

Frans Aaron

Exploring a Code-Injected Aimbot in Doom

Supervisor: Tesch Tom

Coach: Defoort Stephanie

Graduation Work 2023-2024

Digital Arts and Entertainment

Howest.be

CONTENTS

Abstract & Key words	4
Preface	5
List of Figures	6
Introduction	7
Literature Study / Theoretical Framework	9
1. Code Injection vs Image Recognition	9
1.1. Code Injection	Error! Bookmark not defined.
1.2. Image Recognition	Error! Bookmark not defined.
Case study	13
1. Introduction	13
2. Creating the DLL Injector	13
3. Making the doom aimbot	14
3.1 Finding the memory offsets	14
3.2 Making the aimbot	15
3.3 Measuring the performance, and development time	16
4. Making the Image Recognition aimbot	17
4.1 Making the Image Recognition	17
4.2 Measuring the performance	17
5. Making the Call of Duty: world at war aimbot	18
5.1 Finding the memory offsets	18
5.2 Overriding the look function	18
5.3 Measuring the development time	18
Discussion	19
1. Difference in Performance	19
2. Difference In Development Time	19
Conclusion	20
Future work	21
1. Train AI Using Code Injection	21
2. Compare Code Injection to AI Trained for Image Recognition	21
3. Way to Prevent Code Injection	21
Critical Reflection	22
References	23
Acknowledgements	25

Appendices	26
------------------	----

ABSTRACT & KEY WORDS

This paper explores the process of developing a code-injected aimbot for the classic video game, Doom (*DOOM*, 2012/2023). The study provides an in-depth examination of the development process, detailing the benefits of both Code Injection and Image Recognition. It also clarifies the benefits of having access to the source code of the game. This research contributes valuable insights into the technical aspects of aimbot creation within the gaming landscape, offering an understanding of the coding intricacies involved in making an aimbot work. This research can be further used to prevent the future creation of aimbot for online games.

Dit artikel onderzoekt het ontwikkelingsproces van een aimbot met code-injectie voor de klassieke videogame Doom (*DOOM*, 2012/2023). Het onderzoek biedt een diepgaand onderzoek van het ontwikkelingsproces en beschrijft de voordelen van zowel code-injectie als beeldherkenning. Het verduidelijkt ook de voordelen van toegang tot de broncode van het spel. Dit onderzoek draagt bij aan waardevolle inzichten in de technische aspecten van het maken van aimbots binnen het gamelandschap en biedt inzicht in de ingewikkelde codering die nodig is om een aimbot te laten werken. Dit onderzoek kan verder worden gebruikt om het maken van aimbotten voor online games in de toekomst te voorkomen.

PREFACE

This research project stems from a deep fascination with AI in games and the possibilities around Code Injection in this field. The focus on developing a code-injected aimbot for Doom (*DOOM*, 2012/2023) was driven by a keen interest in exploring how Code Injection can unlock new potentials for AI in games lacking such capabilities.

Beyond the exploration of Code Injections, this endeavor holds a dual purpose. Firstly, I aspire for this research to contribute to advancements in anti-cheats within the gaming industry. Secondly, on a personal level, I hope that this undertaking will deepen my understanding of code behavior during compilation, offering insights into the way variables and function behave when compiled.

LIST OF FIGURES

Figure 1 Example of Machine Learning Bot in League of Legends (Lohokare et al., 2020)	7
Figure 2 The Doom Logo (1993) (Doom Logo, n.d.).....	7
Figure 3 An example of Image Templating in a game. (ClarityCoders, 2021).....	8
Figure 4 A screenshot of Process explorer (SALIANKHAM, n.d.)	9
Figure 5 A basic overview of a DLL injection (Wilson, n.d.)	11
Figure 6 An example of Image Templating (OpenCV: OpenCV Modules, n.d.)	12
Figure 7 OpenCV Logo (OpenCV Logo, n.d.)	12
Figure 8 Image of the code injector.....	13
Figure 9 Cheat engine logo ('Cheat Engine Wiki', 2023).....	14
Figure 10 Abbreviated layout of the player_s object (Godbolt, n.d.-c)	14
Figure 11 Abbreviated layout of the mobj_s object (Godbolt, n.d.).....	15
Figure 12 Code used for calculating the new aim direction	15
Figure 13 Measurements for the Code Injection aimbot	16
Figure 14 Another example of Image Templating (Rosebrock, 2021)	17
Figure 15 Measurements for the Image Recognition aimbot.....	17
Figure 16 Cover of Call of Duty: World at War (Call of Duty: World at War, 2023).....	18

INTRODUCTION



Figure 1 Example of Machine Learning Bot in League of Legends (Lohokare et al., 2020)

The combination of machine learning and gaming, exemplified by some bots in League of Legends (*League of Legends*, n.d.), has prompted a keen interest in exploration the realm of artificial intelligence. However, the intricate nature of Code Injection, adds a unique opportunity for this field of research.

