer be valid. So, when the application will call the code at 00A753B1, following our injection, the call will fail and most likely cause the process to crash. This is a representative figure and is by no means showing the actual bit by bit layout of memory space in Windows.

A LoadLibrary call effectively resolves this issue on its own, by loading raw binary data, mapping sections into target process, injecting loader shellcode, performing relocations, fixing imports and executing TLS(thread local storage) callbacks, as described here[8]. We've followed the guide in the reference, fixed deprecated functions and produced a working manual mapping injector. We decided not to go over the implementation steps as this would be laborious and not provide much to our analysis.

Having a working version of our manual mapping injector, we went ahead and tested it on CS GO, notepad and Valorant game process. Our results show that while the injection method works on both CS GO and notepad, as expected, we have found that the reason for our previous failure of employing Extreme Injector, was due to a VirtualAllocEx() api call. This

effectively should allocate memory in desired process but fails which tells us that there is some sort of functionality the kernel driver provides which blocks specific Windows API's from being called on target processes. On the other hand, if the anti-cheat is disabled via systray icon and game launched following which we attempt to manually inject our dll, we find the game client is not unpacked completely in memory and our injection fails at retrieving base address of target main module.

We discovered that particular Windows Apis are being blocked and thus we will continue with the next chapter: Api Tests.

Chapter 4

API Tests

This chapter introduces Windows API's following which we test a predefined set of said functions on target processes. Results are present in Table 2 and a discussion ensues.

4.1 Windows API

Operating system in the context of Windows is responsible for providing means of Process management, Memory management, File management and more[39]. The way in which an application accesses this embedded code functionality is via Windows API's. These can be summed up in the set of functions a user can call, along with their signatures, calling conventions, and associated data types under Windows operating system[39]. An illustrative example would be few of the actions performed during the run of an executable file: create process, allocate memory, load linked modules. All these operations are performed in the background by the host operating system, via CreateProcess, VirtualAlloc, LoadLibrary apis. These are important with regards to our project because we are analysing a kernel level anti-cheat system which has access to all the functionality explained above as well as an extension for kernel-privileged API's which are not accessible from user mode.

4.2 Tests

Having performed our tests for DLL injection, we now go on to investigate which APIs are being targeted by anti-cheat. In order to test only relevant API calls we would need should we develop a cheat, we have taken a look at previously designed cheats for an assignment in taught session of the course. Following the creation of a list of APIs to test, we have implemented a console application in C++ that does exactly that: performs the calls on our target processes.

The following table shows results of our tests.

WINAPI/flag	Process	Error/success	Error	DATE
CreateToolhelp32Snapshot/TH32CS_SNAPPROCESS	"VALORANT.exe"	Success	N/A	11/07/2020
CreateToolhelp32Snapshot/TH32CS_SNAPPROCESS	"VALORANT-Win64-Shipping.exe"	Success	N/A	11/07/2020
CreateToolhelp32Snapshot/TH32CS_SNAPPROCESS	"vgc.exe"	Success	N/A	11/07/2020
CreateToolhelp32Snapshot/TH32CS_SNAPMODULES	"VALORANT.exe"	Failed	Access is denied	11/07/2020
CreateToolhelp32Snapshot/TH32CS_SNAPMODULES	"VALORANT-Win64-Shipping.exe"	Failed	Access is denied	11/07/2020
CreateToolhelp32Snapshot/TH32CS_SNAPMODULES	"vgc.exe"	Failed	Access is denied	11/07/2020

CreateToolhelp32Snapshot/TH32_SNAPTHREADS	"VALORANT.exe"	Success	N/A	11/07/2020
CreateToolhelp32Snapshot/TH32_SNAPTHREADS	"VALORANT-Win64-Shipping.exe"	Success	N/A	11/07/2020
CreateToolhelp32Snapshot/TH32_SNAPTHREADS	"vgc.exe"	Success	N/A	11/07/2020
OpenProcess/PROCESS_ALL_ACCESS	"VALORANT.exe"	Success	N/A	11/07/2020
OpenProcess/PROCESS_ALL_ACCESS	"VALORANT-Win64-Shipping.exe"	Success	N/A	11/07/2020
OpenProcess/PROCESS_ALL_ACCESS	"vgc.exe"	Failed	Access is denied	11/07/2020
OpenProcess/PROCESS_QUERY_INFORMATION	"VALORANT.exe"	Success	N/A	13/07/2020
OpenProcess/PROCESS_QUERY_INFORMATION	"VALORANT-Win64-Shipping.exe"	Success	N/A	13/07/2020
OpenProcess/PROCESS_QUERY_INFORMATION	"vgc.exe"	Failed	Access is denied	13/07/2020
EnumProcessModules	"VALORANT.exe"	Failed	Access is denied	13/07/2020
EnumProcessModules	"VALORANT-Win64-Shipping.exe"	Failed	Access is denied	13/07/2020
EnumProcessModules	"vgc.exe"	Failed	Access is denied	13/07/2020
OpenProcess/PROCESS_VM_READ	"VALORANT.exe"	Success	N/A	13/07/2020
OpenProcess/PROCESS_VM_READ	"VALORANT-Win64-Shipping.exe"	Success	N/A	13/07/2020
OpenProcess/PROCESS_VM_READ	"vgc.exe"	Failed	Access is denied	13/07/2020

Table 2 WinApi tests

One can not help but wonder why we have tried the api calls on all game related processes. This idea stems from the thinking that should one find a user mode technique to inject arbitrary code in any of the processes, albeit the anti-cheat user mode service or game, we should be able to communicate with the game itself with elevated privileges while hiding from the anti-cheat in plain sight. OWASP has defined privilege escalation very well, identifying it as any means of accessing unauthorized data [9]. Thus, if we would be able to perform certain injection attacks to privileged processes, then our tests on other processes concerning the game become meaningful.

Our results above show that all but one Api call targeting the user mode service of the kernel anti cheat, vgc.exe have failed. This was rather to be expected having known that RIOT has taken considerable measures to prevent such vulnerabilities. After all, the company has to

ensure that the only possible scenario of use of their proprietary driver is to block illicit users and not raise other vulnerabilities or potential privacy intrusive damages.

Moreover, the CreateToolHelp32Snapshot api call is used to parse list of processes/modules/threads with flags accordingly set. The kernel driver blocks these methods to enumerate modules inside a process. This is critical in development of a cheat because even if we were not to inject anything in the game and we would want to read/write process memory directly to modify in game values, we would fail doing so. The reason is that ReadProcessMemory and WriteProcessMemory windows API calls take an argument of type LPCVOID lpBaseAddress and is, according to MSDN [10], a pointer to the base address in the specified process from which to read. A hacky method would be to call ReadProcessMemory with lpBaseAddress parameter as null, however this will return false since Windows checks that the open handle supplied to that process has at least PROCESS_VM_READ privilege over that memory region specified by lpBaseAddress.

We have also tested a secondary means of enumerating a process' modules, EnumProcessModule via PSAPI library. Results are to be expected- although they are different win32 APIs, the underlying NT API being called is the same thus two possibilities arise here, either both EnumProcessModule and Createtoolhelp32Snapshot with SNAPMODULES flag are being blocked or the common underlying NT api is blocked. Either way, this is certainly a mechanism the anti-cheat is using, given that the calls to game process launched with anti-cheat disabled return true. To be noted that, while the game launches with anti-cheat disabled, it does instantly prompt a "Vanguard not initialized" error and leaves the game in a suspended state. In the latter state, we can identify that all previous apis tested return true on VALORANT-Win64-Shipping.exe client process (albeit very few modules running in that instance- game not entirely unpacked) while user mode service of driver is no longer present.

Chapter 5

Kernel Space

This chapter concerns actions performed in kernel space. It sheds light on previously found blocks and aims to analyse results from kernel debugging. A basic driver is implemented and various driver signature enforcement spoofers are tested.

5.1 Kernel Debugging

We have found out some interesting mechanisms up until now: module handles are potentially stripped, API calls are being blocked. There is no conclusion yet to how the anti-cheat is performing those actions, so we decided to investigate by using kernel debugging provided my Microsoft, via WinDbg. We have attempted at setting up network kernel debugging sessions between a target and a host machine via ethernet cable. A host machine is the machine where our debugger is running, while a target machine is the machine which we are debugging. Thus, having a clear aim in mind, to figure out functionality of the driver module, we have stumbled upon another issue: in order to run kernel network debugging, the target machine needs to be set to test mode(=debug mode), with secure boot option disabled.

