Fortifying Game Security: Unveiling Vulnerabilities in Valve Anti Cheat and a Novel User Mode Anti-Cheat Solution

Saqif Ayaan Sudheer

Abstract

Video game hacking, or cheating, has emerged as a significant issue over the past decade. To tackle this problem, various video game companies have implemented anti-cheats, a method of malware analysis designed to identify and ban players using hacking software. While some companies have updated their anti-cheat software with kernel drivers to detect almost all forms of memory manipulation, others have stuck to their rudimentary anti-cheats operating in user mode. Valve, the company behind the popular game Counter-Strike: Global Offensive, is an example of the latter. The objective of this paper is to demonstrate how the game can be hacked and its memory exploited by creating a custom hack. Subsequently, I will conduct an in-depth analysis of VAC to pinpoint its vulnerabilities. I will then explain how I successfully circumvented its security measures and provide guidance on how to patch this bypass. The ultimate goal is to construct a comprehensive guide on enhancing VAC's robustness by proposing novel methods to strengthen the system.

Introduction

During the past decade or so, the video game hacking scene has grown tremendously. What first started off as simple triggerbots for retro games like Quake has evolved into complex aimbots and ESPs (extrasensory perception hacks ex. Visual aids that let you see an enemy when you are not supposed to). In fact, in just the game CSGO (Counter-Strike: Global Offensive), almost 9,000 players get banned due to cheating every day [4]. However, as cheats have become more and more prominent, so has the prevalence of anti-cheats and the level of security they provide.

There are two primary types of anti-cheats used in the industry today: anti-cheats with kernel drivers and anti-cheats that run solely in the user mode. To understand the difference between the two types of anti-cheats, we must first understand two distinct levels of protection rings in a computer running on x86/x64 processor architecture: ring 0 and ring 3. Ring 3 (user mode) is where most applications run, and the memory read and write permissions for each application are limited to what the system API can provide, meaning multiple checks are set in place so if an application were to crash, the entire system would not go down. In ring 0 (kernel), however, users can employ drivers that have direct access to all processes' memory on a PC. This means that a driver in the kernel space has read and write access to all memory with no bounds in place. This means that if a kernel driver causes a fatal error, there are no backup measures to ensure vital Windows processes won't go down, meaning the entire system would crash rather than just the application with the kernel driver [6]. In this paper, I will be analyzing

an anti-cheat of the latter type called Valve Anti Cheat. It is the primary and only source of protection for CSGO, a game that generally boasts around eight hundred thousand concurrent players at any given time [5].

Game Specifications

Counter-Strike: Global Offensive, otherwise known as CSGO, is a first-person shooter that revolves around one primary game mode. In this game mode, team A attempts to plant a bomb while the other team, team B, attempts to stop team A or defuse the bomb after it has been planted. CSGO requires critical thinking and mechanical aptitude when it comes to planning attack and defense strategies, as well as good mechanics when it comes to aiming at enemy players. However, the complex factors that define CSGO can be simplified by the unfair use of cheats and hacks. In this paper, we are first going to implement an external glow hack, which functions as an augmented sensory perception device. This tool provides users with an unfair advantage by revealing the precise locations of all opposing players, effectively removing the strategic elements of the game and granting users a significant edge in their gameplay.

Valve Anti Cheat specifications

Valve Anti Cheat, otherwise known as VAC, is a client-sided anti-cheat that scans memory on the user/client's pc. Its cheat detection scope is limited to what it can perform in user mode with the assistance of Windows API function calls. There are many checks that VAC employs, but three of the main ones include signature scanning to see whether the code of a given program matches the signature of any blacklisted cheat programs already existing in a database, hooking Windows API function calls, and detecting hooks to game or VAC modules from external programs. The third check is not applicable in this paper, since my hack will not require any form of hooking or detouring, but what does it even mean to hook something? A hook or a detour is when data flow in the form of a function or even a whole module is redirected to another program [1]. Once you receive the bytes of the given function, for example, you can even patch them to perform a different action then return control flow to the address right after the function you just patched. Many cheats employ this method, and anti-cheat systems also utilize it, not to modify bytes directly, but rather to monitor the invocation of specific functions or their intended actions. In this paper, I will attempt to make VAC more robust and capable of detecting hacks developed both in the user mode and kernel space where users can bypass all system call hooks by developing their own functions to do the job of existing Windows API functions.