My curiosity stems from observing machine learning-driven bots in League of Legends (*League of Legends*, n.d.), coupled with a keen interest in the potentials of Code Injection to enable AI to interpret game data and actively engage in gameplay. Notably, the availability of Doom's (*DOOM*, 2012/2023) free source code presents an opportunity for me to learn the complexities of injecting code into a game with straightforward mechanics, and a simple way of aiming.

This paper aims to dissect the basics of Code Injection, emphasizing the development process in the context of Doom (*DOOM*, 2012/2023). The focus is learning how Code Injection works and in future research use this gained knowledge to train AI to play a game.

The central research question guiding this investigation is: What does the development process entail for creating an aimbot in the game Doom (*DOOM*, 2012/2023) through the utilization of Code Injection?



Figure 2 The Doom Logo (1993) (Doom Logo, n.d.)

This study focuses on nuances of Code Injection in a simplified gaming scenario. There are two major factors that come into play when trying to do this. Firstly, the availability of the source code should have an impact on the development time due to identification of underlying code. Secondly, because Image Recognition is inherently slow, due to the way images are compared, Code Injection should offer a faster, more performant alternative.



Figure 3 An example of Image Templating in a game. (ClarityCoders, 2021)

This translates into the following hypotheses:

1. Using source code in the development cycle will result in a faster development cycle.
2. An aimbot using the method of Code Injection will be more performant than one using Image Recognition.

LITERATURE STUDY / THEORETICAL FRAMEWORK

1. CODE INJECTION VS IMAGE RECOGNITION

1.1. CODE INJECTION

1.1.1 WHAT IS CODE INJECTION

Code Injection is a broad term encompassing attack techniques that involve the insertion of code into an application, subsequently interpreted and executed by the application itself.

(Code Injection | OWASP Foundation, n.d.)

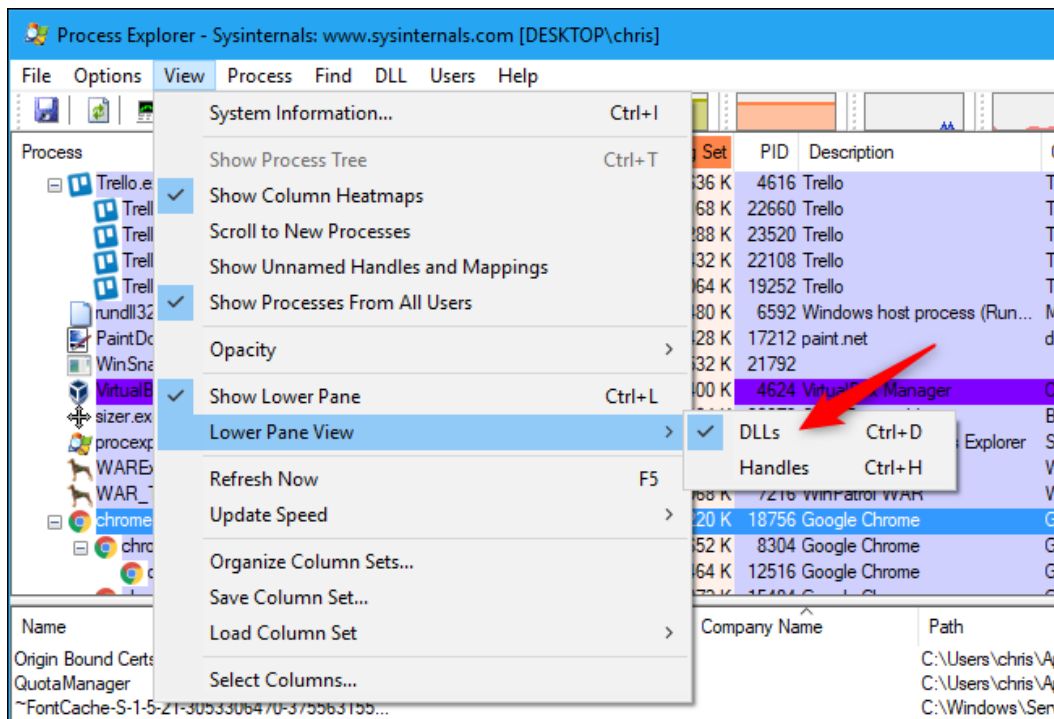


Figure 4 A screenshot of Process explorer (SALIANKHAM, n.d.)

Code Injection on Windows serves a wide array of purposes, ranging from legitimate applications to malicious activities. Examples include:

- Legitimate Use in Antivirus Programs:
 - o Antivirus programs often employ Code Injection into web browsers. This enables them to monitor network traffic, ensuring the identification and blocking of potentially harmful web content.
- Malicious Exploitation in Web Browsers:
 - o Malware can maliciously inject code into web browsers to track users' browsing activities, pilfer sensitive information like passwords and credit card numbers, and manipulate browser settings for unauthorized access.
- Enhancing Desktop Themes with Stardock's WindowBlinds:
 - o Stardock's WindowBlinds, a desktop theming tool, utilizes Code Injection to modify the rendering of windows, enhancing the visual appearance of the desktop.
- Desktop Organization with Stardock's Fences:
 - o Stardock's Fences employs Code Injection to alter the functionality of the Windows desktop, providing users with an organized and customizable desktop environment.
- Custom Scripting with AutoHotkey:
 - o AutoHotkey, a scripting tool allowing users to create scripts with system-wide hotkeys, uses Code Injection to implement custom functionalities seamlessly.
- Graphics-Related Tasks with NVIDIA Drivers:
 - o Graphics drivers, such as NVIDIA's, utilize DLL injection to achieve various graphics-related tasks, enhancing the performance and capabilities of graphics processing.
- Additional Menu Options in Applications:
 - o Some applications employ DLL injection to add extra menu options, extending the functionality of the application beyond its default features.
- Unfair Advantage in PC Game Cheating Tools:
 - o PC game cheating tools leverage Code Injection to modify the behavior of games, granting users an unfair advantage over opponents by altering game mechanics.

(SALIANKHAM, n.d.)

In essence, the versatility of Code Injection on the Windows platform allows for both constructive and malicious manipulations, showcasing its pervasive use across a spectrum of software applications and utilities.

While Code Injection is a broad subject pertaining to multiple facets of code, this paper concentrates its attention on DLL injection. The reason behind this focus lies in the intimate nature of interaction between the injected code and the original program.

1.1.2 WHAT IS DLL INJECTION

In computer programming, DLL injection is a technique used for running code within the address space of another process by forcing it to load a dynamic-link library (DLL). DLL injection is often used by external programs to influence the behavior of another program in a way its authors did not anticipate or intend. For example, the injected code could hook system function calls, or read the contents of password textboxes, which cannot be done the usual way. A program used to inject arbitrary code into arbitrary processes is called a DLL injector. ('DLL Injection', 2023)

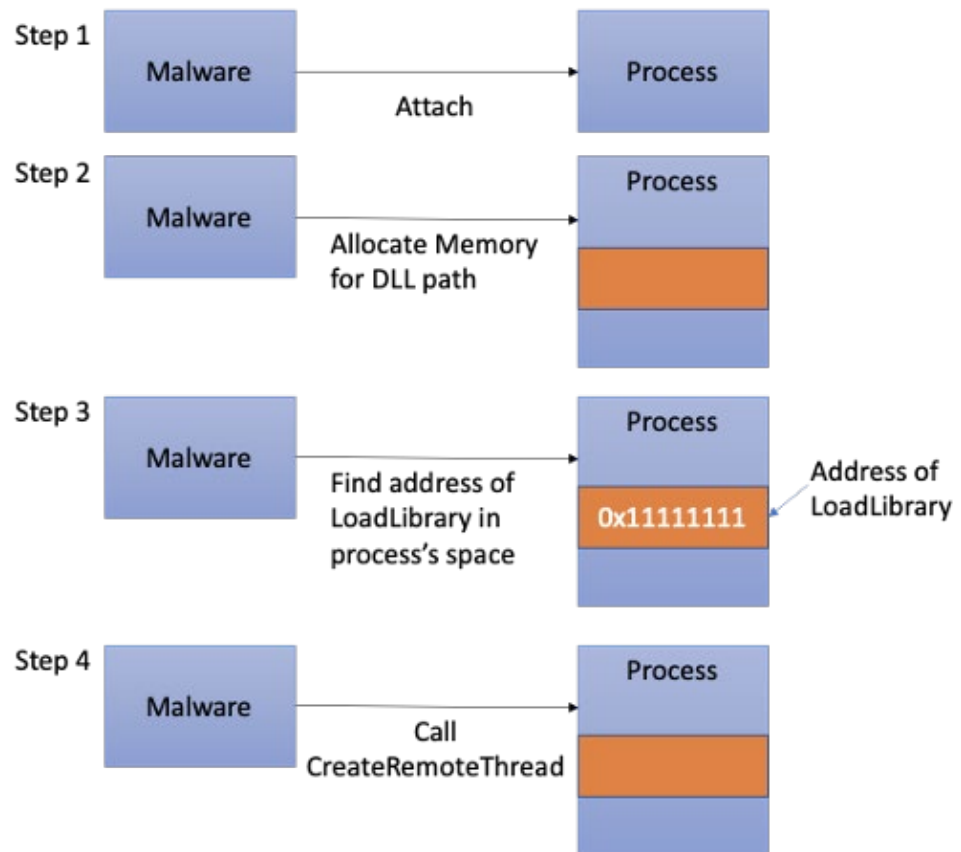


Figure 5 A basic overview of a DLL injection (Wilson, n.d.)

1.2. IMAGE RECOGNITION

1.2.1 WHAT IS IMAGE RECOGNITION

Image Recognition, as the name implies, is the practice of using a computer program to compare or recognize images. There are multiple ways of doing this from teaching an AI to recognize certain objects to comparing 2 images. In this paper a simple technique called Template Matching was used. This is a technique where you take an example image, and the computer tries to find that “template” in another bigger image.



Figure 6 An example of Image Templating (OpenCV: OpenCV Modules, n.d.)