Secure boot option is a security standard developed with the purpose of making sure that a device boots using only software that is trusted, or in other words digitally signed with a certificate[11]. Windows test mode, or debug mode effectively disables driver signature enforcement, along with enabling us to set up the kernel network debugger. Driver signature enforcement has been introduced on all x64 versions of windows 7+[12]. Since our machines are both running windows 10 x64 up to date, it is also important to note that Valorant game requirements only include x64 versions of windows 7+. The similarity is already stunning: the anti-cheat must be using this embedded windows policy otherwise RIOT would lose out on potential users and thus limit their market reach – a price few if any gaming companies are willing to pay.

We turned Secure boot option off, by restarting and changing the setting in Bios boot options. We booted up and found out that our driver is running and we can still play the game. However, if we run the test signing mode command to turn on, after which a restart is needed, we find that the anti-cheat is no longer present in the systray and the game prompts a restart with error “Vanguard not initialized, please reboot.”. After repeating this process an arbitrary number of times we can conclude that the anti-cheat driver unloads itself upon boot time, should test signing mode be turned on.

Since disabling driver signature enforcement unloads anti cheat driver after system has booted, we have given up kernel debugging over network and we looked for other ways to debug kernel, this time locally. After few trials and errors, we have found a driver produced by Microsoft, namely LiveKD[13]. Although there is not much specification in documentation, this utility allows us to perform memory inspection of both kernel and user mode space. To be noted that while the name and the description of the software both mention it as being a live kernel debugging utility, we have found that to be false, in the sense that it is not exactly live, but rather provides a snapshot of kernel memory at software launch.

We found that the user needs to restart the utility, for the snapshot to be actualized, should there be any actions performed in the kernel after the launch of the LiveKD.

At this point, we have employed a WinDbg plugin, namely wdbgark[14], an extension which is aimed at providing anti rootkit functionality. After all, we might not consider the anti-cheat driver to be a rootkit but it could behave in a similar way, given ring 0 privileges.

We now proceed to discover the main culprit of our injection and api tests' blocks: process and thread callbacks registered with ObRegisterCallbacks. This type of functionality has been introduced by Windows because anti-virus software was hard hooking the Windows kernel to intercept WinApi calls. As an example, when a file executable is launched, the anti-virus would be alerted and thus can scan the file for malicious signatures, preventing its execution[15]. Similarly, in the context of an anti-cheat, when an api call is targeting the game process, the anti-cheat checks signature and blocks the call. The following code snippet was referenced from here[15], showing the structure of a callback entry item, with its beginning function of allocating memory:

```
if ( (a1->Version & 0xFF00) == 256 && a1->OperationRegistrationCount )
{
  v7 = (a1->OperationRegistrationCount << 6) + a1->Altitude.Length + 0x20;
  v8 = (OB_CALLBACK_ENTRY *)ExAllocatePoolWithTag(PagedPool, (unsigned int)v7, 'lFbO');
```

```
typedef struct _CALLBACK_ENTRY_ITEM {
LIST_ENTRY EntryItemList;
OB_OPERATION Operations;
CALLBACK_ENTRY* CallbackEntry; // Points to the CALLBACK_ENTRY which we use for ObUnRegisterC
POBJECT_TYPE ObjectType;
POB_PRE_OPERATION_CALLBACK PreOperation;
POB_POST_OPERATION_CALLBACK PostOperation;
__int64 unk;
}CALLBACK_ENTRY_ITEM, *PCALLBACK_ENTRY_ITEM;

typedef struct _CALLBACK_ENTRY{
 __int16 Version;
 char buffer1[6];
 POB_OPERATION_REGISTRATION RegistrationContext;
 __int16 AltitudeLength1;
 __int16 AltitudeLength2;
 char buffer2[4];
 WCHAR* AltitudeString;
 CALLBACK_ENTRY_ITEM Items; // Is actually an array of CALLBACK_ENTRY_ITEMs that are also in
}CALLBACK_ENTRY, *PCALLBACK_ENTRY;
```

Figure 4 ObRegisterCallback structure

The ObRegisterCallbacks function is used to register the callbacks, for a given event. Now that we understand this structure, we will check on the existence of these routines via LiveKd, specifically by running the extension search of callback object types before and after exiting Vanguard. Our approach may seem a bit rough on the edges but that has a good reasoning: LiveKd doesn't allow the functionality of setting breakpoints in the kernel while live local debugging so it is not possible to inspect the register values upon disassembling the memory regions or stepping through code in order to point precisely what events are being watched.

In Figure 5, illustrating a snapshot from Windbg, we can see the callbacks registered with ObRegisterCallbacks of all types (process and thread), before exiting the anti-cheat. Of particular importance are those under symbol of vgk, since this is the name of Vanguard

driver. We can identify a pair of Pre and post operation in both the process and thread type. The pre and post operation effectively relate to before and after event watcher is triggered.

We now exit the anti-cheat, following which we will restart our LiveKd and check for presence of these callbacks again.

[+] Displaying callbacks registered with ObRegisterCallbacks with type *

Address	Name	Symbol	Module	Suspicious
0xffff918214ad1e80	Desktop			
0xfffff8000b5201c0	OB_PRE_OPERATION_CALLBACK	WdFilter+0x401c0	wdfilter	
0000000000000000	OB_POST_OPERATION_CALLBACK			
0xffff918214a8fa60	Process			
0xfffff8000b5201c0	OB_PRE_OPERATION_CALLBACK	WdFilter+0x401c0	wdfilter	
0000000000000000	OB_POST_OPERATION_CALLBACK			
0xfffff8000b482cd0	OB_PRE_OPERATION_CALLBACK	DeepMgmtDriver+0x2cd0	deepmgmtdriver	
0000000000000000	OB_POST_OPERATION_CALLBACK			
0xfffff800108068a0	OB_PRE_OPERATION_CALLBACK	VBoxDrv+0x168a0	vboxdrv	
0xfffff800108039d0	OB_POST_OPERATION_CALLBACK	VBoxDrv+0x139d0	vboxdrv	
0xfffff8000f0ac6ac	OB_PRE_OPERATION_CALLBACK	vgk+0xc6ac	vgk	
0xfffff8000f0ac694	OB_POST_OPERATION_CALLBACK	vgk+0xc694	vgk	
0xffff918214ad17a0	Thread			
0xfffff8000b482cd0	OB_PRE_OPERATION_CALLBACK	DeepMgmtDriver+0x2cd0	deepmgmtdriver	
0000000000000000	OB_POST_OPERATION_CALLBACK			
0xfffff80010806c10	OB_PRE_OPERATION_CALLBACK	VBoxDrv+0x16c10	vboxdrv	
0xfffff80010803a50	OB_POST_OPERATION_CALLBACK	VBoxDrv+0x13a50	vboxdrv	
0xfffff8000f0ac6ac	OB_PRE_OPERATION_CALLBACK	vgk+0xc6ac	vgk	
0xfffff8000f0ac694	OB_POST_OPERATION_CALLBACK	vgk+0xc694	vgk	

Figure 5. Pre Vanguard exit registered callbacks of all types

[+] Displaying callbacks registered with ObRegisterCallbacks with type *

Address	Name	Symbol	Module	Suspicious
0xffff918214ad1e80	Desktop			
0xfffff8000b5201c0	OB_PRE_OPERATION_CALLBACK	WdFilter+0x401c0	wdfilter	
0000000000000000	OB_POST_OPERATION_CALLBACK			
0xffff918214a8fa60	Process			
0xfffff8000b5201c0	OB_PRE_OPERATION_CALLBACK	WdFilter+0x401c0	wdfilter	
0000000000000000	OB_POST_OPERATION_CALLBACK			
0xfffff8000b482cd0	OB_PRE_OPERATION_CALLBACK	DeepMgmtDriver+0x2cd0	deepmgmtdriver	
0000000000000000	OB_POST_OPERATION_CALLBACK			
0xfffff800108068a0	OB_PRE_OPERATION_CALLBACK	VBoxDrv+0x168a0	vboxdrv	
0xfffff800108039d0	OB_POST_OPERATION_CALLBACK	VBoxDrv+0x139d0	vboxdrv	
0xffff918214ad17a0	Thread			
0xfffff8000b482cd0	OB_PRE_OPERATION_CALLBACK	DeepMgmtDriver+0x2cd0	deepmgmtdriver	
0000000000000000	OB_POST_OPERATION_CALLBACK			
0xfffff80010806c10	OB_PRE_OPERATION_CALLBACK	VBoxDrv+0x16c10	vboxdrv	
0xfffff80010803a50	OB_POST_OPERATION_CALLBACK	VBoxDrv+0x13a50	vboxdrv	

Figure 6. Post Vanguard exit registered callbacks of all types

We clearly identify that after exiting the anti-cheat, the callback routines are unregistered. Thus, we can conclude on the previously found function call blocks: the anti-cheat driver is using callback type objects in order to watch particular events and block them upon initialization. We can also add to the events watched bucket the API calls which have failed on various game processes. To be noted that, the reason for these conclusions are not only the above illustrations but also the return values of our api tests, which all return true following game launch with anti-cheat disabled, as discussed earlier.

