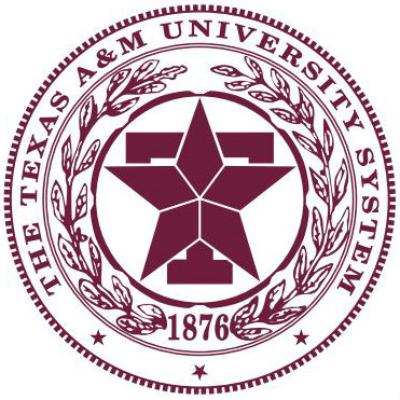
**CSCE 465 - Computer & Network Security**

**Anti-Cheat System Vulnerability Analysis**

**Final Report**

****

***This paper is equally contributed by the following authors:***

|  |  |
| --- | --- |
| **Author** | **UIN** |
| **Han Hong** |  |
| **Jiayao Li** | **423002027** |

*On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work.*

**May 6th, 2018**

**Table of Contents**

[**I. Abstract**](#_s9zw5dfjjkxe) **2**

[**II. Introduction**](#_bbhlv4yx7vyk) **2**

[**III. Motivation**](#_uh0jssraszx9) **2**

[**IV. Work Comparison**](#_663233a5r7yz) **3**

[**V. Challenges**](#_eyzby8lkk84t) **3**

[A. Anti-Cheat System Detection](#_2jgip7drphxr) 3

[B. Human Detection](#_2xqa8u9fi7th) 4

[**VI. Methodologies**](#_3a60swg4sz0n) **5**

[A. Internal Method](#_oyuuh0a9slta) 5

[B. External Method](#_w1nqsdmldywy) 5

[**VII. Results & Discussion**](#_krv6401v3rvr) **6**

[A. Concept and Design](#_43iq7uqbmg9i) 6

[B. Implementations](#_eaq5rhil9kha) 8

[C. Proposed Detect Method](#_vbx6eex33kbk) 9

[**VIII. Scrum Iterations & Project Checkpoints**](#_45tvwwplbvcv) **10**

[**IX. Conclusion**](#_pirdqodapibu) **10**

[**X. Point of Contacts**](#_ehy6jx92z5zg) **10**

[**XI. References**](#_2mepjmk34c5z) **10**

# **I. Abstract**

The increasing stake in competitive gaming has lead to ambot cheating. The current defense system is based on scanning a database of known attack method and players recognizing the bot’s inhuman behavior. The objective of this project is to demonstrate the vulnerabilities of current bot detection systems and the user report based detection system. This project successfully demonstrated an attack method by implementing algorithms to detect targets using pixel’s color, with an additional component to humanize the bot’s behavior. Ultimately, the goal is to have a highly efficient aimbot, that is technically undetectable. By developing an attack with this known concept and algorithm, the project aims to demonstrate the deficiencies underlying the current systems. The project’s result can be used to address the shortcomings, and propose future fixes in similar applications.

# **II. Introduction**

A bot, short for an internet bot, as known as web robot, is a software that runs automated tasks over the internet. Generally, bots often perform tasks that are simple and repetitive with a much higher performance rate compared to human. An aimbot, also known as auto-aim, is a type of computer game bot used in multiplayer first-person shooter games to generate different levels of automated target detection to the player. Aimbot relies on receiving information about other players from one player’s client account and then target the opponents based on their relative location to the player. The objectives of this report is to show the vulnerabilities of current bot detection by developing an attack method that can overcome the current defense system. This report also includes an impact analysis of current offense and defense methods.

# **III. Motivation**

With the rise in popularity and normalcy of video games, the competitive games industry has grown immensely in the past few years. This means that there’s now more at stake than just user pride when it comes to using malicious code to exploit or modify the game to give on particular player an edge over another. Our goal is to demonstrate the vulnerabilities of current bot detection systems by modifying and improving current attacks in order to bypass the current detection system.

# **IV. Work Comparison**

**A. Current Offense**

While some aimbots can be sophisticated, the standard and basic aimbot is a program that automatically locks onto the hitbox of a targets head and stays there. This is obvious to detect by observation, just by watching how jerky or ‘inhuman’ some of the movements are, or how a player will follow a target moving, with their reticle still locked perfectly on the head.

**B. Current Defense**

The current most universal system of bot detection is the honor system. A user can be suspected of bot usage only through other player’s reports. If a user get a certain amount of reports in a time window, the footage of the game will be reviewed manually and judged solely through human observation.