The technique used is the one offered by OpenCV (see 1.2.2)

1.2.2 WHAT IS OPENCV

OpenCV is an open-source Library used for both computer vision and machine learning developed by the non-profit Open Source Vision Foundation. This library is used by big companies such as Google and Microsoft and contains over 2500 optimized algorithms for both computer vision and machine learning. (OpenCV About, n.d.)

This library was used because it has everything needed for the project out of the box, but also because it is a very popular library and as such there is a lot of online documentation and examples.



Figure 7 OpenCV Logo (OpenCV Logo, n.d.)

CASE STUDY

To answer the research question, there was opted to go for a case study. This is because it allowed for an in-depth exploration of Code Injection.

1. INTRODUCTION

The goal of this paper is to help people get a better understanding of the realm of Code Injection. As the usage of Code Injection for the purpose of creating an aimbot is usually seen as undesirable or illegal, it is a very interesting and challenging approach to apply this technique. This paper is meant to be a personal exploration into the topic. This is why the question for this research is: "What does the development process entail for creating an aimbot in the game Doom (*DOOM*, 2012/2023) through the utilization of Code Injection?". To properly test this, two hypotheses were established. The first one "Using source code in the development cycle will result in a faster development cycle," is meant to determine how big of an impact the access to the source code will have on the development time for the aimbot. Meanwhile the second hypothesis "An aimbot using the method of Code Injection will be more performant than one using Image Recognition," aims to help determine whether the performance difference between Code Injection and Image Recognition is big enough to prefer one over the other.

2. CREATING THE DLL INJECTOR

Before making the code injector, the internet was scoured for different ways to accomplish DLL Injection. While searching, a simple way to accomplish this was found in the form of a YouTube video. (CasualGamer, 2019)

The injection works by enumerating all processes that have a visible window and giving the end user a list of both their process ID and the window name. This way the user can choose which process they want to inject the DLL into.

Once the user gives their chosen process ID, they will be asked to give the path to the DLL file they want to inject. Once the path has been given the program verify if it is a valid path and then will load in the DLL and attach it to the given process, thus creating the DLL injection. Now that the DLL is attached to the program, it is seen as part of the program and can access the memory allocated by the program to read and write to.

```

Available Targets:
PID: 25244    D:\School shit\GradWork\DllInjectionTest\Release\DLLInjector.exe
PID: 3180    DllInjectionTest (Running) - Microsoft Visual Studio
PID: 20224    DAE paper V1.docx - Word
PID: 6536     References - Zotero
PID: 15996    @Hard Carry - Discord
PID: 19452    Log in @ Instagram @ Mozilla Firefox
PID: 24408    Honkai: Star Rail
PID: 16620    Snipping Tool
PID: 22940    Calculator
PID: 20948    Microsoft Text Input Application
PID: 22940    Aaron @ Hogeschool West-Vlaanderen ?- OneNote for Windows 10
PID: 10788    DllInjectionTest
PID: 14472    GWLandMass Paper_EN-1.pdf - Adobe Acrobat Pro (64-bit)
PID: 7760     Program Manager

Choose you PID:
25244
Enter the path to the DLL: D
  
```

Figure 8 Image of the code injector.

3. MAKING THE DOOM AIMBOT

3.1 FINDING THE MEMORY OFFSETS

Before making the aimbot, the location in memory of the data related to both the player and the enemies had to be determined.

To accomplish this, the tool Cheat Engine (Cheat Engine, n.d.) was used. Cheat Engine is a freeware memory scanner/ debugger. This tool allows you to look under the hood of a program and at the memory that is used by said program. ('Cheat Engine Wiki', 2023)



Figure 9 Cheat engine logo ('Cheat Engine Wiki', 2023)

While originally planning to use this tool together with the Doom source code (DOOM, 2012/2023), the problem emerged that this code only compiles for Linux. To solve this problem an open-source port that compiles on Windows (Harding, 2013/2023)

A reddit comment was found (DVMirchev, 2017), that had a Compiler Explorer (Godbolt, n.d.) link, where the output is a detailed view of any given data structure.

Using this program, the player's position, look direction, and health values were found in memory. Also, the fact that the player object possesses a movable object pointer was discovered, this movable object contains a double linked list that points to all movable objects, among these the enemies.

```
Dumping AST Record Layout
0 | struct player_s
0 |   mobj_t * mo
8 |   playerstate_t playerstate
12 |   ticcmd_t cmd
12 |   signed char forwardmove
13 |   signed char sidemove
14 |   short angleturn
16 |   char buttons
20 |   int lookdir
24 |   fixed_t viewz
28 |   fixed_t viewheight
32 |   fixed_t deltaviewheight
36 |   fixed_t momx
40 |   fixed_t momy
44 |   int health
48 |   int armor
[...]
```

```
1648 |   int monstersgibbed
      |   [sizeof=1656, dsize=1656, align=8,
      |   nvsiz=1656, nvalign=8]
```

