Game Hacking Academy

A Beginner's Guide to Understanding Game Hacking Techniques

Created from materials originally published from 2019 to 2021 on https://gamehacking.academy and https://github.com/GameHackingAcademy.

Created in 2021.

This book is distributed without charge. If you want to support future works, feel free to donate:

https://www.buymeacoffee.com/gamehackingacad



Have questions or notice issues? Feel free to contact me via email or Twitter:

attilathedud@gmail.com

https://twitter.com/GameHackingAcad

Computer Photo by Luke Hodde on Unsplash

Licensed under the Creative Commons Attribution-NonCommercial 4.0 International Public License.

Table of Contents

Table of Contents 3

Introduction 7

External Resources 8

Basics 9

1.1 Computer Fundamentals 10

1.2 Game Fundamentals 18

1.3 Hacking Fundamentals 24

1.4 Setting Up a Lab VM 26

1.5 Memory Hack 29

Debugging & Reversing 37

2.1 Debugging Fundamentals 38

2.2 Reversing Fundamentals 41

2.3 Changing Game Code 45

2.4 Reversing Code 56

2.5 Code Caves 68

2.6 Using Code Caves 72

2.7 Dynamic Memory Allocation 78

2.8 Defeating DMA 82

Programming 90

3.1 Programming Fundamentals 91

3.2 External Memory Hack 95

3.3 DLL Memory Hack 110
3.4 Code Caves & DLL's 122
3.5 Printing Text 131
RTS Hacks 142
4.1 Stathack 143
4.2 Map Hack 151
4.3 Macro Bot 160
FPS Hacks 175
5.1 3D Fundamentals 176
5.2 Wallhack (Memory) 183
5.3 Wallhack (OpenGL) 196
5.4 Chams (OpenGL) 217
5.5 Triggerbot 221
5.6 Aimbot 230
5.7 No Recoil 253
5.8 Radar Hack 260
5.9 ESP 266
5.10 Multihack 298
Multiplayer 312
6.1 Multiplayer Fundamentals 313
6.2 Packet Analysis 317
6.3 Reversing Packets 331
6.4 Creating an External Client 345
6.5 Proxying TCP Traffic 356

Playrix Ex. 1024, page 4

Tool Development 365
7.1 DLL Injector 366
7.2 Pattern Scanner 372
7.3 Memory Scanner 380
7.4 Disassembler 389
7.5 Debugger 403
7.6 Call Logger 409
Appendix A 415
A.1 Lab VM Setup Script 416
A.2 Wesnoth External Gold Hack 416
A.3 Wesnoth Internal Gold Hack 418
A.4 Wesnoth Code Cave DLL 419
A.5 Wesnoth Stathack 421
A.6 Wesnoth Map Hack 424
A.7 Wyrmsun Macrobot 425
A.8 Urban Terror Memory Wallhack 428
A.9 Urban Terror OpenGL Wallhack 430
A.10 Urban Terror OpenGL Chams 433
A.11 Assault Cube Triggerbot 437
A.12 Assault Cube Aimbot 439
A.13 Assault Cube No Recoil 442
A.14 Assault Cube ESP 443
A.15 Assault Cube Multihack 448
A.16 Wesnoth Multiplayer Bot 480

Playrix Ex. 1024, page 5

A.17 Wesnoth ChatBot 483
A.18 Wesnoth Proxy 487
A.19 DLL Injector 491
A.20 Pattern Scanner 493
A.21 Memory Scanner 495
A.22 Disassembler 499
A.23 Debugger 503
A.24 Call Logger 506

Playrix Ex. 1024, page 6

Introduction

Hacking games requires a unique combination of reversing, memory management, networking, and security skills. Even as ethical hacking has exploded in popularity, game hacking still occupies a very small niche in the wider security community. While it may not have the same headline appeal as a Chrome 0day or a massive data leak, the unique feeling of creating a working aimbot for a game and then destroying a server with it is hard to replicate in any other medium.

When I first started learning game hacking years ago, resources were spread out across several sites and were very sparse. Typically, you would find a section of code that linked to a broken site. You would then search around for some forum that would have some part of the broken site in a post and piece together the information. While this rewarded thorough searching, it was a massive time-sink. These days, there are several places where you can find a variety of information regarding game hacking. You can find boilerplate code for almost any engine, along with the memory offsets for any structure you may care about.

However, one area still underserved by all the information out today is the concepts and fundamentals behind the offsets. My hope is that this book helps fill those gaps.

- attilathedud

Playrix Ex. 1024, page 7

External Resources

This is a list of all the external resources, such as tools and games, used in this book. They are ordered by their appearance. This is intended to help if you plan to read this book in a location without access to the Internet.

- VirtualBox (https://www.virtualbox.org/wiki/Downloads)
- Windows 10 Evaluation Virtual Machine (https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/)
- Cheat Engine (https://cheatengine.org/)
- x64dbg (https://x64dbg.com/)
- The Battle of Wesnoth 1.14.9 (https://www.wesnoth.org/)
- Visual Studio 2019 Community Edition (https://visualstudio.microsoft.com/vs/community/)
- Wyrmsun (https://store.steampowered.com/app/370070/Wyrmsun)
- Urban Terror 4.3.4. (https://www.urbanterror.info/)
- Assault Cube 1.2.0.2 (https://assault.cubers.net/)
- Wireshark (https://www.wireshark.org/)
- cmder (https://cmder.net/)
- ZLib (https://zlib.net/)
- The Battle of Wesnoth 1.14.12 (https://www.wesnoth.org/)

Part 1
Basics

1.1 Computer Fundamentals

1.1.1 Computer Components

A typical computer has several connected components. Among the most important are:

- Hard-drive
- RAM
- Video card
- Motherboard
- CPU

If you were to remove the side of a desktop computer, the parts might be placed in a configuration like so:

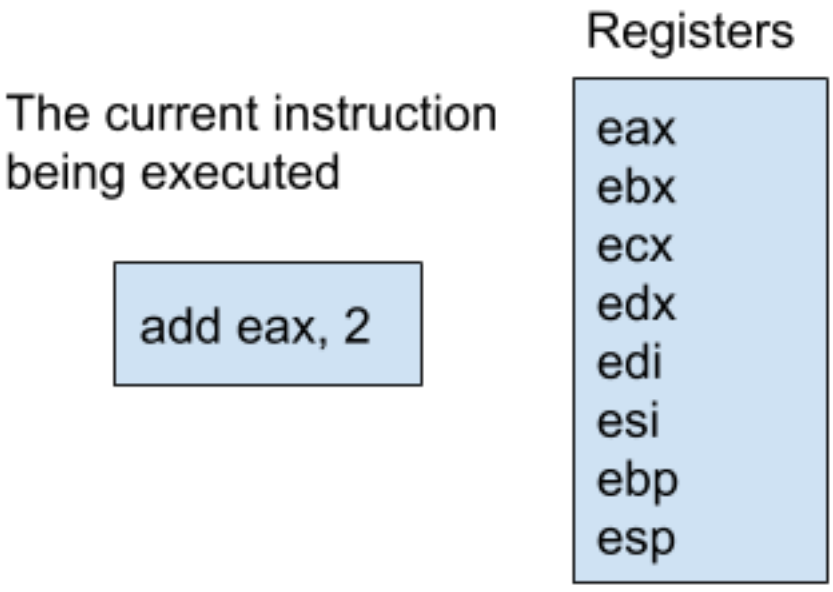


For our purposes, we will only briefly address the first four components, and then we will focus on the CPU in the next section:

- Hard-drives are responsible for storing large files, such as photos, executables, or system files.
- RAM holds data that needs to be accessed quickly. Data is loaded into RAM from the hard-drive.
- Video cards are responsible for displaying graphics to the monitor.
- Motherboards tie all these components together and allow them to communicate.

1.1.2 The CPU

The CPU is the brain of the computer. It is responsible for executing instructions. These instructions are simplistic and vary depending on the architecture. For example, an instruction might add two numbers together. To speed up execution time, the CPU has several special areas where it can store and modify data. These are called registers.



Playrix Ex. 1024, page 11

1.1.3 Instructions

All computer programs are made up of a series of instructions. As we discussed above, an instruction is simplistic and typically only does one thing. For example, the following are some of the instructions found in most architectures:

- Add two numbers.
- Subtract two numbers.
- Compare two numbers.
- Move a number into a section of memory (RAM).
- Go to another section of code.

Computer programs are developed from these simple instructions combined together. For example, a simple calculator might look like:

```
mov eax, 5
mov ebx, 4
add eax, ebx
```

The first instruction (**mov**) moves the value of 5 into the register **eax**. The second moves the value of 4 into the register **ebx**. The **add** instruction then adds **eax** and **ebx** and places the result back into **eax**.

1.1.4 Computer Programs

Computer programs are collections of instructions. Programs are responsible for receiving a value (the input) and then producing a value (the output) based on the received value.

For example, one simple program could take a number as the input, increase the number by 1, and then move it into an output. It might look like:

```
mov eax, input
add eax, 1
mov output, eax
```

A more complex program would have many of these simple programs "inside" of it. In this context, these simple internal programs are called functions. Functions, just like programs, take an input and produce an output. For example, we could make our previous program a function that does the same thing. It might look like:

```
function add(input):
    mov eax, input
    add eax, 1
    mov output, eax
```

We could also make another function that does a similar operation. For example, we could write a function to decrease a number by 1:

```
function subtract(input):
    mov eax, input
    sub eax, 1
    mov output, eax
```

These two functions (**add** and **subtract**) can then be used to create a more complex program. This new program will take a number and either increment or decrement it. It will take two inputs:

1. A number
2. A mathematical operation, in this case, add (+) or subtract (-)

This new program will be longer and have two different ways it can execute. These will be explained after the code:

```
function add(input):
    mov eax, input
    add eax, 1
    mov output, eax

function subtract(input):
    mov eax, input
    sub eax, 1
    mov output, eax

cmp operation, '-'
```

```
je subtract_number
add(number)
exit

subtract_number:
    subtract(number)
    exit
```

This code has two functions at the top. Like we discussed, these take an input and then either add or subtract 1 from the input to produce the output. The **cmp** instruction compares two values. In this case, it is comparing the operation type received as input and a value coded into the program, **-**. If these values are equal, we go to (or jump to) another section of code (**je** = jump if equal).

If the operation is equal to **-**, we go to code that subtracts 1 from the number. Otherwise, we continue the program and add 1 to the number before exiting.

Comparing numbers and then jumping to different code depending on their value is known as branching. Branching is a key component of designing complex programs that can react to different input. For example, a game will often have a branch for each direction a player can move in.

1.1.5 Binary, Decimal, and Hexadecimal

Fundamentally, CPU's are circuits. Circuits either have electricity flowing through them (on) or they do not (off). These two states can be represented by a binary (or base-2) numeral system. In a base-2 system, you have two possible values: 0 and 1. An example binary number is 1101.

We are familiar with a decimal (or base-10) numeral system, which has 10 possible values: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. An example decimal number is 126. This number can be represented in a more explicit format as:

$$(1 * 10^2) + (2 * 10^1) + (6 * 10^0)$$

We can represent the binary number above (1101) in the same format. However, we will replace the 10's with 2's, as we are switching from a base-10 to base-2 system:

$$(1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0)$$

Binary numbers can quickly become unwieldy when they need to represent larger values. For example, the binary representation for the decimal number 250 is 11111010.

To represent these larger binary numbers, hexadecimal (base-16) numbers are commonly used in computing. Hexadecimal numbers are usually prefixed with the identifier `0x` and have sixteen possible values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. An example hexadecimal number is `0xA1D`.

1.1.6 Programming Languages

Instructions are represented as numbers, like all other data on a computer. These numbers are known as operation codes, often shortened to opcodes. Opcodes vary by architecture. The CPU knows each opcode and what it needs to do when encountering each one. Opcodes are commonly represented in hexadecimal format. For example, if an Intel CPU encounters a `0xE9`, it knows that it needs to execute a **jmp** (jump) instruction.

Early computers required programs to be written in opcodes. This is obviously hard to do, especially for more complex programs. Variants of an assembly language were then adopted, which allowed writing of instructions. They look similar to the examples we wrote above. Assembly language is easier to read than just opcodes, but it is still hard to develop complex programs in.

To improve the development experience, several higher-level languages were developed, such as FORTRAN, C, and C++. These languages are easy to read and introduced flow-control operations like **if** and **else** conditionals. For example, the following is our increment/decrement program in C. In C, an **int** refers to an integer, or a whole number (-1, 0, 1, or 2 are examples).

```
int add(int input) {
    return input + 1;
}

int subtract(int input) {
    return input - 1;
}

if(operation == '-') {
    subtract(number);
```


Playrix Ex. 1024, page 15

```
}
else {
    add(number);
}
```

All these higher-level languages are compiled down to assembly. A traditional assembler then turns that assembly into opcodes that the CPU can understand.

1.1.7 Operating Systems

Writing programs to communicate with hardware is a time-consuming and difficult process. To execute our increment/decrement program, we would also have to write code to handle keystrokes from a keyboard, display graphics to the monitor, build out character sets so we could represent letters and numbers, and communicate with the RAM and the hard-drive. To make it easier to develop programs, operating systems were created. These contain code that already can handle these hardware functions. They also have several standard functions that are commonly used, such as copying data from one location to another.

The three main operating systems still in use today consist of Windows, Linux, and MacOS. All of these have different libraries and methods to communicate with the hardware. This is why programs written for Windows do not work on Linux.

1.1.8 Applications

Operating systems need a way to determine how to handle data when a user selects it. If the data is a photo, the operating system wants to bring up a specific application (like Paint) to view the photo. Likewise, if the data is an application itself, the operating system needs to pass it to the CPU to execute.

Each operating system handles executing uniquely. In Linux, a special executable permission is set on a normal file. In Windows, applications are formatted in a special way that Windows knows how to parse. This is referred to as the PE, or Portable Executable, format. The PE format has several sections, such as the **.text** section for holding program code, and the **.data** section for holding variables.

Playrix Ex. 1024, page 16

1.1.9 Games

With all of that out of the way, we can finally discuss games. Games are simply applications. On Windows, they are formatted in a PE format, identical to any other application. They contain a **.text** section that holds program code, made up of opcodes. These opcodes are then executed by the CPU, and the operating system displays the resulting graphics and handles input, like key presses.

Playrix Ex. 1024, page 17

1.2 Game Fundamentals

1.2.1 Parts of a Game

While games are applications, they are complex and made up of several parts. Some of these include:

- Graphics
- Sounds
- Input
- Physics
- Game logic

Due to each part's complexity, most games use external functions for these parts. These external functions are combined into what is called a library. Libraries are then used by other programs to reduce the amount of code written. For example, to draw images and shapes to a screen, most games use either the DirectX or OpenGL library.

For some types of hacks, it is important to identify the libraries being used. A wallhack is a type of hack that allows the hacker to see other players through solid walls. One method of programming a wallhack is modifying the game's graphics library. Both OpenGL and DirectX are vulnerable to this type of hack, but each requires a different approach.

For most hacks, we will be modifying the game logic. This is the section of instructions responsible for how the game plays. For example, the game logic will control how high a character jumps or how much money the player receives. By changing this code, we can potentially jump as high as we want or gain an infinite amount of money.

Playrix Ex. 1024, page 18

1.2.2 Game Structure

Game logic is made up of instructions, like all computer code. Due to the complexity of games, they are often written in a high-level language and compiled. Understanding the general structure of the original code is often required for more complex hacks.

Most games have two major functions:

- Setup
- Main Loop

The setup function is executed when the game is first started. It is responsible for loading images, sounds, and other large files from the hard-drive and placing them in RAM. The main loop is a special type of function that runs forever until the player quits. It is responsible for handling input, playing sounds, and updating the screen, among other things. An example main loop might look like:

```
function main_loop() {
    handle_input();
    update_score();
    play_sound_effects();
    draw_screen();
}
```

All of these functions in turn call other functions. For example, the **handle_input** function might look like:

```
function handle_input() {
    if( keydown == LEFT ) {
        update_player_position(GO_LEFT);
    }
    else if( keydown == RIGHT ) {
        update_player_position(GO_RIGHT);
    }
}
```

Every game is programmed differently. Some games might prioritize updating graphics before handling input. However, all games have a main loop of some sort.

Playrix Ex. 1024, page 19

1.2.3 Data and Classes

Any data that can be updated in a game is stored in a variable. This includes things like a player's score, position, or money. These variables are declared in the code. An example variable definition in C might look like:

```
int money = 0;
```

This code would declare the **money** variable as an integer. Like we learned in the last chapter, integer values in C are whole numbers (like 1, 2, or 3). Imagine if we had to track the money for several players. One way to do this would be to have several declarations, like so:

```
int money1 = 0;
int money2 = 0;
int money3 = 0;
int money4 = 0;
```

One downside to this approach is that it is hard to maintain as the game gets larger and more complex. For example, to write code that increases every player's money by 1, we would have to manually update each variable:

```
function increase_money() {
    money1 = money1 + 1;
    money2 = money2 + 1;
    ...
}
```

If we added another player, we would have to go and update every section of code that altered the players' money. A better approach is to declare these values in a list. We can then use an instruction known as a loop to go through every item in the list. This is known as iteration. In C, lists are commonly implemented using what is known as an array. For our purposes, you can assume lists and arrays are synonymous. One type of loop in C is known as a for loop. For loops are divided into three segments: the starting value, the ending value, and how to update the value after each iteration. An example of the previous code might be written like:

Playrix Ex. 1024, page 20

```
int money[10] = { 0 };
int current_players = 4;

function increase_money() {
    for(int i = 0; i < current_players; i++) {
        money[i] = money[i] + 1;
    }
}
```

We now would only have to update the **current_players** variable to add support for another player.

To make it easier to develop complex applications, developers often use a programming model known as object oriented programming, or OOP. In OOP, variables and functions are grouped together into collections called classes. Classes are usually self-contained. For example, many games will have a Player class. This class will contain several variables like the player's position, name, or money. These variables inside the class are known as members. Classes will also contain functions to modify these members. One example of a Player class might look like:

```
class Player {
    int money;
    string name;

    function increase_money() {
        money = money + 1;
    }
}
```

Games will often contain lists of classes. For example, the game Quake 3 has an array of all the players currently connected to a server. Each player has their own Player class in the game. To calculate the score screen, the game will go over every player in the list and look at the amount of kills they have.

1.2.4 Memory

Games have a lot of large resources, like images and sounds. These must be loaded from the hard-drive, usually in the game's setup phase. Once loaded, they are placed

Playrix Ex. 1024, page 21

in RAM, along with the game's code and data. Because games are so large, they must constantly load different data from the RAM into registers to operate on. This loading is typically done by a **mov** command. This command will move a section of memory into a register. Our **increase_money** example function executed by the CPU might look like:

```
function increase_money:
    mov eax, 0x12345678
    add eax, 1
    mov 0x12345678, eax
```

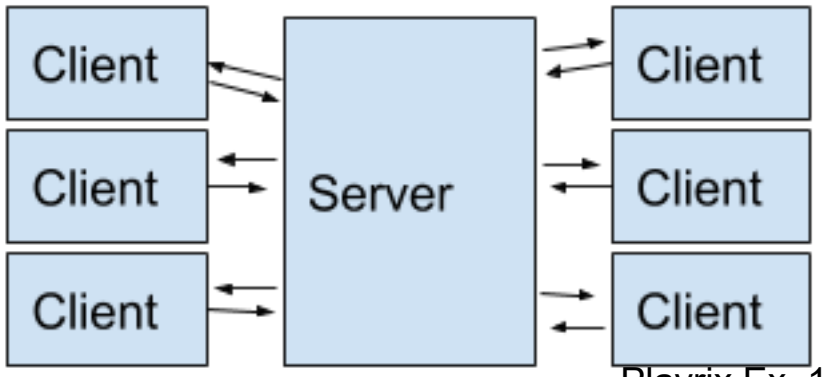
In this example, we use `0x12345678` as the location in RAM of the player's money. Most games will have this structure but a different location. For more complex games, these locations will be based on other locations. If our game had a Player class, the **increase_money** code executed by the CPU would need to use the Player's class location to retrieve the money.

```
function increase_money:
    mov ebx, 0x12345670
    mov eax, ebx + 8
    add eax, 1
    mov ebx+8, eax
```

In this case, the CPU had to offset the money's location based on the location of the Player class.

1.2.5 Multiplayer Clients

Multiplayer games allow multiple players to interact with each other. To allow this, multiplayer games make use of clients and servers. An example of the client-server model is shown below:



Playrix Ex. 1024, page 22

Clients represent each player's copy of the game and contain all the information regarding the local game. For example, each client will contain that player's money. When a player causes an action to change their money, the client is responsible for sending this update to the server.

Information can also be sent in both directions. An example of this is player movement. One client will tell the server that the player has moved their position. The server will then tell all other clients to update their associated positions for the moved client.

1.2.6 Multiplayer Servers

While the client represents each player's copy of the game, the server ensures that all the connected clients are playing the same copy of the game. Servers will often restrict what changes they accept from clients. For example, imagine we wrote a hack to change our money in a game. If it is a multiplayer game, the server will reject our changes. This is why single-player hacks will often not work in multiplayer.

We will discuss multiplayer fundamentals further in a future chapter.

Playrix Ex. 1024, page 23

1.3 Hacking Fundamentals

1.3.1 Hacking Steps

All hacking consists of modifying memory in a game. Writing any hack involves four main steps:

1. Identify what you want to change.
2. Understand what memory you need to locate.
3. Locate that memory in the game.
4. Change that memory.

These steps apply for any hack, regardless of the complexity.

1.3.2 Identify

The first step for any hack is to identify what you want to do. Different hacks will require different approaches. For example, modifying a player's money will require a memory modification of a variable, whereas seeing other players through walls will require a memory modification of the game's code.

1.3.3 Understand

To modify memory, we need to locate it. Before we attempt to locate it, we need to understand what memory we need to locate. In some cases, the memory you want to modify will be a variable. In other cases, you will want to modify large sections of code. There are three main types of modifications:

- Variables, like modifying the value of a player's money
- Code, like modifying how walls are drawn
- Files, like modifying the saved items in your inventory

Playrix Ex. 1024, page 24

1.3.4 Locate

Once you know what you want to change and understand where to look, you can start looking for it. For some hacks, this may involve searching memory with a tool called a memory scanner. For others, it may involve looking through the game's code using a tool called a debugger. Depending on the game and approach, this can be a time-consuming process.

1.3.5 Change

After you have located the memory, the last step is to change it. Initially, this will mean using a memory scanner or debugger to manually modify the memory. Once you have verified that this works, you can write a program to automatically change it for you.

Playrix Ex. 1024, page 25

1.4 Setting Up a Lab VM

1.4.1 Overview

When doing any sort of hacking, it's best practice to separate your personal and hacking machines. One easy way to achieve this is using a virtual machine, or VM. In this chapter, we will set up a game hacking VM running Windows 10. This will be the environment we use for all the other chapters as well.

Due to the hardware requirements of games, you will often find that it will be impossible to run a certain game in a VM. In these cases, one option is to create another section on your hard-drive known as a partition. You can then install Windows on this separate partition and reserve it for game hacking.

1.4.2 VirtualBox

For this book, we will be using a virtual machine hypervisor known as VirtualBox. This software allows you to run and manage virtual machines. You can download it here.

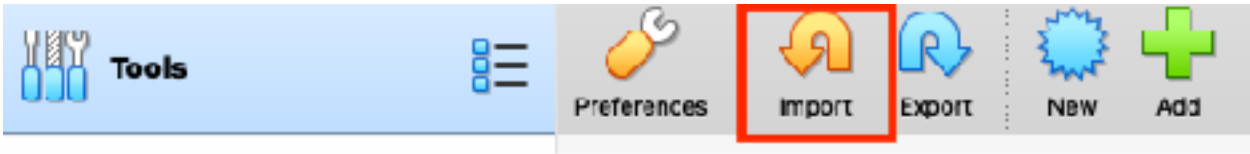
1.4.3 Virtual Machine

Next, we need a virtual machine. Since most games are released for Windows, we will be using a Windows 10 VM. Microsoft releases free Windows 10 VMs for testing old versions of Internet Explorer. These are completely legal but have a 90-day limit for activation. For our purposes, we will download a VirtualBox image. You can download the image here. Be aware, these VMs are about 7GB in size. Once the image is downloaded, extract the OVF file from the archive. VirtualBox uses extensions of OVA and OVF for images. These images contain a saved copy of a pre-configured machine.

Once we set up our machine, we will create a snapshot of it. This will allow us to throw out the old machine and create a new copy when it expires. Never keep notes or anything personal on your VM.

Playrix Ex. 1024, page 26

Open VirtualBox and import the downloaded OVF. Keep all the options as the default.



Once the VM is imported, start it up. Windows 10 will start and send you to a login screen. The password for the user is "Passw0rd!", without the quotes.

1.4.4 Tools

We will use a Boxstarter script to install some game hacking tools. Boxstarter is a set of utilities that allows you to script and recreate installations. In this case, we will use it to install a memory searcher and debugger. Within your VM, open up Powershell and run the following two commands:

```
. { iwr -useb https://boxstarter.org/bootstrapper.ps1 } | iex;
Get-Boxstarter -Force

Install-BoxstarterPackage -PackageName https://
raw.githubusercontent.com/GameHackingAcademy/vmsetup/master/
vmsetup.txt -DisableReboots
```

The *vmsetup.txt* file is also available in Appendix A. If using the local version, run the following second command instead of the version above:

```
Install-BoxstarterPackage -PackageName C:
\location\of\vmsetup.txt -DisableReboots
```

The first command installs BoxStarter and is taken from their website. The second command is a setup script that enables some folder options and installs three programs:

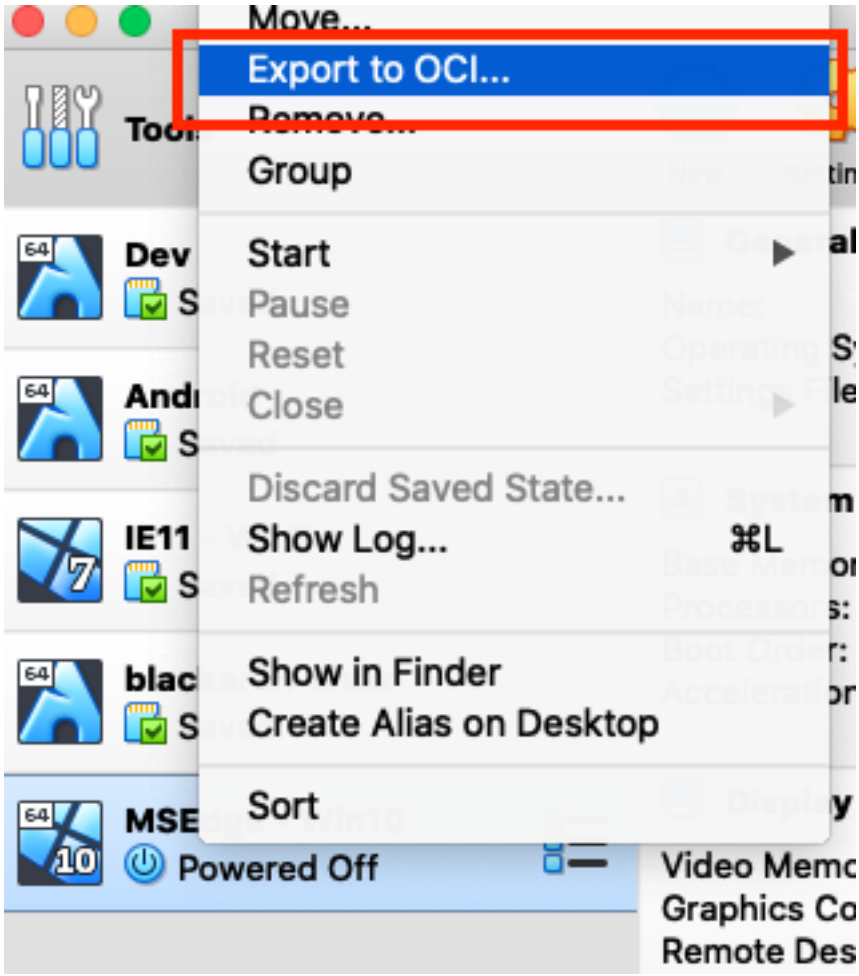
1. Cheat Engine, a memory scanner
2. x64dbg, a debugger
3. Chocolatey, a package manager

As you discover more tools, you should create your own script and add them.

Playrix Ex. 1024, page 27

1.4.5 Cloning VMs

Now that we have our environment set up, we will create a clone. This will allow us to recreate our VM with all the tools already installed. Power off the VM completely and then choose *Export to OCI*. Keep all the defaults and begin the export.



This will create an OVA, similar to the OVF we initially downloaded. If we ever corrupt our environment, we can simply delete it and import this new OVA. It will already have all of our tools installed, so we don't need to waste time reinstalling them.

1.5 Memory Hack

1.5.1 Target

For our first hack, we will be targeting a game called "The Battle for Wesnoth", shortened to Wesnoth. This is a free, open-source game that has no anti-cheating mechanisms in place. Most of the chapters in this book will specifically target version 1.14.9. To install it on our VM, open up Powershell as an administrator and run the following command:

```
choco install wesnoth --version=1.14.9 -y
```

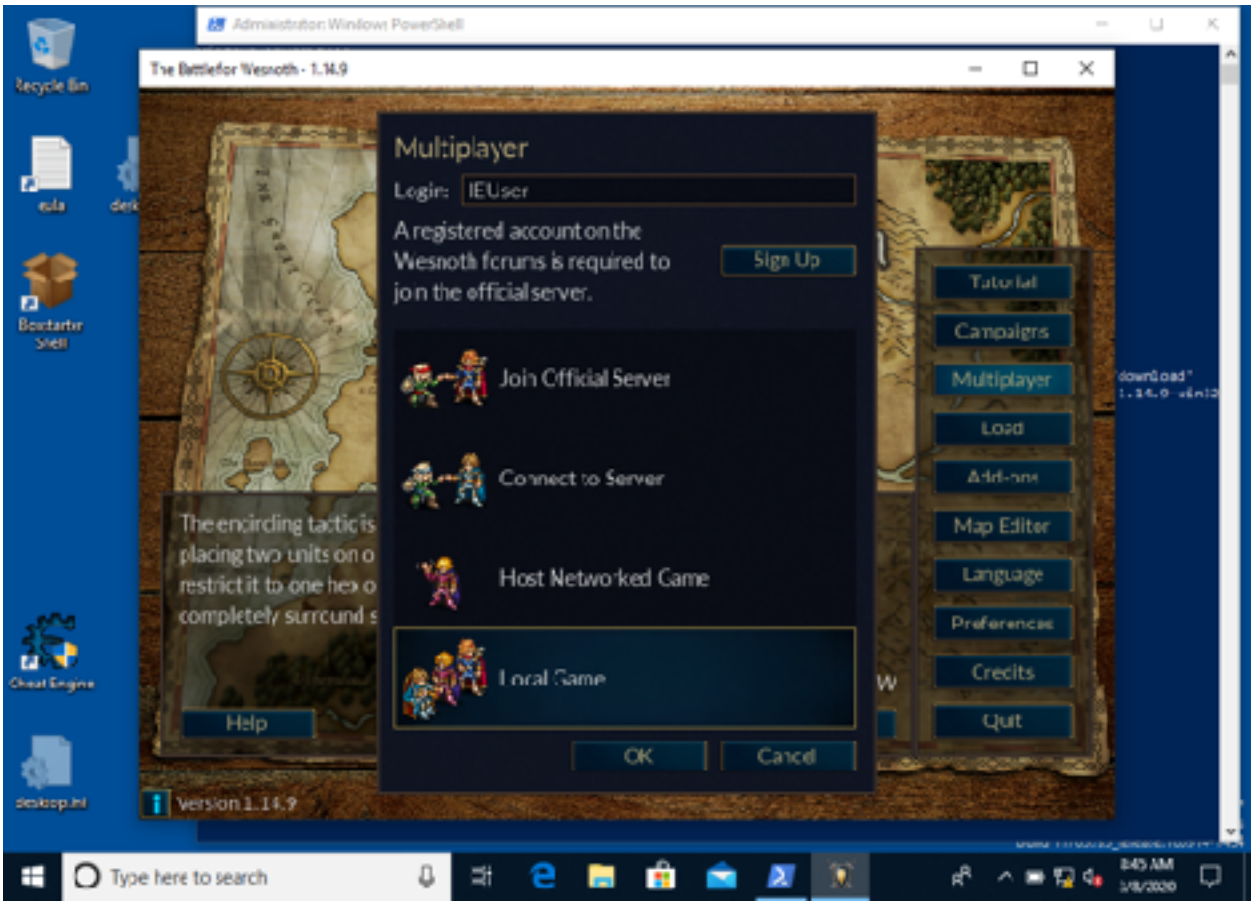
This will use Chocolatey to install The Battle for Wesnoth. Once the installation finishes, open the game and verify that it works. Then, go into the *Preferences* menu and change the game's video mode to *Windowed*. Finally, play the tutorial mission to learn about how the game works.

Playrix Ex. 1024, page 29

1.5.2 Identify

Our goal for this hack is to change our gold. Players use gold to buy troops in the game, so the more gold we have, the more troops we can buy. We will accomplish this by using a tool called Cheat Engine, which allows us to scan and modify game memory. To do that, we will use the steps we learned in Chapter 1.3.

Players only receive gold while playing a scenario. To create a scenario, select *Multiplayer* and then *Local Game*.

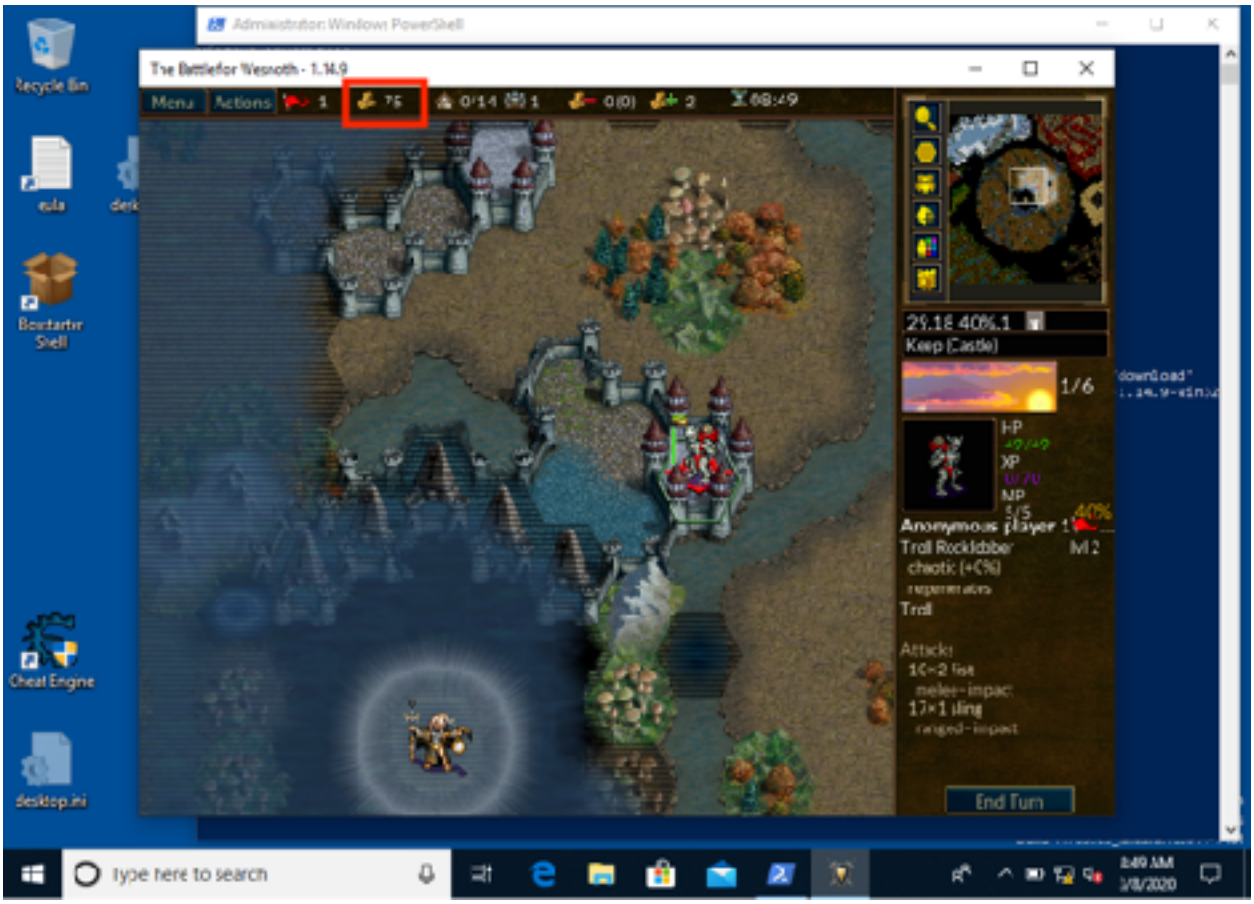


Keep the defaults for the rest of the settings and start the game.

Playrix Ex. 1024, page 30

1.5.3 Understand

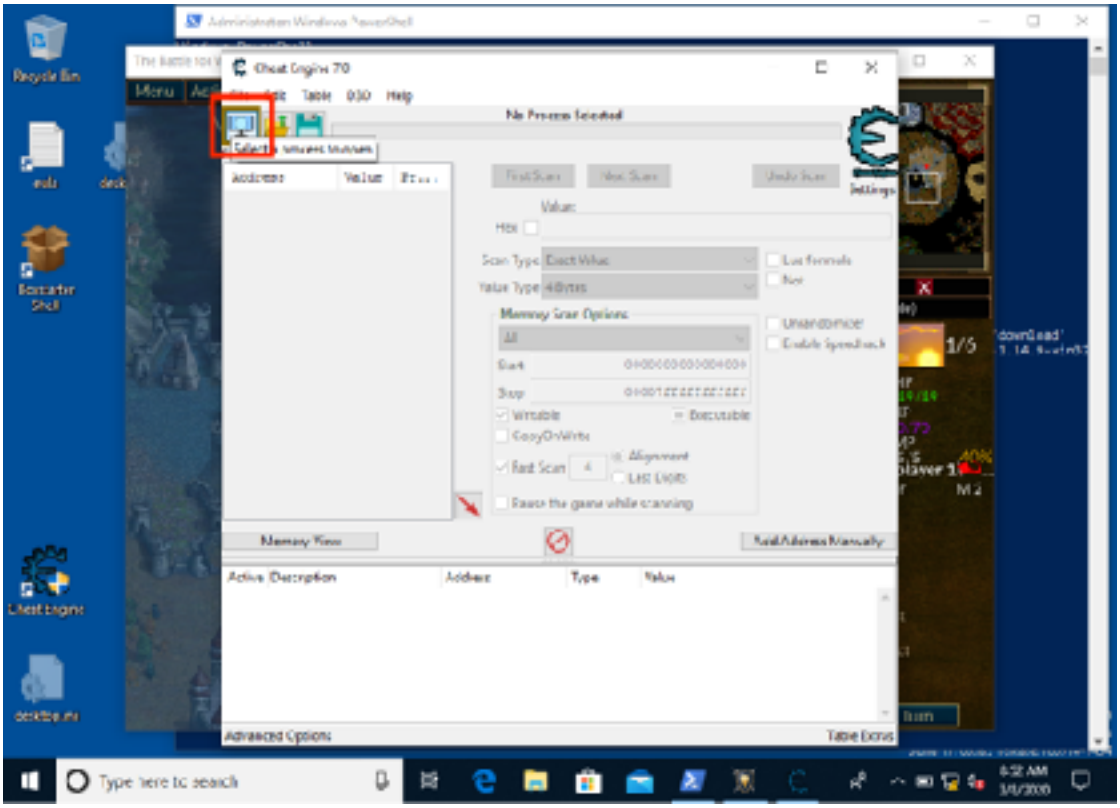
In this game, a player's gold is stored in a variable. This variable is referenced by the code of the game.



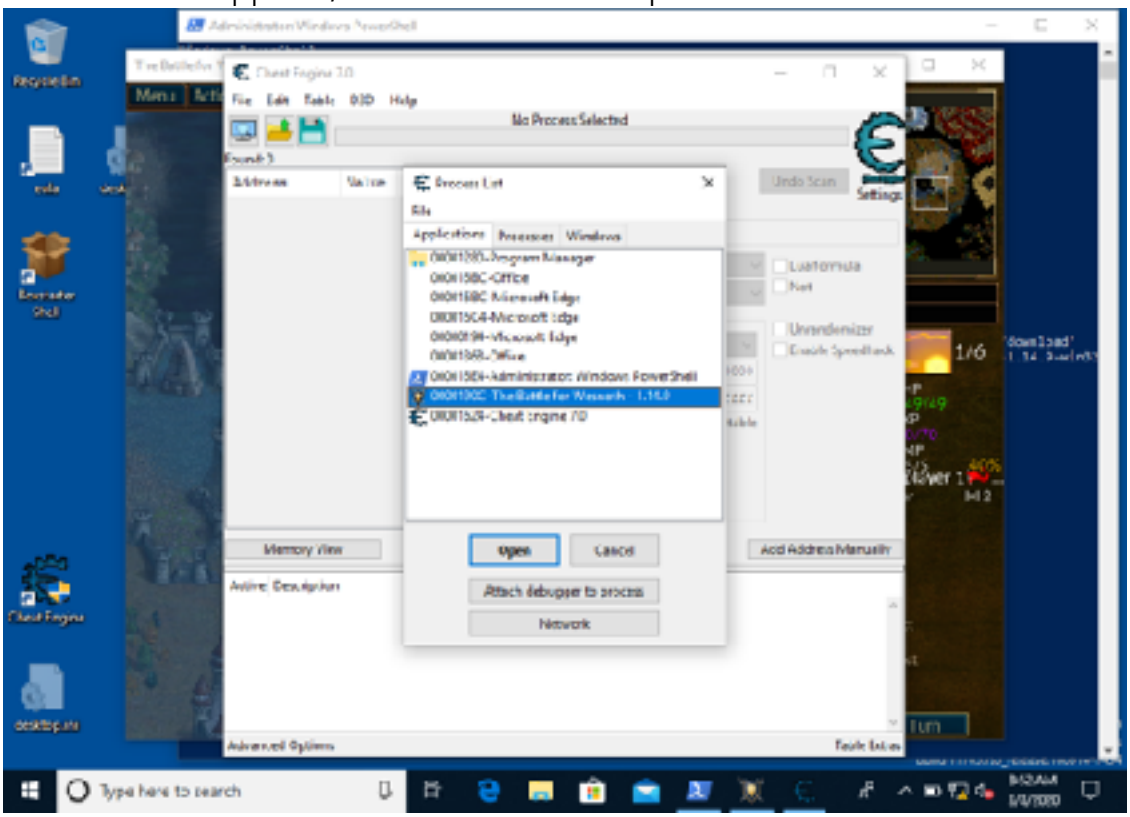
To successfully complete our hack, we will need to find where this variable is stored in memory and change its value. Since we are dealing with a variable, we will use a memory scanner to find it.

1.5.4 Locate

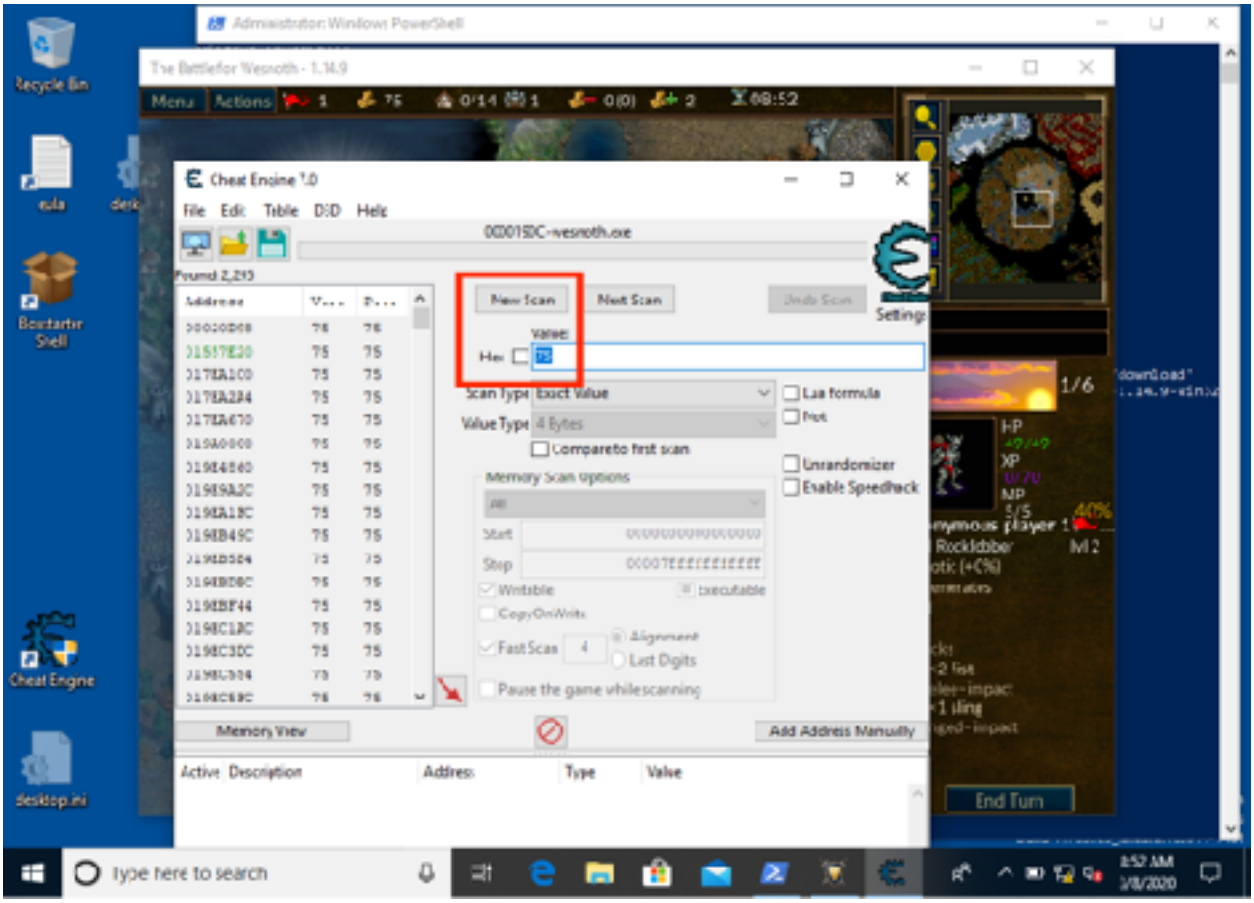
Our next step is to locate the memory holding our gold value. We will start by opening Cheat Engine and attaching it to Wesnoth. In this tool, click on the icon in the top-left that looks like a computer with a magnifying glass.



In the window that appears, choose the Wesnoth process.

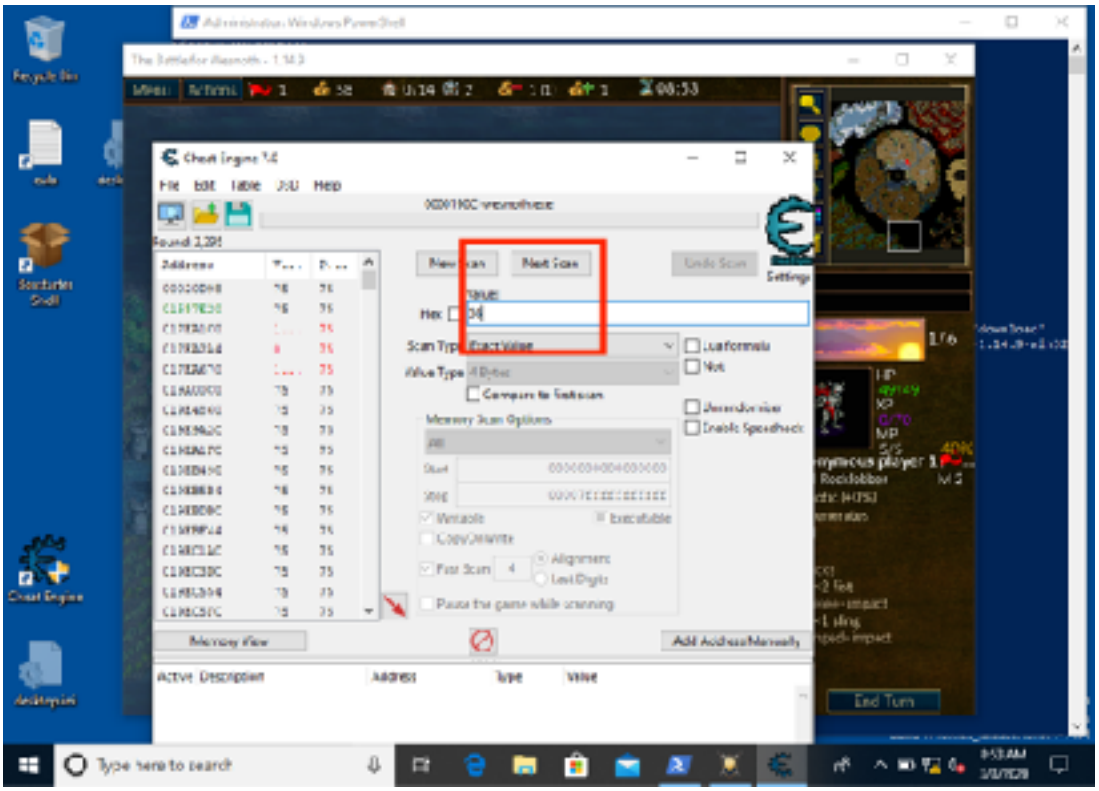


Memory scanners allow you to scan for a value in the game's memory. In the example game, the player has 75 gold, so that is what we will search for. Place your gold value in the *Value* box and select *New Scan*. Several thousand results will come back.

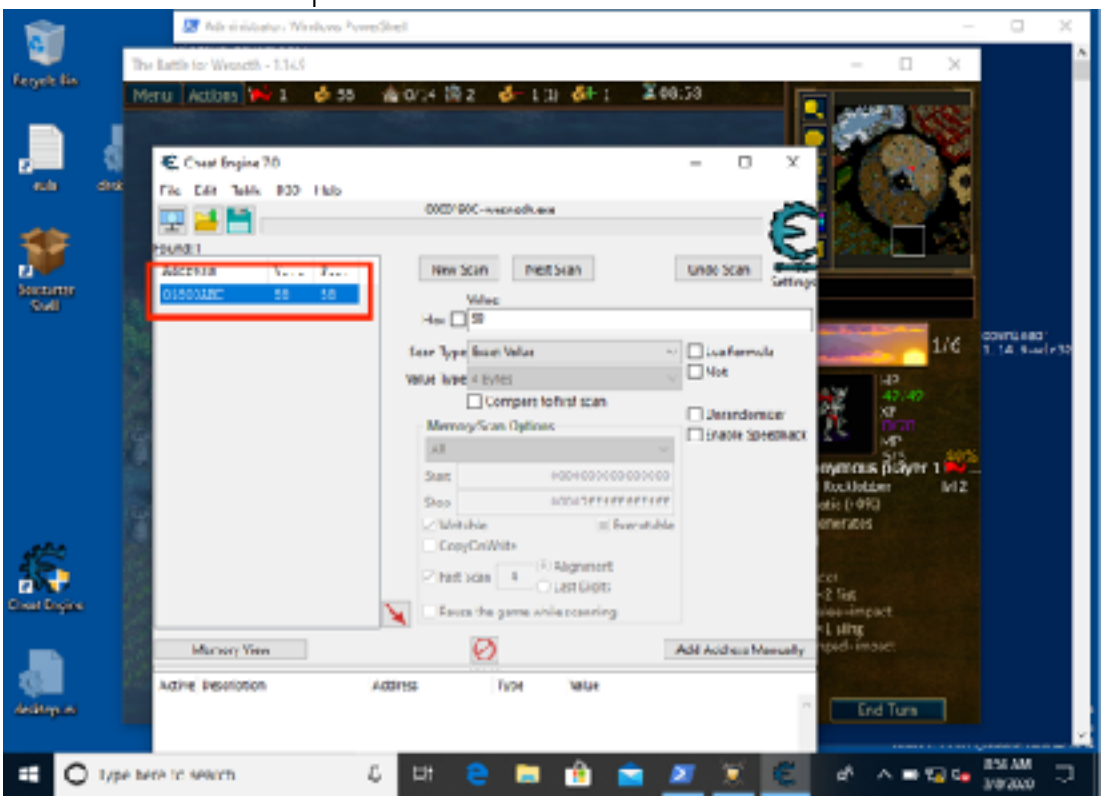


Initial scans in any memory scanner will return thousands of results. This is because games are complex and thousands of section of memory have the same value as our gold. These other sections could be variables like timers, the opponent's gold, or character health. Our goal when using a memory scanner is to filter these results down to one or two values that we can then manually test. To do this, we will modify our gold value in the game and then perform another scan using the *Next Scan* button. The next scan operation will only bring back results that were previously our initial value, in this case, *75*.

Recruit a unit of troops in the game and look at your new gold value. Place this new value in the *Value* box and select *Next Scan*. In the example below, we recruited a unit for 17 gold, leaving us with *58* gold.

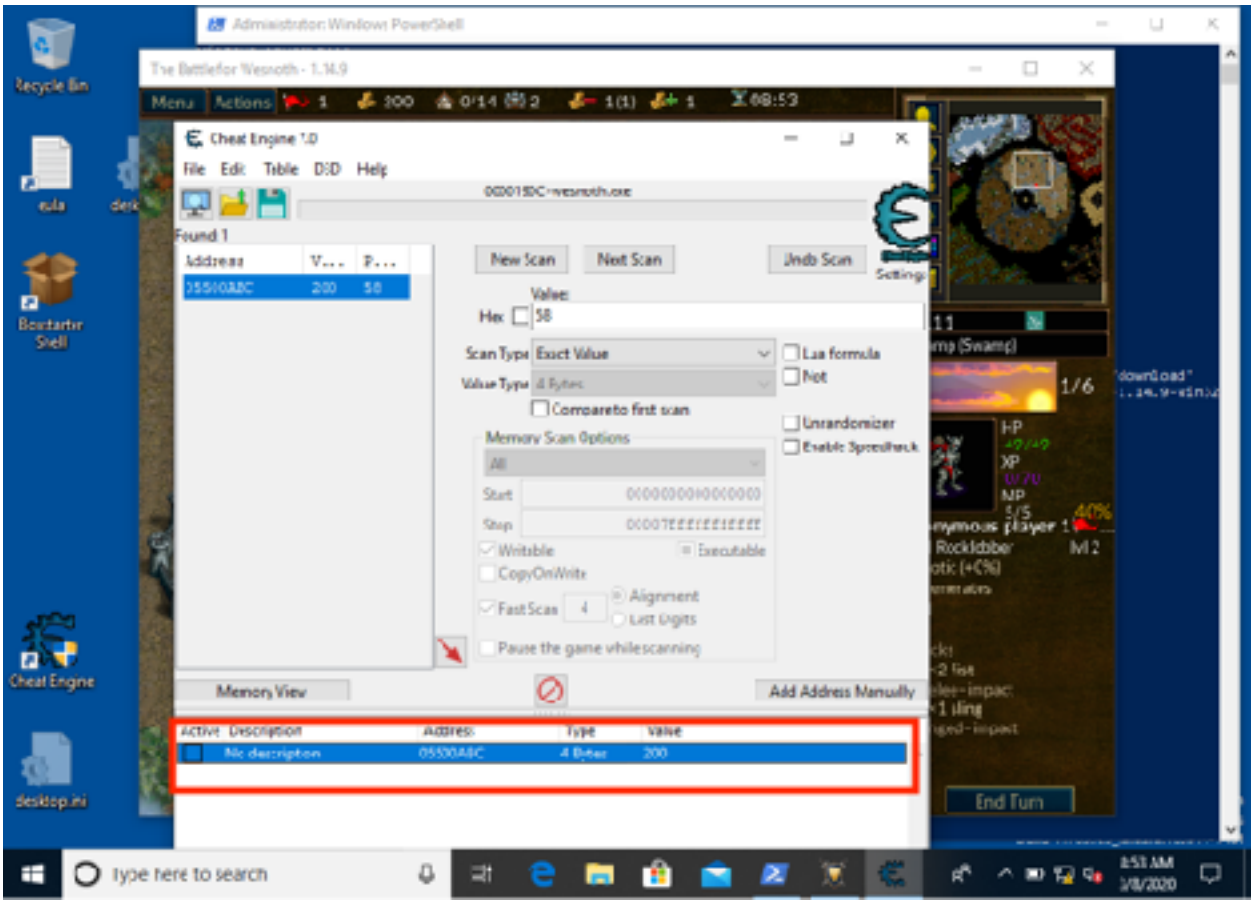


The result of this second scan is only one value. This is most likely our gold, but we will confirm that in the next step.



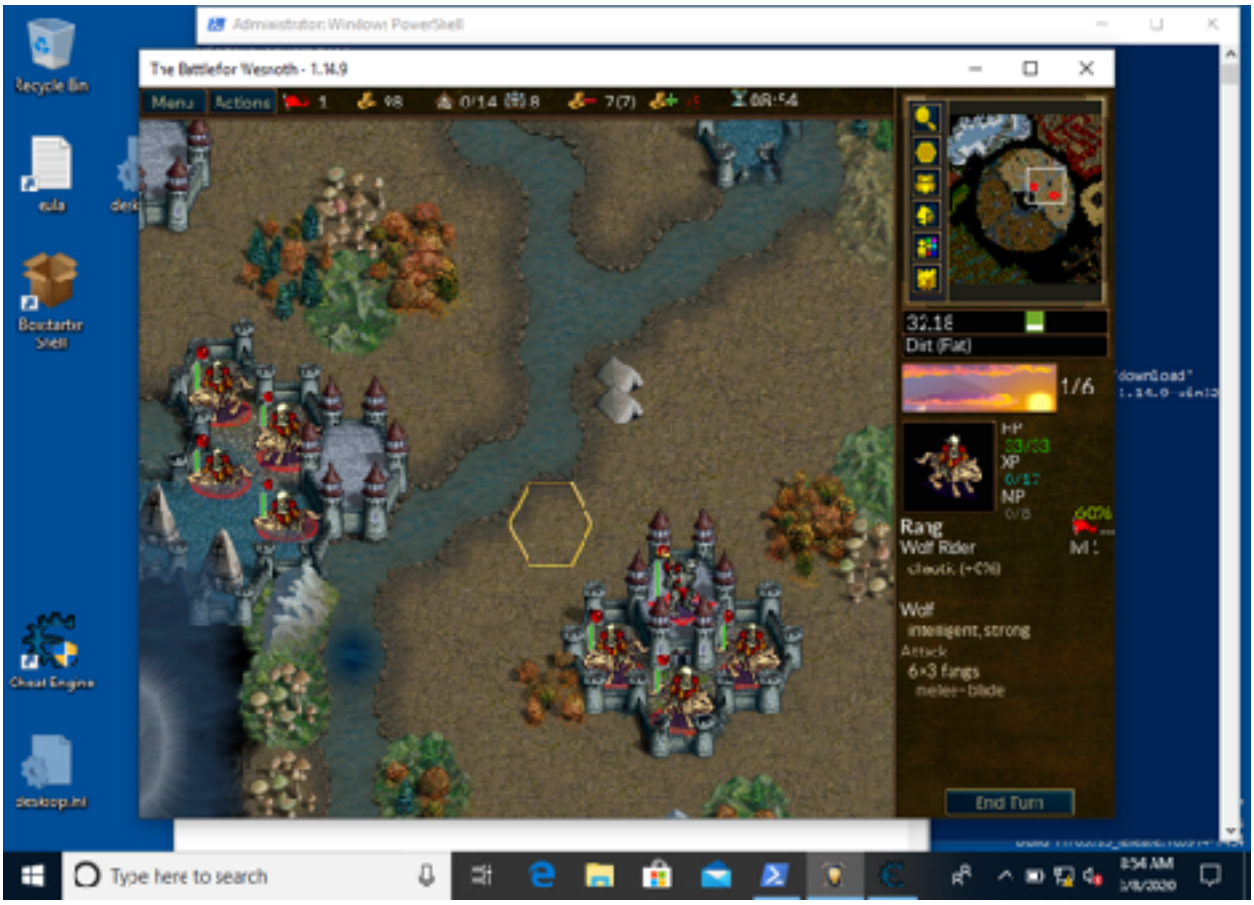
1.5.5 Change

In Cheat Engine, you double-click on a memory location to move it to the bottom box on the screen. This bottom block allows you to edit the value stored in the memory location. Double-click on your result to bring it to the bottom box. Next, double-click on the value to bring up a box that will ask you for the new value. Type something large in there, like *200*.



With the value changed, go back to the game. You should see your gold refresh to 200. Recruit a ton of units to confirm that your change was successful.

Playrix Ex. 1024, page 35


Recycle Bin
eula
Bootstrap Shell
Cheat Engine
desktop.ini
The Battle for Wesnoth - 1.14.9
Menu
Actions
End Turn
Type here to search

Part 2
Debugging & Reversing

Playrix Ex. 1024, page 37

2.1 Debugging Fundamentals

2.1.1 Goals

In our previous chapter, we hacked our gold by modifying a variable in memory. Memory modification like this is powerful, but it also has limitations. More complex hacks will often require you to modify the game's code. For example, imagine if we wanted to create a hack that would allow us to recruit units for no gold. One way to do this would be to constantly monitor our gold and manually increase it whenever we recruited a unit. This would also require you to constantly look for any new units added to the game and that unit's cost. An easier approach would be to modify the game's code to never decrease a player's money when recruiting.

Viewing a game's code as it is running is known as debugging. Understanding and modifying that code to do what you want is known as reversing. You do not have to debug a game to reverse it, but it is very helpful if you can.

2.1.2 Tools Involved

To debug a game, you use a tool known as a debugger. The first step in debugging is "attaching" the debugger to the game you want to debug. Once it’s attached, you are able to view the game's code in memory. You are also able to pause execution of the game, change the game's code, and modify registers. We will see examples of these actions in future chapters.

Debuggers can cause unintended side-effects. For example, if you change the game's code incorrectly, the game can crash. Depending on the game, this can freeze your computer's display. This is another reason to always separate your hacking machine from your personal machine.

There are many debuggers, but some well-known ones include IDA and gdb. Other debuggers, like WinDbg and OllyDbg, are often mentioned but no longer maintained. In this series, we will be using an open-source debugger named x64dbg. Like any other

Playrix Ex. 1024, page 38

tool, it's more important to know the fundamentals than the tool. The same approach we will learn while using x64dbg can be applied to any debugger.

2.1.3 Disassembly and Debugging

After attaching a debugger to a game, the debugger will display the game's code. However, this is not the original game's code. Like we discussed in the first chapter, games are usually programmed in a high-level language, like C++. However, the executable running on our computer only contains the opcodes for the CPU to execute. This lack of the original code is what makes reversing difficult. Often games will contain thousands of these opcodes.

When assembling a program, each line of assembly code is converted to an opcode. Disassembly is the process in which opcodes are converted back to assembly. Normally disassembly and debugging are used interchangeably, especially when reversing a game. However, they can be done separately. It is possible to disassemble a program without debugging it. This is known as static analysis and is commonly done when reversing malware. It is also possible to debug a program without disassembling it. A common example of this would be debugging a program that you have written. In this case, you have the original code and the disassembly is unneeded.

It is possible to partially recreate the original high-level code from the disassembly. This is known as decompiling. However, disassembly is always representative of the code executing, while decompiling is not. Decompilers are often forced to guess at the original structure of the code. While these tools can be helpful, they can also lead you down false paths. In this book, we will not cover decompiling.

2.1.4 Assembly

When debugging and reversing a game, you will mainly be dealing with assembly. Assembly is similar to the first language we covered in Chapter 1.1. Each instruction in assembly does one thing, such as add or subtract. It is not necessary to know every assembly instruction to reverse a game.

While it may appear daunting, any assembly code can be understood by going through it one instruction at a time. When debugging, this is known as stepping through the disassembly. Often there are many instructions in a game that are not critical to understand while reversing. For example, the CPU has to do several instructions when adding numbers that have decimal values. If we are only interested in

the result of this addition, we can skip over many of these instructions. Understanding which instructions can be skipped comes with experience.

Playrix Ex. 1024, page 40

2.2 Reversing Fundamentals

2.2.1 Context

The process of reversing a game can seem overwhelming the first time you attach a debugger to a game. The best way to start reversing a game is to figure out what you want to look at and then find where it is. Once you establish this context, you can step through only those instructions that actually matter to you.

There are many ways to establish a context. In some cases, you may want to search for text that is displayed when the game does a certain action. Any locations that load this text must eventually be related to the action that you are interested in. In other cases, you can use memory addresses found in the memory editor to find the code that you are interested in. Regardless of which approach you take, you will use a breakpoint.

2.2.2 Breakpoints

Breakpoints allow the debugger to pause execution of the game at a specific instruction. With the game paused, you can then step through individual instructions and view the game's memory. You can set breakpoints on any type of memory. This includes memory found using a memory scanner.

Breakpoints can be set to trigger both non-conditionally and conditionally. Conditional breakpoints will only trigger if their conditions are met. These conditions can be things like registers having a certain value or the memory (that the breakpoint is set on) changing. When a breakpoint is triggered, it's also known as popping.

2.2.3 Memory Breakpoints

The best way to illustrate the use of breakpoints is through an example. In this section, we will examine how a memory breakpoint can be used to establish a context.

Playrix Ex. 1024, page 41

Back in Chapter 1.5, we found the memory location of our gold. We can use this memory location to find the game logic responsible for lowering our gold. We do this by setting a conditional breakpoint on the memory location of our gold and then going into the game to recruit a unit. When we recruit the unit, the breakpoint will pop and execution will be paused at the function responsible for lowering the player's gold. This function may look something like below, with the line in blue representing our paused location.

```
mov eax, dword ptr ds:[0x05500ABC]
mov ebx, dword ptr ds:[0x12345678]
sub eax, ebx
mov dword ptr ds:[0x05500ABC], eax
mov esi, ebx
```

The first instruction moves the value stored at `0x05500ABC` into the register **eax**. This value was the location we found in our previous chapter for gold. The next instruction moves a hypothetical value for the unit's cost into the register **ebx**. The game then subtracts the unit's cost from our gold value. Our paused location is responsible for moving the new value of gold back into the memory location that stores our gold value.

You may notice that the game did not pause on the subtraction operation. This is because this operation only modifies the value in the register and not the actual value of the memory we set the breakpoint on. Breakpoints will always pause on the instruction immediately after the affected memory.

2.2.4 Code Breakpoints

Sometimes it may be difficult or impossible to find a memory value to set a breakpoint on. In these cases, you can set a breakpoint on a section of code. A common example of this is setting a breakpoint on a text reference and then using that to find the top-level function we are interested in.

Consider a game for which we want to write a wallhack. The game's main loop may look something like:

```
void main_loop(){
    draw_players();
    draw_walls();
    ...
```


Playrix Ex. 1024, page 42

```
}
```

And the game's **draw_wall** function may look something like:

```
void draw_wall(){
    bool succeeded = load_texture("wall_texture");
    if(succeeded == false){
        print_error();
    }
}
```

Finally, the **print_error** function may look something like:

```
void print_error(){
    print_to_log("Couldn't find wall texture");
}
```

One method of writing a wallhack is to remove this game's **draw_wall** function. Since there is no variable to use as a memory breakpoint, we will instead use a code breakpoint.

Debuggers allow you to view all the text in a game and all the locations that use that text. For example, with a debugger, we could find the *Couldn't find wall texture* text and where it is referenced. It may look something like:

```
mov eax, dword ptr ds:[0x23456789]
push eax
call print_to_log
…
```

This section of code is responsible for loading the string into a register and then calling the **print_to_log** function. By setting a breakpoint on this code and then finding a missing texture in the game, our breakpoint would pop. We could then continue to step through the code until it returned us to the function that called this code. This is known as stepping out of a function. After we have stepped out, we would be in the **draw_wall** function and could then remove the function.

2.2.5 The nop Instruction

A **nop** (opcode `0x90`) stands for no operation. When encountering this instruction, a CPU will do nothing and continue on to the next instruction. This behavior can be used to modify game logic.

For example, in Section 2.3.3 above, we found the portion of code responsible for subtracting our gold. The code looked like:

```
mov eax, dword ptr ds:[0x05500ABC]
mov ebx, dword ptr ds:[0x12345678]
sub eax, ebx
mov dword ptr ds:[0x05500ABC], eax
```

By replacing the **sub** operation with a **nop**, the game will no longer subtract our gold.

2.3 Changing Game Code

2.3.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

2.3.2 Identify

Our goal in this chapter is to change Wesnoth's code so that recruiting units does not decrease our gold.

2.3.3 Understand

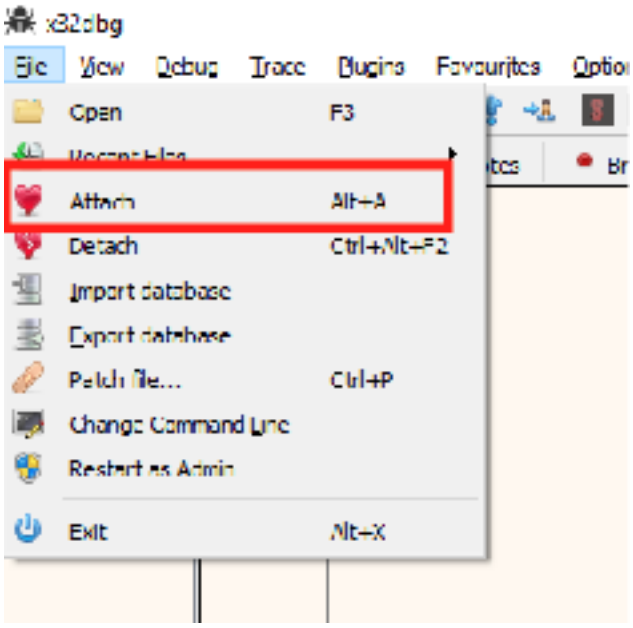
To modify the game's code, we will need to use a debugger. To locate the game code to modify, we will set a breakpoint on our gold address and then recruit a unit. Our debugger will pop at the code responsible for decreasing our gold. We can then **nop** out the **sub** instruction.

2.3.4 Locating Gold

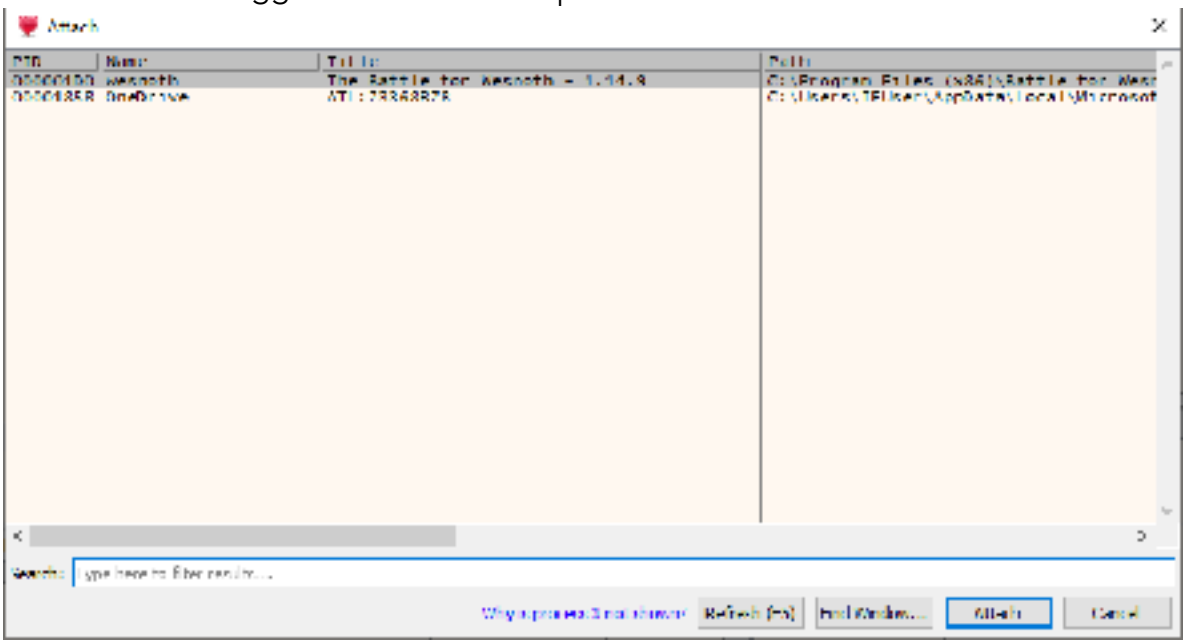
Our first step is opening up Wesnoth and creating a local game. Then you can follow the steps in Chapter 1.5 to find your gold address. Due to a process called Dynamic Memory Allocation (or DMA), it will be at a different address than before. We will cover DMA in a future chapter. Once you have found your new gold address, close down Cheat Engine but keep Wesnoth open. In this chapter, we will use the value of `0x051D875C` as our gold address.

2.3.5 Attaching the Debugger

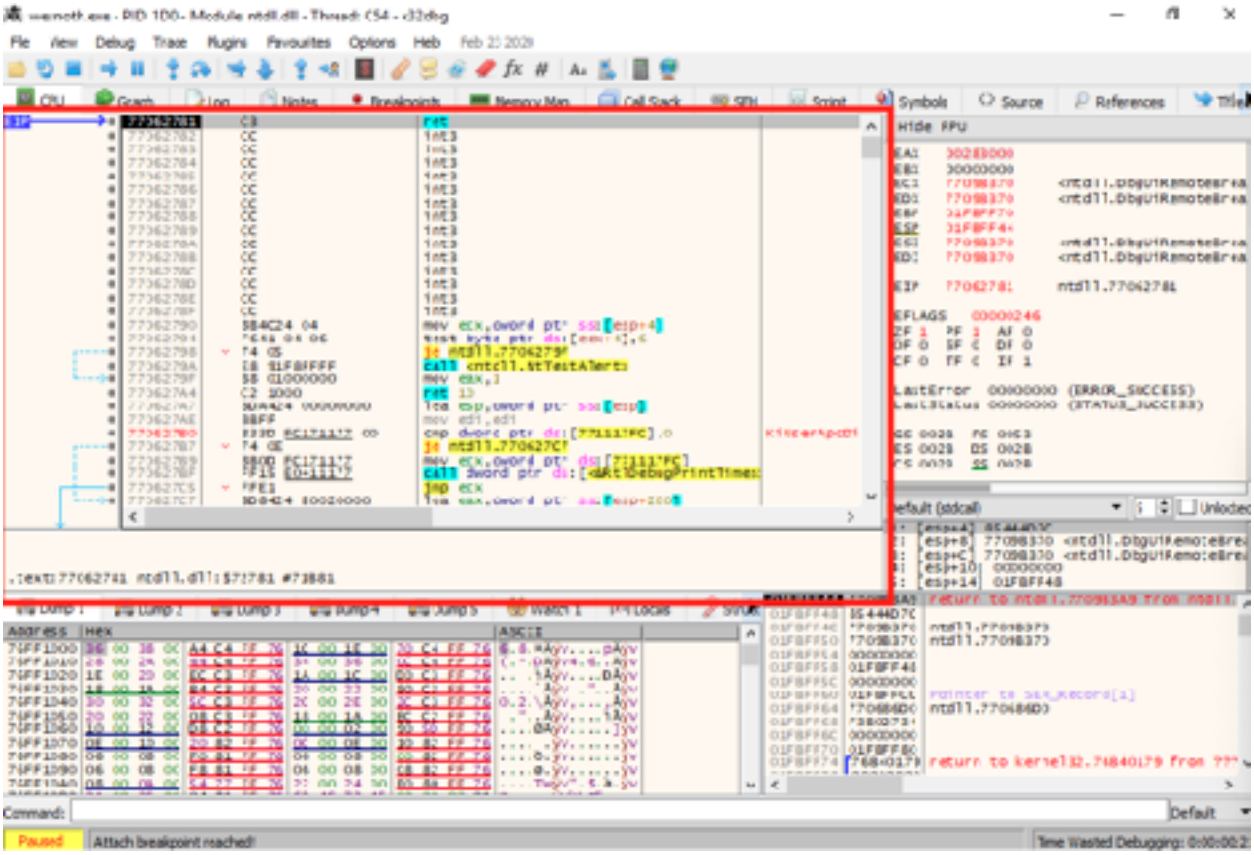
Next, start the 32-bit version of x64dbg. It can be found at *C:\ProgramData\chocolatey\lib\x64dbg.portable\tools\release\x32\x32dbg.exe*. Once started, open the *File* menu and choose *Attach*.



In the attach dialog that opens up, choose the Wesnoth process and hit *Attach*. This will attach our debugger to the Wesnoth process.