Significance of This Paper

The novelty of my anti-cheat lies in the implementation of a user mode solution capable of neutralizing both user mode and kernel-mode hacks, without the necessity of employing a kernel driver-based solution. While the latter approach is typically considered more secure due to its direct memory access, which can effectively detect all forms of hacks in both ring 3 and ring 0, it carries a high degree of invasiveness. If mishandled, such a method can potentially grant unauthorized permission to access and manipulate data on a user's computer, raising significant privacy and security concerns. Kernel anti-cheats such as Easy Anti Cheat and Vanguard have come across a lot of backlash because of the security threat they pose if their kernel drivers are mishandled. User Shun-Pie has addressed this issue by making two very lengthy posts on reddit chastising the invasiveness of both these anti-cheats, but especially Vanguard, an anti-cheat that boots up with your computer and runs even when the game it is meant to protect, Valorant, is not running. His post just addressing Vanguard has garnered support from over 15,000 individuals [7]. Another one of his posts regarding kernel anti-cheats in general, including Easy Anti Cheat, has garnered support from over 3,000 individuals [8]. Furthermore, there have been numerous articles published on various blogs, such as Wired and Linustechtips, highlighting the security concerns surrounding anti-cheats with kernel drivers.

However, my proposed solution will take care of this issue while still effectively combating many kernel driver-based hacks. Rather than focusing on detecting cheat softwares before chaos ensues, my proposed solution focuses on analyzing game data to see if a cheat software has manipulated anything it's not supposed to. VAC attempts to do this by checking if a cheat software calls any Windows API function calls but a hacker could easily bypass this by developing their own functions to read and write process memory in the kernel that don’t require the user to open a handle to the game process or do anything explicit that could alert VAC. Nonetheless, regardless of the tactics cheaters employ to evade the checks imposed by VAC, they will find it impossible to bypass the integrity check I propose. This check does not rely on the specific type of cheat being used but rather on what it alters within the game's memory. As long as the cheat seeks to illicitly manipulate game memory, it can and will be detected. The only possible method to bypass my proposed solution would be to refrain from altering game memory altogether or to hook and spoof all VAC modules responsible for instating my solution, a very complex bypass that is out of the scope of this paper.

The ESP hack

What it is:

The ESP hack acts as an extrasensory perception tool, granting users an unfair advantage by revealing the precise positions of opposing players. This hack will be integrated as an external application, while the ESP functionality itself will be implemented internally. This means the hack will function as an independent program, making use of the Windows API to communicate with the kernel for read and write memory access to the target game process.

However, the ESP will be executed internally through the exploitation of an existing game feature referred to as "spectator glow," as an alternative to crafting an external overlay, which would necessitate access to the game's model and view matrices for the transformation of a player's model into both world and camera space [9]. This procedure typically involves hooking into graphics APIs such as DirectX, thereby resulting in significant performance overhead, which is why we will be using internal glow to implement the ESP hack.

Theory:

Many games, such as CSGO, opt for dynamic memory allocation on the heap due to the vast amount of required memory, which cannot be accommodated on the stack. This leads developers to employ the use of pointers, multi-level pointers, and pointer chains to manage this heap memory efficiently. However, unlike static addresses that consistently load into the same memory space, dynamically allocated objects are assigned different addresses each time the game is launched. Since games heavily rely on a class-based structure to organize information, like client entity (me) -> inventory -> money, the addresses of these dynamically allocated objects or classes tend to shift. However, the base of these pointer chains exist within a module (executable or dynamic link library) whose base address may match the preferred image base (where the module should ideally load in virtual memory) or can be identified using Windows API functions. Starting from this module's base address, we can traverse the pointer chain using the following process:

1. Offset the module's base address by a given number of bytes until we reach the static pointer (generally a class object pointer) that defines the beginning of the pointer chain.
2. Dereference the (class object) pointer which points to some dynamically allocated memory on the heap.
3. Offset this object's address by a given number of bytes until we reach the next pointer we want that points to some dynamically allocated memory.
4. Dereference the (class object) pointer and repeat the process until we reach our desired destination.