Figure 10 Abbreviated layout of the player_s object (Godbolt, n.d.-c)

```

Dumping AST Record Layout
0 | struct mobj_s
0 |   struct thinker_s thinker
0 |   struct thinker_s * prev
8 |   struct thinker_s * next
16 |   think_t function
24 |   struct thinker_s * cprev
32 |   struct thinker_s * cnext
40 |   unsigned int references
44 |   _Bool menu
48 |   fixed_t x
52 |   fixed_t y
56 |   fixed_t z
64 |   struct mobj_s * snext
72 |   struct mobj_s ** sprev
80 |   angle_t angle
84 |   spritenum_t sprite
88 |   int frame
96 |   struct mobj_s * bnext
104 |   struct mobj_s ** bprev
112 |   struct subsector_s * subsector
120 |   fixed_t floorz
124 |   fixed_t ceilingz
128 |   fixed_t dropoffz
132 |   fixed_t radius
136 |   fixed_t height
140 |   fixed_t momx
144 |   fixed_t momy
148 |   fixed_t momz
152 |   mobjtype_t type
[...]
408 |   int giblevel
412 |   int gibtimer
[ sizeof=416, dsize=416, align=8,
  nsize=416, nalign=8 ]

```

Figure 11 Abbreviated layout of the mobj_s object (Godbolt, n.d.)

3.2 MAKING THE AIMBOT

Once the offsets required to get the position of both the player, as well as the players view direction, and all positions of the enemies were found the progress on the actual aimbot started. The making of the actual aimbot was straight forward, since the DLL is running on a thread attached to the original application. Because of this accessing the internal data for reading and writing was easy to accomplish.

Next, the closest enemy was calculated, and after which both positions were used to calculate the new view angle. With the new view angle, the program wrote the data to the correct address and the aimbot worked.

```

unsigned CalculateHorizontalAimDirection(const Position& playerPos, const Position& enemyPos)
{
    // get delta x and y
    int deltaX = enemyPos.x - playerPos.x;
    int deltaY = enemyPos.y - playerPos.y;

    // get the angle in radians via atan2
    double angleRad = std::atan2(deltaY, deltaX);

    // Convert the angle to degrees
    double angleDeg = angleRad * (180.0 / M_PI);

    double aimDirection = (angleDeg < 0 ? angleDeg + 360.0 : angleDeg);

    // Remap the value from 0-360 to 0-intmax
    unsigned remappedAimDirection = static_cast<unsigned>(remap(aimDirection, 0u, 360u, gMinAngle, gMaxAngle));

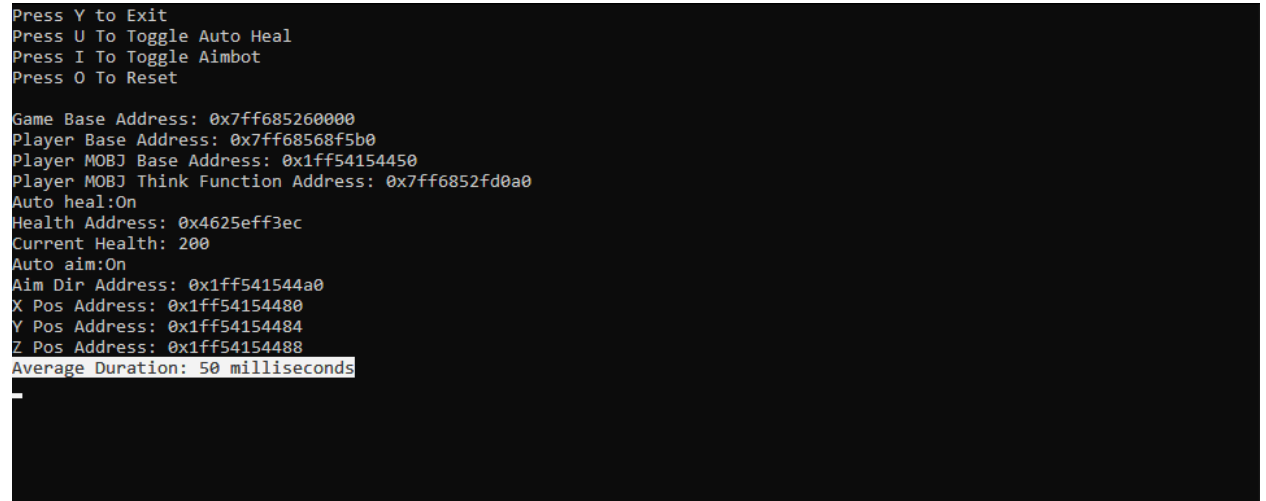
    return remappedAimDirection;
};

```

Figure 12 Code used for calculating the new aim direction

3.3 MEASURING THE PERFORMANCE, AND DEVELOPMENT TIME

With the aimbot completed, the program ran for 10 minutes measuring how long each update loop took. After storing all the data, the bottom and top 10% of the results were pruned. Using this pruned data, the average time one update loop takes was calculated. Originally the console logged a 0-millisecond update time, which was a suspiciously low result, but after adding the input delay to the measurements, the average update time became 50 milliseconds. This was the same time as the delay used by the inputs, thus leading to the conclusion that the aimbot took less than 1 millisecond to complete an update loop.



```
Press Y to Exit
Press U To Toggle Auto Heal
Press I To Toggle Aimbot
Press O To Reset

Game Base Address: 0x7ff685260000
Player Base Address: 0x7ff68568f5b0
Player M0BJ Base Address: 0x1ff54154450
Player M0BJ Think Function Address: 0x7ff6852fd0a0
Auto heal:On
Health Address: 0x4625eff3ec
Current Health: 200
Auto aim:On
Aim Dir Address: 0x1ff541544a0
X Pos Address: 0x1ff54154480
Y Pos Address: 0x1ff54154484
Z Pos Address: 0x1ff54154488
Average Duration: 50 milliseconds
```