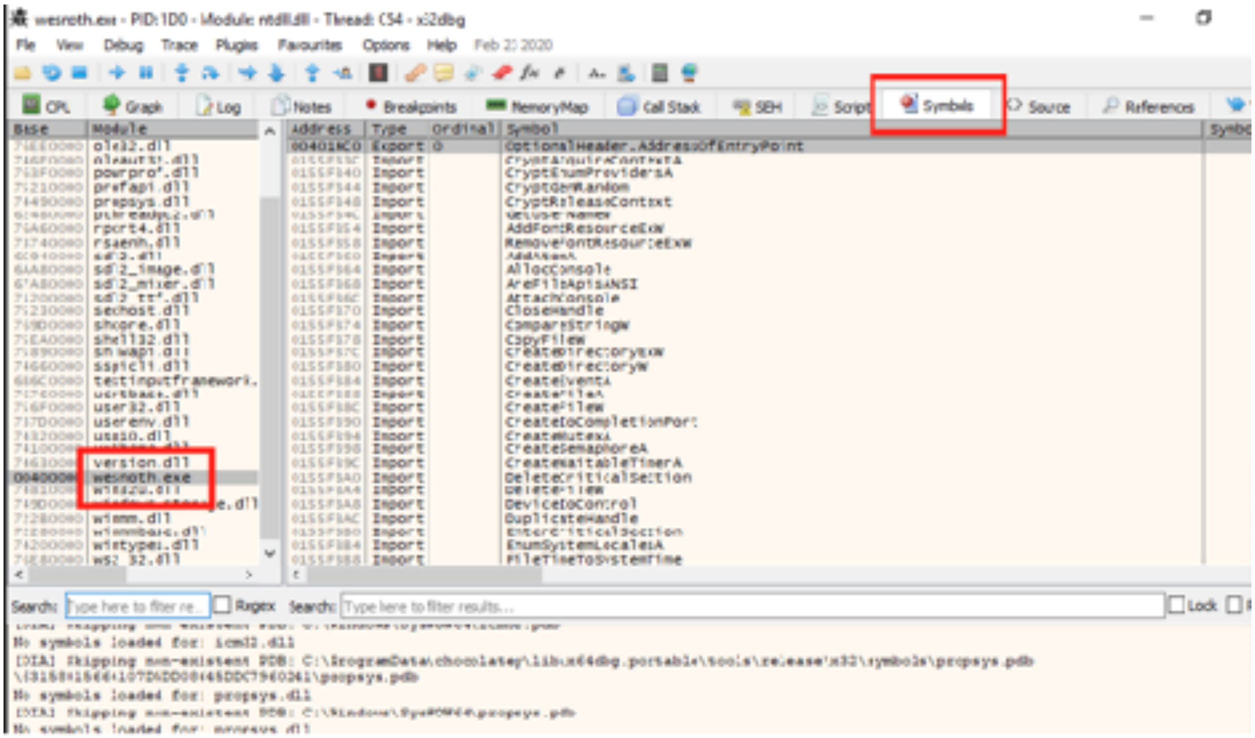
Upon attaching, x64dbg will pop in a module called *ntdll.dll* and display a lot of information.



Before we dive into reversing, let's quickly cover the major components in any debugger. The highlighted section contains the code being executed. The dump section directly below displays the memory of the application in its hexadecimal (hex) and ASCII representation. To the right of the code section is a list of all registers and their values. Below the registers is the application's stack.

In this chapter, we will only be using the code section and the dump section. For more complex hacks, understanding all these sections will be necessary. Different debuggers will always contain all of this information, but they will often arrange them in different ways.

We are currently viewing the ntdll.dll module. This is not our target, but it's a common module loaded into all Windows executables. To view the game's code, we need to navigate to the *Symbols* tab and double-click on *wesnoth.exe*.

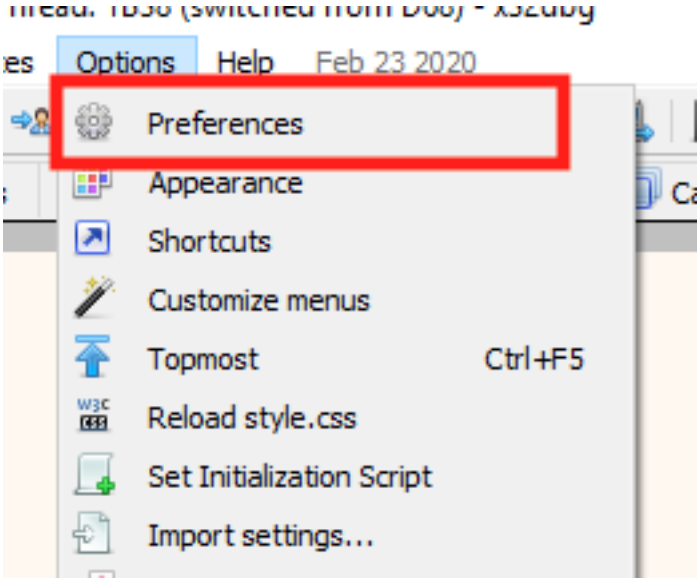


This will switch our view to the game's code and memory space.

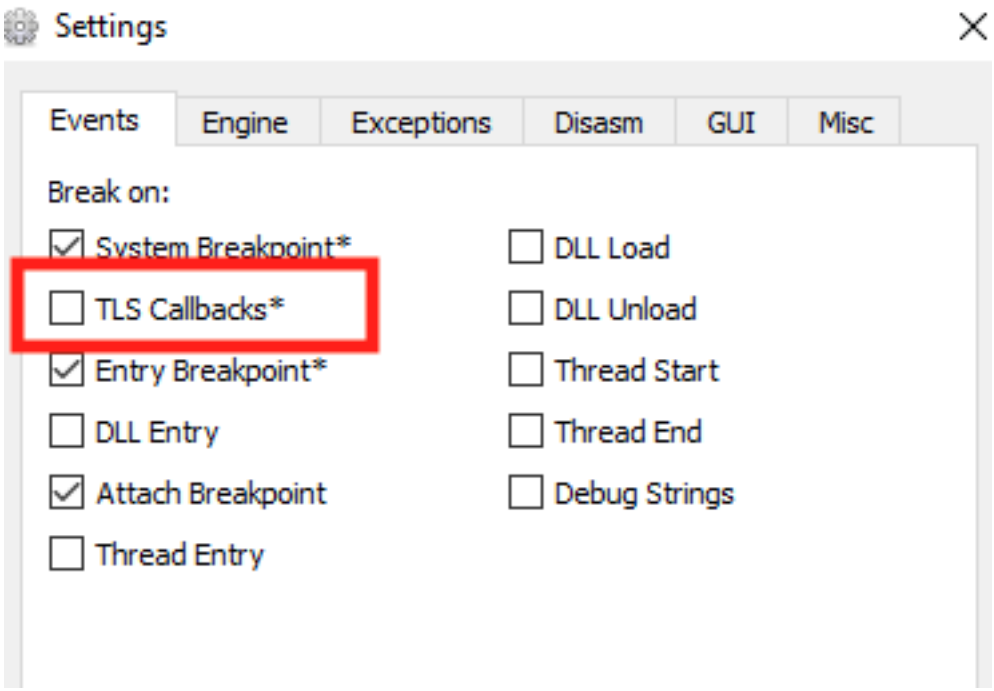
2.3.6 Setting Up the Debugger

Certain default settings in x64dbg will make it pop when we do not want it to. To make reversing in this chapter and future chapters easier, we will disable these settings.

In the top menu, choose *Options* -> *Preferences*.



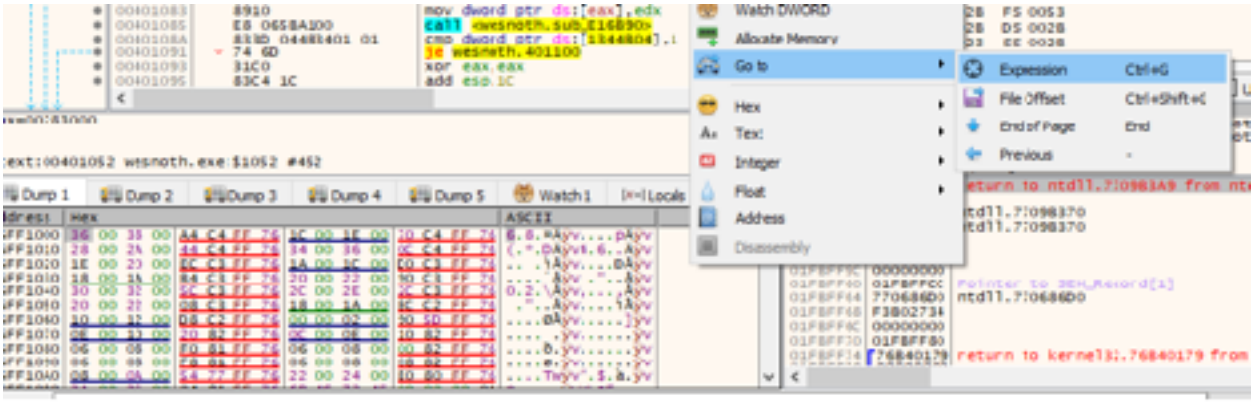
In the modal that opens, uncheck the *TLS Callbacks* option and select *Save*. This will disable x64dbg from automatically popping when receiving a TLS callback.



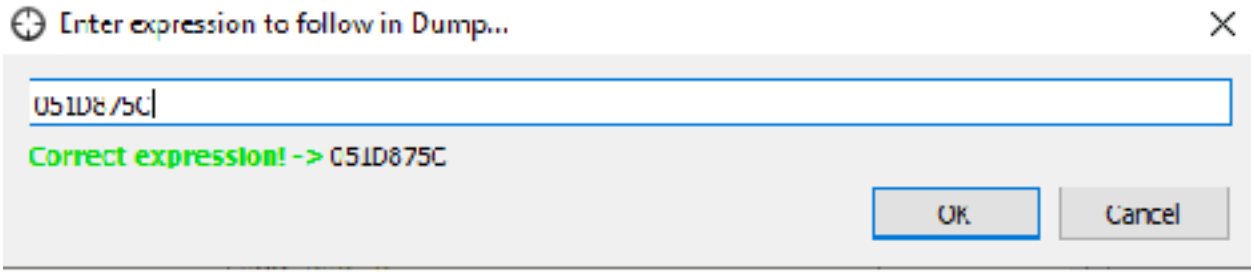
2.3.7 Setting a Breakpoint

Next, we will set a breakpoint on our gold location. After we set this breakpoint, we will go into Wesnoth and recruit a unit. Doing so will cause the breakpoint to pop and pause execution at the location responsible for subtracting gold.

Right-click in the dump section and choose *Go to -> Expression*:

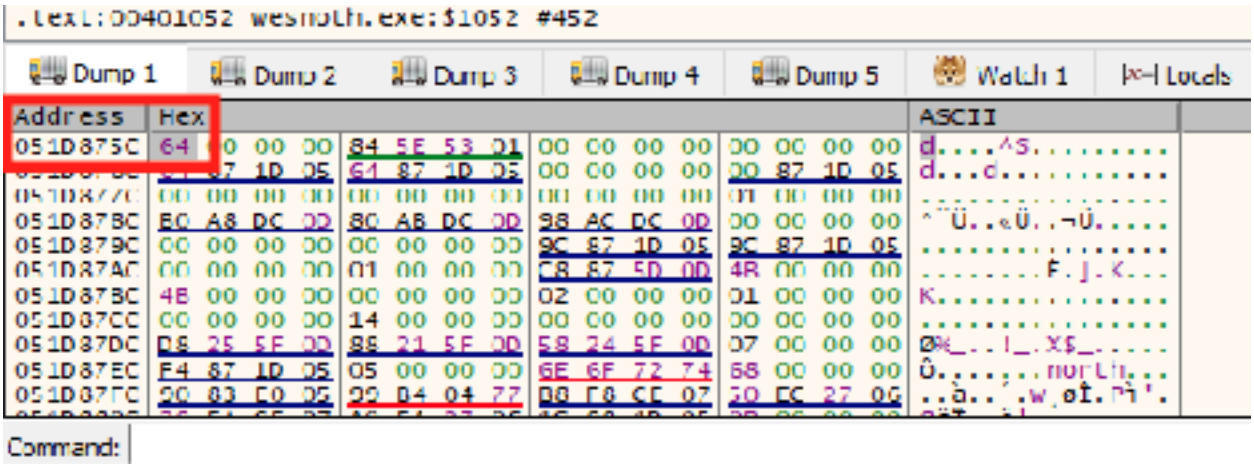


In the dialog that opens, type in our gold address and hit *OK*.

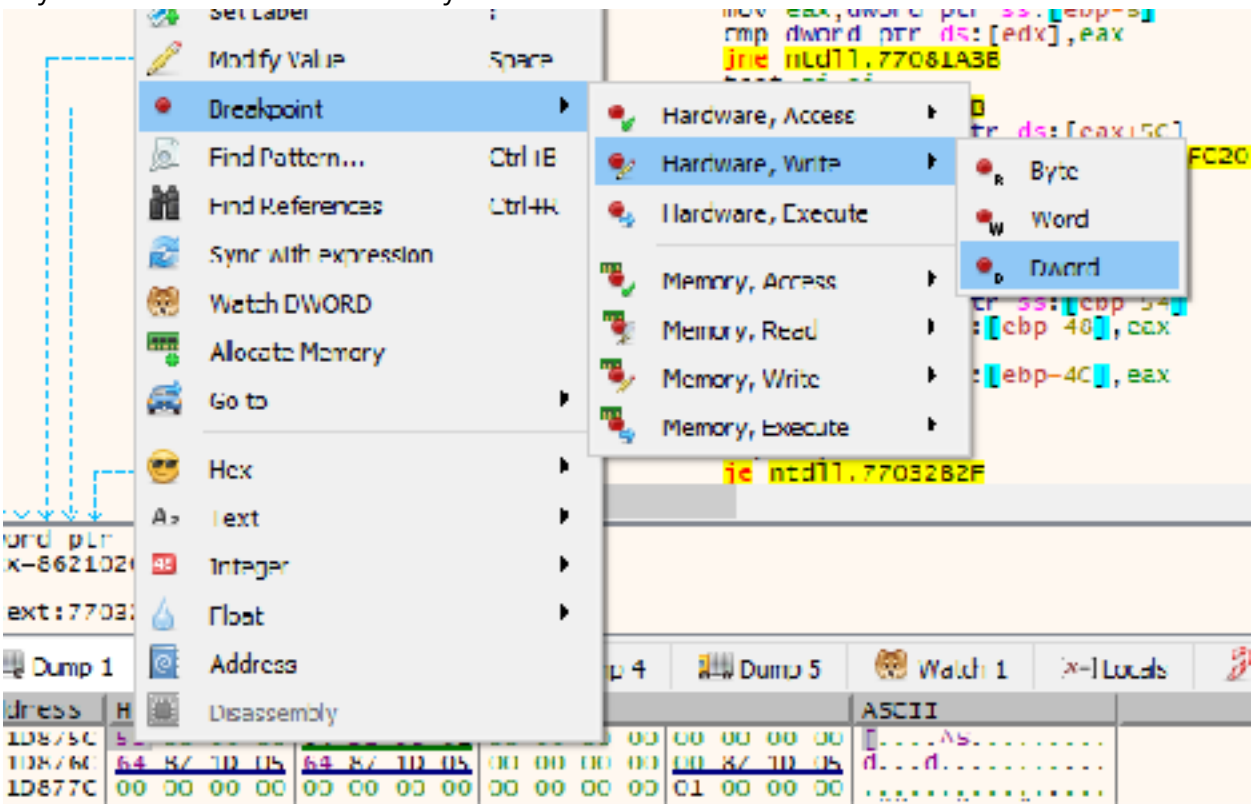


The dump will then show the address we just typed in. The data displayed is in hexadecimal format. In the target game, the player had 100 gold. 100 in hexadecimal format is `0x64`. This is the value displayed in the dump.

Playrix Ex. 1024, page 50

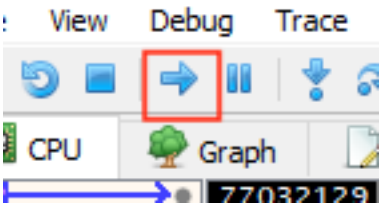


Right-click on the value and choose *Breakpoint -> Hardware, Write -> DWORD*. This will set a conditional breakpoint on this memory address. The condition for popping is any modification of the memory address.



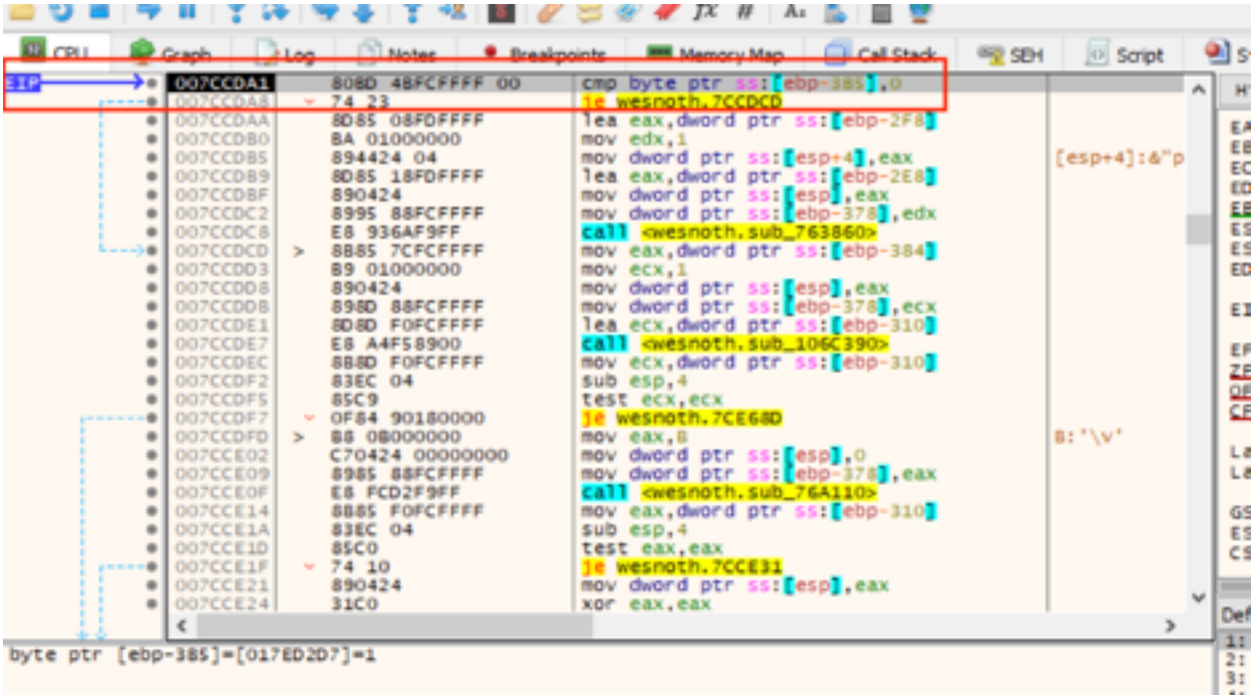
With our breakpoint set, we will now resume execution of the program. This can be done by pressing the *Play* button until the *Paused* status disappears and the game resumes. You will have to do this several times due to the several breakpoints that x64dbg automatically creates.

Playrix Ex. 1024, page 51



2.3.8 Locating Code

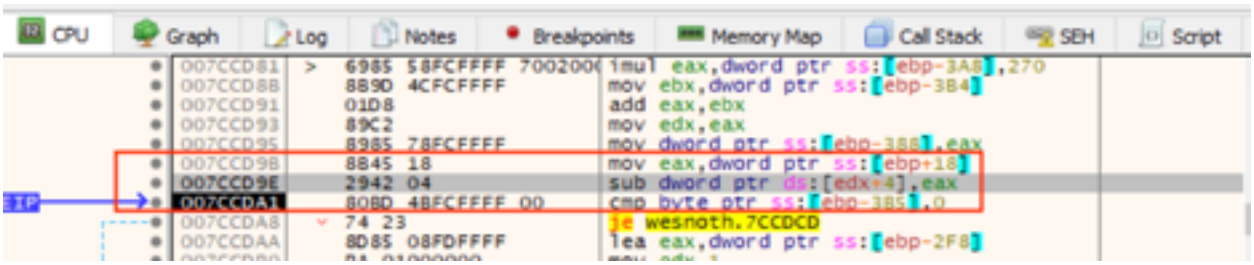
Once the game resumes, go back into Wesnoth and recruit a unit. You will notice that Wesnoth will freeze due to our breakpoint popping and pausing execution. Navigate back to x64dbg to see where it popped.



The highlighted **EIP** represents the current location of execution within the program. **EIP** stands for Extended Instruction Pointer and is a special register used by programs to understand the current execution location.

Playrix Ex. 1024, page 52

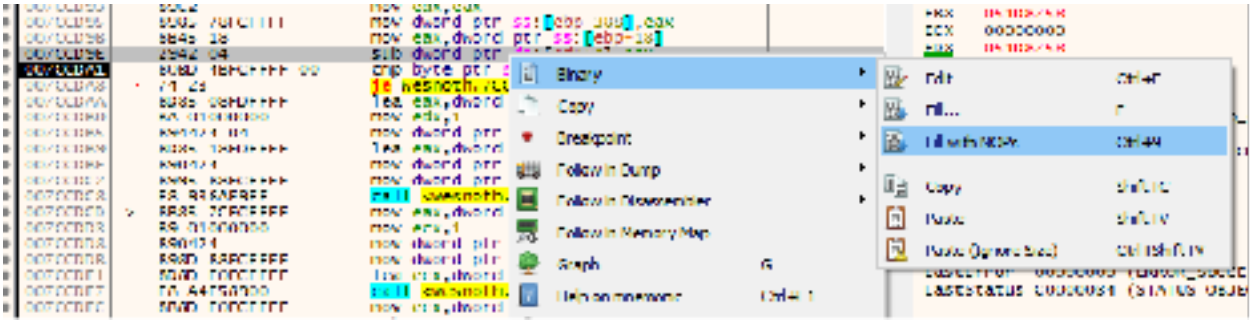
From our last chapter, we know that conditional breakpoints are triggered after the operation that affected the memory in question. Scroll up in the code window to see the previous instructions.



The highlighted **sub** instruction was responsible for modifying our gold value. As we remember from previous chapters, **sub** stands for subtract and is responsible for subtracting two numbers. In this case, it is subtracting the value held in the memory location stored in **edx + 4** and **eax**. The exact specifics of these values are not necessary to know now. All we need to know is that this operation is affecting our gold in some way.

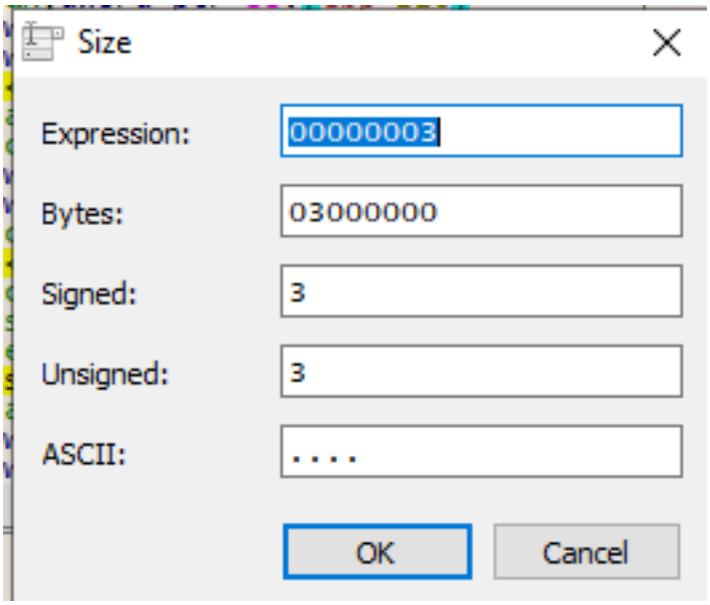
2.3.9 Change

Finally, we will change this code and finish our hack. To do this, we will replace the **sub** instruction with the **nop** instruction. This will replace the subtraction with an operation that does nothing. As a result, our gold will no longer decrease. Luckily, x64dbg contains a built-in way to automatically **nop** out an instruction. Right-click on the line with the subtract instruction and choose *Binary -> Fill with NOPs*.

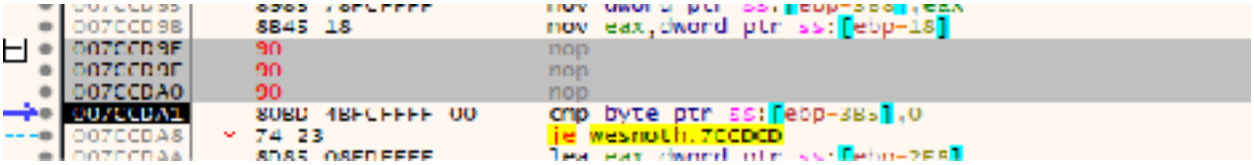


Playrix Ex. 1024, page 53

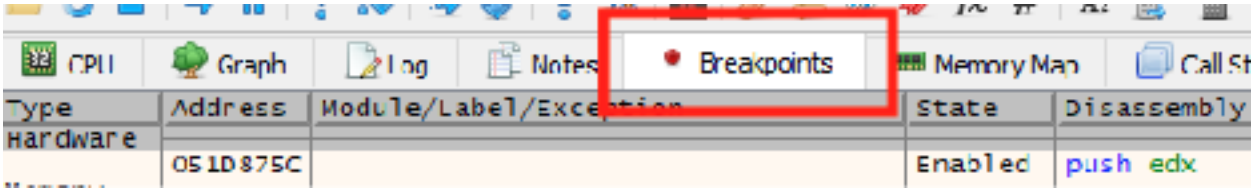
x64dbg will populate the next values for you automatically. Just select *OK* on the next screen.



If done correctly, the code should now look like the image below. You will notice there are three **nop** instructions. We will cover why in a future chapter.

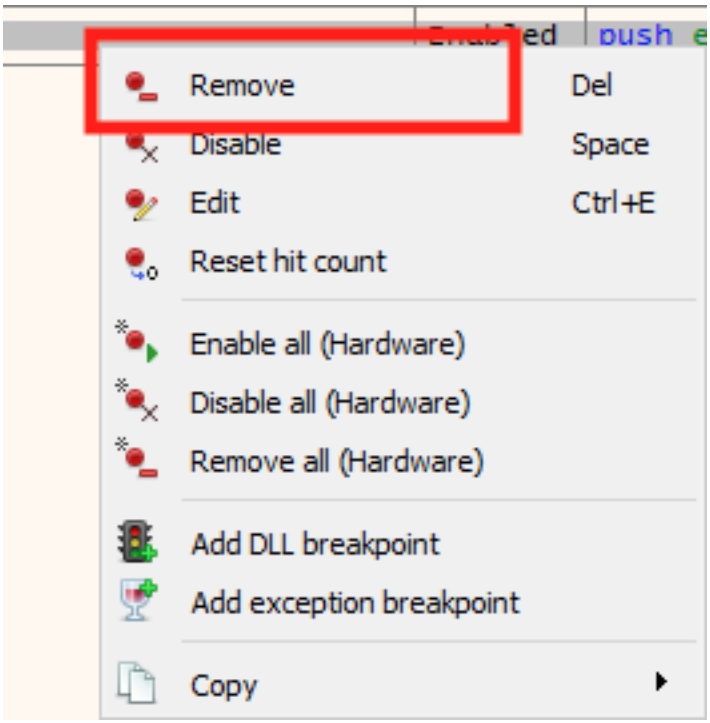


Before we can verify that our change has worked, we need to disable our breakpoint so that it doesn't pop again. To do this, first go to the *Breakpoints* tab. This tab contains a list of all the breakpoints we have set in the application.



Playrix Ex. 1024, page 54

Right-click on the breakpoint you have set and choose *Remove*.



With the breakpoint removed and the code changed, we can now go back into Wesnoth and observe our changes. Recruit a few units and observe that your gold no longer goes down.

Playrix Ex. 1024, page 55

2.4 Reversing Code

2.4.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

2.4.2 Identify

To recruit units in Wesnoth, you right-click on a tile and choose *Recruit*. In Wesnoth, you can only recruit units on specific tiles. Our goal for this chapter is to change this behavior so that we can recruit units anywhere on the map.



2.4.3 Understand

This hack will require us to modify the game's code using a debugger. To conduct this hack, we will first need to find the code executed when right-clicking on a tile and choosing an option. The original game code probably looks something like:

```
switch( option_selected ) {
    case "Terrain Description":
        show_terrain_description(location);
        break;
    case "Recruit":
        recruit_unit(location);
        break;
    case ...
}
```

A switch statement allows you to execute different branches depending on the state of a variable. We want to modify this statement so that clicking on *Terrain Description* instead calls the code for recruiting a unit.

2.4.4 Bubbling

In the previous chapter, we found the code responsible for subtracting gold when we recruited a unit. We will use this code to bubble up to the right-click menu code location. To illustrate the concept of bubbling up, imagine that the code in Wesnoth for recruiting units looks like:

```
function handle_context_menu() {
    ...
    case "Recruit":
        recruit_unit(location);
        break;
    ...
}
...
function recruit_unit(location) {
    ...
    check_location(location);
    find_unit_in_unit_list();
    ...
```

```
}
...
function find_unit_in_unit_list() {
    ...
    get_unit();
    get_unit_cost();
    subtract_unit_cost();
    ...
}
...
function subtract_unit_cost() {
    ...
    check_player_gold();
    subtract_gold();
    ...
}
...
function subtract_gold() {
    player_money = player_money - cost_of_unit;
}
```

A good way to visualize the interactions between all these functions is through the use of a function chain. The function chain for this example would look like:

```
handle_context_menu() -> recruit_unit() -> find_unit_in_unit_list() ->
subtract_unit_cost() -> subtract_gold()
```

The code we found in the previous chapter was in the **subtract_gold** function. By bubbling up from this code, we will eventually locate the **handle_context_menu** function.

To bubble up in our debugger, we will make use of two features: *Execute till return* and *Step Over*. The execute till return feature executes instructions until reaching a return statement. The step over feature executes a line of code. Unlike the *Step Into* feature, step over does not enter a function if the instruction being executed is a **call**. We will elaborate on this later, but first we need to cover how functions are translated into assembly.

2.4.5 Calls and Returns

The **call** instruction is used to invoke a function in assembly. At the end of the called function, the **retn** (return) instruction is used to go back to the code that called the function. For example, the code below uses a **call** to increase the register **eax** by 1:

```
main:
    mov eax, 0
    call increase_eax
    mov ebx, eax

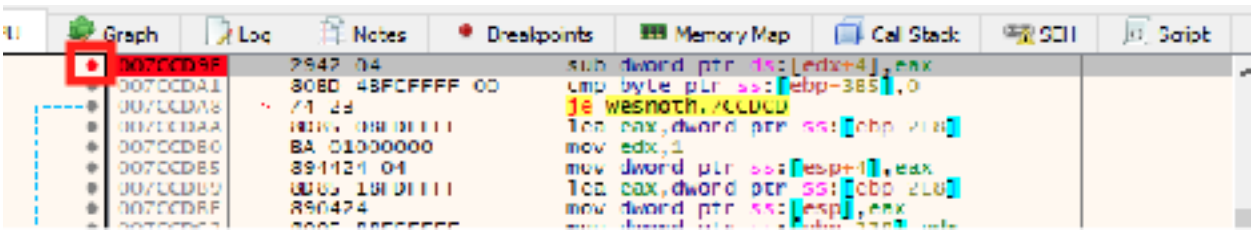
increase_eax:
    add eax, 1
    mov ecx, eax
    retn
```

Imagine we set a breakpoint on the **add eax, 1** instruction. Once it pops, using the execute till return feature would cause the debugger to continue executing code until the first **retn** instruction is reached. Once on the **retn** instruction, the step over feature would then execute the **retn** instruction and arrive at the **mov ebx, eax** instruction. This is a good illustration of bubbling up to a higher function.

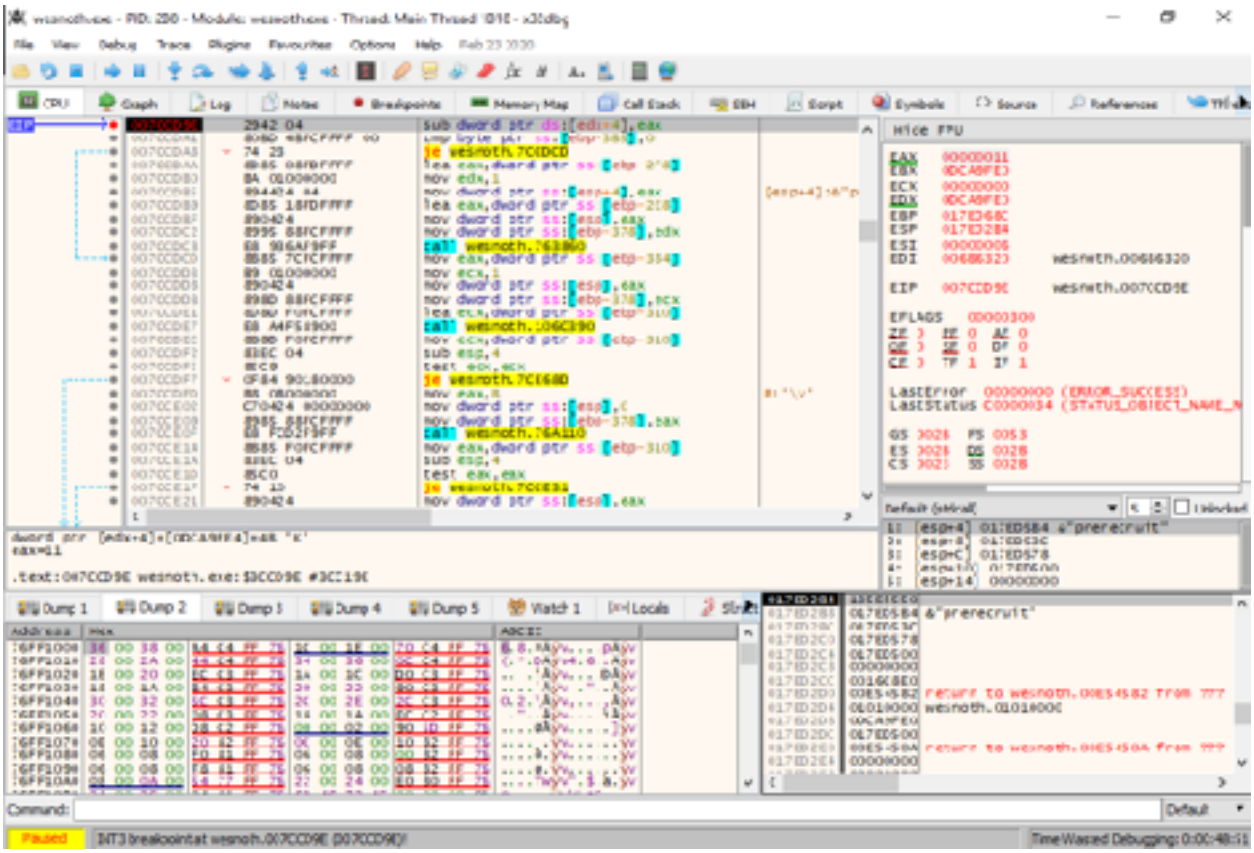
To understand stepping in versus stepping over, imagine we set a breakpoint on the **call increase_eax** instruction. Stepping into this instruction would cause our debugger to go to the first line of the function (**add eax, 1**) and wait there. Stepping over this function would cause our debugger to continue execution until reaching the **mov ebx, eax** instruction. When dealing with lots of low-level code, it is often convenient to step over functions to not waste time.

2.4.6 Locating the Menu

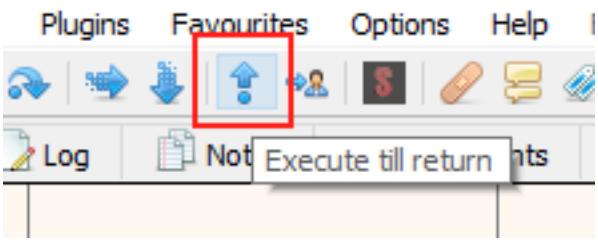
Unlike variables, code locations within a game will usually not change. Because of this, we can use the same location we found in the previous chapter to begin reversing. After attaching x64dbg to Wesnoth, navigate to the location we found in the last chapter (`0x007ccd9e`) and click on the dot to the instruction's left to set a breakpoint. The address will turn red to indicate that a breakpoint has been set. This breakpoint will pop whenever this instruction is executed.



Next, go back into Wesnoth and recruit a unit. Upon doing so, the debugger will pop at the same location we saw in the last chapter.



Click the *Execute till return* button once to execute until the first **retn** instruction.

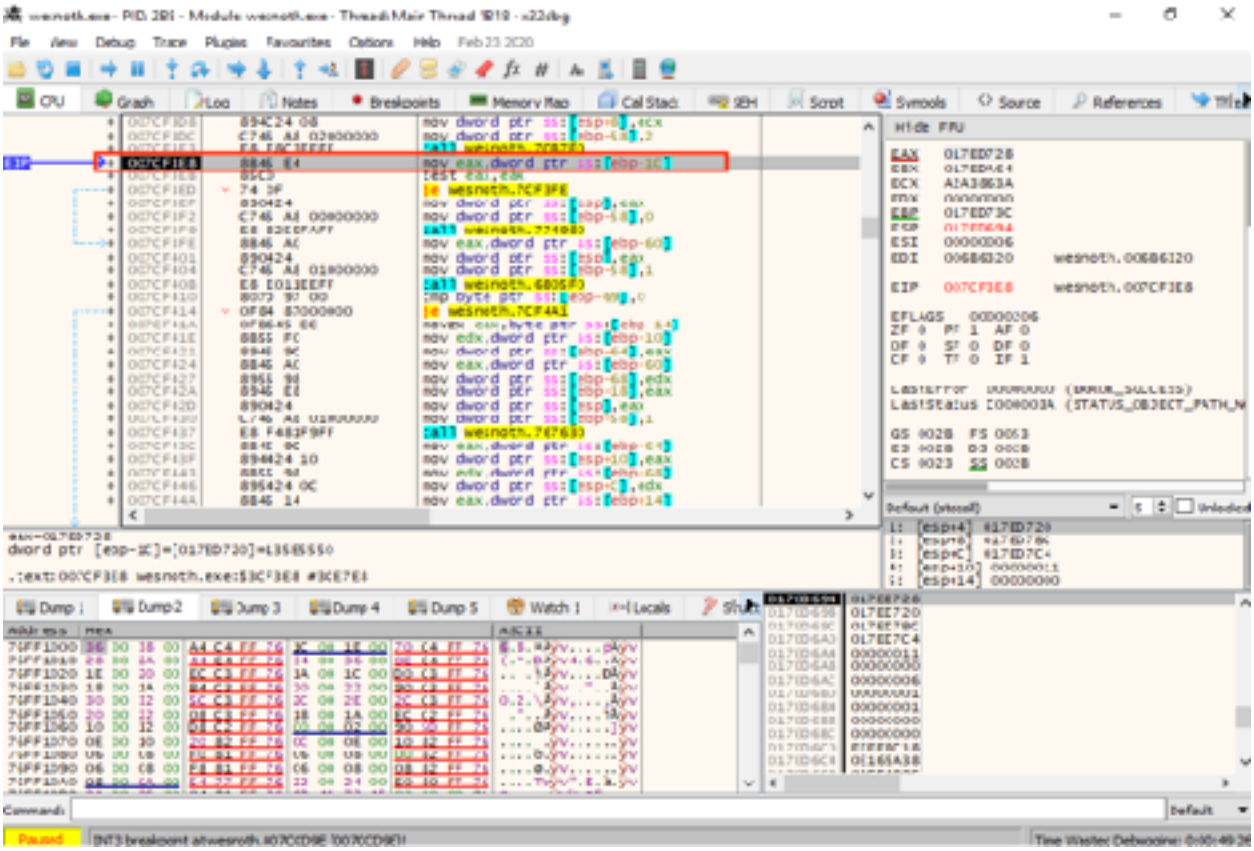


Playrix Ex. 1024, page 60

Once on it, click the *Step over* button to go to the calling code.



You should be sent to the following location:



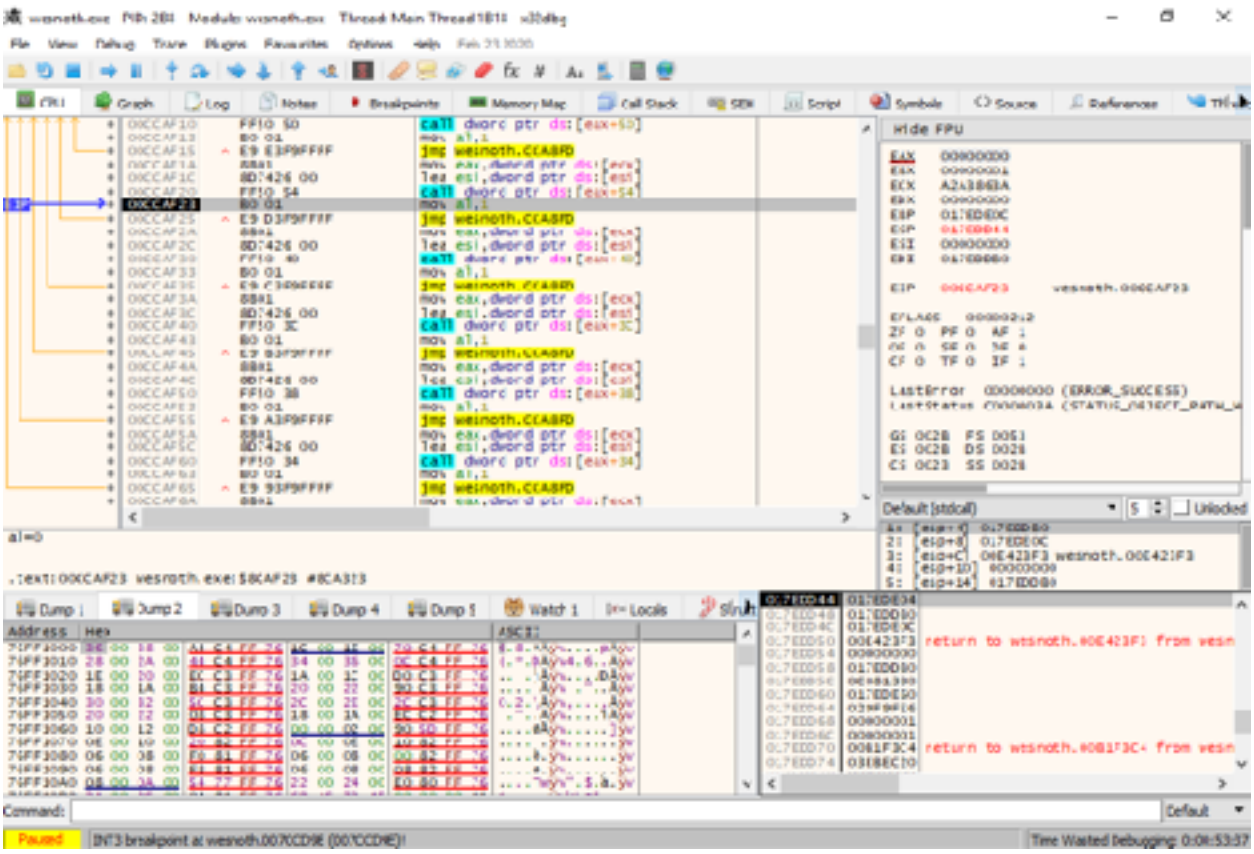
The **call** instruction above the highlighted line is the **call** we were just inside of. The code we are currently at was responsible for calling this function. We can use this technique to keep bubbling up to the function we care about.

We know that the function for handling the right-click menu will have many branches and calls. We can guess that when translated into assembly, the game's switch statement will most likely look something like:

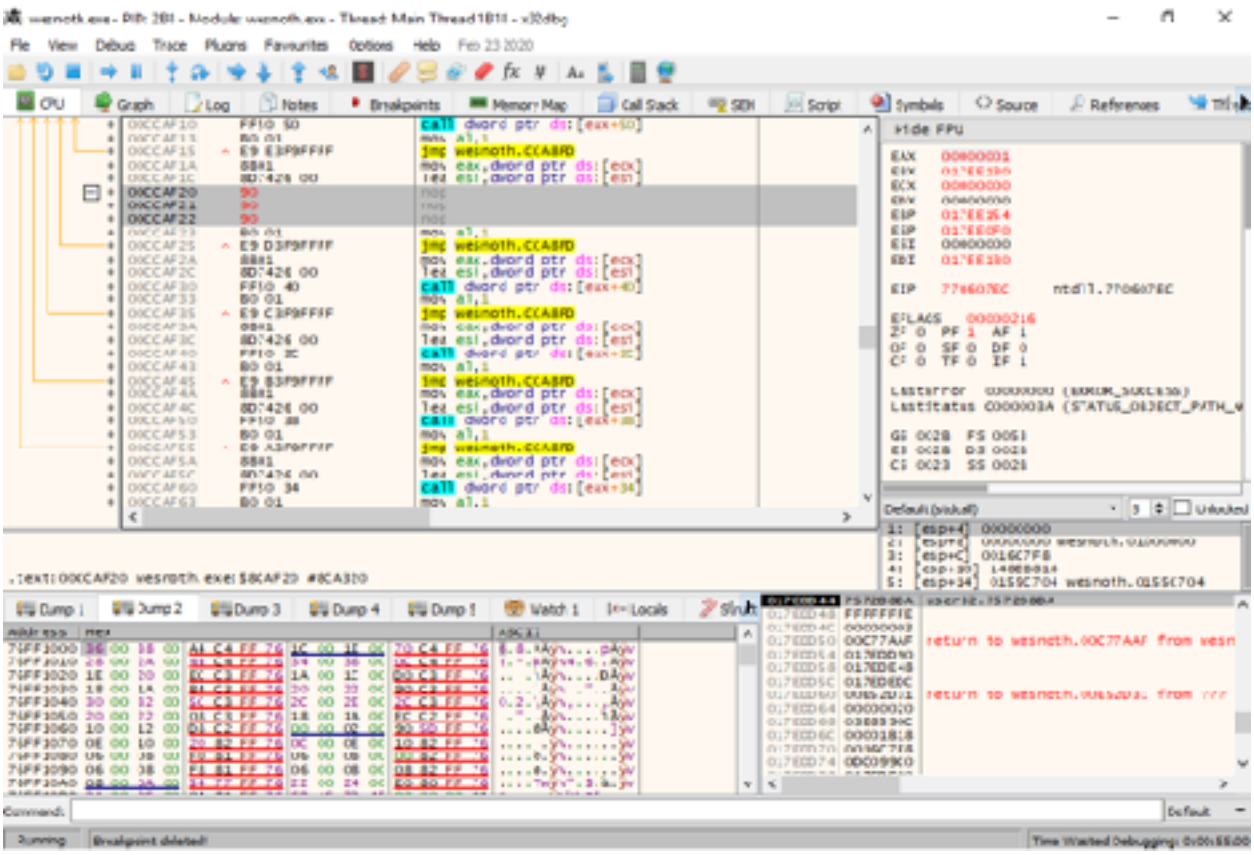
```
call some_address
jmp to_end
call some_address
jmp to_end
call some_address
```

```
jmp to_end
```

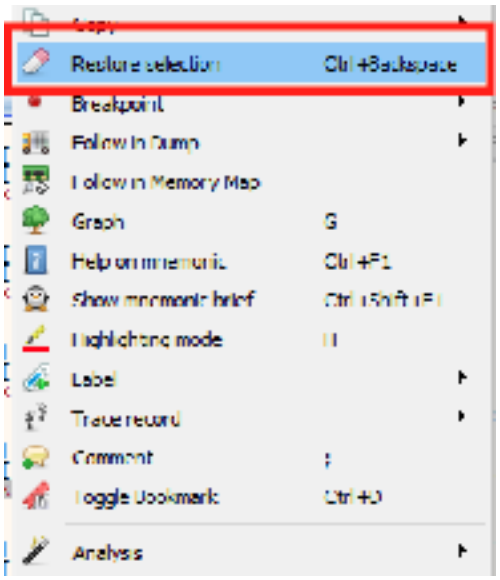
There will most likely be other instructions, but this is the format we are looking for. Keep following the cycle of executing until a return statement and then stepping out. After several times, you should land in the following code:



This pattern looks similar to what we were expecting. We can verify that this is the correct code by **nop**'ing out the **call** we just stepped out of, like so:



If you go back into the game and try to recruit a unit, nothing will happen. This is good verification that we found the function responsible for handling the right-click menu event of recruiting. Go back into x64dbg and right-click on the code we just changed and choose *Restore selection*. This will restore the original instruction.



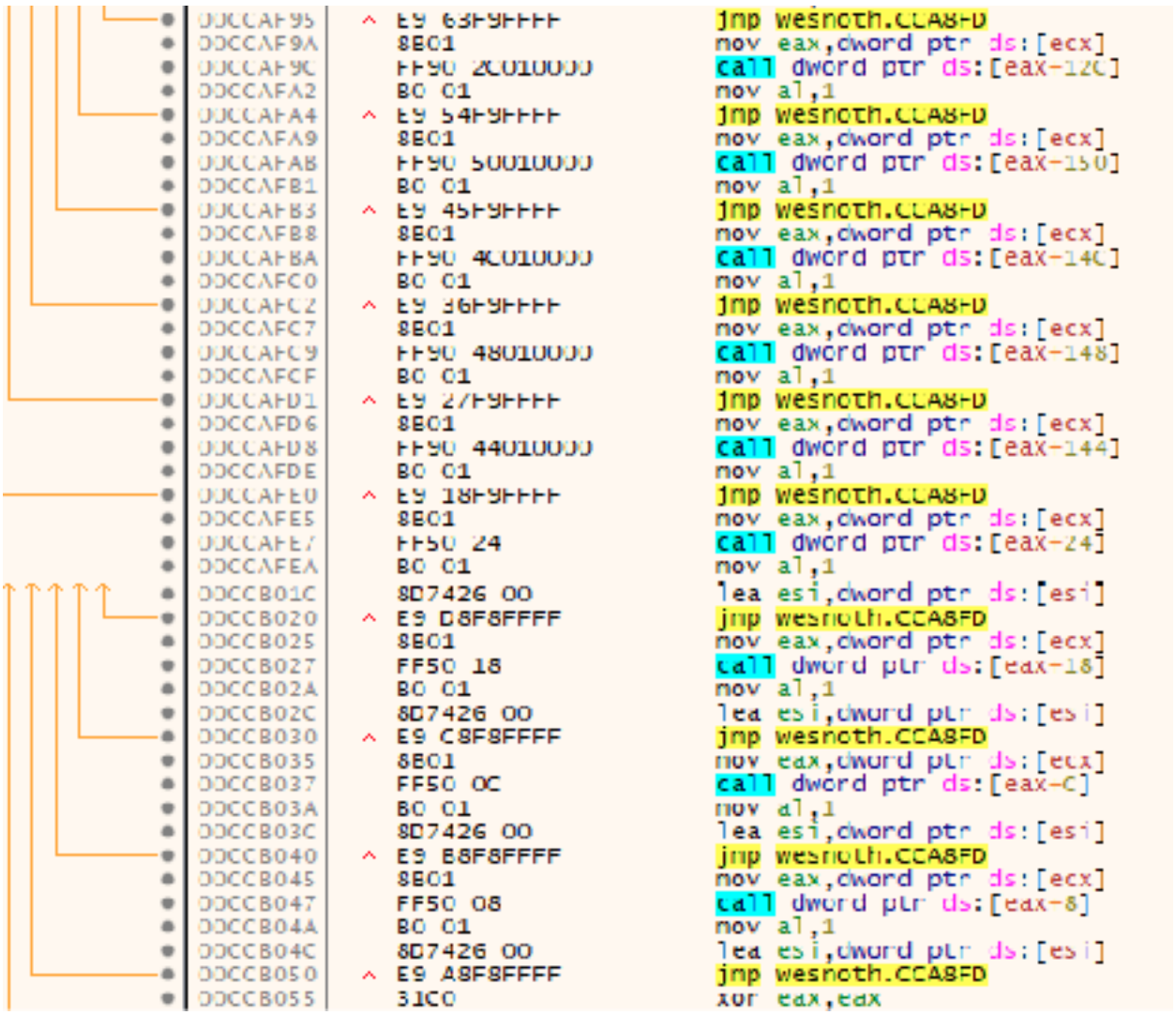
Playrix Ex. 1024, page 63

2.4.7 Locating Other Events

Now that we have found the **call** for the recruit event, we can use its structure to figure out how the other events in the game are called. The **call** looks like:

```
call dword ptr ds:[eax+0x54]
```

This **call** is not calling a static location. Instead, it is calling the location held in memory at **eax+0x54**. If we look at the other calls in the function, we see that they all have a similar form, with only the last number changing.



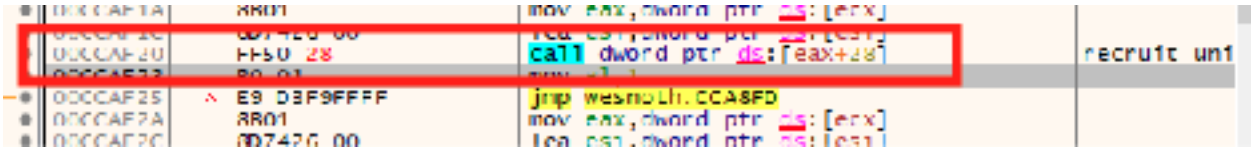
Due to this structure, we have to revise our original code model that had a switch statement. In the screenshot above, we can see that the last number is always a multiple of 4. Therefore, we can assume that these functions are most likely stored in some type of list or array. The original's game code probably looks something like:

```
void* context_menu_functions[MAX_FUNCTIONS] = {
    terrain_description,
    recruit_unit,
    ...
}

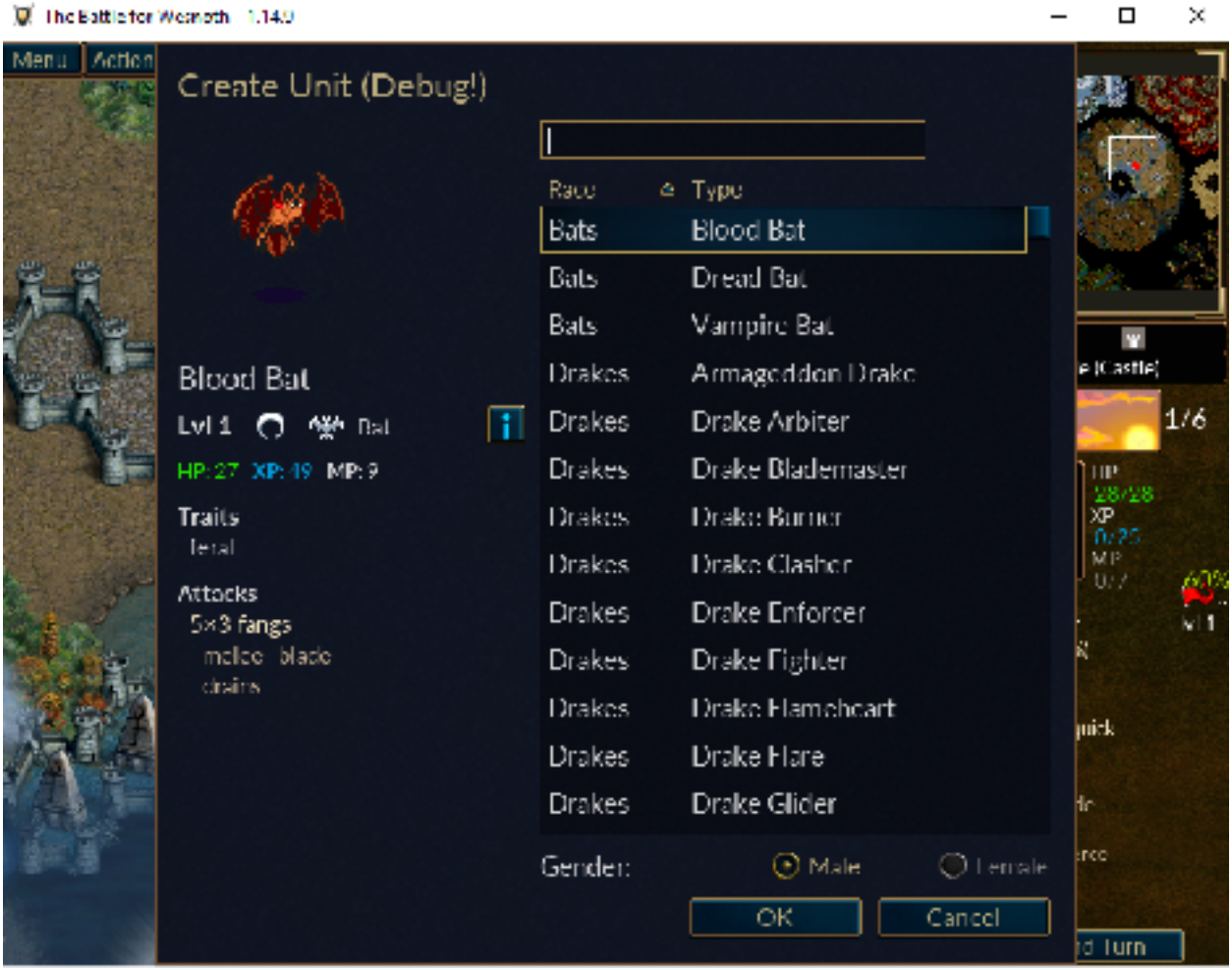
context_menu_functions[option_selected]();
```

This code stores a pointer to each function in an array. The **option_selected** variable can then be used to retrieve the correct function from the array and execute it. We will cover pointers in a future chapter. It's important to note that even though we had the wrong original code in mind, the overall structure of branching will always be obvious in a game's code.

We know that the offset for recruiting is `0x54`. To determine other offsets, we can change the recruiting call to other values and note the result when we use the *Recruit* entry on the context menu. Starting at **eax**, we can try each multiple of 4 and log their result (**eax + 4**, **eax + 8**, **eax + 0xc**, **eax + 0x10**, and so forth). For example, by changing the value to `0x28`, a terrain description will show up when we try to recruit a unit.



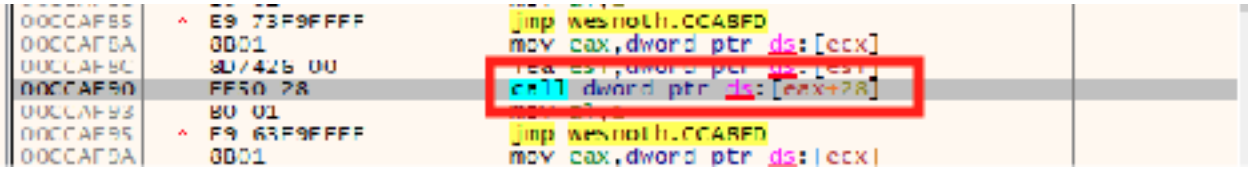
Far more interesting is when we change the value to `0x68`. In this case, a *Debug* menu to spawn units will appear.



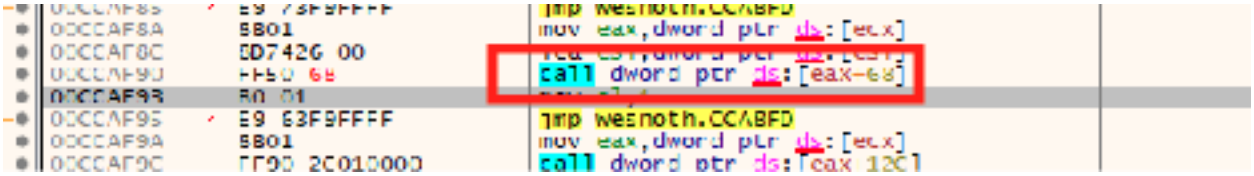
2.4.8 Change

We can use the two values we found above to create our hack. First, we will locate the menu item code responsible for showing the terrain description. Then we will change this value to call the debug menu instead.

We know that the value for the terrain description is `0x28`. By observing the area around the recruit call, we will eventually find the code responsible for the terrain description event.



Next, we will change this value to `0x68`. This will invoke the debug menu anytime we select *Terrain Description*. Since the terrain description is available on any tile, this will allow us to recruit units anywhere.



Once this change is made, go back into Wesnoth, select a random tile, and choose *Terrain Description*. Select a unit from the debug menu and verify that the hack works.



2.5 Code Caves

2.5.1 Background

In the previous two chapters, we made changes to the game's code to alter its functionality. For both of these changes, we replaced the original instruction with a new instruction. But what if we want to keep the original instruction or replace it with multiple instructions? In these cases, we will need to use a code cave.

A code cave is a section of the game's memory that we fill with instructions. We then change the game's original code to call these instructions. The name comes from the fact that we are creating a hidden "cave" of instructions. Most games will have large sections of unused memory between functions or at the end of the executable. These locations are perfect for creating a code cave in.

2.5.2 Redirection

In our last chapter, we changed the function for displaying a terrain description to instead call a debug menu. By using a code cave, we can still invoke the debug menu, but also call the terrain description function after. By doing this, we won't lose any functionality in the game.

The original instruction for the terrain description **call** looked like:

```
0x00CCAF90 call dword ptr ds:[eax+28]
```

For this example, assume that there is an empty section of memory at `0x00D00000`. Our first goal is to recreate the original **call** at `0x00D00000` and then redirect the original code to this new code. First, we will copy the original instruction to `0x00D00000`:

```
0x00D00000 call dword ptr ds:[eax+28]
```

Next, we will redirect the original code to this call:

```
0x00CCAF90 jmp 0x00D00000
```

Finally, in our code cave, we need to go back to the original code. This can be accomplished by jumping to the instruction that comes after the one we replaced. In this case, the next instruction in the game is at `0x00CCAF93`. Our completed code cave would then look like:

```
0x00D00000 call dword ptr ds:[eax+28]
           jmp 0x00CCAF93
```

2.5.3 Restoring Instructions

As of right now, this code cave only recreates the original instruction. This is an important first step to ensure that our redirection isn't breaking anything. This is not always the case, especially when dealing with game functions that modify the stack. We will discuss how to deal with these in future chapters. For now, just be aware that redirecting the game's code will not always be a smooth process.

When writing a code cave, it's critical to only modify what you require and nothing else. Accidentally changing other registers, sections of memory, or the stack can cause the game to crash. To illustrate this principle, imagine we had the following code that we intended to redirect:

```
mov eax, 999
call 0xDEADBEEF
```

Let's say we redirected the call to a code cave that looked like:

```
mov eax, 123
call some_other_function
call 0xDEADBEEF
jmp back
```

If the function at `0xDEADBEEF` required **eax** to be 999, the game would throw an exception and crash. While this is a trivial example, calling game functions will often have many side-effects that you won't be aware of.

Playrix Ex. 1024, page 69

To save and restore the game's register values (**eax**, **ebx**, **ecx**, and so forth), we will use two instructions: **pushad** and **popad**. **pushad** pushes (or saves) all register values on the stack. **popad** pops (or restores) all register values from the stack. In future chapters, we will cover the stack itself and how to restore the game's stack. However, restoring the register values will prevent most crashes.

2.5.4 Cave Skeleton

With these instructions, we now have a basic skeleton for a code cave. It looks like:

```
pushad
execute new functionality
popad
invoke original instruction
jmp back to game's code
```

Let's return to our Wesnoth example. In Section 2.5.2 above, we had the following code cave:

```
0x00D00000 call dword ptr ds:[eax+28]
           jmp 0x00CCAF93
```

With **pushad**/**popad**, we can now safely introduce new instructions. The code to invoke the debug menu looked like:

```
call dword ptr ds:[eax+68]
```

Let's assume that this **call** doesn't modify the stack in any way. We can safely call this function in our code cave by saving the registers, calling the function, and then restoring the registers. We will then execute the original instruction and jump back to the game's code after the original instruction. Our final code cave would look like:

```
0x00D00000 pushad
           call dword ptr ds:[eax+68]
           popad
           call dword ptr ds:[eax+28]
           jmp 0x00CCAF93
```

By doing this, we have created a cave in the game's code that replaced one instruction with multiple instructions. As long as everything is restored, there is no limit to the amount of new code that can be called.

Playrix Ex. 1024, page 71

2.6 Using Code Caves

2.6.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

2.6.2 Identify

Our goal in this chapter is to create a code cave inside Wesnoth. This code cave will be executed whenever we select *Terrain Description*. The code cave will give us 999 gold before bringing up the terrain description box.

2.6.3 Understand

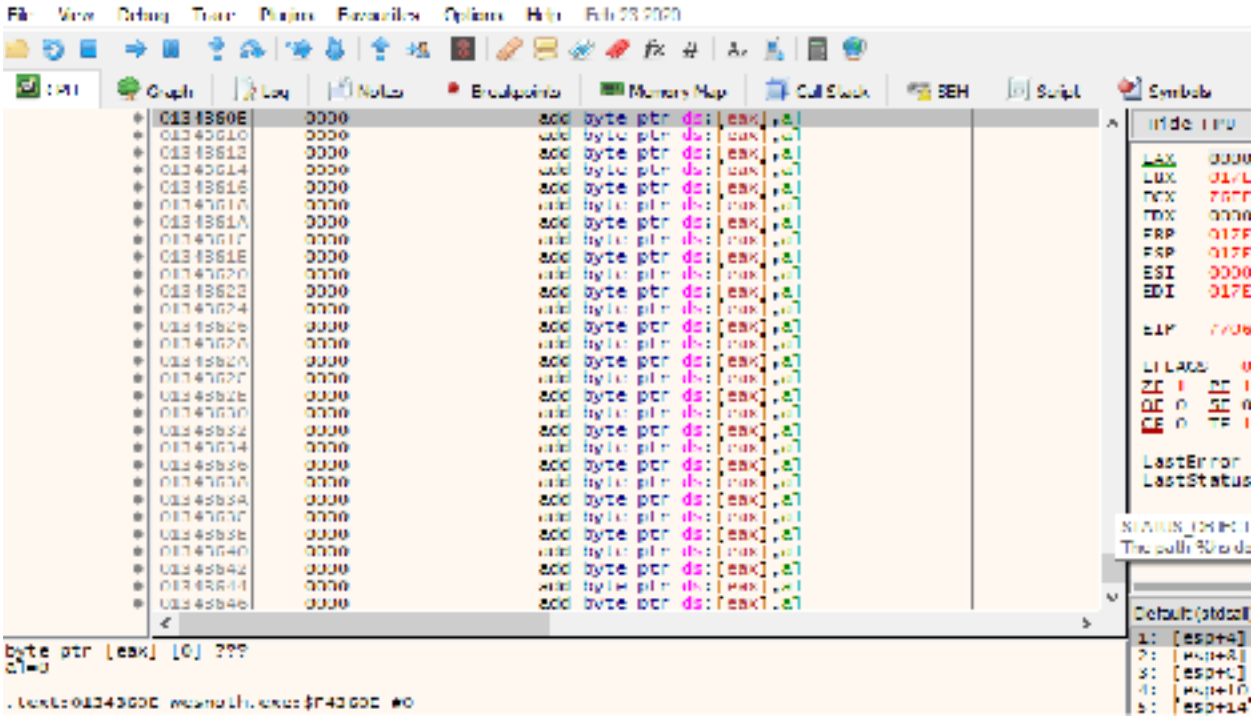
To create a code cave, we will first need to find two locations: the location to redirect and the location to place the cave skeleton. We will then create a cave skeleton at the second location. With that created, we will redirect the first location to jump to the skeleton.

2.6.4 Locating Gold

Since we will be modifying our gold with this hack, we need to find our gold address. Our first step is opening up Wesnoth and creating a local game. Then we'll follow the steps in Chapter 1.5 to find our gold address. Like we discussed in Chapter 2.3, our gold address will be at a different address than in previous chapters. Once we have found our new gold address, we will close down Cheat Engine but keep Wesnoth open. In this chapter, we will use the value of `0x05F3B85C` as our gold address.

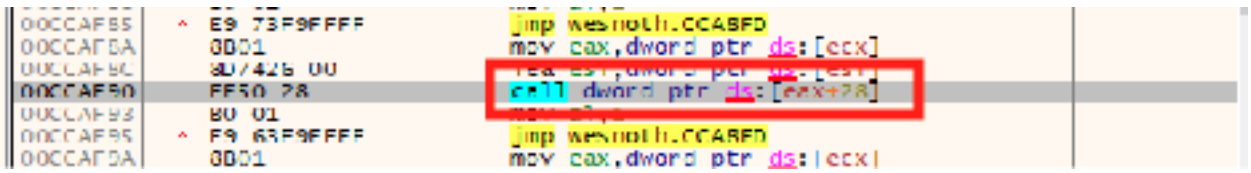
2.6.5 Locating Code Cave

First, we will locate where to place our code cave. While there are many places where we can find empty sections of memory, the quickest and easiest approach is to scroll to the end of the Wesnoth module. At the end of most executable modules, there is a large section of empty data that can be modified. In our example, this memory is around `0x0134360E`.



2.6.6 Hooking Location

Next, we need to identify the address to hook. In this chapter, we will be hooking the method that displays the terrain description. To find this address, we can use the same method and address we identified in Chapter 2.4. This **call** is at `0xCCAF90`.



2.6.7 Redirection

With our code cave and hooking locations identified, we can now create the code cave. When modifying a game's code in a debugger, it's important that the game is in a paused state. This prevents the game's normal execution from accidentally entering our code cave before we have finished it. If it does, the game could jump to non-existent code and crash. To pause the game, we can use the *Pause* button next to the *Continue* button.



We can now redirect the hooking location to jump to our code cave. In Chapter 1.1, we covered the idea of opcodes. Each instruction has a different opcode and also a different opcode length. This is because certain instructions require additional pieces of data to execute. For example, the **pushad** instruction is represented by the 1 byte opcode `0x60`. This is due to the instruction not requiring any additional data to execute. As a contrast, the instruction **mov al, 1** is represented by the 2 byte opcode `0xB0 01`. This is because the **mov** instruction requires the data to be moved (in this case, the value 1) to be encoded somewhere inside the opcode.

The **jmp** instruction's opcode is 5 bytes long. This is because the opcode encodes the address that will be jumped to. Because of this, we will need to find a location of at least 5 bytes to place our **jmp** instruction. To do this, let's examine the method for displaying the terrain description:

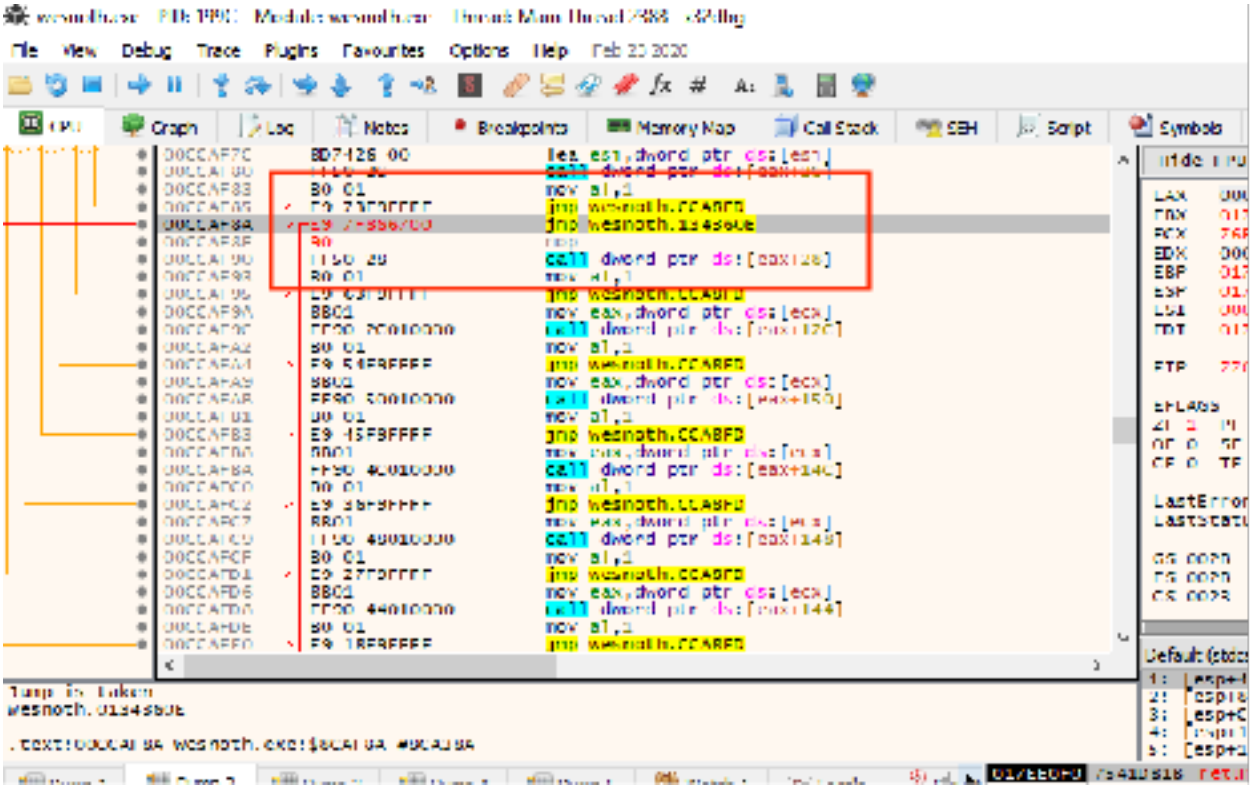
```
opcode      instruction
----------------------------------------
8B01        mov eax, dword ptr ds:[ecx]
8D7426 00   lea esi, dword ptr ds:[esi]
FF50 28     call dword ptr ds:[eax+28]
B0 01       mov al,1
```

Unfortunately, there is no instruction in this method that has a 5 byte opcode. In this case, we will need to replace the first two instructions. When writing our code cave, we will need to remember to replace both of these. However, the opcodes of these first two instructions (`0x8B01` and `0x8D742600`) combine to 6 bytes total. When we replace the first 5 bytes with our jump, the last byte (`0x00`) will stay and potentially be

Playrix Ex. 1024, page 74

executed. To ensure that our change does not cause the game to crash, we will replace the last byte with a **nop** instruction. x64dbg will automatically do this when assembling instructions.

With all of this out of the way, we can finally make our code change. Navigate to the first **mov** instruction at `0x00CCAF8A` and change the instruction to **jmp 0x0134360E**. Make sure that the *Fill with NOPs* option is checked when assembling this instruction. This jump will be responsible for jumping to our code cave.

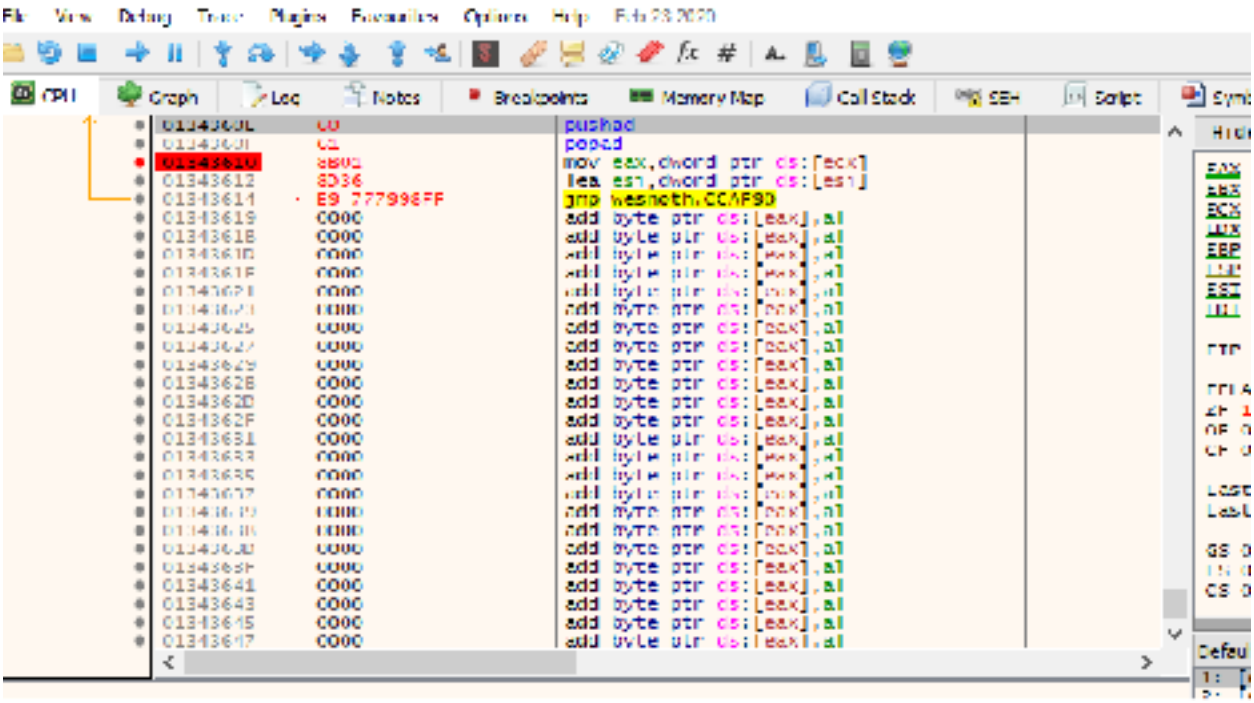


2.6.8 Cave Skeleton

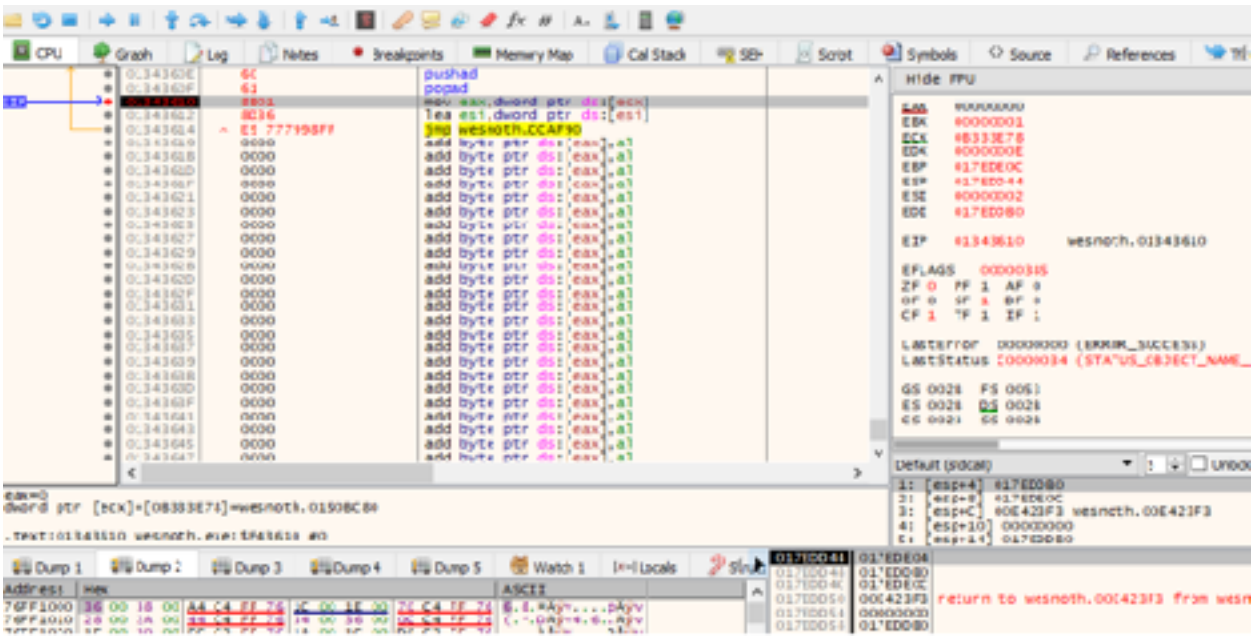
Next, we can write our cave skeleton. For now, our cave skeleton will just save and restore the registers, replace the original instructions, and then jump back to the original code. This will execute identically to the original code and will allow us to verify that our redirection was successful. Navigate to `0x0134360E` and insert the following code:

```
pushad
popad
mov eax, dword ptr ds:[ecx]
```

```
lea esi, dword ptr ds:[esi]
jmp 0xCCAF90
```



To verify that this code cave is being called, place a breakpoint at `0x01343610` and then continue execution of the program. When you bring up the terrain description, the breakpoint in our code cave should pop.

