ECE-455: Final Project

Multiplayer Video Game Hacking

Prof. Gitzel
ECE 455 Fall 2021
Victor Zhang
Shine Li

Table of Contents

Abstract

Multiplayer video games have become a popular pastime in the 21st century. Many genres of games are competitive. As such, for nearly as long as video games have existed, there have been bad actors attempting to gain an unfair advantage over other players, or otherwise disrupt (“grief”) other players. The widespread adoption of personal computing and wireless networking have given these bad actors the opportunity to execute “hacks” by modifying the behavior of the game as it executes on their local computer. This paper details our efforts to reproduce these hacks in the game “Counter-Strike: Global Offensive”(CSGO) and primarily explores the challengers the attackers (hackers) face. A brief overview of the strategies employed by the defenders (game developers) will also be given.

Problem Statement

In order to ensure a fair playing field, video games are designed with explicit and implicit constraints on player behavior. To illustrate this, consider the implementation of CSGO. CSGO is a first-person shooter (FPS) in which there are typically two teams of players - the “Terrorists” and the “Counter-Terrorists”. Winning in CSGO typically involves eliminating the entire enemy team by killing them in-game. In FPS fashion, this is done by having each player controll a player character with access to an arsenal of weaponry which they can use to shoot enemy players, reducing the enemy player’s health upon a successful hit. Explicit constraints can be found in the player character implementations: each player has a finite integer amount of health (typically 100), and each players’ weapons deal a predefined amount of damage to enemy players. Implicit constraints are expectations of normal player behavior. These can be deduced by the design of the game itself: terrain obscures player line of sight, so human players should not be aware of an enemy player’s exact position if the latter is obscured by terrain. The controls of the game function as another implicit constraint; FPS games are generally designed such that the shots players fire travel towards their point of aim. This point of aim can be controlled via the mouse, and is represented on screen with a pair of crosshairs. Thus, the implicit constraint here is that humans are expected to control the mouse with imperfect acuity, which translates to imperfect accuracy in-game.

An attacker may attempt to give themself an unfair advantage by violating these constraints. To violate an explicit constraint, an attacker may manipulate the underlying code that governs the game’s behavior, or modify the data (stored in memory) the code operates on. Violations of implicit constraints can be achieved by reading data. After reading the game data, an attacker can direct their computer to display the data to them, giving them access to information normal human players would not be privy to. Alternatively, an attacker can direct their computer to generate a series of inputs based on the data read - generally, these inputs mimic standard player interaction with the game, but are performed at superhuman levels (e.g. having perfect aim in an FPS).

The problems attackers face are how to gain access to and modify this data in a manner advantageous to themselves. The problems defenders face is how to limit the attacker’s ability to do this, either through prevention or detection. As with most problems in Cybersecurity, there is a “cat-and-mouse” element to this - as attackers mount more sophisticated attacks, defenders create more sophisticated defenses, and vice versa.

Prior Work

Unsurprisingly, there is a large body of existing work in this subject. Hackers have attempted and succeeded in gaining unfair advantages in almost every popular game. Conversely, there has been a long line of well documented "anti-cheat" software aimed at combating these hacks. However, while many hacks are similar on a high-level, their underlying implementations differ considerably. This is because each hack needs to be tailored to a specific version of a specific game running on a specific engine on a specific operating system. Thus the vast majority of hacks in existence were developed with a specific target in mind. Changes to operating systems, game code, network APIs etc. render most hacks obsolete within weeks-months (or possibly even within days or hours, if the hack is widely publicized and deemed to be a critical threat to legitimate players' experiences, and there is a competent development team behind the video game). Despite this, there is much to be gleaned from studying old hacks. The hacks we developed for this report followed the methodologies used by others to reverse engineer CSGO. Where possible, we borrowed implementations for functions that, while not directly related to the hacks, aided in their functionality (e.g. Windows thread handling). Finally, there exist a number of widely used utilities which expedite parts of the reverse engineering process. These included Cheat Engine (memory scanning & modification tool) and Reclass.NET (a tool which allows users to reverse engineer structs and classes). These tools were vital to this project's success, and are due credit.