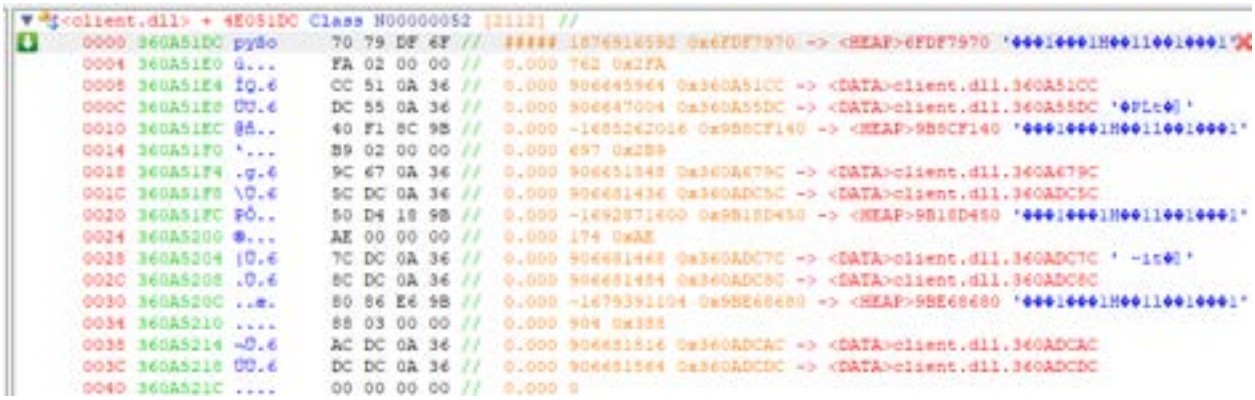
Process:

Now that we have the foundations of game memory down, we can start implementing the ESP hack. To implement this hack, we need to identify the memory address responsible for storing the spectator glow values for a specific player and overwrite them with our own values. This can be done through the use of 3 tools called Cheat Engine, ReClass.NET, and IDA Pro along with a supplemental tool called a dumper.

Cheat Engine was used to identify the current game process's "my entity" address by monitoring which addresses were updated with my health variable until I could narrow it down to a few addresses. From here, I backtracked to resolve the base of a pointer chain that started

from the client.dll module, since I knew that this dynamic link library held the entity list. Now, I had a list of offsets that when added to the base address of client.dll could be the starting point of the entity list.

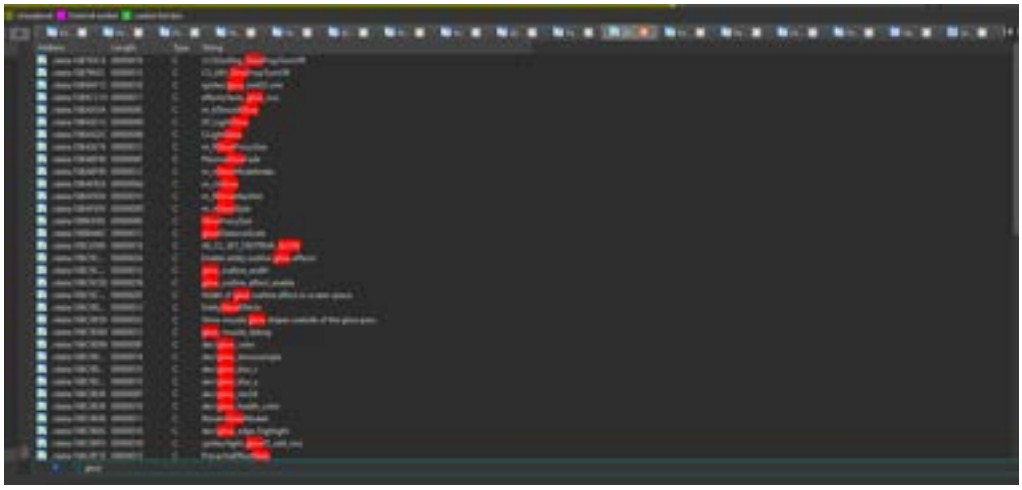
Then, I took to ReClass.NET to analyze the addresses at each of the offsets until I ran into this:



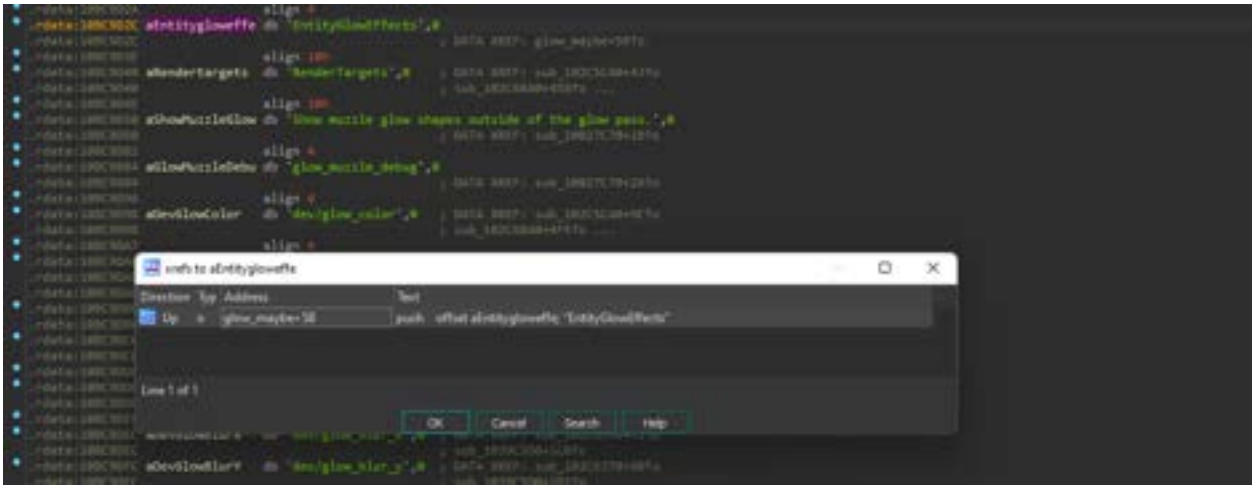
There was a pattern of pointers to the heap that matched the exact number of entities in my game. Additionally, new pointers to the heap would appear at addresses such as 0x360A521C as I added new entities to the world. This was clearly an array of entity object pointers, so I had found my entity list. From here, I dereferenced the first pointer to the heap since that was the local player entity(me) and derived the team (ct or t) offset through further analysis of game memory and trial and error.

Now, it's time to introduce the dumper. To determine the remaining offsets, I used a dumper created by frk1 on GitHub, known as the Haze Dumper. This tool essentially scans for byte patterns associated with strings (e.g., "m_iTeamNum") and retrieves the offset used by instructions that reference this specific set of bytes. The term "dumper" comes from the fact that after running the program while the game is active, it collects offsets and their corresponding string names, and then “dumps” them in a file.

However, I still needed to identify what string I should be looking for in the dump file that corresponds to CSGO’s internal spectator glow. To solve this issue, I took to IDA Pro where I statically analyzed the client.dll module’s binary file by both disassembling it and decompiling specific sections to comprehend the low-level code structure. I then initiated a string search for the term "glow" to see what relevant results pop up:



From here I went through multiple subroutines(functions) with the string “glow” to try and find something pertaining to player glow. Eventually I came across a data offset characterized with the string “EntityGlowEffects”.



(I replaced the original address with glow_maybe for easy access in the future.)

Unfortunately, Haze Dumper did not have any offsets for “EntityGlowEffects”, so I decided to jump to the subroutine that cross referenced this data offset and decompiled it. In the decompiled subroutine, I noticed that “EntityGlowEffects was being called by via another subroutine in a virtual function table:

```
(*(void (__thiscall **)(int *, int, const char *))(*v6 + 0x240))(v6, -682993, "EntityGlowEffects");
```

I tried finding the base of the virtual function table but was unable to do so, so I decided to view any cross references to the glow_maybe function I was currently in, and I came across another function that did this:

```
  return glow_maybe(v5, v2, a2, v7);
```

The glow_maybe function’s value was being returned with four parameters. Among these, one seemed related to the entity object (v5, which I later discovered to be dwGlowObjectManager),

while the others represented RGB values (red, green, blue). This was enough evidence to indicate to me that “EntityGlowEffects” was indeed pertinent to my search. Additionally, by examining analyses of other Source Engine games like Team Fortress 2, I realized that "EntityGlowEffects" was a string reference within dwGlowObjectManager. Haze Dumper did have an offset for this class, which housed an array holding all the glow objects for entities in the entity list.