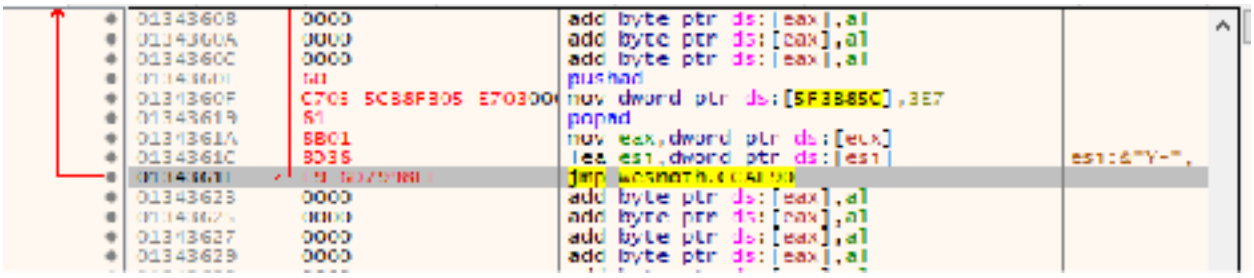


2.6.9 Change

With a working skeleton, we can now change our gold. In between the **pushad** and **popad** instructions, we will insert our instruction to modify our gold. To do this, we will move the value of 999 into the address holding the gold value. This instruction will look like:

```
mov dword ptr ds:[0x5F3B85C], 0x3E7
```

With this instruction, our final code cave will look like so:



If we go back into Wesnoth and select *Terrain Description* on a tile, our gold will change to 999 before the terrain description box appears.



2.7 Dynamic Memory Allocation

2.7.1 Overview

In previous chapters, we modified the player's gold in Wesnoth. Whenever we restarted the game, we had to repeat the process of finding the player's gold memory address, as it was different each time. This is because of Dynamic Memory Allocation, or DMA. To write hacks that can be reused and distributed, we will need to somehow convert these "random" addresses into consistent addresses. There are many methods to accomplish this task, but first we need to discuss how DMA works and why it exists.

2.7.2 Background

As we discussed in Chapter 1.2, games are large programs with many resources. There is no way to fit all of a game's data into RAM at one time, so it must be loaded when it is needed. For example, a game will not load an enemy's model or image until the player is about to encounter them. This process is known as dynamic loading of resources.

These dynamically loaded resources must be placed in some section of memory so that the game can access them again. The game is responsible for creating and destroying these sections of memory. The creation is known as allocation, and the destruction is known as deallocation. Dynamic Memory Allocation is therefore the process of creating memory sections to hold resources when they are needed by the game. The game can only ask the OS for memory and cannot control where this memory is located.

Let's consider the player's gold in Wesnoth and how it is created. When Wesnoth is started, a player's profile is loaded into the game's Player class. The player can then select from a variety of game modes and other options in the main menu. If the player then starts a game, several items are allocated and placed in the Player class, such as the player's race, their available units, and their gold. When the player quits the game,

Playrix Ex. 1024, page 78

these values are then destroyed. This is why the player's gold address is always different.

2.7.3 Programming

To program hacks, we need some way to consistently find the gold address without searching in Cheat Engine. There are several ways to accomplish this:

- An automated scanner, such as Cheat Engine
- Code Caves
- Reversing

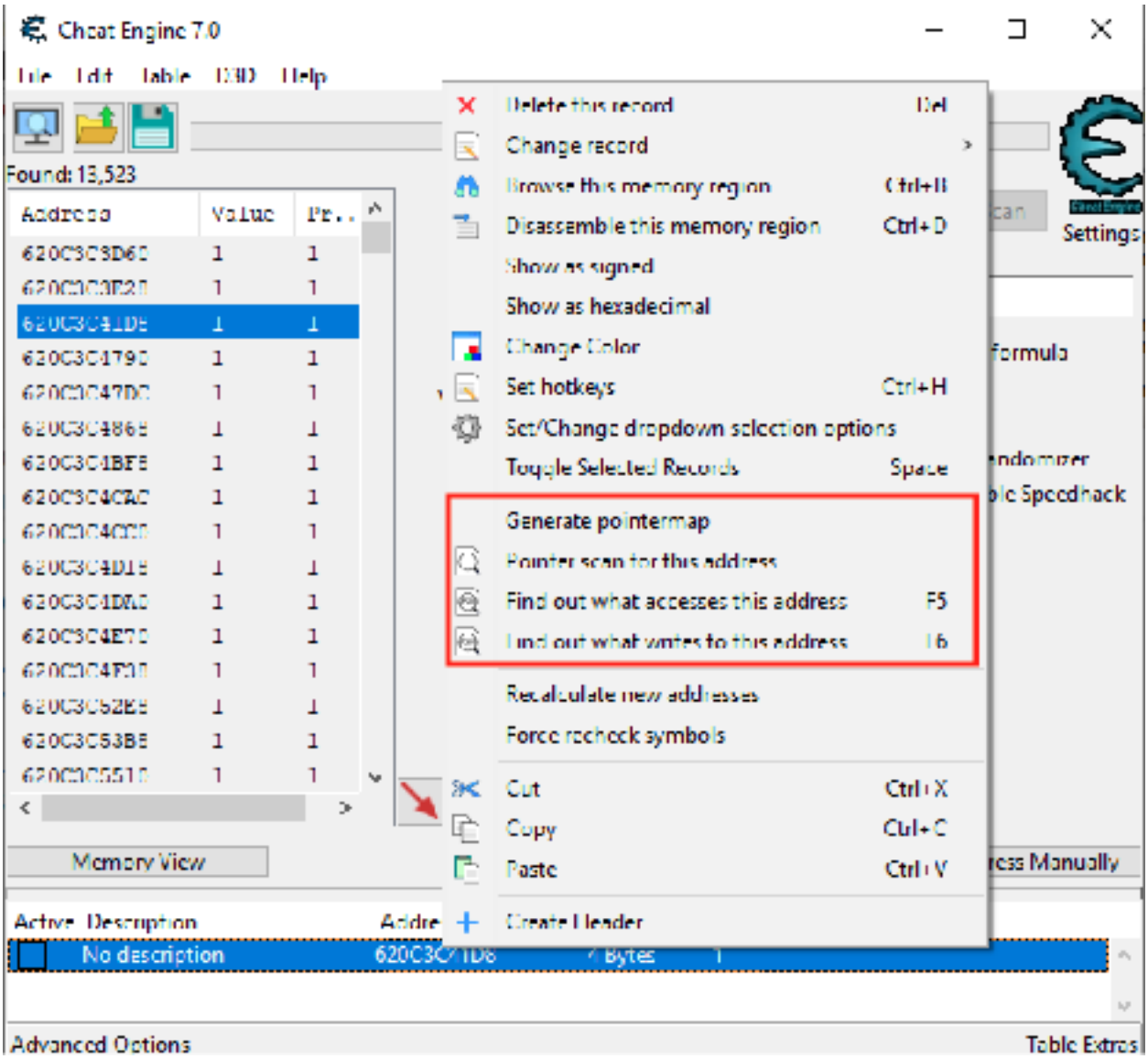
These methods can be used to find any dynamic address. We will discuss each of these briefly in this chapter, but in future chapters we will use all of the methods.

When using Cheat Engine or reversing, our goal will be to find something known as the base pointer. In general, the base pointer represents a memory address that is always consistent and can be used to offset to the values we care about. This method works because there are some addresses that must be constant for the game to find them. For example, in Wesnoth, the game needs to know where the Player class is. If we find the Player class, we can then use it as a base pointer to offset to our gold address.

2.7.4 Cheat Engine

One feature of Cheat Engine is the ability to conduct a pointer scan. This can be done by finding an address (such as the player's gold) and then right-clicking on it to bring up a context menu. This context menu contains all the pointer scanning functions.

Playrix Ex. 1024, page 79



This feature returns all the memory locations that currently reference the selected address. These addresses can then be saved to a scanfile. After that, the game can be reloaded and the address can be found again. By then comparing the new memory locations to the scanfile, only the consistent locations can be narrowed down, similar to regular memory searching. Eventually, we will be left with only the pointers that always point to our selected address.

2.7.5 Code Cave

Another method to defeat DMA is using a code cave. With this approach, a location is found where the desired value is accessed. For our Wesnoth example, this can be

Playrix Ex. 1024, page 80

anywhere that gold is changed. Immediately after this location, the code is redirected to our code cave. In our code cave, we can then save the current value to a piece of memory we control. This memory can then be accessed consistently by our hack.

For example, the code in Wesnoth responsible for decreasing our gold when recruiting a unit looks like:

```
sub dword ptr ds:[edx+4], ecx
```

When this instruction executes, **edx + 4** contains a reference to the gold memory address and **ecx** contains a reference to the cost of the unit just recruited. By redirecting the code immediately following this address to a cave, we can then save the address's value in the cave. An example cave is shown below that would accomplish this:

```
pushad
mov dword ptr ds:[0x12345678], edx+4
popad
...original instruction replaced...
jmp 0xredirect_location
```

With this done, our hack could then reference `0x12345678` to get the current value of the gold address.

2.7.6 Reversing

The final method of dealing with DMA is reversing the target. This method uses a combination of the previous two methods and is the most versatile. In this approach, we first find an instruction that modifies the value we care about, such as the **sub** instruction in the previous section. Then, we analyze the function before that instruction and determine where the register we care about (in this case, **edx**) is assigned. Often, this will be assigned the value of another register with an offset, such as **eax+60**.

We then repeat this process to find where this previous register is assigned. Eventually, we will find the base value or pointer used to assign all these values. This base pointer can then be combined with all the offsets we reversed to retrieve the address we care about.

Playrix Ex. 1024, page 81

2.8 Defeating DMA

2.8.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

2.8.2 Identify

Our goal in this chapter is to locate the base pointer for our Player class. After that, we need to figure out how to offset our gold address from this base pointer.

2.8.3 Understand

When a player begins a game, Wesnoth uses DMA to allocate several values, including the player's gold. This means that the player's gold address will be at a different address for each game. By contrast, there are several values that remain constant between games, like the player's profile name. These constant values must be stored in some sort of Player class. Since these values persist for every game, there must be a static address that Wesnoth uses to locate them. If we can find this static address, we can then offset to our dynamic gold address while in game.

To visualize this, let's imagine that Wesnoth's Player class looks something like:

```
class Player {
    string player_name = "IEUser";
    int wins = 100;
    Game game = null;
}
```

And the Game class looks something like:

```
class Game {
    string side;
    int gold;
    int turn;
}
```

When a player enters a game, the game's code will allocate memory for this game object and all the values that it contains:

```
player.game = new Game("Human", 100, 1);
```

By finding the value of the gold address, we can reverse the game to find the value of the Game address for the current game. We can then use that address to find the address of the Player class. Since the Player class is always loaded, it will be a consistent address. From the Player class, we can then use the addresses we found while reversing to offset to the current gold address.

2.8.4 Locating Gold

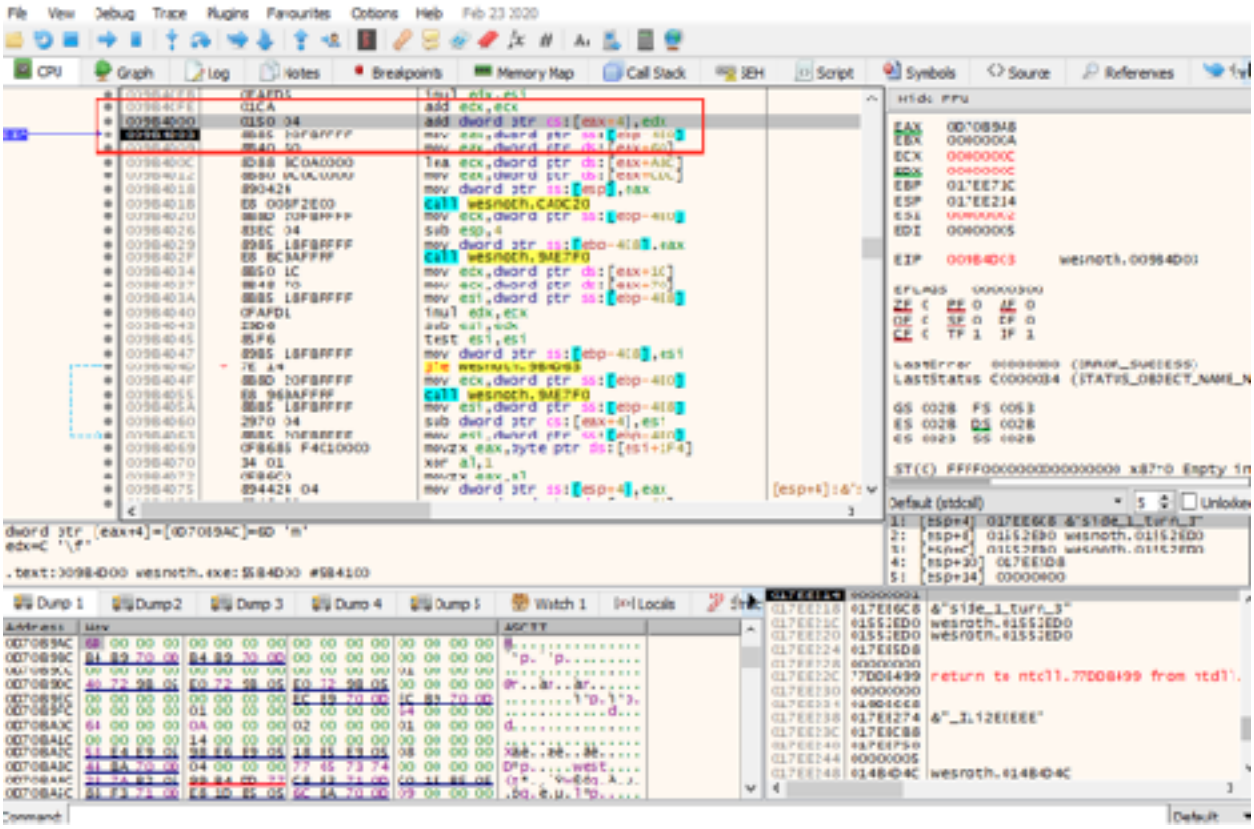
For the last time in this book, we will need to find the address of our gold value. Our first step is opening up Wesnoth and creating a local game. Unlike the previous chapters, make sure that you give yourself *Income* to make the reversing process easier. Also, make sure the second player is set to a *Computer* opponent.



Then follow the steps in Chapter 1.5 to find your gold address. Once you have found your new gold address, close down Cheat Engine but keep Wesnoth open.

2.8.5 Base Pointer

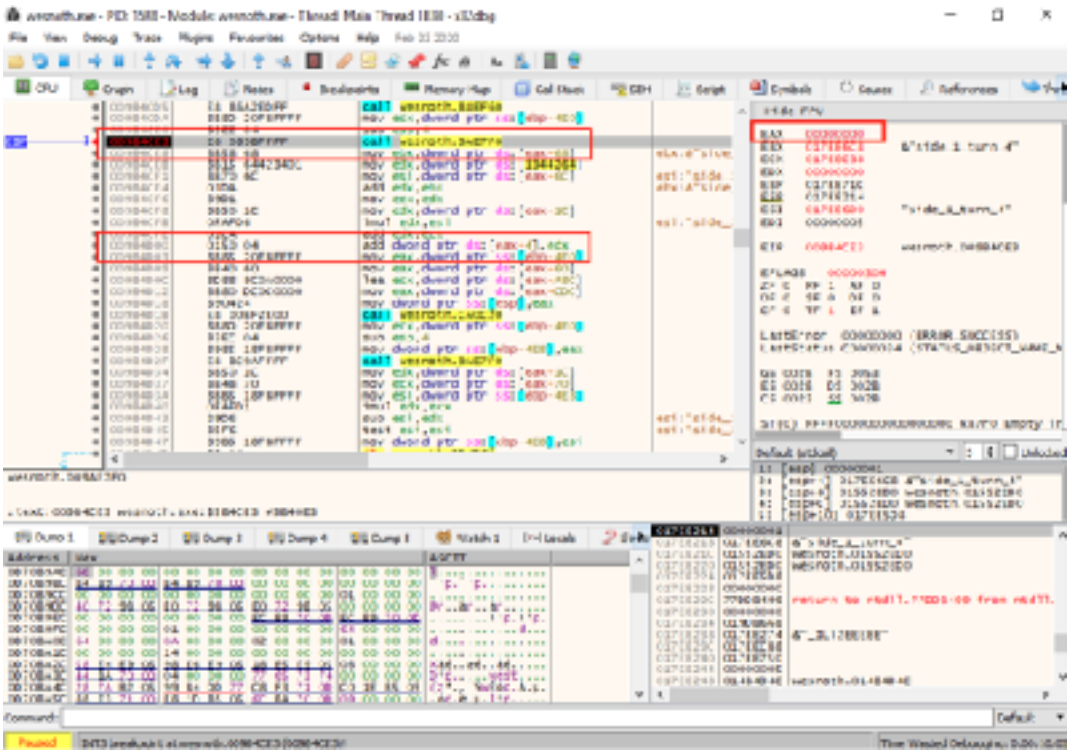
Next, attach x64dbg to Wesnoth and set a breakpoint on write on the gold address that you found. Unlike previous chapters, do not recruit a unit. Instead, choose to end your turn. Upon ending your turn, your breakpoint should pop as income is added to your gold.



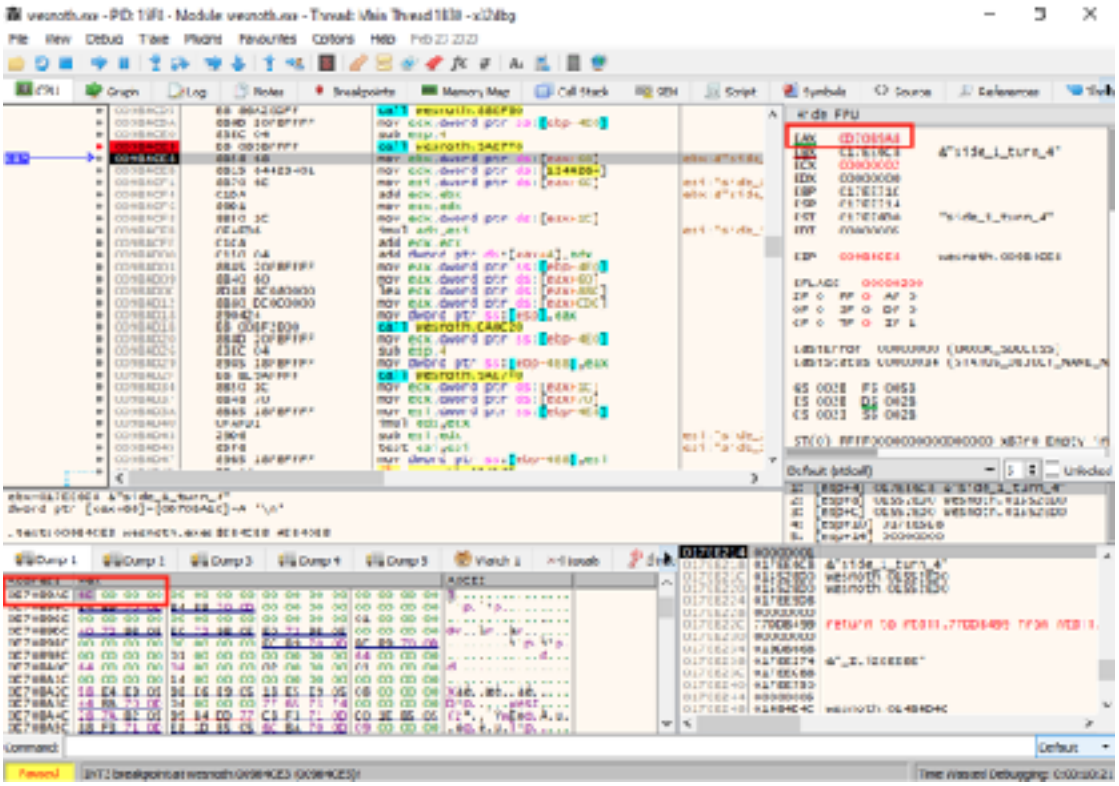
Let's briefly examine the instruction that our breakpoint popped on:

```
009B4D00 add dword ptr ds:[eax+4], edx
```

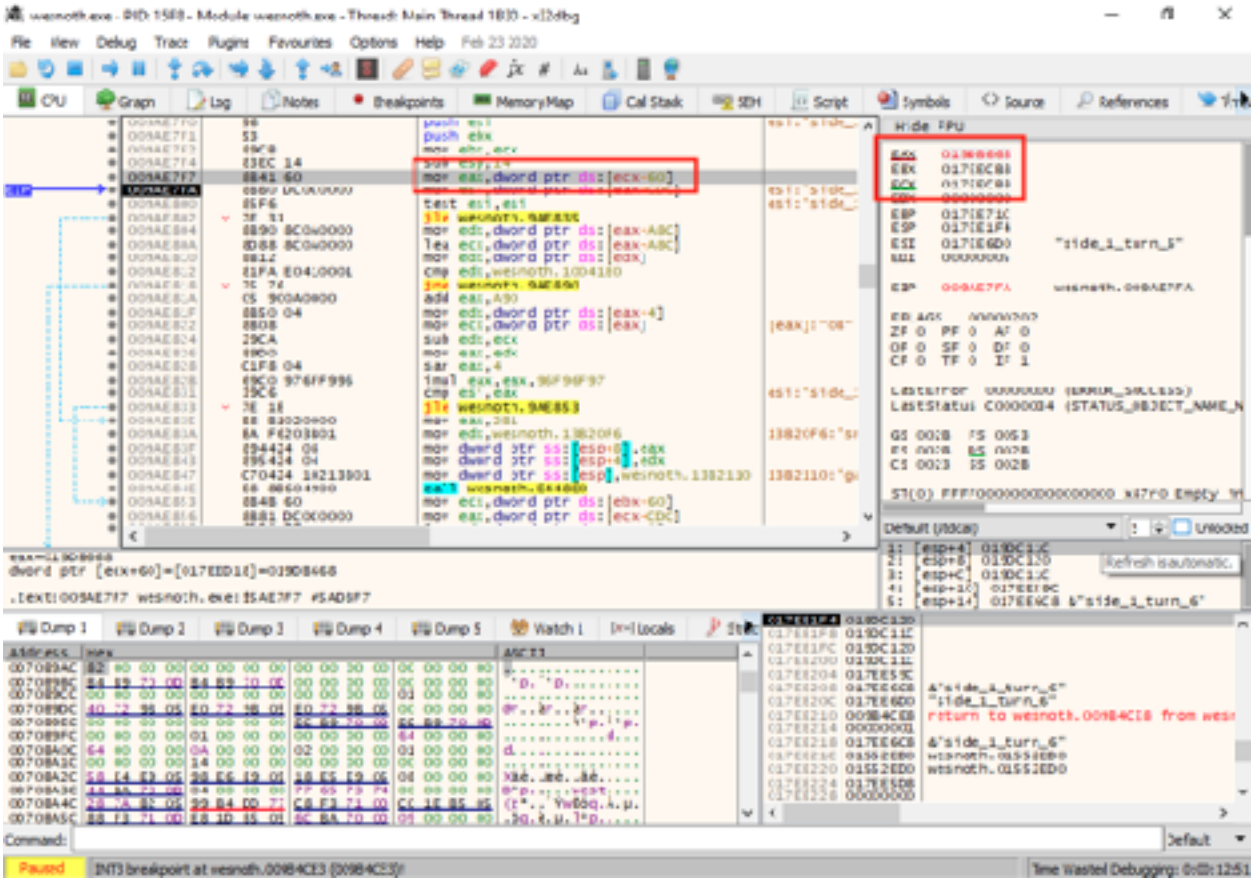
When this instruction is executed, **eax+4** holds the value of our gold address (in this instance, `0x0D70B9AC`). Our next step is to determine how **eax** is assigned. If we look above the **add** instruction, we see several **mov** instructions that reference the value of **eax**. Above these, we have a **call** instruction to an unknown function. To determine if this function is responsible for setting **eax**, we can set a breakpoint on this **call** and then resume Wesnoth. When we end our turn again, this breakpoint will be hit.



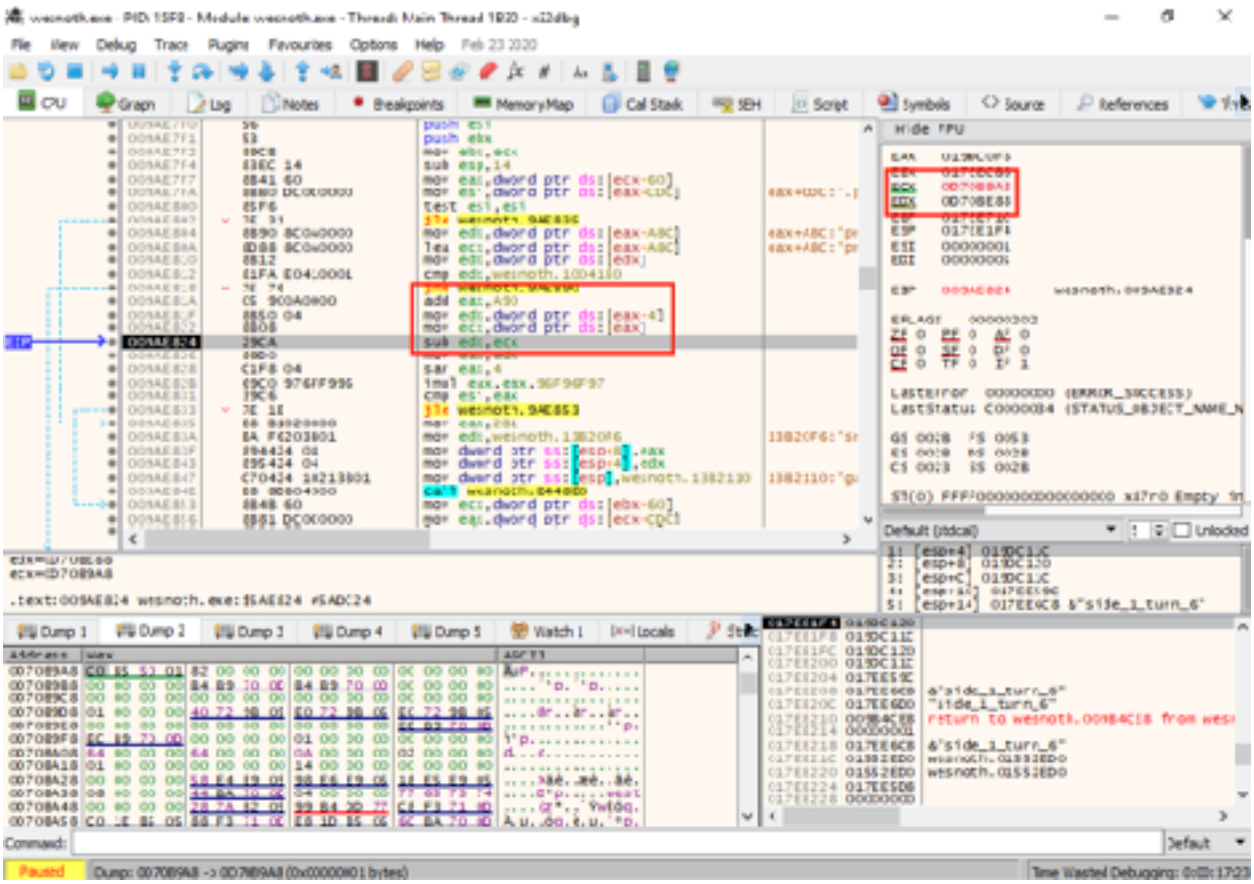
Observing the registers, we see that **eax** is set at 0 entering this function, meaning this function must be acquiring the correct value for **eax**. We can confirm this by stepping over the function and noting the new value of **eax**.



With this confirmed, resume Wesnoth and end your turn again to trigger the same breakpoint. This time, step into the function. After stepping through a few lines, we see that **eax** is being set based on the value of **ecx + 60**.



For now, we will note that we need to determine the value of **ecx**. However, **eax** is still not close to the value of our gold address. Before we move on to determining the value of **ecx**, we need to determine how **eax** is modified from the initial assignment to reach the gold address. If we continue down the function, we see that the value of `0xA90` is being added to **eax**. After that, **edx** is loaded with the value of **eax + 4**. Let's step to the address after that code to see what value is being loaded into **edx**.



We can see here that **edx**'s value is only 4 away from our gold address, identical to how the initial **add** instruction referenced it. From this, we know so far that **[[ecx + 60] + 0xA90] + 4** is our gold address. Our next step is determining **ecx**.

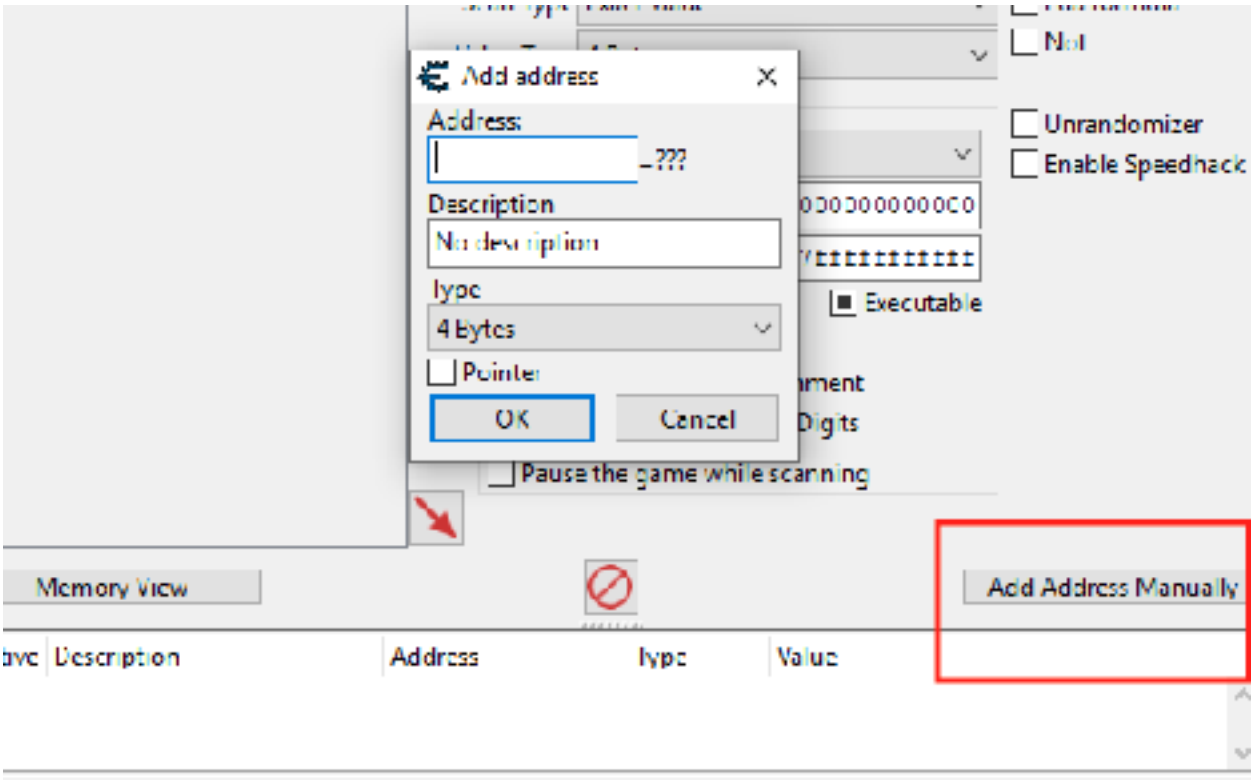
When locating base pointers, it's important to stop each time a new register or address is introduced and ensure that it is actually random. To do this, first make a current note of **ecx**'s value when it is loaded into **eax**. In this case, that value is `0x017EECB8`. Next, make a note of the address of the instruction that assigns the value so that you can place another breakpoint here. In this case, it will be the **mov** instruction at `0x009AE7F7`.

Now, detach x64dbg and then close Wesnoth. Once it is closed, start Wesnoth again, recreate a game with the same income settings, and reattach x64dbg. Place a breakpoint at the address noted (`0x009AE7F7`) and then end your turn. When the breakpoint pops, observe the new value of **ecx**. In this case, it's the exact same as the last time (`0x017EECB8`). Therefore, we know that this must represent a base pointer that doesn't change. If it did change, we would have to continue with the reversing process.

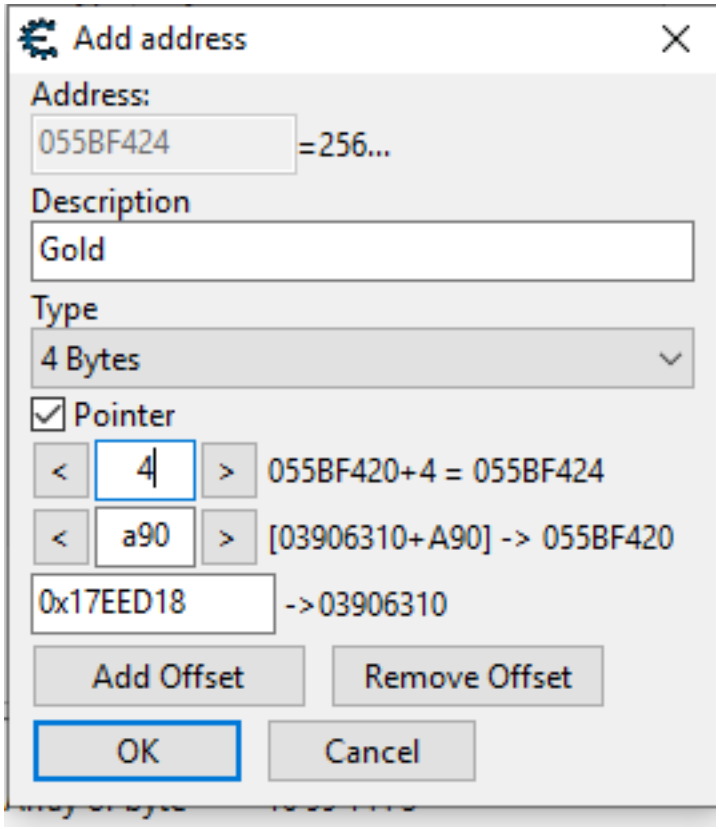
2.8.6 Change

With our base pointer found, we can now save its value for use in future projects. From our reversing, we know that the value of the player's gold in Wesnoth is always at: `[[0x017EECB8 + 0x60] + 0xA90] + 4`. To simplify, we know that `0x017EECB8 + 0x60` will always be `0x017EED18`, so the actual offset can be represented as: `[[0x017EED18] + 0xA90] + 4`.

Cheat Engine allows us to manually add pointers with offsets as addresses. We can use this to verify that our value is correct. First, open Cheat Engine and attach it to Wesnoth. After attaching, select the *Add Address Manually* button.



In the box that pops up, select the checkbox for *Pointer*. When doing this, Cheat Engine will prompt you for a base address and one offset. Type in `0x017EED18` as the base address and `0xA90` as the offset. Then add another offset and type in 4. This is all the information we found while reversing.



In the *Address* box at the top, you should notice that Cheat Engine has correctly resolved our offset to the current amount of gold that we have. If you close and restart Wesnoth, this pointer will then change to the new value for gold.

Playrix Ex. 1024, page 89

Part 3
Programming

Playrix Ex. 1024, page 90

3.1 Programming Fundamentals

3.1.1 Overview

In the previous chapters, we explored techniques to hack Wesnoth, including changing memory (such as gold) and changing code (such as recruiting units). However, all of these changes only persisted until we closed Wesnoth. To regain these hacks upon reopening the game, we would then have to repeat the initial process in a memory scanner or debugger.

This is both tedious and impossible to distribute to a larger audience. However, since we can now defeat DMA, we know that any memory we need to change is always in a static location. Because of this, we can create a set of instructions that contains the changes we wish to make. Creating this set of instructions in a way that a computer can understand is known as programming. By programming hacks, we can create programs that can be executed and will automatically change the memory we care about. We can also distribute these programs to other people who want to experiment with our hacks.

Programming, as a whole, is too large of a topic to comprehensively cover in these chapters. Instead, we will focus on the subset of programming that is relevant for creating hacks.

3.1.2 Programming Languages

In Chapter 1.1, we briefly covered programming languages. Programming languages allow code to be written in a human-readable form. This code is then translated down to instructions that a CPU can understand. There are many programming languages, and they can be broken down into roughly two categories: what they execute on, and how they execute.

Programming languages can create code that either executes directly on a CPU or executes through an interpreter. An interpreter works by dynamically translating the initial code into a form that the CPU can understand. These two types are known as

Playrix Ex. 1024, page 91

compiled languages and interpreted languages, respectively. Programming languages can also either execute instructions in order (top to bottom), or through the declaration and resolution of functions. These types are known as imperative and functional, respectively.

A language can be classified by applying those two modifiers. For example, C is a compiled, imperative language. Java is an interpreted, imperative language. Haskell is an example of a compiled, functional language. Interpreted languages can be compiled as well, often by bundling the interpreter and the initial instructions together.

There is no correct or best language. Some languages are better suited for different purposes, but all languages can achieve every purpose. However, when programming game hacks, we have several restrictions that limit our choice of language. The language we pick needs to support three main features:

- Direct access to the Windows API
- Modification of other applications' memory
- Loaded and executed on the CPU

All of these requirements will be explained later, but they basically exclude interpreted languages. In addition, languages that don't allow direct memory access, such as Java, are excluded.

3.1.3 C++

There are several languages that support the three criteria above. These include C and C#, as well as compiled versions of python. However, C++ offers the best combination of high-level language features (such as classes and strings) and low-level direct access to memory. This makes C++ ideal for programming game hacks.

C++ is a compiled, imperative language. It is a relatively difficult language to learn, but we will only need to understand a subset of its features to create game hacks. One of its most important features, for us, is the ability to create pointers that can directly modify memory addresses.

3.1.4 Pointers

Pointers are another complex topic that we will only cover briefly. Pointers are a type of variable that point to another section of memory. For example, take the following C++ code:

```
int x = 5;
int *y = &x;
```

In C++, a ***** represents a pointer declaration. The **&** returns the address of a variable. So, after executing this code, the variable **y** points to the variable **x**. Consider the following code:

```
*y = 6;
```

This code will dereference (or get the address it points at) **y** and then assign that value to **6**. After this code executes, the variable **x** will also be **6**, since this was the value that **y** points to.

Applying this to game hacking, let's say we find a gold value at `0x12345678` and this value is not dynamically allocated. If we were to load our C++ program into the game's address space, we could use a pointer to modify the value of the gold:

```
int *gold = (int*)0x12345678;
*gold = 999;
```

After executing, the gold value at `0x12345678` will now be set to 999. Pointers give us a large amount of control over a game's memory, but they can be hard to understand. We will explore them more in following chapters.

3.1.5 Types of Hacks

There are three main types of game hacks that can be programmed. These are:

- External executables
- Injected DLL's (dynamic-link libraries)
- Custom wrappers

Each of these has its own use-case. External executables are stand-alone programs that can be executed normally. These executables use functions built into Windows, known as Application Programming Interfaces (API's), to read and modify memory of another executable. By contrast, injected DLL's need to be loaded into the game's memory in some way. Once loaded, they execute within the memory of the game and can directly access the game's memory through pointers. Custom wrappers are used when creating hacks that target the game's drawing libraries, such as DirectX and OpenGL. By loading a custom version of these libraries that "wrap" the original functionality, we can cause the game's drawing logic to be altered.

Playrix Ex. 1024, page 94

3.2 External Memory Hack

3.2.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

3.2.2 Identify

In this chapter, we will create a C++ program that will modify a player's gold in Wesnoth.

3.2.3 Understand

In Chapter 2.8, we defeated DMA in Wesnoth and located the player's base pointer at `0x017EECB8`. We then determined the offsets necessary to locate the player's gold from the base pointer. This allowed us to start at a static address, add a series of static offsets, and reach a dynamic address.

Since these addresses and offsets are static, we can create a program to perform this operation. We covered several approaches to do this in Chapter 3.1. In this chapter, we will create an external executable.

3.2.4 Visual Studio

To create C++ programs, we need a compiler and a linker. A compiler is used to turn high-level language code into opcodes. A linker is used to then create an executable that the OS understands from these opcodes. These two components are normally bundled into an Integrated Development Environment, or IDE. IDE's contain other components as well, such as code-completion and interactive debugging. Visual Studio is an IDE that Microsoft has released for Windows. The community edition is free to

Playrix Ex. 1024, page 95

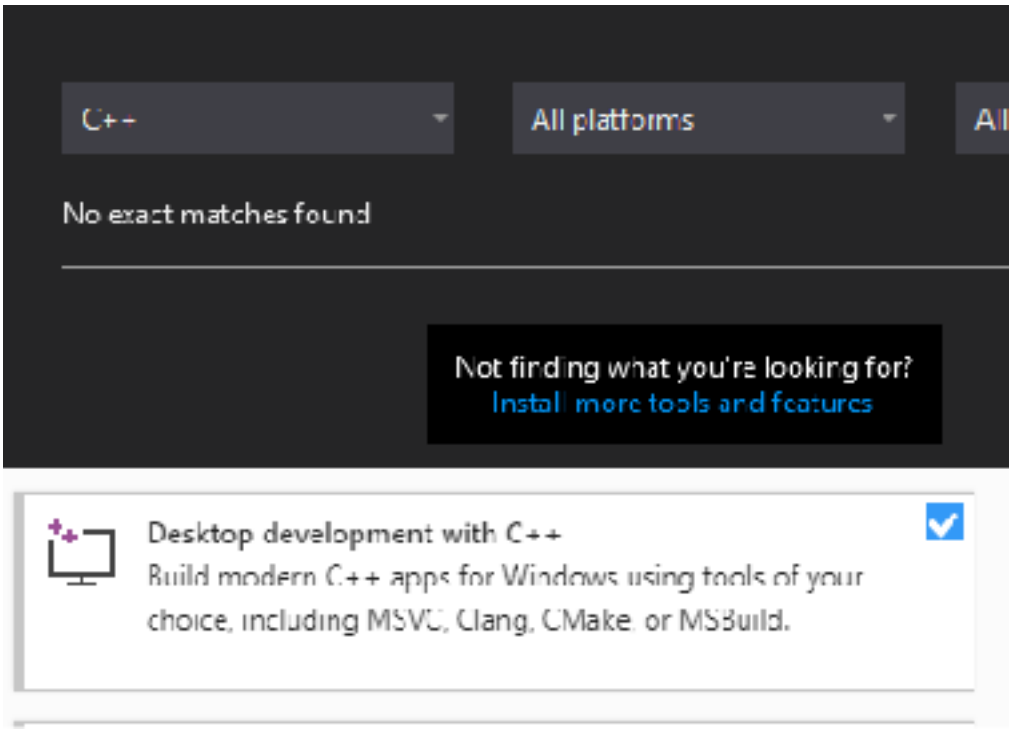
download and use in personal projects. It can be installed using Chocolatey, which we installed when first setting up the VM. To do so, open up Command Prompt or Powershell and run the following command:

```
choco install visualstudio2019community
```

In Visual Studio, source code files are contained within projects. Several of these projects can be contained within a solution. For example, the Visual Studio solution for Wesnoth might look like:

```
Game - Solution
  Engine - Project
    Player.cpp - Source Code
    main.cpp - Source Code
  UI - Project
    ...
  Network - Project
    …
```

To create a C++ solution, we will first need to install some C++ components. This can be done by selecting the *Install more tools and features* link, and then selecting *Desktop development with C++* in the wizard.

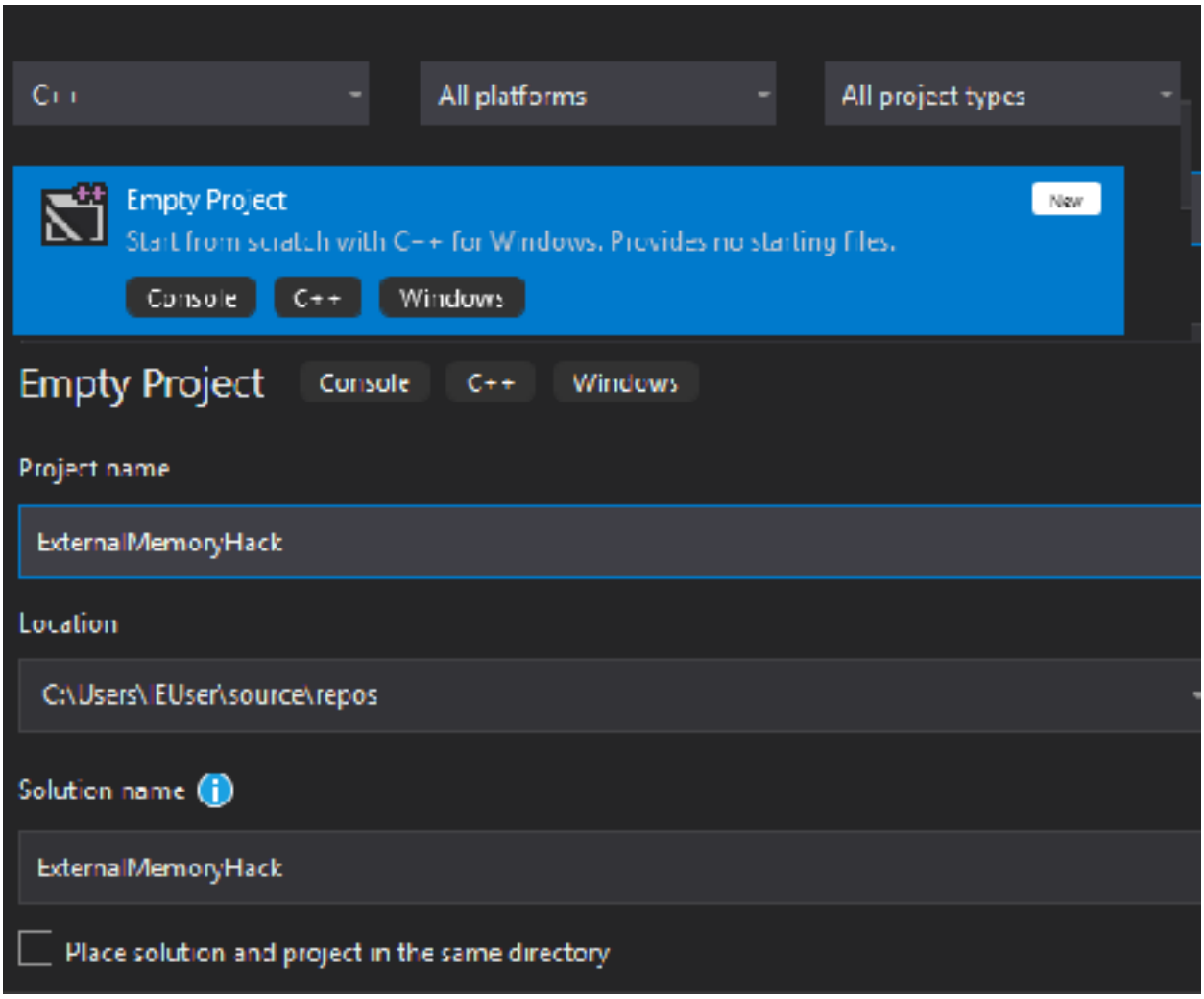


Playrix Ex. 1024, page 96

With these components installed, we can now create C++ projects and compile them.

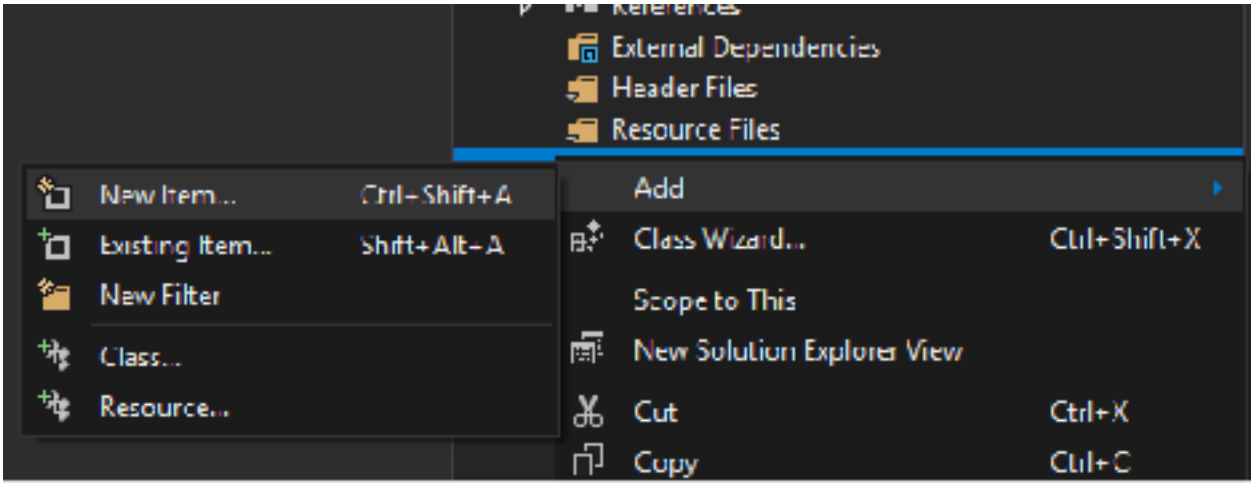
3.2.5 Creating Projects

Once these components are installed, create an empty C++ project and name it *ExternalMemoryHack*.

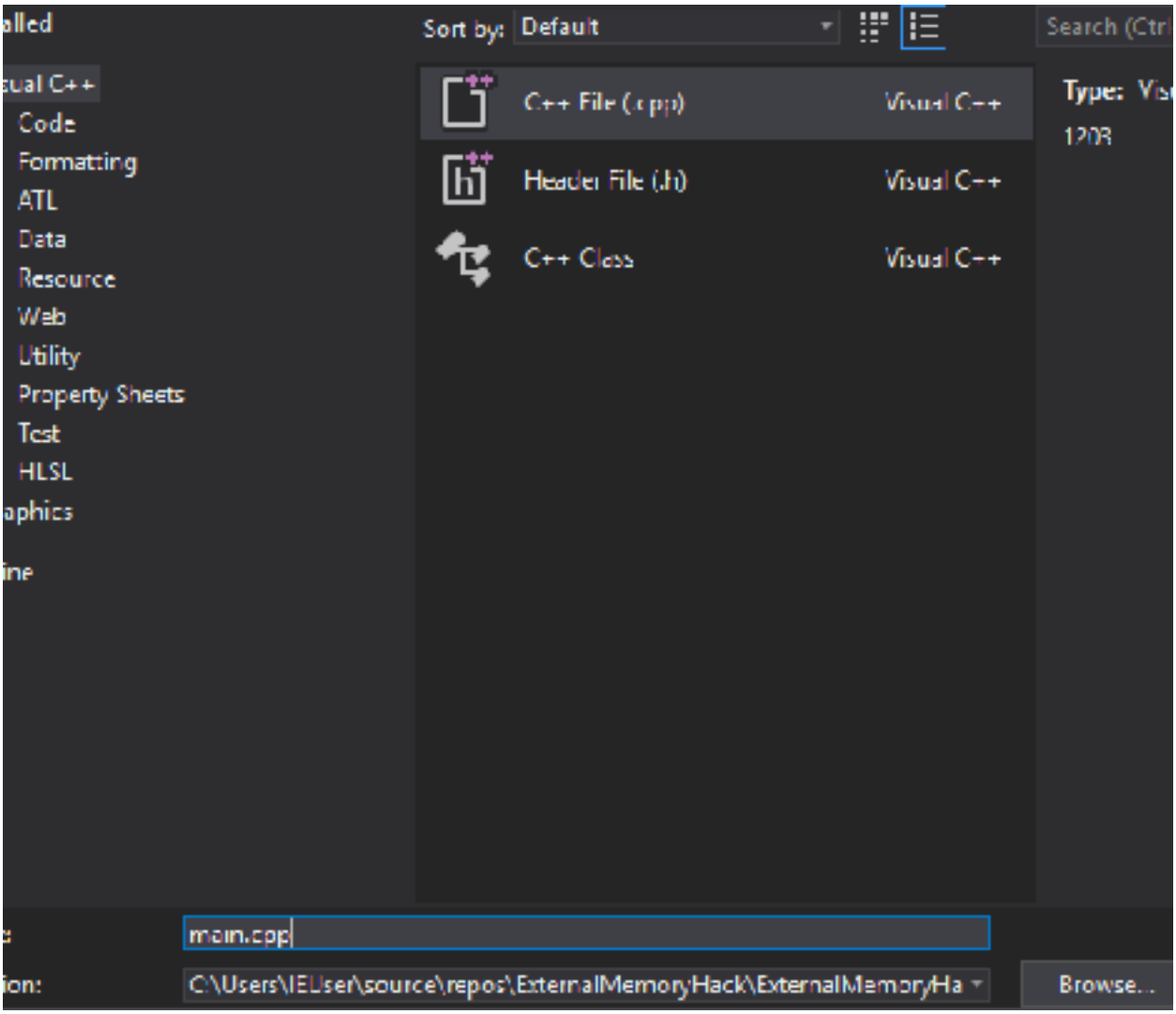


With the new project created, we can now add our source code file that will contain all of our code. To do this, right-click on *Source Files* and select *Add -> New Item*:

Playrix Ex. 1024, page 97



In the modal that appears, choose *C++ File* and name it *main.cpp*:



Playrix Ex. 1024, page 98

With the project and source file created, we can now begin programming.

3.2.6 C++ Basics

C++ has many features and is a versatile language. For our purposes, we will focus on the most important ones for creating hacks. For this chapter, we need to know two things about the language:

1. Programs start at a function called **main**. This function has to return a value.
2. Programs can call other functions built into Windows.

Any C++ executable needs to have a **main** function that returns an integer. This function takes two parameters, which are not important to know at the moment. When executed, the OS looks for and executes this function. When the function returns, it signals to the OS that the program is finished executing.

In addition to functions we create, we can call other functions that are built into Windows. Windows has many API's to do things like displaying text, playing sounds, and creating files. Windows also has API's that allow us to read and write values to an address in a process. These are what we will use to create our hack.

To use these functions, we need to include certain header files. Header files contain definitions for functions that are defined outside of our source file. To read and write memory, we will need to include the header file **Windows.h**.

3.2.7 Reading Values

We need to read several values in Wesnoth to locate our gold value. The API to read another process's memory is called **ReadProcessMemory**. If you google this name, the first result will be documentation by Microsoft. This documentation describes how the API works, including what parameters it takes and what values it returns.

By examining this documentation, we can determine what values we need to provide. For any values we still need, we can then determine how to get them.

ReadProcessMemory's function definition is:

```
BOOL ReadProcessMemory(
  HANDLE  hProcess,
  LPCVOID lpBaseAddress,
  LPVOID  lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesRead
);
```

Looking at the code block above, we can start at the first parameter and work our way down to determine what values we need. First, we do not have a handle to a process, so we will need to find that. We will discuss how to do this in the next section. We have the base address (in this case, our base pointer). The buffer needs to be provided by us, so we will need to create that. The size parameter will be the size of the data to read. In this case, the size will be 4 bytes, due to the size of the registers we saw while reversing. Finally, we will need to create another variable to hold the number of bytes actually read when the function is executed.

While this might seem overwhelming, this gives us a starting point to program from. First, we know that we need to include **Windows.h** so that we can use **ReadProcessMemory**:

```
#include <Windows.h>
```

Next, since this is a C++ executable, we will create our **main** function:

```
int main(int argc, char** argv) {

  return 0;
}
```

Finally, in the **main** function, above the **return** statement, we can insert our call to **ReadProcessMemory**. For values we don't have yet, we will put in a variable name. As we figure out how to retrieve these values, we can then assign these variables.

```
ReadProcessMemory(wesnoth_process, 0x017EECB8, gold_value, 4, bytes_read);
```


Playrix Ex. 1024, page 100

Both the **gold_value** and **bytes_read** values are provided by us and populated by the API. Therefore, we need to initialize two variables to hold these values. Since we are reading 4 bytes, these variables need to be large enough to hold amounts of this size. One option to accomplish this is to use a **DWORD**, which is 32 bits (or 4 bytes) long. We need to place these declarations above the call to **ReadProcessMemory**:

```
DWORD gold_value = 0;
DWORD bytes_read = 0;
```

Since both of these parameters are expected to be pointers, we need to also change our **ReadProcessMemory** call. Instead of passing the variable's value, we need to pass the address of these variables using **&**:

```
ReadProcessMemory(wesnoth_process, 0x017EECB8, &gold_value, 4, &bytes_read);
```

3.2.8 Opening Processes

Our next step is retrieving a process handle. To do this, we can use an API called OpenProcess. The definition for this API is:

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess,
  BOOL  bInheritHandle,
  DWORD dwProcessId
);
```

From this definition, we see that **OpenProcess** returns a handle to a process. This handle can be used as the first parameter for **ReadProcessMemory**. Looking at the documentation, we want our desired access to be **PROCESS_ALL_ACCESS**, so that we can both read and write to the process. The second parameter does not matter for what we are doing, so we will set it to the value of true. We will need to find the last parameter, so for now, we will create a variable. Since we need the result of this function to call **ReadProcessMemory**, we will place the call to it above **ReadProcessMemory**. Our code should now look like:

```
HANDLE wesnoth_process = OpenProcess(PROCESS_ALL_ACCESS, true, process_id);

DWORD gold_value = 0;
```


Playrix Ex. 1024, page 101

```
DWORD bytes_read = 0;
ReadProcessMemory(wesnoth_process, 0x017EECB8, &gold_value, 4, &bytes_read);
```

Next, we will retrieve a **process_id** for the OpenProcess call. Similar to the previous two API's, we will use another API and then fill in any information we need. In this case, the API will be GetWindowThreadProcessId. This API retrieves a process ID when provided with a window handle, which is different than a process handle. The definition for this API is:

```
DWORD GetWindowThreadProcessId(
  HWND    hWnd,
  LPDWORD lpdwProcessId
);
```

This function requires a handle to a window and a variable to hold the process ID. Just like before, we will add this code above our call to **OpenProcess**:

```
DWORD process_id = 0;
GetWindowThreadProcessId(wesnoth_window, &process_id);
```

To get a window handle, we can use the API FindWindow. This function takes the name of a window title and returns a handle to the window. The definition is:

```
HWND FindWindowA(
  LPCSTR lpClassName,
  LPCSTR lpWindowName
);
```

Since we want to search all windows, we will set the first parameter to **NULL**. For the second parameter, we know the name of the Wesnoth window, as it is displayed in the game's title bar. We can insert this final call at the top of our **main** function. Right now, our code will look like:

```
#include <Windows.h>

int main(int argc, char** argv) {
  HWND wesnoth_window = FindWindow(NULL, "The Battle for Wesnoth - 1.14.9");

  DWORD process_id = 0;
```


Playrix Ex. 1024, page 102

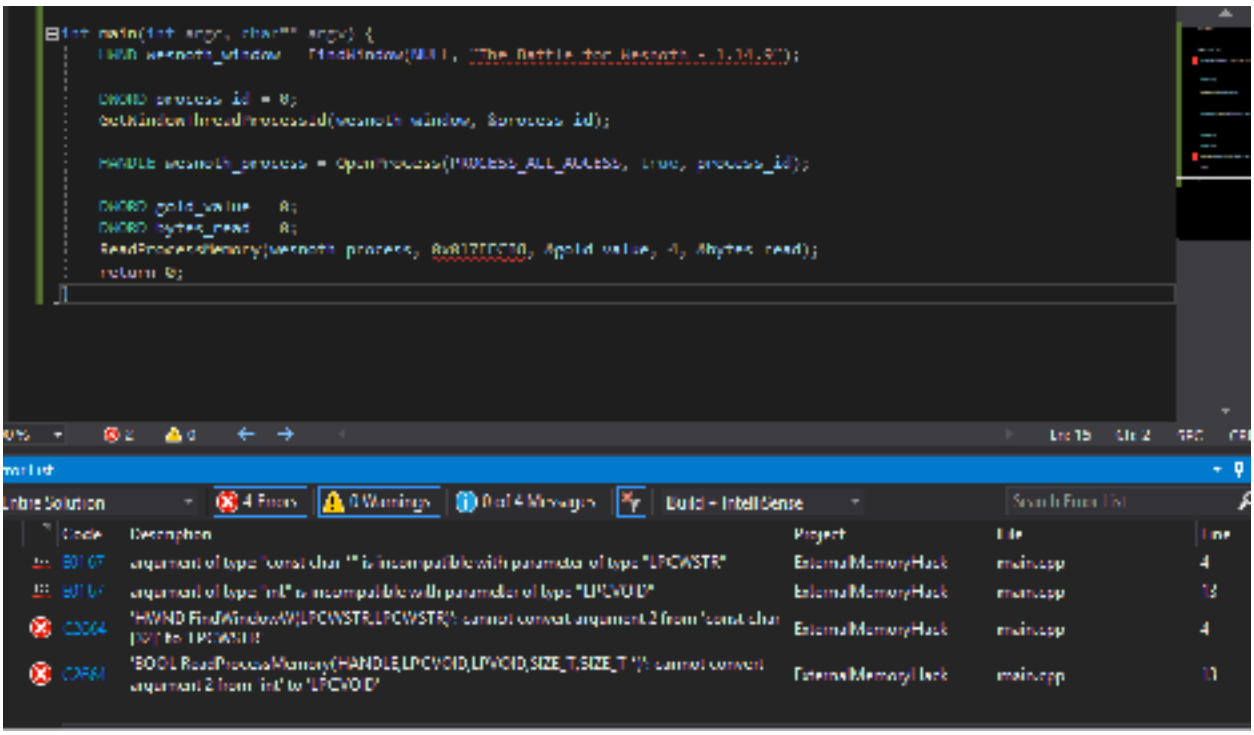
```
  GetWindowThreadProcessId(wesnoth_window, &process_id);

  HANDLE wesnoth_process = OpenProcess(PROCESS_ALL_ACCESS, true, process_id);

  DWORD gold_value = 0;
  DWORD bytes_read = 0;
  ReadProcessMemory(wesnoth_process, 0x017EECB8, &gold_value, 4,
&bytes_read);
  return 0;
}
```

3.2.9 Casting Parameters

Visual Studio will display several errors for the code we have written. This is because we have not properly casted two of our variables. As a result, the compiler cannot understand how we want to pass data to a function. If we want to compile our program, we will need to fix these errors. Luckily, by reading the *Error List* in Visual Studio, we can determine what we need to do to fix these errors.



Let's address the two **ReadProcessMemory** errors that occur on *Line 13* of the code. The first error for *Line 13* is *argument of type "int" is compatible with parameter type*

"LPCVOID". This indicates that we have a parameter that is an integer that is supposed to be a **LPCVOID**. The second error, *cannot convert argument 2 from 'int' to 'LPCVOID'*, indicates which parameter this is. *Argument 2* is our address `0x017EECB8`. If we google **LPCVOID**, the first result is Microsoft's documentation regarding LPCVOID. The documentation shows us that **LPCVOID** is defined as a **void***:

```
typedef const void* LPCVOID;
```

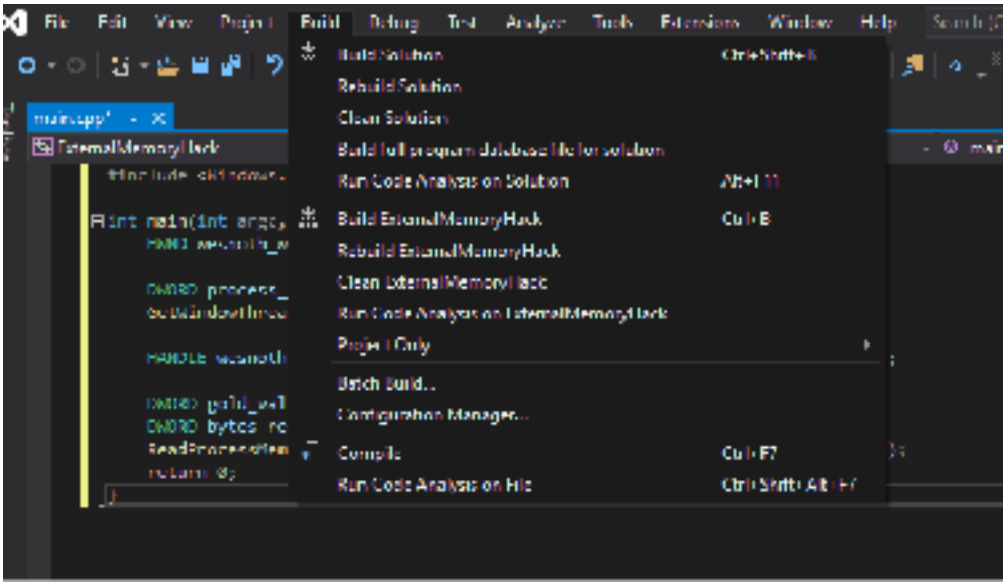
To solve these errors, we can cast our address as a **void***. The resulting code looks like:

```
ReadProcessMemory(wesnoth_process, (void*)0x017EECB8, &gold_value, 4,
&bytes_read);
```

With this, those errors will disappear. Now we can examine the second set of errors that occur on *Line 4*, regarding the **FindWindow** call. These errors indicate that the **The Battle for Wesnoth - 1.14.9** string is not cast correctly. It is a **const char*** and needs to be cast as a **LPCWSTR**. To do this, we can prefix the string with an **L**. Our **FindWindow** call now looks like:

```
HWND wesnoth_window = FindWindow(NULL, L"The Battle for Wesnoth - 1.14.9”);
```

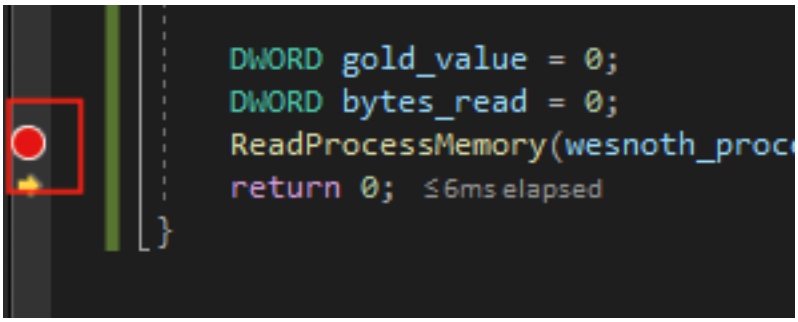
After making this change, the errors will disappear and we can now compile and execute our program. To do this, go to the *Build* menu item and select *Build Solution*. Once this completes, we will have a program that we can execute.



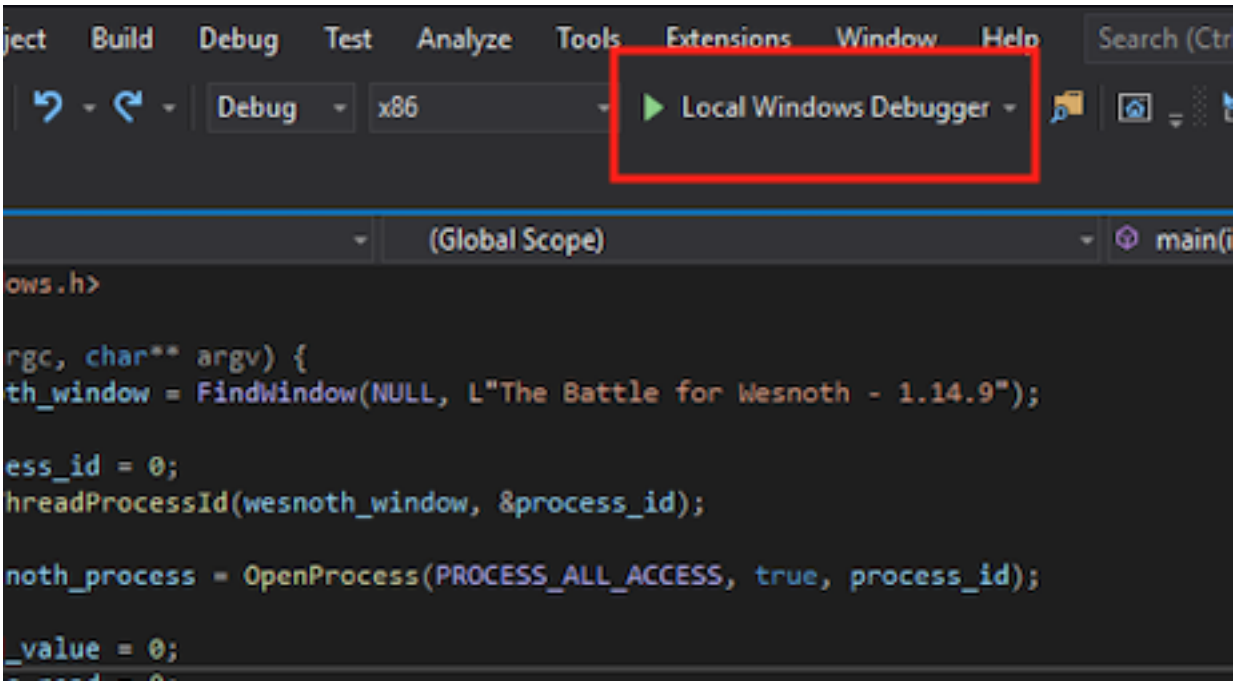
3.2.10 Debugging

To verify that this code works, we need to make sure that we are actually reading the correct value at the memory address `0x017EECB8`. To do this, we will debug our program inside Visual Studio and compare the results of our **ReadProcessMemory** call against Cheat Engine.

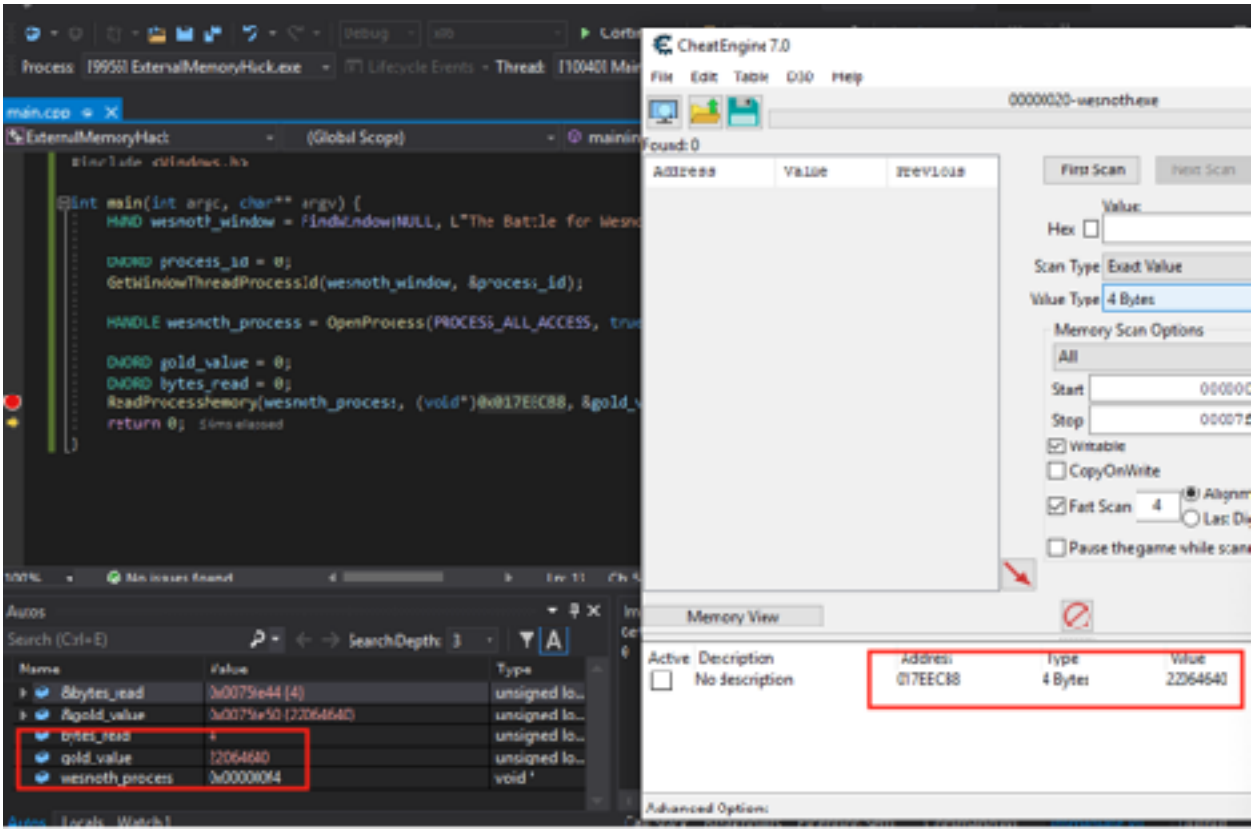
First, open up Cheat Engine and manually add the address `0x017EECB8`. Then, set a breakpoint on the **ReadProcessMemory** line. This can be done by left-clicking on the area to the left of the line of code you wish to breakpoint. If done correctly, a red circle will appear.



Next, click on the *Local Windows Debugger* button at the top of the IDE. This will begin executing our program with a debugger attached.



Since we have written the source code, debugging our program will be far easier than debugging Wesnoth. When our breakpoint is reached, the debugger will pop and let us explore various elements of the code, including our variables. These variables will be shown in the bottom left of the IDE. To make sure we are reading memory correctly, we want to look at the **gold_value** variable and make sure its value matches Cheat Engine.



Since the values match, we know that we are correctly reading memory in Wesnoth. Hit the *Continue* button at the top of the IDE to finish executing our program. Now we can move on to finding our gold value.

3.2.11 DMA

Since we can read memory, we can now retrieve our gold value. In Chapter 2.8, we determined that our gold address is stored at `[[0x017EECB8 + 0x60] + 0xA90] + 4`. This can be further simplified to `[[0x017EED18] + 0xA90] + 4`. To retrieve the gold address in our program, we can first read the value at `0x017EED18`, then add `0xA90` to

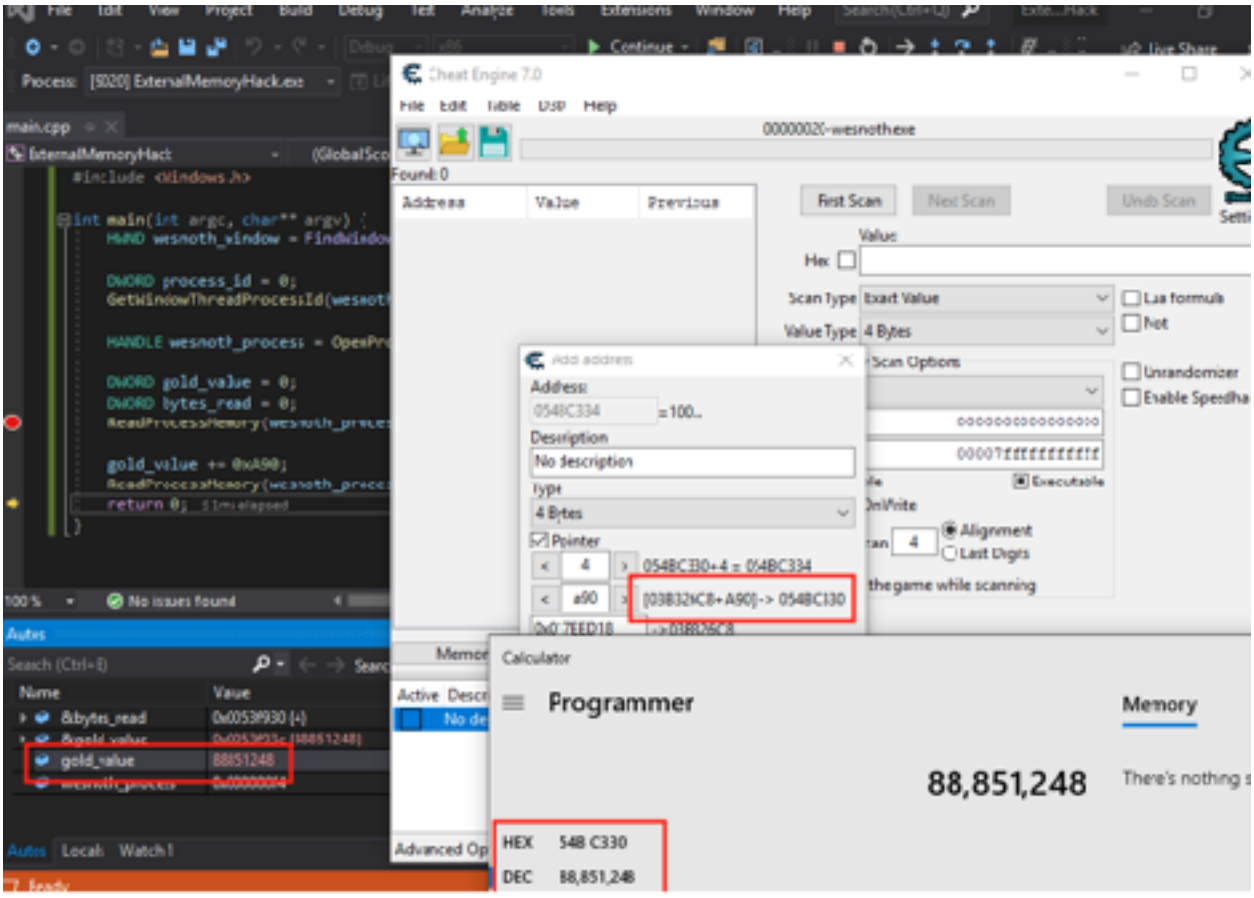
that value. We can then read this address and add 4 to it. Once we have done that, we will have our gold address.

To do this entire process, we can use the **ReadProcessMemory** call identically to our previous code. First, we will read in the value of `[[0x017EED18] + 0xA90]`.