Figure 13 Measurements for the Code Injection aimbot

The time used to make the DLL injector was not measured, due to it being used by both the Doom (*DOOM*, 2012/2023) and Call of Duty: World at War (*Call of Duty: World at War*, 2023) aimbots. The remaining time used for the Doom aimbot excluding the above, amounts to 22 hours and 15 minutes.

4. MAKING THE IMAGE RECOGNITION AIMBOT

4.1 MAKING THE IMAGE RECOGNITION

For the Image Recognition aimbot, a technique called Template Matching (ClarityCoders, 2021) was used. Here you take a reference image and use that image as a template to find in a bigger image. However, while making the aimbot, the fact that Doom (*DOOM*, 2012/2023) uses multiple sprites for each enemy became a problem. This made the templating so slow that it became a non-viable way to make the aimbot, thus proving the first hypothesis. The remaining time was spent focusing on the second hypothesis.

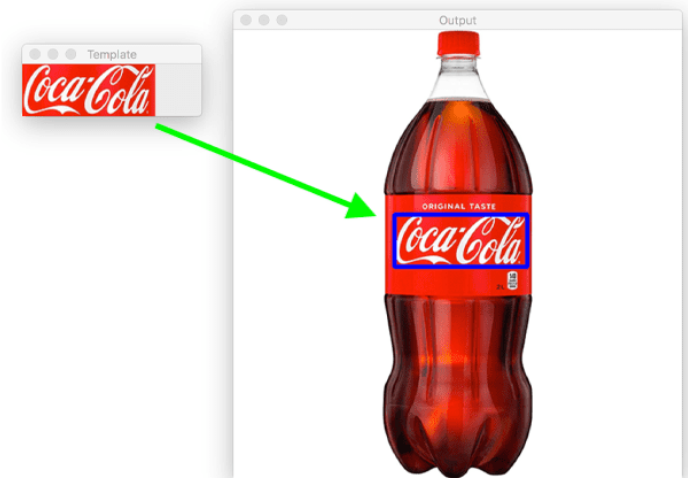


Figure 14 Another example of Image Templating (Rosebrock, 2021)

4.2 MEASURING THE PERFORMANCE

During the development of the Image Recognition aimbot, the fact that the window displaying the computer's vision was taking a considerable amount of time between frame updates was noticed. Further emphasizing the non-viability of this method. To salvage as much relevant data as possible it was decided to measure the update loop between identifiable milestones.

The first milestone used was being able to recognize the Doom (*DOOM*, 2012/2023) logo and the pause menu pointer with the Image Templating technique. This resulted in a slow, but still usable interval (approximating 100ms).

The second milestone was looking for possible sprites of the enemies in the first level, this resulted in at least a doubling of the time per update loop (exceeding 200ms). Seeing as each enemy has multiple sprites, and these sprites can all change size depending on the distance between the player and the enemy, it was not surprising the Image Recognition loop time increased even further (as shown in the image below).

```
Enter the path to the folder with enemies: D:\School shit\GradWork\ImageRecognitionBot\ImageRecognitionBot\Enemies
-----Starting the image recognition-----
-----Ending the image recognition-----
Average Duration: 502 milliseconds
Done
```

Figure 15 Measurements for the Image Recognition aimbot

Because of these measurements being almost unusable it was decided to focus on the second hypothesis.

5. MAKING THE CALL OF DUTY: WORLD AT WAR AIMBOT

5.1 FINDING THE MEMORY OFFSETS

Unlike Doom (*DOOM*, 2012/2023) where there was access to the source code, and thus the structure of the objects could be looked at, in Call of Duty: World at War (*Call of Duty: World at War*, 2023) that luxury was not available. Here, different techniques to find the offsets to all the needed variables were used. The unspecified variables in the game were reverse engineered by using Cheat Engine (Cheat Engine, n.d.) to identify whether a variable had changed as expected. Examples of this include looking left, looking right, moving the player, ...

This way the players health, position and look direction, as well as the list of zombies/enemies were found. Also, whether a zombie was dead or alive was identified.



Figure 16 Cover of Call of Duty: World at War (Call of Duty: World at War, 2023)

5.2 OVERRIDING THE LOOK FUNCTION

As part of the creation of the aimbot, the aim direction of the player needed to be changed, but changing the aim direction values did not change the players looking direction. Thus, creating an inconsistency between the aim of the player and what the player sees. After research, a video detailing how to override the function responsible for changing the player aim direction (CppObjectOrientedProgrammer, 2017) was found. For this a program called OllyDbg (OllyDbg, n.d.) was used, this tool allows users to override the assembly of a function with their own code. Using this program, the code responsible for taking the change in the mouse X and Y positions and storing them with the aim direction functions was transformed into code that just uses values stored in the aim direction x and y memory addresses. This change made it possible to calculate where the player had to look to aim at a zombie.

