A Comprehensive Analysis of Game Hacking through Injectors: Exploits, Defenses and Beyond

Francis Martinson
Computer Science
North Dakota State University
Fargo, North Dakota

Dylan Rangel
Computer Science
North Dakota State University
Fargo, North Dakota

ABSTRACT

The emergence and rapid adoption of online gaming have resulted in massive multiplayer games and virtual landscapes that enable players to interact in real-time within a digital ecosystem. Competitive gaming has led some players to succumb to the temptation of employing illegitimate methods to achieve an unfair advantage over others, thereby compromising the gaming experience's authenticity and fairness. Game hacking is a pervasive problem that challenges the industry and raises crucial questions about the extent and the implications of such behavior in the gaming community. One of the most common methods employed by hackers is the use of injectors to compromise a game's code and modify its structure. As sophisticated software defenses evolve, questions arise as to how effectively developers can deter and combat hackers.

This research discusses the mechanisms behind injectors, their methods of exploitation, and the potential strategies employed by game developers and anti-cheat systems to prevent hacking.

General Terms

Injectors, Exploits, Game Hacking, Defense Mechanisms, Software Development, Game Security, Anti-Cheating Measures, Code Injection, Cybersecurity.

Keywords

Consoles, injectors, dynamic link libraries, hackers, developers, modding, Real Money Trade (RMT), Piracy, Game Hacking.

1. INTRODUCTION

From the text-based MUD associated with the creation of ARPANET in the 1980s to modern-day graphics like World of Warcraft and League of Legends, online gaming has dramatically evolved over the past decades (Newzoo, 2018). This evolution was systematically fostered during the internet explosion in the late 90s and the rise of technology in the early 2000s (Rouse, 2020).Online gaming has evolved considerably since its inception in the early 1990s. The rapid proliferation of the internet and computing technologies has resulted in unprecedented growth within the industry, leading to mainstream adoption via platforms such as consoles, PCs, and smartphones. Concurrently, this evolution has urged users seeking an unfair advantage to develop innovative ways of hacking game code. One such method, "injectors," enables hackers to modify the data or code of a game, thereby allowing them to cheat and gain an unfair advantage over other players.

The evolution brought alongside several instances of exploitation. Gaming platforms became rich hunting grounds for cybercriminals who capitalized on users' vulnerable security defenses to steal delicate account information. Exploitation is poisoned with phishing schemes, malware threats, and other avenues for identity theft (Gramigna, 2019). Overwhelmed by the wave of exploitation and its impact on the gaming community, developers engendered rigorous security protocols to thwart these threats.

The usage of injectors dates back to the 1980s, with the emergence of computer viruses and worms designed to infiltrate and compromise the integrity of systems. These initial attempts at injection laid the foundation for interference in secure domains and the unauthorized extraction of confidential data (Mansfield-Devine, 2009). Hackers use software injectors to modify a game's code and give themselves unfair advantages (Cimpanu, 2019). For example, in a shooting game, hackers can use injectors to manipulate aim, speed, and immortality, destroying the game's integrity.

To combat these exploitations, game developers have dedicated substantial resources to enhance game security and mitigate exploitation and hacking risks. Anti-cheat software becomes the first line of defense to guard against game hacking, translating into more secure gaming environments and a more level playing field for participants (Gizmodo, 2018).

2. BACKGROUND

Injectors have become an indispensable tool in modern gaming, as they serve to enhance, customize, or manipulate game content, thus providing players with a remarkable gaming experience.

This section reviews injectors in games, the wide variety of uses and techniques employed, the benefits and drawbacks associated with their use, and how they have paved the way for innovation and creativity within the gaming community.

2.1 Mechanics of Injectors

Injectors are a form of hacking tool that manipulates and injects code into the memory space of a specific target process (i.e., the game software). Game injectors work by modifying the assembly instructions or game code executed by the central processing unit (CPU). The injected code can alter the game's behavior by manipulating existing code and function calls, or otherwise "injecting" arbitrary code, providing the hacker with advantages, such as invincibility, infinite resources, or accelerated movement. This code injection process typically occurs during the loading and execution of the game software. The injectors can be standalone executables or dynamic link libraries (DLL) that contain functionality enabling the insertion of foreign code into the targeted game's memory space.]

2.2 Exploitation and Application

There are various techniques that injectors use to exploit a game's software. A prevalent method is the injection of DLLs, which involves utilizing Windows-specific features that allow

DLLs to be loaded and executed in the game's memory space. Hackers use tools called "loaders" to execute the DLL injector, which subsequently loads the target DLL file into the game process. Another less-known method is the use of "hooks," where the hacker intercepts and alters the code flow of the target process, essentially changing the game's behavior at predetermined points.