Security Analysis/Threat model

CSGO is a 32-bit (x86) program that can run on the Windows operating system. It operates under a client-server model, where the server is given authoritative control over a number of variables (such as an entity list, each entity's health int., money int., etc.). The server typically runs on a dedicated machine, hosted by either a 3rd party entity, or Valve - the developers of CSGO. Naturally, the server's role ability to arbitrate games of CSGO gives it the unparalleled capability to unconditionally alter a wide variety of parameters in a CSGO game, such as the match timer, map, and the aforementioned entity variables (health, ammo, weapons). While it is possible for a server to be compromised or controlled by a bad actor, our analysis will preclude this possibility. We will focus entirely on what the attacker can do from their client side, which we assume to be an unremarkable computer remotely connected to the server.

In order for the client to run the game, their local computer must have access to a minimum amount of data. For instance, their computer must store an entity list containing the positions of each player entity in the game - otherwise, their client would not be able to render these entities in game. Thus a copy of relevant data is stored on each client's machine. But it is not enough to merely store data on each machine while giving the server the sole right to alter this data. In order to ensure that the game is responsive for all clients, the game permits all clients to have control over some parameters, including the means to alter them. For example, the data governing a client's local player must be instantly responsive to input from the player. Mere 10s of milliseconds of delay between input and response would be perceptible to humans as "lag", and would degrade the player experience. If the client needs the server's permission to move its local camera angle after a player moves their mouse, this operation can easily take 100-200+ ms to complete, depending on network latency. Hence the client is given complete control over a number of parameters (typically governing player movement and aim).

A number of methods are implemented with a compromise between client responsiveness and server authority. For instance, when a player shoots an enemy, their local computer will display the animation of the enemy being hurt, while packets describing the damage dealt are sent to the server for validation. If the server determines that the hit was valid, it sends back a response allowing the client to commit changes to the enemy's health in their own computer. CSGO in particular does this with many important variables such as entity health - one cannot simply change a player's health on their own instance of the game and have that change replicate to every client connected to the server. CSGO goes even further by including in every entity a "bDormant" boolean (typically, though implementations of bDormant type can differ). bDormant is set to true if the entity is too far from the local player for them to interact. In this dormant state, the changes to the entity's member variables (such as health) are not replicated to the player client. Only when the player gets close enough to the enemy entity does bDomant get reset, allowing the client to see up-to-date information on the other entity.

CSGO does not implement any obfuscation - in fact, the source code of the game has been leaked in the past. Neither does CSGO take any measures to prevent debuggers and other reverse engineering programs from dissecting its code. This makes it an ideal target for an introductory attack - this is what motivated our choice to hack CSGO. In general video games may implement obfuscation and debugger detection to make analysis and reverse engineering of the program more difficult.

Our attack mainly focuses on testing the integrity aspect of the game and its anti-cheat system. The anti-cheat system “Valve Anti Cheat” is supposedly to scan all files, running processes, registry, hooks and open event handles. To penetrate the anti-cheat, Manul Mapping mode is used for injection, and VMT hooks are replaced with trampoline hooks. Our attack DLL can be successfully loaded into the game using the previously mentioned method, and the game cheat is tested to work as intended, providing aimbot and other functions.

Description and Technical Analysis of Attack

Before any code is written, the attacker must understand how the game works. However, in most cases the attacker will not have access to the source code of the game (CSGO is an exception). Instead, the developers of a game will distribute the binaries directly to the clients via installer executables. The attacker will need to deduce how the game works using **reverse engineering**. In this context, reverse engineering refers to the use of debuggers and other utilities to examine memory as the game runs, before using this knowledge in combination with experimentation and experience to deduce the structs and classes in the game.

A common way to begin is by attaching Cheat Engine to the csgo.exe process. Cheat Engine is a tool that allows attackers to scan the process memory for values matching a specified value. For instance, if an attacker wants to know the memory address at which an entity object's health int is stored, they could scan for the value '100' (assuming the entity's health is currently 100). Subsequent Cheat Engine scans after the first only examine memory addresses that were found in the previous scan, allowing attackers to narrow down the possible memory addresses in the following manner: scan for value -> change value in game (take damage to reduce health) -> scan again -> change again.