5.3 MEASURING THE DEVELOPMENT TIME

Similar to the Doom (*DOOM*, 2012/2023) aimbot, it was measured how long it took to finish the Call of Duty: World at War (*Call of Duty: World at War*, 2023) aimbot, excluding the creation of the DLL injector. The time spent on this aimbot turns out to be 22 hours and 30 minutes, roughly the same as the Doom (*DOOM*, 2012/2023) aimbot (22 hours and 15 minutes). While this is not a big difference, it should be mentioned that both the experience from making the Doom aimbot and the video explaining how to change the functions assembly immensely sped up the process for this aimbot, thus concluding that having the source code impacts the development time significantly.

DISCUSSION

1. DIFFERENCE IN DEVELOPMENT TIME

While the difference between the development time for the Doom (*DOOM*, 2012/2023) aimbot (22 hours and 15 minutes) and the Call of Duty: World at War (*Call of Duty: World at War*, 2023) aimbot (22 hours and 30 minutes), is only 15 minutes, this small difference can be attributed to multiple things. Firstly, when I was making the Doom (*DOOM*, 2012/2023) aimbot, I had no experience with Code Injection, or the tools that are used for this. Secondly, finding an explanation on how to do the aim part of the Call of Duty: World at War aimbot made the development a lot faster than if I had to figure out how to do this from scratch. So, while these results might not differ a lot from each other, I think I can still safely say that having the source code does help speed up development time.

Further it might be interesting to investigate how these factors influence the development of an aimbot for a more modern game, as well as how these findings can be used to prevent the easy creation of invasive software.

2. DIFFERENCE IN PERFORMANCE

As expected, the aimbot with Code Injection was more performant than the one made with Image Recognition. The difference between less than 1 millisecond (Code Injection) and more than 500 milliseconds (Image Recognition) means that there is no contest. Still, it could be useful to confirm this conclusion by making both aimbots for a different game.

It also turns out that the Code Injection method has other benefits that you cannot replicate with Image Recognition. Things such as infinite ammo, infinite health and wallhacks, can never be achieved with just Image Recognition. Both these factors confirm the second hypothesis.

CONCLUSION

The results from my research gave a better insight into Code Injection, and some of the effort that goes into doing it. I think this paper will serve as a good basis for people looking to get a better understanding of the concept and will also be useful for people looking to use Code Injection for goals other than an aimbot. It also demonstrates that Code Injection has superior performance compared to Image Recognition, leading to the conclusion that it is wise to focus on making Code Injection less available to prevent people from gaining an unfair advantage.

With a development time of approximately 23 hours the creation of an aimbot is not needlessly long or unattainable for people with more experience. This makes the effort versus reward ratio attractive for potential cheaters.

FUTURE WORK

1. TRAIN AI USING CODE INJECTION

One avenue that could be researched further is the use of Code Injection together with AI machine learning models such as a neural network to complete certain goals in a game, such as training an AI to beat Doom (*DOOM*, 2012/2023) using Code Injection. This could be an interesting continuation of what was researched in this paper.

2. COMPARE CODE INJECTION TO AI TRAINED FOR IMAGE RECOGNITION

While the attempts to make an aimbot with image templating failed, the endeavor to explore the possibility of using Image Recognition with AI, might give better results. For example, training AI to recognize certain things, such as enemies, might result in a more performant aimbot.

3. WAY TO PREVENT CODE INJECTION

Exploring ways to prevent the use of Code Injection in a game, such as detecting processes that do not belong to said game, might be interesting to explore. This could help with the development of anti-cheat software, offering game developers ways of safeguarding their games.

CRITICAL REFLECTION

This paper has been a great opportunity for me to explore a topic that I have had some interest in for some time. While I would have liked to explore the topic of Code Injection more broadly, the time limit for this paper unfortunately prevented me from being able to do so. I plan on broadening my knowledge on both the topic of Code Injection and assembly soon. The process of going through this research, while hard at times, has been a great endeavor, and has thought me some valuable things such as what a good source code is, as well as the value of documentation, and how to correctly plan for undergoing a bigger task such as this paper.