Regardless of the injection method used, once the foreign code is injected and executed within the game's memory, it can alter the game's flow or logic, providing the hacker with an unfair advantage. However, the degree of the unfair advantage varies, depending on the injected code's extent and sophistication. Some injectors may grant superficial benefits, such as those stated before, while others might result in destructive and disruptive activities, such as crashing game servers or flagrant, pervasive malicious hacking affecting user accounts.

2.3 Defenses and Strategies

As game hacking persists as a pressing concern, developers continuously aim to devise innovative anti-cheating mechanisms that prevent injectors' exploitation. Various standard practices include the use of code obfuscation, which conceals the game's code to make it more challenging for hackers to reverse-engineer or understand. Developers may also use integrity checks that evaluate and verify the game's software throughout its execution to confirm that no unauthorized code or data manipulations have tampered with the software.

Additionally, developers utilize real-time monitoring and anomaly detection techniques to track and flag suspicious activities within a game. This process enables the developers to detect and react to signs of injectors and other hacking activities during gameplay, safeguarding the experience for non-cheating players.

One notable example of a sophisticated anti-cheat system is "Vanguard" by Riot Games. Vanguard not only detects standard cheating techniques but also employs a kernel-level driver that operates at the operating system's most secure level. This driver scans and blocks potential hacking tools and techniques before they can inject or alter the game process.

2.4 Wide Variety and Uses of Techniques

There exist several techniques that hackers utilize when employing injectors. One such method is SQL (Structured Query Language) injection, a technique used by hackers to manipulate the data in a target's databases by injecting malicious SQL queries (Halfond, Viegas & Orso, 2006). This allows hackers to steal sensitive data from an organization or individual databases.

Injecting code into games can be accomplished in several ways, some of which include Dynamic-link library (DLL) injection, which entails a third-party DLL being forced to load into the game process, with the injector executing the code contained within the DLL to induce desired effects. This method allows the hacker to execute arbitrary code within the context of the target program, effectively gaining unauthorized access and control over the system (Arefin, Islam & Chy, 2018). Memory manipulation, which is where a game's memory addresses is altered to modify variables in the game, such as character health, currency, or speed. Script injectors: LUA, C#, C++ or Python injectors, like JASS or Pythonista, can be utilized to create scripts that enable custom game modifications.

2.5 Piracy , Exploits and Modding

Game hackers use cheats and exploits to gain unfair advantages over other players, often in competitive online games. These hacks can include Aimbots, which automatically target enemies for the player, wallhacks, which reveal enemy positions through walls and other objects, and speedhacks, which allow the player to move faster than intended (Bohannon, 2010). The use of these hacks can significantly diminish the enjoyment of other players and undermine the competitive nature of the game.

Game hackers may be involved in cracking games, bypassing copyright protection, and sharing the games for free, undermining the financial interests of developers and publishers (Yar, 2005). Some game hackers engage in modifying or "modding" existing video games to alter their content, graphics, sound, or other aspects of the game. While some mods enhance the gaming experience or provide entertainment, others create unwanted or inappropriate content that may harm the game’s reputation or violate the developer's intellectual property rights (Postigo, 2007).

Real money trade involves the buying and selling of virtual items, currencies, and accounts for real-world currency. Game hackers may exploit in-game mechanisms to obtain these virtual items and then sell them on various websites, profiting from the hard work of legitimate players (Heeks, 2008). The dark web provides an anonymous platform for game hackers to share their exploits, tools, and techniques. Hacking forums serve as a virtual marketplace, where hackers can buy and sell various tools and information related to hacking video games. These platforms allow hackers to refine their skills and collaborate with others, which can lead to the development of more sophisticated hacks (Kshetri, Voas, & Zhang, 2019).

3. METHODOLOGY AND TECHNIQUE DETAILS

3.1 Context

To develop an actual exploit in order to achieve the unfair advantage of drawing opponents through walls, the first thing that must be understood is the game’s engine. Counter-Strike: Source is powered by the Source Engine, an engine that developed by Valve in 2004 in order to display advanced calculations and predictions of physics, a technique called “occlusion culling” is used by this engine in order to improve performance and involves determining which objects in the game world are visible to the player based on the players position and where the player is looking at, this information is obtained by the server calling the functions GetEyeAngles(ply) and GetPos(ply), where “ply” is the current player that the engine is checking. In the context of Counter-Strike: Source, all players are checked each “tick” of the server, a tick refers to a single iteration of the game loop that updates the game state and checks for events such as user input. Counter-Strike: Source runs at a fixed tick rate of 60 ticks per second, which means that the game is updated 60 times every second, which in turn means that the functions GetEyeAngle() and GetPos() are called 60 times every second. and based on what these functions return to the server, the server will prevent certain objects from being visible to the player which will significantly reduce the number of resources and processing power that is required for the computer or system to run the game. The “wall-hack” that was created for the purpose of demonstrating the weakness of the engine and the anti-cheat system exploits this technique of occlusion culling by intercepting these function calls and manipulating the game’s rendering system. By modifying the engine’s code, the wall-hack can disable occlusion culling entirely for specific objects, such as player skeletons and player models which allows the player using the

cheat to draw these models through ALL objects, allowing the player to see opponents through walls. Because this hack exploits a weakness in the game's engine and model drawing rather than making unauthorized changes to the server or game's rules, it is very difficult to detect by the Valve Anti-Cheat protection system.