```
DWORD gold_value = 0;
DWORD bytes_read = 0;
ReadProcessMemory(wesnoth_process, (void*)0x017EED18, &gold_value, 4,
&bytes_read);

gold_value += 0xA90;
ReadProcessMemory(wesnoth_process, (void*)gold_value, &gold_value, 4,
&bytes_read);
```

We can use Cheat Engine to examine offsets to ensure that we are reading the value correctly. We can then use a breakpoint on the second **ReadProcessMemory** call to ensure that the values match. Since Visual Studio displays variables in a decimal format, we will need to convert these numbers to hexadecimal to check.



Since our values match, we can add a final offset of 4 to the address to retrieve our gold address. Next, we will focus on writing memory.

```
gold_value += 4;
```

3.2.12 Writing Memory

The API to write to another process's memory is called [WriteProcessMemory.](#) Its definition is very similar to **ReadProcessMemory**:

```
BOOL WriteProcessMemory(
  HANDLE  hProcess,
  LPVOID  lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesWritten
);
```

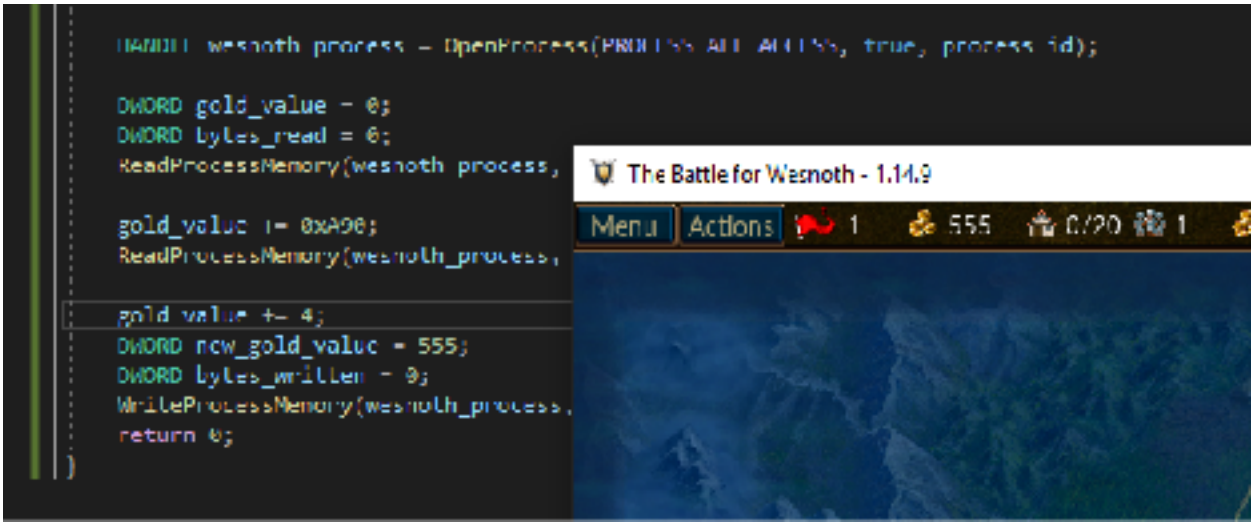
The major difference is that this function writes the value of a buffer into a section of a memory, instead of reading a section of memory into a buffer. Like before, we will need to declare two variables for the buffer and the number of bytes written.

```
DWORD new_gold_value = 555;
DWORD bytes_written = 0;
```

Then, we can call **WriteProcessMemory** in an almost identical manner to **ReadProcessMemory**. Like with **ReadProcessMemory**, we will cast our **gold_value** to **(void*)**:

```
WriteProcessMemory(wesnoth_process, (void*)gold_value, &new_gold_value, 4,
&bytes_written);
```

When this is executed, our gold will be set to 555 and our hack will be complete. We can now run this executable whenever we want to change our gold. We can also distribute it to other players to execute on their machines.



The full code for this chapter is available in Appendix A for comparison.

3.3 DLL Memory Hack

3.3.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

3.3.2 Identify

In this chapter, we will create a dynamic-link library (DLL) that will modify the player's gold in Wesnoth. This DLL will modify the player's gold every time the user presses a certain key.

3.3.3 Understand

In the previous chapter, we created an external C++ program that used **ReadProcessMemory** and **WriteProcessMemory** to modify the player's gold. While these API's are useful, they also have several limitations. Due to their definitions, they require us to cast parameters into a defined type. Their definitions make it easy to modify simple values like gold, but they make it difficult to read and write full classes or complex data types.

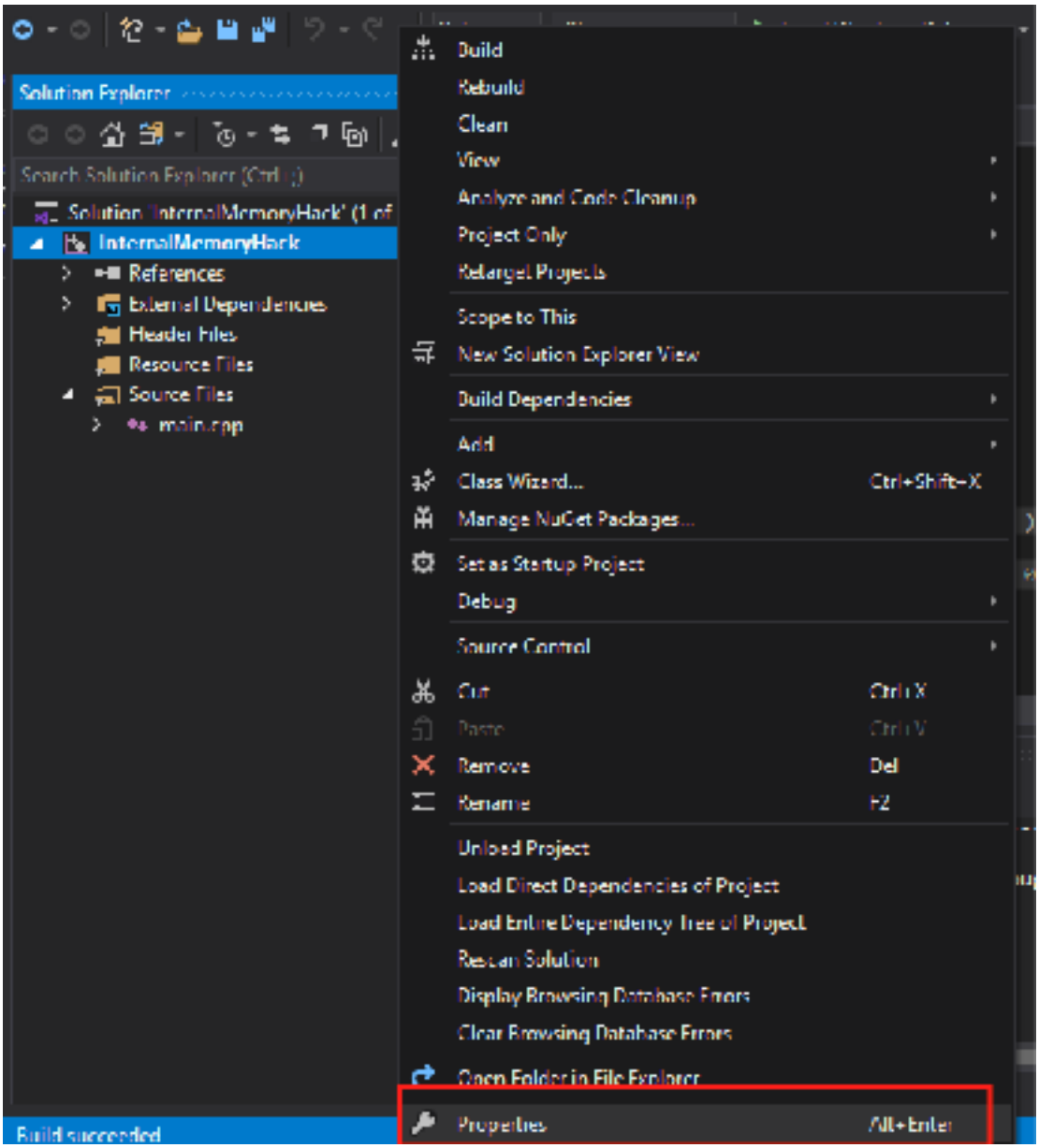
Since these API's are executed from an external program, we would struggle to do things like listen to key presses from the game. In addition, if we wanted to create a code cave in the game, we would have to manually convert that code cave into its opcode representation. We would then need to find a memory location to place it at for **WriteProcessMemory** to work.

To bypass all of these limitations, we can instead inject a DLL into Wesnoth. Once injected, this DLL will be loaded into the game and can directly access the game's memory through the use of pointers. We can also create threads that execute inside the game, allowing us to listen for user input and other events.

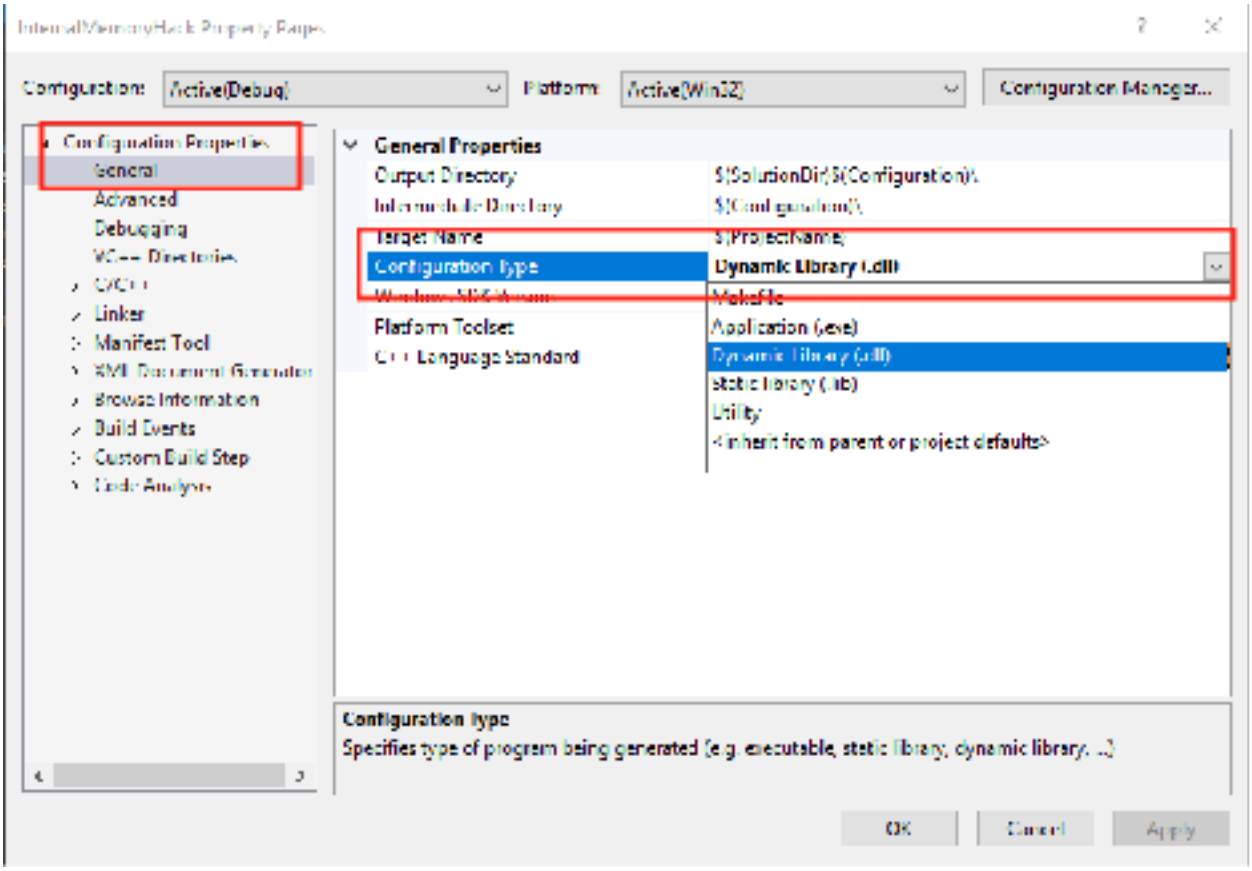
3.3.4 Creating DLL's

First, create an empty project, this time named *InternalMemoryHack*. After the project is created, add a *main.cpp* file. The process to do these steps is identical to the previous chapter.

By default, empty projects in Visual Studios are set to build as executables. To build a DLL, we will need to change the project's *Configuration Type*. This can be done in the project's preferences. First, right-click on the project's name and select the *Properties* menu item.



Next, under *Configuration Properties*, choose *General*. Then, change the *Configuration Type* from *Application* to *Dynamic Library*. Choose *Apply* and then hit *OK* to close the modal.



Our project will now be built as a DLL instead of an executable.

3.3.5 DLL Basics

DLL's cannot be executed by themselves. Instead, they need to be loaded into an executable. DLL's allow developers to create libraries of functions that can be loaded dynamically. These libraries can then be used across several executables and reduce the amount of code that developers need to write.

For example, user32.dll contains code that displays modals, alerts, and other Windows UI elements. Most executables released for Windows load this DLL automatically and gain access to this functionality without needing the original code. If Microsoft updates this code and changes how an alert box looks, all executables that load this library will benefit from this change.

Playrix Ex. 1024, page 112

DLL's have several differences from normal executables. For our purposes, we need to know three of them:

1. DLL's have a **DllMain** function instead of a **main** function.
2. This **DllMain** function is called when a process loads or unloads a DLL.
3. DLL's run inside their parent process. Variables declared in DLL's are created in the parent's memory.

The **DllMain** function has different parameters from a **main** function. Its definition is:

```
BOOL WINAPI DllMain(
    _In_ HINSTANCE hinstDLL,
    _In_ DWORD     fdwReason,
    _In_ LPVOID    lpvReserved
  );
```

The **fdwReason** parameter contains the reason that the **DllMain** function was called. For example, when the DLL is loaded into a process, this parameter will hold the value of 1. This value is also defined by the constant **DLL_PROCESS_ATTACH**. To ensure that our code only executes once, we will check this parameter in our final hack.

Since DLL's execute in another process's memory, we will need to load them in some manner. In hacking, this is often known as injecting, as we are falsely loading our DLL into a process. It can often be hard to detect if a DLL has injected successfully. One approach is to attach a debugger to a process and observe all of the process's loaded modules. However, this approach can be time-consuming and is not always feasible. Another approach is to create a DLL that will display an obvious indicator when it is injected. This is the approach we will use to test our DLL injection.

3.3.6 MessageBox

The Windows API has a function to display a message box in a process. The definition for this function is:

```
int MessageBox(
    HWND    hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT    uType
  );
```


Playrix Ex. 1024, page 113

However, due to how C++ handles parameter casting, we can ignore the types for these values. By calling the **MessageBox** function like below, we will display a blank message box with an *Error* title and no text.

```
MessageBox(0,0,0,0);
```

We can use this behavior to ensure that our DLL is injected successfully into Wesnoth. In main.cpp, add the following code:

```
#include <Windows.h>

BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpvReserved ) {
  MessageBox(0,0,0,0);

  return true;
}
```

This code will make our DLL display a message box inside the parent process whenever the DLL is loaded or unloaded. We will use this behavior to ensure that our DLL is being injected successfully. Build this code using the *Build* option to produce a DLL. This DLL will be placed in the location you specified when creating the project. By default on our lab machine, this will be *C:\Users\IEUser\source\repos\InternalMemoryHack\Debug\InternalMemoryHack.dll*.

3.3.7 Injecting DLL's

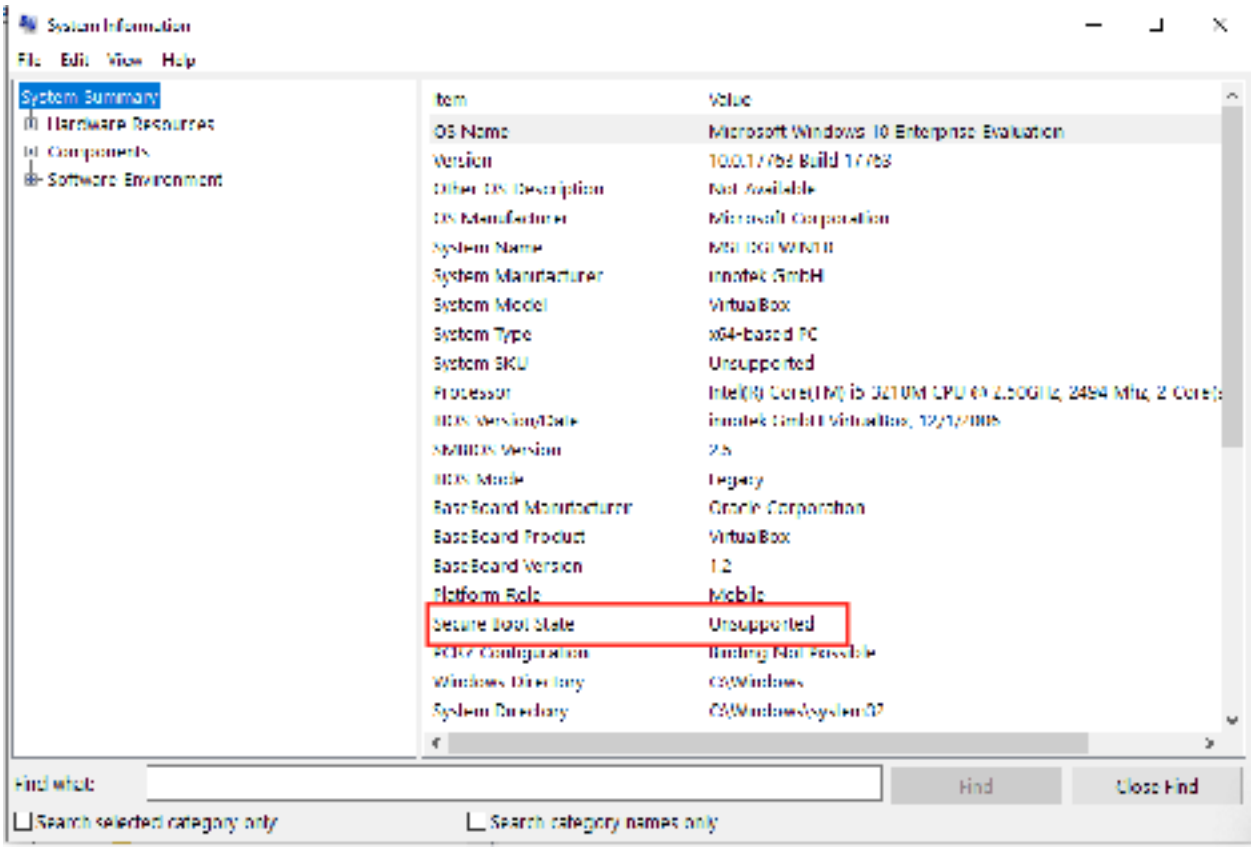
DLL's are normally loaded into a process through the use of the **LoadLibrary** API. However, since we are not modifying the original source code of the game, we will need to find another way to load our DLL into Wesnoth.

One approach we can use is a DLL injector. DLL injectors are external programs that create a thread inside the target process. This is done through the use of the API **CreateRemoteThread**. This thread then calls the **LoadLibrary** API inside the process. In Chapter 7.1, we will cover how to create a DLL injector.

For this chapter, we will use a feature of Windows that will inject user-defined DLL's into every executable that is started. This feature is called AppInit_DLLs and can be controlled via the registry.

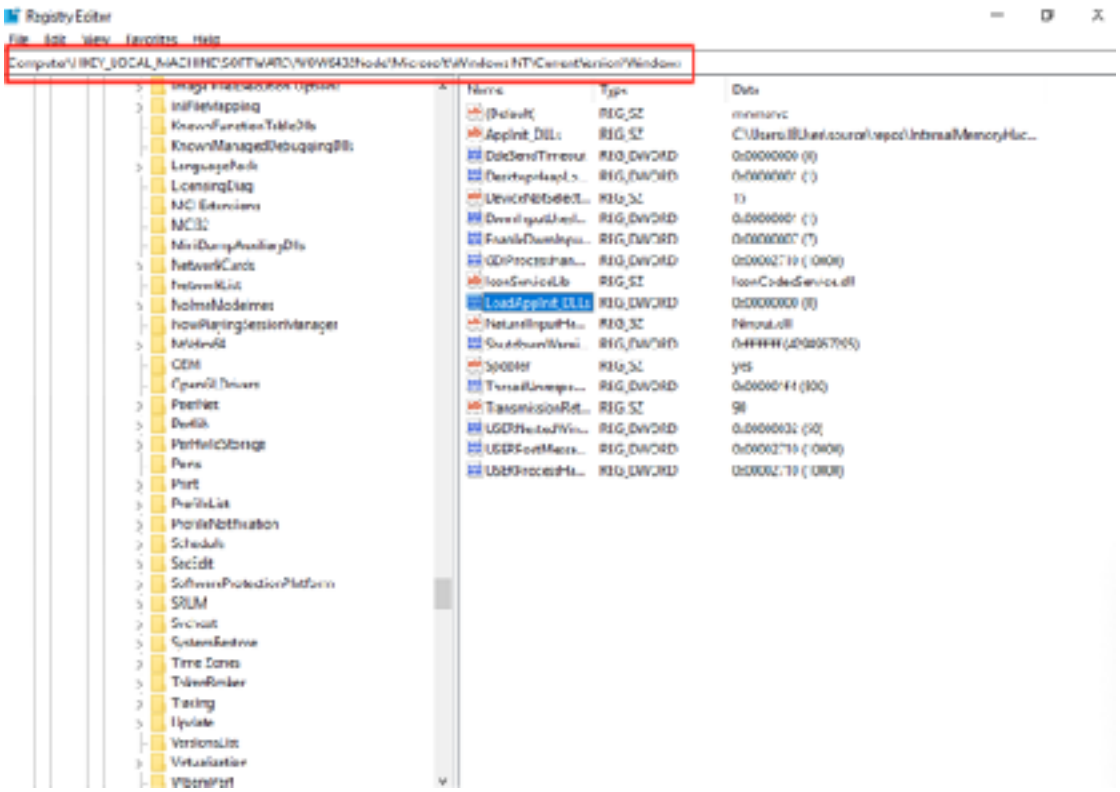
Playrix Ex. 1024, page 114

Since this feature is often used by malware, Windows 10 requires *Secure Boot* to be disabled for the feature to work. By default, VirtualBox does not support this feature and it will be disabled. If you are using actual hardware, you will need to disable it through the BIOS. Its current state can be determined through the *System Information* program:

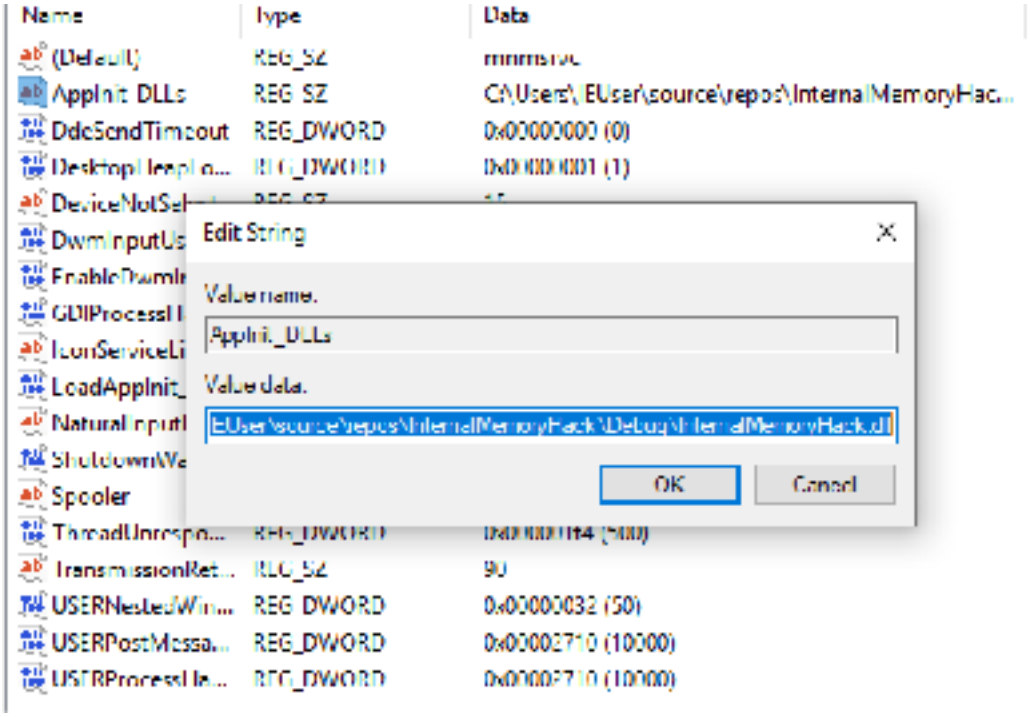


Once *Secure Boot* is disabled, we need to modify the registry to enable AppInit_DLLs. This can be done by first opening up the regedit program. The Windows registry contains keys and values that change OS and individual program functionality. It is similar to the file system on Windows in that these keys and values are contained in paths. The path for the AppInit_DLL feature on 64-bit Windows computers is *Computer\HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Windows NT\CurrentVersion\Windows*. Navigate to this location in regedit.

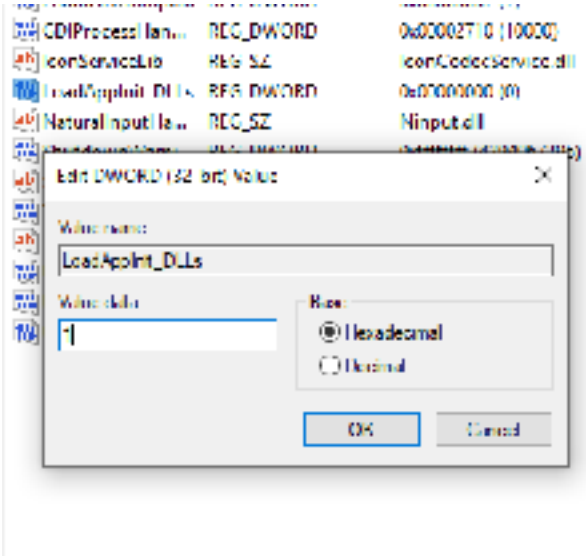
Playrix Ex. 1024, page 115



AppInit_DLLs will load any DLL's specified in the AppInit_DLLs value into all started programs. Double-click on the *AppInit_DLLs* value and change the string to the location of our DLL:



Next, we need to enable the feature by changing the value of *LoadAppInit_DLLs*. After making this change, our DLL will be loaded into every new process. When the value is set at 1, this feature will be enabled. When it is 0, this feature will be disabled.

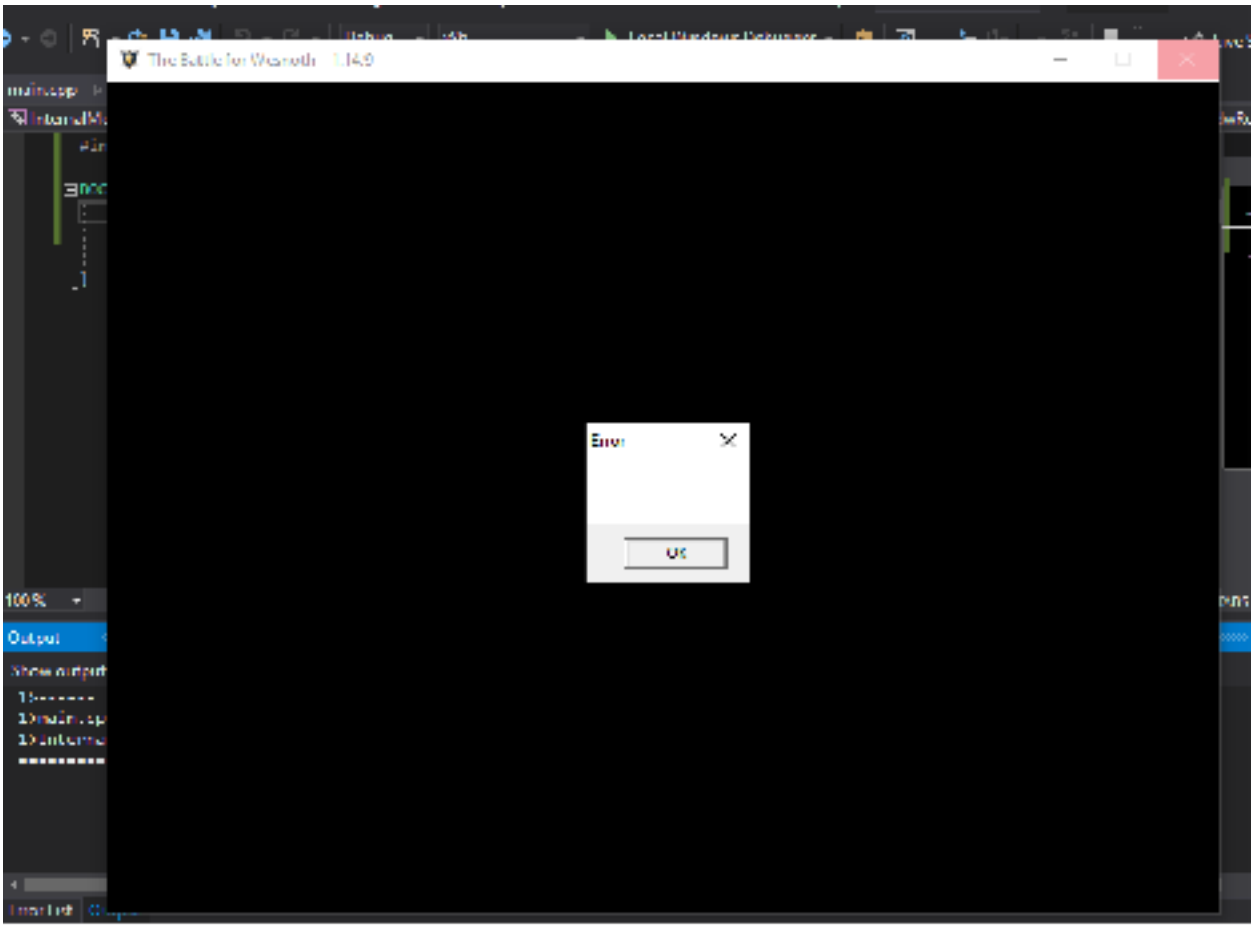


After these changes, the registry key should look like:

Name	Type	Data
(Default)	REG_SZ	mnmsrvc
AppInit_DLLs	REG_SZ	C:\Users\IEUser\source\repos\ nternalMemoryHac...
DdeSendTimeout	REG_DWORD	0x00000000 (0)
DesktopHeapLo...	REG_DWORD	0x00000001 (1)
DeviceNotSelect...	REG_SZ	15
DwmInputUsesI...	REG_DWORD	0x00000001 (1)
EnableDwmInpu...	REG_DWORD	0x00000007 (7)
GDIProcessHan...	REG_DWORD	0x00002710 (10000)
IconServiceLib	REG_SZ	IconCodecService.dll
LoadAppInit_DLLs	REG_DWORD	0x00000001 (1)
NaturalInputHa...	REG_SZ	Ninput.dll
ShutdownWarni...	REG_DWORD	0xffffffff (4294967295)
Spooler	REG_SZ	yes
ThreadUnrespo...	REG_DWORD	0x000001f4 (500)
TransmissionRet...	REG_SZ	90
USERNestedWin...	REG_DWORD	0x00000032 (50)
USERPostMessa...	REG_DWORD	0x00002710 (10000)
USERProcessHa...	REG_DWORD	0x00002710 (10000)

Playrix Ex. 1024, page 117

We can now start Wesnoth. Upon starting the game, several message boxes should appear, indicating that our DLL was injected successfully and is being both loaded and unloaded.



One important thing to remember is that AppInit_DLL will inject DLL's into every started process. This includes the process spawned to build our DLL as we make changes. To avoid any issues, we will have to disable this feature when we build our DLL. Make sure, after testing the DLL, to set the value of LoadAppInit_DLLs to 0. After building our DLL, set this value back to 1 to re-enable DLL injection.

3.3.8 Creating Threads

Now that we have verified that DLL injection is working, we can start programming our hack. We want this DLL to wait for a user to press a key before changing the gold. To do this, we will create a thread in the Wesnoth process. This thread will run until the game is exited.

First, we will change our **DllMain** to only execute our code when our DLL is first loaded into the process. This will ensure that we only create one thread in the game. We can do this by checking the **fdwReason** parameter:

```
BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpvReserved ) {
  if (fdwReason == DLL_PROCESS_ATTACH) {
    // Code to execute when the process is loaded
  }

  return true;
}
```

To create threads in a process, we can use the CreateThread API. Its definition is:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  SIZE_T                  dwStackSize,
  LPTHREAD_START_ROUTINE  lpStartAddress,
  __drv_aliasesMem LPVOID lpParameter,
  DWORD                   dwCreationFlags,
  LPDWORD                 lpThreadId
);
```

Since we are creating a thread within Wesnoth with no special attributes, we can ignore most of these parameters. The only parameter we are concerned with is **lpStartAddress**, which represents the function we want to execute when the thread is started. Because this function does not need to return, we will create it as a **void** function.

```
void injected_thread() {

}

BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpvReserved ) {
  if (fdwReason == DLL_PROCESS_ATTACH) {
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread, NULL, 0,
NULL);
  }

  return true;
```

```
}
```

When loaded, this code will create a thread that will execute the **injected_thread** function and then exit. To ensure that our thread remains active, we will use an infinite **while** loop in our **injected_thread** function:

```
while (true) {

  Sleep(1);
}
```

while loops will execute until their condition is false. Since true can never equal false, this while loop will run until our thread is exited by the closure of the game. To prevent our thread from causing slowdowns, we can use the **Sleep** API to pause its execution for a millisecond.

3.3.9 Detecting Key Presses

To detect a keypress, we can use the GetAsyncKeyState API. This takes a single parameter, which is the key to check for. If the key is down, it will return true. Otherwise, it will return false. For this chapter, we will check for the user to press **M**:

```
while (true) {
  if (GetAsyncKeyState('M')) {
    // Change the player's gold
  }

  Sleep(1);
}
```

One important caveat about **GetAsyncKeyState** is that it will constantly return true if the key is held down. This will not affect us in this chapter, but if we want to toggle a value off and on in the future, we will need to account for this behavior.

3.3.10 Pointers

In Chapter 3.1, we discussed pointers. Since our DLL is injected into Wesnoth, we can access memory in the game through the use of pointers. This allows us to bypass

ReadProcessMemory and **WriteProcessMemory**. However, we will still use the same offsets and addresses that we used in the previous chapter.

First, we will get the player's base address by reading the value at `0x017EED18`:

```
DWORD* player_base = (DWORD*)0x017EED18;
```

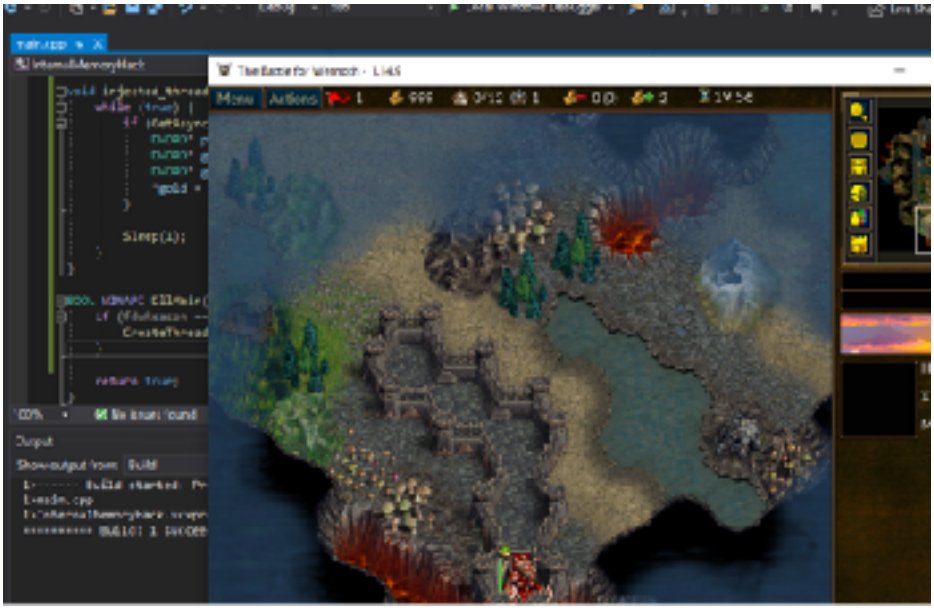
This will declare **player_base** as a pointer to a **DWORD** value. The location it will point at is our player's base address at `0x017EED18`. We can then dereference this pointer to "read" or retrieve this value. Using this, we can get our game base address by adding an offset:

```
DWORD* game_base = (DWORD*)(*player_base + 0xA90);
```

Finally, we can dereference the **game_base** address and add an offset to retrieve our gold value. We can then dereference this gold value and set its value directly:

```
DWORD* gold = (DWORD*)(*game_base + 4);
*gold = 999;
```

After building the DLL and re-enabling LoadAppInit_DLLs, we can inject this hack into Wesnoth. Create a game and then hit the "M" key. After you move your camera, the gold value will be updated to our new value. The full code for comparison is available in Appendix A.



3.4 Code Caves & DLL's

3.4.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

3.4.2 Identify

Our goal in this chapter is to create a code cave inside a DLL. The code cave will be executed whenever we select *Terrain Description*. The code cave will give us 888 gold before bringing up the terrain description box.

3.4.3 Understand

In Chapter 2.6, we created a code cave in the game's memory. We then adjusted the opcodes in the *Terrain Description* feature to **jmp** to this code cave. We used x64dbg's built-in instruction assembler to create the code cave and adjust these opcodes.

To create this behavior inside a DLL, we will first need to create a code cave in our DLL. We will then need to modify the opcodes in the *Terrain Description* feature to jump to this code cave inside our DLL.

3.4.4 Assembly in C++

One feature of C++ is the ability to insert assembly code into a C++ source file. This assembly will not be modified during the compiling steps. To do this, you use the **__asm** keyword. For example, the following code can be used to execute the instruction **pushad** in a C++ source file:

```
__asm {
  pushad
```

```
}
```

You can also mix C++ and assembly in a function. For example, the following code will save all registers, create a variable **x**, add 1 to it, and then restore all registers:

```
__asm {
  pushad
}
int x = 0;
x = x + 1;
__asm {
  popad
}
```

Finally, variables declared in C++ can be referenced in these assembly blocks. We will use this behavior later when programming our hack.

3.4.5 Assembled Functions

To jump to our code cave from Wesnoth's code, we will need to know our code cave's location. The easiest way to accomplish this in C++ is to declare our code cave as a function. We can then use the **&** operator on it to retrieve its address, identical to other variables. The pseudo code for this might look like:

```
void codecave() {
  //our code cave
}
...
terrain_description_jump_location = &codecave;
```

However, when assembled, functions are normally created with stack frames. Stack frames allow the compiler to easily offset and compute the location of local variables and function arguments. We will discuss this behavior more in future chapters as we explore the stack. For this chapter, we need to know that the **codecave** function above will be assembled into:

```
codecave:
  push ebp
  mov ebp, esp
```

```
...
mov esp, ebp
pop ebp
ret
```

These extra instructions can cause our code cave to corrupt the game when we jump to it. This corruption can then cause the game to crash. To avoid this behavior, we will use the **__declspec** C++ keyword to modify how the function is assembled. When using this keyword with the **naked** attribute, the compiler will not add a stack frame.

3.4.6 Cave Skeleton

Now, we can move on to creating our code cave. First, create a DLL in Visual Studio identically to how we have done it in previous chapters. The name for this project will be *CodeCaveDLL*.

After creating the DLL, we can add our code cave function. Like we discussed above, the function will use the **__declspec** keyword to avoid the compiler adding a stack frame. Its definition will look like:

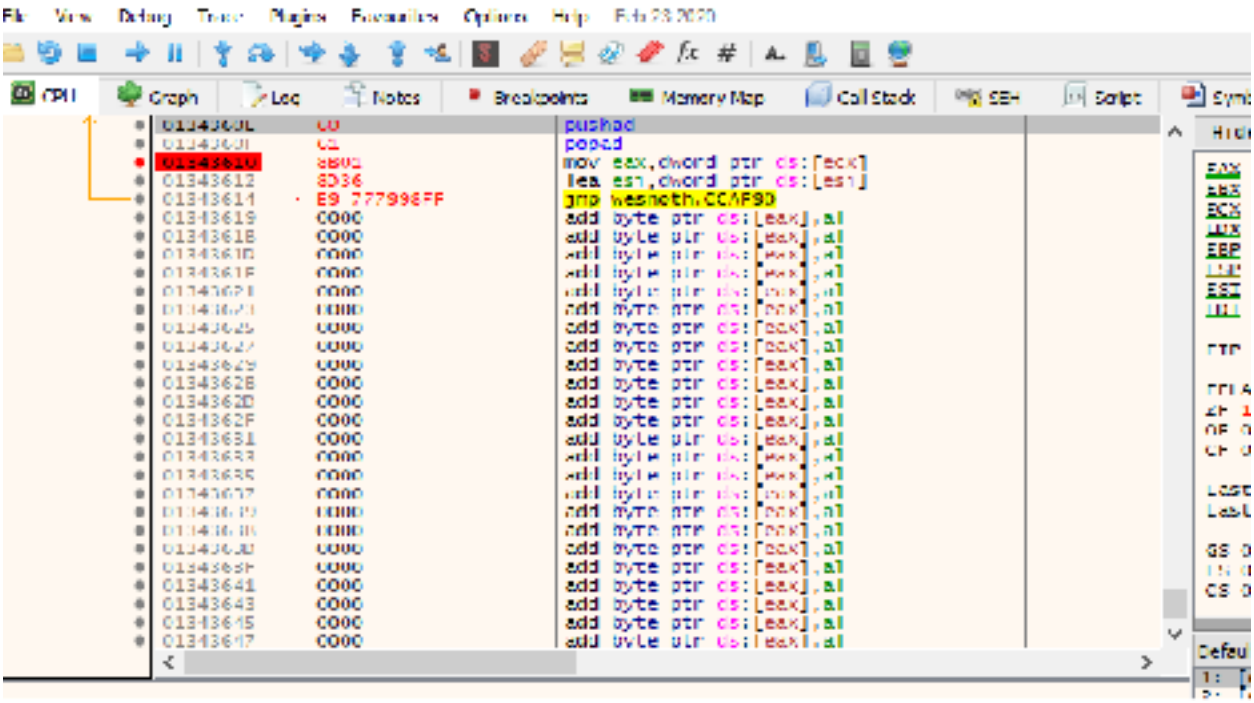
```
__declspec(naked) void codecave() {

}
```

As we discussed in Chapter 2.6, the first step when creating a code cave is to save and restore the registers and then restore the overwritten instructions. We identified these instructions in Chapter 2.6.

```
pushad
popad
mov eax, dword ptr ds:[ecx]
lea esi, dword ptr ds:[esi]
jmp 0xCCAF90
```

When this code cave was created in x64dbg, it looked like:



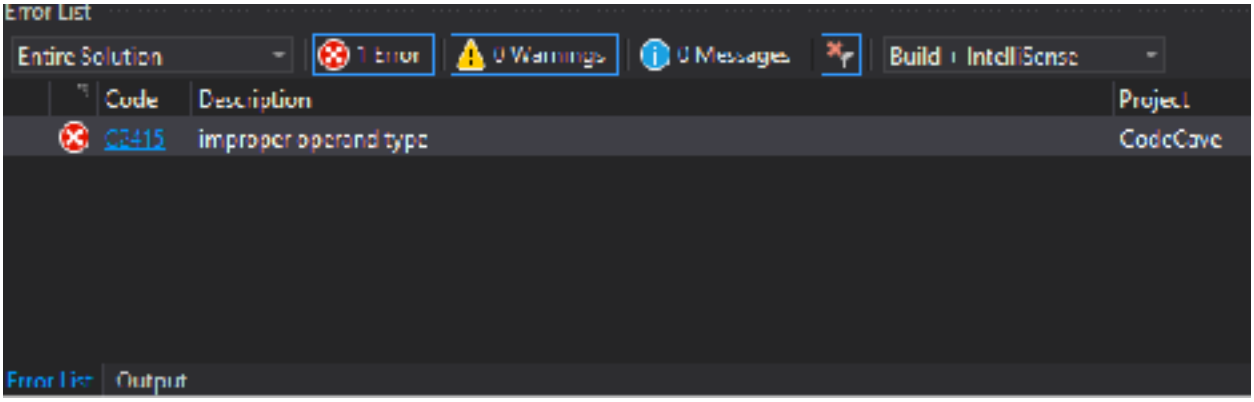
In our DLL, we will create two separate blocks of assembly instructions. The first block will save all of the registers. The second block will restore the registers and then execute the original instructions we have overwritten. Between these two blocks, we will place C++ code to modify our player's gold.

```
__asm {
  pushad
}

// code to modify gold

__asm {
  popad
  mov eax, dword ptr ds:[ecx]
  lea esi,dword ptr ds:[esi]
  jmp 0xCCAF90
}
```

If you attempt to compile this code, you will get an error on the **jmp** instruction:



This is because the compiler cannot resolve the **jmp** instruction when a static address is provided. There are many types of **jmp** instructions which differ based on the length of the jump. Without this knowledge, the compiler does not know how to encode the instruction. There are several ways to resolve this ambiguity, the easiest of which is to create a variable. That is what we will do in this chapter.

Since our code cave has no stack frame, we cannot declare variables inside of it. To bypass this, we will declare all of our variables globally, right below the include statements. Since we need to hold a static address value, we will declare the address as a **DWORD**:

```
#include <Windows.h>

DWORD ret_address = 0xCCAF90;

__declspec(naked) void codecave() {
  __asm {
    pushad
  }

  // code to modify gold

  __asm {
    popad
    mov eax, dword ptr ds:[ecx]
    lea esi,dword ptr ds:[esi]
    jmp ret_address
  }
}
```

3.4.7 Changing Gold

With our code cave function created, we can now use the same approach discussed in Chapter 3.3 to modify the dynamic address of our gold through the use of several pointers.

As we discussed in the previous section, we will place this code between the two assembly blocks so that our code cave properly saves and restores all the game's registers. The code to change our gold will be mostly identical to the previous chapter. The only difference is that we will have to initially declare our variables globally outside of our code cave function:

```
DWORD* player_base;
DWORD* game_base;
DWORD* gold;
...
__declspec(naked) void codecave() {
  __asm {
    pushad
  }

  player_base = (DWORD*)0x017EED18;
  game_base = (DWORD*)(*player_base + 0xA90);
  gold = (DWORD*)(*game_base + 4);
  *gold = 888;

  __asm {
  …
```

3.4.8 Redirection

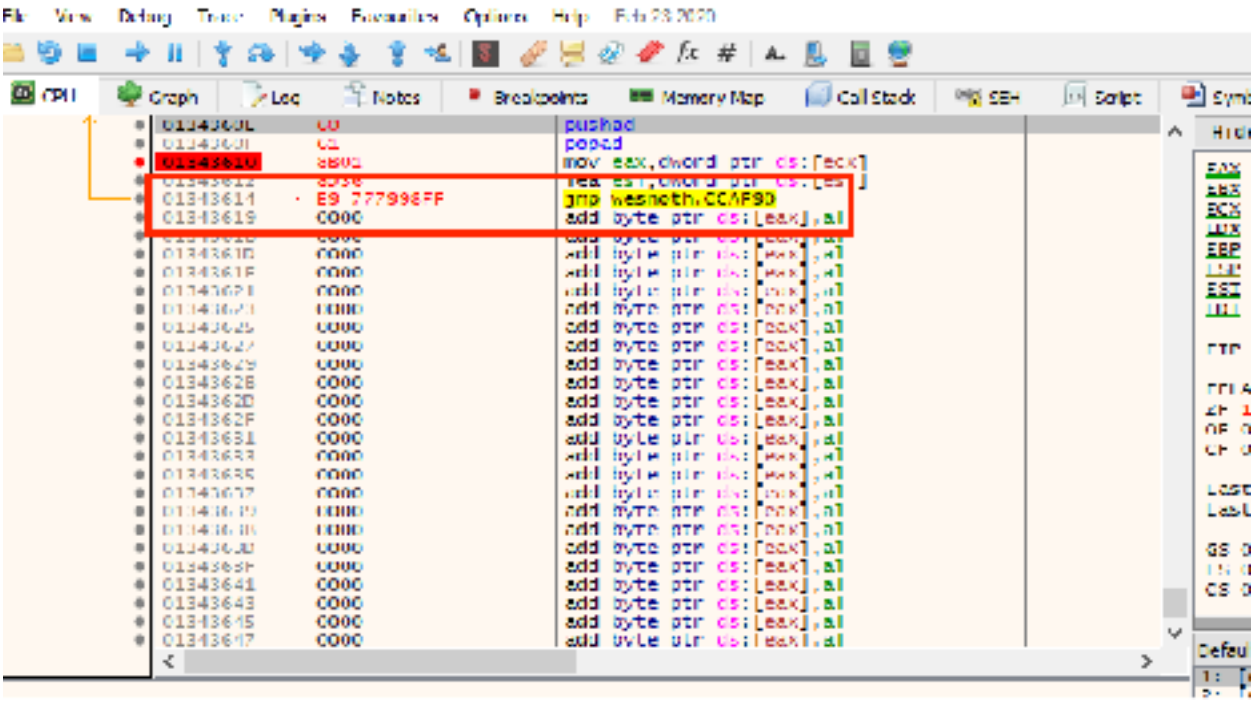
Next, we can work on redirecting the game's code to call this function. To do this, we will again use a pointer. However, this time we will declare the pointer to point to the address of the game's code responsible for displaying the *Terrain Description* feature. This will be the same hooking location we found in the previous code cave chapter at `0x00CCAF8A`.

We need to take a slightly different approach to modify the game's code using a pointer. Since we want to modify individual bytes, we will declare our pointer as an **unsigned char** (short for character). Unlike a **DWORD**, an **unsigned char** represents 1

byte of data. Declaring our pointer like this will give us the flexibility to modify individual bytes.

Before we can modify the game's code, we will need to change its protection type. Code is only intended to be executed, so Windows will, by default, not allow other processes or DLL's to write data to code addresses. To change this protection, we will use the API VirtualProtect and reassign the protection type.

Finally, we need to understand how the **jmp** opcode is structured. We know that **jmp**'s start with the opcode value of `0xE9`. However, there are an additional 4 bytes after this `0xE9`. These additional bytes direct the CPU where to jump to. These 4 bytes are not simply the new address. We can see an example from our previous code cave that we created with x64dbg:



There are several resources online that describe how this opcode is structured. The basic formula is:

```
new_location - original_location + 5
```

Let's verify this formula with the code cave above:

```
0xCCAF90 - 0x1343614 + 5 = FF 98 79 77
```

This initially looks incorrect. However, bytes are stored in a "reverse" order on all Windows-compatible CPU's. This is a concept known as endianness, which we will cover more in future chapters. If we reverse the byte order from the value we found above, we find that it matches the opcode in x64dbg:

```
77 79 98 FF
```

Since we verified that the formula works, we can implement it into our own code to **jmp** to our code cave.

3.4.9 Redirection Function

We will handle the redirection in our **DllMain** function, when our DLL is first injected. First, we will need to declare a pointer to our hook location. In addition, the **VirtualProtect** API requires a parameter to hold the previous protection type. We will declare that as well:

```
DWORD old_protect;
unsigned char* hook_location = (unsigned char*)0x00CCAF8A;
```

Next, we will change the protection type for our hook location. The **VirtualProtect** API has similar parameters to the **ReadProcessMemory** and **WriteProcessMemory** API's. Like we did in our previous code cave chapter, we will need to rewrite 6 bytes.

```
if (fdwReason == DLL_PROCESS_ATTACH) {
  VirtualProtect((void*)hook_location, 6, PAGE_EXECUTE_READWRITE,
&old_protect);
  //redirection
}

return true;
```

With the location now writable, we can begin the process of reassigning the bytes to jump to our code cave. First, we will set the first byte to `0xE9`:

```
*hook_location = 0xE9;
```


Playrix Ex. 1024, page 129

We will then write the additional opcodes needed for the **jmp** using the formula we tested above. These opcodes will begin 1 byte after the hooking location:

```
*(hook_location + 1) = &codecave - (hook_location + 5);
```

However, this code will not work as intended. Instead of writing 4 bytes, this will only write 1 byte. This is because **hook_location** is defined as a pointer to an **unsigned char,** which is 1 byte long. To write the 4 bytes we need, we will cast **hook_location** as a pointer to a **DWORD**. We will also cast the other variables to **DWORD**'s:

```
*(DWORD*)(hook_location + 1) = (DWORD)&codecave - ((DWORD)hook_location + 5);
```

Finally, just like we did in the previous chapter, we need to make the sixth byte a **nop**. This can be done in an identical manner to the method we used to set the first byte to a **jmp**. We add 5 (instead of 6) as values are indexed from 0 in C++:

```
*(hook_location + 5) = 0x90;
```

With this done, we can build and inject the DLL identically to how we did it in the previous chapter. When in game, select *Terrain Description* on any tile. Before displaying the description, your gold should be set to 888.

The full code is in Appendix A for comparison.

3.5 Printing Text

3.5.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

3.5.2 Identify

In this chapter, we will print our own text in Wesnoth. To accomplish this task, we will first locate a section of code responsible for printing text. Then, we will use a code cave to modify the game's memory to display our text.

3.5.3 Understand

There are multiple approaches to print our own text inside a game:

1. Use an external overlay.
2. Create a code cave inside the game's main display loop and call the function responsible for displaying text.
3. Create a code cave inside a function responsible for displaying text and modify the text about to be displayed.

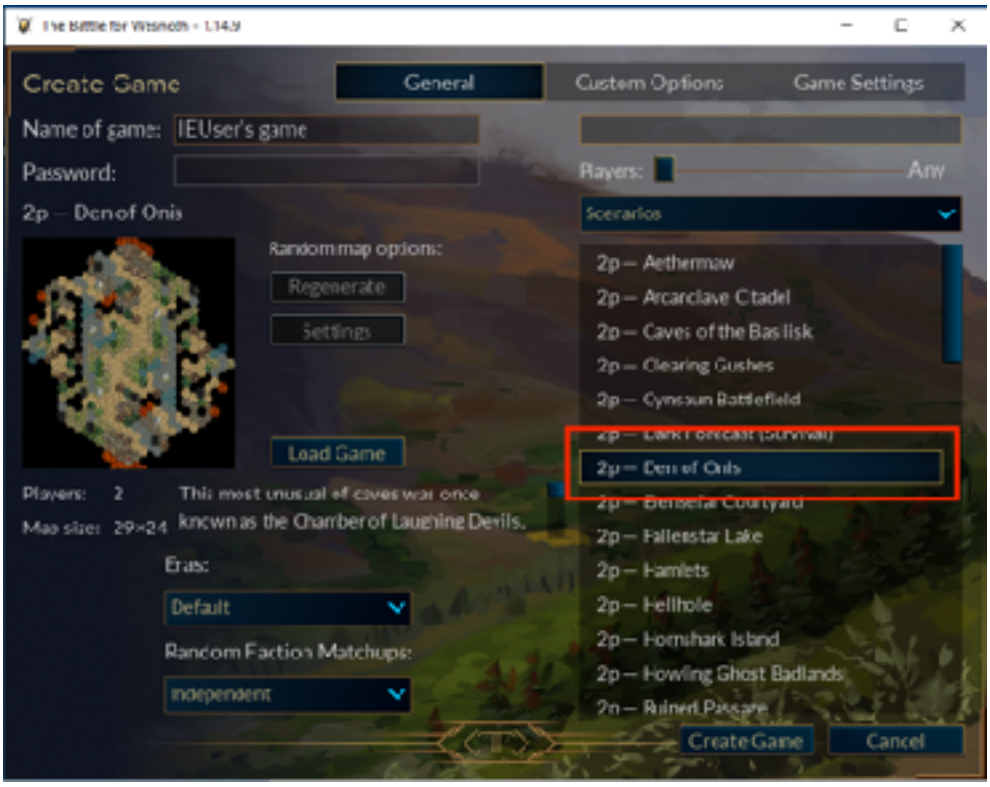
Different games are suited best for different methods. For this chapter, we will use the third approach as it is the easiest to do in Wesnoth. We will examine the other approaches more in-depth in future chapters.

3.5.4 Locating Text

Our first task is to locate the game's code that is responsible for displaying text. To start, we need to find a string of letters that appears in the game. For Wesnoth, we can use the *Terrain Description* text that is displayed when clicking on a tile. Any description will work, but for this chapter, we will use the description for the *Ford* tile. For other games, chat messages are a good starting location.

First, select a map that has *Ford* tiles on it. *Den of Onis* is one example:

Playrix Ex. 1024, page 131



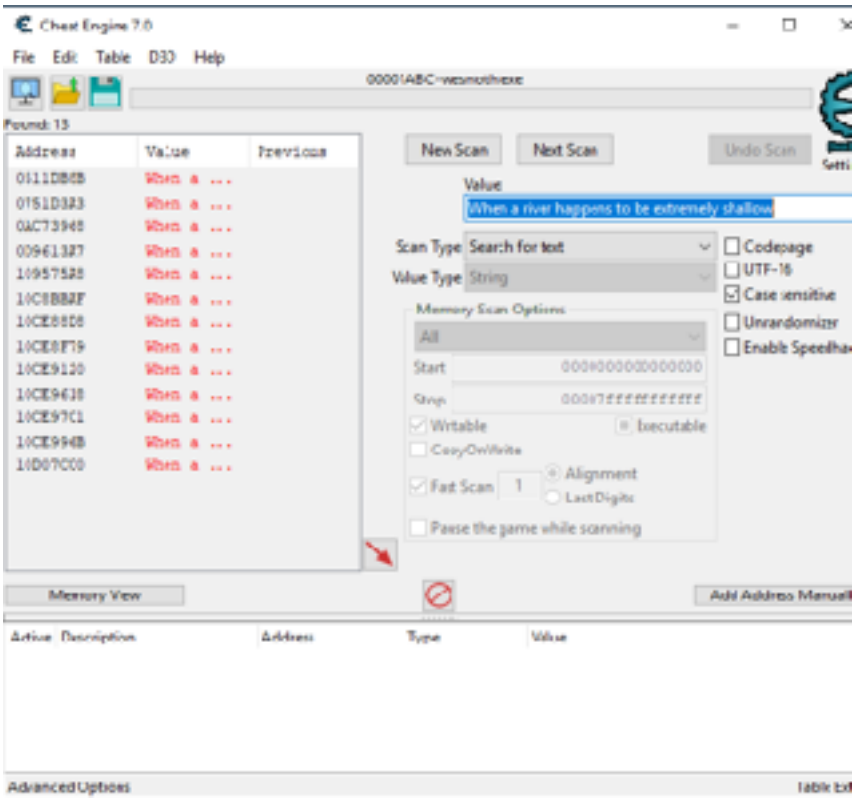
When the map loads, select a *Ford* tile and select the *Terrain Description* entry on the context menu:



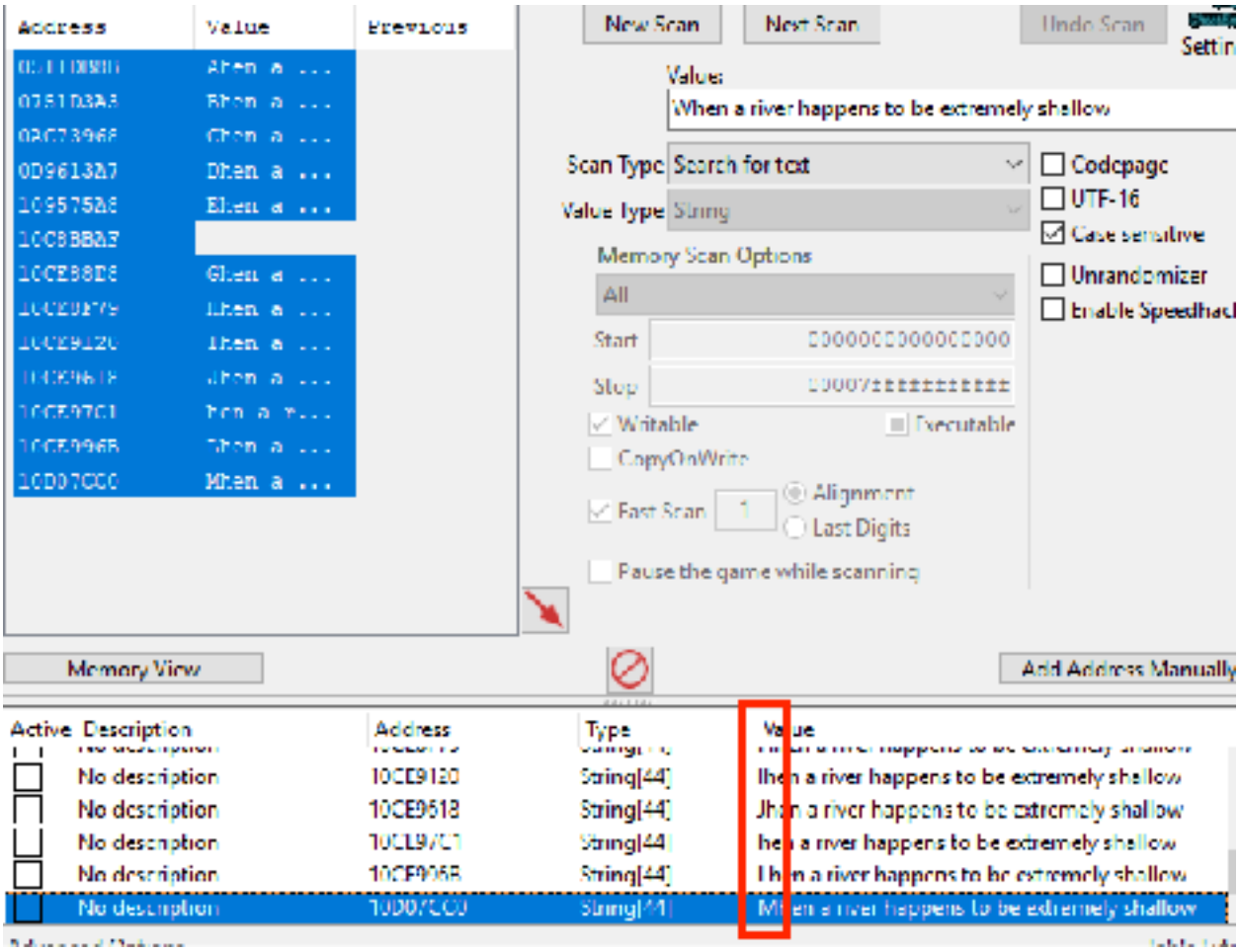
This will bring up the description for the tile, which contains a long string of text:



We can use Cheat Engine to search for where this text is stored within the game. Make sure to close down the terrain description box before searching to reduce the amount of results. Due to the unique nature of the text, we only need to search for a couple words:



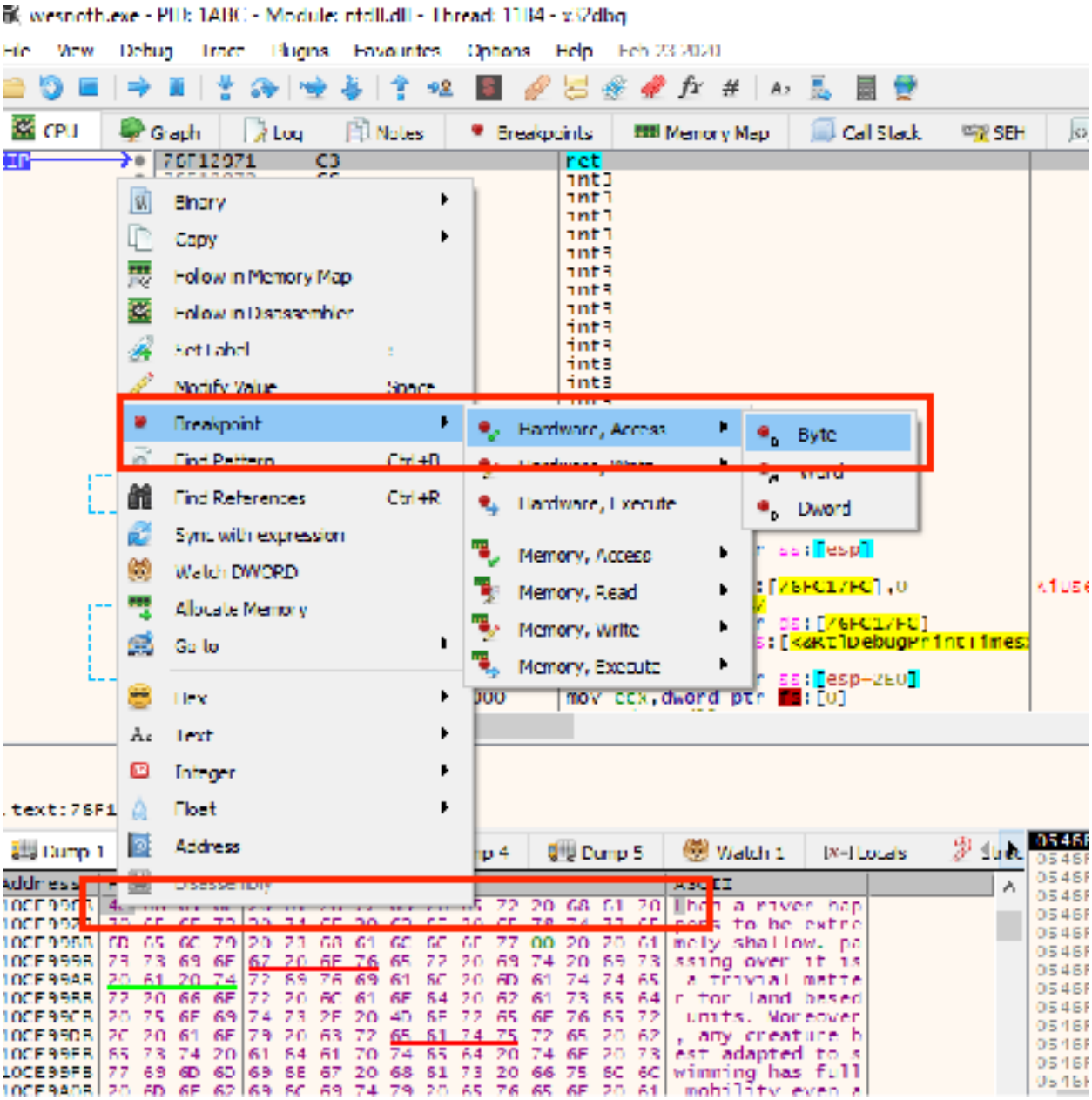
To narrow down which address represents the string we are interested in, change the first letter of every string. After changing these values, go back into Wesnoth and examine the terrain description again. The version of the string that is displayed in game will match up with the correct address. In this case, the string starting with *Lhen* was displayed, making our address `0x10CE996B`.



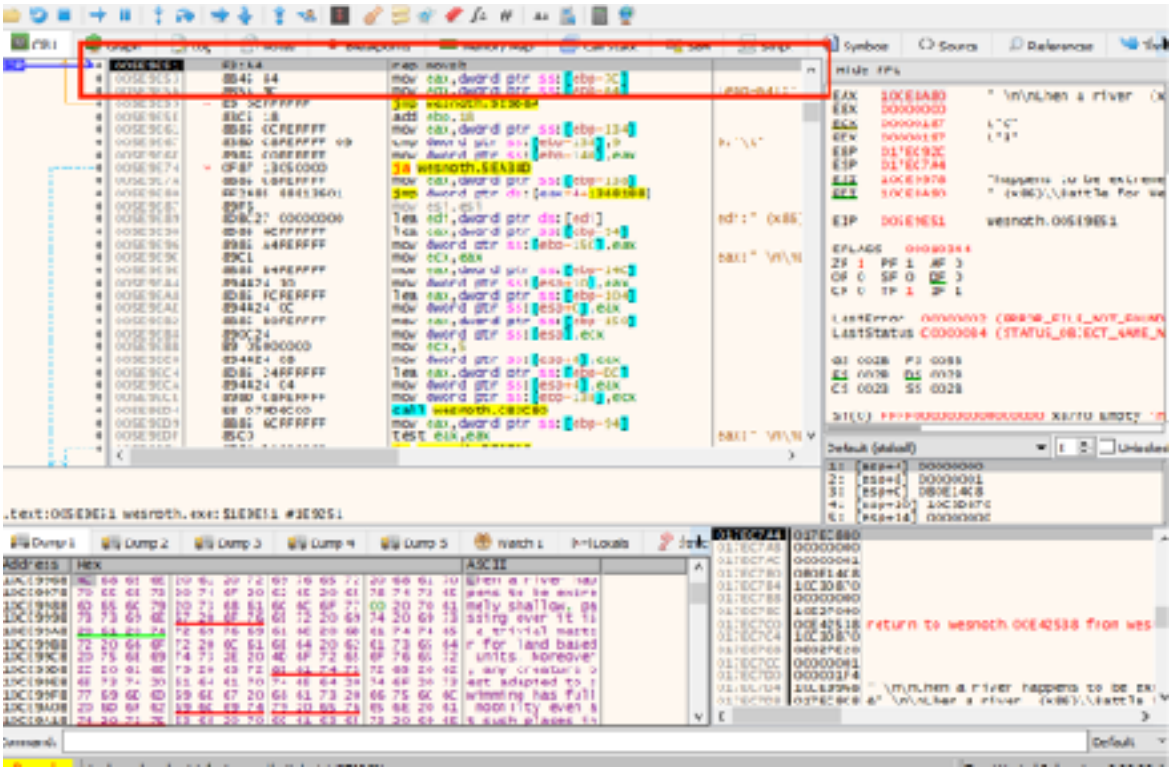
3.5.5 Locating PrintText

We can now use the address we found to locate the function responsible for printing text. We know that the print text function must access this text in some way to print it. To determine where this function is, we can set a breakpoint on a byte of the text.

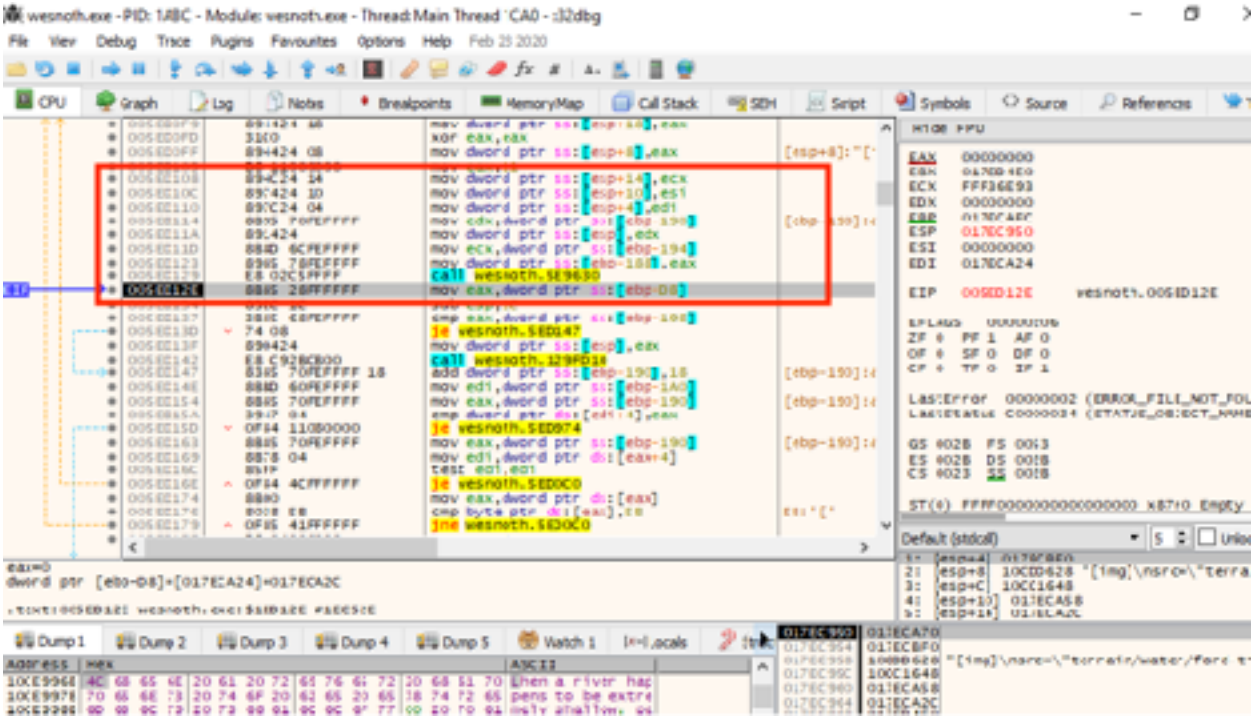
Playrix Ex. 1024, page 134



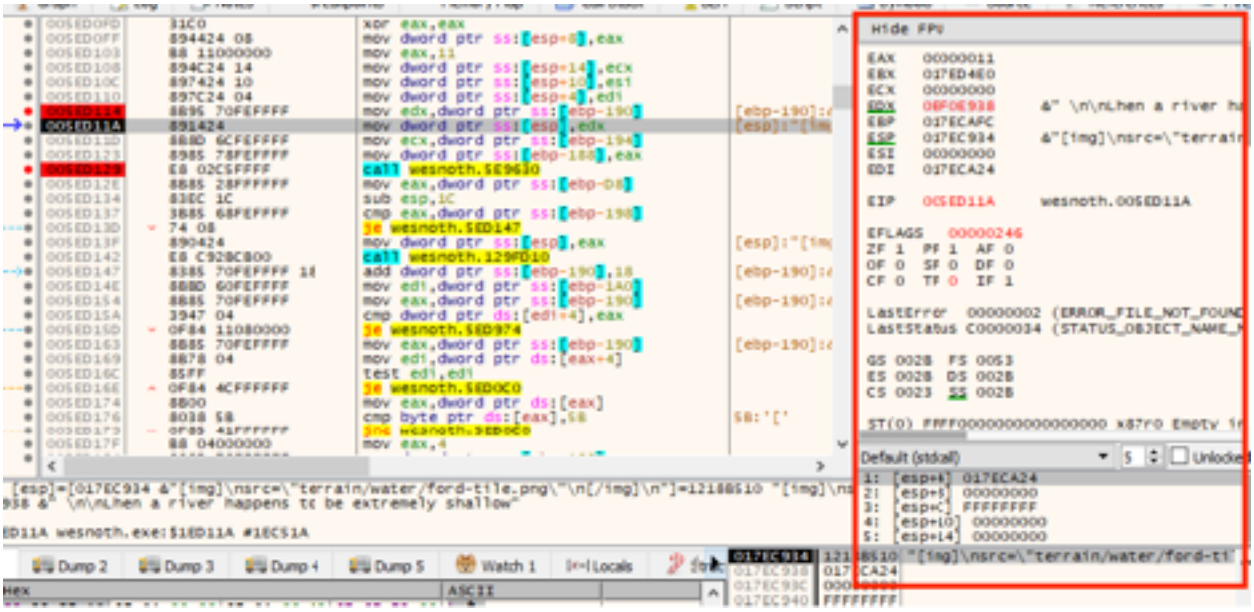
With the breakpoint set, go back into Wesnoth and invoke the *Terrain Description* action again. Your breakpoint will pop immediately:



Examining this code, we appear to be in a loop responsible for moving each byte of the text into a buffer. Like we did in previous chapters, we want to navigate to the code that called this lower-level code by using execute until return and stepping out.



This call looks like it could be responsible for populating the terrain description box with text. If we continue execution, we notice that this code is called multiple times for each section of the terrain description box. To determine the parameters passed to this call, we can set a breakpoint on `0x005ED114` and invoke the *Terrain Description* action again:

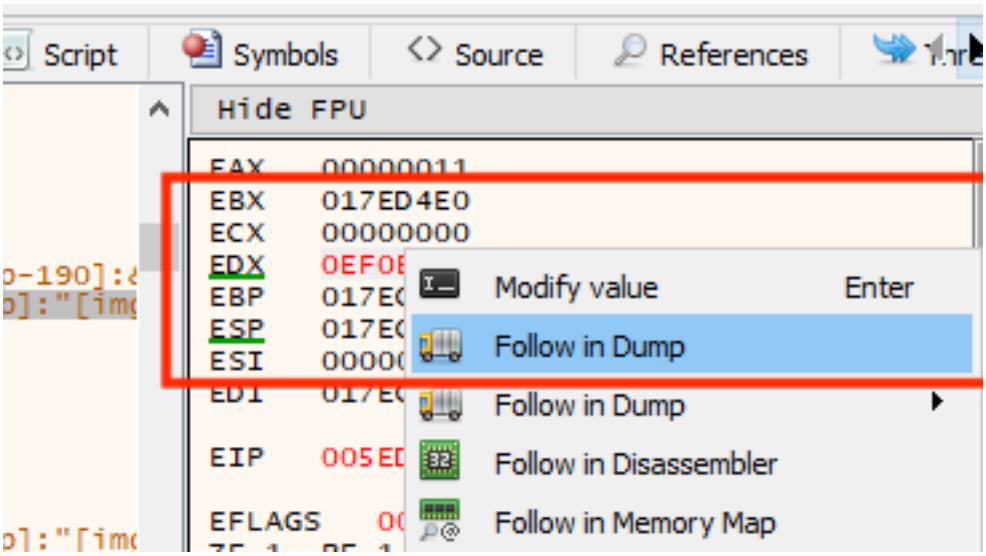


The highlighted section shows that the text is loaded into the register **edx**. The code at `0x005ED11A` then moves the value in **edx** into the location pointed at by **esp**. This is identical to pushing the value of **edx** on the top of the stack. While we have not discussed the stack yet, for the purpose of this chapter, we need to know that functions will often retrieve values off of the stack for use in execution.