There are more sophisticated detection systems, and most commonly use a known cheat database. The detection software scans the running processes and game memories looking for hashes and signature that is already detected and blacklisted in the past. This requires authentication of the client’s system which can be easily bypassed, and is currently not effective in many competitive games.

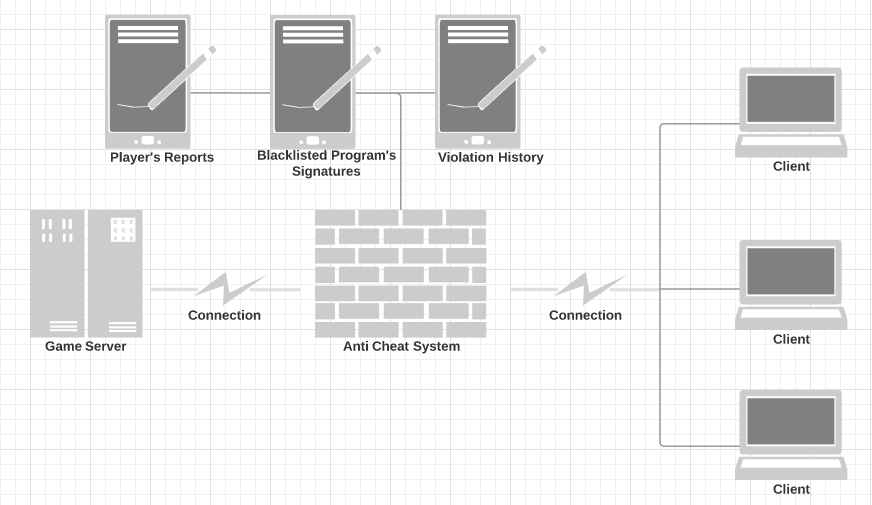
**C. New Attack Method**

Our attack method beats the current detection system by making it less obvious during observation. The attack method modifies the targeting algorithm so that it doesn’t always aim straight for the head hitbox. Instead, a larger hitbox is created and the point where the bot would hit is randomized. Furthermore, the movements are smoothed out by reducing the reaction time, making it look like there is ‘human error’ in targeting another player. The hit box is also only activated when the offending player has ‘line of sight’ on another player. This keeps the bot from tracking other players through walls or solid objects, making it less obvious, to observation, that the player is using an aimbot.

# **V. Challenges**

## **A. Anti-Cheat System Detection**

The Valve Anti-Cheat System (VAC) proposes the first major challenge when it comes to attacking. VAC is a software that detects cheating in a gaming environment. It is developed by the Valve Corporation. If a user connects to a VAC-secured server and gets caught cheating by the software. The VAC system will ban the user’s steam account permanently from playing multiplayer games on VAC-secured servers [1]. The VAC works like a virus scanner. It has a database of known cheats to detect and scans the user’s computer system while the game is running [2]. The VAC ban is costly, once a ban is effective, the only way for the user to get around the ban is by creating a new account and playing another copy of the game [2].



**Figure 1.** Anti-Cheat System Schematic

## **B. Human Detection**

Another major challenge for developing attacking methods is to be discrete and avoid getting caught by other players in the game. Otherwise, a programmed ambot can easily perform tasks at a rate human beings cannot achieve. The bot could be spotted and get reported by other uses if the program does not perform tasks in a discrete manner. This leads to the challenger of designing the attacking method to be less obvious, avoid performing the same tasks over and over again at a high rate, and act more human in a game.

# **VI. Methodologies**

## **A. Internal Method**

It is achieved through modification of the game memory. Internal method requires direct access to the game and knowledge of the game’s data packages. It is similar concept to a buffer overflow in that the attacker have to accurately guess the memory address, calculate the offset and overwrite with a correct package. The end goal here is to repack the manipulated data and inject into the process.

**Table 1.** Overview of Pros and Cons for Internal Method

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| * Higher performance * More features | * Unknown detection * Bug and crashes * Vary from game to game |

## **B. External Method**

Opposite to internal method, external hack runs independently from the game and requires no modification or access to the game memory. It utilizes the current environment, taking current system’s informations such as pixels, mouse input, etc.