Furthermore, these results also present an ever-greater challenge: how can cheat developers bypass those callbacks, if they can be at all bypassed? The answer to that resides in the idea that a user has full control over their own machine and should in theory be able to unregister those callbacks by diving into kernel space. As such, the next subchapter will cover the idea behind our tests at bypassing these event watchers.

5.2 Kernel drivers

If a user has access to kernel space, they should be able to bypass any mechanism of protection the anti-cheat is using. Thus, we went ahead and developed our own “hello world” driver for testing, following Microsoft documentation. If we are able to load this driver, then we can manipulate those callbacks mentioned before and increase attack vector surface. However, we have faced a few challenges.

The main problem resides in the previously mentioned Driver Signature Enforcement policy. We now know that to load a driver without a digitally signed certificate, we need to turn test signing mode on along with secure boot off. This in turn, unloads Vanguard module at boot time, so the idea behind running our self-signed driver to remove the previously mentioned callback structure suddenly is not feasible anymore without obtaining a certificate from regulated authority. On a side note, we have also found few ‘illicit’ certificates that one could re-use for this purpose but due to ethical considerations we have not explored that route.

What if we try to spoof the DSE, such that we can trick the anti-cheat into thinking that the bit of the variable of DSE(namely g_CiOptions) is 1 instead of 0? Well, we went ahead and tested that on our machine. Most DSE spoofers that we employed are available on Github[19][20][21] and make use of a vulnerable driver with an officially signed certificate. One thing to note here is that the vulnerable drivers we have attempted to use for testing purposes are very old (dating back >10 years) and besides compatibility issues we have found towards the end of our project timeline that these drivers are also blocked by the kernel anti-cheat at startup, with the error present in Figure 7. As such, we can make a conclusion here: using a vulnerable driver, i.e with a backdoor, is limited because of blacklisted drivers as well as compatibility issues.

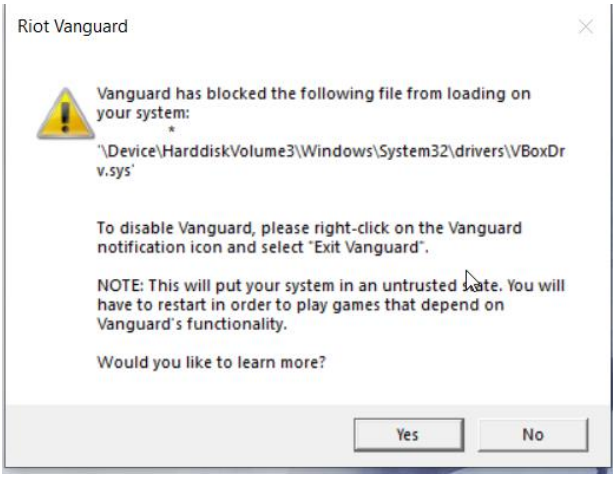

Figure 7. Vulnerable VirtualBox driver blocking

One may ask so is this game impossible to cheat on without a digitally signed certificate? Well, probably not but the answer to that resides in the game requirements: players can choose to use an option between any windows x64 7+ versions. Unfortunately, we have not been able to perform tests on Windows 7 given our current machine OS and lack of Windows 7 license. The reason for this testing is represented by compatibility issues we have

encountered during our project. We will not look into this further given it is out of scope of our project but we will mention that, should Riot have decided to limit the platform requirements even more, that is only to Windows 10 users, the market share loss would be too great and as such we will continue to explore other potential attack surface vectors.

Chapter 6

Memory analysis

This chapter provides insight into memory scanners as well as taking a memory dump of the system to analyse unpacked binary of game client. Limitations of our approach ensue and results are discussed.

6.1 Memory editing and scanners

Modifying the game data while running has been the most common approach towards obtaining various in game advantages. Considering our goal to pinpoint defence mechanisms in kernel anti cheat software, we decided to employ the most widely known memory scanner: Cheat Engine. This software effectively allows the user to surf through memory, scan for specific values such as health, walking speed or other objects inside the game.

In this context, we discovered a previously unseen behaviour, after launching the game with anti-cheat enabled. We have attempted at starting Cheat Engine, although we have failed at doing so, instantly receiving a system error from the game, rendering the process to a crashed, suspended state until the game is exited and a potential crash dump can be generated. We have also found out that after the game exits, our cheat engine application launches from the previous launch, that is we have not relaunched it following the crash. Our hypothesis here is that the anti-cheat driver knows about the name or the functionality of the cheat engine process and suspends it if game process is active. Thus, we decided to erase all strings related to “Cheat Engine” inside the application, by opening the executable in hex editor and replacing all those references with arbitrary characters. So, if the anti-cheat would just check for window names for example, our approach at modifying Cheat Engine should steer clear from the driver’s radar. We proceeded to start the game again, launch our modified cheat engine and we find out that the same system error is thrown. Ever vigilant, we decide we should also try to launch cheat engine prior to launch of game, however, we shortly find out that the same system error is thrown upon launch of game while cheat engine is in a crashed, unresponsive state until game process has fully exited.

We can conclude from here that Cheat Engine memory scanner software is blacklisted by Vanguard and the means of this blacklisting resides most likely in byte signature verification rather than some blunt window name checks. To be noted that we have also attempted to trigger the system error by having a browser page open on cheat engine website for robustness of results, which also has not triggered that error.

Since scanning memory via existing software is blacklisted but not punished with a ban, we decided to look into taking a memory dump of the game process, after which we can investigate and search for different in game parameters. As such, the next subchapter will cover our approach at obtaining a memory dump and its analysis.

6.2 Memory Dump

Commercial applications generally take precautions with their proprietary source code. Back in 1960s, the service, software and source code were supplied to customers without separate

charge[16]. This has led software developers who were also customers to redistribute the proprietary software free of charge. As a consequence, vendors have stopped shipping source code altogether in order to protect their products from ill intended distribution.

Nowadays, most commercial applications have gone to an even greater length to obfuscate their executables besides the initial high-level obfuscation phase of compilation. As explained earlier in our report, there are hardly any methods available to retrieve source code in full from a pre compiled executable image. So, the reasoning behind our approach towards taking a memory dump of the game resides partly in our previous failures and partly in the idea that in order for the game to run with acceptable performance and respond in real time, it has to deobfuscate or unpack its binary in memory. Should one be able to take a memory dump without closing or crashing the game, a cheat developer could find the players objects, iterate through each of them individually depending on their structure and create an overlay of displaying other players through texture and terrain. Or in simpler words, they could develop an Extra sensorial perception cheat.

Before we continue, we need to understand the difference between virtual memory and physical memory, specifically in Windows. The operating system uses virtual memory as a way to manage processes execution on the physical memory (RAM). As such, every individual process will receive a "ticket" which reserves a slot on the RAM upon launch. This manager of memory will in turn decide if there is enough available space on the physical ram for the last launched process to be paged – or in other words mapped to their corresponding slot[17]. If there is enough space, the process will be mapped to physical memory and will execute. Otherwise, it will enter a queue and wait until all previously mapped processes finished their execution and will be paged out, allowing queued processes to be executed. Furthermore, another feature typical of most if not all operating systems that is critical to our analysis is ASLR, or Address Space Layout Randomization, introduced in Windows since Vista versions. This effectively mixes and spreads out the address space of main process – main program, dynamic libraries, stack and heap, memory mapped files and others[18]. The aim of this latter specification is to ensure that memory corruption attacks are harder to implement, by preventing the attacker from knowing the layout of certain runtime applications in memory without accessing any resources on target machine. Moreover, ASLR also ensures that if we launch the same user mode application twice in a row of launch-exit-launch, its memory layout will be slightly different.