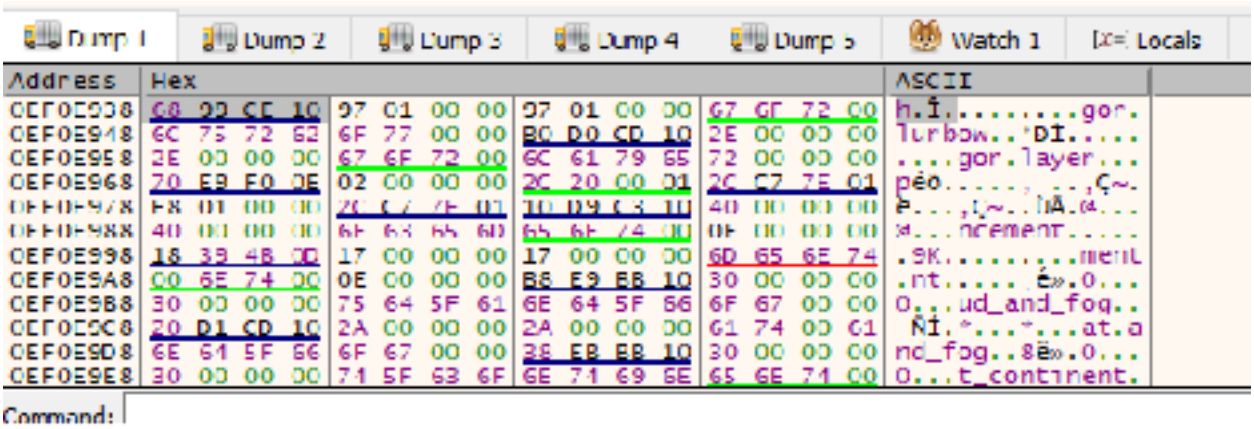
3.5.6 Memory and Endianness

You may have noticed that the address in **edx** does not match the address we found in Cheat Engine. Since the text space is dynamically allocated, we will need to understand how to retrieve the value of the text from **edx** to create our code cave.

Invoke the *Terrain Description* action again to force our breakpoint to pop. Once it does, right-click on the value of **edx** and choose *Follow in Dump*:



This will change the current address displaying in the dump to the value of **edx**. As we discussed in previous chapters, the dump section displays the current running memory of a process. It's important to remember that both the dump section and Cheat Engine are displaying and searching the same data.



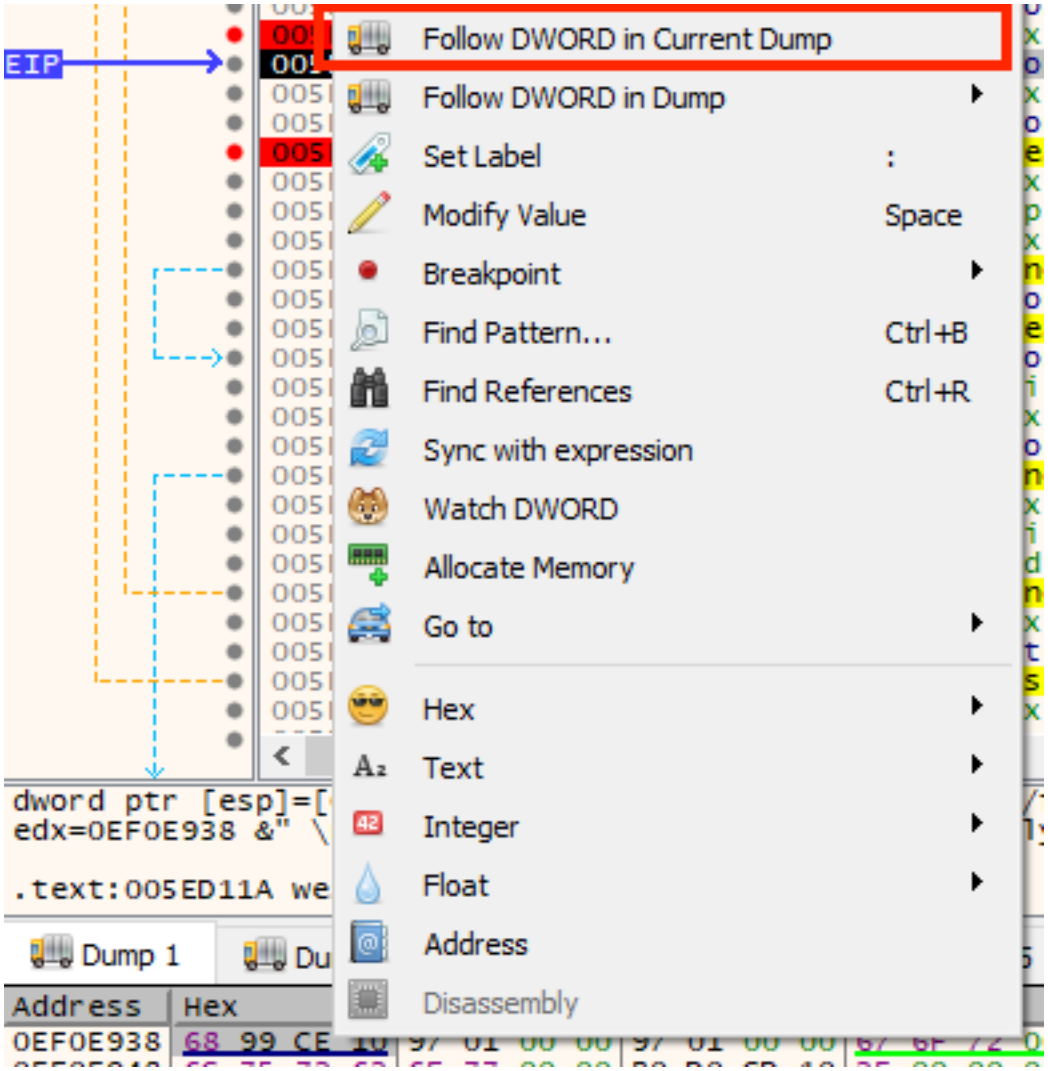
This value stored in **edx** is obviously not a text string. However, if we examine the value of the bytes, we see that they share many similarities to the address we found in Cheat Engine. In the previous chapter, we briefly discussed a concept known as endianness. Most Windows-based CPU's are little endian. By definition, this means that the least-significant byte is stored in the smallest address.

In practice, this means that when the address `0x12345678` is stored in memory, it will be stored as `0x78 56 34 12`. In this case, `0x78` represents the least-significant byte, or the smallest value. A good comparison is to imagine the number 123. Expressed in a

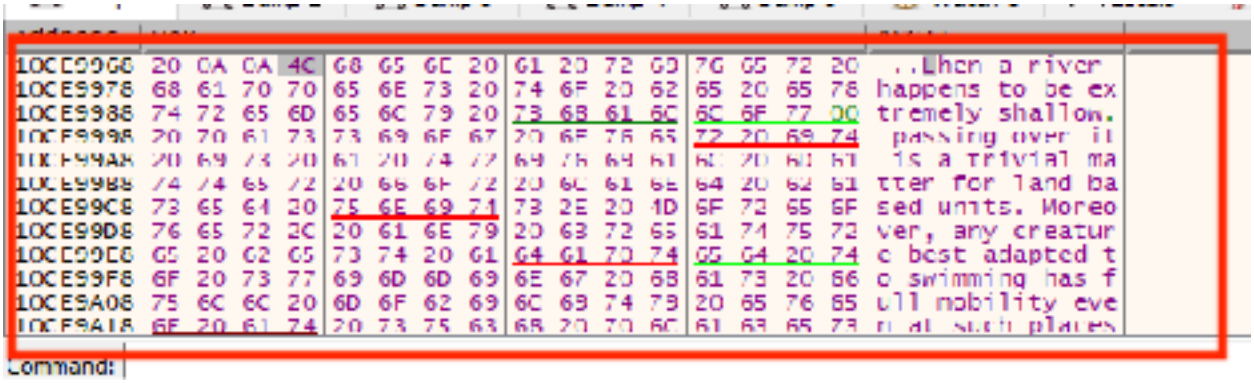
longer form, this value can be understood as 1*100 + 2*10 + 3*1. The smallest value in this form is the number 3.

The second part of this definition can be understood by examining the dump. In the dump, memory addresses grow from a lower value to a higher value. Because of this, the least-significant byte will be stored "first" in memory. The combination of these factors make addresses stored in memory appear to be "reversed".

Now that we understand endianness, we can conclude that the value stored at **edx** is an address. We can quickly navigate to this address in the dump by selecting all the bytes and selecting *Follow in Dump* again:



After selecting this, we arrive at our string's location in memory:



To reference this value in assembly, we can make use of the **ptr ds** keyword:

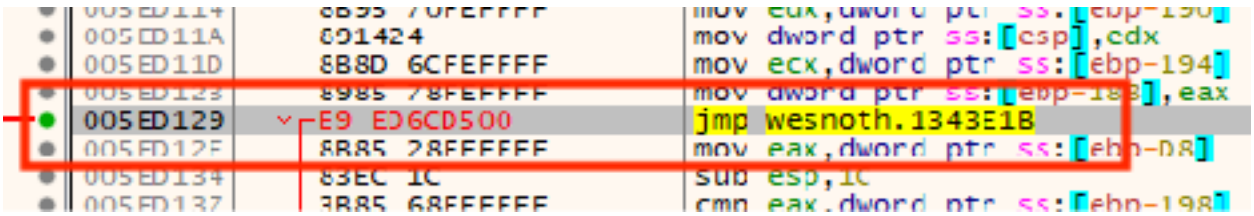
```
mov eax, dword ptr ds:[edx]
```

This will load the value of the address stored in **edx** into **eax**. In this case, it would load the value `0x10CE9968` into **eax**. We could then use the **ptr ds** keyword again to access the individual bytes of the text.

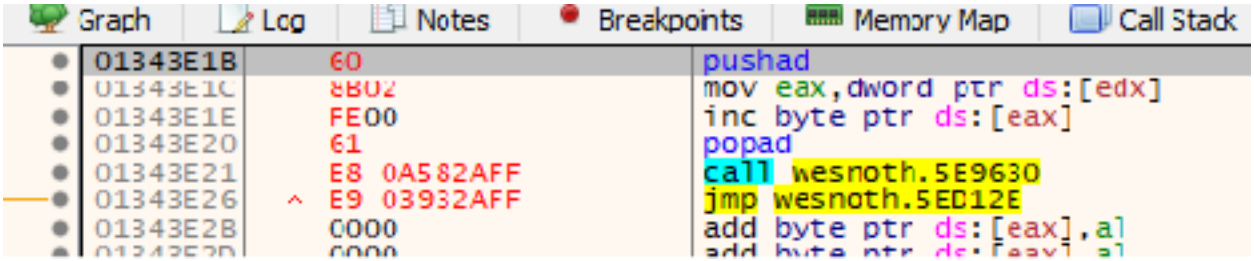
3.5.7 Changing Text

With this reversing done, we can start creating our hack. To verify that we have the correct method, we will create a code cave that will change the text displayed each time the *Terrain Description* action is invoked. To do this simply, we will increase the value of the first byte each time our code cave is executed. This will change the value of the character and allow us to confirm that our hack is working.

Since we know the **call** at `0x005ED129` is responsible for printing the text and is also 5 bytes long, we will use it as our redirection point. As discussed in previous code cave chapters, any location near the end of program's memory will work for our cave location. In this case, we will create it at `0x01343E1B`. As usual, we will replace the hooking location with a jump to our code cave:



We will first save the registers in our code cave. Then we will use the **ptr ds** keyword to load the value of the text from **edx** into **eax**. After that, we will use the **inc** operator to increase the value of the first byte of the string. For example, if the first byte is currently A (ASCII value 65), it will be increased to B (ASCII value 66). Finally, we will restore the registers, recreate the **call**, and then jump back to the original code.



With this completed, go back into Wesnoth and invoke the *Terrain Description* action multiple times. You will notice that a character after the image changes each time, demonstrating that we have successfully modified the text displayed.



Part 4
RTS Hacks

Playrix Ex. 1024, page 142

4.1 Stathack

4.1.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

4.1.2 Identify

In this chapter, we will create a statistic hack, more commonly known as a stathack. This type of hack displays information to us about other players, such as their gold or number of units.

In this chapter, our stathack will display the gold of the second player.

4.1.3 Understand

To create our stathack, we need to accomplish two steps:

1. Find the second player's gold.
2. Print this value to the screen.

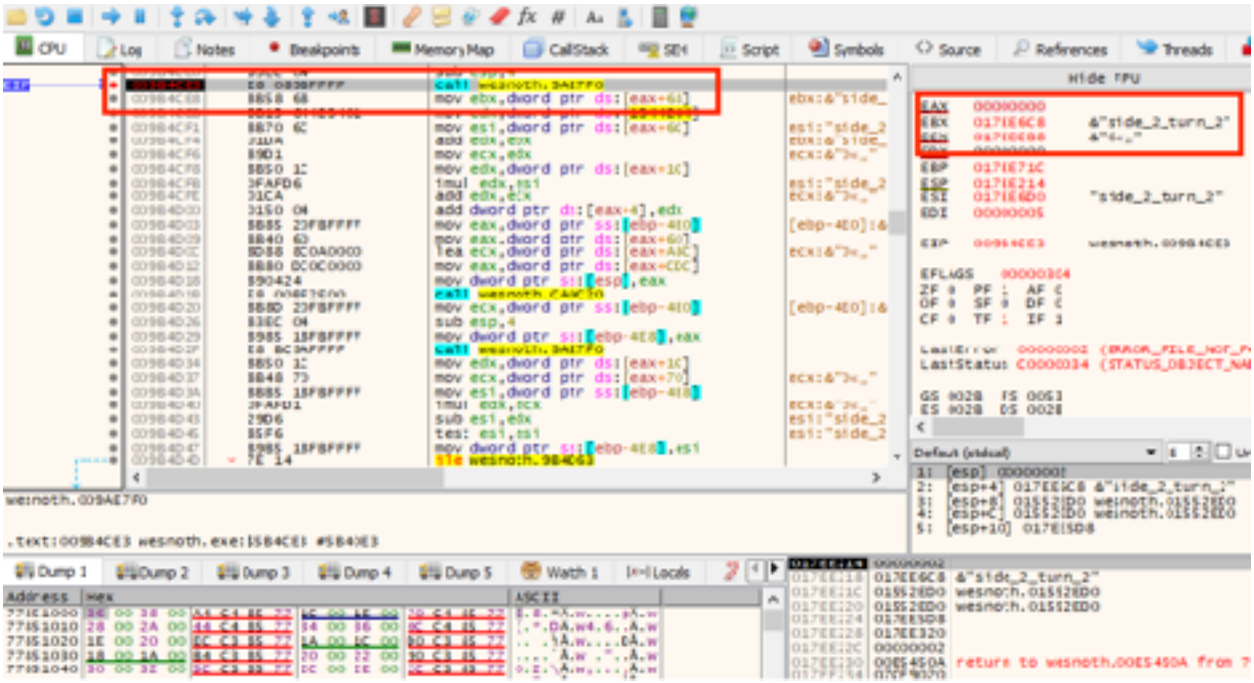
In previous chapters, we covered the techniques to do both of these steps.

4.1.4 Second Player's Gold

Back in Chapter 1.2, we explored how games will often allocate similar data and classes in arrays. These arrays can then be iterated over by the game to locate and update data. While we do not know if this is how Wesnoth works, we can use it as a model to try to locate the second player's gold value.

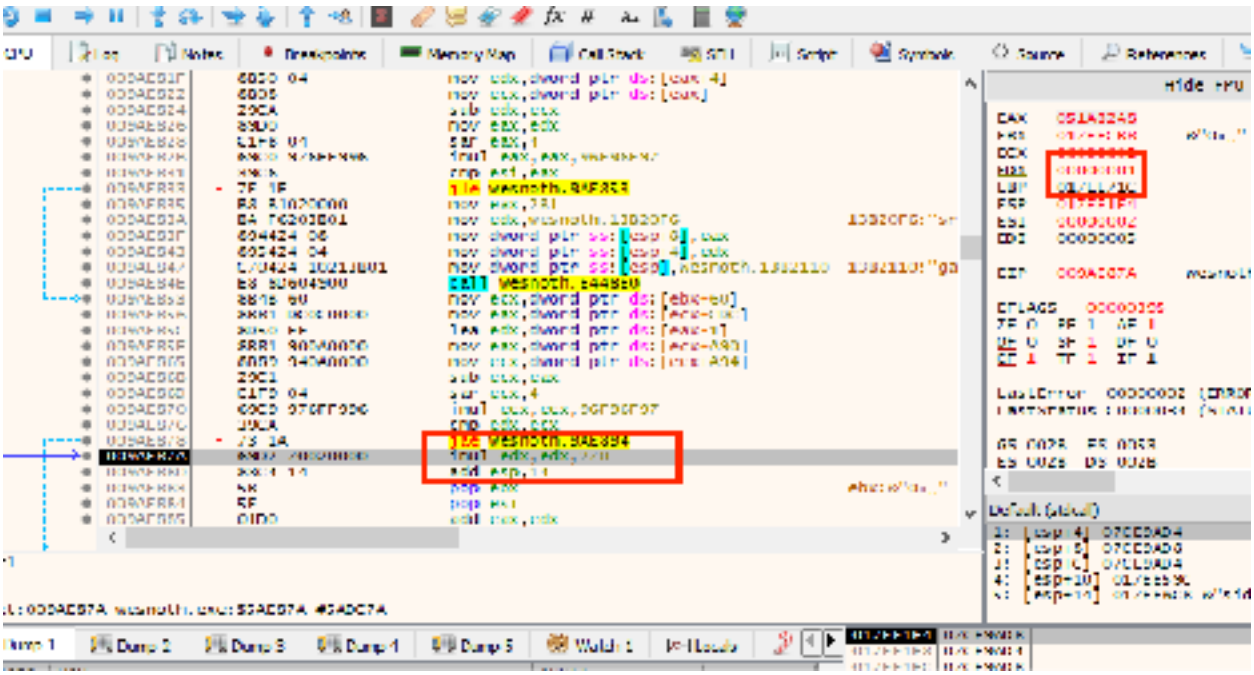
We know from Chapter 2.8 that the game dynamically allocates player classes based on a base pointer. We also identified the game's and first player's base pointers. By closely examining the code we located in that chapter, we can determine if the game uses an array for its player classes and, if so, locate the second player's base pointer.

First, create a local game with two local players. Make sure both players receive income each turn. Start the game and attach x64dbg. Play one turn for each player to make sure any first-turn initialization code has executed. Next, set a breakpoint in x64dbg on `0x9B4CE3`, the same **call** we identified before. In Wesnoth, end the first player's second turn and the breakpoint should pop:

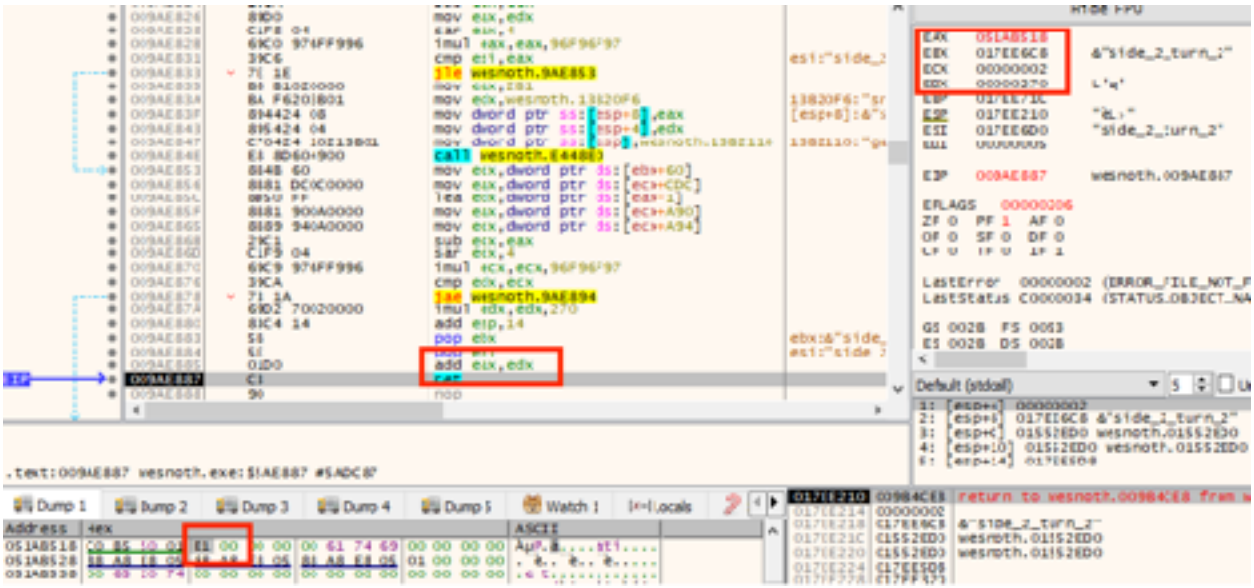


The value in **ebx** indicates that this function is invoked for every player each turn. Furthermore, we know that the value in **ecx** is the game's base pointer. From these two facts, we can assume that the game has an array of player classes.

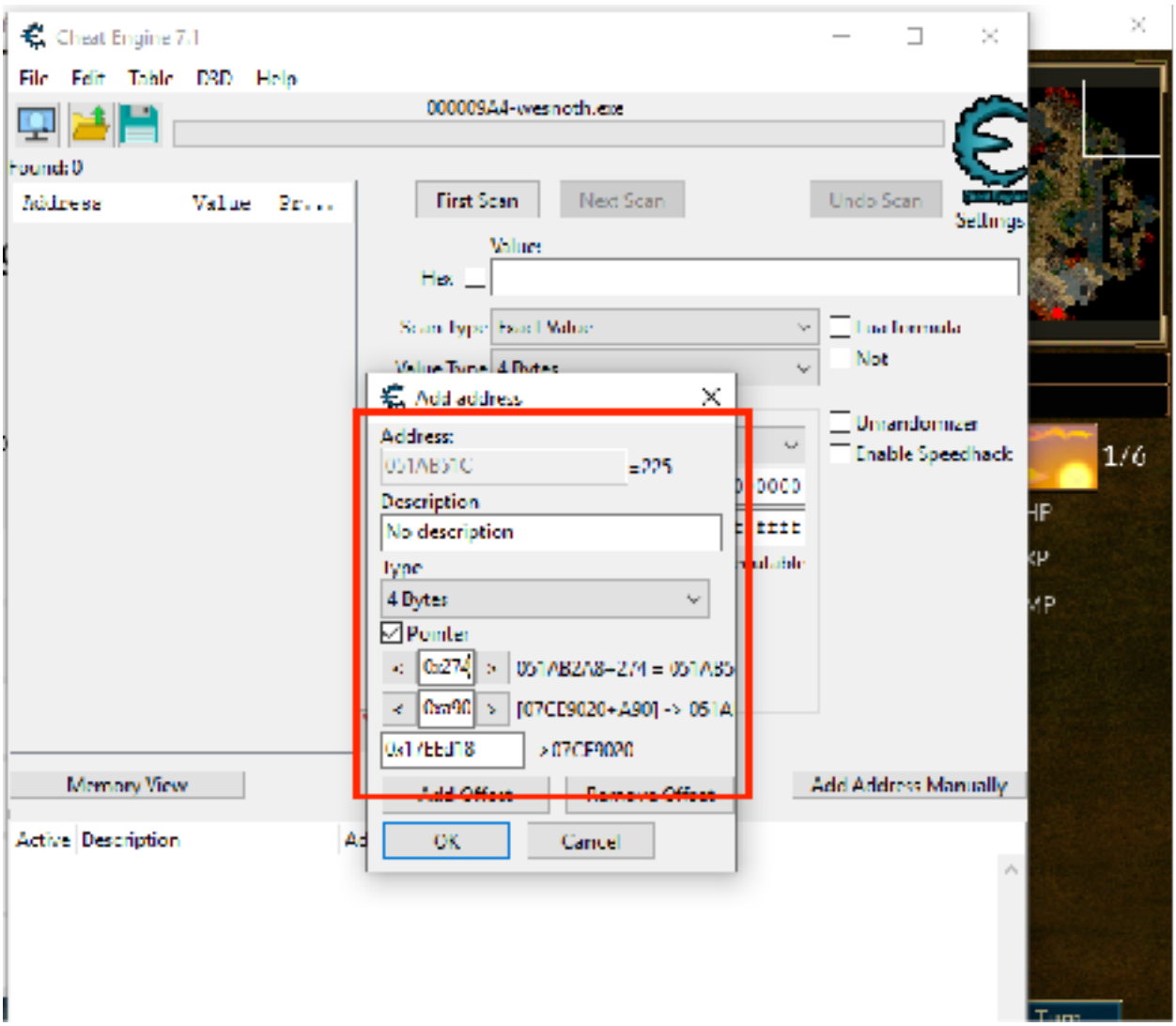
Our next step is to determine the size of each player in the array. The game must know this size to advance to the next player in the array. Step into the **call** at `0x9B4CE3` and step through each line of code. From the previous chapter, we know that this code will return the player's gold address in **eax**. For the majority of this function, the addresses and values used are identical to the values we observed when looking for the first player's gold address. However, near the bottom of the function is an **imul** (signed multiply) instruction:



When we step through this function as the first player, **edx** is set to the value of 0. However, when we step through with the second player, **edx** is set to the value of 1. From this code, it appears that the game uses **edx** to offset the current player and then multiplies **edx** by the value of `0x270` to identify the current player. If we step down a few more lines of code, we see that this value is then added to the value of **eax** to get our current player's gold address:



From this code, we can identify how to offset our second player's gold value. Like we have found previously, we will use `[[0x017EED18] + 0xA90]` to offset the game's base pointer. If we add 4, we will get the first player's gold address. To get the second player's gold address, we can instead add `0x270 + 4`, or `0x274`. We can verify this calculation using Cheat Engine:



In previous chapters, we have already written code to offset the first player's gold address. We can modify this code with the new value of `0x274` to retrieve the second player's gold address like so:

```
player_base = (DWORD*)0x017EED18;
game_base = (DWORD*)(*player_base + 0xA90);
gold = (DWORD*)(*game_base + 0x274);
```

4.1.5 Printing Value

In the previous chapter, we covered a method to print text. We determined that by creating a code cave, we could access the text for the *Terrain Description* method by referencing the value pointed at by **edx**. Once we accessed it, we could store bytes in this location to be displayed by the game.

Using the method from Chapter 3.4, we can implement this functionality in a DLL. Since we already discussed the method to redirect code, we will examine only the code cave function now. We will start with the skeleton:

```
DWORD ori_call_address = 0x5E9630;
DWORD ret_address = 0x5ED12E;

__declspec(naked) void codecave() {
  __asm {
    pushad
  }

  // new code

  _asm {
    popad
    call ori_call_address
    jmp ret_address
  }
}
```

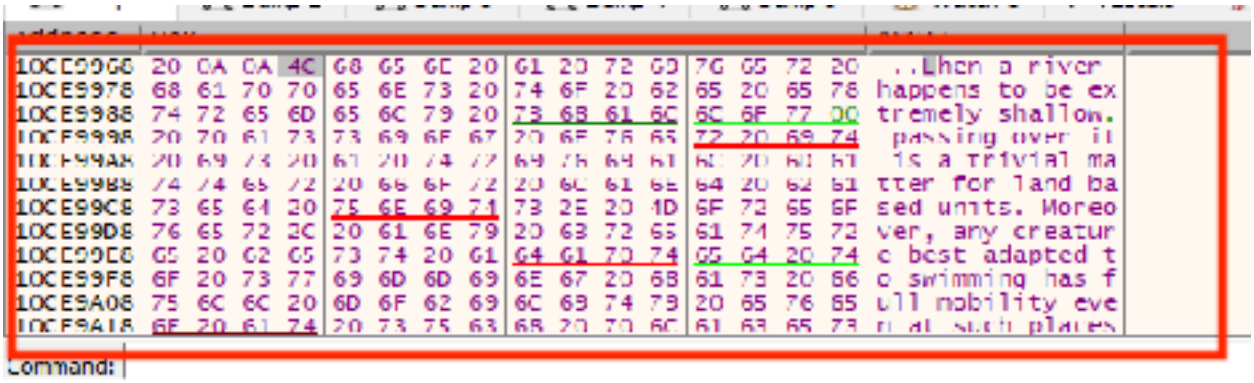
We have seen this code before. The major difference is that the instruction we are replacing for the text printing is a **call**.

This code cave will be called each time the *Terrain Description* method is invoked. In it, we want to retrieve the second player's gold value. We can do this using the code we discussed in the previous section:

```
__asm {
  pushad
}

player_base = (DWORD*)0x017EED18;
game_base = (DWORD*)(*player_base + 0xA90);
gold = (DWORD*)(*game_base + 0x274);
```

We now have the second player's gold value stored in the **gold** variable. To display this value in the game, we need to convert it to a string of characters. To understand what we are trying to accomplish, here is the memory dump containing the text string we found in the previous chapter:



Looking at this, we see that even though the game displays *Lhen*, the values stored in memory are `0x4C 68 65 6E`. This is due to the game using ASCII encoding to encode character values as certain numbers. The game then knows to decode these values and display the corresponding character in game.

Therefore, if our gold value to display is 225, we cannot simply write 225 into the game and expect the game to display it successfully. Instead, we need to convert it to `2 2 5`, or `0x32 32 35`. There are several ways to do this, the easiest being through the use of the sprintf_s API:

```
#include <stdio.h>

char gold_byte_array[4] = { 0 };

...
gold = (DWORD*)(*game_base + 0x274);

sprintf_s(gold_byte_array, 4, "%d", *gold);
```

This will convert the value pointed at by the **gold** variable into its string representation and store that value in **gold_byte_array**.

Finally, we will move this converted value into the memory that will be displayed. Since **sprintf_s** alters several registers, we will first restore them and save them again to ensure that the game does not crash:

```
__asm {
  popad
  pushad
  mov eax, dword ptr ds:[edx]
```

In the previous chapter, we simply incremented the first character pointed to by **eax**. To display our gold value, we will move the values stored in the **gold_byte_array**:

```
mov bl, gold_byte_array[0]
mov byte ptr ds:[eax], bl
```

First, we move the first byte of the **gold_byte_array** into **bl**. To understand why we are using **bl**, we need to understand the different sizes of data used by the CPU. A bit is the smallest unit of data, representing either 0 or 1. A byte is 8 bits. A word is 16 bits. A double word (or **DWORD**) is 32 bits. When looking at the memory dump in x64dbg, we are looking at byte values. 4 of these byte values combined together form a **DWORD**.

Currently, all the registers we have seen, like **ebx**, are **DWORD'**s. Early Intel CPU's, like the 8088, used 16-bit or **WORD** registers, like **bx**. To access each byte in the **bx** register, you would use **h** and **l**, such as **bh** and **bl**. On modern CPU's, even though we are using extended (or **DWORD**) forms of these registers, these same rules apply. Since we are moving individual bytes, we need to move them into a value that can hold a byte. We then move this value into the location pointed at by **eax**, which is also a byte long.

This code will move our first character into the text. We can repeat it several times to move additional characters. For this chapter, we will only display 3 characters' worth of data:

```
mov bl, gold_byte_array[1]
mov byte ptr ds:[eax + 1], bl
mov bl, gold_byte_array[2]
mov byte ptr ds:[eax + 2], bl
```

Finally, identically to Chapter 3.4, we will redirect the game's print text function to our code cave in **DllMain**:

```
DWORD old_protect;
unsigned char* hook_location = (unsigned char*)0x5ED129;
```

```
if (fdwReason == DLL_PROCESS_ATTACH) {
  VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
  *hook_location = 0xE9;
  *(DWORD*)(hook_location + 1) = (DWORD)&codecave - ((DWORD)hook_location +
5);
}
```

Build and inject this the same exact way we did it in previous chapters. Finally, go inside a game and open up the *Terrain Description* on a tile. We should see the second player's gold value printed several times due to how we hooked the function:



The full code for this chapter is available in Appendix A.

Playrix Ex. 1024, page 150

4.2 Map Hack

4.2.1 Target

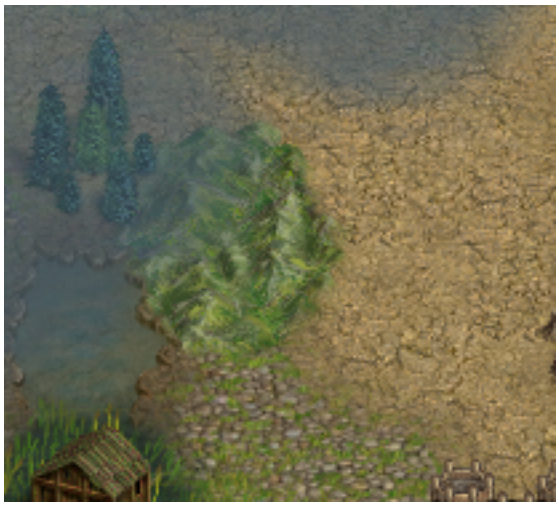
Our target in this chapter will be Wesnoth 1.14.9.

4.2.2 Identify

Our goal in this chapter is to create a map hack, a type of hack that displays the entire map to the player and removes elements like fog-of-war.

4.2.3 Understand

In strategy games like Wesnoth, tiles on the map can either be visible or hidden by fog-of-war:



We know that the game must store whether the tiles are visible or not somewhere in memory. These locations are most likely in one large block of memory. One way a game might choose to represent the map is through the use of an array. In this array, each element would represent one map tile's visibility status:

```
int map[map_size] = {0, 0, 1, 0, 0, 1, 1, 1, 0, …}
```

The game could then iterate over each tile in the array to determine whether fog should be drawn over the tile.

We also know that the game must calculate the values for each tile every time the player moves a unit. If the player moves a unit in range of a tile, the game needs to set the tile's visibility to true. If the player moves a unit out of range of a tile, the game needs to set the tile's visibility to false. To make this calculation easier, games will often first set all tiles to an invisible state:

```
for(tile in map) {
  map[tile] = 0
}
```

Then the game can go through each unit that the player controls and set all the surrounding tiles to visible.

While every game will have its own way of handling map data, they all must follow a similar set of steps to calculate visible tiles. We can use the following approach to create a map hack for any strategy game:

1. Search for an unknown value.
2. Move a unit to reveal part of the map.
3. Filter for changed values.
4. Move a unit to hide the revealed part.
5. Filter for changed values.
6. Repeat this process until you have a reasonable amount of results (~50).
7. Look for patterns in the results and edit each one until you figure out which ones represent tile data.

Once you have found the tile data, a breakpoint can be set on one of the tiles. Then, a unit can be moved and the breakpoint will pop in the function responsible for writing values to the map data.

4.2.4 Locating Map Data

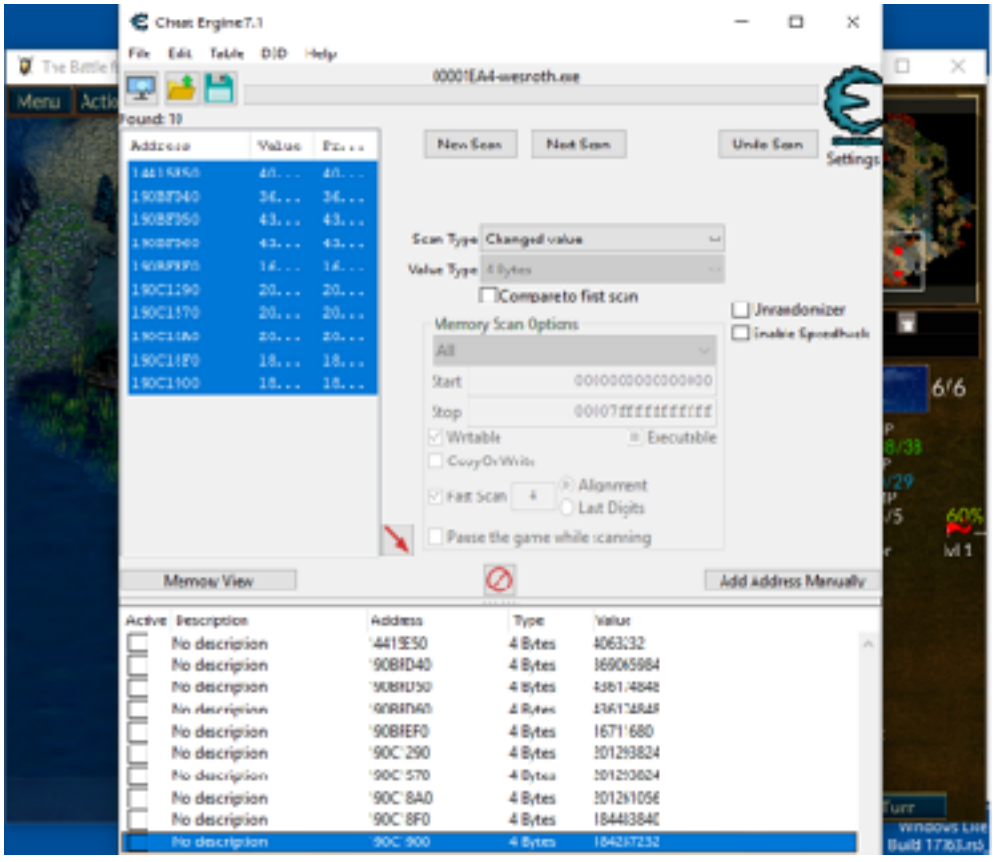
Locating the map data is the most time-consuming part of creating a map hack. First, create a local game in Wesnoth with a single player-controlled opponent. Since we do not know what values we are searching for, start a new scan for an *Unknown value* type.

After the scan completes, select a unit and move it to a new location to reveal additional tiles. Make sure you remember this location to use on all future requests.

After moving the unit, change the scan type to *Changed value* and filter the results. When the filtering has completed, move the unit back to the start location and end your turn to hide the terrain again. Quickly end the next player's turn and then scan again for *Changed value* when you regain control of the first player. Continue this process until you filter the results down to a manageable amount.

To quickly reduce the amount of results, you can also change the scan types to eliminate values that may change constantly but not in a manner related to the map tiles. For example, if you recruit a unit that does not reveal additional squares and then search for *Unchanged value*, many results will be filtered out. This approach can be used with different conditions (for example, pausing and resuming the game and searching for unchanged values) to quickly reduce the search size.

Eventually, you will get your set of results down to a reasonable level that will allow you to observe game behavior manually. In this chapter, we have managed to narrow down the result set to 10 possible addresses. Due to DMA, these addresses will be different each time we start a new game:



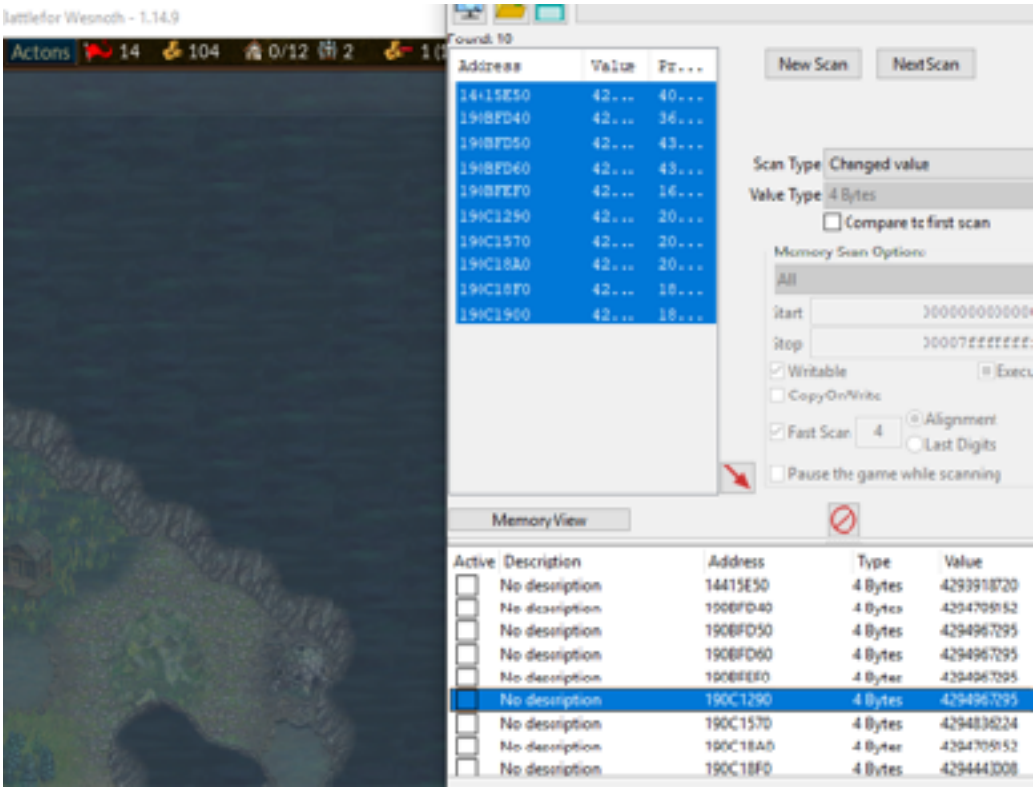
Playrix Ex. 1024, page 153

Initially, these values do not appear correct, as the values seem almost random. To determine if we have the correct addresses, check the box next to each address in the *Active* column. Checking this box will disable modifications to the addresses' values. With the addresses inactivated, move your unit away from the tiles you have been testing on. You should notice that the tiles no longer display fog-of-war when moving away. This test confirms that we have found the correct addresses.

4.2.5 Locating Map Code

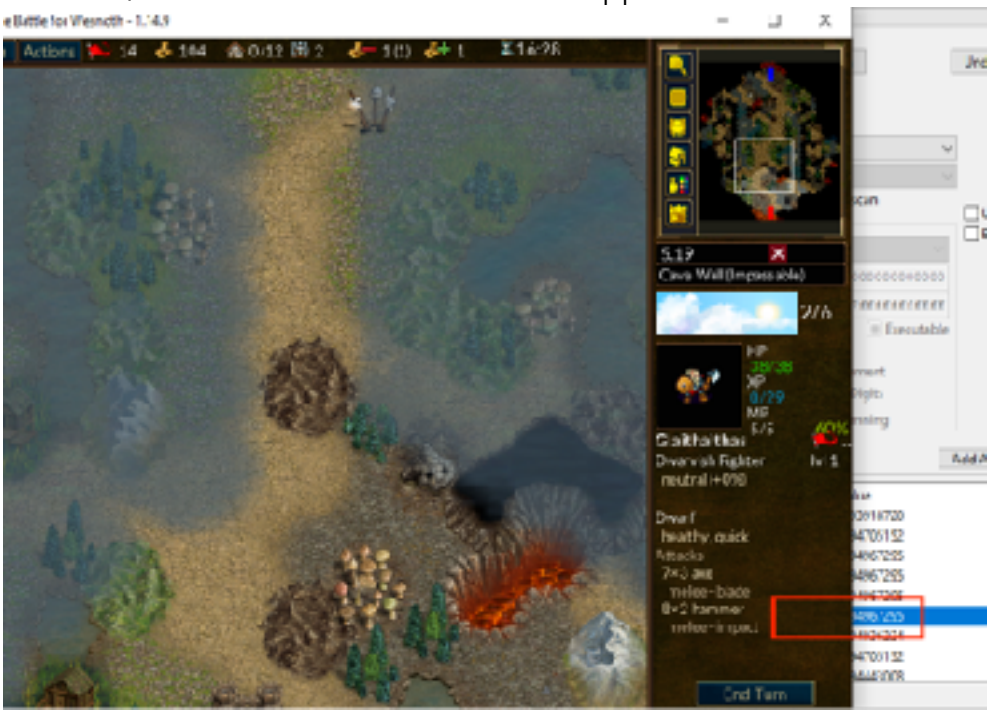
Now we need to determine how the game handles map tile data. When reversing an unknown game, we start by making educated guesses. In this case, we guessed that the game stored individual tiles with a simple visible/invisible scheme to determine visibility. We used this model to help track down the data we are interested in, but we now realize that this model is incomplete. Before we can continue, we have to update our model to reflect our findings.

Observing Cheat Engine, move a unit to reveal and hide tiles. You should notice that the values we found change consistently when doing this. When a tile is hidden, it appears to be several large values. However, when a tile is visible, it appears to always be set to *4294967295*:



Playrix Ex. 1024, page 154

This is a distinct value, so let's try setting another address we identified to this value. When this is done, a whole column of tiles should appear visible:



By scrolling up the map, you can see that the whole column from the top of the map down is controlled by this one value:

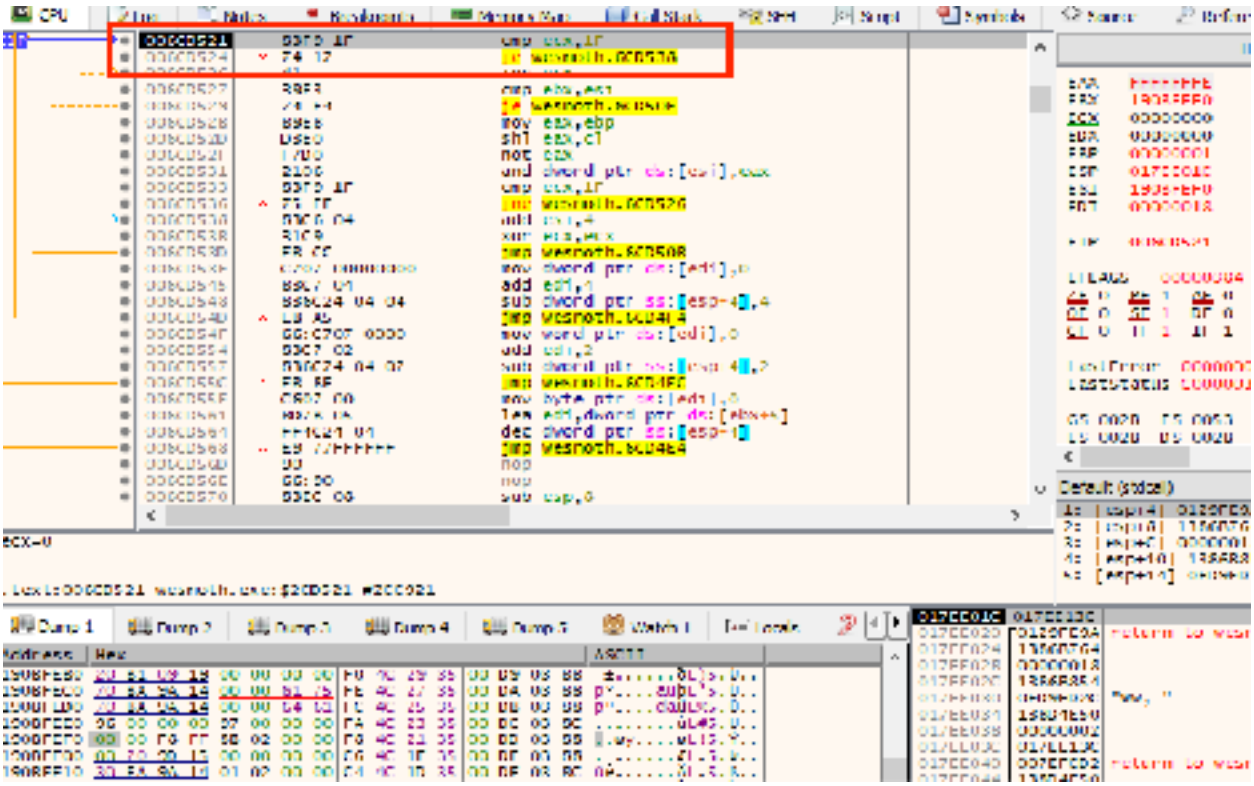


With this information, we can now alter our model. Instead of an array of tiles, Wesnoth appears to use an array of tile columns. These columns are then set to a value between 0 and 4294967295, depending on the number of tiles in the column that are visible:

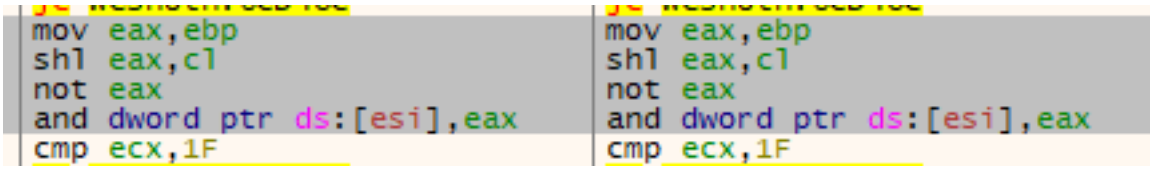
```
int map[column_size] = {0, 0, 4294967295, …}
```

The value of 4294967295 converted into hexadecimal is `0xFFFFFFFF`.

Now we can begin tracking down the code responsible for setting these values. Attach x64dbg and set a breakpoint on write on one of the map column addresses. Move your unit to reveal that tile, and the breakpoint should pop:



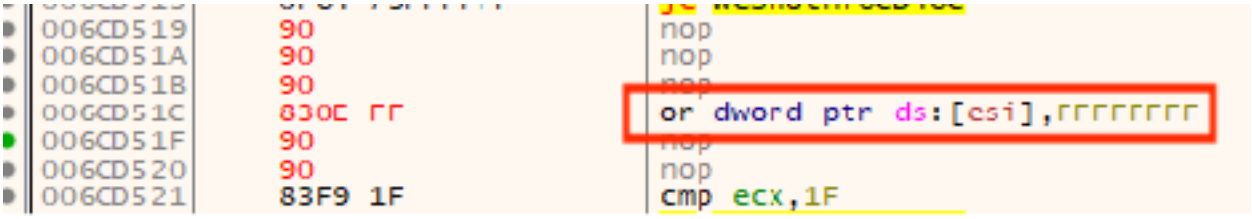
Looking at the registers, we can identify that **esi** holds our map column data. Scrolling up, we immediately see the code responsible for setting this column's value:



If we examine this code, we can see that a value is loaded into the register **eax**, modified, and then used to set our column's value.

4.2.6 Changing Map Code

Since we always want the column to appear visible, we can modify this code to set the column's value to `0xFFFFFFFF`. We will do this through the use of an **or** operation. An **or** operation takes two sets of bits and creates a new set in which the value of each bit is 0 if both source sets are 0, and 1 if either source set is 1. Since `0xFFFFFFFF` translates to all 1's, **or**'ing a value with this value will always produce `0xFFFFFFFF`. We will conduct this operation on our tile column and **nop** out the other instructions:



We could also use a **mov** instruction here to accomplish the same goal. However, one drawback to the **mov** operation is its size, or the amount of opcodes we would require. We simply do not have enough room and would require a code cave. To avoid this extra complexity, we use the **or** instruction instead since it is shorter.

With this modification made, go back into Wesnoth and observe that the entire map is now visible:



We can use a similar approach covered in Chapter 3.4 to create a DLL to accomplish this behavior. First, note down and copy the opcodes generated by x64dbg when making our alteration. We will place these values into an array so we can iterate over them:

```
unsigned char new_bytes[8] = { 0x90, 0x90, 0x90, 0x83, 0x0E, 0xFF, 0x90, 0x90
};
```

Next, just like we did in other chapters, we will unprotect the memory at the hooking location. Then, we will iterate through each opcode in our **new_bytes** variable and write it into the game's memory:

```
unsigned char* hook_location = (unsigned char*)0x6CD519;

if (fdwReason == DLL_PROCESS_ATTACH) {
```

```
  VirtualProtect((void*)hook_location, 8, PAGE_EXECUTE_READWRITE,
&old_protect);
  for (int i = 0; i < sizeof(new_bytes); i++) {
    *(hook_location + i) = new_bytes[i];
  }
}
```

This DLL can then be injected like we did in all the previous chapters. When injected, our map hack will reveal the tiles for every map in the game.

The full code for this chapter is available in Appendix A.

Playrix Ex. 1024, page 159

4.3 Macro Bot

4.3.1 Target

In this chapter, we will switch our target to the game Wyrmsun, version 5.0.1. This is because Wesnoth, our target so far, does not support gameplay mechanisms (such as real-time control of building units) that would allow us to write a macro bot. Wyrmsun is free and similar to other traditional RTS games, such as StarCraft, WarCraft, or Command & Conquer.

4.3.2 Identify

A macro bot is a type of hack that will monitor our resources and automatically build worker units. In this chapter, we will create a macro bot that will automatically build a worker out of the currently selected structure when our money is over 3000.

4.3.3 Understand

To write a macro bot, we need to understand how RTS games handle unit creation. Typically, RTS games have a list of units associated with each player. When creating a unit, the game performs several operations and then adds the new unit to that list. The code may look something like:

```
recruit_unit(unit_type) {
    memory = initialize_memory(sizeof(unit))
    unit = create_unit(unit_type, memory)
    player->decrease_money()
    player->add_unit(unit)
    player->increase_population_counter()
}
```

To create units, we need to find this function and call it ourselves. To locate this function, we can use two different approaches, depending on how the game handles unit creation.

- If we create a unit and the game instantly decreases our money, we will locate our money and then set a breakpoint on write on our money value.
- If we create a unit and the game instantly increases our population, we will locate our population and then set a breakpoint on write on our current population.

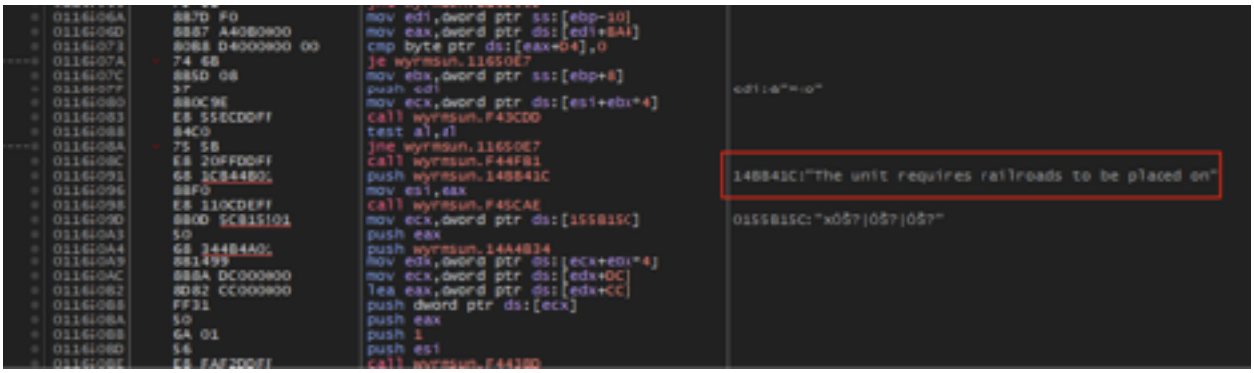
When these breakpoints trigger, we will be inside of the hypothetical **decrease_money** or **increase_population_counter** functions in our example code above. We will therefore need to go up several functions. We will do this by executing the function until it returns and then stepping out.

Wyrmsun (our target in this chapter) loads code dynamically. As a result, the addresses you see in this chapter will be different when following along. However, the instructions and methods described will not change. We will discuss how to deal with this behavior when we create our DLL.

4.3.4 Locating the Create Unit Function

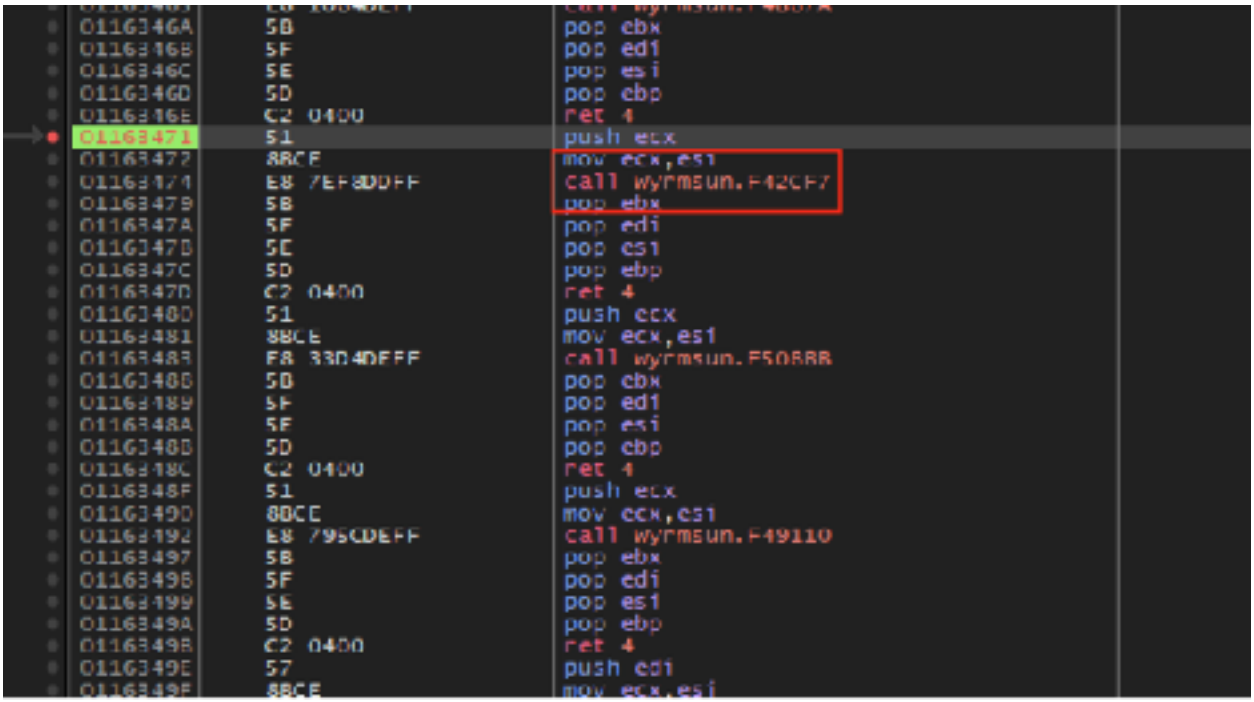
In Wyrmsun, money is decreased instantly when recruiting a unit, so we will use the first approach mentioned in the previous section. To find our money address for this new target, we can use the method discussed in Chapter 1.5.

Next, attach x64dbg to the game. Make sure that no operations are taking place that will alter your money, and set a hardware breakpoint on write on the money address. Recruit a worker and the breakpoint should instantly pop. Using execute until return/ step over, go up several levels of code until you see the following string:

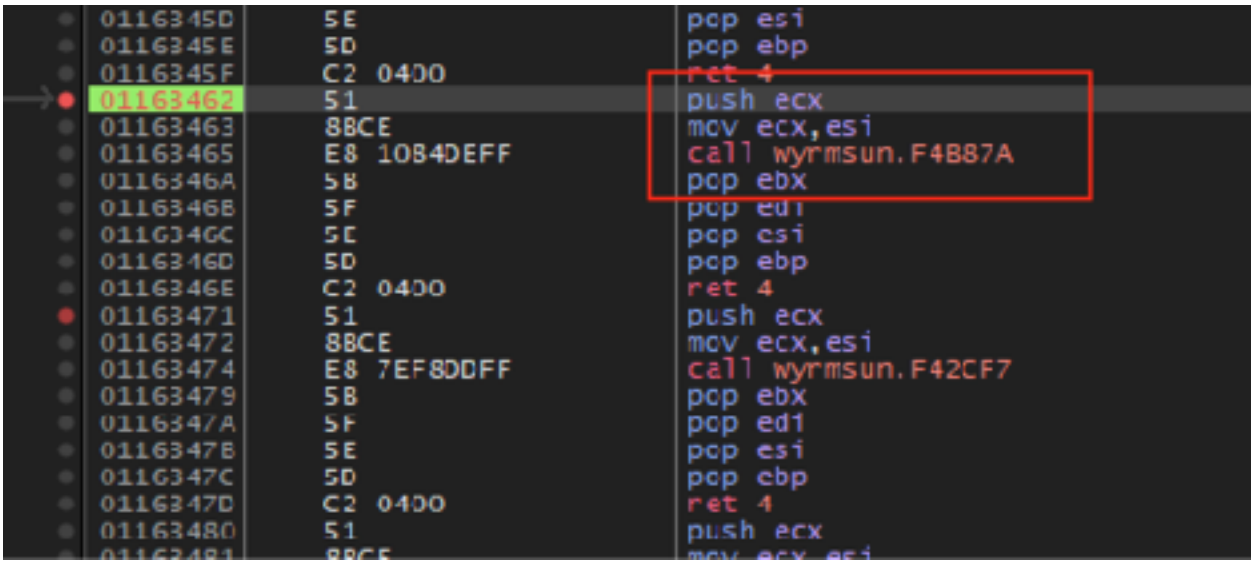


From this string, we can see that we are in the right place, as this logic is clearly related to recruiting and placing units.

From here, navigate up one more level to the parent calling function:



If we examine this code, we can see it is a series of very similar calls that take a single parameter. If we set a breakpoint on the **call** to `0xF42CF7`, we can see that it is called only when a unit is recruited. Next, place a breakpoint on the **call** above the **call** to `0xF42CF7`. With that set, do various actions in the game, such as moving units, attacking, and building. When conducting a build action, your breakpoint at the **call** above the unit recruitment function should pop:



Given this behavior, we can guess that all these calls are related to functions in the unit card (the bottom-right of the screen). We can imagine the code may look something like:

```
switch(menu_event) {
    case BUILD:
        build_structure(event_data);
        break;
    case RECRUIT:
        recruit_unit(event_data);
        break;
    case MOVE:
        move_unit(event_data);
        break;
}
```

Therefore, we can assume the **call** to `0xF42CF7` (in our example) is responsible for recruiting units. We can verify this by **nop**'ing out the following instructions:

```
push ecx
mov ecx, esi
call 0xF42CF7
```

When **nop**'d out, clicking the recruit button on a structure no longer creates a unit.

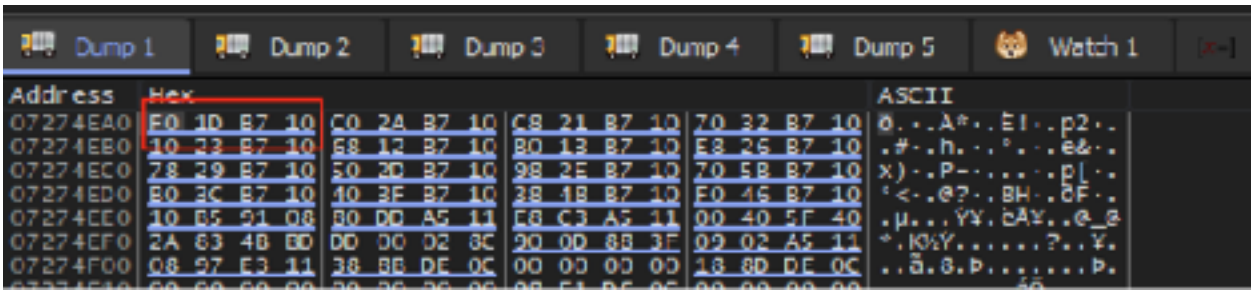
4.3.5 Reversing Event Data Structure

Now that we have located the unit creation function, we need to reverse the data provided to it so that we can call it ourselves. We can see that there are two pieces of data potentially being passed to the function:

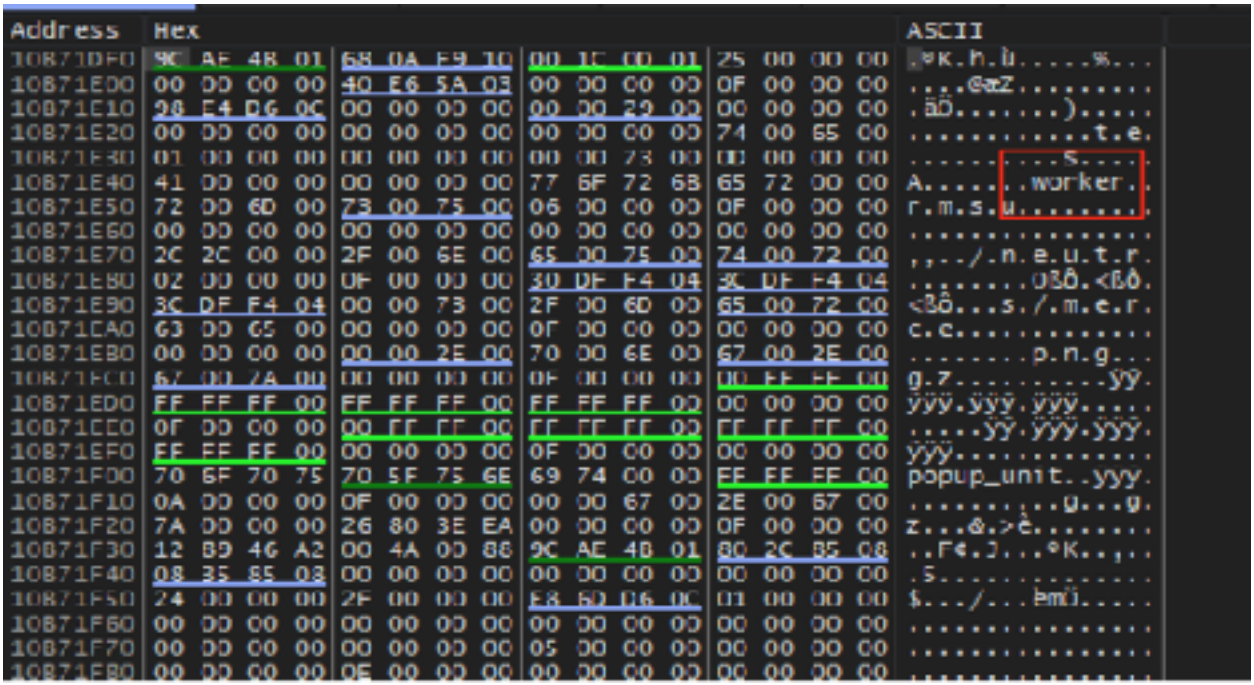
- A value in **ecx**, which is pushed on the stack
- A value in **esi**, which is moved into **ecx**

Let's determine if both of these are necessary. First, replace the **push ecx** instruction with another register, such as **push eax**. If you try to recruit a unit in the game, you will notice the game will instantly crash, indicating that the push is important. Next, restart the game and **nop** out the **mov ecx, esi** instruction. You will notice that the game responds normally, indicating that this operation is not used by the unit creation event. As a result, we only need to reverse the value of **ecx** when pushed.

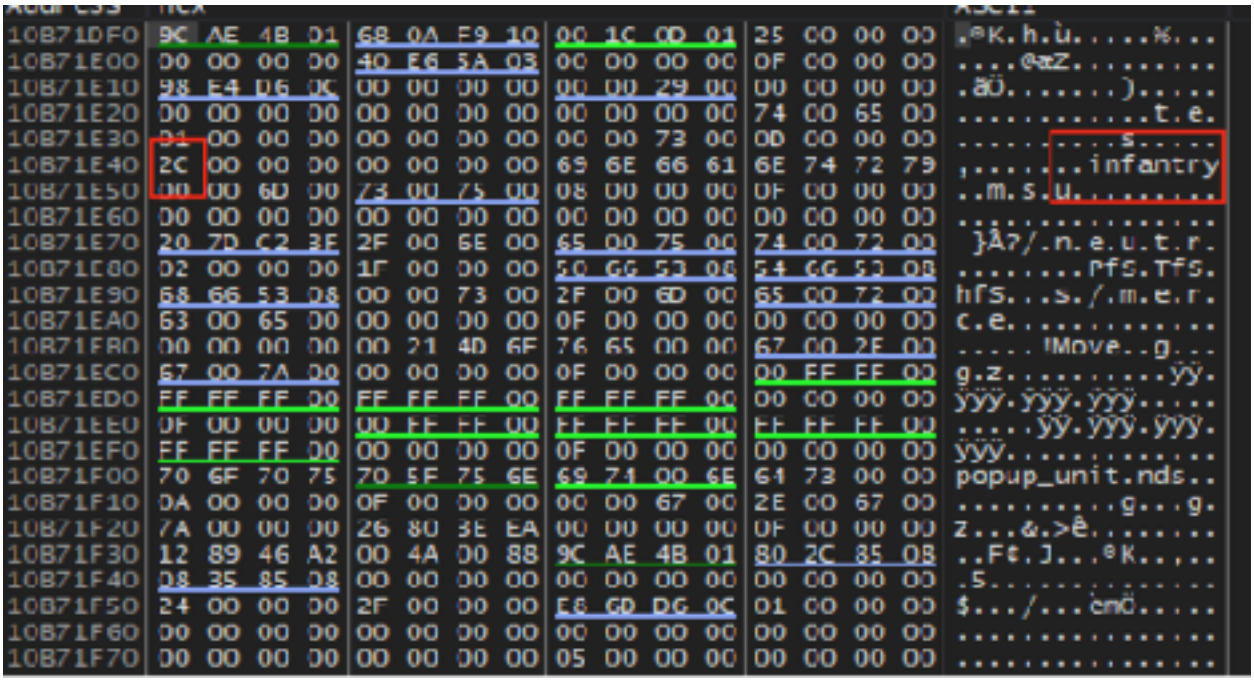
Set a breakpoint on the **push ecx** and recruit a worker. When the breakpoint pops, right-click on the value of **ecx** and choose *Follow in dump*:



This data does not appear to contain all of the information we would expect. The similar values of the data (the first three entries repeat `0xb710` at their end) indicate that this may be a pointer in a table of pointers. To validate this assumption, select the 4 bytes (`0xf01db710`) and choose *Follow in dump*. Your dump view should change to `0x10b71df0`, like so:



Immediately the text *worker* should jump out at you. It appears that this structure contains data on the unit to be created. Back in game, create another structure to create units (such as a War Hall, or barracks-type building) and create an infantry unit. When the breakpoint pops, examine the section of memory at `0x10b71df0` again:



We can see two main pieces of data changed between a worker and an infantry unit being recruited: the text (*worker* vs *infantry*) and the number beforehand (`0x41` vs `0x2c`). We can assume that this number may be the internal representation of the type of unit.

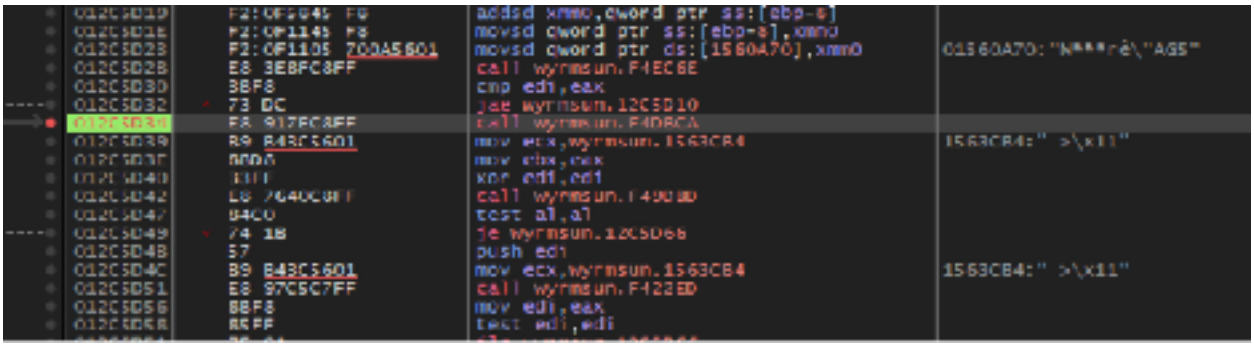
Let's verify that this works, as we are still guessing. First, set a breakpoint on the **push ecx** instruction and create an infantry unit. Next, when the breakpoint pops, change `0x2C` to `0x41` and *infantry* to *worker*. If you then resume the execution and go back in game, you can see that despite clicking infantry, we are now creating a worker out of the barracks structure.

There is also other data in this structure that changes but is not directly tied to the unit being created. While we could reverse the entire structure, we will instead copy the structure when a worker is created and use those values for our hack.

4.3.6 Locating the Main Game Loop

Now that we have located the function responsible for recruiting units and understand how to call it, we need to locate a place to call it from. For this chapter, we will choose to hook the main game loop. Finding the main game loop is easy in our case. First, place a breakpoint on the **call** to recruit a unit and create a unit in game so that the

breakpoint will pop. Next, continue to step out of each function. Eventually, you will reach the following **call**:

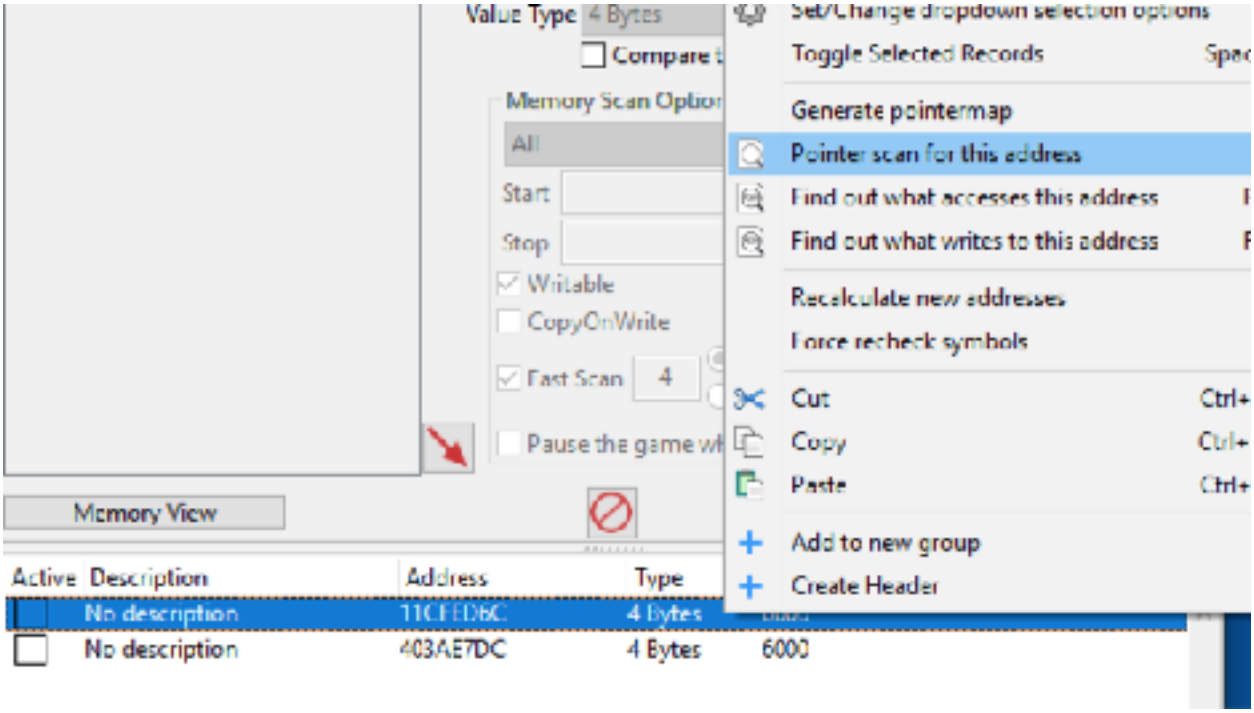


If you attempt to execute until return here, you will notice the game will begin and continue to execute. This is due to the fact that we are in a loop and no **ret** instruction is being encountered. We can verify this behavior by setting a breakpoint on this **call**. You will notice the breakpoint pops continuously. Both of these factors indicate that this code is part of the main game loop.

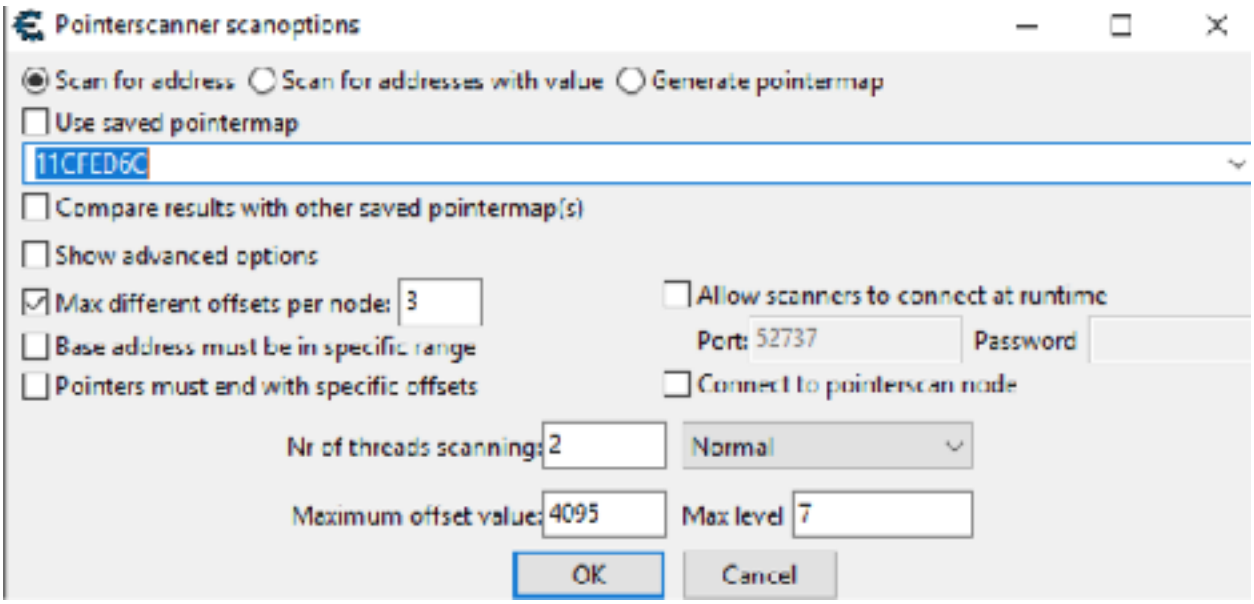
4.3.7 Locating the Player's Money

Finally, we want to monitor our player's money for our hack. Wyrmsun, like other games, allocates a player's money dynamically, meaning it will be different for each game. In previous chapters, we have discussed methods to defeat DMA. For this chapter, we will use Cheat Engine's pointer scan feature instead of reversing the target.