The first step in coding the ESP involved receiving the module base address of “csgo.exe” and establishing a handle to CSGO, enabling both read and write memory access through the Windows API. From there, I followed the same four-step method I previously mentioned for navigating a pointer chain, although this time utilizing the ReadProcessMemory (RPM) Windows API function to follow and dereference pointers, as demonstrated in the example below:

```
//dereferencing the first entity address in the entity list to get access to local player(me) struct
uintptr_t localPlayerEntityAddr = 0;
ReadProcessMemory(gameHandle, (LPCVOID)(entityListAddr), &localPlayerEntityAddr, sizeof(localPlayerEntityAddr), NULL);
```

As I iterated through all the entities in the lobby and gathered their team values and glow indices, I was able to segregate the enemy players and locate their positions in the glow object manager array by multiplying the glow index by 0x38, which represents the number of bytes reserved by each glow object. Inside each enemy's glow object, I utilized the WriteProcessMemory (WPM) Windows API function to replace the existing ARGB float variables that dictate the glow's color. I also manipulated an occluded boolean to ensure that the glow would render even when players were concealed from the local player's (me) line of sight. The glow enablement process is depicted below:

```
//checking if current entity's team is different from mine, to know if glow should be activated or not
if (currEntTeam != localPlayerTeam) {
    glow currGlowObj(255.0f, 0.0f, 0.0f);
    WriteProcessMemory(gameHandle, (LPVOID)(currGlowObjIdx + 0x8), &currGlowObj, sizeof(currGlowObj), NULL);
}
//keep glow active at all times even when player is occluded from local player's vision
bool whenOccluded = true;
WriteProcessMemory(gameHandle, (LPVOID)(currGlowObjIdx + 0x28), &whenOccluded, sizeof(whenOccluded), NULL);
```

To optimize the process, I decided not to use WPM for each individual ARGB value. Instead, I organized all the values into a class named glow and instantiated an object of this class named currGlowObj. Because I knew that each of these values were stored contiguously in memory, by writing 16 bytes (4 floats for A, R, G, and B each of which has a size of 4 bytes) at once, I could significantly reduce the performance overhead that would have been incurred by performing WPM four separate times for each value.

Why it should bypass VAC:

As I mentioned before, VAC focuses on three primary criteria to detect whether a program is benign or malicious. However, the most relevant and practically applicable aspect is

the first one. VAC scans the signature of a process's opcode and cross-references it with a database of known malicious signatures, including popular publicly available cheat applications. In the case of my privately developed ESP hack, where the source code has been kept confidential and shared only with a select group, VAC has no access to my code and is unable to add it to its list of blacklisted signatures. In essence, this should afford me full immunity against game bans, otherwise known as VAC bans.

Testing:

In order to evaluate the effectiveness of my program in bypassing VAC, I conducted a thorough test by running my program for a total of 5 hours on VAC-secured servers. I introduced an hour-long interval between each session on the main lobby page to eliminate the possibility of immediate bans while allowing the anti-cheat system ample time to assess my program as either benign or malicious.

Upon completing the day of testing, I noted that no VAC bans had been issued to my account. However, it's worth noting that VAC often delays bans to gather intelligence on the behavior of cheaters and to target larger cheating communities rather than individual users. To ensure that there were no delayed VAC bans associated with my account, I waited for a period of 4 weeks. At the end of this waiting period, I confirmed that neither VAC bans nor game bans had been applied to my account.

In accordance with my results, I find it fair to say that my ESP hack successfully circumvents any VAC bans, whether immediate or delayed, and that it successfully gives users an unfair advantage with no repercussions.