3.2 Injector

To successfully manipulate the Source Engine, first an injector must be developed to implement any arbitrary scripts that would be able to change the result of how the engine handles occlusion culling. For the most efficient injector, C# was the prime choice. This code defines a static class called "VACBypass" which contains static methods for interacting with the game process and the Windows API. The "Run" Function is the initial point of the bypass which takes the path to the DLL containing the malicious code.

```
public static bool Run(string pathToDLL)
{
    Init();

    pid = GetGamePID();

    if (pid == UInt32.MinValue)
    {
        throw new ApplicationException("The game was not found.");
    }

    hGame = OpenProcess(ProcessAccessFlags.All, false, (int)pid);

    if (hGame == IntPtr.Zero)
    {
        throw new ApplicationException("Failed to open process.");
    }

    BypassCSSHook();
    InjectDLL(pathToDLL);
    RestoreCSSHook();

    return true;
}
```

Fig 1: DLL containing the malicious code

This method initializes the global variables necessary for the bypass and then locates the running ID of the game, in this case the running ID would depend on the user's computer and the GetGamePID() function simply looks for the name of the game in memory and returns the process ID in windows back to the variable pid.

If the game process is found, then the Run method opens a handle to the process using the "OpenProcess" API function. Then it will call 2 methods on it, "InjectDLL()" and "BypassCSSHook". The BypassCSSHook function un-hooks several system functions which are used by the game and the Valve Anti-Cheat system, which prevents the game from detecting the injected DLL and flagging it as malicious code. The InjectDLL() method creates a remote thread in the game process and loads in the specified DLL file into the process using a load library function. Once the code has run successfully, the RestoreCSSHook() function is called to restore the original hooks and system functions that were previously disabled to prevent any suspicion from the system. Finally, the Run function will return a true value in order to indicate the injection process was successful.

A. Exploitation

Now that the injector has successfully injected into the Source Engine, the possibilities are endless as to what could be done. For this example, C++ will be injected into the Source Engine to exploit the way that the Source Engine handles occlusion culling. First, #pragma directives must be instantiated to modify behavior of the code, in this case they are modifying the compiler in order to suppress certain warnings generated by the compiler in order to make the code run properly.

```
4
5   #pragma clang diagnostic push
6   #pragma ide diagnostic ignored "hicpp-signed-bitwise"
7   #pragma ide diagnostic ignored "cppcoreguidelines-pro-type-member-init"
8   #pragma ide diagnostic ignored "readability-convert-member-functions-to-static"
9
```

Fig 2: Code Exploitation

Next, an initializer function is written to:

- Find the specified handle of the game window.
- Return the game ID (in this case it's the ID of the windows process, NOT the game ID).
- Open a handle to the running process.
- Calculate the memory addresses of the hack's feature in the game memory.
- Return the current state of the hack.

```
void init(const Config &config) {
    handleWindow = FindWindowA(nullptr, config.window.c_str());
    GetWindowThreadProcessId(handleWindow, &pid);

    processHandle = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION, FALSE, pid);
    uintptr_t clientBase = getModuleBaseAddress(const_cast<char *>(config.clientLib.c_str()));
    wallhackAddress = clientBase + config.offset;

    setState();
}
```

Fig 3: Malicious code injection

FindWindowA() returns the handle of the game's window. The code also uses GetWindowThreadProcessId() to return the ID of the windows process back to the code and store it into the pid variable, this will be important later. A handle is then opened to the game process using the OpenProcess() function with the parameters being PROCESS_VM_WRITE and PROCESS_VM_OPERATION which grants this program access to read, write, and execute operations on the virtual memory of the specified process, which in this case would be the id returned by GetWindowThreatProcessId() function or in simpler terms, the game. Next the code calculates the memory addresses of the exploit within the game's memory by calling the getModuleBaseAddresses() functions specified in the config object. It does this by first creating a snapshot of the target process using the CreateToolhelp23Snapshot() function which will take a snapshot of all modules in use by the specified process which in this case is stored in the config object. The only thing that the config object contains is the ID of the game that this code is being injected into. It will then look through the list of modules using the Modul32First() function and Module32Next(), checking each module's name using strcmp(), if it finds a module with a matching name it sets the moduleBaseAddress variable to the base address of the module and stops the loop. Finally, the function will return the variable and this address is then used in the initialize function in order to calculate the address of the hack feature within the module, this way the hack can be called no matter where it is in memory. Finally, the actual exploit code is quite simple, the complex part is accessing it and injecting it into memory which has already been completed.

```
foreach(ply in ply.Session){
    z = ply.GetPos();
    a = ply.GetEyeAngles();
    r_draw(z, a, 3);
}
```