Cheat Engine's pointer scan works similar to regular memory scanning. First, we need to locate our money address as usual. Then, right-click on the address and choose *Pointer scan*:

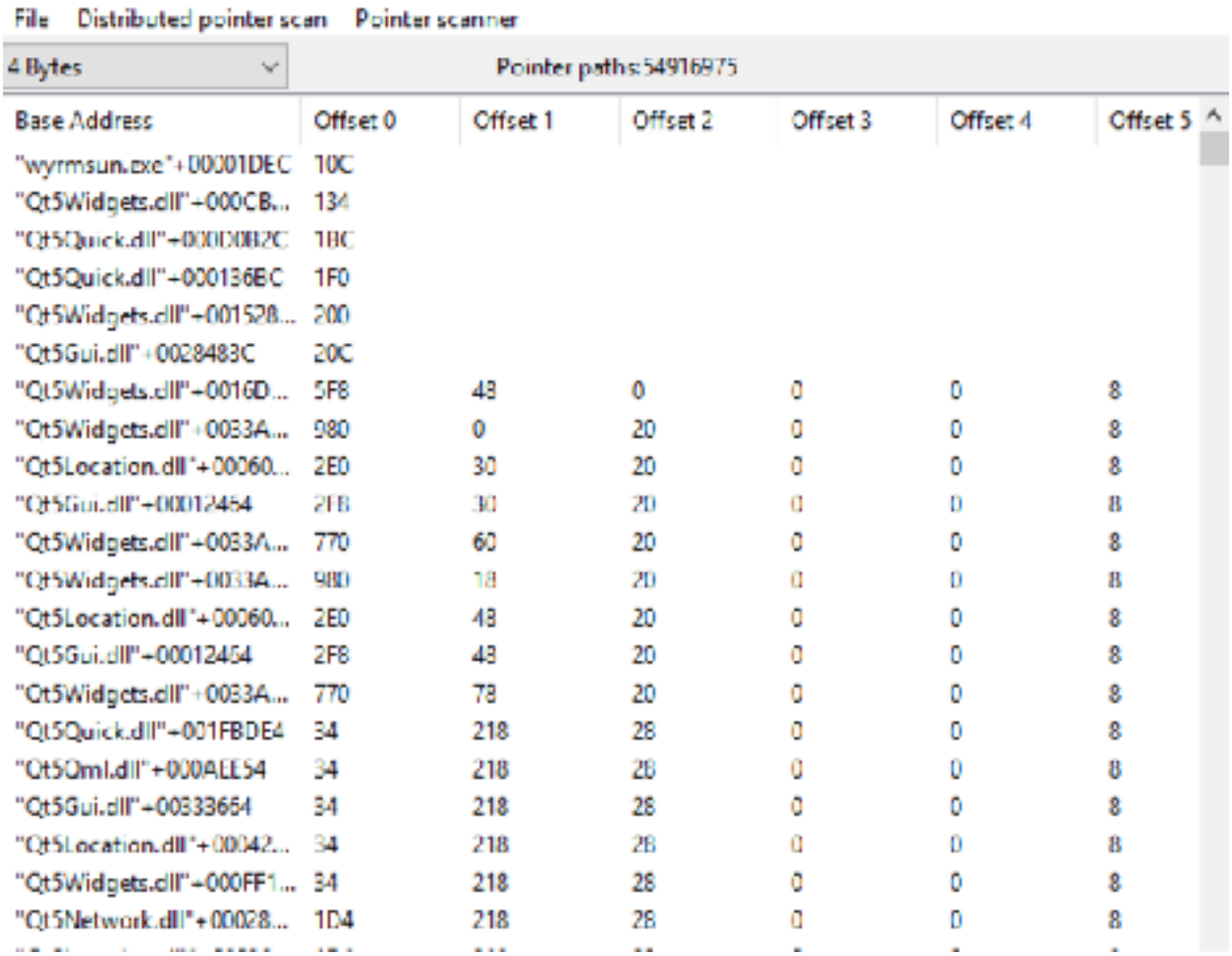


In the dialog that appears, keep all the default options and choose *OK*. When prompted, select a file anywhere:

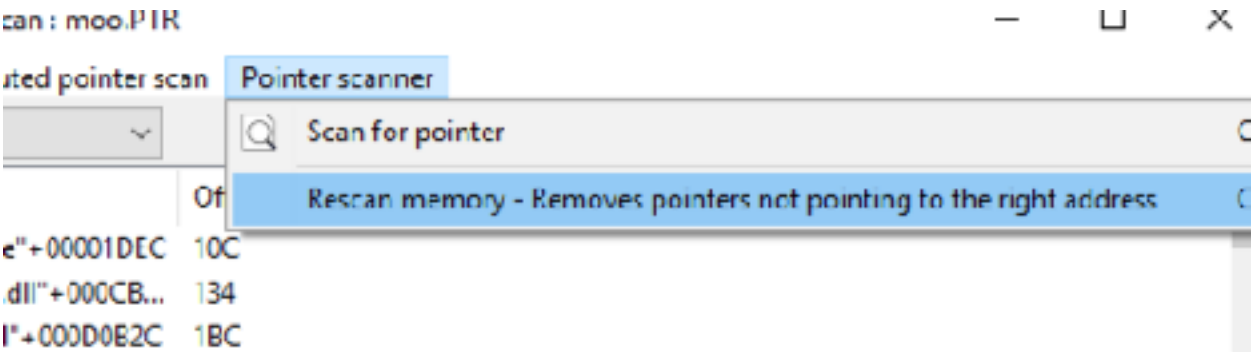


Playrix Ex. 1024, page 167

Cheat Engine will now search the target for all pointers that point to the selected address in some way. When it is finished, you will get thousands of results back:

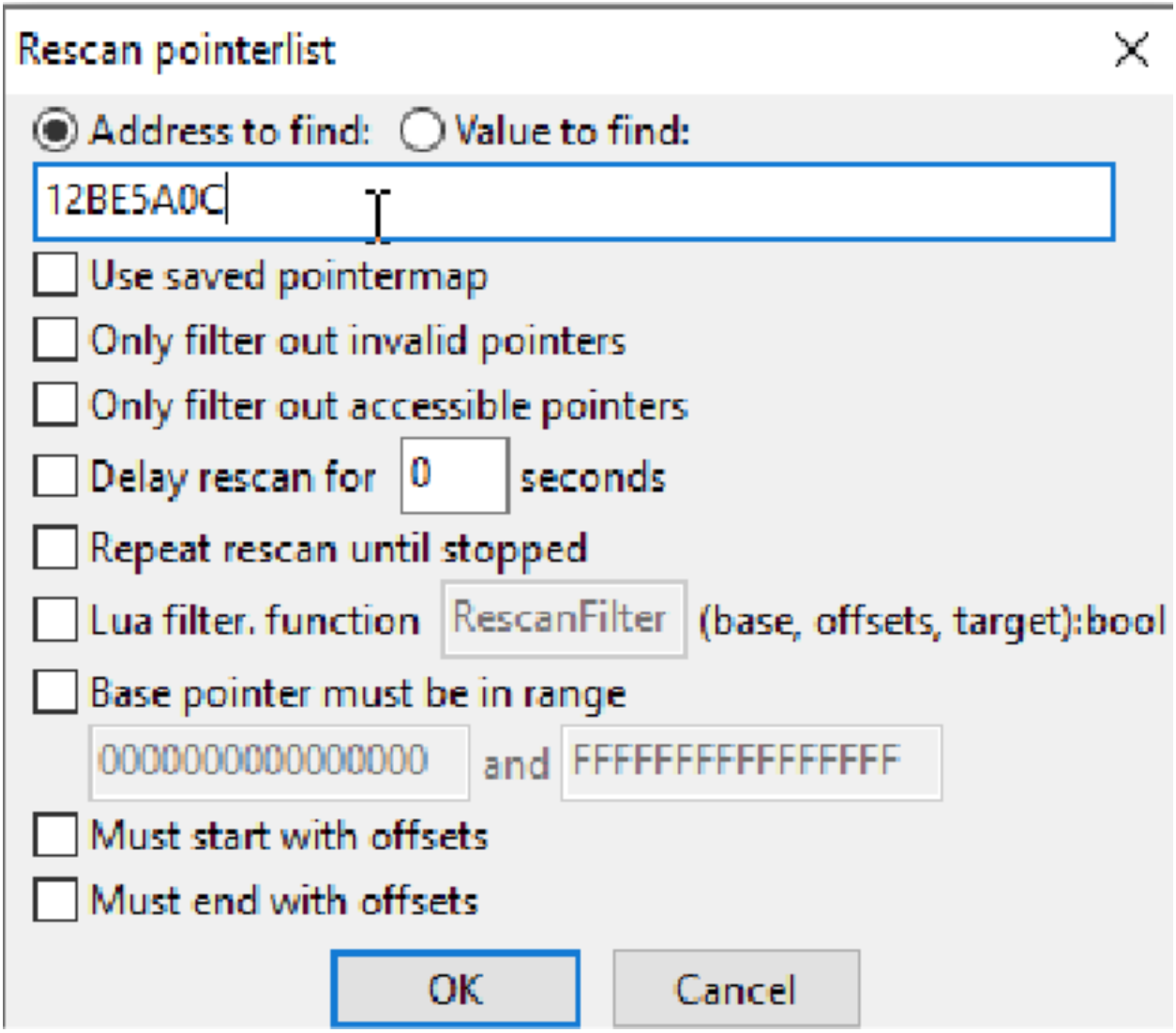


Like regular scanning, we now need to filter these addresses down. Restart your match so that your goal is moved to a new address. Next, find your money address again. Then, in the pointer scan window, choose *Rescan Memory*:



Playrix Ex. 1024, page 168

In the dialog that appears, enter your new address and hit *OK*:



Like regular filtering, Cheat Engine will now rescan all the previously identified pointers and see if they are still correctly pointing at your new address. Repeat this operation several times, and eventually you will find a few pointers that always correctly point to the player's money value. For this chapter, we will use:

```
+0x14
[+0]
[+0x4]
[+0x8]
[+0x4]
[+0x78]
wyrmsun.exe + 0x0061A504
```

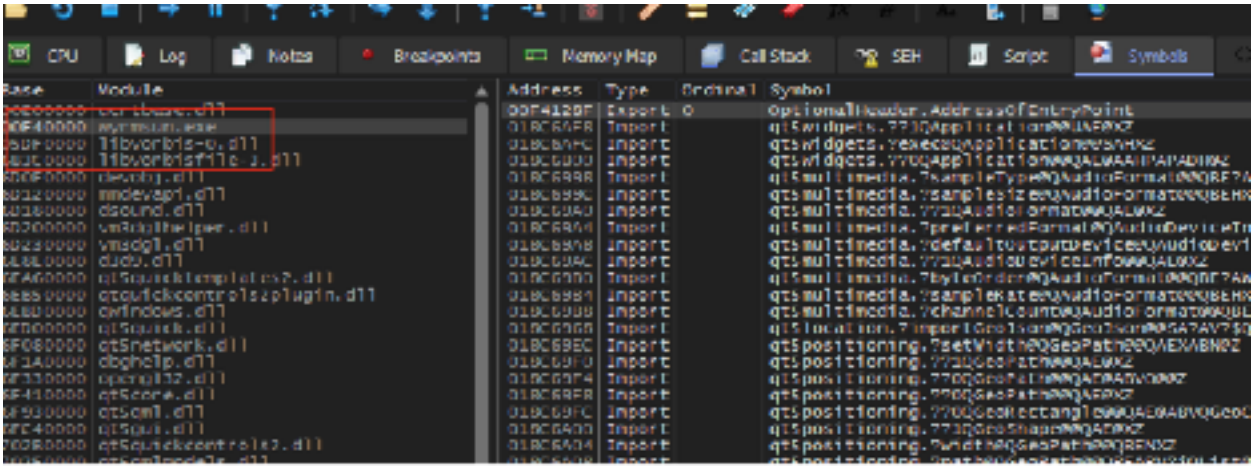
We will discuss how to use these values in our code, so feel free to substitute in whatever value you find.

4.3.8 Dealing with Dynamic Code

At the beginning of this chapter, we discussed how code was dynamically loaded and, as a result, addresses would not be consistent. You can verify this behavior by starting Wyrmsun, noting an address, restarting your VM, and opening Wyrmsun again. You will notice that the address no longer contains the same code.

Just like DMA, we know that the game must have some way to locate its code. When dealing with dynamic code, generally games will offset all addresses from the game's module base address. We can observe this behavior by looking at the creating unit **call**. While the first byte will change, the **call** always ends in `0x2CF7` (e.g., `0xF42CF7` or `0x292CF7`).

We can determine the base address of the main module by going into the *Symbols* tab in x64dbg:



We can see here that our base address is `0x00F40000`. As such, we know that the create unit function will exist at the `base address + 0x2CF7`. Likewise, we saw that in this chapter, the **call** to create unit was at `0x01163471`. If we subtract this address from the base address, we get an offset of `0x223471`. We can use these offsets when creating our DLL to automatically calculate the addresses of functions we care about.

4.3.9 Creating our DLL

Like in previous chapters, we will create a DLL to inject into our target. First, we will start with our base:

```
#include <Windows.h>

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      DWORD old_protect;

      if (fdwReason == DLL_PROCESS_ATTACH) {
        //hooking code here
    }

    return true;
}
```

Our DLL will have two code caves. First, we will create a code cave at the code responsible for creating a unit. In this code cave, we will retrieve the value of **ecx** and copy the structure it points to into our DLL's memory. Our second code cave will hook the main game loop. In this code cave, we will check the current player's money value and then call the create unit function.

4.3.10 Create Unit Code Cave

When we reversed the create unit function, we identified the structure that was pushed as an argument to the function. While we identified several components of this structure, we did not fully reverse it. Since this structure does not change when creating worker units, we will use a code cave to copy a valid form of the structure. We will then use this copied form in our main game loop code cave.

First, we will hook the address for creating a unit and direct it to our code cave, as we have done in previous chapters. Since the code's addresses change, we will determine the address based on the base address of the module. We will also use this base address to calculate the recruit unit call location:

```
HANDLE wyrmsun_base;
```

```
DWORD recruit_unit_ret_address;
DWORD recruit_unit_call_address;

...

wyrmsun_base = GetModuleHandle(L"wyrmsun.exe");

unsigned char* hook_location = (unsigned char*)((DWORD)wyrmsun_base +
0x223471);
recruit_unit_ret_address = (DWORD)hook_location + 8;
recruit_unit_call_address = (DWORD)wyrmsun_base + 0x2CF7;

VirtualProtect((void*)hook_location, 8, PAGE_EXECUTE_READWRITE,
&old_protect);
*hook_location = 0xE9;
*(DWORD*)(hook_location + 1) = (DWORD)&recruit_unit_codecave -
((DWORD)hook_location + 5);
*(hook_location + 5) = 0x90;
*(hook_location + 6) = 0x90;
*(hook_location + 7) = 0x90;
```

In our recruit unit code cave, we will first retrieve the value of **ecx** and place it in a variable:

```
DWORD* base;

__declspec(naked) void recruit_unit_codecave() {
    __asm {
        pushad
        mov base, ecx
    }
```

With this pointer now stored in the **base** variable, we can dereference this pointer to retrieve the location of the structure. With the pointer dereference, we can then copy the entire structure into another variable to use in our other code cave. We can retrieve the size by observing the size of the structure in x64dbg. Additionally, we will create an **init** variable to track whether this has occurred yet:

```
DWORD* unitbase;

unsigned char unitdata[0x110];
bool init = false;
```

```
...

unitbase = (DWORD*)(*base);
memcpy(unitdata, unitbase, 0x110);
init = true;
```

Finally, we will restore our registers and the original instructions:

```
_asm {
        popad
        push ecx
        mov ecx, esi
        call recruit_unit_call_address
        jmp recruit_unit_ret_address
    }
}
```

4.3.11 Game Loop Code Cave

With our data copied into a buffer, we can now create our game loop code cave. Like before, we will begin by hooking the address that we identified earlier:

```
DWORD gameloop_ret_address;
DWORD gameloop_call_address;

hook_location = (unsigned char*)((DWORD)wyrmsun_base + 0x385D34);
gameloop_ret_address = (DWORD)hook_location + 5;
gameloop_call_address = (DWORD)wyrmsun_base + 0xDBCA;

VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
*hook_location = 0xE9;
*(DWORD*)(hook_location + 1) = (DWORD)&gameloop_codecave -
((DWORD)hook_location + 5);
```

In our game loop code cave, we will first check the value of our player's money. We will use the pointer and offset that we received from Cheat Engine to do this:

```
DWORD *gold_base, *gold;
```


Playrix Ex. 1024, page 173

```
...
__declspec(naked) void gameloop_codecave() {
    __asm {
        pushad
    }

    gold_base = (DWORD*)((DWORD)wyrmsun_base + 0x0061A504);
    gold = (DWORD*)(*gold_base + 0x78);
    gold = (DWORD*)(*gold + 4);
    gold = (DWORD*)(*gold + 8);
    gold = (DWORD*)(*gold + 4);
    gold = (DWORD*)(*gold);
    gold = (DWORD*)(*gold + 0x14);
```

Next, we will check to see if our unit buffer has been initialized and if the player's money is over 3000. If so, we copy our buffer for the worker into the buffer pointed to by the game, and move the base into **ecx** before calling the recruit unit function:

```
if (init && *gold > 3000) {
    memcpy(unitbase, unitdata, 0x110);
    __asm {
        mov ecx, base
        push ecx
        call recruit_unit_call_address
    }
}
```

Once again, we need to restore the original instructions:

```
__asm {
        popad
        call gameloop_call_address
        jmp gameloop_ret_address
    }
}
```

Finally, we can build this DLL and inject it into our game. In game, recruit a unit to copy our buffer and then start collecting money. You should notice that workers begin to get recruited instantly.

The full source code for this chapter is available in Appendix A.

Playrix Ex. 1024, page 174

Part 5
FPS Hacks

Playrix Ex. 1024, page 175

5.1 3D Fundamentals

5.1.1 Overview

In the previous chapters, we focused on hacking two-dimensional (2D) games. While many of the techniques we have covered can be applied to any game, there are unique techniques that only apply to three-dimensional (3D) games. To make hacks like wallhacks or aimbots for 3D games, we need to understand how 3D games actually work.

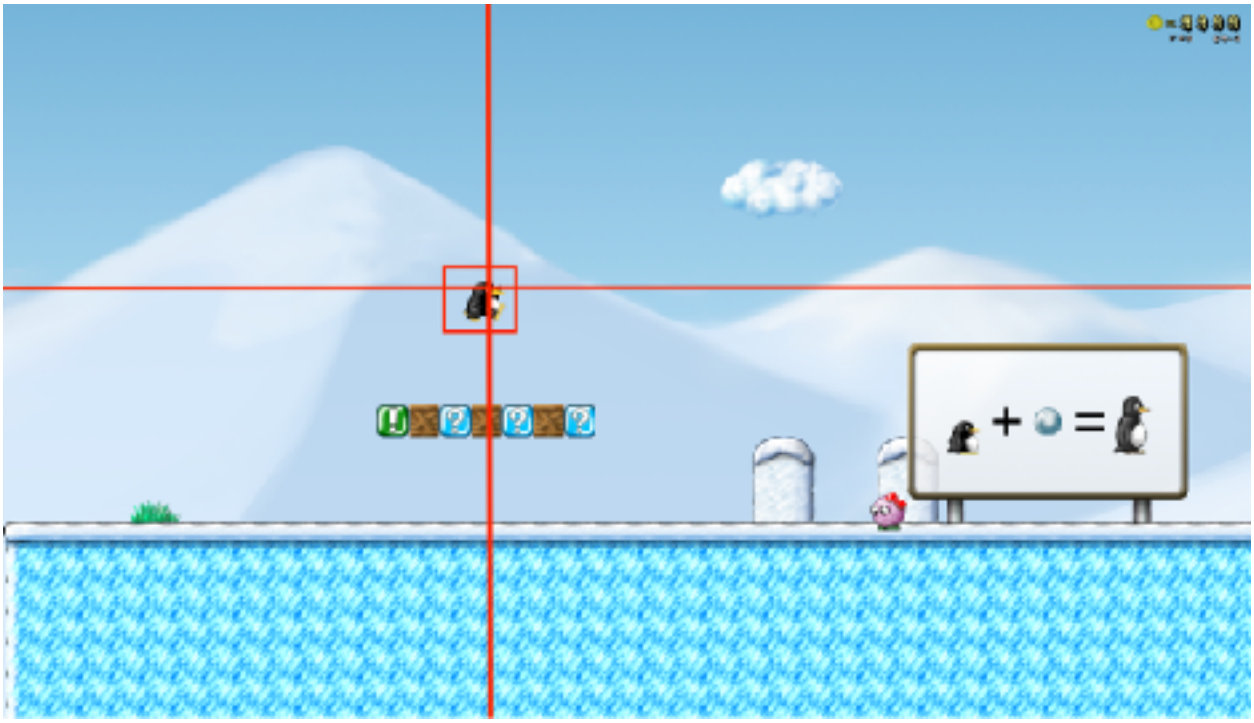
5.1.2 Coordinates

When we say a game is 2D, we are referring to the fact that all objects in the game can be located by a coordinate pair. These coordinate pairs contain two values: X (horizontal position) and Y (vertical position). Coordinate pairs are usually referenced with parentheses around them, like (X, Y).

Using the screenshot from Wesnoth as an example below, let's imagine we had a point (0, 0) in the bottom-left of the screen and a point (10, 10) in the top-right. The highlighted unit could be represented by the coordinate (7, 5) and the un-highlighted unit could be represented by the coordinate (3, 1).



The game uses these coordinates for many critical operations. For example, when a player attempts to move, the game will verify that the player's new coordinates will not be in water or impassable terrain. All 2D games use coordinates in this manner, whether the game has a top-down or side view.



5.1.3 3D Space

When playing Wesnoth, one thing you may notice is that two units can never share the same coordinates. This is because the game would not be able to properly display each unit to the player without having special logic to handle switching between the two images. However, in 3D games, two units can share the same horizontal and vertical coordinates.

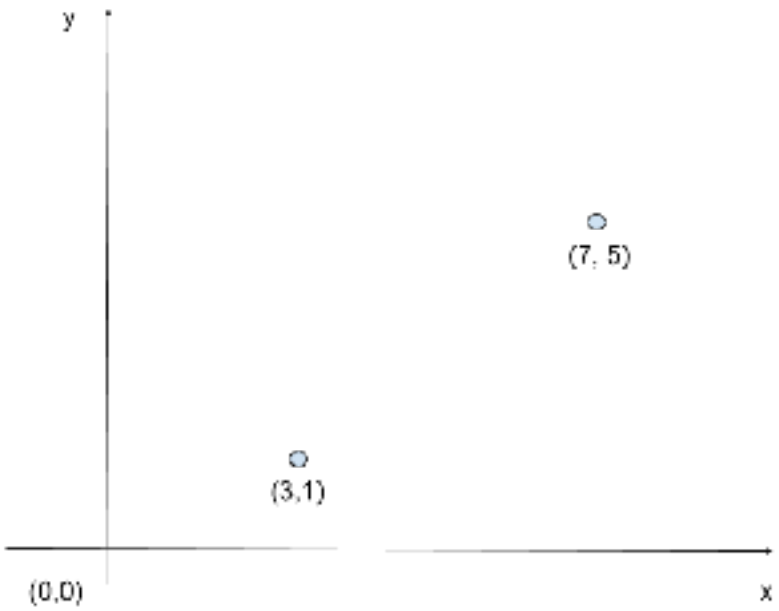


As you can see, the other player and our current player are both in the middle of the screen and have the same (X, Y) values. However, in 3D games, coordinates are represented with three values: X, Y, and also Z (depth). In the example above, both players could be at (5, 5) in 2D space, but their 3D coordinates could be (0, 0, 0) and (0, 0, 5).

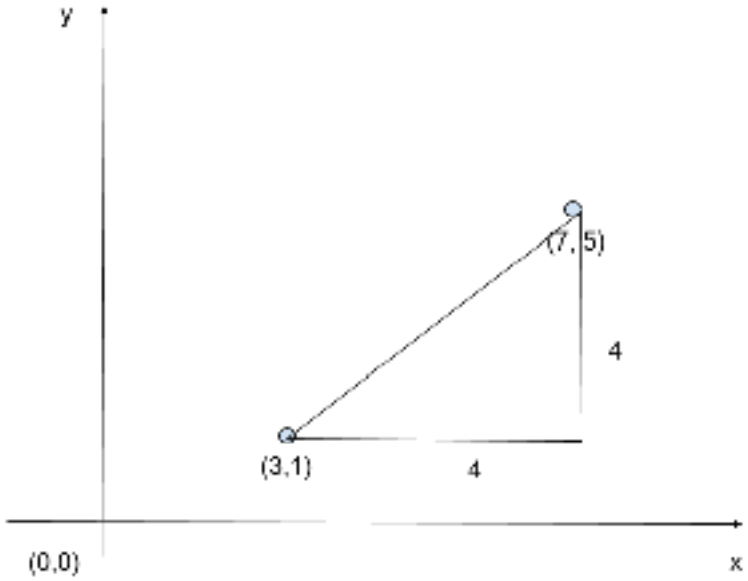
5.1.4 Cartesian Coordinates

One of the easiest ways to understand the relationships between coordinates is through the use of Cartesian coordinate systems. For example, we could graph our first Wesnoth example like so:

Playrix Ex. 1024, page 178

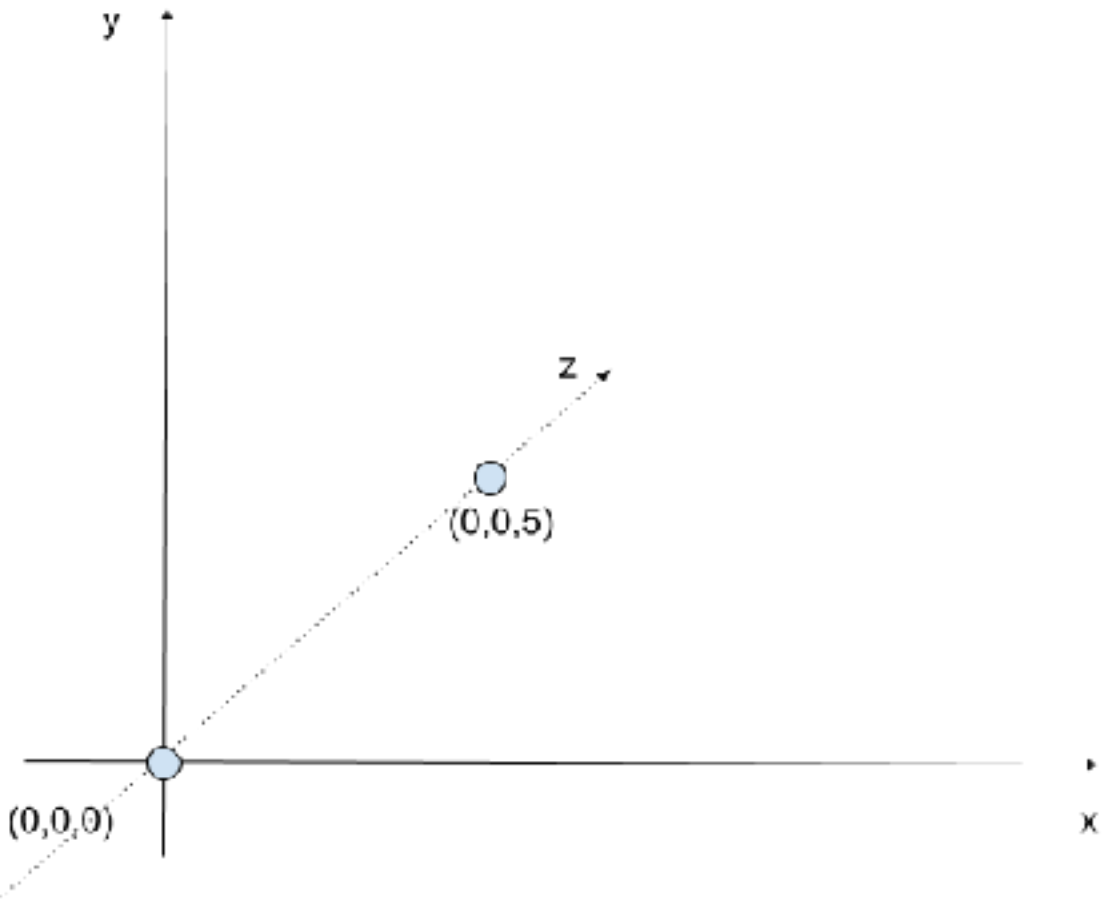


The strength of visualizing the coordinates like this is that we can then use normal geometric operations. Let's say we wanted to get the distance between these two units. By creating a right triangle from the two units, we can use the Pythagorean theorem to calculate that triangle's hypotenuse. Due to the way we created this triangle, this hypotenuse would represent the distance between these two units:



Playrix Ex. 1024, page 179

3D coordinates can also be graphed with the addition of another axis. Our 3D game example above might be graphed like:



Notice how, despite each player having identical X and Y coordinates, they exist in different places on the graph.

5.1.5 Viewports

Monitors display a 2D image on a flat screen. Therefore, it is impossible for monitors to render a 3D scene directly. Instead, the 3D world must be converted into a 2D scene, like taking a picture. Games will often have functions for this, typically called some variation of **WorldToScreen**. Sometimes, when programming hacks such as displaying text above a player's head, you will need to write this code yourself. In Chapter 5.9, we will discuss how to write this code for any game.

A key aspect of 3D-to-2D conversion is that games will choose a viewport, or a view into the game's world. This will often be the current player's view, but in games that support free cameras, this could be any position. For this viewport, the game will then calculate the depth for all objects in the scene. It will also draw objects that are farther away "behind" objects that are closer. For example, in the following viewport, the game first draws the building in the background. It then determines that the trees are in "front" of the building in the current viewport and draws them on top of the building. In this way, the game achieves the illusion of depth.



5.1.6 3D Movement

Operations in 3D space are expensive to compute. Because of this, games will often take several shortcuts to optimize their performance. One these shortcuts is always placing the player at the origin, or point (0, 0, 0). This way, all distances and angles for objects can be calculated by just retrieving that object's coordinates instead of having to subtract the object's position from the player's position. However, if the player is

Playrix Ex. 1024, page 181

stuck at (0, 0, 0), they will be unable to move. To achieve the illusion of movement, some games will instead rotate the entire world around the player. For example, if you press the key to move forward, the game will respond by moving the whole world toward you instead of moving your player forward. Not all games work like this, but several well-known ones use this model.

Playrix Ex. 1024, page 182

5.2 Wallhack (Memory)

5.2.1 Target

Since we are shifting to a new dimension, we have to shift to a new target. Several of the following chapters will be targeting Urban Terror 4.3.4. This game is an FPS based on the Quake engine.

Like Wesnoth, this game is open-source and has no integrated anti-cheat. It also runs well on low-spec hardware. Unlike Wesnoth, the Chocolatey package is broken. Due to this, the best way to install the game is to download and run the installer from the site.

You will need to enable 3D acceleration in VirtualBox for the game to function. Depending on your computer's hardware, it may not be possible for your machine to run a 3D game inside a VM. In this case, you have several options. Some are better than others:

1. Explore another hypervisor, like VMWare or Hyper-V.
2. Use another machine as a dedicated hacking computer and isolate it from your home network.
3. Find another target game with even fewer requirements and follow along with the concepts of the following chapters.
4. Partition your hard-drive and dual-boot. Even if you encrypt your personal drive, it is possible for malicious tools to access your personal data.
5. Run the target and tools on your personal machine and hope that nothing malicious happens.

5.2.2 Identify

Our goal in this chapter is to create a wallhack, a type of hack that allows us to see other players through walls. We will not modify any of the graphics functions of the

Playrix Ex. 1024, page 183

game. Instead, we will use the game's rendering logic and modify sections of the game's memory.

5.2.3 Understand

In 3D games, depth testing is used to determine when an item should be visible in the player's viewport. For example, if a player is behind a wall, depth testing will tell the rendering logic of the game to not draw the player.

All wallhacks operate on the principle of disabling depth testing. One way to do this is by hooking the graphics library of the game and disabling depth testing through library functions. We will cover this approach in the next chapter. In this chapter, we will rely on the game's built-in rendering logic to achieve our goal.

Games have to draw many dynamic objects, including players, weapons, and map assets like doors. These objects are normally referred to as entities. To simplify development and increase performance, games will often use the same function for drawing all of these entities.

However, these entities often have different rendering considerations. A game may want to draw shadows on characters, but not on static entities like doors that can be opened. Games will often have structures for each entity and store these rendering considerations in the entity's structure. When the entity is rendered, the game will check this member and render the entity according to it.

For some entities, like puddles of water or glass, games will want to disable depth testing. Because of this, the render member in the entity class will have a disabled depth testing value. If we can locate the function responsible for drawing entities and then modify all entities to contain this disabled depth testing value, players will appear through walls.

5.2.4 Target Setup

All games that are based on the Quake engine have a console. This console can be accessed by hitting the tilde (**~**) key while in game. This console allows you to run commands, such as moving the player or changing a map. These commands typically start with a backslash (****) and can be auto-completed by hitting tab. Some helpful commands for our purposes are:

- *devmap abbey* - start the map Abbey with cheats enabled

- \g_gametype 0 - set the default game mode to deathmatch
- \bot_enable 1 - enable bots to join a game
- \reload - restart the current map
- \addbot boa 1 - add a bot

In addition to these commands, we can easily switch the game to a windowed mode by hitting *Alt+Enter*.

5.2.5 Locating Draw Entities

By exploring the commands available to us, we can find several drawing commands under \r_:



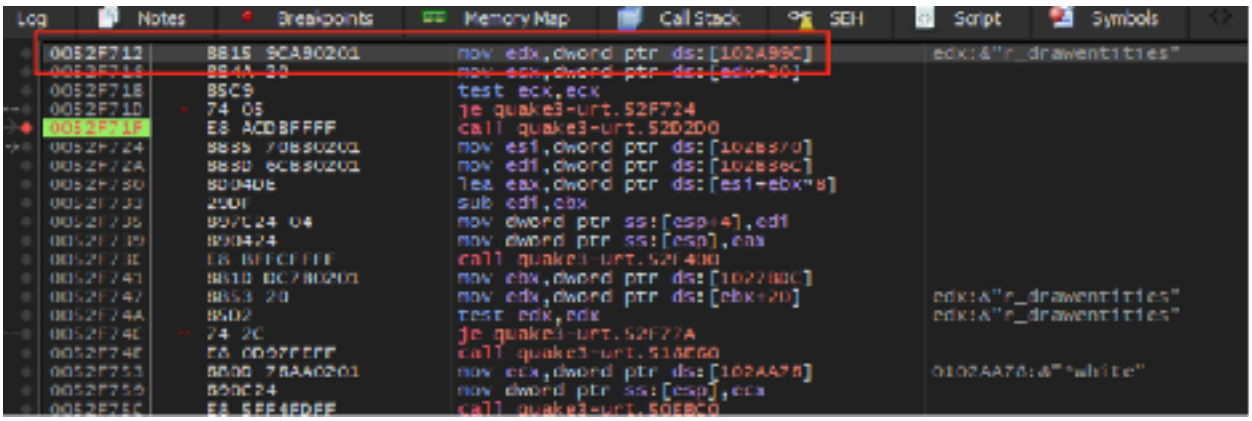
The most important command to us is *r_drawentities*. By setting this value to 0, entities are not drawn in the game, including our player:



We can assume that the game's code looks something like:

```
if(r_drawentities == 1) {
  draw_entities();
}
```

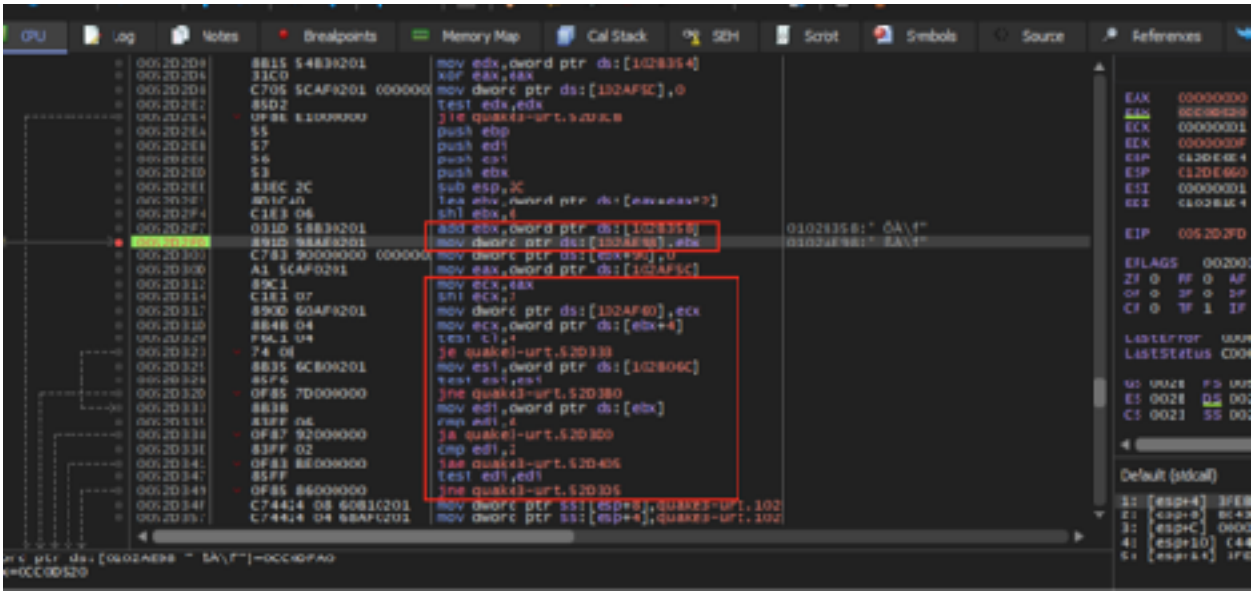
To find this code, we will use Cheat Engine to find the address of the variable holding the **r_drawentities** value. We can switch the value of **r_drawentities** in the console from 0 to 1 to narrow this value down. Then, we can use a breakpoint on access in x64dbg to locate the code that accesses this value. The breakpoint should pop at the following code:



We can see that the value of **r_drawentities** is loaded into **ecx** and then tested. Testing a register against itself compares that register's value to 0. If the value is equal to 0, the game jumps over the **call** at 0x52F717. This **call** is most likely responsible for drawing entities in the game. We can confirm this by **nop**'ing out this **call**. When it is **nop**'d, the game will not draw any entities.

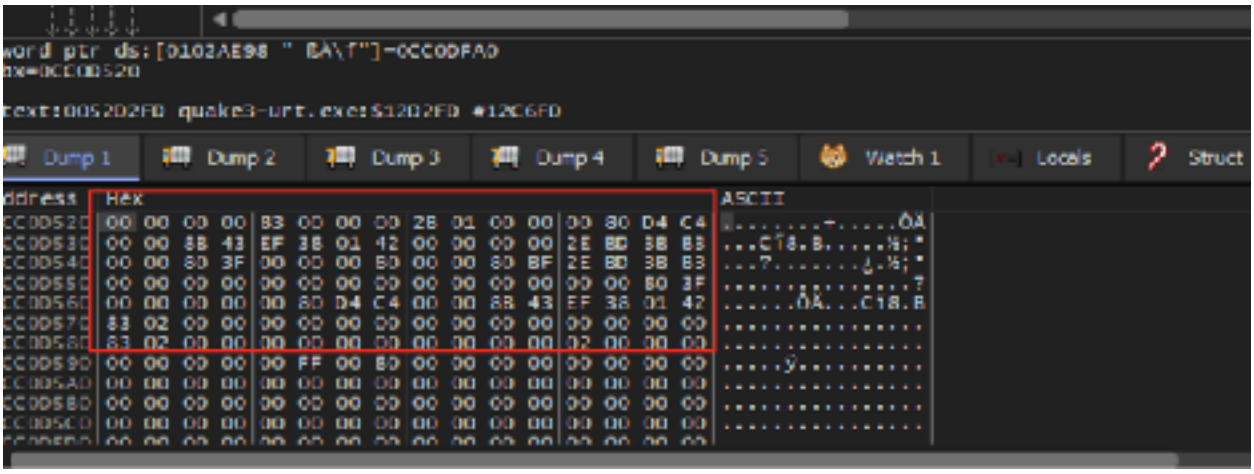
5.2.6 Entities and Rendering

If we step inside the **call** at 0x52F717, we see the following code:



We can see in the second highlighted block that values are loaded into several registers and compared to certain values. If these values are equal, the game jumps to different locations and executes different rendering code. If we look closely, we can see

that the registers are based on values of the address held in **ebx**. If we look up at the first highlighted block, we find the closest location in which **ebx** is set. We now know that at address `0x52D2FD`, **ebx** contains what is most likely the current entity to render. If we set a breakpoint at this address and observe **ebx**'s address in the dump, we see a chunk of data:



Since this chunk of data is isolated from other data and in one continuous section, we can assume that it represents a structure of some type. For example, it might look something like:

```
struct entity {
  int type;
  int render_type;
  float location[3];
  ...
}
```

To determine what location of the structure holds the render type, we must reverse this structure.

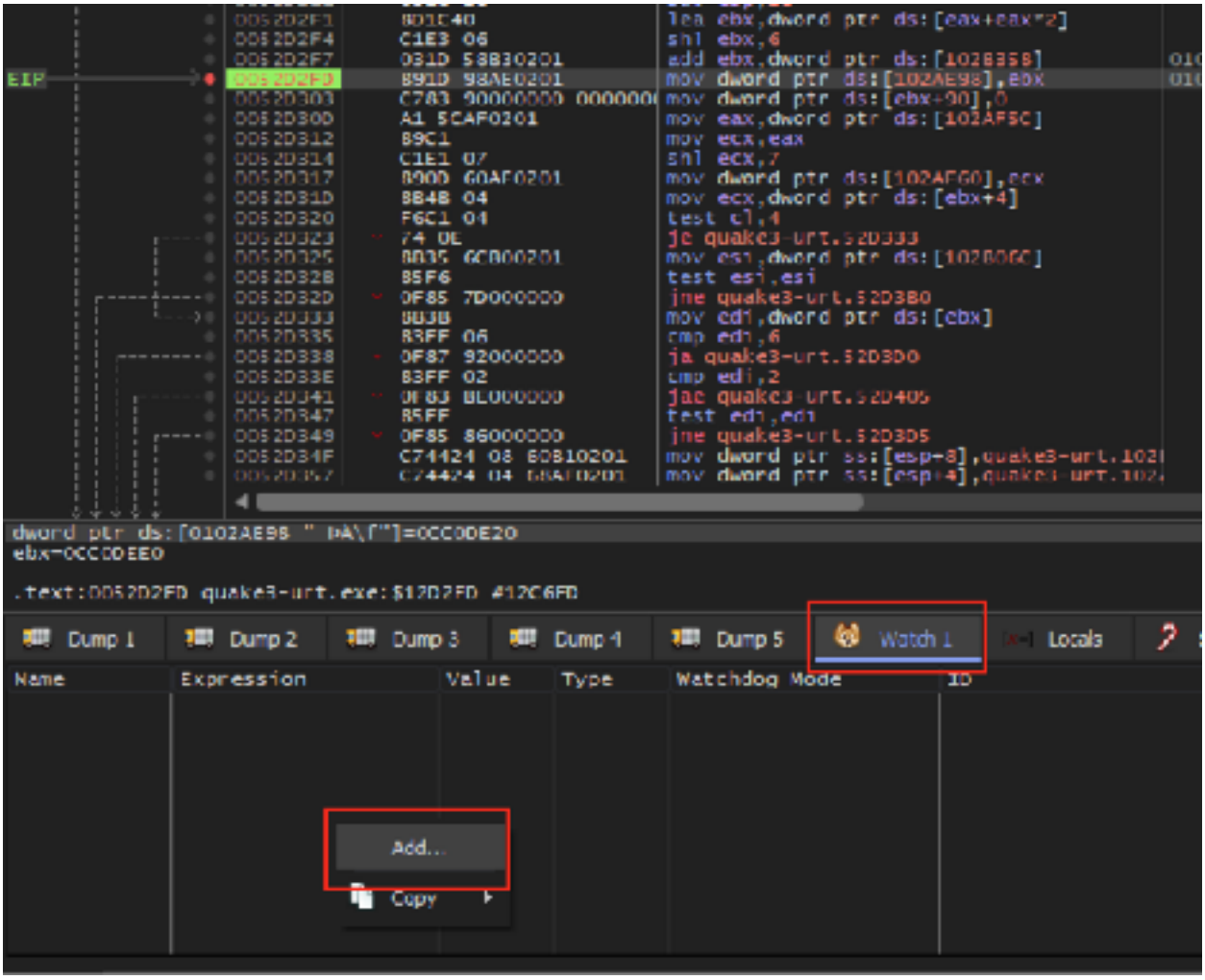
5.2.7 Reversing the Entity Structure

There are many ways to reverse an unknown structure in a game. One way is to build up a dataset of valid values and then make inferences based on these values. For example, if all the structures contain one member that constantly increases, we can assume that this member is being used as a counter of some type.

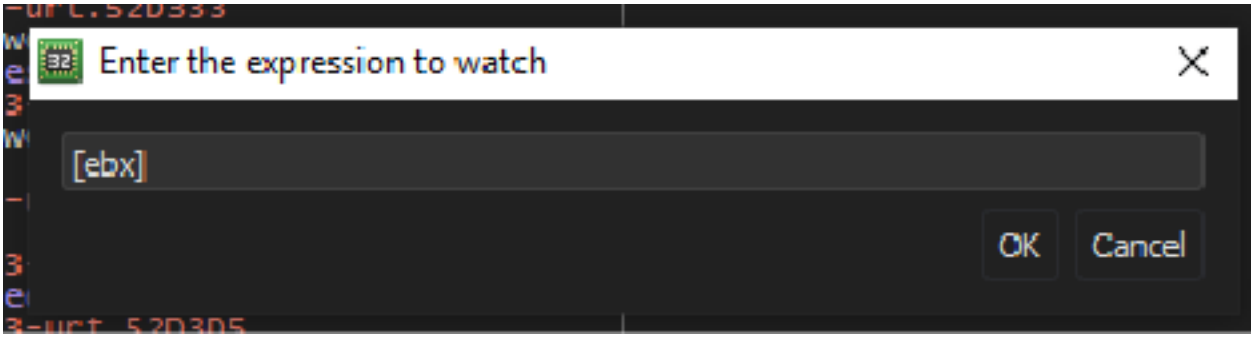
In this case, our goal is not to fully reverse the entity structure, but to only reverse enough to find the render type variable. Since we have located the code responsible for drawing entities, we can set a breakpoint in that code and observe entity structures. Like we discussed in the last section, at address `0x52D2FD`, **ebx** holds the address of the current entity to render.

You will notice that each time our breakpoint is hit, **ebx** contains a different value. While we could manually follow **ebx** in the dump each time the breakpoint is hit, a more convenient way is to use the W*atch* feature of x64dbg. Adding an expression to the *Watch* panel allows us to observe it independently of the dump. In this case, we can watch the expression **[ebx]** and always view the current value of the address in **ebx**.

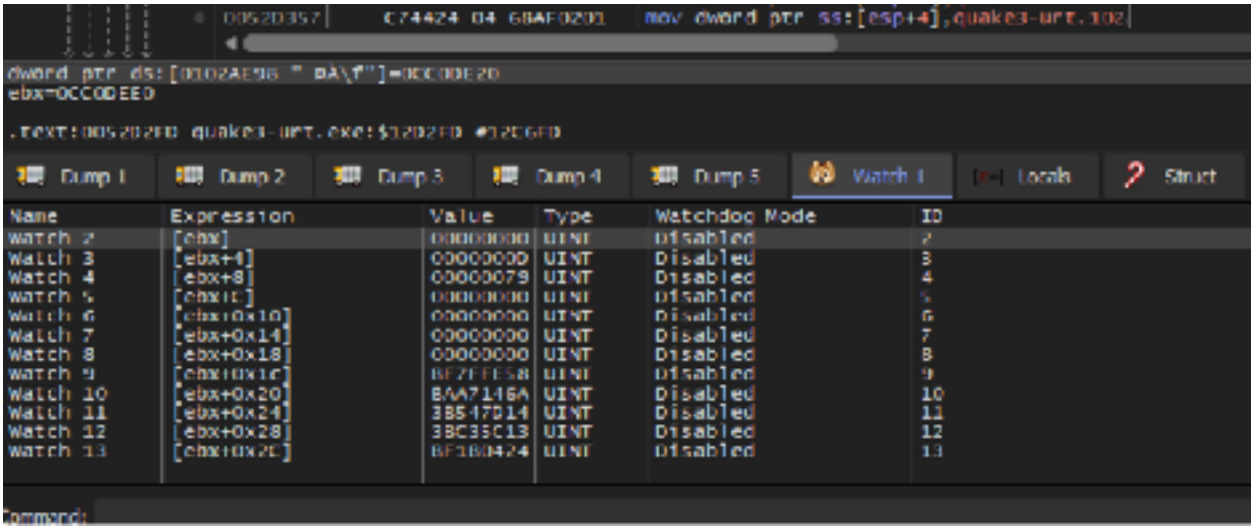
To add a value to watch, open up the *Watch* panel (near the dumps), right-click, and choose *Add*:



In the modal that appears, type your expression. In this case, we want to start with just **[ebx]**:



We want to also observe the first chunk of the entity structure. For now, we will assume that all these values are 4 bytes long. Add watches for **[ebx+4]** through **[ebx+2C]**. After you are finished, the *Watch* panel should look like:

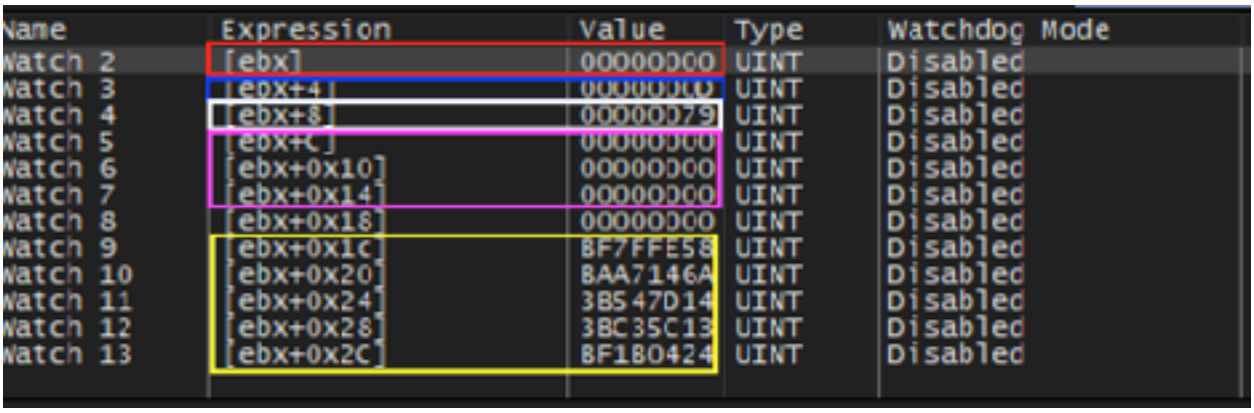


With all of this set up, disable your breakpoint and load into a map with water. The map *Abbey* has a fountain near the top-left of the map. Make sure you are facing the water and can see the ground beneath it.

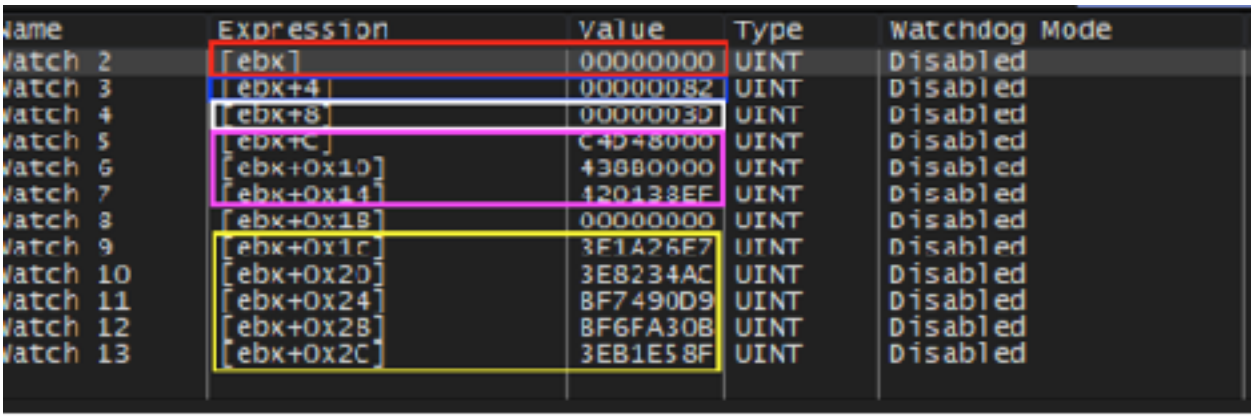


With all of this set up, re-enable your breakpoint at `0x52D2FD` and it should pop instantly. After observing the value of the watch panel, continue execution. After observing many iterations, you should start to notice some trends.

ame	Expression	Value	Type	Watchdog Mode
atch 2	[ebx]	00000000	UINT	Disabled
atch 3	[ebx+4]	00000000	UINT	Disabled
atch 4	[ebx+8]	00000078	UINT	Disabled
atch 5	[ebx+C]	00000000	UINT	Disabled
atch 6	[ebx+0x10]	00000000	UINT	Disabled
atch 7	[ebx+0x14]	00000000	UINT	Disabled
atch 8	[ebx+0x18]	00000000	UINT	Disabled
atch 9	[ebx+0x1C]	BF7FFDCA	UINT	Disabled
atch 10	[ebx+0x20]	BB25B2C4	UINT	Disabled
atch 11	[ebx+0x24]	3B9C493A	UINT	Disabled
atch 12	[ebx+0x28]	3BE0A705	UINT	Disabled
atch 13	[ebx+0x2C]	BF1AFADF	UINT	Disabled



Name	Expression	Value	Type	Watchdog Mode
Watch 2	[ebx]	00000000	UINT	Disabled
Watch 3	[ebx+4]	00000000	UINT	Disabled
Watch 4	[ebx+8]	00000079	UINT	Disabled
Watch 5	[ebx+C]	00000000	UINT	Disabled
Watch 6	[ebx+0x10]	00000000	UINT	Disabled
Watch 7	[ebx+0x14]	00000000	UINT	Disabled
Watch 8	[ebx+0x18]	00000000	UINT	Disabled
Watch 9	[ebx+0x1C]	BF7FFE58	UINT	Disabled
Watch 10	[ebx+0x20]	BAA7146A	UINT	Disabled
Watch 11	[ebx+0x24]	3B547D14	UINT	Disabled
Watch 12	[ebx+0x28]	3BC35C13	UINT	Disabled
Watch 13	[ebx+0x2C]	BF1B0424	UINT	Disabled



Name	Expression	Value	Type	Watchdog Mode
Watch 2	[ebx]	00000000	UINT	Disabled
Watch 3	[ebx+4]	00000082	UINT	Disabled
Watch 4	[ebx+8]	00000083	UINT	Disabled
Watch 5	[ebx+C]	C4D48000	UINT	Disabled
Watch 6	[ebx+0x10]	43BB0000	UINT	Disabled
Watch 7	[ebx+0x14]	420138EF	UINT	Disabled
Watch 8	[ebx+0x18]	00000000	UINT	Disabled
Watch 9	[ebx+0x1C]	3F1A26F7	UINT	Disabled
Watch 10	[ebx+0x20]	3E8234AC	UINT	Disabled
Watch 11	[ebx+0x24]	BF7490D9	UINT	Disabled
Watch 12	[ebx+0x28]	BF6FA30B	UINT	Disabled
Watch 13	[ebx+0x2C]	3EB1E58F	UINT	Disabled

The value of **[ebx]** (red) always appears to be 0. The value of **[ebx+4]** (blue) appears to alternate between `0xD`, `0x40`, `0x82`, and `0x83`. The value of **[ebx+8]** (white) appears to increase consistently, from `0x79` to `0x80` to `0x81`, and so on. The values highlighted in pink appear to alternate between seemingly random values and 0. Likewise, the values highlighted in yellow appear to be random, yet consistently tied to **[ebx+8]**.

All this data can be overwhelming, but we can make sense of it by eliminating values we do not care about. We know that we have at least three entities on the screen: our player model, our weapon, and the water. We can assume there are probably other entities, such as doors, as well. Since most of these entities share many similarities, we want to look for data that is relatively consistent between at least two entities. However, we also know that some of the entities should not share this value.

With this model, we can eliminate **[ebx]** (red), since it is always 0. We can also eliminate **[ebx+8]** (white), since it is unique for each entity. Both the values in pink and yellow appear unique for each object. This leaves us with **[ebx+4]** (blue), which alternates between `0xD`, `0x40`, `0x82`, and `0x83`. For now, we will guess that this is our rendering value and investigate each value.

5.2.8 Modifying Rendering Value

If we set the value of **[ebx+4]** for each entity, it will be overwritten the next time the draw entities function is called. It appears that the entity is being loaded into **ebx** from another location. Therefore, the easiest way for us to explore our assumed rendering value is by hooking the location `0x52D2FD` and setting **[ebx+4]** for every entity. We could create this code cave in x64dbg, but to make it easier for us to test multiple values, we will create our hook in a DLL.

We will use the same hooking structure discussed in Chapter 3.4. Our hook will be at `0x52D2FD`, since we know **ebx** will contain the correct value at that point. Our hook itself will be relatively simple: we will save the registers, set the value of **[ebx+4]**, restore the registers, and then execute the original **mov** instruction:

```
DWORD ret_address = 0x0052D303;

__declspec(naked) void codecave() {
  __asm {
    pushad
    mov dword ptr ds:[ebx+4], ???
    popad
    mov dword ptr ds:[0x102AE98], ebx

    jmp ret_address
  }
}
```

For our first value, let's start on the highest end and try `0x83`:

```
mov dword ptr ds:[ebx+4], 0x83
```

Once the DLL is injected and you are back in the game, you will notice that nothing appears to change. Likewise, if you try `0x40`, you might notice that some shadows seem different, but everything looks pretty similar. Next, let's try `0xD`:

```
mov dword ptr ds:[ebx+4], 0xD
```

Immediately, you should notice that your character's model now appears see-through in front of the camera:



This is a good sign that depth testing may have been disabled. Next, switch to third-person mode (*cg_thirdperson*) and add some bots. As you move around, you should notice that you can now see all bots through walls:



With this, we have successfully set the rendering value for all entities to disable depth testing. We can see that other entities, such as guns and stairs, appear through walls as well.

One improvement is to re-enable depth testing for our player model so that first-person mode is not corrupted. To do this, you will need to identify the player structure and your current player.

The full source code for this hack is available in Appendix A for comparison.

Playrix Ex. 1024, page 195

5.3 Wallhack (OpenGL)

5.3.1 Target

Our target for this chapter will be Urban Terror 4.3.4.

5.3.2 Overview

Most games make use of external graphics libraries for rendering. The two most popular graphics libraries are DirectX and OpenGL. Both of these libraries are loaded by games dynamically. Once they are loaded, games then invoke functions in these libraries. For example, with OpenGL, games can make use of the **glDrawElements** function to draw a series of elements from data stored in an array. Since these libraries are external, the game's developers do not need to implement the rendering logic themselves.

5.3.3 Identify

Our goal in this chapter is to create a wallhack by hooking the game's graphics library and modifying its logic to display entities through walls.

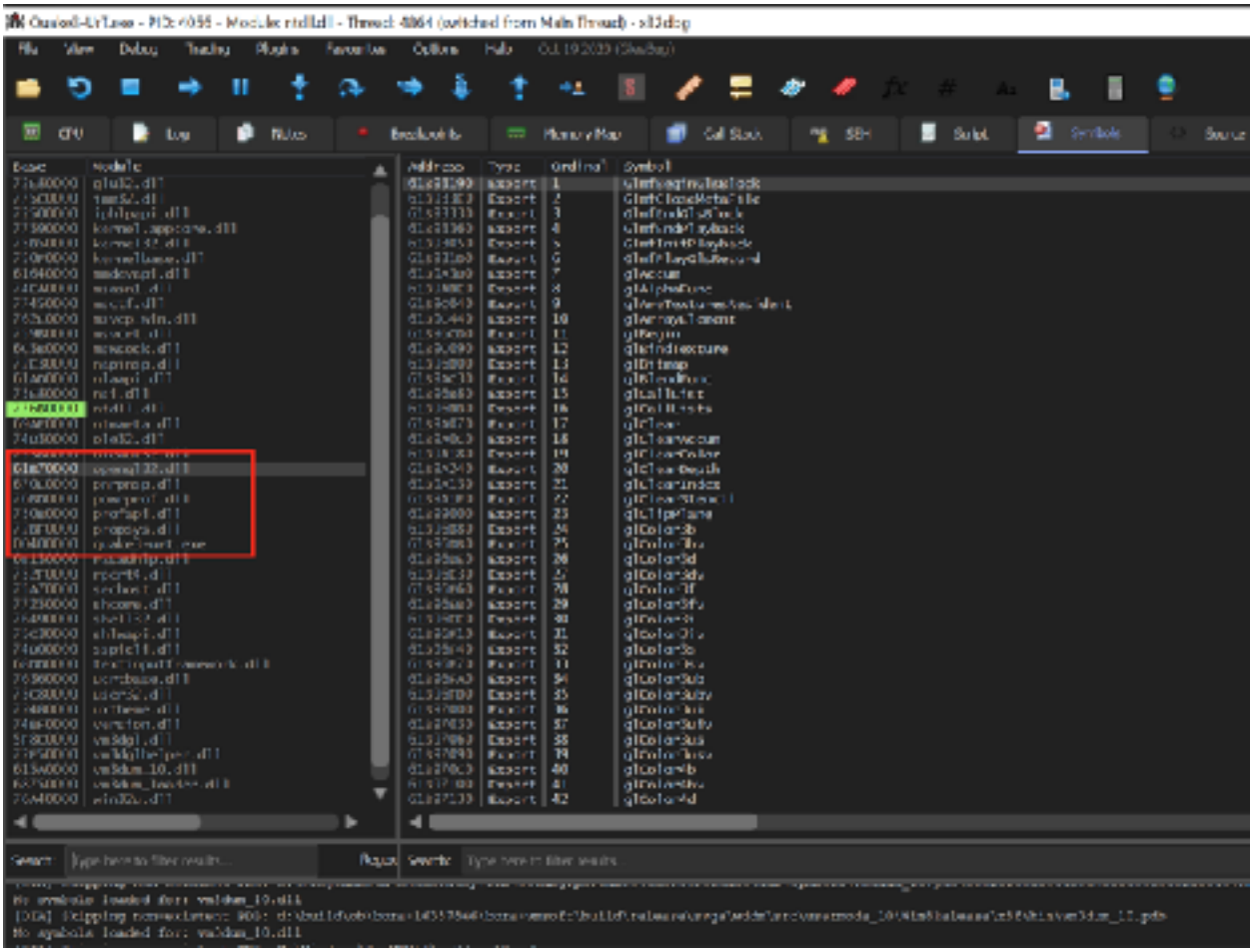
5.3.4 Understand

DirectX and OpenGL each have different functions for rendering that require different approaches to hook. Our first goal is to identify the library that the game is using. As each library has several functions to handle rendering and shading, we will then need to find a function that is used by the game for rendering. With the function identified, we can then hook it and disable depth testing through the use of a code cave. This will cause all entities to be rendered regardless of where they are in the 3D world.

Playrix Ex. 1024, page 196

5.3.5 Locating Drawing Library

Since graphics libraries are loaded dynamically, they must expose their functions to the main executable. Most debuggers allow you to view all the libraries loaded into an executable when attached. In x64dbg, this information is collected under the *Symbols* tab.



As we can see in the highlighted elements, *opengl32.dll* is being loaded into the game's process. By selecting the OpenGL module, we can see that it exports many drawing-related functions. From this information, we can conclude that this game is using OpenGL to render its graphics.

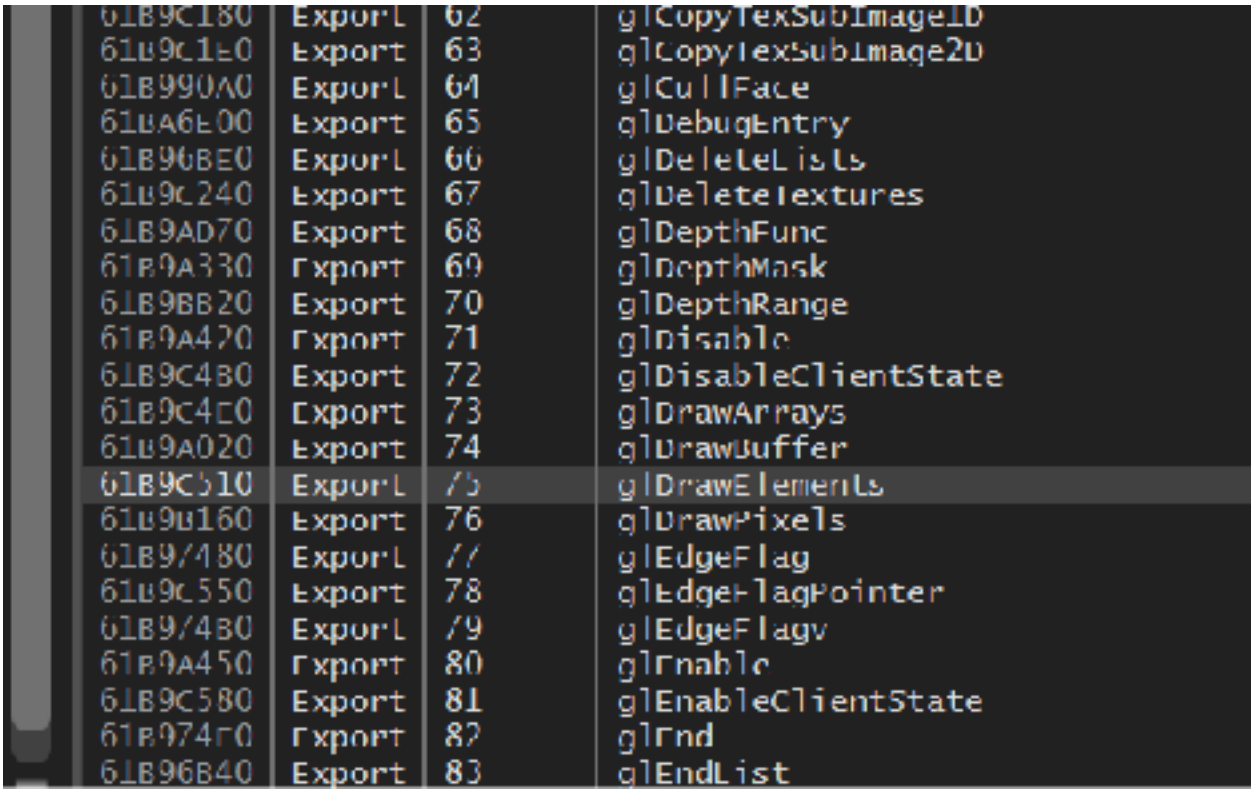
Playrix Ex. 1024, page 197

5.3.6 Locating the Drawing Function

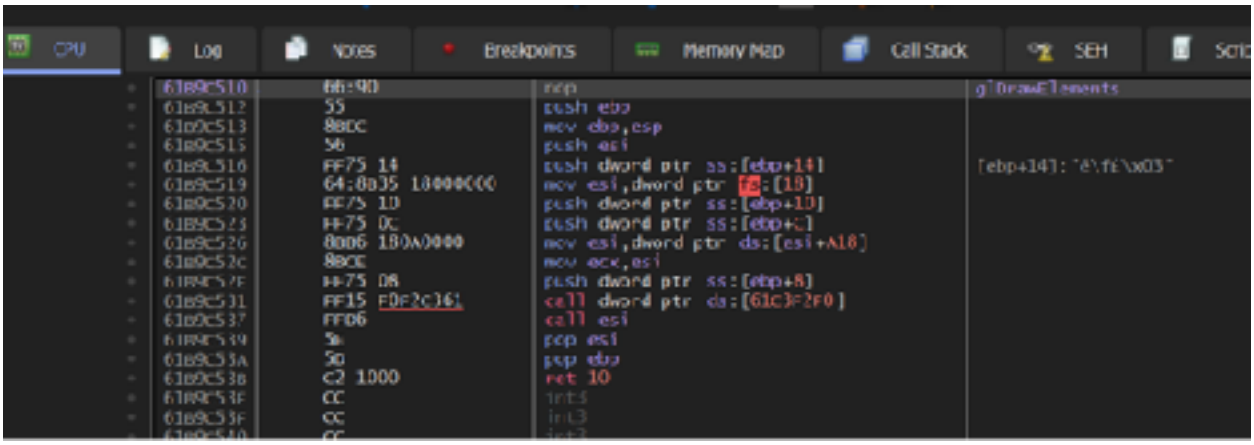
OpenGL has several rendering approaches, and different games will use different approaches. For example, older games may use **glBegin**, **glVertex**, and **glEnd**; some games may use **glDrawArrays**; and others may use **glDrawElements**. Some even use a combination of these approaches to render different aspects, such as **glDrawElements** for player models and **glBegin** for screen effects like blood.

Typically, modern games will not use **glBegin**, **glVertex**, and **glEnd**, as these functions are considered deprecated. For that reason, we won't be focusing on those functions right now. Instead, we will first investigate **glDrawElements**, as this is a commonly used function. Due to how OpenGL works, we will expect this function to be called constantly if it is used by the game.

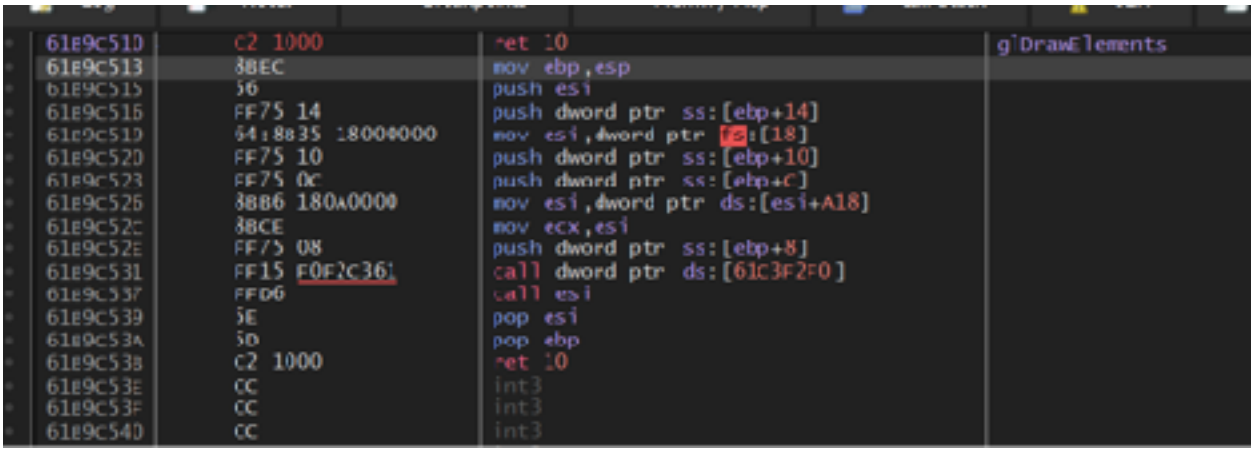
By scrolling down to the **glDrawElements** export in the *Symbols* tab, we can see that OpenGL exports it to the process, though this is not a guarantee that it is being used:



By double-clicking on the export entry, x64dbg will display the function:



Next, we can start a game and set a breakpoint on **glDrawElements**. You will notice that it will immediately pop, and pop continuously every time the game is resumed. This is a good indication that this function is responsible for rendering entities. To verify that this is the case, we can replace the first instruction with the **ret** statement we see at the end of the function. The effect of this will be to immediately return to the calling code without executing any of the **glDrawElements** logic:



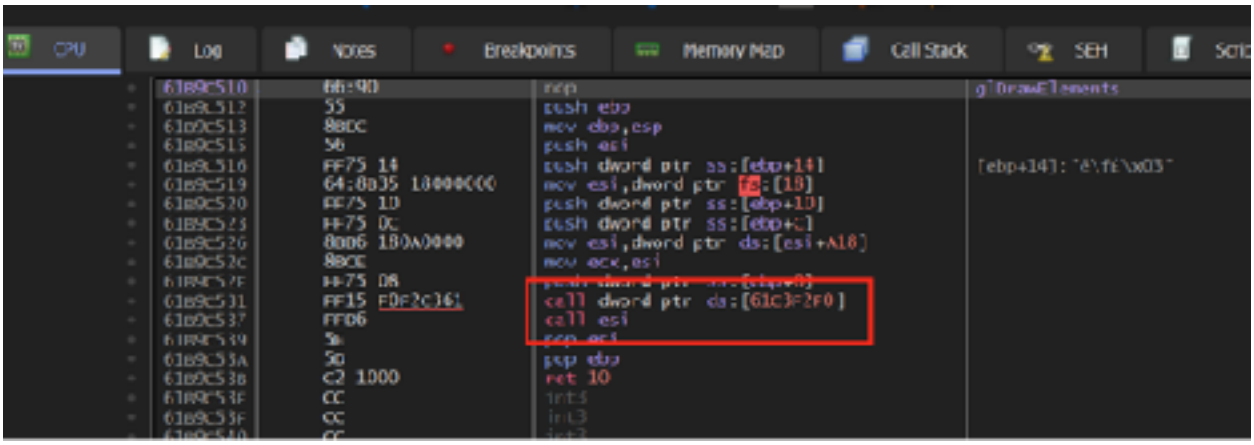
If you then resume execution and attempt to play the game, you will notice that no new entities are being rendered to the screen:



This gives us strong proof that Urban Terror is using **glDrawElements** to display entities.

5.3.7 Hooking glDrawElements

Examining the **glDrawElements** function, we can see that it has very few instructions. Given the complexity of rendering entities to a screen, the majority of the code must be contained in the two calls near the end of the function:



Therefore, if we hook an instruction before these calls, we should be able to accomplish our goal of disabling depth testing. A good candidate instruction is the **mov** at `0x61B9C526`.

Since OpenGL is loaded dynamically, our hooking approach will have to be slightly different. First, we will need to ensure that OpenGL is actually loaded. As we are injecting our DLL into the application when it is first started, this will not be the case. After we ensure that OpenGL is loaded, we need to figure out where it is loaded. Once we determine the base address of OpenGL, we can then determine where **glDrawEntities** is located inside the OpenGL module.

We will use a combination of techniques that we explored in previous chapters. To address the issue that OpenGL will not be loaded when our DLL is injected, we will create a thread to handle the hooking logic. This will allow us to create an infinite loop that waits until OpenGL is loaded, similar to the thread we saw in Chapter 3.3:

```
if (fdwReason == DLL_PROCESS_ATTACH) {
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread, NULL, 0,
NULL);
}
```

In our thread, we will create an infinite loop that will call GetModuleHandle. This API returns the module handle, or base address, for a loaded module. If the module is not loaded, it will return **NULL**:

```
HMODULE openGLHandle = NULL;

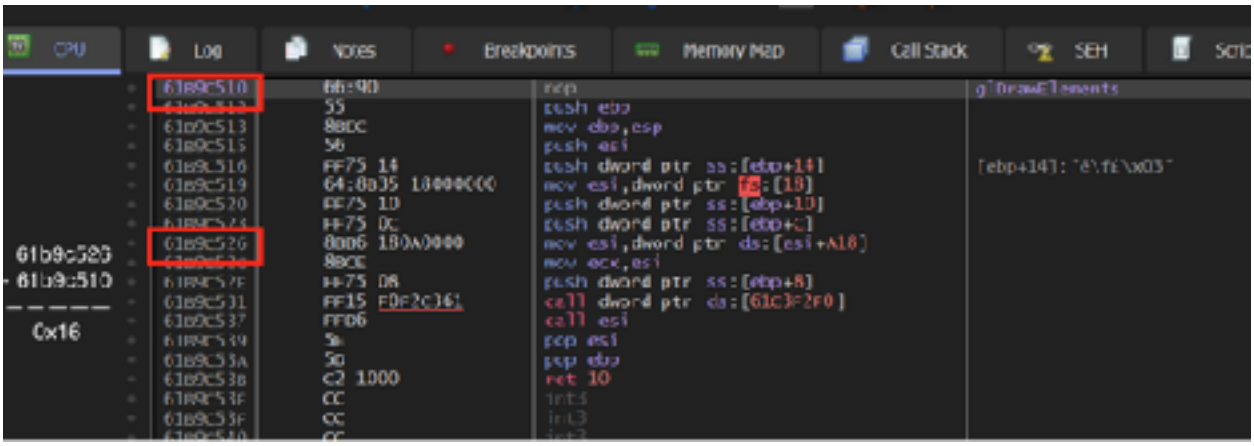
void injected_thread() {
    while (true) {
```

```
        if (openGLHandle == NULL) {
            openGLHandle = GetModuleHandle(L"opengl32.dll");
        }
    ...
    Sleep(1);
```

When we have the base address of OpenGL, we can then find where **glDrawElements** is located. To do this, we will make use of the [GetProcAddress] API. When given a module and the name of a function, this API returns the address of the function:

```
unsigned char* hook_location;
...
if (openGLHandle != NULL) {
    hook_location = (unsigned char*)GetProcAddress(openGLHandle,
“glDrawElements");
```

This API will return the location of the first instruction in the function, in this case **nop**. Since we want to hook the **mov** instruction, we can subtract its distance from the first instruction and then add that difference to the result of **GetProcAddress**. The distance between these two instructions will always be the same, as they are part of the library's code and not loaded dynamically.



This offset can be added directly to the **hook_location** variable to get our location:

```
hook_location += 0x16;
```

Finally, we can hook the code as we have done in previous chapters:

```
VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
*hook_location = 0xE9;
*(DWORD*)(hook_location + 1) = (DWORD)&codecave - ((DWORD)hook_location + 5);
*(hook_location + 5) = 0x90;
```

5.3.8 Function Pointers

With **glDrawElements** hooked, we can start working on the code cave. Our goal is to disable depth testing when an element is being drawn. To do this, we can use an OpenGL function called glDepthFunc. **glDepthFunc** allows you to set the function used for depth comparisons when OpenGL attempts to render the screen. This can be several values, but the ones we are interested in are **GL_LEQUAL** (draw if the element is in front of another element) and **GL_ALWAYS** (always draw).

For our wallhack, we will set the depth function to **GL_ALWAYS** right before any element is drawn. This will have the effect of making all elements always appear, regardless of where they actually are in the 3D space.

To start, we will need to locate **glDepthFunc**. We can use **GetProcAddress** in a similar manner to **glDrawElements**. However, instead of finding an address to hook, our goal with this call to **GetProcAddress** is to store the function's address in a way that we can then invoke in our code cave. The easiest way to do this is through a function pointer.