Now that we understand these two concepts, we will also note that at runtime, the addresses which we would find in virtual memory are very different to those that are mapped on physical RAM because of the aforementioned mechanisms. As such, there is no relationship to be drawn between our memory dump(a snapshot of physical RAM) and the actual layout in memory at runtime(virtual). At runtime, a cheat developer will perform a memory byte scan of the required objects, find pointers to these addresses via subtracting the address where the object was found from the base address of the client process. This works because at runtime even though virtual addresses differ across different launches, all of game modules and data will be loaded in a sequential manner following base address.

We have employed several different applications such as: Process dump, native task manager from windows interface, Process Hacker and others. All had the functionality of creating a memory dump from a target process. While the latter were visible in all of them, no

application successfully produced a full memory dump of the specific game process. All have thrown the ever-seen error of “Access is denied” even under administrator privileges. This effectively tells us that should we wish to obtain a memory dump, we must have kernel privileges.

However, after thorough research, we have found that we could take a full system memory dump, including the kernel memory, upon crashing the windows machine with a Blue Screen of Death. Initially, we did not have access to the embedded windows crash simulation technique. The reason we did not have access was because of layout of our machine’s keyboard – the registry key embedded in Windows for simulating a BSOD requires the CTRL+SCROLL button pairs, while general laptop keyboards don’t have the SCROLL button and modifying the registry key to other button values renders it unusable. Thus, we had to look for a different approach at triggering a BSOD, either acquire an external keyboard or even better simulate the dump without crashing the machine.

Following trials and errors, we have found that the LiveKD utility described earlier has the functionality to create a complete system dump, including kernel memory without crashing the machine. Additionally, it also has the functionality of taking a process specific memory dump. While we have attempted at targeting the process only, our tests have shown that due to size of the game executable image, process specific approach fails and would be too laborious to take dumps of maximum size of 100 KB where the image is roughly 4 GB.

We launched LiveKD while in game and we have taken the memory dump of the whole system without any errors of access violation or game crashes by using the command “.dump /f [pathtosave] memory.dmp”. So far, the future looked promising given that we can take a dump including game memory – which will be the size of the physical RAM available on that machine. We can potentially search for the various parameters such as player name, health or others at the time of our dump.

While inspecting the memory dump in Hex editor we were unable to find important game parameters. The reason for that we discovered later was due to the fact that even in memory, security engineers from RIOT have employed a proprietary obfuscation technique entitled “packman”. The following snapshot further elaborates on our finding. Although quite comically, we must mention here that the method of taking the dump of process has not been seen in any of cheat forums we have investigated. Most if not all solutions available online relate to use of a driver in order to perform said action and we have explained in the earlier chapters why that approach was not available in our toolkit.

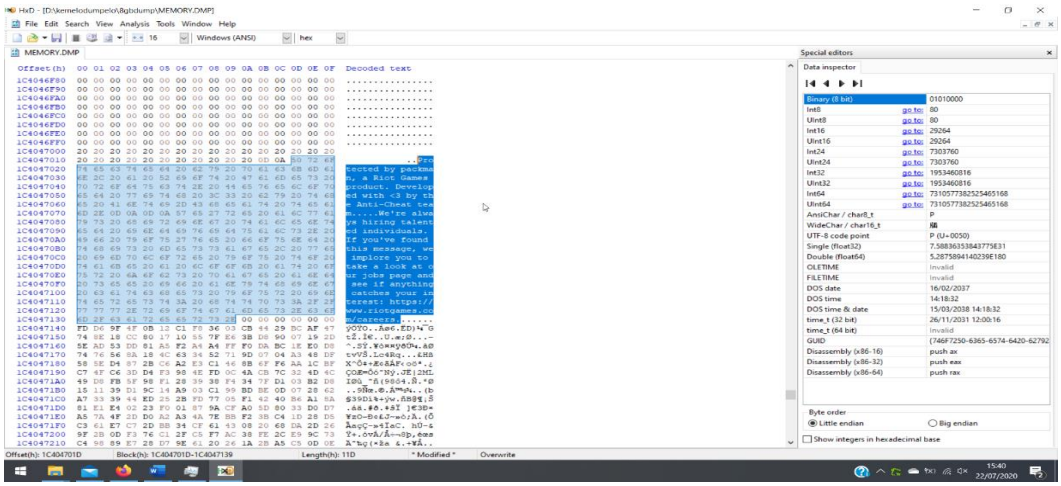

Figure 8. Packman obfuscation in game memory note from developers

Moreover, we also find out some specification of this packman obfuscation done on physical ram, present in Figure 9.

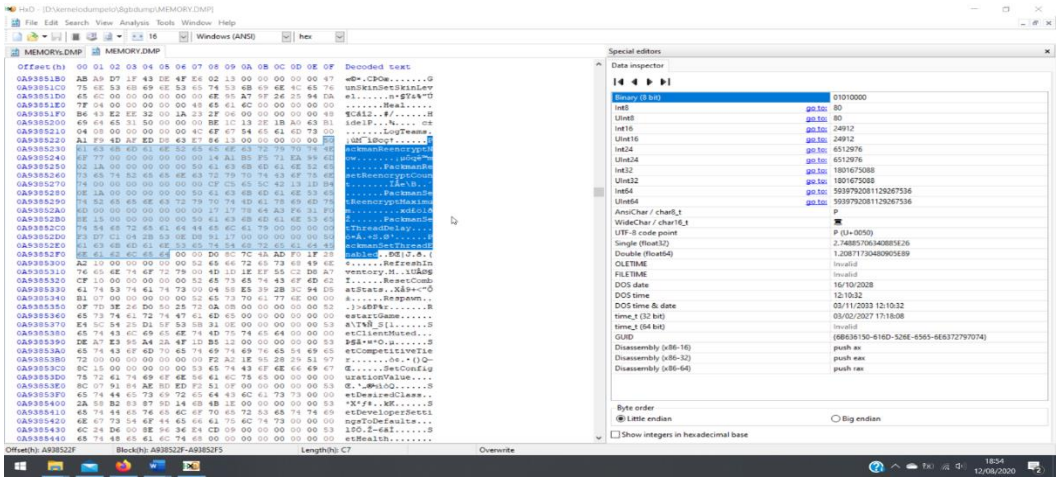

Figure 9. Packman Reencrypt functions

We can easily identify PackmanReencryptNow, PackmanResetReencryptCount, PackmanSetReencryptMaximum, PackmanSetThreadDelay, PackmanSetThreadEnabled functions being called. If their name is of any relevance it would suggest that this obfuscation is revisited with a certain timer. We managed to find these function calls in 3 other places on the physical ram at the time of our dump and many more if we leave the game on for a while and then take the memory dump. So, we can precisely say that this encryption routine shifts its place in memory according to a timer.

We have attempted to figure out what obfuscation is being used by taking an arbitrary number of dumps in different instances of the game. We can essentially freeze the game memory in any time interval by suspending the process via task manager. Unfortunately, this was deemed rather too laborious for our project timeline because the loading time of game on our machine took on average 40 seconds. Considering processing power and machine resources, we would need to perhaps take a full system dump every second of the game launch with no guarantee that we would capture the encryption routine in plain sight, given we believe this obfuscation routine to be performed in a matter of milliseconds. Our project

aim as discussed earlier was not the development of a cheat but rather understanding of mechanisms of protection against cheat developers taken by RIOT which we hope we have achieved with regards to this chapter. We should also mention that we believe this packman obfuscation to not be anything more severe than few byte shifts operations or xor with some wild numbers. The reason for that is performance – running AES 256 every 'reencrypt' sequence while the user is looking to have his shot landed a fraction of a second after he clicks, is unrealistic at best.

While we did not make use of a custom-built kernel driver to dump the memory of the process precisely, we managed to obtain the memory of process via alternative route. We conclude this chapter with the take home that provided we could obtain a digitally signed certificate from Microsoft, we could remove the ObRegistercallbacks registrations, take our dump via a user mode service or perhaps take the memory dump from kernel space directly, after identifying the virtual address range of the game process.

Chapter 7

Anti-cheat communications

This chapter decrypts the network traffic of the anti-cheat user mode service. A frequency analysis is performed along with identification of heartbeat services.