What will remain after multiple iterations will be the health address. There are some complications that can arise - for instance, what if health is stored as a float; what if health is unknown; what if there are multiple addresses that contain a value equal to health. There are techniques an attacker can use if these situations arise, but in the interest of brevity they will not be covered here. We proceed assuming a health address for an entity has been found. Cheat Engine has a built-in debugger that can detect and flag any instructions that access a particular address. Using this functionality, an attacker may find that when they take damage in game, the instruction "mov [eax + 0x8], ebx" accesses the health address (where eax+0x8 is the address of the health integer found earlier). In that case, coding experience would suggest that the address of eax is possibly the address of the player entity, and that the health int is offset by 8 bytes inside the entity class:

```
class entity {
        int intOne;
        int intTwo;
        int health;
        //…
};
```

This hypothesis may or may not be true, but through continuous trial and error experimentation the attacker may eventually deduce the class structure. Next, the attacker is interested in finding a static address to this address. Game memory is allocated dynamically, so there is no guarantee that the memory addresses found in one reverse engineering session will hold true the next time the game is launched. However, by working backward the player may eventually find a static address and a series of offsets that lead to the desired variable. For instance, the health may be offset 0x8 from the player's entity, which may itself be pointed to by

a pointer stored inside an entityList object (offset by 0x0), whose address is in turn stored by a static pointer. Schematically, it might look like this:
Address = Value = ?

base ptr -> address + offset4 = address

base ptr -> address + offset3 = address

base ptr -> address + offset2 = address

static base -> address + offset1 = address

Reclass.NET is another tool that can be used in the reverse engineering process. Like Cheat Engine, it examines a processes memory and users attempt to deduce the structs from the memory view. To aid in this process, it displays each 4-byte chunk of memory's integer, float, and string (character) representation side-by-side. It also allows users to assign types to certain memory addresses, and will automatically adjust the display accordingly. Reclass.NET can also export a reverse engineered struct as c++ code, allowing it to be used in internal hacks.

Once all relevant data structures & classes have been reverse engineered, the actual coding of the hack can begin. Hacks fall into 2 broad categories: **external** and **internal**. The difference is that external hacks use the Windows API and admin privileges to read and write to memory, while internal hacks take the form of Dynamically-Linked Libraries (dll)s which are "injected" into a running process. In either case, the attacker is attempting to read some memory (based on the reverse engineering work done previously), before perhaps writing to memory or specifying an input.

For instance, the "aimbot" hack tries to automatically aim at enemy entities. On each frame, the hack will: read the nearest enemy entity's x and y coordinates (stored as floats inside the enemy's entity); read the player's current position; determine the vector from the player's camera to the enemy's head using trigonometry; write this vector to the player's view angle such that the player is now aiming at the enemy's head. Note that since the player's view angle falls within the client's control (as opposed to being server controlled, like entity health), the hack may write to it with impunity. The vast majority of hacks follow this 1. read memory 2. do something pattern; the challenge is in the reverse engineering to determine the correct memory addresses to read and write to.

Methodology and Setup

The game cheat is implemented via dll injection written in C++, using the GuidedHacking injector in Manual Mapping mode. The manual mapping mode is used to prevent VAC detecting loaded processes and The memory address used for the attack is dumped using the hazedumper tool, which provides all necessary addresses for reading the state of entities (players). A custom entity class is constructed with the correct bytes of padding, and the module reads the consecutive memory address for info on all players. The players' info is then processed by simple linear algebra to guide the aimbot.

The graphic part of the cheat is done using the old d3dx9 library through a trampoline hook, which prevents VAC detection of vtable hooking.The gameplay enhancing graphics are implemented by grabbing entity info including location and direction, then calculating the desired drawing.

Conclusion and Future Work

We predict that game hacking will continue to be a popular sub-genre of cybersecurity for as long as video games are popular. Although the methods explored in this report are rudimentary as far as game hacking goes, they cover the fundamental concepts behind game hacking. There is another critical component to game hacking which was purposefully excluded from the scope of this project: countering anti-cheat. Developers are vigilant against hackers, and will attempt to make their games more difficult to hack through a variety of means. These include:

- Structuring their programs such that very little information is available to, let alone modifiable by, a client. For example, League of Legends implements a “fog of war”, where if a client is not supposed to know about the position of an enemy, the server will not even send this information to the client
- Adding anti reverse engineering features. This includes code obfuscation and debugger detection.
- Detecting hacks and removing hackers. If a hack is detected, the hacker may be swiftly banned by the server.
- “Kernel level” anticheats, which disable insecure drivers commonly used by hackers to read and write to game memory.

Future work can (and is) being done to counter these countermeasures.