Fig 4: Code to position eye angle

For each player in the current session, draw their position and the angle they are looking at, the "2" parameter is referring to what mode the model should be drawn as, 0 is typically the standard model with occlusion culling enabled, 2 is a wireframe with occlusion culling disabled, and 3 is the same as 2 although it is the standard model instead of a wireframe. This is the very core of the attack. The r_draw function is ran client-side so it is entirely undetectable by the server which is a massive vulnerability in the Source Engine.

Fig 5: Display with no exploit running

Fig 6: Display with exploitation

3.3 Ethics and Solution

The ethics of exploiting the Source Engine in this way is to demonstrate the dangers of leaving extremely important functions such as r_draw() available to the player, while inputting this command alone into the console will not allow a player to gain an advantage, if injected into the actual game via a DLL, the r_draw function can still be run clientside because all instances of this function depend on the user's graphical settings and are different from person to person, while this is good for maintaining each user's personal graphical preference, if left unchecked by the server, instances of this command can be used to exploit the game engine in order to run in unauthorized ways. There are many ways to fix this issue.

The first approach would be to make instances of r_draw() checked by the server, which would drop performance for the server but this exploit and many other like it would no longer be functional.

An additional software could also be added to the game directory in order to monitor the game's memory, if any other program aside from the game attempts to access the memory such as in the way the exploit did using the ByPassCSS() and InjectDLL() functions, the program auto terminates and issues a VAC ban to the user, preventing them from playing the game online. By monitoring the game's memory, the engine becomes a lot more secure.

Machine learning can also be implemented in game that keeps track of a player's statistics, and how often they manage to kill players through walls, since this could also be a luck, instead of issuing a ban to players that receive many kills through walls, the software could flag the players user ID and automatically record something called a "demo" that is sent to a real human employee for inspection to determine whether or not the player is cheating. A "demo" is a recording of gameplay that can be played back using the game's built-in demo player, each and every tick of a single game is saved to a .demo file during recording which provides a 3D replay in which the camera can be moved around independent from the players view instead of a simple .mp4 recording which allows for closer inspection of whether or not the player is cheating

4. DATA ANALYSIS

The data for this research has collected through a combination of primary and secondary sources, including interviews with game hackers and developers, online hacking forums, industry reports, scholarly articles, and news articles. The researchers also developed injectors (Ikariiillustration/injector (github.com)) that was used to test injection (on 5 iterations) and detection (on 10 iterations).

This methodology ensured a comprehensive perspective of the game hacking landscape.

4.1 Game Hackers

15 Questionnaires were sent out with invitations to be interviewed after the questionnaire is completed and returned. 14 respondents returned the questionnaires and 12 agreed to be interviewed. Out of the 12 respondents, all completed interviews via Zoom and Teams chat.

4.1.1 Respondent Demography

Respondents were randomly selected from a sample of gamers, ensuring unbiased representation and increasing the generalizability of the findings (Pew Research Center Poll, National sample study). This random selection process helped to minimize selection bias and allowed for accurate conclusions to be drawn.

Table 1. Respondent Demography

Number of Respondents		Gender		Average Age
	Third Party Software	Male	Female	
3	RuneScape	3	0	30.5
	Call of Duty (Modern War Fare 2019)	1	1	19.5
2	Valorant	1	1	19.5
1	Destiny 2	0	1	19.5
2	Guild Wars 2	2	0	20.5
2	Final Fantasy 14	2	0	33
12	Total	9	3	23.75

4.1.2 Game Categorization

Many of them utilized the main servers of popular gaming software such as RuneScape, Call of Duty, Valorant, Destiny 2, Guild Wars 2, and Final Fantasy 14. The survey findings indicated that approximately 83.3% of the hackers employed injectors, which are software tools used to modify game code and gain an unfair advantage. Furthermore, within this group, 66% of the hackers reported using either Macro/Hacker (33%) or Aimbot (33%) functionalities. These results highlight the prevalence of hacking practices in the gaming community and shed light on the specific games and tools commonly targeted by hackers.