7.1 Networking

Networking has been a subject of increased interest across cheat developers, especially in the early days of the online gaming community. Back then, modifying a simple bit in a packet could cause unwanted exploits such as receiving memory of server back as a response[21]. However, in recent times exploits like these with heavily funded games are very rare. This chapter will investigate the network structure put in place, perform a frequency analysis while also identifying the "heartbeat" type of connection between game processes.

We decided to inspect the network traffic of the anti-cheat user mode service, namely vgc.exe. We employed Process hacker, in order to check active connections of processes via services tab. We find that the user mode service of the driver establishes only one connection upon game launch with either of the 2 following ip addresses: 104.18.7.209 and 104.18.6.209. Before we go into frequency analysis and decryption of packets, we decided it would be interesting to firewall connections sequentially in both directions, inwards and outwards. We made use of netsh embedded windows tool, similar to IPSEC in Linux platforms. We block incoming traffic from the above addresses to our host machine while in game and we find out in Wireshark that the server is trying to finish the connection although does not succeed in doing so. The reason is obvious, the FIN,ACK packets are not being ACK'd by host and the user mode service keeps sending packets to RIOT server. This still allows us to play the game indefinitely and we have also found out that it can be exploited by blocking opposite direction traffic – game has a mechanism of punishing early match leavers for disrupting other players' gameplay. Thus, if we block outwards packets from reaching riot servers, we trigger a disconnect roughly 1 minute 30 seconds after the last sent packet by the host. Essentially this tells us that there is a heartbeat type of connection between the user mode service and the game client process, as well as a connection between the vgc.exe and the Riot Servers. However, there is no direct communication between the game client to that specific Riot server.

After testing of firewalling the vgc.exe process, we moved on to decrypt TLS encrypted packets sent by the same process. In this scenario, we set up SSLKEYLOGFILE in environmental variable path and we attached the file to our Wireshark packet capture before monitoring. The results are present in Figure 10. We can identify the fact that although we decrypted headers of packets, application data contents are obfuscated prior to shipment of packets.

However, while performing a simple strings operation on the game client, albeit with anti-cheat disabled, we find some very interesting packet contents available in Figure 9. These are unencrypted as we discovered that the user mode service of the anti-cheat establishes a Winsock type of local connection between itself and the game client. Although we have not found an adequate test other than preventing this connection, we strongly believe this is the

primary heartbeat of the anti-cheat. The reason for this is that we suspended the thread in the vgc.exe process responsible for this connection – after which we faced an instant kick from the game with a “connection error”. The secondary heartbeat will be described in the frequency analysis.

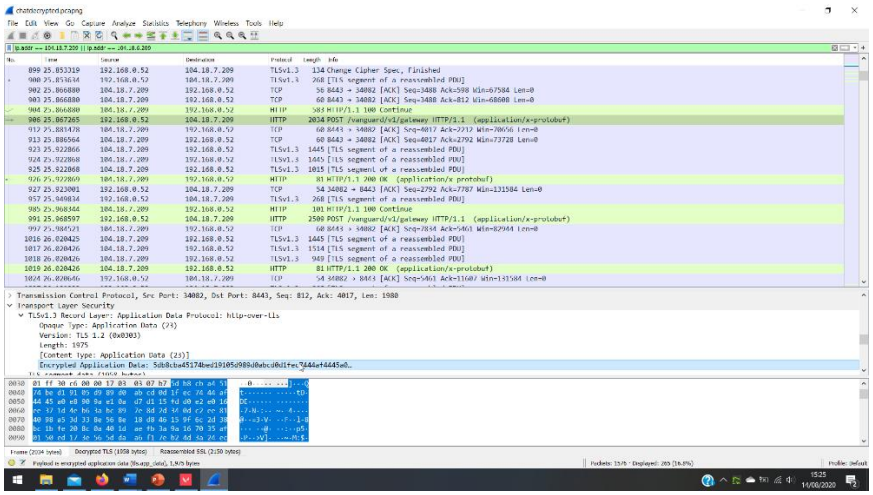

Figure 10. Decrypted TLS traffic of vgc.exe

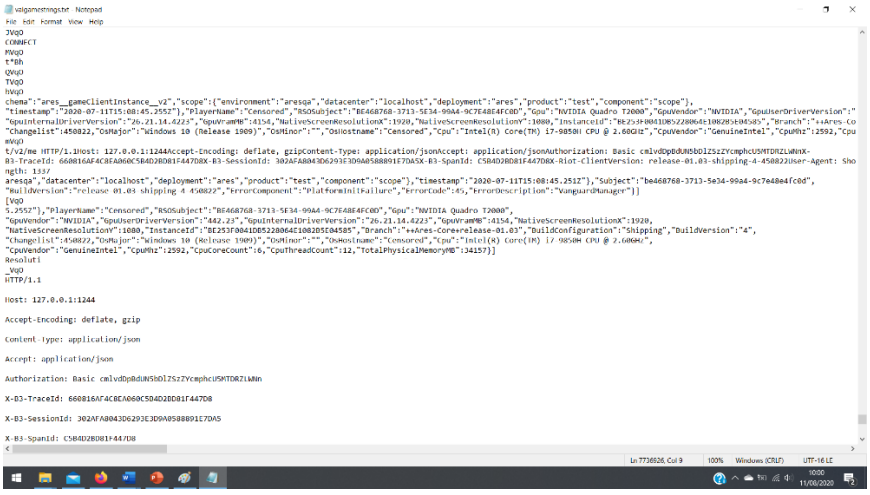

Figure 11. Valorant game client strings with ac disabled

As we can see, a lot of internal machine information is revealed: Gpu information, censored player name for packet shipment and privacy purposes, censored OsHostname, Cpu information, as well as display device information. While a bit uneasy with the fact that Valorant communicates these machine related identifications, it should be expected given ring 0 privileges of the anti-cheat. We also note here that we believe these ids to be responsible for what is known as “Soul banning”. Effectively, once a cheater is caught, in order to make the ban persisting, the ban is bound to several hardware IDs. Thus, even if the player creates a different account, internal systems of Riot would find out if the player uses the same “banned” hardware ids by simply running a search against blacklisted HWIDs and block the new account as well. We took a leap of faith when we enquired for permission to use university computers to figure out banning mechanism precisely, however we were faced with a strong no access policy because of the ongoing pandemic. We iterate here that further

research should make use of multiple machines and use a publicly available and known to be detected cheat with the purpose of reverse engineering banning system.

We have decrypted the packets, we found that the application data is obfuscated prior the shipment of packets and we move on to frequency analysis subchapter.

7.2 Frequency analysis

In this context, we aimed to test previously failed attempts: cheat engine blacklisting, api tests and process dumping with the scope of finding whether packet sizes differ according to type of error. Table 3 describes a sample which was present in almost all our tests, of the 4 packets seen to be sent every 1 minute and 30 seconds with corresponding sizes in bytes. This is in turn our secondary heartbeat of the anti-cheat, connecting it with Riot servers and establishing the complete heartbeat cycle:

Game(1st->)anti-cheat(2nd->)Riot Servers.

We find no correlation between types of errors/failed tests and packet sizes. However, since the 4 packets being sent follow the iterations presented in the table, we can make a soft conclusion about iteration 8: at irregular intervals the anti-cheat might take chunks of memory and send them back to RIOT for further analysis. We should note that we have tested utilizing a great amount of RAM available while monitoring and this has not increased the frequency of iteration 8, packet number 3.

Packet number	1	2	3	4
Iteration1	268	101	2512	1180
Iteration2	268	101	2512	1180
Iteration3	268	101	2512	1180
Iteration4	268	101	2512	426
Iteration5	268	101	2613	1180
Iteration6	268	101	2512	1180
Iteration7	268	101	2608	1180
Iteration8	269	101	16460	1180
Iteration9	268	101	2512	1231
Iteration10	268	101	2512	1180

Table 3. Frequency analysis packet sizes(bytes)

At this stage, we conclude the networking chapter, mentioning that the type of security employed is state of art with respect to networking structure, taking care with censoring private user information before shipment.

Chapter 8

Other Tests

This chapter analyses two other tests we have performed. Results are discussed and potential for future research is addressed.

8.1 SeDebugPrivilege

In the windows security model, privileges are a native control that grants particular rights for accounts to perform privileged operations within the operating system[23]. Our topic of interest here is the SeDebugPrivilege, which states that any process owning debug privilege has access to adjusting privileges of other processes.