Just like pointers we have used in previous chapters, function pointers point to an address. However, unlike the pointers we have been using to modify data and code, we can also declare a pointer to point to a function. We can then call this function, or address, like we would call any other C++ function.

To declare a function pointer, we need to know the original function's definition. The definition of a function includes its return type and its parameters. We can get this information from the Khronos Group site:

```
void glDepthFunc(GLenum func);
```

Looking at the gl.h header file, we can find out what **GLenum** is:

```
typedef unsigned int GLenum;
```

So far, we can define our **glDepthFunc** function like so:

```
void glDepthFunc(unsigned int) = NULL;
```

Next, we will modify this declaration to have it act as a pointer to this function:

```
void (*glDepthFunc)(unsigned int) = NULL;
```

We can now assign this to the result of **GetProcAddress**:

```
glDepthFunc = GetProcAddress(openGLHandle, “glDepthFunc");
```

However, if we try to build this, we will get the following error:

```
error C2440: '=': cannot convert from 'FARPROC' to 'void (__cdecl *)(unsigned
int)'
message : This conversion requires a reinterpret_cast, a C-style cast or
function-style cast
```

Like we have seen in previous chapters, we need to cast the result of **GetProcAddress** properly for the compiler to understand how to translate the result. We can use the error message to quickly figure out how we need to cast the result:

```
glDepthFunc = (void(__cdecl *)(unsigned int))(openGLHandle, “glDepthFunc");
```

5.3.9 glDrawElements Code Cave

Our code cave will be similar to code caves we have written previously. We will start with our skeleton and restore the original code:

```
DWORD ret_address = 0;

__declspec(naked) void codecave() {
    __asm {
        pushad
    }
    ...
    __asm {
```

```
        popad
        mov esi, dword ptr ds : [esi + 0xA18]
        jmp ret_address
    }
```

Unlike previous chapters, we do not have a static location to jump to. Instead, we will need to calculate our return location similarly to how to we calculated the hook location. In our thread, after we assign the hook location, we can also dynamically assign the return location:

```
ret_address = (DWORD)(hook_location + 0x6);
```

With our skeleton in place, we can now add in our call to **glDepthFunc**. First, we need to find the value for **GL_ALWAYS**. We can find this in the gl.h header file:

```
#define GL_ALWAYS 0x0207
```

Next, we can invoke **glDepthFunc** to disable depth testing. Since it is a function pointer, we need to dereference the pointer to invoke the function:

```
(*glDepthFunc)(0x207);
```

Our code looks like:

```
#include <Windows.h>

HMODULE openGLHandle = NULL;

void (*glDepthFunc)(unsigned int) = NULL;

unsigned char* hook_location;

DWORD ret_address = 0;
DWORD old_protect;

__declspec(naked) void codecave() {
    __asm {
        pushad
    }
```


Playrix Ex. 1024, page 205

```
    (*glDepthFunc)(0x207);

    __asm {
        popad
        mov esi, dword ptr ds:[esi+0xA18]
        jmp ret_address
    }
}

void injected_thread() {
    while (true) {
        if (openGLHandle == NULL) {
            openGLHandle = GetModuleHandle(L"opengl32.dll");
        }

        if (openGLHandle != NULL) {
            glDepthFunc = (void(__cdecl *)(unsigned
int))GetProcAddress(openGLHandle, "glDepthFunc");

            hook_location = (unsigned char*)GetProcAddress(openGLHandle,
"glDrawElements");
            hook_location += 0x16;

            VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
            *hook_location = 0xE9;
            *(DWORD*)(hook_location + 1) = (DWORD)&codecave -
((DWORD)hook_location + 5);
            *(hook_location + 5) = 0x90;

            ret_address = (DWORD)(hook_location + 0x6);
        }

        Sleep(1);
    }
}

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    if (fdwReason == DLL_PROCESS_ATTACH) {
        CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread, NULL,
0, NULL);
    }

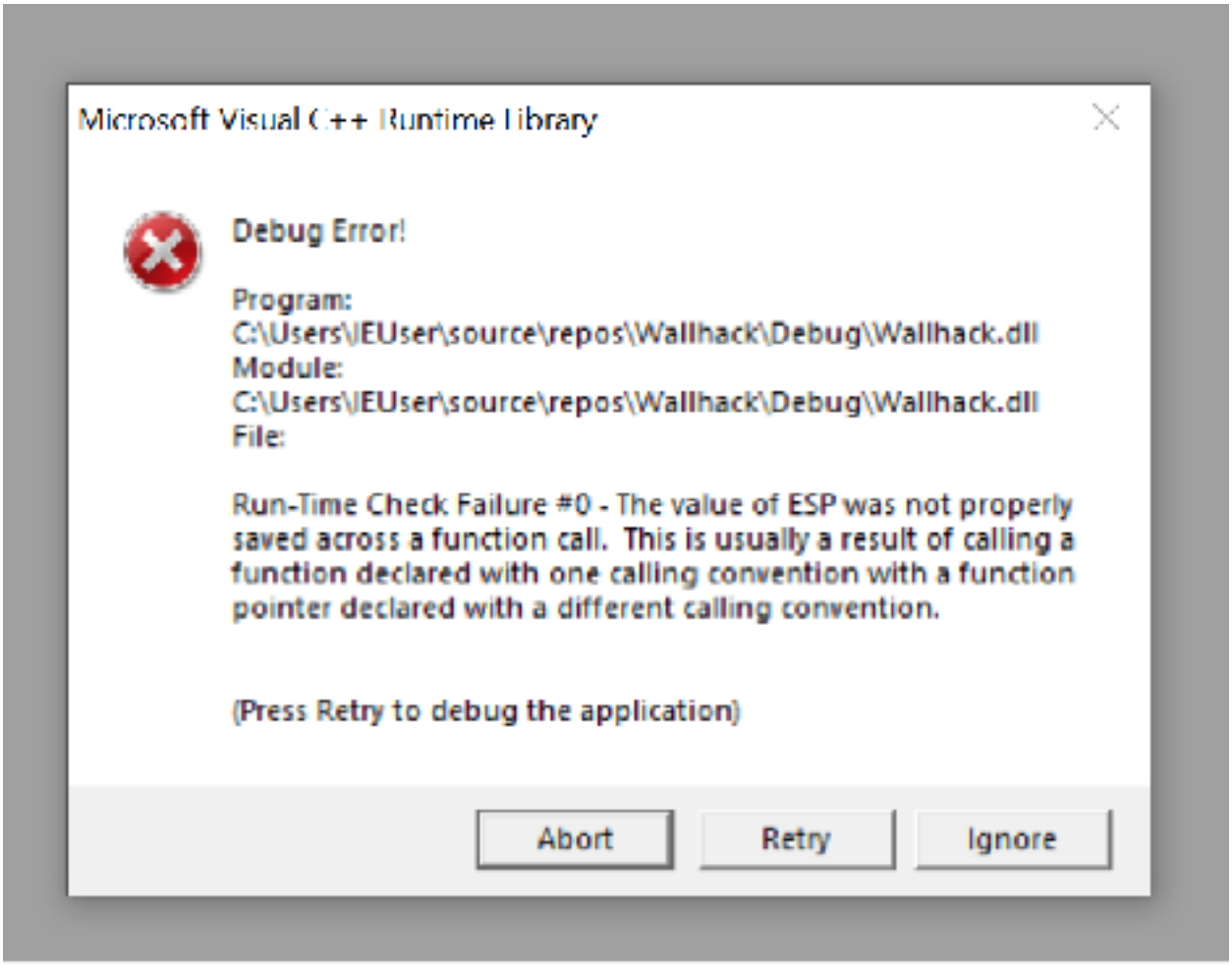
    return true;
```

```
}
```

We can now build this code and inject it into Urban Terror to see if it works.

5.3.10 Calling Conventions

When our DLL is injected, you will notice that the game will crash instantly when starting with the following error:



If we remove the call to **glDepthFunc** in our code cave, the game no longer crashes. It looks like our function pointer is not correct in some way. If we look at gl.h, we see that **glDepthFunc** is defined as:

```
GLAPI void APIENTRY glDepthFunc (GLenum func);
```

In Microsoft's documentation on data types, we see that **APIENTRY** is a reference for **WINAPI**. If we look at the entry for **WINAPI**, we see that it is a reference for **__stdcall**. Let's try adding this prefix to our function pointer:

```
void (__stdcall *glDepthFunc)(unsigned int) = NULL;
```

Building this code results in a familiar error:

```
error C2440: '=': cannot convert from 'void (__cdecl *)(unsigned int)' to
'void (__stdcall *)(unsigned int)'
```

This can be fixed by changing the cast as we did before:

```
glDepthFunc = (void(__stdcall*)(unsigned int))GetProcAddress(openGLHandle,
“glDepthFunc");
```

Calling conventions control how parameters are handled by functions when called. There are many different types, but for our purposes, we just need to know that Visual Studio uses **__cdecl** by default, whereas OpenGL defaults to **__stdcall**.

With this change, build the code and inject it again. You will notice that Urban Terror no longer crashes.

5.3.11 Checking Counts

If you join a game, you will notice that you are now able to see entities through walls. The only problem is that you can see too many things:

Playrix Ex. 1024, page 208



In our current hack, we are disabling depth testing for every element drawn on the screen, including walls and stairs. Ideally, we only want to draw players through walls. To accomplish this, we will have to filter out elements that we do not care about.

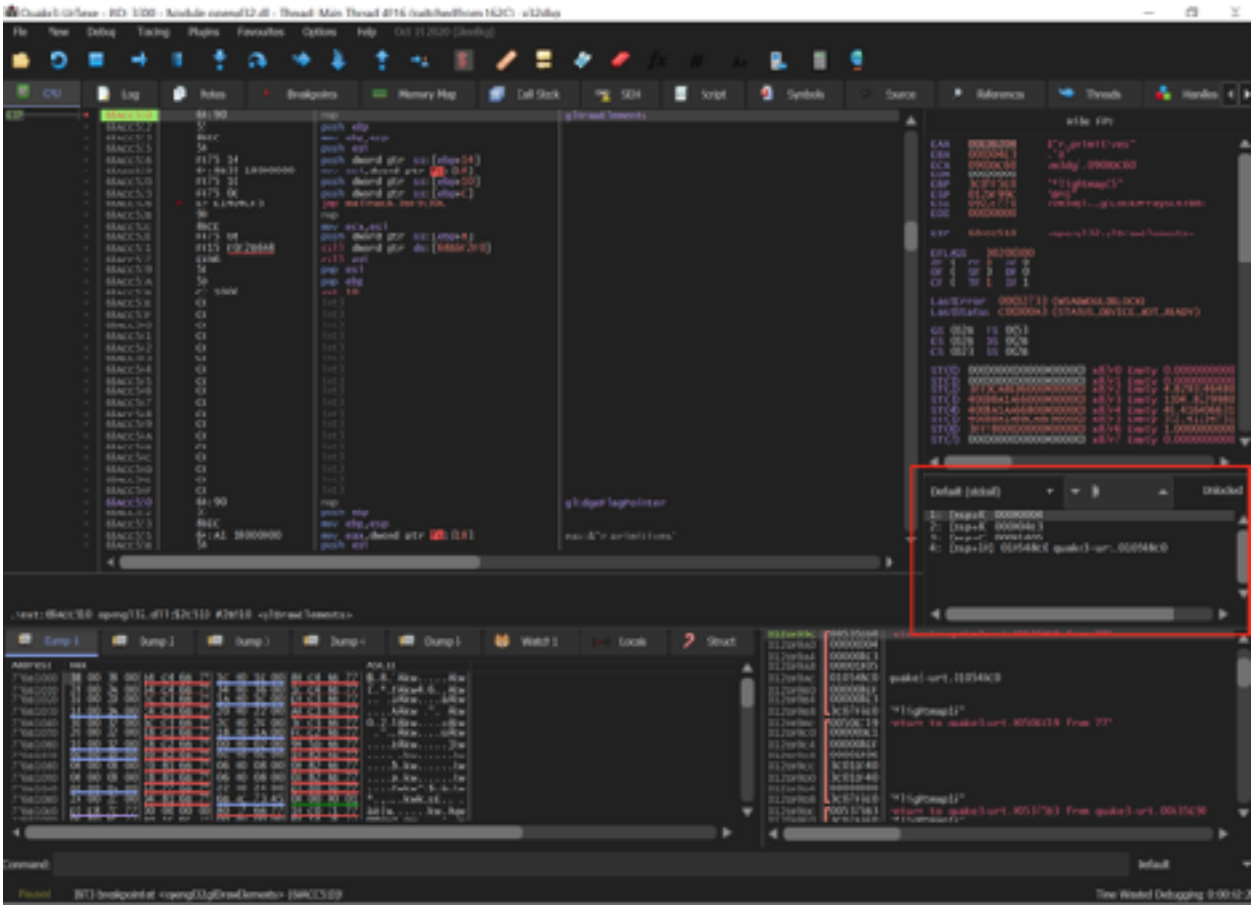
glDrawElements has the following definition:

```
void glDrawElements(    GLenum mode,
                        GLsizei count,
                        GLenum type,
                        const void * indices);
```

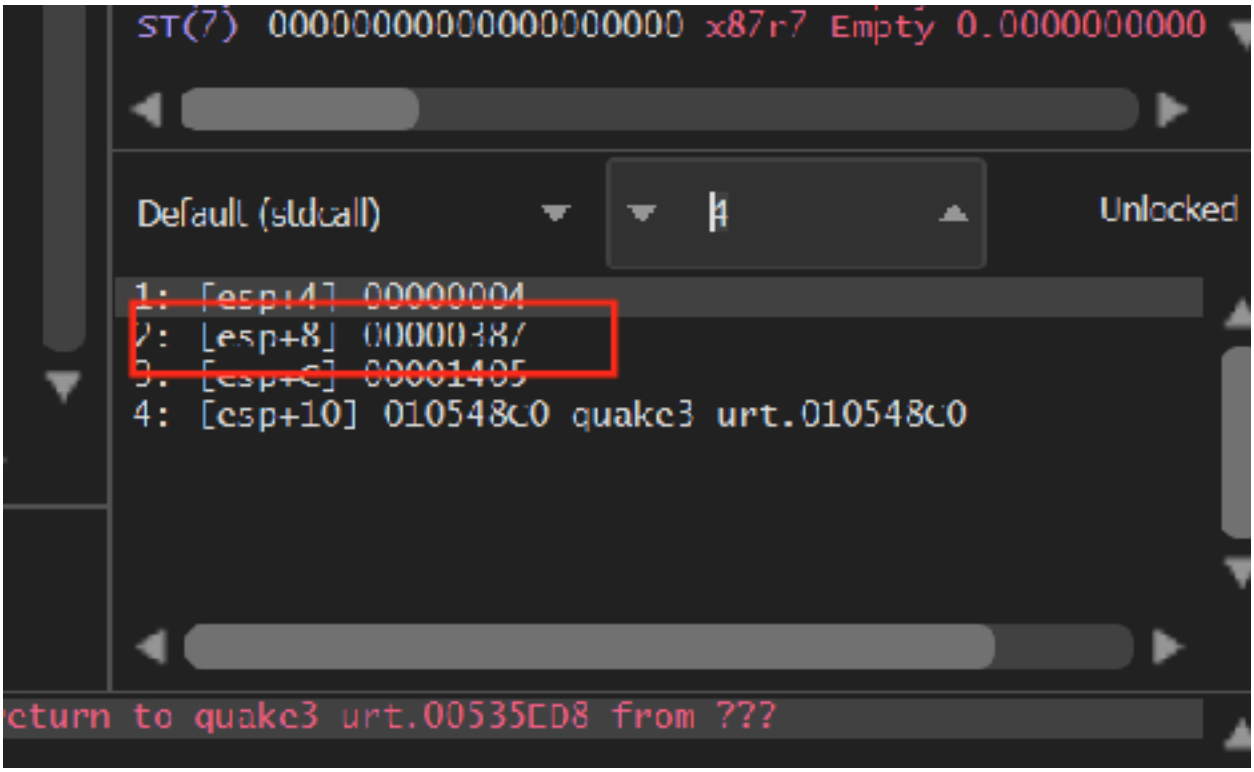
The **count** parameter specifies the amount of elements, or vertices, to be rendered. More detailed objects will have a higher amount of vertices. For example, a player model will have more detail (nose, hands, fingers, etc.) than a floor. By ensuring that the **count** parameter is a certain value, we can filter out elements that we do not want to display through walls.

We know that this parameter will be on the stack when our hook is jumped to. To retrieve its exact location, we can inject our DLL and set a breakpoint on **glDrawElements**. As we step through the code, we can identify where it is on the stack at the time our code cave gets called.

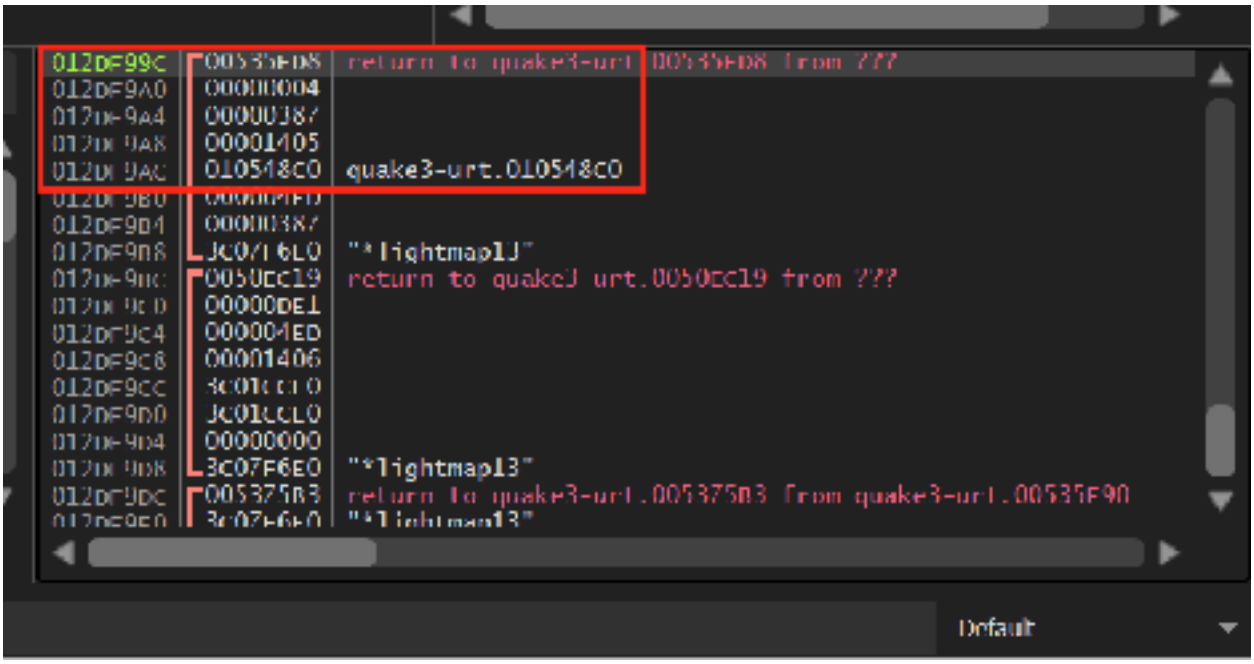
One feature of x64dbg is the ability to view the current parameters on the stack in a similar manner to how they would be passed in C. This feature is under the panel showing the values of the registers:



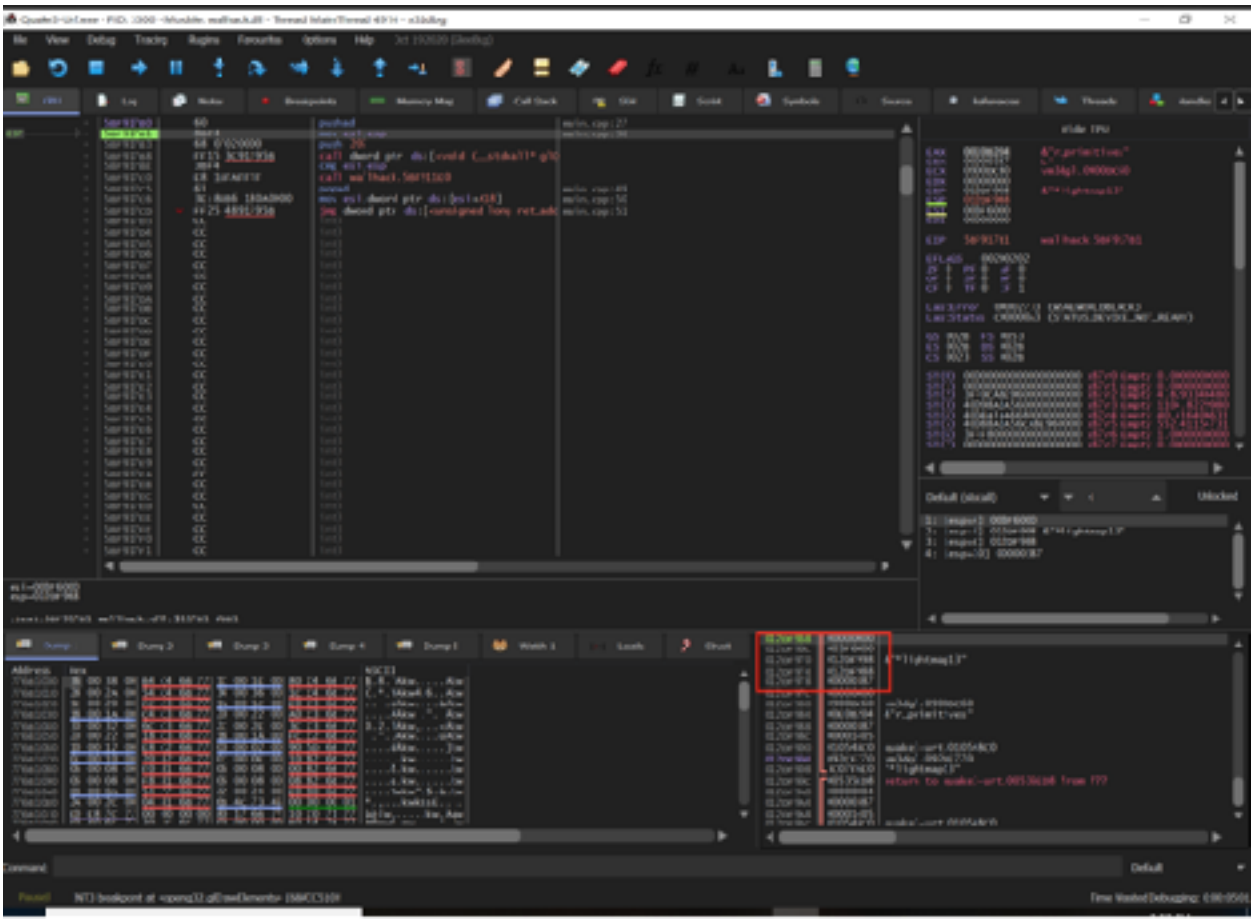
x64dbg is not able to fill this in automatically, so you will need to set the calling conventions and the number of parameters. If we trigger our breakpoint on **glDrawElements** multiple times, we can see that **[esp+8]** is the only value that appears to change. We can assume that it holds the value for the count parameter at the start of the function:



If we look at the stack panel at the bottom right, we can see how this information is represented on the stack. By default, the top of the stack (**esp**) will always appear at the top of the window:



Continue stepping through the function and then step into the jump to our code cave. After the **pushad** instruction in our code cave, examine the stack again:



At this point, we can see that the **count** parameter is at **esp+0x10**. We can reference this value in our code cave to retrieve the current count value of the element being rendered. In the first **asm** block, after the **pushad** instruction, we can take the value of **esp+0x10** and store it in a local variable:

```
DWORD count = 0;
...
__asm {
    pushad
    mov eax, dword ptr ds : [esp + 0x10]
    mov count, eax
    popad
    pushad
}
```

We now have a local variable **count** that will hold the value of count passed to **glDrawElements**. We can then compare this value to a baseline and only disable depth testing if we exceed that baseline. If we don't exceed it, we will re-enable depth testing. The value for **GL_LEQUAL** (`0x203`) can be found in the same way that we found the value for **GL_ALWAYS**. For now, we will use 500 as a baseline value:

```
if (count > 500) {
    (*glDepthFunc)(0x207);
}
else {
    (*glDepthFunc)(0x203);
}
```

If we build and inject this, we can see that our view is much cleaner now, and only certain elements appear through walls:



5.3.12 Clipping Planes

Now we are filtering many elements, but we've encountered the problem that no player models are appearing. Instead, we can only see their weapons and blood effects through walls:



If we enable third-person view, our player model is also invisible. The only place our player model will appear is if we turn on no-clip and fly out-of-bounds. This is most likely due to our player model being drawn first, when the scene is being rendered, and then other elements of the level drawn on top of it. When we disable depth testing, these entities are all drawn on top of the player.

```
draw_player();
draw_guns();
draw_doors();
draw_level_walls();
```

To force players to be drawn above these elements, we can use the glDepthRange function. This function sets the near and far clipping planes for the scene. Clipping planes are planes that extend across the game scene and clip (or remove) any entities behind them. By setting these values to be equal to 0, the planes will intersect, causing all elements to be drawn on the same plane and "fight" for rendering space. This will result in some flickering, but the player models will appear through walls.

We can create a function pointer for this function identically to the approach we used for **glDepthFunc**. The only alterations we need to make are in the parameters:

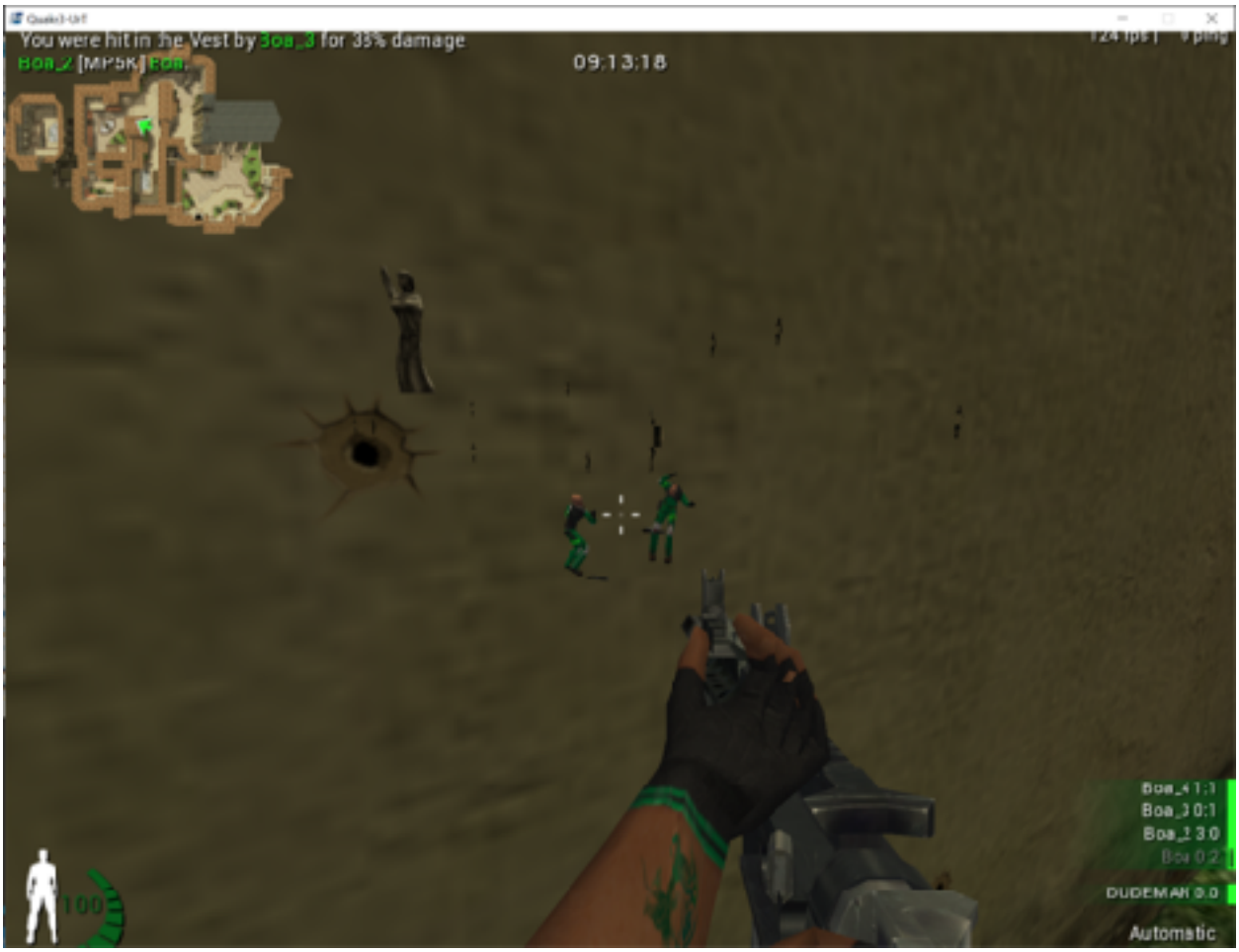
```
void (__stdcall* glDepthRange)(double, double) = NULL;
...
glDepthRange = (void(__stdcall*)(double, double))GetProcAddress(openGLHandle,
“glDepthRange");
```

We can then call this function in the same location that we change the depth function. The default values for these planes are (0,1), which we will reset if the count is too low.

```
if (count > 500) {
    (*glDepthRange)(0.0, 0.0);
    (*glDepthFunc)(0x207);
}
else {
    (*glDepthRange)(0.0, 1.0);
    (*glDepthFunc)(0x203);
}
```

With this, player models will now appear through walls, indicating that our wallhack is successful:

Playrix Ex. 1024, page 215



The full code for this chapter is available in Appendix A.

Playrix Ex. 1024, page 216

5.4 Chams (OpenGL)

5.4.1 Target

Our target for this chapter will be Urban Terror 4.3.4.

5.4.2 Identify

Our goal in this chapter is to create a chams hack, which is a type of hack that colors all players a bright color. We can accomplish this by hooking the game's graphics library and modifying its code to make all player models render with a bright color instead of a texture.

5.4.3 Understand

When entities are rendered to the screen, they are just filled polygons. To make these entities have visuals (such as eyes, camouflage, or hair), textures have to be applied to the polygons. These textures are specially formatted images, which wrap around the entity when applied to it. For example, the oil barrel texture from Urban Terror looks like:



Playrix Ex. 1024, page 217

When this is applied to the circular barrel model, it wraps around it. This is how 2D textures are applied to 3D models.

To create a chams hack, we will modify this rendering flow. After the polygons have been rendered, we will disable textures in OpenGL. When textures are disabled, OpenGL will fall back to using the lighting (or color) array specified by the game. If we disable that as well, OpenGL will fall back to using whatever color was last specified by a call to **glColor**. If we set our own color and then render the entity, we can make the entity appear as a bright solid color, such as red.

5.4.4 Texture Function Pointers

To make our development easier, we will build off the OpenGL wallhack we created in the previous chapter. For that, we created function pointers for two functions related to depth testing: **glDepthFunc** and **glDepthRange**. To disable and enable textures, we will need to create function pointers to four additional functions:

- **glEnable**
- **glDisable**
- **glEnableClientState**
- **glDisableClientState**

We plan to use **glEnable** and **glDisable** to enable and disable **GL_COLOR_MATERIAL**. In addition, we will need to call **glEnableClientState** and **glDisableClientState** to enable and disable **GL_COLOR_ARRAY** and **GL_TEXTURE_COORD_ARRAY**. We will enable and disable all these elements to ensure that OpenGL falls back to a mode where we can set the color.

To set the color after we have done those steps, we will also need to create a function pointer to **glColor**. **glColor** has many forms that allow you to pass in different type of parameters. Any of these functions will work, but for our hack, we will use **glColor4f**, the version of **glColor** that takes 4 floats (values that allow decimals): one for red, green, blue, and alpha. The alpha float is responsible for controlling the opacity of the color.

We can declare these function pointers right below the pointers for **glDepthFunc** and **glDepthRange**:

```
void(__stdcall* glColor4f)(float, float, float, float) = NULL;
void(__stdcall* glEnable)(unsigned int) = NULL;
```


Playrix Ex. 1024, page 218

```
void(__stdcall* glDisable)(unsigned int) = NULL;
void(__stdcall* glEnableClientState)(unsigned int) = NULL;
void(__stdcall* glDisableClientState)(unsigned int) = NULL;
...
...
glColor4f = (void(__stdcall*)(float, float, float,
float))GetProcAddress(openGLHandle, "glColor4f");
glEnable = (void(__stdcall*)(unsigned int))GetProcAddress(openGLHandle,
"glEnable");
glDisable = (void(__stdcall*)(unsigned int))GetProcAddress(openGLHandle,
"glDisable");
glEnableClientState = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glEnableClientState");
glDisableClientState = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, “glDisableClientState");
```

5.4.5 glDrawElements Code Cave

In our code cave, we already have the logic built out to display models through walls if they have a count greater than 500. We will expand on this code to also color them. First, we will disable **GL_COLOR_ARRAY** and **GL_TEXTURE_COORD_ARRAY**. The game uses these client states to let OpenGL know that the game wants to map textures and color arrays (for lighting) to polygons. To apply a static color to a model, we need to tell OpenGL that we are not using these features. Since these are considered client states, we will use **glDisableClientState** to disable them. We can get their values from gl.h:

```
if (count > 500) {
    ...
    (*glDisableClientState)(0x8078);
    (*glDisableClientState)(0x8076);
```

Next, we will enable **GL_COLOR_MATERIAL** and set our color to red. **glColor4f** takes a value between 0 and 1 for all values. If we want a red color, we will set the red value to 1 and the alpha to 1. However, if we leave green and blue at 0, our ending red color will be dark and muted. To make it vibrant, we will set these values to 0.6. Adding an **f** on the end of a number in C++ will cause the number to be interpreted as a float:

```
(*glEnable)(0x0B57);
(*glColor4f)(1.0f, 0.6f, 0.6f, 1.0f);
```


Playrix Ex. 1024, page 219

Finally, just like with our wallhack, we will disable this coloring when the model's count is less than 500. To do this, we will enable **GL_COLOR_ARRAY** and **GL_TEXTURE_COORD_ARRAY** and then disable **GL_COLOR_MATERIAL**. Finally, we will make another call to **glColor**, this time setting the color to a pure white. This is not strictly necessary for Urban Terror, but for some games, this will prevent the colors of effects from getting corrupted if they use the previously set color:

```
(*glEnableClientState)(0x8078);
(*glEnableClientState)(0x8076);
(*glDisable)(0x0B57);
(*glColor4f)(1.0f, 1.0f, 1.0f, 1.0f);
```

With this done, you can inject the DLL into the game, and you will see all the models appearing through walls with a bright red color:



The full source code for this hack is available in Appendix A.

5.5 Triggerbot

5.5.1 Target

Our target for this chapter will be the game Assault Cube 1.2.0.2, since it has an easy way to create bots and disable their movement. However, this same technique will work on any FPS that displays a player's name when you hover over a player.

5.5.2 Identify

Our goal in this chapter is to create a triggerbot, a type of hack that automatically fires whenever we look at another player.

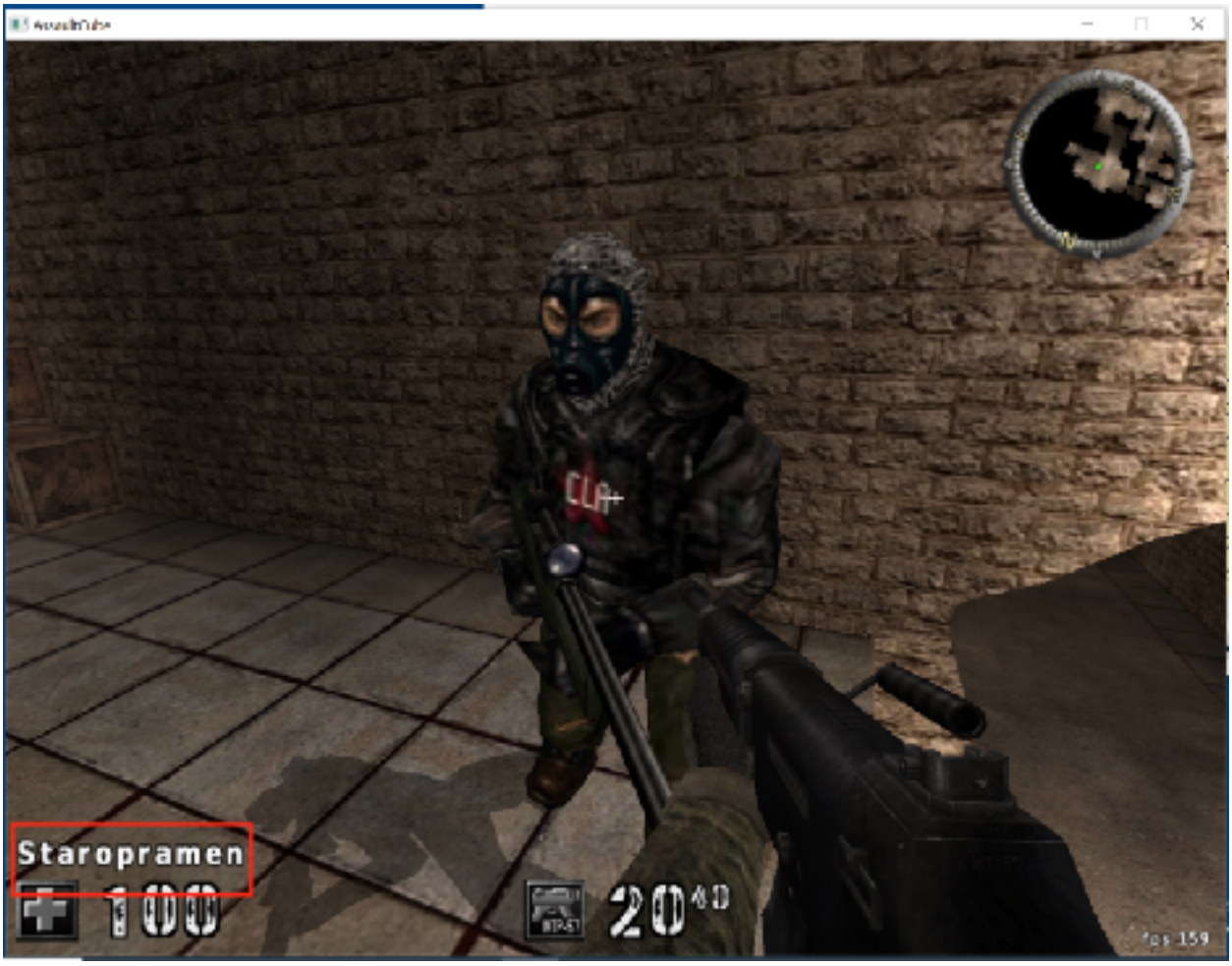
5.5.3 DLL Injection

While the following chapters can be done using the AppInit technique discussed in previous chapters, using a DLL injector will vastly speed up development time. From this point on, the rest of the book will assume that you are using an injector. Creating a DLL injector is discussed in Chapter 7.1. General-purpose DLL injectors can also be found online.

5.5.4 Understand

To write a triggerbot, we need to calculate where our player is looking and identify if we are looking at another player. Luckily for us, most games already have this functionality in their code to display a nametag when you hover over a player or change the crosshair to a different color. Assault Cube displays a nametag, as seen in the bottom left-hand corner:

Playrix Ex. 1024, page 221



If we locate the code responsible for displaying this text, we can hook it and write custom code to send a mouse press.

5.5.5 Locating Code

The method for locating the responsible code will depend on how the game reacts to hovering over a player. In general, games will react in two different ways:

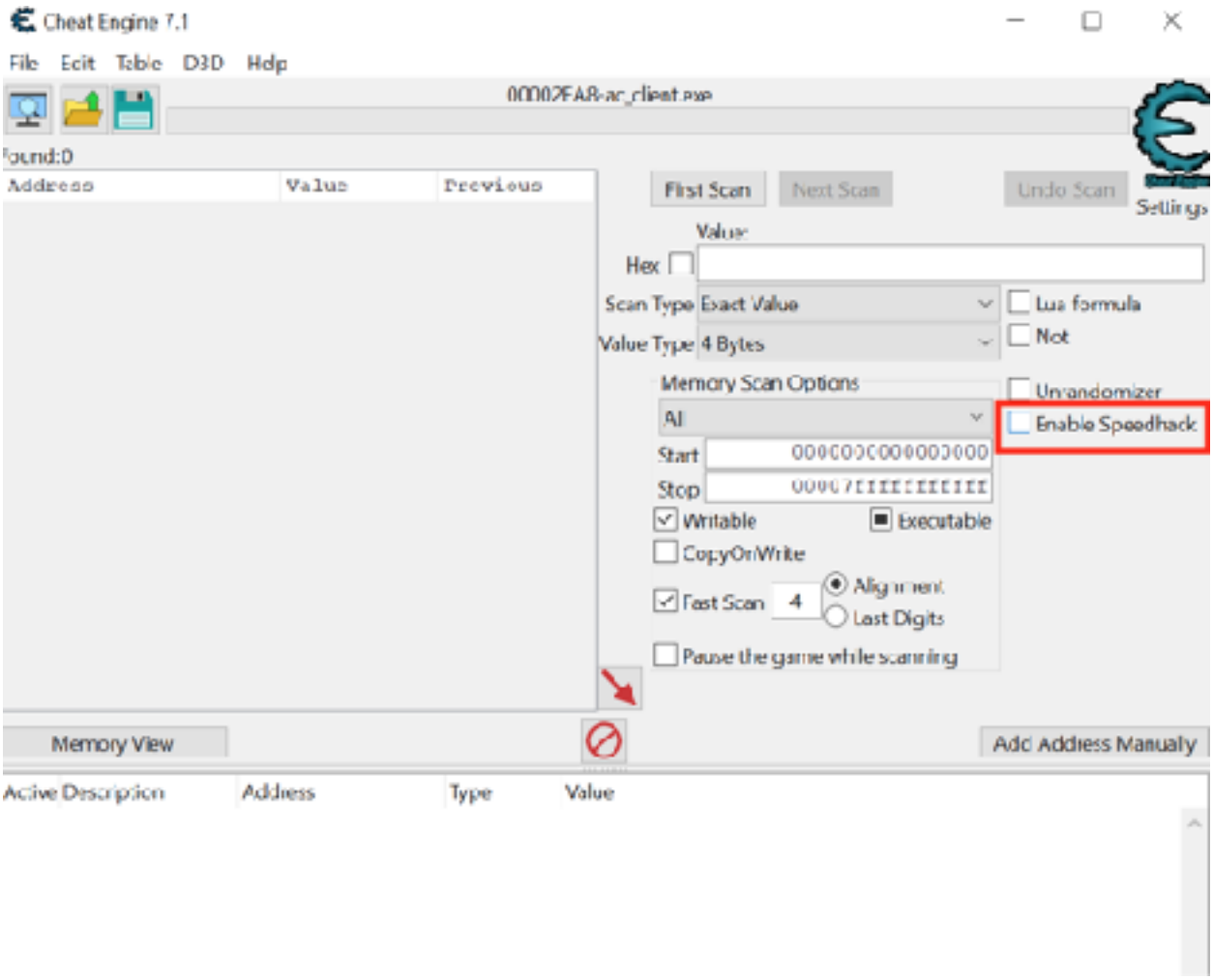
1. The crosshair will change, either in size or color.
2. The player's name we are looking at will be displayed somewhere on the screen.

In games with the first reaction, we will have to search for one value while not looking at a player and then filter for a different value while looking at a player. After enough filtering, a value will remain that will generally be 0 when not looking at a player, and 1

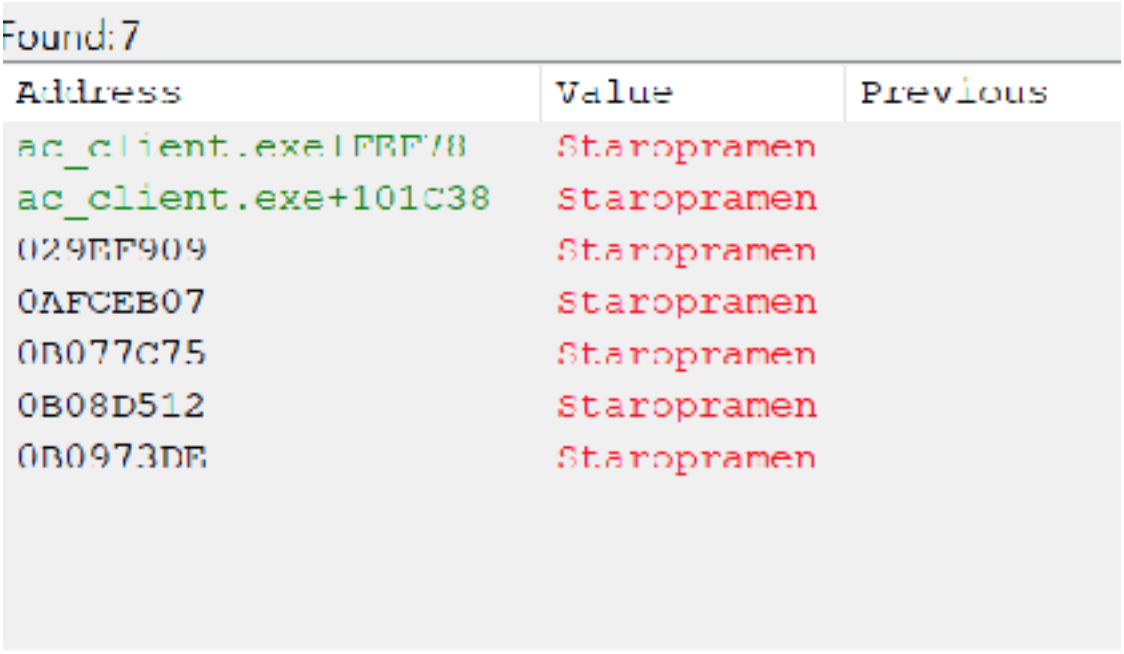
(or a value linked to the player's location in the entity list) when looking at a player. You can then set a breakpoint on this memory address and see what code writes to it.

Other games, like Assault Cube, will display a string that represents the player's name. In these cases, we can locate a player's name in memory and then set a breakpoint on access on the name. When we hover over the player, this breakpoint should pop at the code responsible for determining if we are looking at a player.

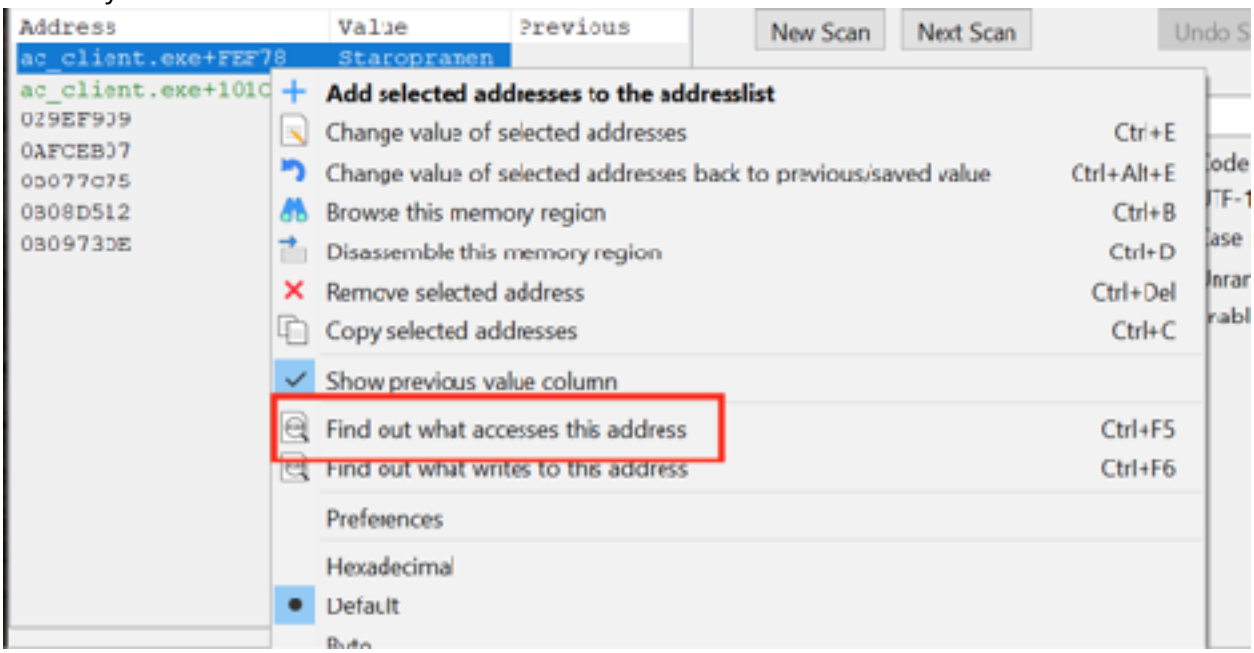
First, make sure the nametags option is enabled in the HUD settings. Then, start a new single-player deathmatch game with 8 bots. When the game starts, hit the "~" key to open the console and run the command *idlebots 1*. This command will disable bots from moving and shooting, making it easier to search for the information we want. In games that do not have a way to disable bot movement, you can use Cheat Engine's *Enable Speedhack* feature to slow down the game and allow you to search easier:



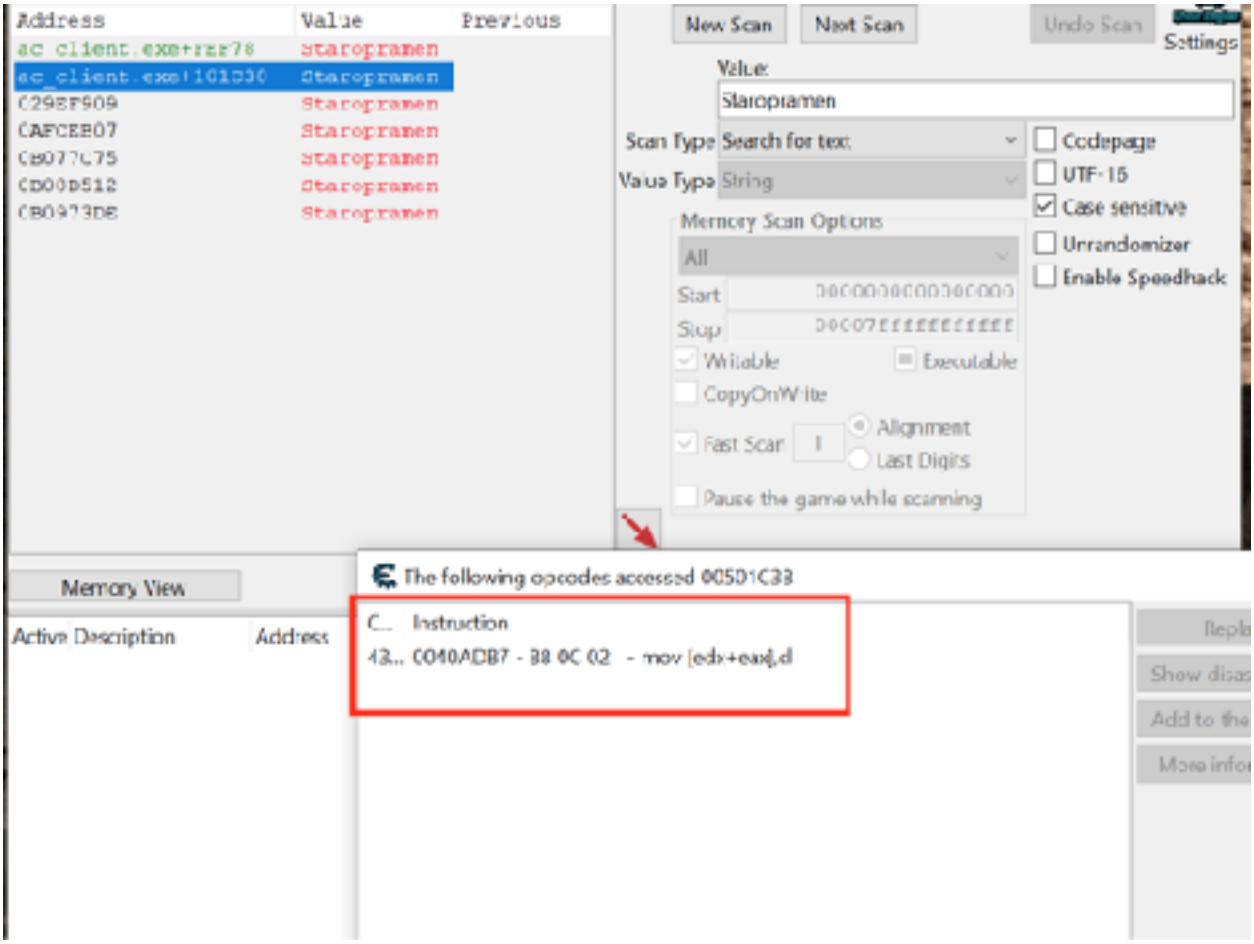
With the game started, find a particular bot and note its name down. Search in Cheat Engine for this name, which should return about 10 results:



Next, look away from the bot so that its nametag is no longer displayed. For each address identified, use Cheat Engine's *Find out what accesses this address* option, which will attach a debugger to the process to determine what code is touching the memory:



For each address, look at the bot again so that the nametag displays. We are looking for an address which has a ton of accesses only when looking at the bot. After going through several of the addresses, you should find one that is always accessed only when looking at a bot:

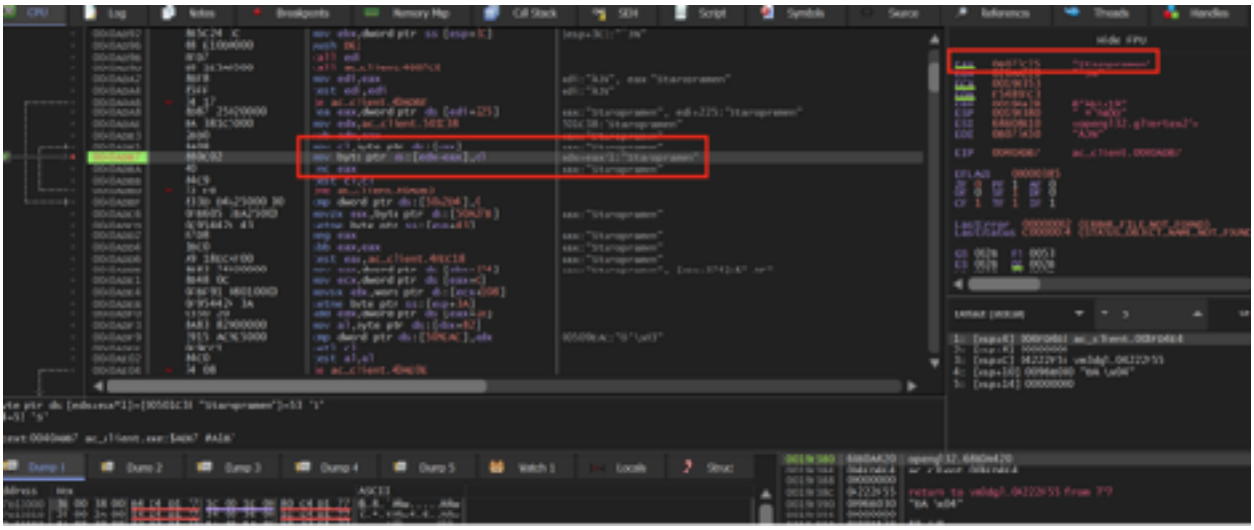


With this code found, we can close Cheat Engine and start working on reversing the responsible code.

5.5.6 Locating Code Cave

Open up x64dbg and attach it to Assault Cube. In Assault Cube, make sure that the nametag is still displaying. Navigate to the address we found in Cheat Engine and place a breakpoint on the code. It should pop immediately:

Playrix Ex. 1024, page 225



You will notice that we are inside a loop that is loading the player's name into a buffer. We see that this loop is only entered if the **je** at `0x0040ADA6` does not jump. The condition for this jump is the **test edi, edi** instruction. Testing a register against itself compares its value to 0. From this, we can assume that the nametag is only displayed if **edi** is not 0. We can confirm this behavior by setting a breakpoint on the **test edi, edi** instruction. When looking at a player, **edi** will have a value:



However, if we are not looking at a player, **edi** will hold the value 0:



Looking at the **call** above, we can see that **edi** gets its value from **eax**, which is the return value from the **call**:



Since this **call** is 5 bytes, we will overwrite this **call** with a code cave to our own code.

5.5.7 Writing Code Cave

The logic for our code cave will be simple. First, we will execute the call we hooked. Then, we will read the value of **eax** into a variable. After that, we will read the value of the variable. If we are looking at a player, we will use the SendInput API to send a left mouse down event to the game. Otherwise, we will send a left mouse up event to the game. We need to use this approach since **SendInput** sets a key's state permanently. If we do not send a left mouse up event, the mouse button will act as if it is held down.

Like we discussed above, we will hook the **call** at `0x0040AD9D`. We will do this in an identical manner to previous chapters:

```
unsigned char* hook_location = (unsigned char*)0x0040AD9D;
...
VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
*hook_location = 0xE9;
*(DWORD*)(hook_location + 1) = (DWORD)&codecave - ((DWORD)hook_location + 5);
```

In our code cave, we will first start by calling the method that we overwrote and then moving its return value (**eax**) into a variable that we declare:

```
DWORD ori_call_address = 0x4607C0;
DWORD edi_value = 0;

__declspec(naked) void codecave() {
      __asm {
            call ori_call_address
            pushad
            mov edi_value, eax
      }
```

Next, we will check the value of our **edi_value** variable to determine if we should send a left mouse down or mouse up event:

```
if (edi_value != 0) {
    //looking at player
}
else {
    //not looking at player
}
```

SendInput takes an array of input events, which allows you to send multiple events. This can be useful if we want to do multiple actions at once, such as firing and then reloading. In this chapter, we will only send one input, which is the mouse down or mouse up event:

```
INPUT input = { 0 };
...
if (edi_value != 0) {
    input.type = INPUT_MOUSE;
    input.mi.dwFlags = MOUSEEVENTF_LEFTDOWN;
    SendInput(1, &input, sizeof(INPUT));
```


Playrix Ex. 1024, page 228

```
}
else {
    input.type = INPUT_MOUSE;
    input.mi.dwFlags = MOUSEEVENTF_LEFTUP;
    SendInput(1, &input, sizeof(INPUT));
}
```

Just like in previous chapters, we want to restore the registers and jump back to the original code:

```
DWORD ori_jump_address = 0x0040ADA2;
...
_asm {
    popad
    jmp ori_jump_address
}
```

If you are using the DLL injector from Chapter 7.1, we can make some small modifications to use it for this target:

```
const char *dll_path = "C:\\Users\\IEUser\\source\\repos\\triggerbot\\Debug\
\triggerbot.dll";
...
if (strcmp((const char*)pe32.szExeFile, (const char*)L"ac_client.exe") == 0)
{
```

With those changes, we can inject the DLL into Assault Cube and hover over a player. When we pass over a player, we will automatically fire.

The full code for this chapter is available in Appendix A.

Playrix Ex. 1024, page 229

5.6 Aimbot

5.6.1 Target

Our target for this chapter will be Assault Cube 1.2.0.2.

5.6.2 Identify

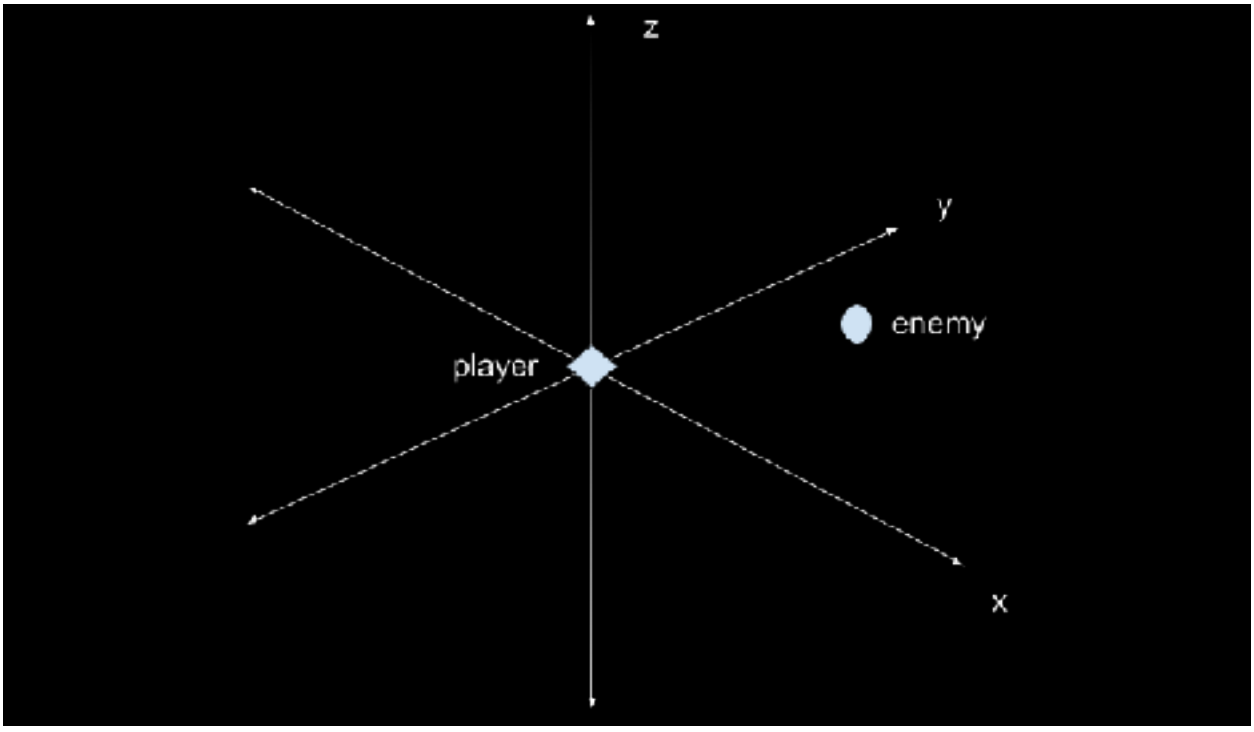
Our goal in this chapter is to create an aimbot, a type of hack that automatically aims at other players.

5.6.3 Understand

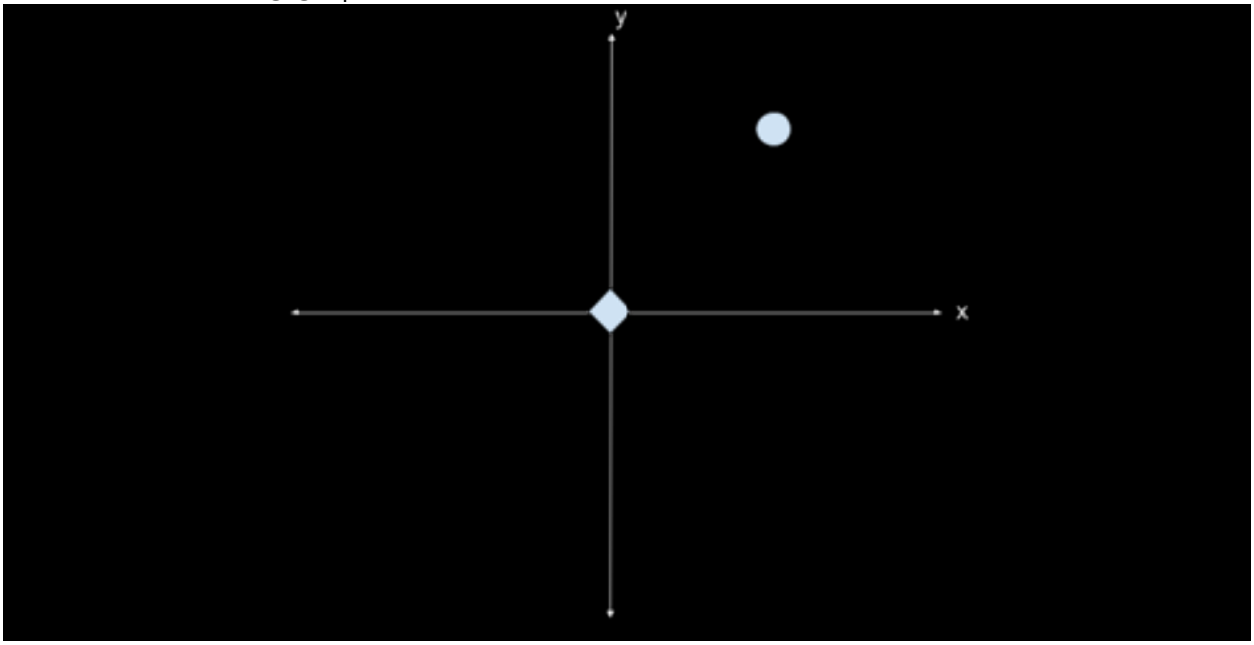
The core fundamentals of an aimbot rely on trigonometry. Take the following scene from our target game, Assault Cube:



Focusing on just our in-game player and the enemy, this scene can be mapped onto a 3D graph that looks like:

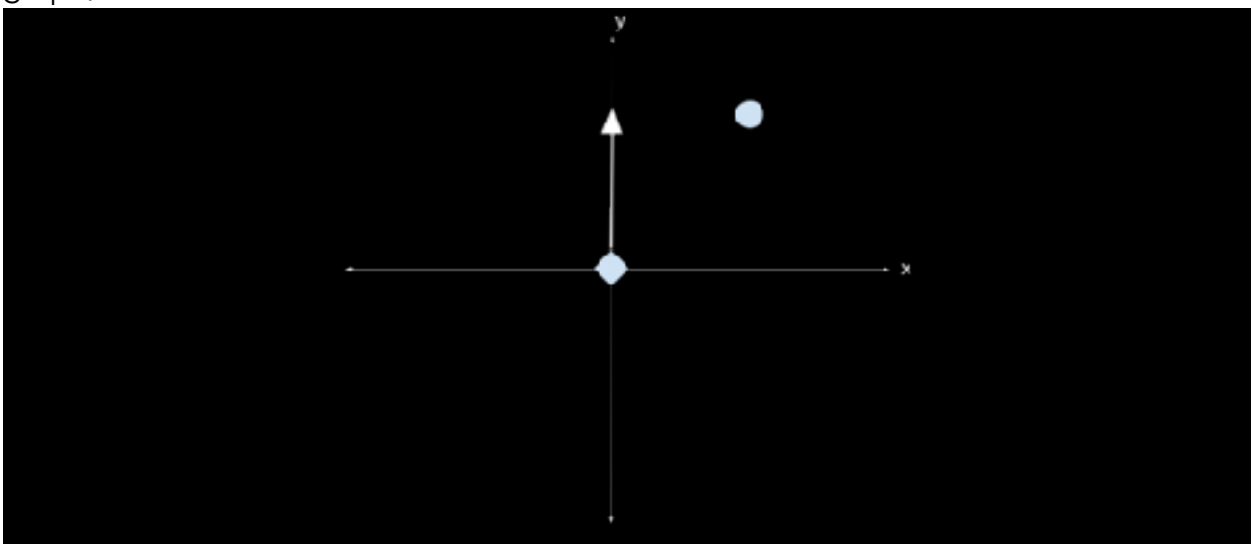


To simplify, we can convert this into a 2D graph by fixing our perspective and eliminating one of the axes. By choosing a top-down perspective, we can eliminate the Z axis. The resulting graph would look like:



Playrix Ex. 1024, page 231

Every first-person or third-person shooter allows the player to look left and right to aim. For example, in our first screenshot, our player is looking straight ahead. On our 2D graph, this would look like:

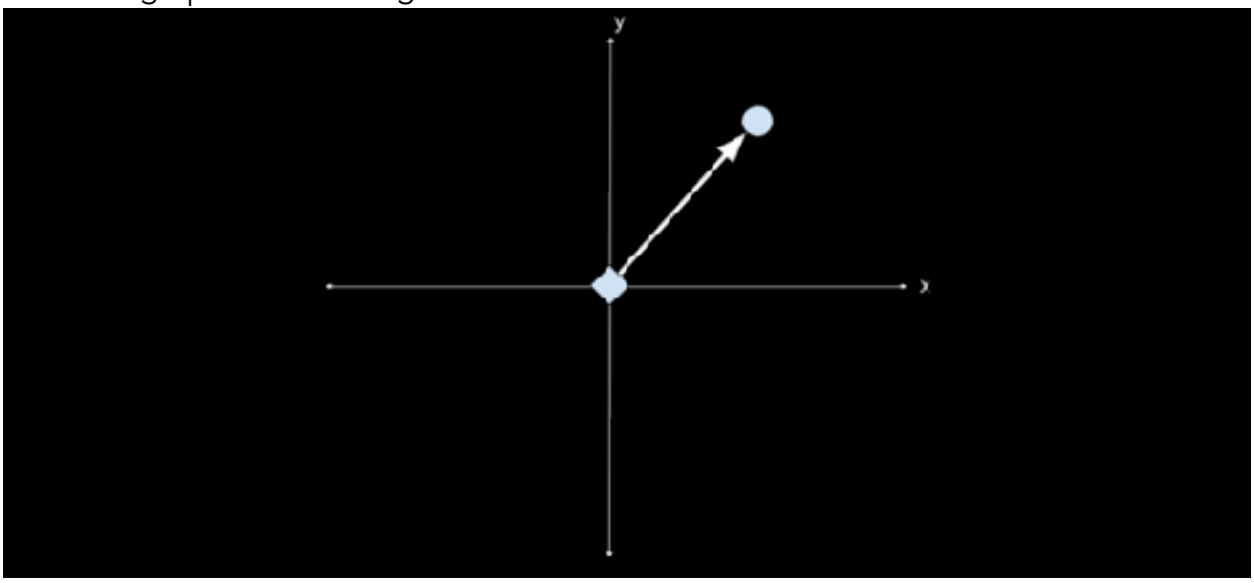


If we are looking at an enemy, like so:

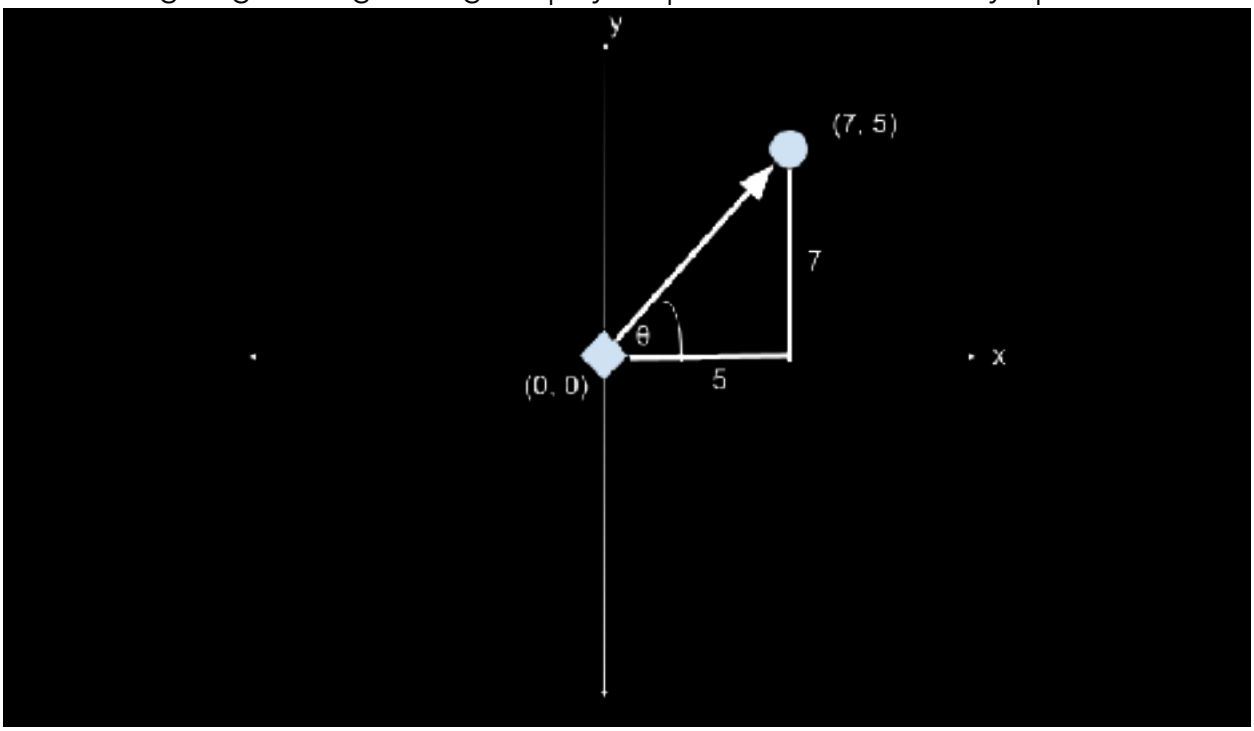


Playrix Ex. 1024, page 232

Then our graph would change to look like:



Games represent this left and right value as an angle. They can represent this angle in multiple ways, such as a vector, a radian, or a degree. However, for our current example, we will assume the view angle is represented in degrees. To create an aimbot, we need to find a way to calculate this angle for an enemy. We can do this by first creating a right triangle using our player's position and the enemy's position:



If we knew the value of θ, we could use the tangent operation to determine the ratio between the opposite (7 above) and adjacent (5 above) sides. In our case, we have the

Playrix Ex. 1024, page 233

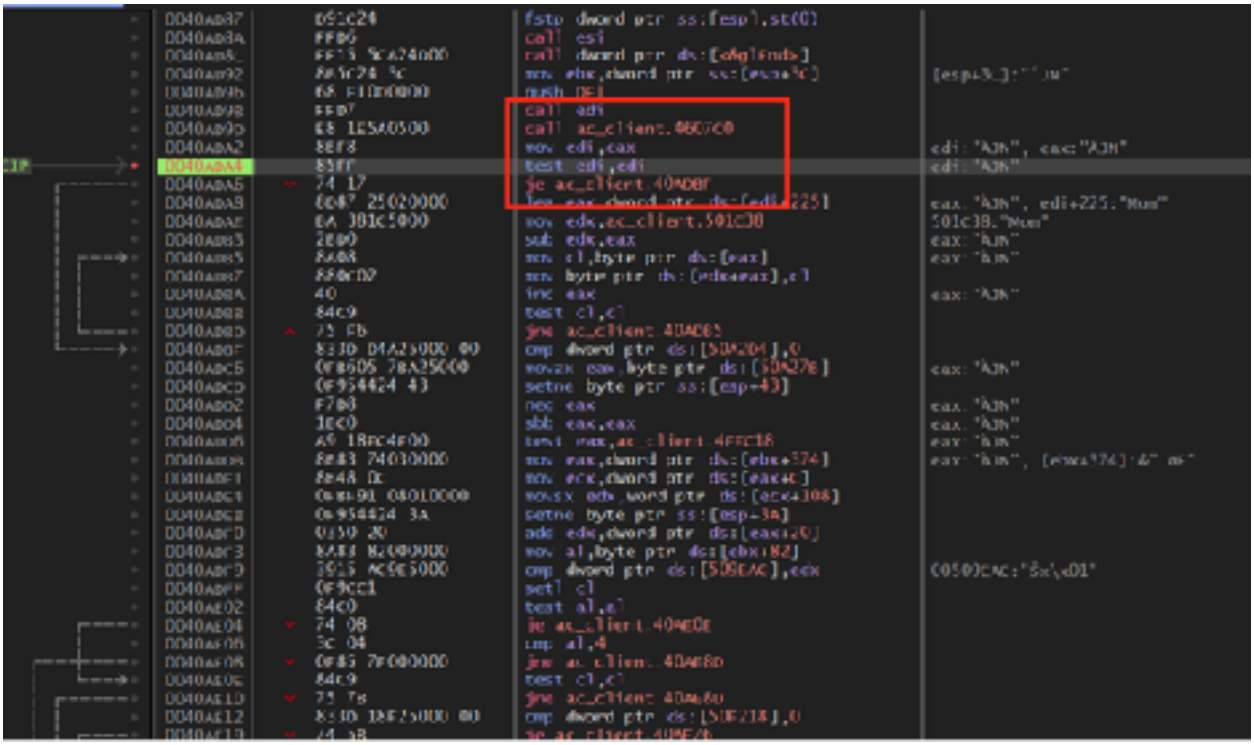
opposite and adjacent sides and want to determine θ. To do this, we can calculate the inverse tangent or arctangent. The arctangent will then represent the angle we need to set our player's aim to aim at an enemy.

However, this will only correctly aim to the left and right. To aim up and down, we will need to do a similar operation for the Y and Z axes.

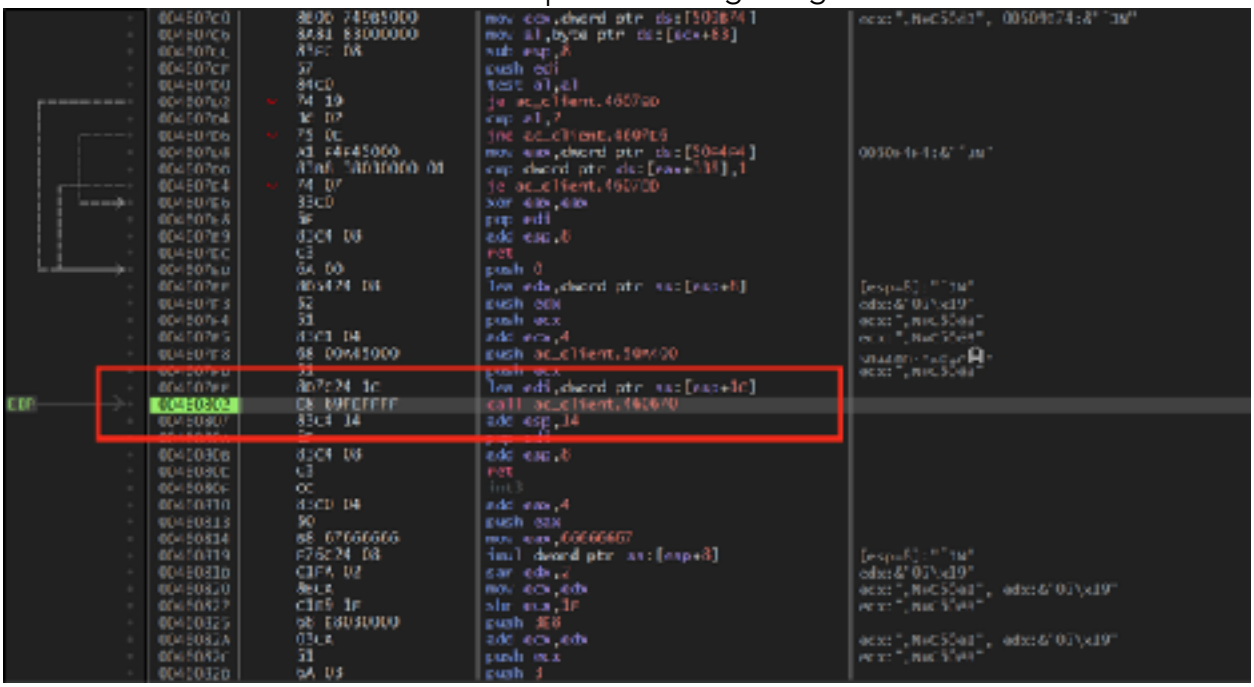
Before we can do any of this, though, we will need to locate where the game stores enemies. Then, we will need to locate where the game stores our player. Finally, we will need to reverse the player structure to locate the X, Y, and Z members in the structure, as well as the view angle members.