Table 2. Game Categorization

Main/Local Server	Third Party Software	Injector	Software Type
Main	RuneScape	No injector	Macro-Software
Main	Call of Duty (Modern War Fare 2019)	Injector	Aimbot Wall Hacks
Main	Valorant	Injector	Macro-Software
Main	Destiny 2	Injector	Aimbot (Modify values)
Main	Guild Wars 2	Injector	Macro/Hacker
Main	Final Fantasy 14	Injector	Macro/Hacker

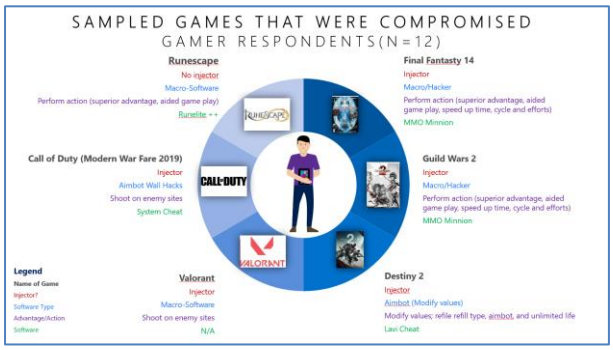

Fig 1: Sampled Games for Respondents

4.1.3 Game Hacking

All respondents in the survey reported using public gaming platforms and played games that had anti-hacker measures in place. Among the hacking software options mentioned, NMO minion was found to be the most favored by the majority of respondents. It was observed that a significant portion of respondents intended to utilize this hacking software to gain an unfair advantage in the games they played. Specifically, 50% of respondents expressed a preference for cheats and hacks that would enable them to achieve a superior advantage through actions such as aided gameplay. Additionally, 25% of respondents sought to exploit hacks that could speed up game time or reduce the effort required. Furthermore, 30% of respondents admitted to hacking in order to gain an advantage in shooting enemies on their side. These findings shed light on the motivations and preferences of hackers within the gaming community.

Table 3. Game Hacking by Software and Anti-Cheat Presence

Third Party Software	Advantage/Action	Software	Anti-Cheat?	Public
RuneScape	Perform action (superior advantage, aided game play)	Runelite ++	Yes	Yes
Call of Duty (Modern War Fare 2019)	Shoot on enemy sites.	System Cheat	Yes	Yes
Valorant	Shoot on enemy sites.		Yes	Yes
Destiny 2	Modify values; refile refill type, aimbot, and unlimited life	Lavi Cheat	Yes	Yes
Guild Wars 2	Perform action (superior advantage, aided game play, speed up time, cycle and efforts)	MMO minion	Yes	Yes
Final Fantasy 14	Perform action (superior advantage, aided game play)	MMO Minion	Yes	Yes

Based on the survey findings, it was determined that a significant majority (83.3%) of the cheating tools utilized by hackers were not available for free. Furthermore, a similar percentage (83.3%) of the hackers managed to avoid being banned, indicating a high success rate in evading detection. Interestingly, all of the hackers maintained a sense of normalcy while playing, despite having a superior advantage due to their cheating activities. This approach likely helped them avoid suspicion and scrutiny from other players and game administrators.

Among the hackers, there were distinct variations in their strategies. One hacker preferred to play exclusively with average gamers, potentially to minimize the risk of being detected. In contrast, two other hackers adopted a different approach by targeting chests, taking intermittent breaks, and then returning to cheating. This suggests a calculated and strategic approach to maximize their advantage while minimizing the chances of being caught.

Notably, 50% of the hackers intentionally avoided Player vs Player (PvP) gameplay, as engaging in such activities could subject them to consistent observation and scrutiny from other players and game administrators. This avoidance strategy likely helped them maintain a lower profile and reduce the risk of being reported or detected.

It is worth mentioning that one of the hackers in the survey was

eventually banned, and this occurred because their hacking activities were detected over an extended period. This emphasizes the importance of vigilance and effective detection measures in combating cheating and maintaining fair gameplay environments.

Table 4. Game Hacking Detection Avoidance Strategy

Third Party Software	Third Party Software	Free	Ban Yet?	Avoid Detection
Main	RuneScape	Depends on type of cheat	No	Play in line with average gamers (hrs./time/effort/advantage)
Main	Call of Duty (Modern War Fare 2019)	No	No	Aim for chest, intermittent breaks from chest, play on normal
Main	Valorant	No	No	Aim for chest, intermittent breaks from chest, play on normal
Main	Destiny 2	No	No	Avoided PVP (Player vs Player)
Main	Guild Wars 2	No	Yes	Avoided PVP (Player vs Player) || Detected for hacking too long
Main	Final Fantasy 14	No	No	Avoided PVP (Player vs Player)

4.2 Researchers' Experimental Injections

In the experimental discovery, it was found that the VAC anti-cheat system, designed to detect and prevent cheating in the game, failed to identify the exploit being utilized. Throughout each iteration of the experiment, the responsibility for detecting and addressing the exploit rested solely on the players themselves.