All debugging processes must have SeDebugPrivilege token enabled in order to successfully attach to a process or start a process in a debugging state of their own. As such, our hypothesis here is that debuggers are being blocked from attaching to any of game processes by a potential callback type of structure which adjust the privilege token of the calling process upon instantiation.

In this sense, we implemented a C++ script which makes use of AdjustTokenPrivileges WinApi that changes the privilege token value of a target process. We have successfully run it on a range of target debuggers(x64dbg, WinDbg) after which we attempted to attach to game processes. We find that the kernel anti cheat is not having an influence over this variable while the debugger attachment is still blocked throwing the ever seen “Access is denied error”. We note here that while this method hasn’t shed any light upon functionality of anti-cheat, during the course of the project we found a high-level description of Riot security model development[37]. Within that, the engineer wrote that with every patch, anti-debugging methods are randomized and changed with the purpose of preventing cheat developers even further and thus, this result might be useful in further research.

We conclude here that the debugger attachment is blocked by other means not related to SeDebugPrivilege token and we proceed with our analysis to protected processes feature introduced in Windows.

8.2 Protected processes light (PPL)

The concept of protected processes has been introduced since Windows 8.1 versions which ensure that operating system only loads trusted services and processes[24]. In other words, this is similar with the secure boot option described earlier, however it concerns the user mode run time processes rather than kernel space drivers. Out of the functionality of this security control, our main concern is that the anti-cheat driver is flipping the bit responsible for this protection, in the context of game related processes, blocking debugger attachment and access to virtual memory. Additionally, since our platform is up to date Windows 10, we note that ever since Windows 10 RS 2(‘Redstone 2’,Windows 1703) patch, a binary signature enforcement policy has been introduced in the kernel EPROCESS structure, with the aim of preventing injection of code into “light protected” processes[25][26]. As such, by means of

removing this protection on game processes, we will figure whether our previous DLL injection could be blocked via this mechanism as well.

We successfully compiled the code available here[25], produced a driver and we proceeded at loading of this driver. We have time again failed to load the driver, given the driver signature enforcement policy and unloading of anti-cheat. However, we did attempt at modifying the driver code to switch the protection on rather than off while anti cheat was unloaded with test signing on, with the hypothesis in mind that the game system error being thrown at launch was caused by this feature being turned on. We conclude that the anti-cheat driver is not solely changing the Protected Processes light structure of game process as we receive the same system after suspending the game process at launch and running the driver: “Vanguard not initialized, please reboot” with the possibility that multiple checks are performed including PPL bit. Further research should ensue with caveat that this test should be performed while anti-cheat is enabled and driver signature enforcement policy spoofed under Windows 7 versions.

Given earlier discussed knowledge of Riot engineers shuffling anti debugging techniques throughout their patching and updating timeline, we believe this type of security control could be employed in future patches and thus future research should include this testing as well.

Chapter 9

Pixel Scanning Cheats

This chapter explores an unconventional method of cheating in online games. We illustrate implementation and reasoning for pivoting and failed tests. Advantages and disadvantages of potential detection mechanisms are discussed.

9.1 Background

In digital imaging, a pixel is a physical point in a bitmap image, or the smallest addressable element in an all points addressable display device; it is the smallest controllable element of a picture represented on the screen [27]. For the purpose of our investigation, we decided to implement a pixel scanner such that it will scan the current active screen pixels, figure out a particular pixel structure of enemy players and move cursor target crosshair to the body of the enemy in real time. Effectively, this concept can be referred to as a pixel aimbot. The reasoning behind this test was that our research has proved most publicly available cheats relate to pixel scanning and most of these have been regarded as detected and punished with a ban. Thus, we proceed with the implementation of the aimbot in python.

9.2 Implementation

First part of the implementation resides in capturing the current active screen window. We make use of the Pillow library, store the picture in a buffer and use the Win api GetPixel to extract the RGB values of target pixel. During this phase, we also test the SendInput WinApi in order to simulate virtual mouse movements. By using coordinates of our screen resolution i.e 1980x1080, we figure out a specific square box we want to search in the focal of view surrounding player crosshair. The purpose of this is efficiency and stealth, as scanning the entire screen can be quiet time consuming and would cause delayed actions which defeat the purpose of the cheat of providing an illicit advantage. We should note here that the reason why Pixel aimbot is particularly easy to implement resides in a game functionality, namely Enemy Highlight Colour. This feature creates an outline of enemy characters with 3 choices available for colours. Following our testing, we decide upon purple colour, given its pixel RGB values were the least present in the screenshots we have taken of in game play and thus our aimbot won't point towards terrain/textures if ever they are present in the FOV.

Having identified the pixel structure we want to scan for and move camera to point towards, we proceed with testing in game. We shortly find out that while the cursor moves in menu, the camera of the player is not moving at all. We have attempted at sending various types of coordinates inputs, both relative from the current cursor position and absolute. We find that these are still unresponsive in game and thus we can conclude that virtual mouse movement events are filtered through some mechanism.

However, during our testing we have found that virtual mouse click events are not filtered and are being accepted by the game instantly with no errors. Thus, we will pivot our pixel aimbot to an auto fire type of cheat. This effectively scans the pixels surrounding crosshair and if an enemy player is present, it will automatically shoot. We proceed to create a working auto fire cheat and we find a certain delay which is easily distinguishable in the GetPixel

calls. We remove these calls, we transform the screenshot into an array of pixels which we parse and we proceed to extended multiplayer testing for detection and ban purposes.

9.3 Discussion

Our optimisation proved so successful that we believe human reaction time is slower than the cheat reaction time. Within the scope of our analysis, we tested this yet to be blocked cheat by playing with it active at all times during 10+ hours of gameplay. We can easily conclude that this type of cheat is undetected as we have yet to receive a ban. Moreover, we believe the potential detection vectors to be the device type structure in the mouse event being sent along with the screenshot taking api. The former has been identified as being of type "None", hence anti-cheat could just adopt a callback type event that checks device type any time a mouse click is sent. In practice however, we believe this would cause a terrifying overhead with respect to game performance and this could be the motive for exclusion of detection mechanism. Regarding the latter, screenshot taking, we believe that anti-cheat developers face the never ending internal conflict: since taking screenshots is used by means of reporting various in game rule breaking and potential bugs, blocking these would cause more harm than good to company's user base due to lack of marketing, publicity and reporting method. After all, security engineer's scope is not to completely remove cheaters, given this is impossible with current specifications of client owning the machine the game is played on, but rather to restrict the impact they have upon their user base as much as possible without an unacceptable cost on performance side.

Furthermore, another method of detection of virtual mouse events such as those sent via SendInput api, is to use a low level hook via SetWindowsHookEx(). As such, WH_MOUSE_LL callbacks effectively provide an "INJECTED" flag for the virtual inputs[38]. We believe this mechanism to be in place for the mouse camera movements we attempted initially for the pixel aimbot. The reason for not including this prevention method with the mouse clicks reside in the overhead space, as the player shots should land as quickly as possible from the real click being performed while mouse movements can be offset via sensitivity to provide real time action. We have also tested a means of intercepting mouse events via a driver [28] at the kernel level although we haven't been able to load it with the ever seen caveat that the driver does not have a digitally signed certificate and it was not a solution anymore. We do suggest for further research this should be attempted on a Windows 7 platform with the spoofed DSE explained earlier.

Lastly, we mentioned earlier that during our research, the pixel scanning type of cheats were reported to be detected and punished with a ban. The fact that we haven't received a ban yet proves the method of detection taken by Riot is either blacklisting applications (AutoHotKey script software for example) or byte signature scanning of running processes against known cheats signatures which we seemingly have evaded by writing our own original code and not publishing it.

Chapter 10

Static analysis

This chapter will make use of debugging tools in order to assess functionality of anti-cheat. We reinforce our previous conclusions via dependency analysis and provide more insight into potential methods used.

10.1 Debugging and PE header

At the start of our static analysis, we attempted to either attach a debugger to running game/anti cheat process or launch either of those from inside the debugger. We have found yet another event being watched by the earlier described callbacks: CreateProcess api. This has been confirmed across multiple debuggers: x64dbg, WinDbg. If we disable the anti-cheat, we discover, as expected, that we can attach debuggers to processes and even open a process inside a debugger.