**Table 2.** Overview of Pros and Cons for External Method

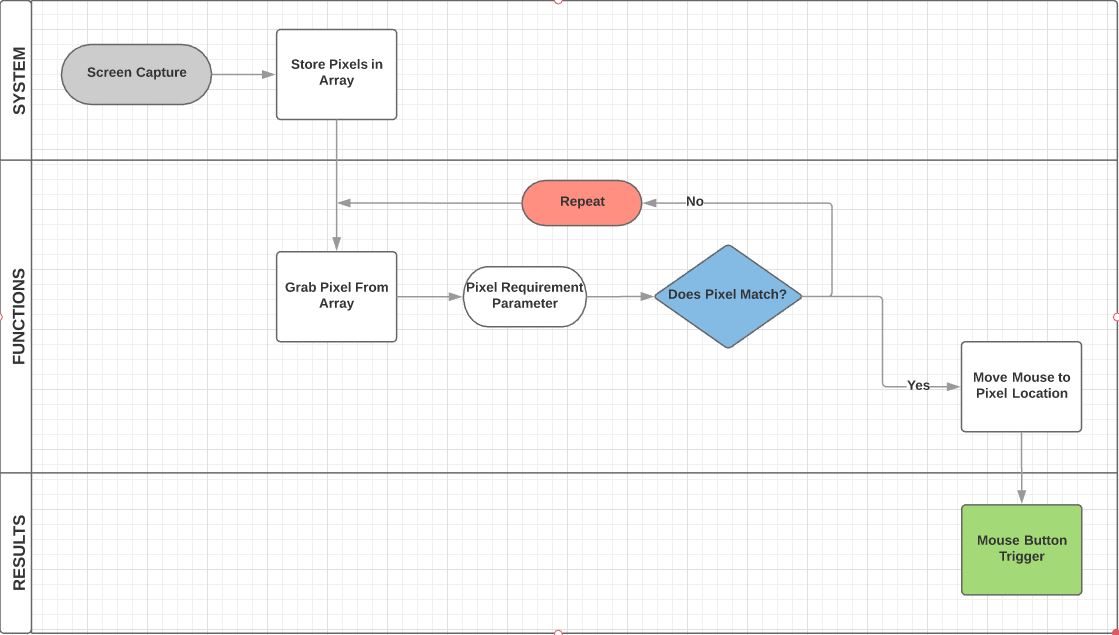
|  |  |
| --- | --- |
| **Pros** | **Cons** |
| * Less detectable * Wider ranges of games | * Performance issue * Less features * Less accurate |

# **VII. Results & Discussion**

## **A. Concept and Design**

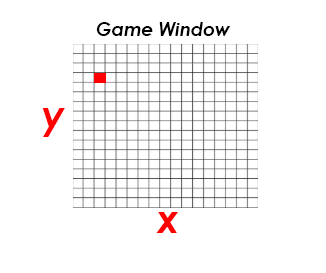
Figure 2 shows the design flow chart, with the following steps:

* Call to the system to capture game window screen
* Store screen into an array of pixels
* Check each pixel if pixel is matched with a predefined value (color of the target)
* If matched, current mouse position is updated to move to that pixel location
* Depend on current setting, a mouse trigger can be called to automatically shoot.



**Figure 2.** Design Flow Chart

Figure 3 shows a demonstration of the system detecting the pixels, from left to right and from top to bottom. On the left, it shows that the system detects a “head” using a pixel’s color information. On the right, it shows that the pixels can travel back and forth in a selected range. The system reads the color of the pixel in the selected range and detects targets in the game based on its pixel colors. By selecting a middle range to scan, the system can successfully avoid Head Up Display, which is the information box at the edges of the screen.



**Figure 3.** (a) Pixel Scanning (b) Scanning the Middle Range

Figure 4 shows a demonstration of the system detecting all the pixels with matched criteria. In this case, it marks out pixels that is below the RGB 81-74-32 color, for testing purposes. In an actual attack, the attacker can pick the range of criteria based on the target’s information.



**Figure 4.** All pixels with matched criteria is marked

## **B. Implementations**

The algorithm below shows the “Screenshot Function”, which stores screenshot in an object and then generates a bitmap and pixels.

|  |
| --- |
| bool ScreenShot(string current, BITMAP &bm, HBITMAP &hbmap, BITMAPINFO &bmi, HDC &hdcShot, HBITMAP &hbitmapOld, HWND &window) {  RECT rc;  GetWindowRect(window, &rc);   hdcShot = CreateCompatibleDC(0);  hbmap = CreateCompatibleBitmap(GetDC(0), rc.right - rc.left, rc.bottom - rc.top);  SelectObject(hdcShot, hbmap);   BitBlt(hdcShot, 0, 0, rc.right - rc.left, rc.bottom - rc.top, GetDC(0), rc.left, rc.top, SRCCOPY);   if (!GetObject(hbmap, sizeof(BITMAP), (LPSTR)&bm))  return false;   int bitsPerPixel = bm.bmBitsPixel;   if (bitsPerPixel != 32 || bm.bmPlanes != 1)  return false;   SetupBitmapInfo(bmi, bm.bmWidth, bm.bmHeight, bitsPerPixel);  return true; |