The specific game in question, Counter-Strike, has a mechanic where players who are eliminated in a round enter a spectating mode until the round concludes. During this time, the spectator could observe the perspective of the surviving teammate. This mechanic inadvertently provided an opportunity for exploiting the game by gaining an unfair advantage through viewing opponents' locations through walls.

In one of the iterations, a play tester was falsely accused of cheating and subsequently reported. This incident occurred because the play tester was observed looking at opponents through walls, an action that is not within the bounds of legitimate gameplay. It highlights the potential for misunderstandings and unwarranted accusations when exploiting such game mechanics.

Interestingly, in another iteration, the exploit was only detected when other players noticed suspicious activities, such as being killed by the player through walls. This suggests that the detection of the exploit relied heavily on the vigilance and observation skills of the other players rather than the effectiveness of the anti-cheat system itself.

This discovery underscores the limitations of the VAC anti-cheat system in effectively detecting and addressing certain exploits. It also emphasizes the importance of player awareness and reporting in identifying suspicious activities during gameplay. The findings highlight the need for continuous improvement and updates to anti-cheat systems to effectively combat cheating and maintain fair gameplay environments.

4.2.1 Counter-Strike Injection and Detection

It was observed that the successful functioning of the injector software relied on the game initializing specific hooks necessary for exploiting the game engine. The design of the injector software was specifically tailored to accommodate this requirement, ensuring compatibility and effectiveness.

Once the game initialized the exploitable hooks, the injection process consistently achieved a 100% success rate in executing the desired hack. This indicates that the injection method was capable of effectively manipulating the game engine to gain an unfair advantage throughout the iteration of the experiment.

Table 5. Success Game Injections

Successful Game Injection (1 = Successful; 0 = Failed)					
Mins Before Game Load	Try One	Try Two	Try Three	Try Four	Try Five
Immediately	0	0	0	0	0
0-1 min during game load	1	1	1	1	1
2-5 mins during game load	1	1	1	1	1
After game load	1	1	1	1	1
During run time	1	1	1	1	1

However, it is important to note that despite the initial success, the injection was eventually detected in 4 out of 5 iterations. The detection occurred on the 4th, 5th, 8th, and 10th attempts, all of which were made during the second attempts. This suggests that the initial injection went undetected, allowing the hack to remain active until subsequent attempts were made. Reports were made upon each detection, indicating the vigilance of individuals in identifying and reporting the presence of the hack.

Overall, the experimentation recorded an 80% success rate, indicating that the injection method was effective in bypassing detection in the majority of cases. However, the fact that the hack was eventually detected in a significant portion of the iterations highlights the potential vulnerability of the injection method and the importance of robust detection measures in maintaining fair gameplay environments.

This discovery underscores the ongoing cat-and-mouse game between hackers and game developers, where hackers constantly seek new methods to exploit game engines, while developers work to enhance detection mechanisms to identify and prevent such exploits. The findings emphasize the need for continual improvement in anti-cheat systems and the importance of player reporting in maintaining fair and balanced gameplay experiences.

Table 6. Cheat Detection and Reporting

Cheat Detection (5mins iteration; 5 mins incremental)						
Iterations	Try One	Try Two	Detection (Yes = 1; No = 0)	Time of Detection	Reported (Yes = 1; No = 0)	Action Taken Against You
One	0:00	5:01	0	N/A	0	N/A
Two	0:00	10:00	0	N/A	0	N/A
Three	0:00	15:10	0	N/A	0	N/A
Four	0:00	20:01	1	15:25	1	Reported
Five	0:00	25:04	1	20:02	1	Reported
Six	0:00	30:12	0	N/A	0	N/A
Seven	0:00	35:02	0	N/A	0	N/A
Eight	0:00	40:01	1	31:24	1	Reported
Nine	0:00	45:10	0	N/A	0	N/A
Ten	0:00	50:02	1	45:54	1	Reported

5. FINDINGS AND CONCLUSION

5.1 Findings

The findings of the hacking experiment shed light on the specific mechanism through which the exploit operates, taking advantage of a vulnerability inherent in the game's engine rather than targeting the anti-cheat system. This distinction is crucial as it highlights the need for game developers to focus not only on fortifying their anti-cheat measures but also on addressing potential vulnerabilities within the engine itself.

The vulnerability exploited in this experiment revolves around the unchecked nature of the r_draw function by the server, allowing it to run on the client side without proper validation. This lack of server-side scrutiny is primarily driven by the need to accommodate different users' graphic settings, which vary based on their individual processing power. By allowing the client to execute the r_draw function, the server can adapt the graphical output according to the user's selected settings, whether it be high, medium, or low.