We also inspected the PE header of the “vgc.exe” and we found under sections tab that raw addresses and raw sizes of .text , .rdata , .data , .pdata are all 0. This is a means of obfuscation technique which strips important information about code sections implementing yet another method of reverse engineering deterrent: obfuscation of original entry point. Most of the aforementioned sections which have data stripped have a reference to a .stub1 section which we identified as being responsible for encryption decryption routines noting that the highest entropy is seen under this part with a file ratio of 98.73%.

While inspecting the disassembly code of the user mode service of anti-cheat(“vgc.exe”), we employed a malware analysis tool to understand the type of packer used given encrypted routines. The malware analysis tool we used is available here [29] and the results show that the packer used is VM Protect v1.70.4. This packer effectively protects code by executing it on a virtual machine with non-standard architecture that makes it extremely difficult to analyse and crack software [30]. As such, we decided that the reverse engineering of the packed binaries was out of scope of our project timeline and we proceed with dependencies analysis.

10.2 Dependencies

Dependencies are an important part of static analysis given we can identify functionality of the anti-cheat by inspecting the imported and exported functions. For this purpose we make use of Dependencies software available here[31].

Firstly, we inspect the vgk.sys driver file and we quickly recognize HAL kernel mode library, which provides routines for accessing hardware present on the machine. In this context, we identify HalRequestRealTimeClock and we believe this is used to control time taken as a means of anti-debugging feature – clock in the time it takes for routines to execute and if it takes longer than a threshold then the process is being debugged and action can be performed accordingly. Information regarding processor type is retrieved via the export “HalEnumerateProcessors” and this is in line with our findings in the networking chapter.

Under ntoskrnl.exe imports, we find NtQuerySystemInformation function which can retrieve with SYSTEM_POLICY_INFORMATION & SYSTEM_CODEINTEGRITY_INFORMATION parameters data about local security policies, namely Secure boot policy, test signing mode, and more thus we identify the functionality through which the driver checks for test signing mode that triggers the unloading before completion of boot. Moreover, we identify the RtlQueryRegistryKey which gives the ability to access registry keys values. ObRegisterCallbacks function discovered earlier is present as well, further confirming our results in the Kernel debugging chapter. The function export PsIsProtectedProcessLight is identified with regards to our tests in the respective subchapter along with PsSuspendProcess functionality which we have identified under Cheat Engine tests.

Secondly, we inspect the vgk.dll library of the anti-cheat and we further identify a range of registry keys manipulation functions such as RegEnumValue, RegEnumKey, RegOpenKeyExA, RegDeleteKey and more under the kernel32.dll library imports. Additionally, we discover the ability of enumerating process modules via K32EnumProcessModules and K32EnumProcesses as well as K32EnumDeviceDrivers. We can infer from here that the driver has the ability to scan processes and even device drivers and potentially block them from loading if a blacklisted driver is recognized. This is further reinforced by chance: on one machine a specific driver was blocked by Vanguard, namely NOX emulator driver which was already installed at the time of Valorant installation. This is part of an android emulation driver and we believe was blocked because of functionality that it provides: virtualizing software with the purpose of reverse engineering it. Handle manipulation functions such as GetHandleInformation and CloseHandle are present and thus can act as a means of controlling open handles to game processes. File access features such as FindFirstFileW, CreateFile, ReadFile, WriteFile are discovered, which effectively illustrate the driver has potential of inspecting all files present on the machine it runs on.

Thirdly, we inspect the vgc.exe user mode service of anti-cheat and we shortly identify process enumeration functions which can explain the first heartbeat connection we have identified in the networking chapter. Under kernel32.dll exports, we can see ToolHelp32ReadProcessMemory which can inspect the game memory for illicit memory editing as well as provide explanation for the irregular sized packets being sent by this process to riot servers – memory chunks being sent to Riot for further analysis. Additionally, under user32.dll we observe the GetClipboardData function which provides the anti-cheat access to the clipboard data along with FindWindow api calls function exports which are aimed at perhaps identifying the game process window upon instantiation. In the same library exports we also recognize cursor pointer related functions such as: GetPointerCursorID, GetPointerDevice, GetPointerDeviceProperties and these could be used as a countermeasure for the pixel scanning cheats that make use of cursor movements.

Chapter 11

Summary and conclusions

Beginning our study with the never-ending game of cat and mouse illustrated by cheat developers and security engineers of online gaming companies, we have dissected the most recent in house developed kernel anti-cheat system. In contribution, we identified the reasons for certain api tests failures as well as iterated that the type of protection is state of art.

We illustrated implementations and failures of user mode cheat methods against the kernel anti-cheat. We have concluded that apart from pixel scanning cheats, all user mode cheat methods are blocked and/or detected with respective discussion on chosen procedure.

With regards to kernel space, we discussed the challenges faced and we have concluded that due to our operating system version this is limited. Driver signature enforcement spoofing has been shown to fail because of vulnerable drivers being blocked by anti-cheat.

We identified the core protection mechanism put in place via ObRegisterCallback types and iterated a means of bypassing provided a driver digital signature is available. Memory editing methods have been tested and analysed with the caveat that the memory obfuscation being done even at run time is rather shallow: the damage to performance of game would be unacceptable for a cryptographically secure encryption routine.

We performed other tests which although have failed to produce an interesting result, we believe further research should include those as Riot engineers are shuffling anti-debugging techniques according to their insight. We created, implemented and tested a working pixel scanning cheat and we elaborated on potential detection mechanisms, their drawbacks and further solutions.

Lastly, we performed a static analysis and we further confirmed on our previous findings.

List of References

[1]Tom Wijman (2020) *The World's 2.7 Billion Gamers Will Spend $159.3 Billion on Games in 2020,* Available at: *https://newzoo.com/insights/articles/newzoo-games-market-numbers-revenues-and-audience-2020-2023/* (Accessed: 04/08/2020).

[2] Joel St. John and Nicolas Guigo (2014) *Next Level Cheating and Leveling Up Mitigations*, 2014 edn., Black Hat Conference, Available at: *https://www.youtube.com/watch?v=bsYxcpz3w8A* (Accessed: 04/08/2020)

[3] Joel Noguera (2020) *Unveiling the Underground World of Anti-Cheats,* Available at: *https://www.youtube.com/watch?v=yJHyHU5UjTg&t=18s* (Accessed: 04/08/2020).

[4] Callum Smith (05/2020) *Valorant: Is its anti-cheat client malware? Riot Games respond to accusations on Reddit,* Available at: *https://www.hitc.com/en-gb/2020/04/17/valorant-anti-cheat-client-malware-reddit-riot-games/* (Accessed: 05/08/2020).

[5] B. Schwarz, S. Debray and G. Andrews, "Disassembly of executable code revisited," *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, Richmond, VA, USA, 2002, pp. 45-54, doi: 10.1109/WCRE.2002.1173063.

[6] Microsoft (01/2020) *LoadLibrary,* Available at: *https://docs.microsoft.com/en-us/cpp/build/loadlibrary-and-afxloadlibrary?view=vs-2019* (Accessed: 06/08/2020).

[7] Microsoft (2018) *FreeLibraryAndExitThread,* Available at: *https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-freelibraryandexitthread* (Accessed: 06/08/2020).

[8] Broihon (2018) *Manual Mapping of dll,* Available at: *https://guidedhacking.com/threads/manual-mapping-dll-injection-tutorial-how-to-manual-map.10009/* (Accessed: 05/07/2020).

[9] OWASP (2020) *Privilege Escalation,* Available at: *https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/05-Authorization_Testing/03-Testing_for_Privilege_Escalation* (Accessed: 10/08/2020).

[10] Microsoft (2020) *ReadProcessMemory,* Available at: *https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-readprocessmemory* (Accessed: 10/08/2020).

[11] Microsoft (2020) *Secure boot,* Available at: *https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot* (Accessed: 11/08/2020).

[12] Microsoft (2020) *Driver signature enforcement,* Available at: *https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-* (Accessed: 11/08/2020).

[13] Microsoft (2020) *LiveKD,* Available at: https://docs.microsoft.com/en-us/sysinternals/downloads/livekd(Accessed: 11/08/2020).

[14] swwwolf (2020) *wdbgark anti rootkit extension,* Available at: *https://github.com/swwwolf/wdbgark* (Accessed: 11/08/2020).

[15] douggem (2015) *ObRegisterCallback and Counter measures*, Available at: *https://douggemhax.wordpress.com/2015/05/27/obregistercallbacks-and-countermeasures/* (Accessed: 11/08/2020).