REFERENCES

- Call of Duty: World at War. (2023). In *Wikipedia*.
https://en.wikipedia.org/w/index.php?title=Call_of_Duty:_World_at_War&oldid=1191009145
- CasualGamer (Director). (2019a, November 15). *Hacking Terraria in C++ | (1/3) | Self unloading DLL*.
<https://www.youtube.com/watch?v=uUMg7CeJF1k>
- CasualGamer (Director). (2019b, December 25). *How To Make A DLL Injector C++*.
<https://www.youtube.com/watch?v=44-TOflGBzk>
- Cheat Engine*. (n.d.). Retrieved 29 December 2023, from <https://www.cheatengine.org/index.php>
- Cheat Engine: View topic—Can't find pointer base of a non-static address*. (n.d.). Retrieved 17 December 2023, from <https://forum.cheatengine.org/viewtopic.php?t=552974>
- Cheat Engine Wiki. (2023). In *Wikipedia*.
https://en.wikipedia.org/w/index.php?title=Cheat_Engine&oldid=1191213789
- ClarityCoders (Director). (2021, April 19). *Dominating an Online Game with Object Detection Using OpenCV - Template Matching*. <https://www.youtube.com/watch?v=vXqKnivE6P8>
- Code Injection | OWASP Foundation*. (n.d.). Retrieved 27 December 2023, from https://owasp.org/www-community/attacks/Code_Injection
- CppObjectOrientedProgrammer (Director). (2017, November 3). *Hacking the game Call of Duty, World at War—Part 7—C++ Programming—VERY basic AimBot*. <https://www.youtube.com/watch?v=yswLzKgDAbs>
- DLL injection. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=DLL_injection&oldid=1184659932
- DOOM*. (2023). [C]. id Software. <https://github.com/id-Software/DOOM> (Original work published 2012)
- Doom Logo*. (n.d.). Retrieved 23 December 2023, from https://static.wikia.nocookie.net/logopedia/images/0/09/Doom_1.gif/revision/latest/scale-to-width-down/1000?cb=20180226150858
- DVMirchev. (2017, March 19). *Is there an online compiler showing clang output?* [Reddit Post]. R/Cpp. www.reddit.com/r/cpp/comments/608rtm/is_there_an_online_compiler_showing_clang_output/
- Godbolt, M. (n.d.-a). *Compiler Explorer*. Retrieved 29 December 2023, from <https://godbolt.org/z/M8cvcGxM3>
- Godbolt, M. (n.d.-b). *Compiler Explorer—MOBJ*. Retrieved 11 December 2023, from <https://gcc.godbolt.org/z/WE93ds8z3>
- Godbolt, M. (n.d.-c). *Compiler Explorer—Player*. Retrieved 29 December 2023, from <https://gcc.godbolt.org/>
- Guided Hacking (Director). (2020, January 17). *How To Find Offsets, Entity Addresses & Pointers*.
<https://www.youtube.com/watch?v=YaFlh2pIKAg>
- Harding, B. (2023). *Bradharding/doomretro* [C]. <https://github.com/bradharding/doomretro> (Original work published 2013)

Hex Murder (Director). (2016, November 30). *How to find an Entity List / Object List | Cheat Engine*.
<https://www.youtube.com/watch?v=0Hr-8rH3nWs>

League of Legends. (n.d.). Retrieved 14 January 2024, from <https://www.leagueoflegends.com/fr-fr/>

Lohokare, A., Shah, A., & Zyda, M. (2020). Deep Learning Bot for League of Legends. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1), Article 1.
<https://doi.org/10.1609/aiide.v16i1.7449>

OllyDbg. (n.d.). Retrieved 30 December 2023, from <https://www.ollydbg.de/>

OpenCV About. (n.d.). OpenCV. Retrieved 27 December 2023, from <https://opencv.org/about/>

OpenCV: OpenCV modules. (n.d.). Retrieved 27 December 2023, from <https://docs.opencv.org/3.4/index.html>

OpenCVLogo. (n.d.). GitHub. Retrieved 27 December 2023, from
<https://github.com/opencv/opencv/wiki/OpenCVLogo>

Rosebrock, A. (2021, March 22). OpenCV Template Matching (`cv2.matchTemplate`). *PyImageSearch*.
<https://pyimagesearch.com/2021/03/22/opencv-template-matching-cv2-matchtemplate/>

SALIANKHAM, K. (n.d.). *What is Code Injection on Windows? | Codementor*. Retrieved 27 December 2023, from
<https://www.codementor.io/@arksong123/what-is-code-injection-on-windows-vd66o7wsz>

Shemirani, B. (2019, July 25). *Answer to 'How to turn a screenshot bitmap to a cv::Mat'*. Stack Overflow.
<https://stackoverflow.com/a/57207445>

Wilson, T. (n.d.). *DLL Injection: Part One*. Retrieved 26 December 2023, from <https://blog.nettitude.com/uk/dll-injection-part-one>

ACKNOWLEDGEMENTS

First, I would like to thank Tom Tesch, both for guiding me throughout the process and for advising me to use the game DOOM (*DOOM*, 2012/2023) instead of more recent and difficult games.

Next, I would like to thank Arno Poppe for helping me decide what my hypotheses should be, and for giving insight into some of the choices made during the project.

Finally, I would like to thank Bart Coppens for giving insight not only on how to make a good bibliography, but also how to reference this correctly. Further he also helped me decide what methodology would be best for this paper.

APPENDICES

1. This is the link to the GitHub repository with both the source code for the DLL injector and with the source code for the DLLs used in this project: <https://github.com/AaronFrans/DLL-Injection>.
2. This is the link to the GitHub repository with the source code of the Image Recognition aimbot: <https://github.com/AaronFrans/ImageRecognition>.