Memory integrity scanner:

What it is:

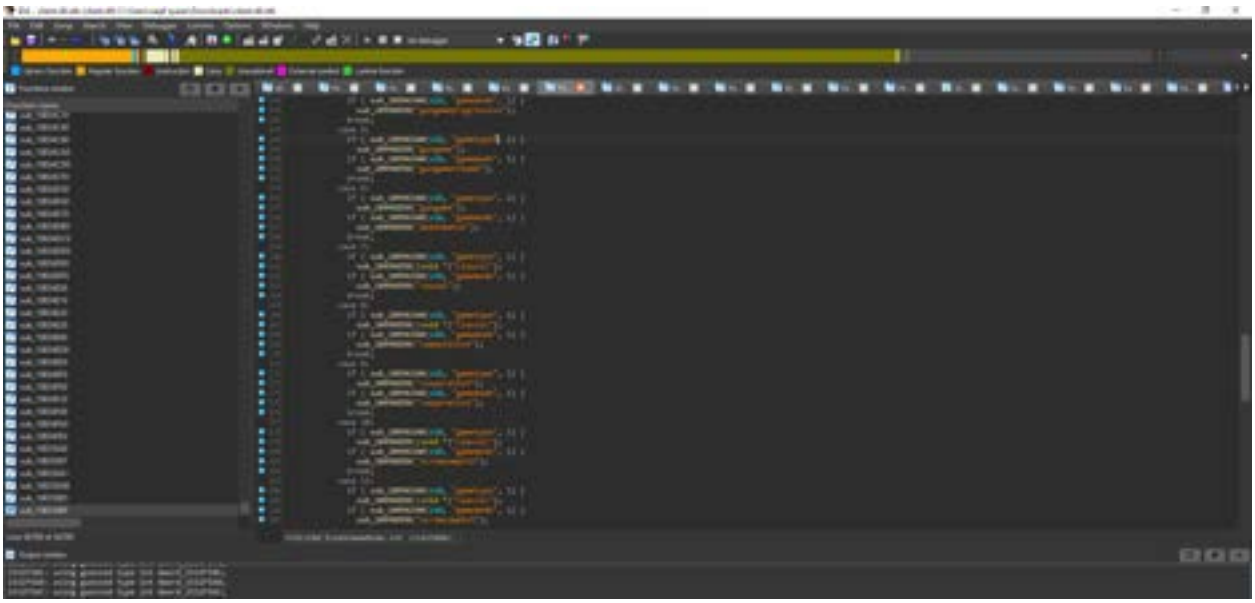
The Memory Integrity Scanner is a sophisticated C++ program meticulously engineered to identify hacks, including the ESP hack mentioned earlier. It works by monitoring game memory passively until it detects any anomalies, at which point it becomes an active process. These anomalies are determined when a memory value deviates from the expected parameters, which are continually updated in real-time. Such anomalies typically occur when external programs (ex. cheat applications) attempt to overwrite a variable's value within the memory. Once the program shifts into active scanning mode, it tallies a specific number of anomaly positives, and if this number goes beyond a given threshold, a user-kick message will be initiated. In the context of this paper, we will demonstrate its application for detecting anomalous values within a client's memory pertaining to the players' glow structs. If such anomalies are identified, the program will prompt a user-kick message. If this program is integrated into the

official VAC module, any client found in violation will be officially flagged as a cheater and banned at the discretion of VAC's established protocols and policies.

Process:

The skeleton of this program is exactly the same as the ESP hack. Both require the same traversal of the entity list pointer chain and access to the dwGlowObjectManager array. However, in contrast to the ESP hack, which alters the ARGB values within each player's glow structure, the memory integrity scanner focuses solely on reading these values and comparing them against expected values. Furthermore, we will need to establish new offsets for the client entity's life state and the game mode. This adjustment is necessary because the application of spectator glow is restricted to casual matches when the client entity is dead in the given round.

To find the life state offset, I utilized Haze Dumper, and for the game mode offset, I turned to IDA Pro. After doing extensive static analysis, I came across this :



It was evident that this subroutine was related to the classification of a game mode, potentially using the value stored in the variable v18. Unfortunately, when I tried putting a breakpoint in a debugger with the address of the subroutine, I did not come across anything substantial that could hint to me that an offset with a given game mode was being pushed onto the stack. Instead of spending more time to find an offset for game mode, I decided to move on with just one parameter, life state as follows:

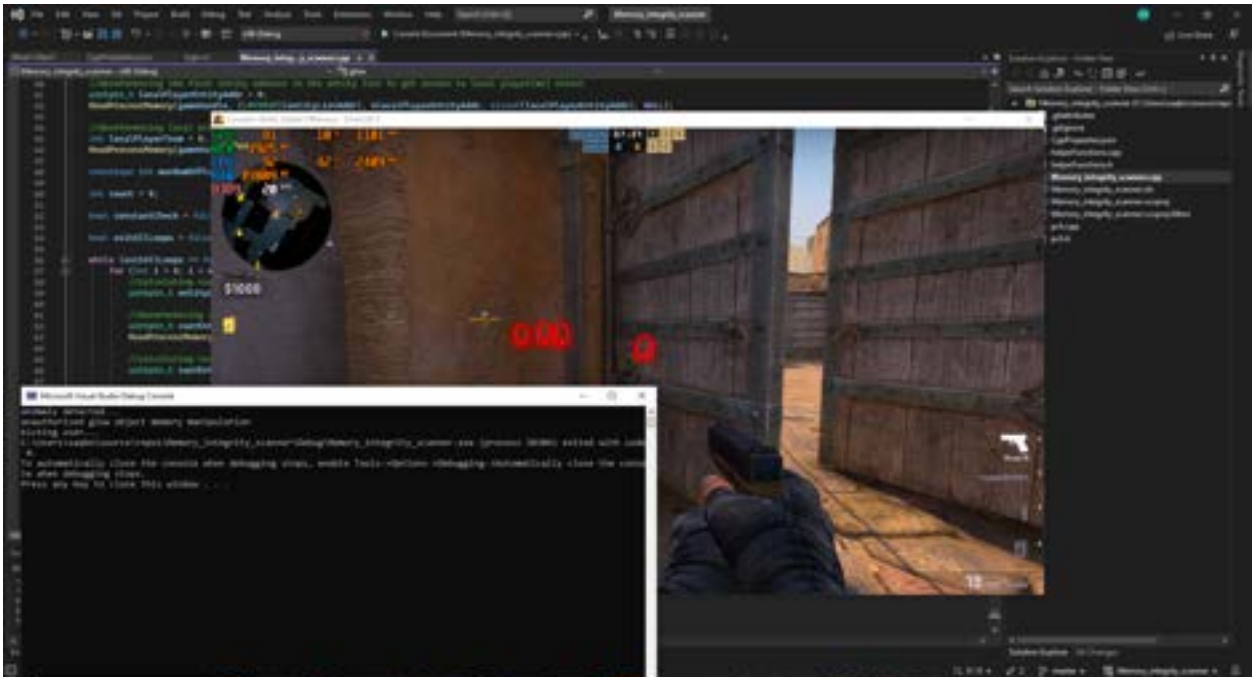
```
//checking if entity is dead or alive (0 means alive, 1 means dead)
if ( lifeState == 0.0f ) {
    //DISCLAIMER: if using this to check memory integrity of non-casual gamemodes, omit the lifestate check, because spectators should not see enemy
    if (enemy == true && (currGlowObj.r != 0.0f || currGlowObj.g != 0.0f || currGlowObj.b != 0.0f || currGlowObj.a != 0.0f)) {
        if (constantCheck == false) {
            std::cout << "anomaly detected..." << '\n';
        }
```

Here, I first made sure the client entity was alive by making sure the value of the life state variable was 0. Then I checked the values within currGlowObj (A, R, G, and B) to make sure they were all 0 (deactivated), and if they weren't, I would issue a positive scan with an “anomaly detected” string.

While testing my program, I came to the realization that, like other external applications, my scanner was introducing a significant performance overhead. This overhead could be efficiently mitigated by adopting a passive scanning approach at predefined intervals. I implemented a process where my program would pause for 5 milliseconds between scans. However, if the client was detected making unauthorized modifications to any entity's glow struct variables, I would initiate active scanning with only a 0.1 milliseconds pause between scans. Once 100,000 positive scans were reached, a user-kick message would be prompted, and the user would be confidently flagged as a cheater.

Testing:

To assess the efficacy of my memory integrity scanner, I ran it in parallel with my ESP hack to determine if it could identify what VAC had missed. I initiated this test four weeks after my initial evaluation of the ESP hack, ensuring that my ESP was truly undetectable and that my memory integrity scanner could assess a wider range of hacks obscure from VAC. The final result looked like this:



From this image, it is clear that my ESP hack was working successfully with no VAC alerts or game bans, while my memory integrity scanner (bottom left) successfully found anomalies in the client’s game memory and flagged the client as a cheater (user-kick message).