[16] Wikipedia (2020) *Proprietary Software,* Available at: *https://en.wikipedia.org/wiki/Proprietary_software* (Accessed: 12/08/2020).

[17] Sushovon Sinha (2015) *System Memory,* Available at: https://answers.microsoft.com/en-us/windows/forum/windows_10-performance/physical-and-virtual-memory-in-windows-10/e36fb5bc-9ac8-49af-951c-e7d39b979938(Accessed: 12/08/2020).

[18] Jacob Thompson (2020) *Six Facts about address space layout randomization on windows,* Available at: https://www.fireeye.com/blog/threat-research/2020/03/six-facts-about-address-space-layout-randomization-on-windows.html(Accessed: 12/08/2020).

[19] Volkanite (2018) *TestSpoof,* Available at: https://github.com/Volkanite/TestSpoof(Accessed: 13/08/2020).

[20] hfiref0x (2018) *Windows Driver Signature Overrider(DSEFIX),* Available at: https://github.com/hfiref0x/DSEFix(Accessed: 13/08/2020).

[21] hfiref0x (2019) *Turla Driver Loader,* Available at: https://github.com/hfiref0x/TDL(Accessed: 13/08/2020).

[22] Manfred (2019) *DEF CON 25 Manfred Twenty Years of MMORPG Hacking Better Graphics and Same Exploits,* Available at: https://www.youtube.com/watch?v=QOfroRgBgo0(Accessed: 13/08/2020).

[23] Palantir (2019) *Windows Privilege Abuse: Auditing, Detection, and Defense,* Available at: https://medium.com/palantir/windows-privilege-abuse-auditing-detection-and-defense-3078a403d74e(Accessed: 16/08/2020).

[24] Kaspersky (2018) *Windows Protected processes light(PPL),* Available at: https://support.kaspersky.com/13905(Accessed: 16/08/2020).

[25] MattiWatti (2020) *PPL Killer,* Available at: https://github.com/Mattiwatti/PPLKiller(Accessed: 16/08/2020).

[26] Alex Ionescu (2013) *The Evolution of Protected Processes – Part 1: Pass-the-Hash Mitigations in Windows 8.1,* Available at: https://www.crowdstrike.com/blog/evolution-protected-processes-part-1-pass-hash-mitigations-windows-81/(Accessed: 16/08/2020).

[27] Wikipedia (2020) *Pixels,* Available at: https://en.wikipedia.org/wiki/Pixel(Accessed: 16/08/2020).

[28] Github (2017) *Interception,* Available at: https://github.com/oblitum/Interception(Accessed: 16/08/2020).

[29] Hybrid-Analysis.com (2020) *Malware analysis sample,* Available at: https://www.hybrid-analysis.com/sample/09c034539862dfbec16b482a6499cb0573cc495fecfefbb0c513d8e189e05c32/5efd590dcef78c44890808c5(Accessed: 17/08/2020).

[30] Vmprotect.com (2020) *VMProtect Commercial Packer,* Available at: https://vmpsoft.com/support/faq/(Accessed: 17/08/2020).

[31] lucas (2020) *Dependencies,* Available at: https://github.com/lucasg/Dependencies(Accessed: 18/08/2020).

[32] RIOT (2017) *Evolution of security at Riot,* Available at: https://technology.riotgames.com/news/evolution-security-riot(Accessed: 18/08/2020).

[33] Josh Phillips (2012) *DEFCON 19: Hacking MMORPGs for Fun and Mostly Profit,* Available at: https://www.youtube.com/watch?v=hABj_mrP-no&t=916s(Accessed: 18/08/2020).

[34] Samuli Lehtonen (2020) *Comparative Study of Anti-cheat Methods in Video Games*, Online, Helda: Helsingin yliopisto.

[35] Sukrit Kalra, Rishabi Sanghi, Mohan Dhawan (2018) 'Blockchain-based real-time cheat prevention and robustness for multi-player online games', *CoNEXT '18: Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pp. 178-190.

[36] Wu-Cheng F., Ed K., Travis S. (2008) 'Stealth measurements for cheat detection in on-line games', *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pp. 15-20.

[37] Riot (2017) *The evolution of security at Riot,* Available at: https://technology.riotgames.com/news/evolution-security-riot(Accessed: 24/08/2020).

[38] Remy Lebeau (2018) *Detection of virtual mouse events,* Available at: https://stackoverflow.com/questions/4553777/detecting-simulated-keyboard-mouse-input(Accessed: 24/08/2020).

[39] LMU.edu (2018) *A Programmer's Introduction to Windows,* Available at: https://cs.lmu.edu/~ray/notes/introwindows/(Accessed: 25/08/2020).

Appendix

First, the Valorant game must be downloaded and installed on a Windows 10 x64 up to date patch from here:

https://playvalorant.com/en-gb/

We provide the account we have performed our testing on : (ACC : bhammscproject01 / Pass: Pentest123)

Reboot after installation and check that Vanguard is present in systray/game can be played.

Under oxb509 – mod-msc-proj-2019:
Process explorer + process monitor software used for checking network services and identification of vgc.exe and riot servers connection + handles checking (process hacker can be downloaded from here : https://processhacker.sourceforge.io/)
Anti-cheat files folder contains the driver, the dll and the executable of the anti-cheat. These can be easily inspected for static analysis via dependency software available here:
https://github.com/lucasg/Dependencies
And PE studio here**: https://www.winitor.com/features**
DSEFix spoofer – extract&run the executable inside the zip archive then try loading the self signed hello world kernel driver in the kernel debugging folder/ driver/KMDF Driver1/ x64/ Debug/ KMDFDriver1.sys via sc service:
First create the service in an administrator cmd :
sc create service helloworld binPath= "FULL/PATH/TO/Driver.sys" type= kernel start= demand
to load driver:
sc start helloworld
This should fail if testsigning mode is turned off/ should work if testsigning mode on.(by default test signing is off, in order to turn it on one needs to restart, disable secure boot in bios boot options, then write in an admin cmd : bcdedit /set testsigning on after which another restart is required)
Warning, if hdd is bitlocked one may need its key after disabling secure boot.
TDL.zip – different DSE spoofer, similar to previous spoofer

Extreme injector is used for initial dll injection tests – select browse to find the offsettest.dll in manualmapping + simpledll/ simple dll solution/ x64/debug /offsettest.dll , for different settings change injection settings in settings->injection method
Manual mapping dll : executable present in x64/debug won't work most likely because path to dll to inject is hard coded,so after locating both the simple dll and manual mapping dll, open the solution in visual basic 2019 community edition and edit szDllFile[] in main.cpp to full path with double \\ back slashes in between folders , for changing process name change szProc[] to target process then build solution and run the executable after opening game by default or if changed szproc whatever target process needs to be opened first.
MouClassinput injection – kernel driver to simulate real mouse inputs – repeat process of create service then start with sc showed in dse spoofer but this time for the driver located in Mouclassinputinjection / bin / x64/debug MouClassInputInjection/MouClassInputInjection.sys
Then open powershell in MouiCL folder, type .\MouiiCL.exe. – instructions to use should appear
Networking: packet captures are created with wireshark, use wireshark to open them, go in Edit->Preferences->Protocols->TLS-> pre master secret log file name Browse-> add premaste.txt or alternatively set up environmental path variable of sslkeylogfile and monitor manually
Triggerbot: open game up , go to settings ->general-> Enemy outlier color-> set to purple
Join a game, run the executable triggerbot.exe and wait for text to appear, then switch to valorant and don't press left click and aim at enemy, python script is provided, Pillow library needs to be installed alongside ctypes and numpy
SeDebugprivilege : by default the console application needs to be run from power shell with -pid 1234 , tag where 1234 is the process id which can be found via process hacker/explorer of the target process to enable SeDebugPrivilege, more information can be found on Microsoft documentation, code referenced from there.

PPLkiller is run in the same way as DSE spoofer, and the driver is present in bin folder under PPLKiller-by default removes protected process light in all processes(that are protected)-> to be run after the game has launched.
Pics contains various screenshots we have taken during testing phase.
LiveKD contains the livekd utility for kernel debugging. To use, place livekd64.exe in the same installation folder as WinDbg. Then run as admin a cmd console and navigate to that folder, then type livekd64.exe -w and a live local kernel debugging WinDbg session should show up. For anti-rootkit extension check github “wdbgark”