However, this unchecked execution of the r_draw function on the client side poses a significant security risk. It enables malicious actors to manipulate and exploit the game's engine, potentially granting them unfair advantages or compromising the integrity of the gameplay experience for others.

To address this vulnerability, it is crucial for game developers to implement stricter controls over function calls like r_draw. One potential solution involves associating specific variables with each graphical setting, such as 0 for low, 1 for medium, and 2 for high. These variables can then be securely transmitted to the server, allowing it to draw graphics based on the user's preference while maintaining control over the execution of sensitive commands.

The exploit's focus on the game engine rather than the anti-cheat system highlights the importance of a multi-layered security approach in game development. While anti-cheat systems play a crucial role in detecting and preventing cheating, vulnerabilities within the game engine itself can be equally detrimental. This emphasizes the need for developers to thoroughly assess and fortify the engine against potential exploits.

The unchecked nature of the r_draw function by the server exposes a fundamental flaw in the game's architecture. Allowing the client to execute this function without proper validation opens the door for unauthorized manipulation and compromises the integrity of the gameplay experience. This vulnerability stems from the inherent trade-off between accommodating users' varying graphic settings and maintaining control over sensitive commands.

5.2 Summary and Recommendations

The gaming industry has been proactively seeking solutions to the issue of game hacking. Despite continuous efforts, these hackers have continued to adapt and evade the implemented measures. Combating game hacking requires a multi-tiered approach. Some potential solutions include:

5.2.1 Strengthening Legal Frameworks

Enforcing stricter laws against hacking, piracy, and RMT and increasing international collaborations to prosecute hackers can serve as a deterrent (Lu, 2018).

5.2.2 Building Community Awareness

Educating the gaming community about the consequences of game hacking could potentially discourage individuals from participating in these activities.

5.2.3 Technology Innovations

Developing and implementing new technologies, such as advanced anti-cheat software and encryption methods, can help developers to detect and block hacking attempts (Bohannon, 2010).

5.2.4 Industry Collaboration

Aligning the interests of developers, platform providers, and community leaders through shared objectives and coordinated actions to counter hacking initiatives can have greater impact and success in combating game hacking.

5.3 Conclusion

Injectors pose a significant challenge to the gaming industry's integrity, undermining the authentic and fair experience that online gaming intends to provide to its global audience. The use of injectors enables hackers to introduce foreign code into a game's memory space, allowing them to manipulate and exploit the game software for their benefit. The gaming industry continuously adapts its defense mechanisms to tackle the increasingly sophisticated hacking techniques presented by injectors and other hacking tools.

In the context of the context of physics engines used to power games such as Counter-Strike: Source, or Valorant, it is important to consider while developing these engines how they could possibility exploited and what their vulnerabilities are in order to prevent any issues at the source such as instances of r_draw() functions that are unchecked by the server, or leaving the engine open to injection by not monitoring what is making use of the game's or application's memory.

While anti-cheat systems, such as Riot Games' Vanguard, demonstrate promising advances in thwarting injectors and other hacking attempts, there remains a relentless arms race between hackers and game developers. If the desire to cheat persists, so too will the efforts to develop more advanced hacking tools.

To address this issue effectively, game developers must adopt a defense-in-depth strategy. This involves implementing a combination of server-side validation, client-server communication protocols, and secure variable storage mechanisms. By validating critical function calls on the server side, developers can ensure that only authorized actions are executed, minimizing the risk of exploits.

Introducing robust client-server communication protocols is crucial to prevent unauthorized manipulation of sensitive commands. By implementing secure channels for transmitting user preferences and graphic settings, developers can maintain control over the execution of functions like r_draw while still accommodating individual user requirements. This approach

ensures that the server retains the final say in determining the graphical output, preventing potential exploits.

Furthermore, storing variables associated with graphical settings in a secure manner can enhance the overall security of the game engine. By assigning specific values to each setting and transmitting them to the server, developers can ensure that the server interprets and executes the appropriate commands based on the user's preference. This approach not only enhances security but also provides a standardized and controlled environment for gameplay.

It is important to note that addressing vulnerabilities in a game engine requires a comprehensive understanding of software security principles, threat modeling, and rigorous testing. Game developers must prioritize security throughout the development lifecycle, employing techniques such as code reviews, penetration testing, and continuous monitoring to identify and mitigate potential vulnerabilities.

To further expand on the topic of using low, medium, and high variables to address vulnerabilities in the game engine, let's explore their implementation and potential benefits.