Conclusion

Ultimately, it is evident that VAC has critical vulnerabilities that become exposed through the private development of cheats that exploit the protected game's memory. Nevertheless, bypass methods like mine, which rely on unique cheat signatures, can be effectively countered with my proposed anti-cheat solution. By scanning the client's game memory to distinguish between anomalous and legitimate values, my solution not only advocates for increased efficiency but also employs a robust detection methodology that resists easy circumvention, requiring a more extensive effort to hook and spoof the entire module housing my anti-cheat system.

The novelty of my proposed anti-cheat solution comes from the fact that it can detect both user mode and kernel driver-based cheats as a solely user mode based program. Since my anti-cheat does not rely on reading a general process's memory, such as the cheat software, to determine if it is safe or unsafe, users cannot simply circumvent my anti-cheat by flying under the radar in the kernel, developing their own RPM and WPM functions, or using polymorphic code to evade signature detection. My anti-cheat directly and continuously monitors the client's game memory, making it resilient against various evasion tactics. Regardless of the countermeasures users employ, as long as their cheat program alters game memory in an unauthorized manner, it will be detected.

For future works, I wish to see the adaptation of my anti-cheat methodology for safeguarding against a broader spectrum of cheats, including aimbots and radar hacks. While this paper predominantly addressed its efficacy in detecting ESP hacks, it's essential to emphasize that the methodology I've developed can be extended to counter any type of hack, provided that the attack vector involves unauthorizedly overwriting the game process's memory.

Acknowledgements

I would like to acknowledge Ross Greer and Priya Srikumar, both Ph.D. candidates studying at UC San Diego and Cornell, respectively, for helping make this paper a reality.

Bibliography

1. Grinberg, Shiran. "API Hooking - Tales from a Hacker's Hook Book." *Cynet*, 7 Apr. 2023, www.cynet.com/attack-techniques-hands-on/api-hooking/#:~:text=This%20is%20called%20Hooking%E2%80%94the,is%20not%20always%20the%20case. Accessed 24 Oct. 2023.
2. Baker, John. "Understanding the Differences between Obfuscation and Encryption." *DESkey*, des.co.uk/blog/difference-obfuscation-and-encryption#:~:text=Encryption%20provides%2

0confidentiality%20for%20sensitive,just%20more%20difficult%20to%20understand. Accessed 24 Oct. 2023.
3. Kotwani, Bharat. “VAC Detects a Popular CSGO Cheat, Various Users Banned in the Aftermath.” *TalkEsport*, 17 May 2021, www.talkesport.com/news/csgo/vac-detects-popular-csgo-cheat/. Accessed 24 Oct. 2023.
4. Kotwani, Bharat. “300k CSGO Hackers Banned in a Massive Vac Wave.” *TalkEsport*, 26 Sept. 2022, www.talkesport.com/news/csgo/csgos-free-to-play-status-has-attracted-cheats-but-valve-has-ignored-the-problem/#:~:text=According%20to%20CSGO%20numbers%2C%20the,is%20a%20really%20large%20figure.
5. “Counter-Strike 2 Live Player Count and Statistics (2023).” *The Game Statistics Authority : ActivePlayer.Io*, 28 Sept. 2023, activeplayer.io/counter-strike-global-offensive/#:~:text=There%20are%20about%201%2C063%2C668%20people,on%20all%20platform%20it%20supports.
6. Sercan, Sari. “What Are Rings in Operating Systems?” *Baeldung on Computer Science*, 11 June 2023, www.baeldung.com/cs/os-rings.
7. “Why Valorants Vanguard Anti-Cheat Has to Be Changed ASAP.” *Reddit*, www.reddit.com/r/pcgaming/comments/g2zu1c/why_valorants_vanguard_anticheat_has_to_be/.
8. “Root Level Anti-Cheat Is Getting out of Hand - Again.” *Reddit*, www.reddit.com/r/pcgaming/comments/y5jvzf/root_level_anticheat_is_getting_out_of_hand_again/.
9. “The View Matrix Finally Explained.” *Game Development Stack Exchange*, 1 Apr. 1966, gamedev.stackexchange.com/questions/178643/the-view-matrix-finally-explained#:~:text=%22The%20View%20matrix%20converts%20from,vertices%20in%20camera%2Fview%20space. Accessed 24 Oct. 2023.