**Figure 5.** The Screenshot Function

Figure 6 belows shows the pixel scanning captured screenshots. It iterates over capture screen, from left to right, top to bottom. In this case, for performance, screen is set to capture only mid section (middle ¼ of the screen). At each pixel, it checks if the pixel matched the predefined values (color of target), and if matched, mouse button is triggered to shoot. And at the last step, it considers humanized factor by taking a detour to slow the speed down instead of using a computer speed.

|  |
| --- |
| void ScanBMP(ScreenCapture \*scan) {  for (int y = (scan->screenWindows.bottom - scan->screenWindows.top) / 4;  y < ((scan->screenWindows.bottom - scan->screenWindows.top) - (scan->screenWindows.bottom - scan->screenWindows.top)/3.5);  y++)  {  for (int x = (scan->screenWindows.right - scan->screenWindows.left) / 4;  x < ((scan->screenWindows.bottom - scan->screenWindows.top) - (scan->screenWindows.bottom - scan->screenWindows.top) / 3.5);  x++)  {   if (CompareColor(scan->pixels, scan->screen.bmHeight, scan->screen.bmWidth, x, y)) {  int z = x;   //More accurate aim loop  while (z < (int)((scan->screenWindows.right - scan->screenWindows.left) - (scan->screenWindows.right - scan->screenWindows.left) / 4)) {  if (!CompareColor(scan->pixels, scan->screen.bmHeight, scan->screen.bmWidth, z, y))  break;  z++;  }   SetCursorPos((z - (z - x) / 2) + scan->screenWindows.left, (y + 4) + scan->screenWindows.top);  POINT currentPos;  GetCursorPos(&currentPos);   //Restrict shooting while moving to pixel location  if (currentPos.x < currentMouse.X + 4 && currentPos.x > currentMouse.X - 4  && currentPos.y < currentMouse.Y + 4  && currentPos.y > currentMouse.Y - 4)   {  Shoot(x + scan->screenWindows.left, scan->screenWindows.top);  }   currentMouse.X = currentPos.x;  currentMouse.Y = currentPos.y;  //system("pause");  return;  }  }  }  } |

**Figure 6.** Pixel Scanning Captured Screenshots

The developed program successfully bypass VAC without being detected. However, due to being pixel bound, false positive pixel matches can occur frequently. By minimizing pixel matching error, the program accuracy can greatly be improved. Performance wise, games tested in this project do not require an above average computing power to run. Therefore, with more demanding games, performance will be greatly hindered.

## **C. Proposed Detect Method**

This section describes the solution of the pixel attack.

# **VIII. Scrum Iterations & Project Checkpoints**

The scrum methodology is used for this project. Table 3 shows an overview of the scrum iterations and the project checkpoints for each phase during development.

**Table 3.** Overview of Project Checkpoints

|  |  |  |
| --- | --- | --- |
| **Scrum I: Analysis** | **Scrum II: Implementations** | **Scrum III: Deploy** |
| * + Research and information retrieval   + Data Analysis | * + Bot scripting   + Case studies | * Testing * Live Demo |

# **IX. Conclusion**

This approach is one of many types of hacks that are currently active in the gaming industry. This project successfully demonstrated a humized attack method bypassing the VAC anti-attack in a multi-player gaming environment. However, the point of this project is not about developing the best hack. With the background of increasing stake and the booming of the monetary value in competitive gaming, the current anti-cheat system is dangerously inefficient. Recognizing this shortcoming is the first of many steps to solve this issue.

# **X. Point of Contacts**

* Han Hong: [hongichhan@tamu.edu](mailto:hongichhan@tamu.edu)
* Jiayao Li: [jiayaoli@tamu.edu](mailto:jiayaoli@tamu.edu)
* GitHub repository: <https://github.com/hongsolos/CSCE-465-Honor-Project>

# **XI. References**

1. “Valve Anti-Cheat System (VAC)”. Steam Support. 2017. <https://support.steampowered.com/kb\_article.php?p\_faqid=370>
2. “VAC Integration”. Steam Works. 2018. <https://partner.steamgames.com/doc/features/anticheat/vac\_integration>