By associating specific variables with each graphical setting, such as 0 for low, 1 for medium, and 2 for high, game developers can establish a standardized framework for interpreting and executing graphical commands. This approach allows for more controlled and secure gameplay while accommodating the varying hardware capabilities of different users.

When a player selects a particular graphical setting, the corresponding variable is securely transmitted to the server. The server then uses this information to determine the level of graphical detail to render, ensuring consistency across all players while maintaining control over the execution of sensitive commands.

Implementing these variables offers several advantages. Firstly, it allows the server to validate the received variable against a predefined range, ensuring that only legitimate values are accepted. This prevents potential exploits that may attempt to manipulate the graphical settings to gain an unfair advantage.

Secondly, using variables enables developers to establish a hierarchy of graphical settings. For instance, the server can prioritize high-quality graphics (variable 2) over medium-quality (variable 1) or low-quality (variable 0) if the hardware capabilities of the player's system allow for it. This ensures that players with more powerful hardware can enjoy enhanced visuals without compromising the integrity of the gameplay experience for others.

Additionally, the use of variables facilitates easier maintenance and updates. If developers decide to introduce new graphical settings or optimize existing ones, they can simply modify the corresponding variables on the server side. This eliminates the need for individual client updates, streamlining the process and ensuring a consistent experience for all players.

However, it is important to implement proper security measures when transmitting these variables from the client to the server. Encryption and secure communication protocols should be employed to prevent unauthorized access or tampering of the data during transmission.

6. REFERENCES

[1] Arefin, S., Islam, S. S., & Chy, A. M. (2018). Identification of DLL injection technique based malware from process monitoring data. Journal of King Saud University-Computer and Information Sciences, 30(4), 465-474.

[2] Chen, H. T., Chen, S. Y., & Tu, C. H. (2009). Why do people immerse in the subculture of online game hacking: Contextual factors and their interactions. Social Behavior and Personality: An International Journal, 37(10), 1269-1280.

[3] Chen, H. T., Tu, C. H., & Wang, S. Y. (2008). Discovering the Motivation behind the Hacker's Mind: An Integrative Structural Model of Hacking Behaviors. Social Behavior and Personality: An International Journal, 36(2), 237-244Tavel, P. 2007 Modeling and Simulation Design. AK Peters Ltd.

[4] Hamari, J., & Sjöblom, M. (2017). What is eSports and why do people watch it? Internet research, 27(2), 211-232.

[5] Halfond, W. G., Viegas, J., & Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. Proceedings from IEEE International Symposium on Secure Software Engineering, Arlington, VA.

[6] Hutchins, B., & Rowe, D. (2018). eSports, skins betting and wire fraud vulnerability. Communication Research and Practice, 4(2), 145-162.

[7] Klumbyte, N., & Kerner, D. N. (2019). Digital cheating: the case of esports. European Journal for Sport and Society, 16(1), 85-104.

[8] Krebs, B. (2014). The Target breach, by the numbers. KrebsOnSecurity. Retrieved from https://nam06.safelinks.protection.outlook.com/?url=https%3A%2F%2Fkrebsonsecurity.com%2F2014%2F02%2Fthe-target-breach-by-the-numbers%2F&data=05%7C01%7Cfrancis.martinson%40microsoft.com%7C450cfc6861cb40b0fec008db534b447a%7C72f988bf86f141af91ab2d7cd011db47%7C1%7C0%7C638195354962725148%7CUnknown%7CTWFpbGZsb3d8eyJWIjoiMC4wLjAwMDAiLCJQIjoiV2luMzIiLCJBTiI6Ik1haWwiLCJXVCI6Mn0%3D%7C3000%7C%7C%7C&sdata=pARaQygDrg6OylTp%2B%2Fow0qCbmf0%2FcibjQYxX2cOIel0%3D&reserved=0

[9] Kushner, D. (2013). The real story of Stuxnet. IEEE Spectrum, 1-7.

[10] Li, Y., Yan, G., Xia, Z., Chen, B., Li, H., & Wu, D. D. (2018). Detecting Cheat Activities in Online Game: A Data Mining Approach. Decision Analysis, 15(3), 160-175.

[11] Mansfield-Devine, S. (2009). A short history of hacking. Network Security, 2009(2), 4-6.

[12] Sadeghi, A. R. (2018). The internet of insecure things: End-to-end security and privacy challenges. Proceedings of the IEEE, 106(9), 1706-1710.

[13] Richards, P. (2018). Tackling cheaters and pirates: Strategies for combating digital piracy in the video game industry. Interactive Entertainment Law Review, 1(1), 40-54.

[14] https://developer.valvesoftware.com/wiki/Visibility_optimization

[15] https://developer.valvesoftware.com/wiki/Func_occluder

[16] https://developer.valvesoftware.com/wiki/Source

[17] Ikariiillustration/injector (github.com).