5.6.4 Locating Enemies

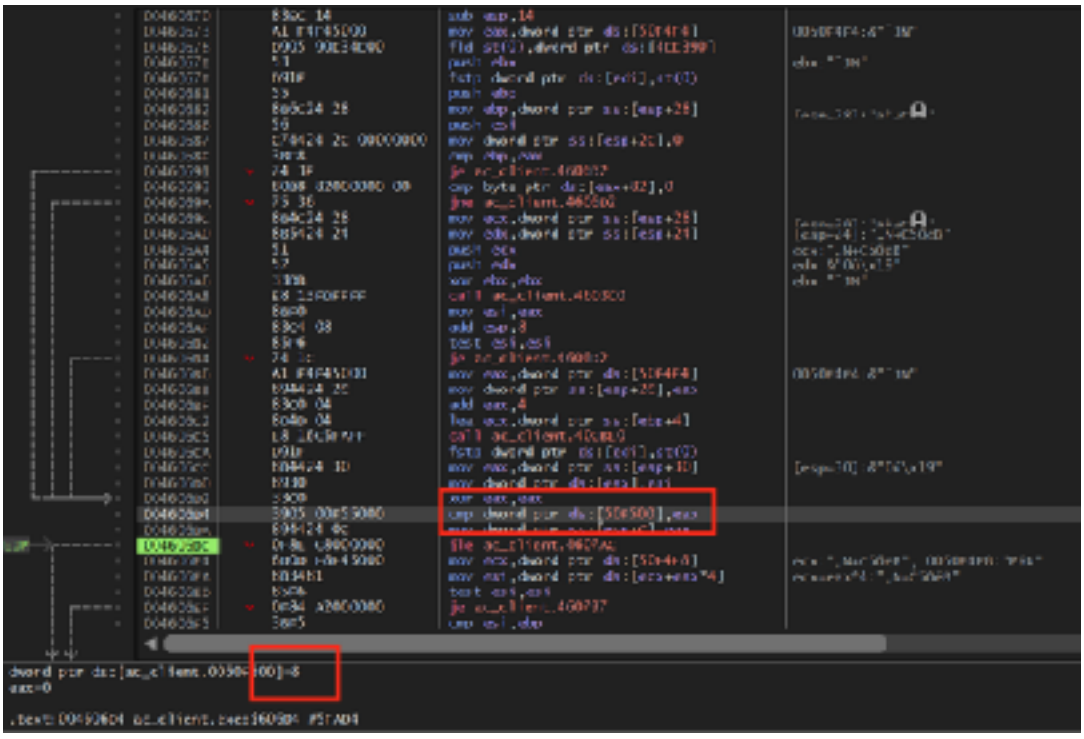
To locate enemies in the game, first create a game with 8 bots and set them to idle. Typically, games will store enemies in a list and hold a static location to this list. In the previous chapter, we found the game code responsible for displaying a player's name when you hovered over that player. To do this, the game must have code inside that function that iterates over the enemies in the game and retrieves their names. This was the code we located in the last chapter:



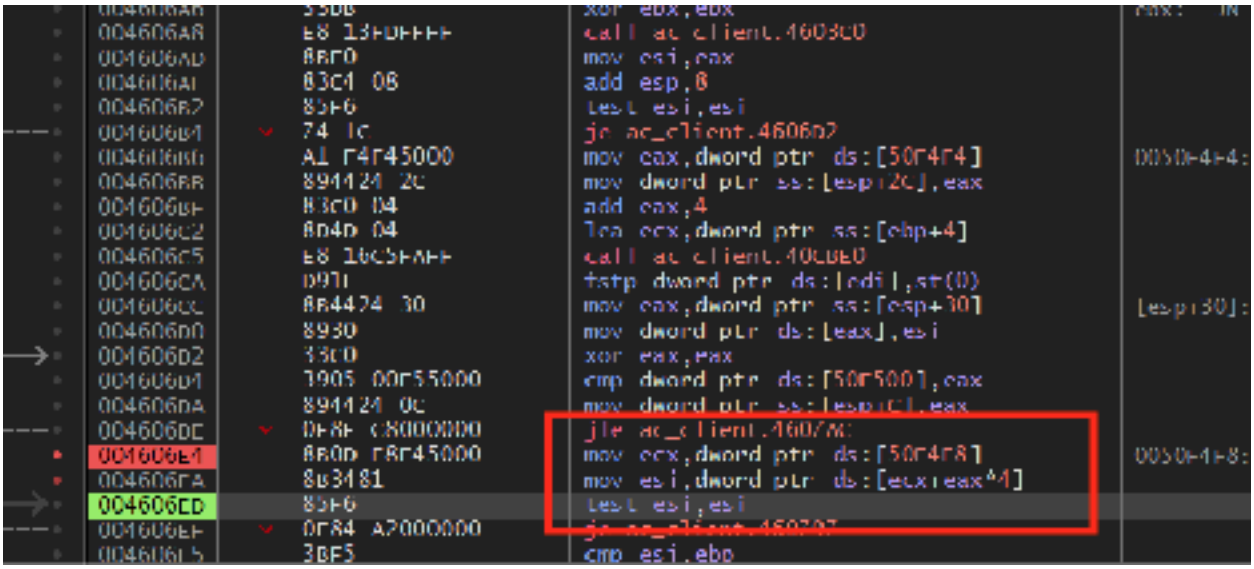
When reversing this code, we determined that the **call** to `0x4607c0` at `0x40ad9d` was responsible for loading the current player looked at into **eax**. If we step into this **call**, we can see that a **call** at the end is responsible for getting this value:



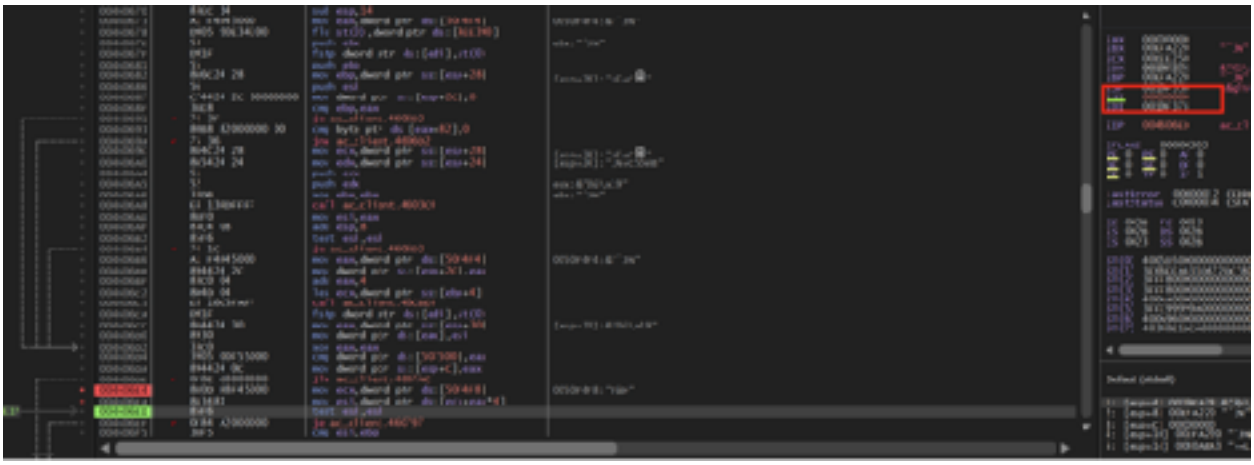
Stepping into this **call**, we can see that it is rather long with many loops and conditionals. As we step through the code, you will notice the following line:



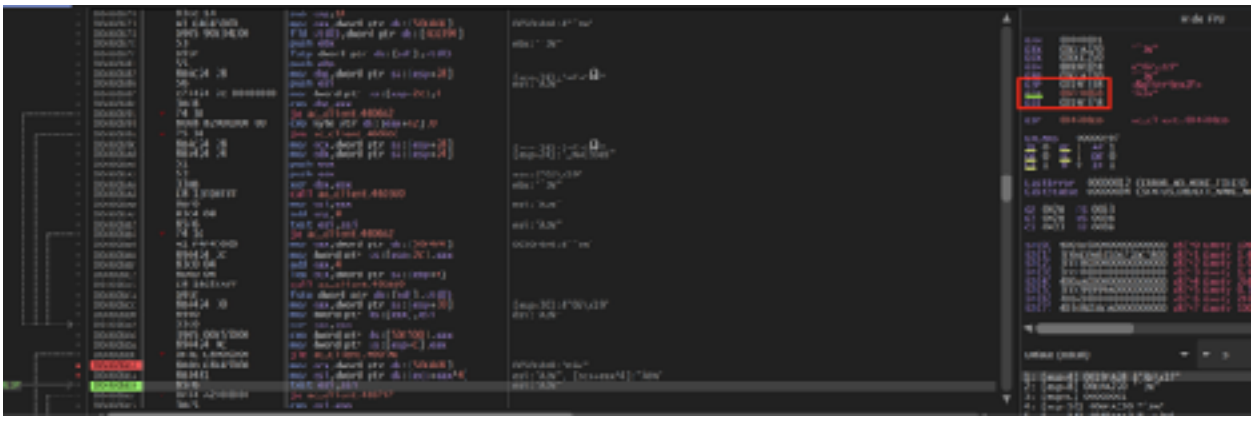
From this, we can determine two things. The first is that **[0x50f500]** will hold the current number of players in the game. We will need this value later when we are iterating through all the players to aim at them. The second is that **eax** is being compared to this value, with a **jmp** below that executes if **eax** is less than this value. This means that we are most likely in code responsible for looping through all the active players. A few lines below, you will notice the following code:



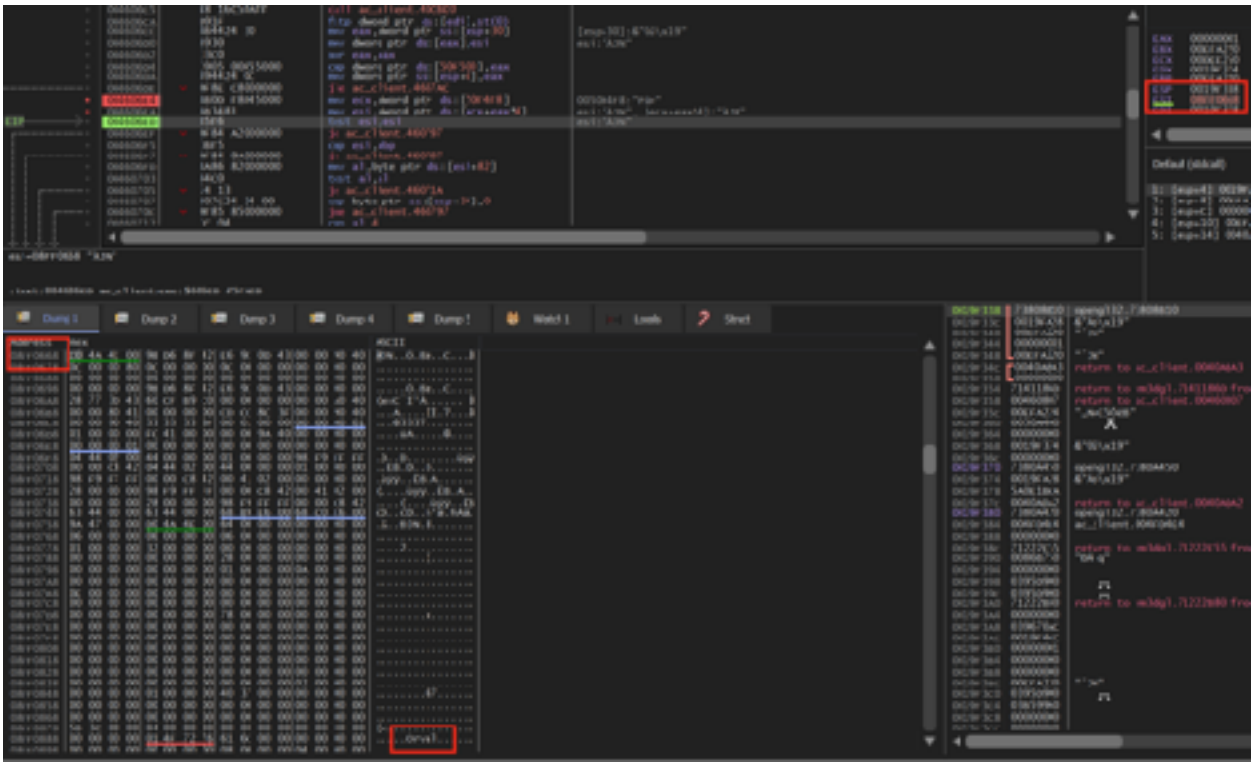
This code is loading a static memory address into **ecx** and then retrieving a new address based on that address's value combined with an offset from **eax**. This new value is then loaded into **esi**. The first time this loop occurs, the value of **esi** is 0:



However, if we continue and execute the loop again, **esi** holds a different value:



If we examine **esi**'s value in the dump, we can see that it is always near an address that holds a player's name:



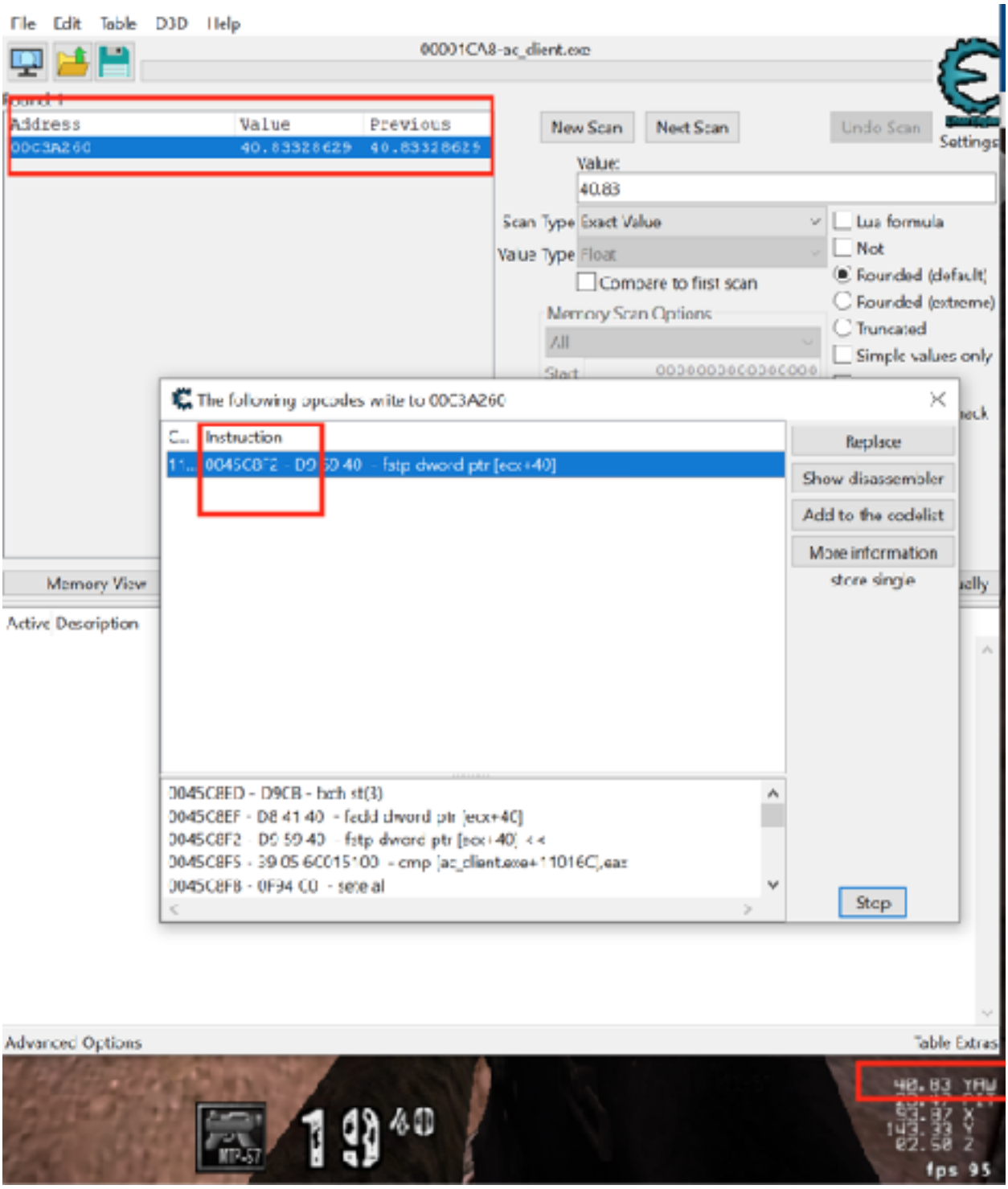
This is most likely the enemy player's structure in memory, as one of the values that will be held in this structure is the player's name. For now, we have identified that the list of enemies is held at **[0x50f4f8]**, with each enemy being at **[[0x50f4f8] + 4,8,C...]**. Once we find our player's structure, we will reverse exactly how the player structure is laid out in memory.

5.6.5 Locating Our Player

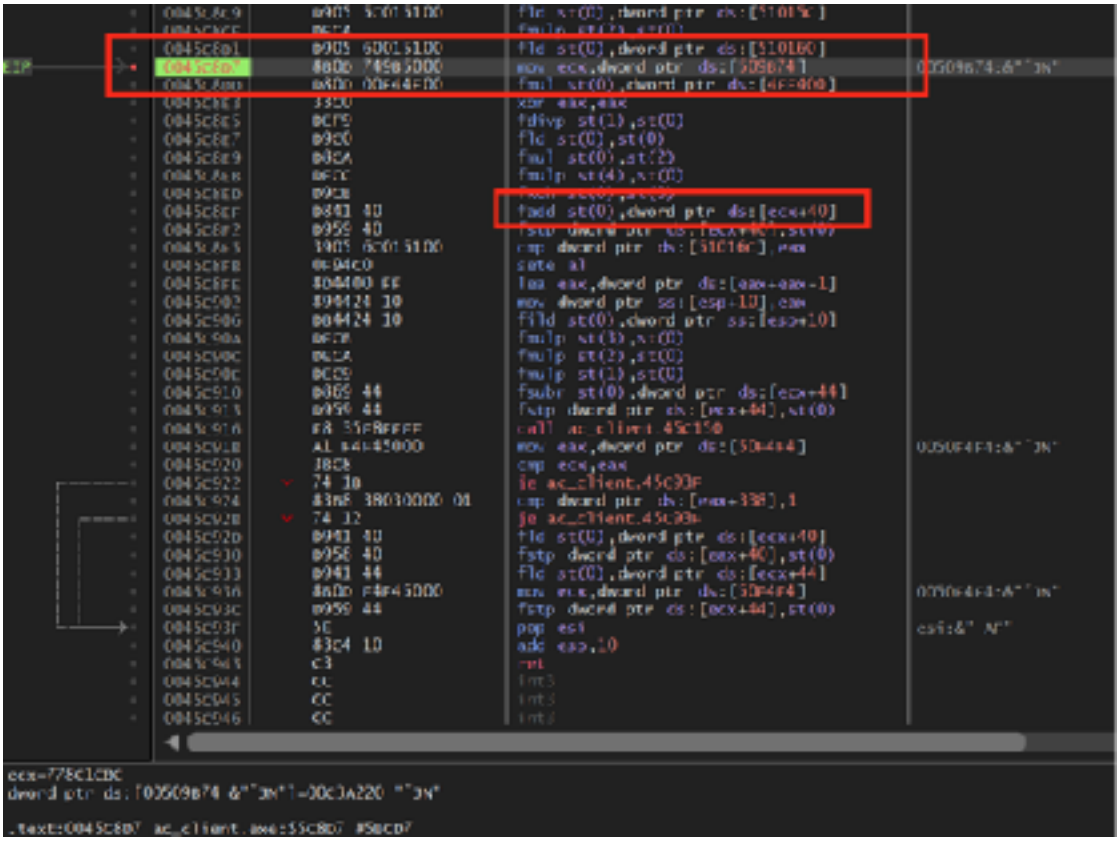
Next, we need to locate where our own player is stored in memory. Since we can never look at ourselves, we need to find our player in a different manner. Many games have a way to print your current position and view angle to the screen. In Assault Cube, this can be done with *dbgpos 1*. When turned on with *showstats 1*, the output looks like:



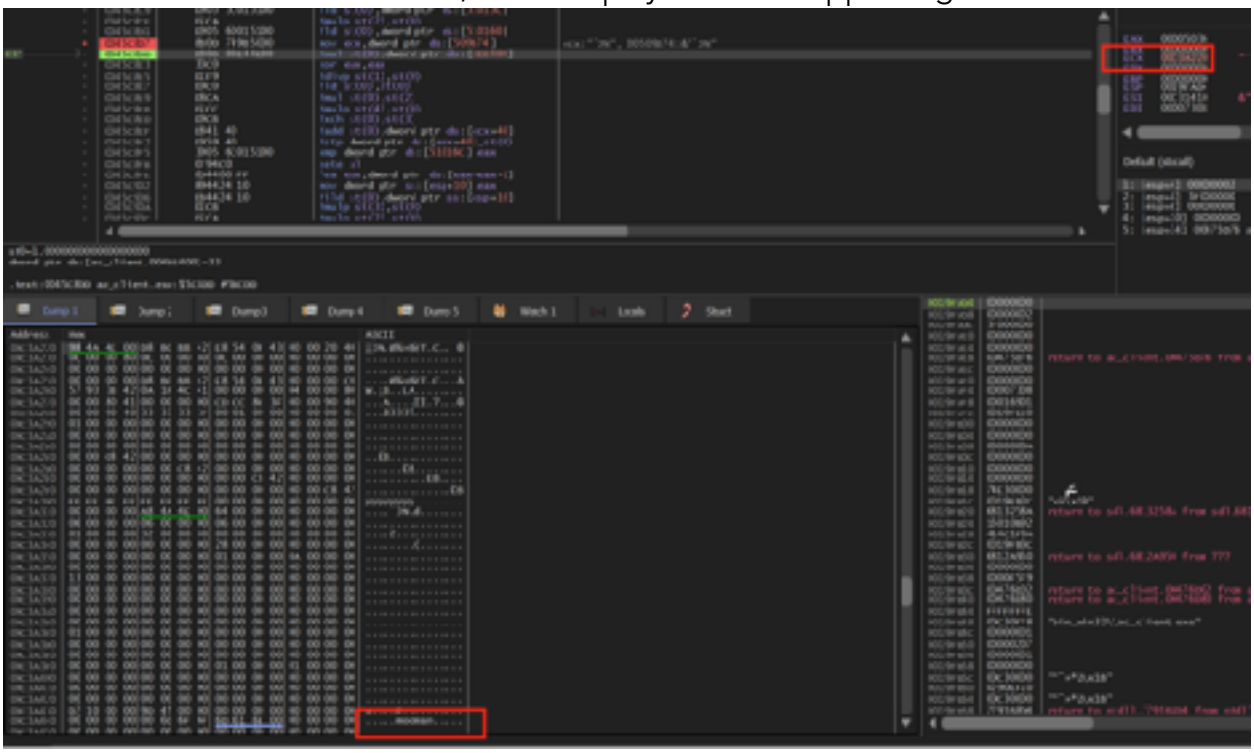
If your target does not have this feature, you will instead have to search for unknown values and then filter while carefully moving your mouse or player in a single direction. In Assault Cube, it looks like our view angle is represented by *yaw* (left and right) and *pitch* (up and down), with both values in degrees. We will need to keep this model in mind for later. For now, we know that our player structure will have to contain these values. In Cheat Engine, we can search for our yaw in memory and then see what accesses the address:



The **fstp** instruction copies a floating point number into the address specified. In this case, that address is based on **ecx**. If we examine this instruction in x64dbg and then view several lines above, we can find where **ecx** is being set:



If we examine the memory pointed to by **[0x509b74]**, we see a structure identical to what we observed with enemies, with our player's name appearing near this address:



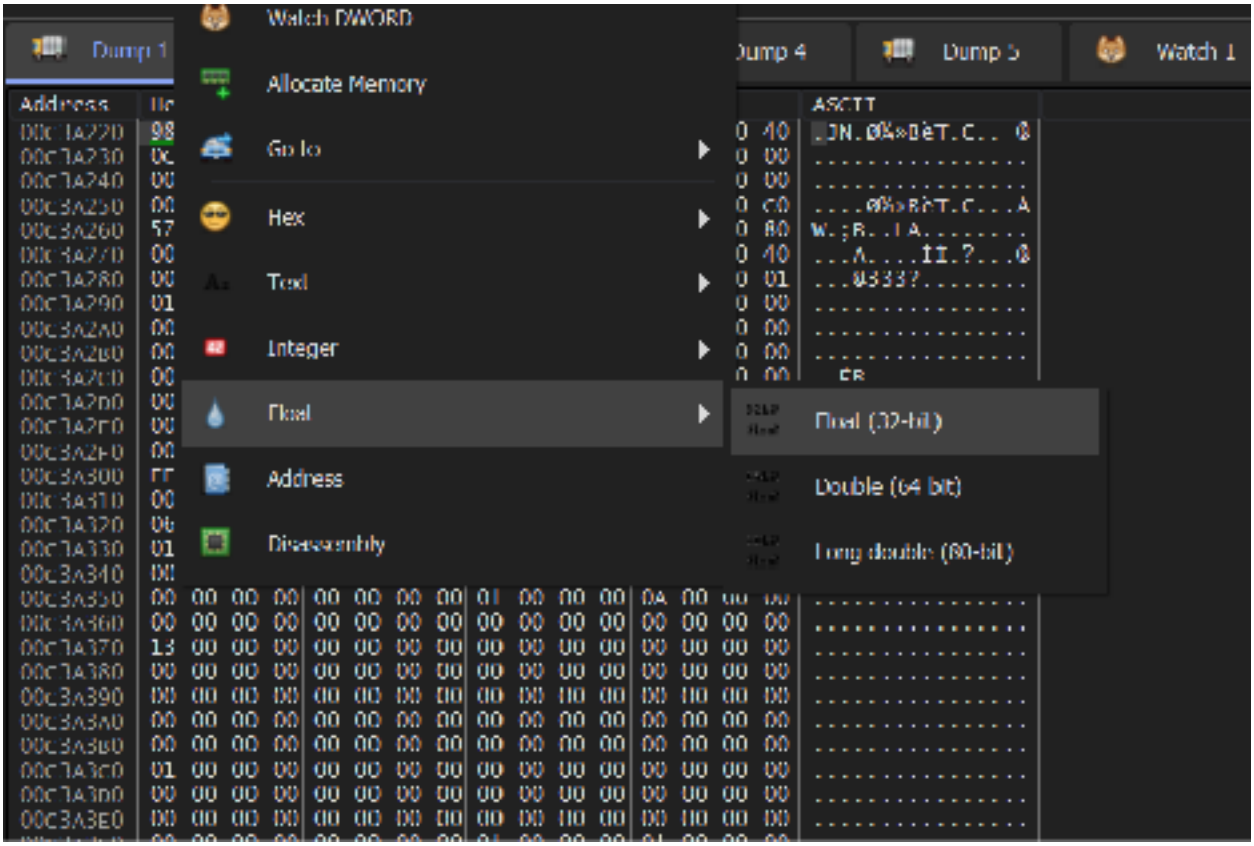
This most likely represents our player's structure. Since we have more control over the values in this structure, we can begin reversing it.

5.6.6 Reversing Player Structure

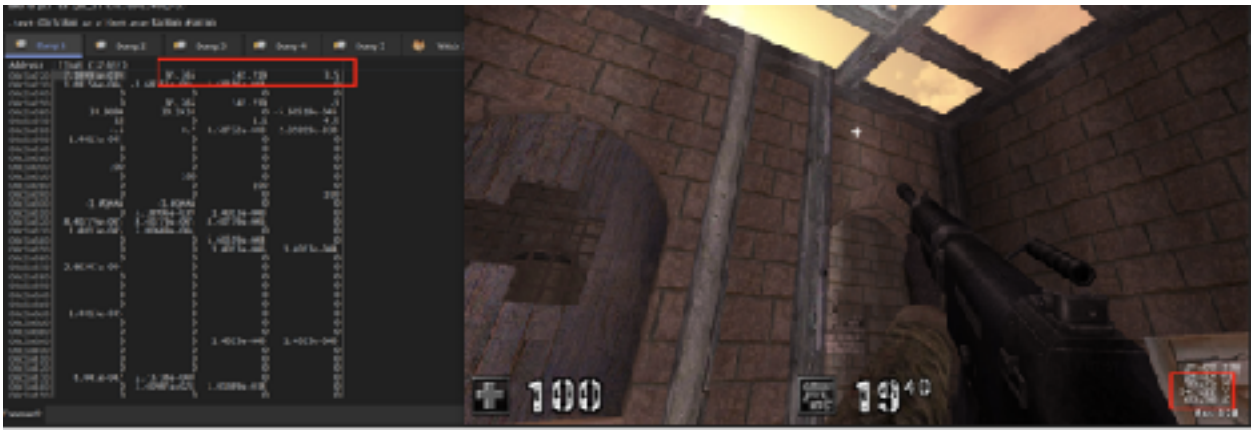
We know that the game must store data about each player in memory. This data will generally be in a continuous section of memory. In C or C++, this would be represented as a structure or a class. For example, a game might define the **Player** structure like:

```
struct Player {
    float x;
    float y;
    float z;
    float yaw;
    float pitch;
    char model_texture_path[128];
    char name[128];
    bool alive;
}
```

When viewed in x64dbg, this structure will appear as a long section of memory since data has no concept of its type. To identify this data, we will need to reverse the structure. x64dbg allows you to modify the data representation in the dump. The default view is hex with ASCII representation. We will start by trying to find the values for our position, which is represented by three float values. We can right-click and choose *Float* to have the dump data displayed in this format:



Upon doing so, several values should jump out immediately, which represent our X, Y, and Z:



Similarly, our yaw and pitch are easily observable as well:



5.6.7 Changing our View Angle

At this point, we have all the offsets we need to start creating our aimbot. The first step is making a DLL that will continuously spin our player in a circle. We want to start with this to ensure that we have correctly located our player and reversed the player structure correctly. Like we have done previously, we will start by creating a thread that will run inside the game's process:

```
#include <Windows.h>

void injected_thread() {
	while (true) {
	  //aimbot code
		Sleep(1);
	}
}

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
	if (fdwReason == DLL_PROCESS_ATTACH) {
		CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread,
NULL, 0, NULL);
	}

	return true;
}
```

We have covered this code several times in previous chapters. Next, we need to define our **Player** structure above our **injected_thread** function:

```
struct Player {

}
```

From our work before, we know that our X, Y, and Z members are at the **base+4**, **base+8**, and **base+C**, respectively. We don't know what the first 4 bytes represent, but we luckily don't need to. Instead, we can create a placeholder member that is an array of characters. We choose characters since they are 1 byte long. We can then create float members for our X, Y, and Z values:

```
struct Player {
	char unknown1[4];
	float x;
	float y;
	float z;
}
```

Next, we need to add in our **yaw** and **pitch** members. If we look at the memory, we see that **Z** ends at `0xc3a230` and **yaw** begins at `0xc3a260`. Like the placeholder above, we will use a **char** member to add `0x30` bytes of padding before adding our **yaw** and **pitch**:

```
struct Player {
	char unknown1[4];
	float x;
	float y;
	float z;
	char unknown2[0x30];
	float yaw;
	float pitch;
}
```

We will then create a pointer from this structure that we will use to map the game's memory into later:

```
Player *player = NULL
```

With our structure created, we can now map the game's memory of the player to our structure. First, we will create a pointer to `0x509b74` in our **while** loop, since this represents the base address of our player:

```
DWORD *player_offset = (DWORD*)(0x509B74);
```

Next, we will dereference this pointer to get the value of the player's base address. We will then map the dereferenced address to our **Player** structure pointer. This will store the values we observed in the dump into this structure so that we can reference them in our code.

```
player = (Player*)(*player_offset);
```

Finally, we will increase the **yaw** member in a loop to cause our player to spin in a circle:

```
player->yaw++;
```

From here, we can build and inject this DLL. Our player will now spin around in a circle, showing that we have correctly reversed the player structure.

5.6.8 Aiming Left and Right

With all of this in place, we can create the first version of our aimbot. This version will aim left and right at a single opponent. When testing this out, make sure to create a two-player game, with you and a single bot. This will let you nail down the math.

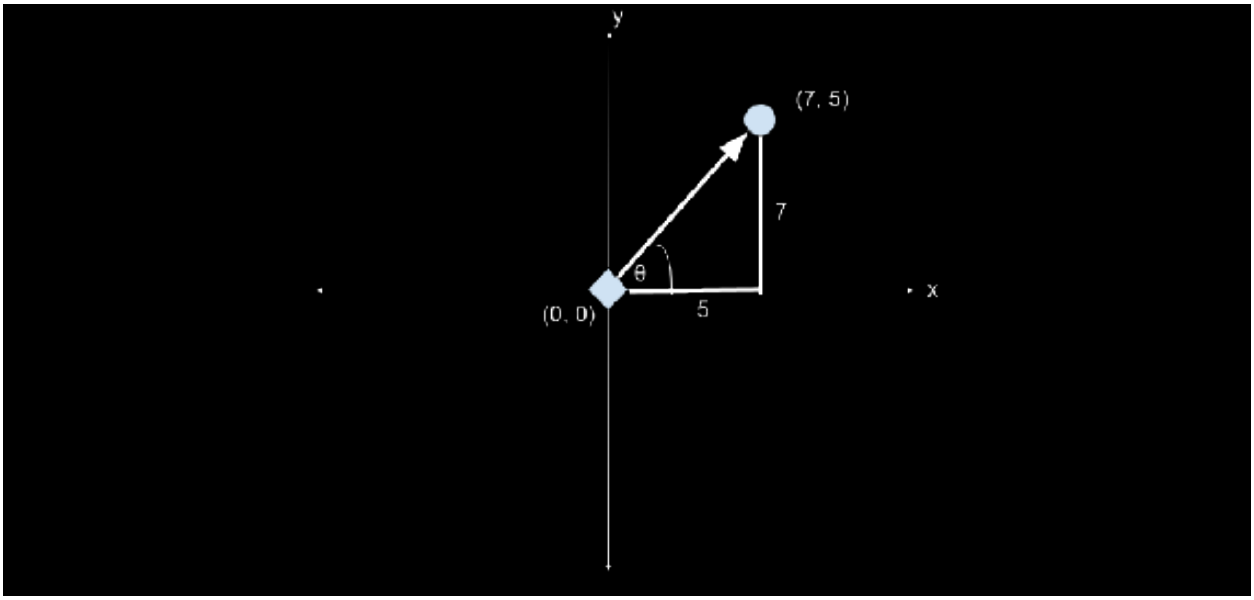
First, we will use the same approach as before to map the first enemy into a **Player** structure. When reversing the code, we identified that the first enemy was at **+4**:

```
DWORD* enemy_list = (DWORD*)(0x50F4F8);
DWORD* enemy_offset = (DWORD*)(*enemy_list + 4);
Player* enemy = (Player*)(*enemy_offset);
```

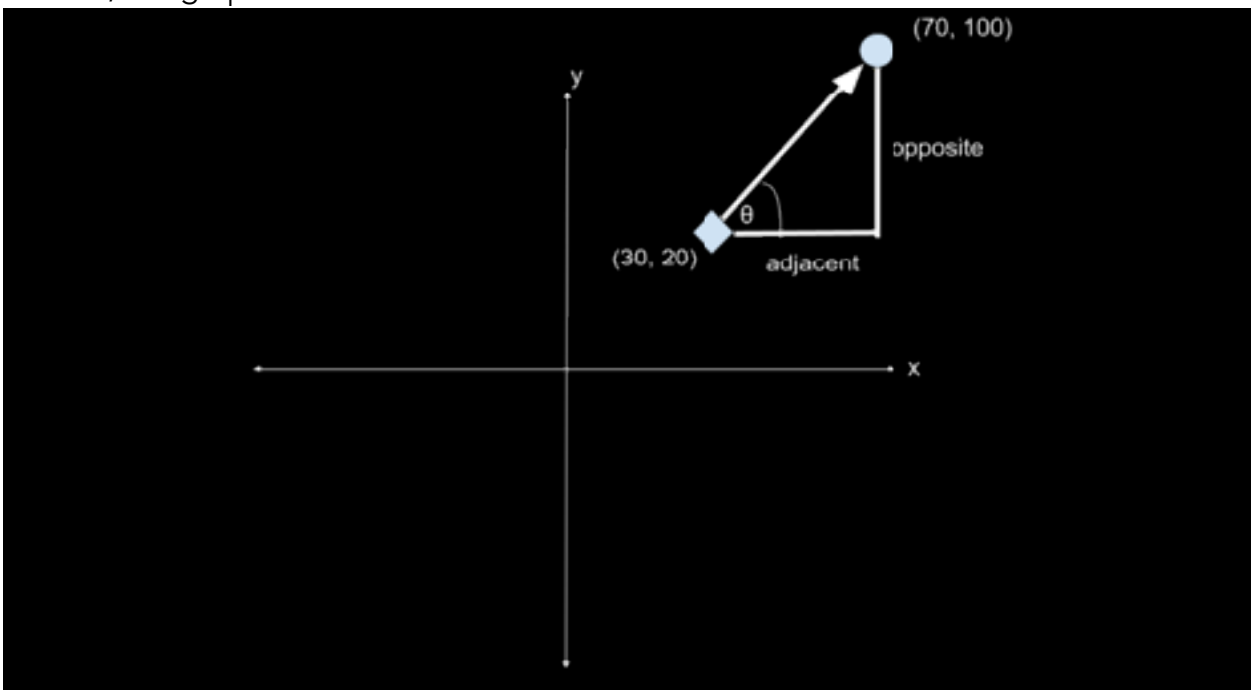
One issue with our previous code was that we would crash if we were not in a game. That's because we were accessing memory that wasn't valid. To prevent this, we will check to make sure both of our pointers are valid before continuing:

```
if (player != NULL && enemy != NULL) {
```

At the beginning of this chapter, we had the following graph:



In this graph, we knew the opposite and adjacent distances based on the enemy's position. However, as we have seen when reversing, our position is never (0, 0). Instead, the graph would look more like:



If we attempt to use the enemy's position, our calculations will be incorrect. Instead, we need to determine these values by subtracting the enemy's position from the player's position. This will give us values that will act as if the player is always at (0, 0), or the absolute position (**abspos**) between our player and the enemy:

```
float abspos_x = enemy->x - player->x;
float abspos_y = enemy->y - player->y;
```

Next, we can calculate the arctangent using the **atan2f** function. We use this function as opposed to **atanf**, as it takes care of the case in which **abspos_y** is less than 0. Since the inverse tangent is an unsigned operation (i.e., it doesn't have a concept of positive or negative), our aimbot would aim in the opposite direction if the enemy was directly behind us. We could manually check for this by checking **abspos_y**, but **atan2f** takes care of this calculation for us:

```
#include <math.h>

...
float azimuth_xy = atan2f(abspos_y, abspos_x);
```

The **atan2f** function produces a radian value. When reversing, we saw that the game represents our yaw as a degree value. To convert the radian to a degree value, we can multiply the radian by **(180 / π)**:

```
#define M_PI 3.14159265358979323846

...
float yaw = (float)(azimuth_xy * (180.0 / M_PI));
```

Finally, we can set our player's **yaw** to this value:

```
player->yaw = yaw;
```

If you inject this code into the game, you will notice that you aim close to the player, but always a consistent amount of pixels to the left or right, depending on where you are standing. This is because in our graphing model, we assumed that 0° was facing straight ahead. However, if you join a game without the hack, you will notice your player's starting yaw is 90°. To compensate for this, we can simply add **90** to our calculated yaw:

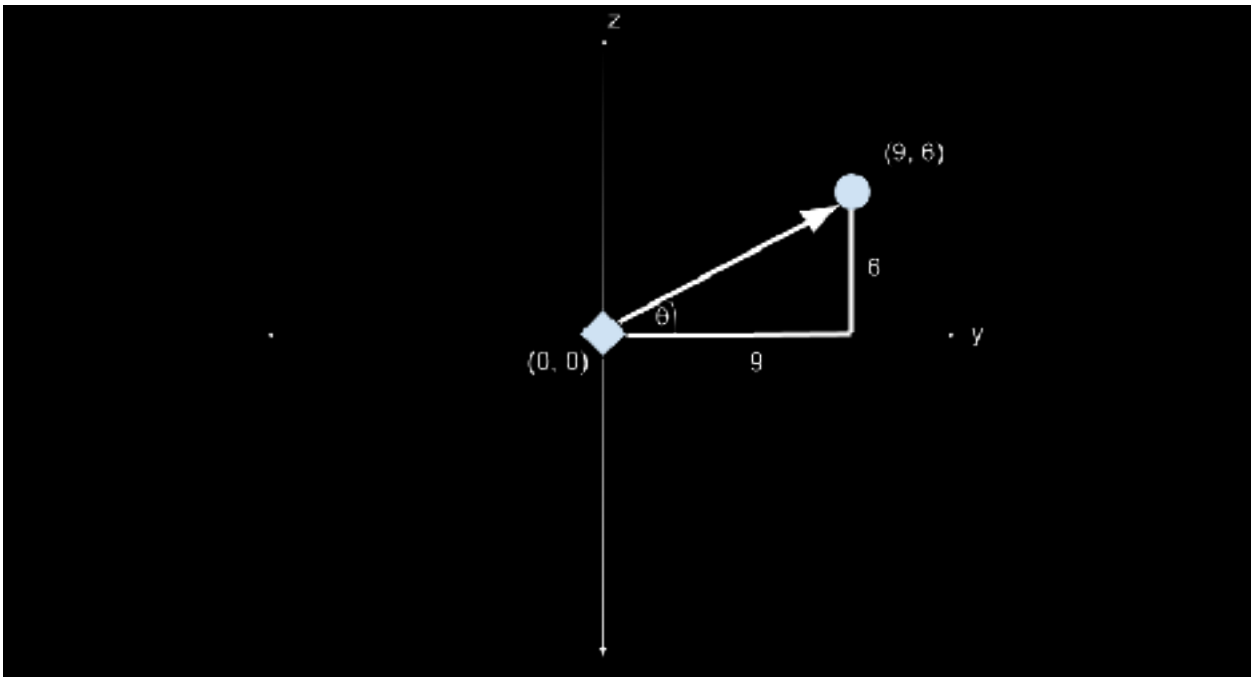
Playrix Ex. 1024, page 247

```
player->yaw = yaw + 90;
```

With this change, we can run around the map and constantly stay locked on a player. However, if we jump up and down or go up an incline, we will be aiming above or below the enemy. Our next step is to set our pitch (or up and down) value correctly.

5.6.9 Aiming Up and Down

When we first approached this problem, we quickly set our perspective as top-down to eliminate the Z axis. To calculate our up and down angle, we will now fix our perspective as right-left (i.e., on the right of the player, looking directly at the player's right side). The most important thing to note in our new graph below is the different axis values:



We can use a similar approach to the left and right angle to calculate the up and down angle. First, we will get the absolute distance:

```
float abspos_z = enemy->z - player->z;
```

Then, like before, we will calculate the inverse tangent. Unlike the yaw, our initial pitch starts at 0, so we don't need to add any value to it:

```
float azimuth_z = atan2f(abspos_z, abspos_y);
player->pitch = (float)(azimuth_z * (180.0 / M_PI));
```

If you inject this code, it will appear to initially work. However, when you get within arm's distance of an enemy, your player will suddenly look straight up or straight down. This is due to the game having very limited Z values. For example, most maps in the game have Z values between 0 and 6. When the value of Y gets too small, the resulting equation ends up being skewed. Imagine the case where the difference in Z values was 3 but the Y value difference was 1, or arctan(3 / 1). This resolves to 75°, which is effectively straight up in the air when it comes to pitch.

To account for this behavior, we will look at the value of Y and ensure that it is reasonably large. If it's not, we will use X. This is not perfect, but it will help alleviate some of the issues. We will also ensure that the value is positive, regardless:

```
if (abspos_y < 0) {
    abspos_y *= -1;
}
if (abspos_y < 5) {
    if (abspos_x < 0) {
        abspos_x *= -1;
    }
    abspos_y = abspos_x;
}
```

Now you will notice that you can run up directly to the enemy and your aim will not jump in the air. Our aimbot is now working for a single enemy.

5.6.10 Multiple Enemies

With this foundation down, we can modify our aimbot to work with multiple enemies. To do this, we will change the code to iterate through the enemy list, pick an enemy to aim at, and set our yaw and pitch to aim at them. To pick the enemy, we will choose to always select the enemy closest to us. This will not always be the best case. For example, if one enemy is down a hall and one enemy is behind a wall next to us, our aimbot will always pick the enemy behind the wall. However, for the purpose of this chapter, this method is the easiest to implement.

To find the enemy closest to us, we will calculate the Euclidean distance between our player and the enemy. The lower the value, the closer the enemy is to us:

Playrix Ex. 1024, page 249

```
float euclidean_distance(float x, float y) {
      return sqrtf((x * x) + (y * y));
}
```

Since we need to iterate over a list of enemies, we will create a variable to hold the closest enemy distance, as well as their associated yaw and pitch values:

```
while (true) {
    DWORD* player_offset = (DWORD*)(0x509B74);
    player = (Player*)(*player_offset);
    ...
    float closest_player = -1.0f;
    float closest_yaw = 0.0f;
    float closest_pitch = 0.0f;
```

At the beginning of this chapter, we determined the address that held the current number of players in the game. We can finally use that value now:

```
int* current_players = (int*)(0x50F500);
```

We can now iterate over all the enemies in the game. Unlike before, where we always added 4, we will now add the current loop index multiplied by 4, identically to how the game did it:

```
for (int i = 0; i < *current_players; i++) {
    DWORD* enemy_list = (DWORD*)(0x50F4F8);
    DWORD* enemy_offset = (DWORD*)(*enemy_list + (i*4));
    Player* enemy = (Player*)(*enemy_offset);
```

We can then calculate the absolute positions like we did before. However, before calculating the yaw or pitch, we will calculate the distance from our player to the enemy and ensure that they are the closest enemy. If they are, we will then set the **closest_player** value to their distance for future checks:

```
float temp_distance = euclidean_distance(abspos_x, abspos_y);
if (closest_player == -1.0f || temp_distance < closest_player) {
    closest_player = temp_distance;
```


Playrix Ex. 1024, page 250

Next, instead of directly setting the player's yaw and pitch, we will store these in our variables. Once we have iterated over all the enemies, we will set the player's yaw and pitch. This ensures that we aren't constantly flickering through multiple enemies:

```
closest_yaw = yaw + 90;
...

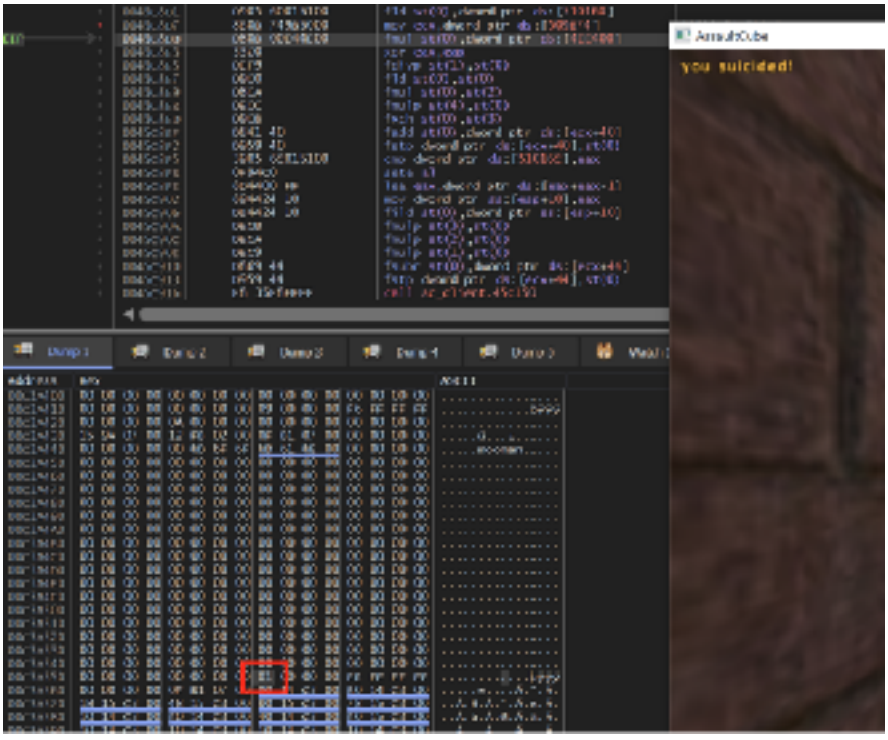
closest_pitch = (float)(azimuth_z * (180.0 / M_PI));
...

player->yaw = closest_yaw;
player->pitch = closest_pitch;

Sleep(1);
```

We now have a working aimbot that will iterate through multiple enemies and aim at the closest one correctly in the X and Y axis.

Finally, we can add a check to see if the enemy is alive, to ensure that we instantly switch from a target when we shoot them. This value can be found by observing the player structure for values that change when you are alive or dead. After killing yourself several times, you will notice that one value is set to 0 when you are alive and 1 when you are dead:



We can add this to our player structure, ensuring that we correctly offset it:

```
      float yaw;
      float pitch;
      char unknown3[0x2f0];
      int dead;
};
```

We can then check this value in our initial check to ensure that the player and enemy are valid:

```
if (player != NULL && enemy != NULL && !enemy->dead) {
```

The full code is available in Appendix A for comparison.

Playrix Ex. 1024, page 252

5.7 No Recoil

5.7.1 Target

Our target for this chapter will be Assault Cube 1.2.0.2.

5.7.2 Identify

Our goal in this chapter is to create a no recoil hack, a type of hack that eliminates recoil when firing. Recoil is defined as the automatic upward motion of your player's view when firing a weapon.

5.7.3 Understand

When firing a weapon in an FPS, different effects are applied by most games:

- Recoil (up and down movement)
- Spread (crosshair widening, random distribution of shots)
- Pushback (player pushed in the opposite direction they are firing)

Our focus in this chapter is to remove recoil only.

In most games, these effects are applied consecutively after each shot is fired. Recoil generally works by increasing the player's up and down view angle by adding a certain value to it. Because view angles are usually floating point numbers, this operation will typically take the following assembled form:

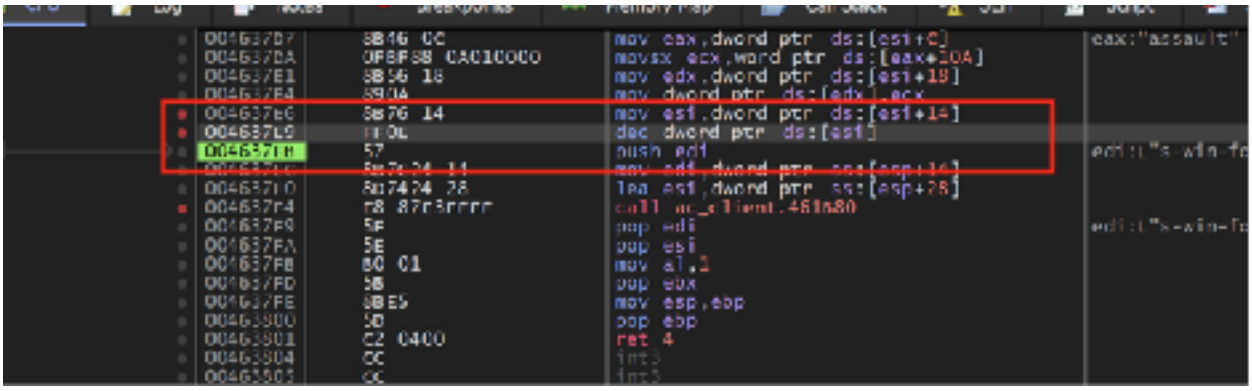
```
fld recoil_amount                          ; load recoil amount into st(0)
fadd st(0), players_y_view_angle           ; add recoil_amount to view angle
fstp players_y_view_angle, st(0)           ; store result in view angle
```

Unlike integers, float values must be pushed on a special register stack to be operated on known as the FPU stack. However, like normal instructions, if this code is **nop**'d out, recoil will not be applied to the player.

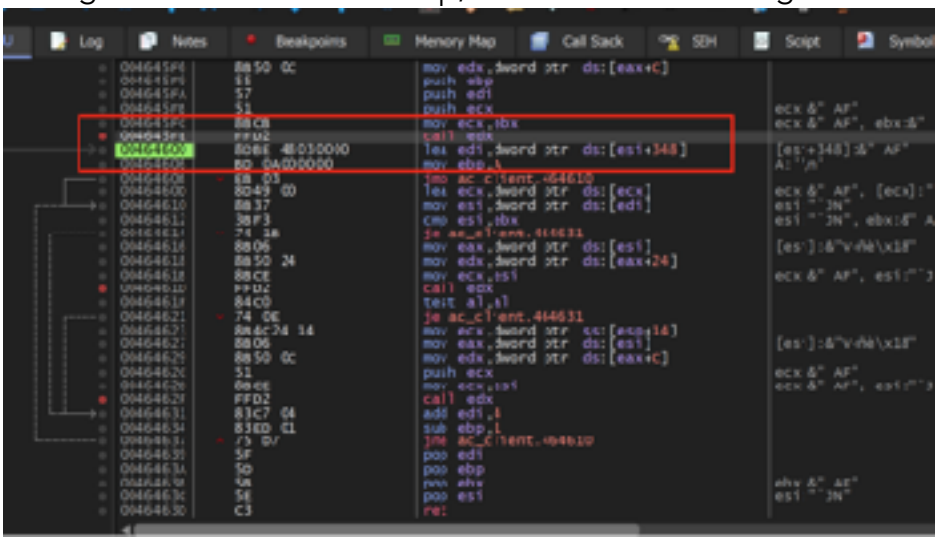
When firing a weapon, games execute several functions, including playing a sound, displaying a firing animation, and decreasing the player's ammo. These functions are often located near the function that applies recoil to the player's view. We can use these functions to help locate the recoil code. We have multiple approaches that we can use to locate this code. In this chapter, we will use the code responsible for decreasing the player's ammo, as this value is easy to search for.

5.7.4 Locating Firing Function

Start a game of Assault Cube and use Cheat Engine to locate your current ammo count, using the same approach discussed in Chapter 1.5. Once that is identified, attach x64dbg to Assault Cube and set a hardware breakpoint on write on the identified address. When you go back to Assault Cube and fire, the breakpoint should pop at the following location:

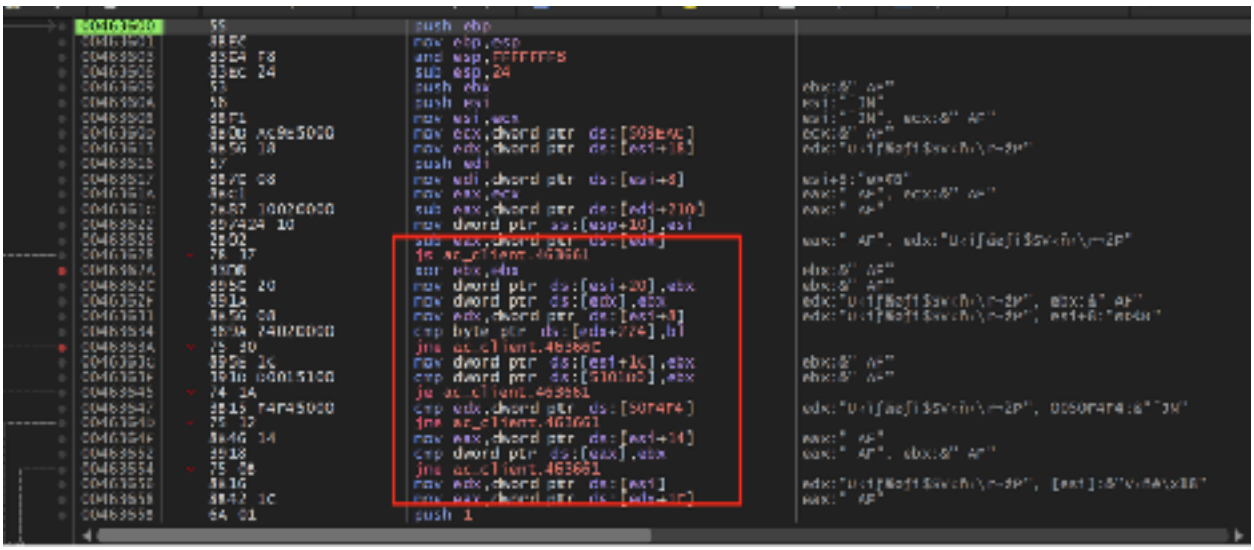


We can see that this code is responsible for decreasing the ammo count. If we step out of this code using execute until return/step, we see that the calling location is here:

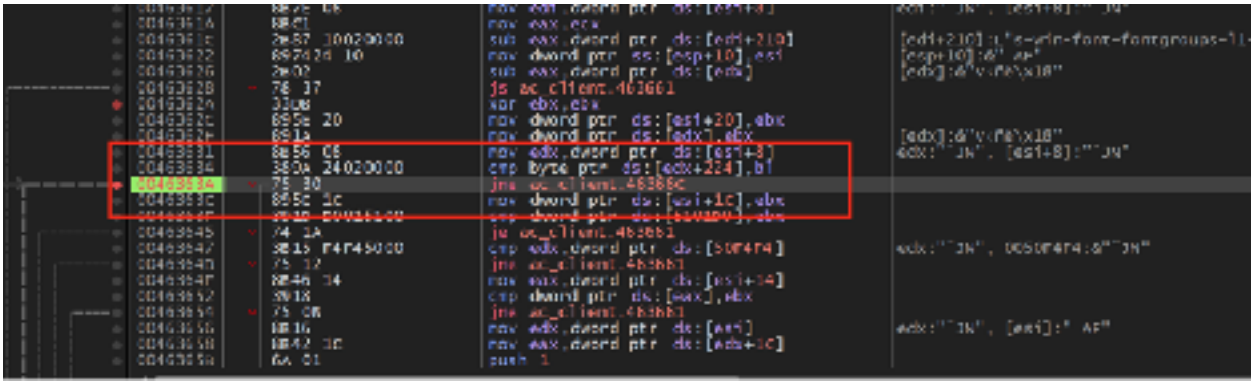


Next, let's determine our context in the code. We want to determine if we are in the code responsible only for setting the ammo count or if we are in the general firing code. We can do this by setting a breakpoint on the **call edx** instruction. After that, we can see that this code is called constantly, whether we are firing or not. This means that we are too high-level and we will need to dig into this function.

If we step into the **call** after the breakpoint is triggered, we can see that the function has several branches:

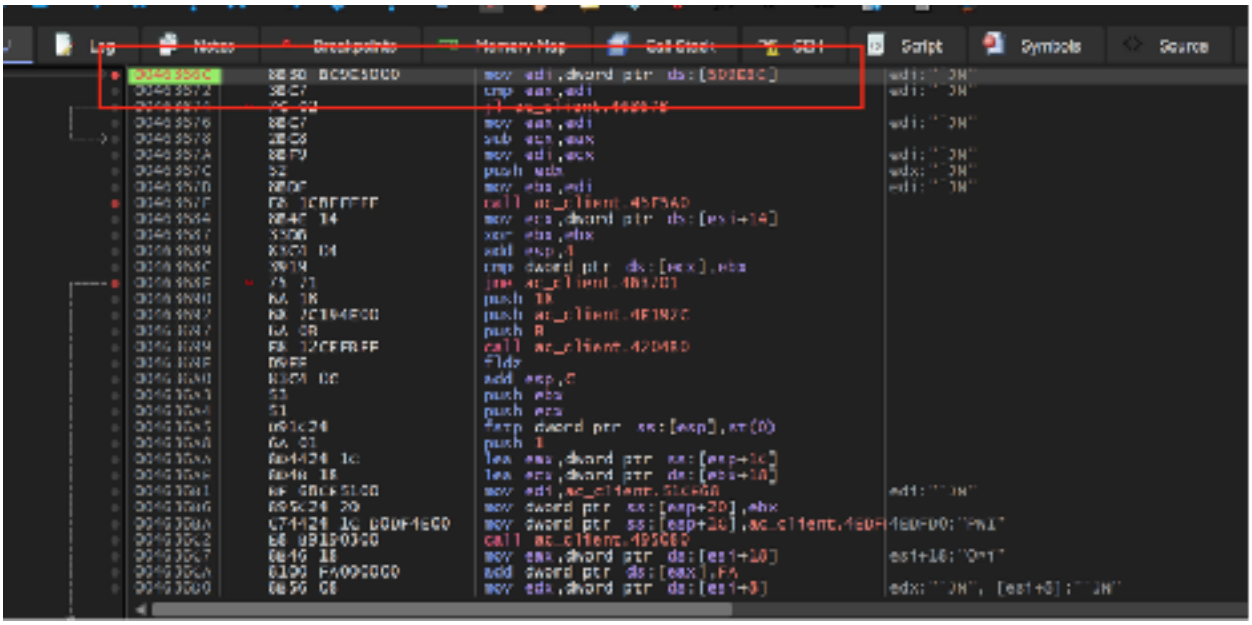


If we step through the code, we can see that the **jmp** at `0x46363A` is not taken if we are not firing:



If we change this to a **jmp** and go back in the game, we will notice that our player now fires constantly, even if we are holding down the mouse button. This **jmp** appears to be

responsible for checking if the player is firing. If we follow this **jmp**, we can see that it jumps past a return statement and to the following instruction:



When we set a breakpoint on this instruction, we see that this code is only called when we are actively firing.

5.7.5 Locating Recoil

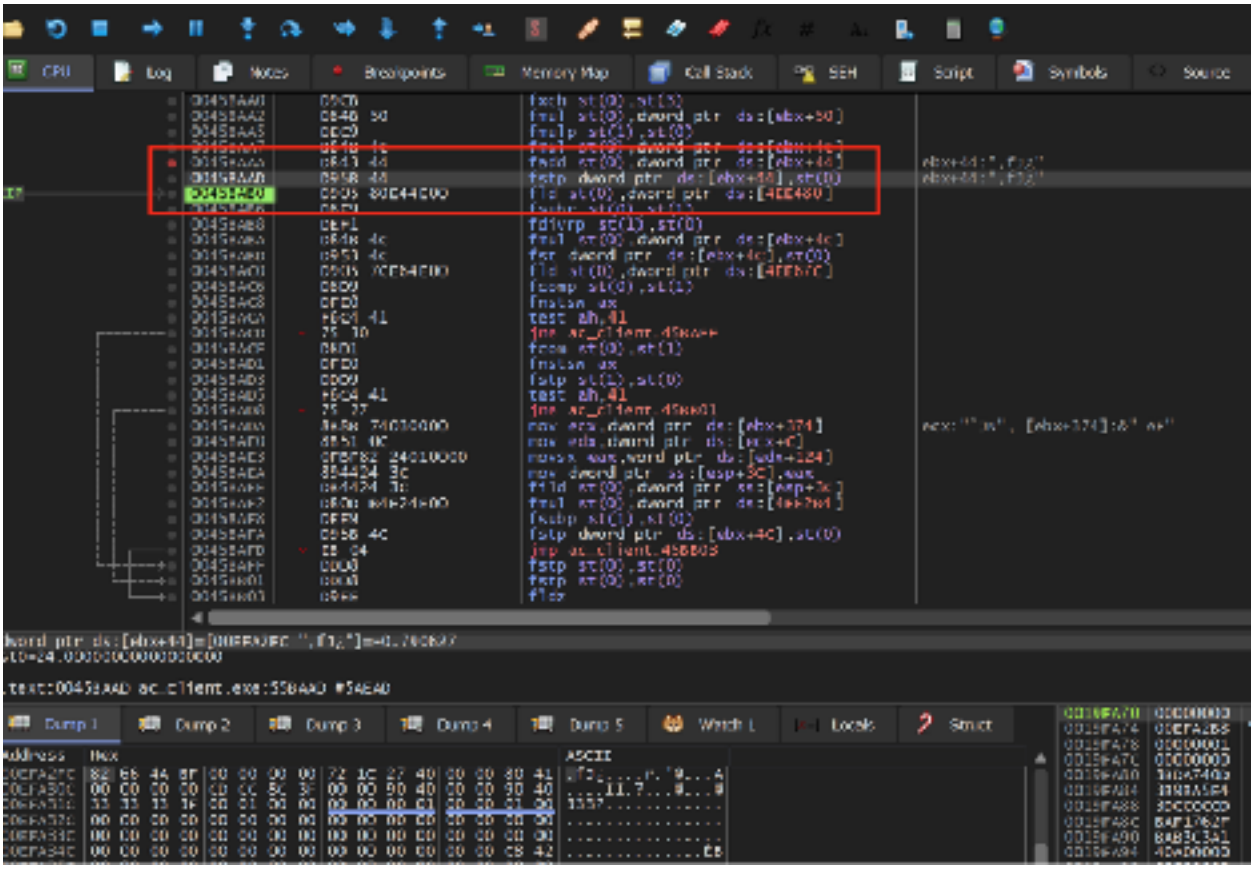
We have now found the beginning of the weapon firing code. We could also see that one of the final instructions in the weapon firing code is responsible for decreasing the ammo. Therefore, somewhere between these two instructions is the code responsible for adding recoil.

While we could step through this code to identify the instruction, we can use a quicker approach. We know that the recoil instruction must modify the player's yaw value. After we hit our breakpoint on our weapon firing, if we then set a breakpoint on the yaw value, we can continue execution and wait for the breakpoint to pop. This prevents us from stepping through a large amount of code.

It's important that we only set the breakpoint on the yaw value after the firing code is started. Assault Cube, like many other games, constantly writes to the yaw value. If we just set a breakpoint on it without being in the firing code, we will end up in another section of code.

Playrix Ex. 1024, page 256

We can locate the address of the yaw value using the same approach discussed in the previous chapter or by searching for it in Cheat Engine. After that, set a breakpoint on the start of the firing code at `0x46366C`. Then, fire a single shot so the breakpoint pops. When it does, set a breakpoint on write on the address of the yaw value. Continue execution and the write breakpoint should pop at the following code:

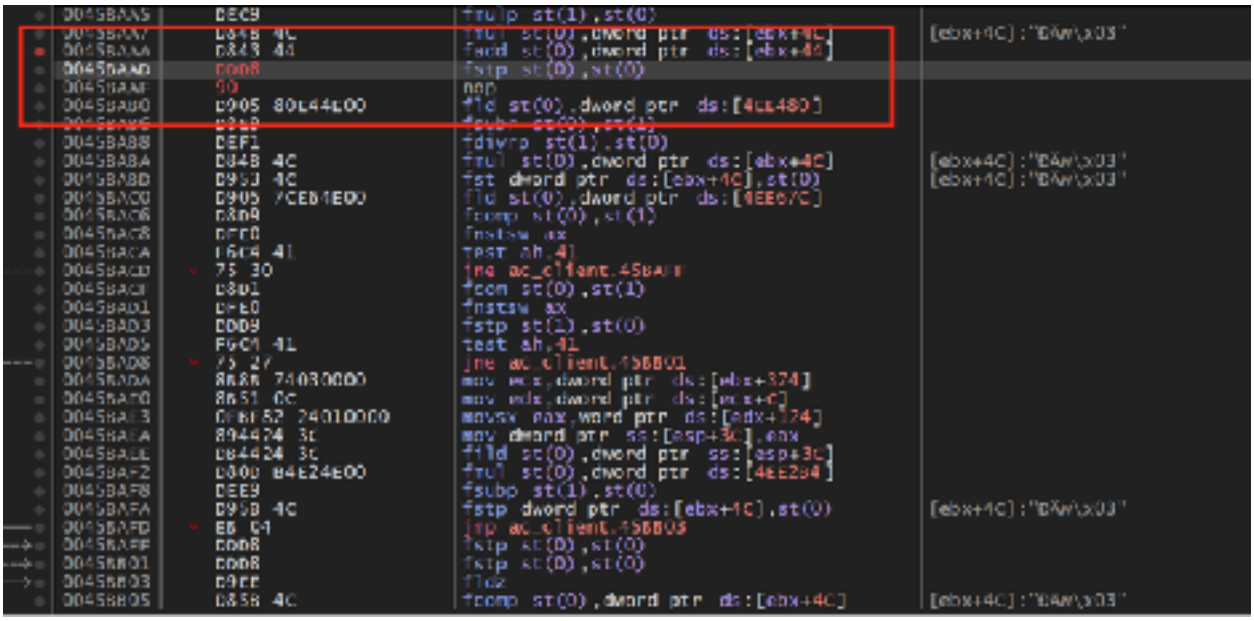


We can see that this code matches the pattern we expected. In this particular code, **dword ptr ds:[ebx+0x44]** is responsible for holding the player's yaw value. The recoil value is held on the top of the FPU stack, which is pointed to by **st0**.

The operation to calculate recoil appears to be composed of several instructions. While we could investigate the exact way in which the recoil is set, we can skip that process to make a no recoil hack and simply prevent the recoil value from being placed in the player's yaw value.

The **fstp** instruction is responsible for popping the top value off the FPU stack into the provided address. Since we do not want to corrupt the stack, we do not want to **nop** this instruction, as the stack would then have an extra value on it. Instead, we will just

pop the value off the top of the stack into **st0**. Since **st0** is then set in the next instruction, this will result in the value effectively disappearing:



With this change made, you will notice that you no longer have any recoil in the game.

5.7.6 Changing Recoil

Finally, we can write a DLL to make this change automatically. Since this hack only requires us to change bytes at an instruction, we can use the same template we covered in Chapter 4.2. For this hack, we will change the code for editing the bytes at `0x45BAAD` to the identical values we observed in x64dbg:

```
#include <Windows.h>

unsigned char new_bytes[3] = { 0xDD, 0xD8, 0x90 };

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
	DWORD old_protect;
	unsigned char* hook_location = (unsigned char*)0x45BAAD;

	if (fdwReason == DLL_PROCESS_ATTACH) {
		VirtualProtect((void*)hook_location, 3, PAGE_EXECUTE_READWRITE,
&old_protect);
		for (int i = 0; i < sizeof(new_bytes); i++) {
```

```
                *(hook_location + i) = new_bytes[i];
            }
        }

        return true;
}
```

The code for this hack is also available in Appendix A.

Playrix Ex. 1024, page 259

5.8 Radar Hack

5.8.1 Target

Our target for this chapter will be Assault Cube 1.2.0.2.

5.8.2 Identify

Our goal in this chapter is to create a radar hack, a type of hack that displays both enemies and friendly players on the radar.



5.8.3 Understand

Many FPS games have a radar that will display an icon for each player on top of a scaled-down version of the current map. When playing in a game with teams, these radars will only show the players on the same team as the active player. In Chapter 5.6, we discovered the location of the list of players in the game. The code for most games will iterate over this list when drawing a player's icon on the radar, in a similar manner to the following block:

```
void draw_radar() {
    for(int i = 0; i < max_players; i++) {
        if(player_list[i]->team == current_player->team) {
            //draw on radar
        }
    }
}
```

If we locate this code, we can change the if conditional to always draw the player on the radar regardless of the team. To locate the code, we will first need to identify the **team** member in our player structure. Then, we can set a breakpoint on access to identify where this member is accessed in code.

5.8.4 Locate Player's Team

We can use two approaches to locate the player's team in the player structure:

1. Use Cheat Engine and search for an *Unknown initial value*. Then, alternate between teams and search for *Changed value*.
2. Use x64dbg and locate the player's structure in a dump. Then, alternate between teams and search for a member that changes.

We have covered both of these approaches in past chapters. Using the techniques discussed previously, you should be able to identify a member that alternates between 0 and 1 depending on your team. This member is relatively close to the member we identified previously that held whether the player was alive or not.

5.8.5 Locate Radar Function

We know that the radar function must access the player's team. Therefore, we can place a breakpoint on access on the team member we just identified. Immediately, the breakpoint should pop. However, if you continue execution several times, you should see that the breakpoint pops in completely different sections of code. This is most likely because several sections of code access this member. Our next step is identifying the section of code responsible for drawing the radar.

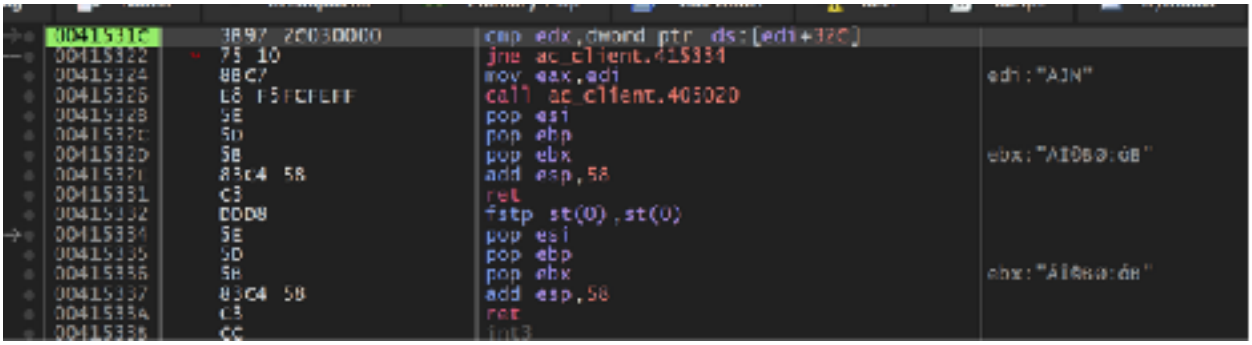
We can assume a few things about the code we are looking for, based on the pseudo code we described above:

1. It will have a **cmp** (or **test**) instruction followed by a conditional jump (**je**, **jne**, **jg**, etc.).
2. It will either call a function or have a fair amount of code, due to the many operations involved.
3. It will most likely make use of the floating point operations (**fld**, **fstp**) to position the icon on the radar.

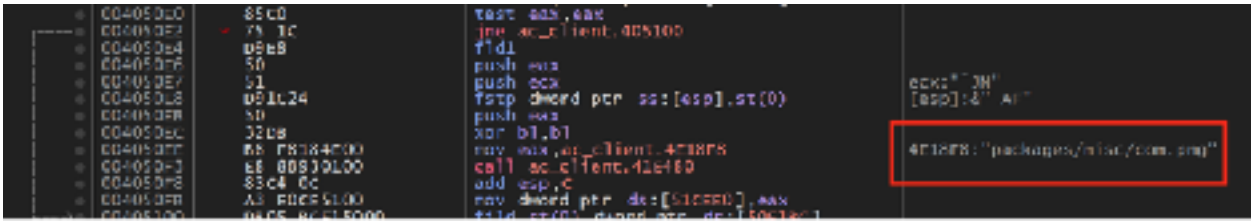
We can use these features to help figure out which location we care about in the code.

Because we are interrupting program execution, the breakpoints will not pop in a consistent manner. In this chapter, we will examine each piece of code in the order they were encountered when writing the chapter. In your environment, the order of pops will most likely be different.

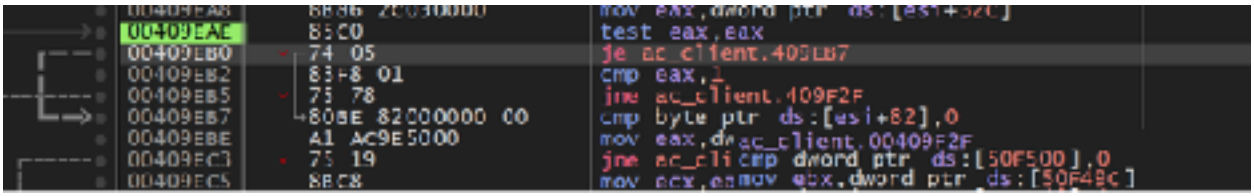
The first pop occurs at the following code:



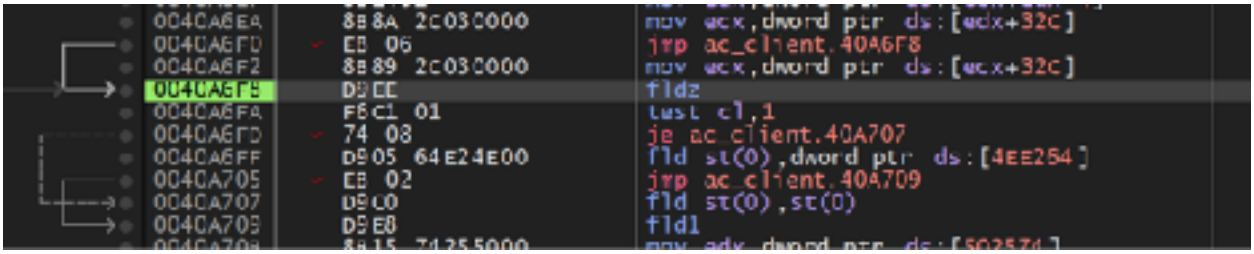
This initially looks like it checks off several of the conditions we care about. However, if we **nop** the **jne** instruction at `0x415322`, we notice that there is no change in the game. If you explore the **call** at `0x415326`, you should see the following code:



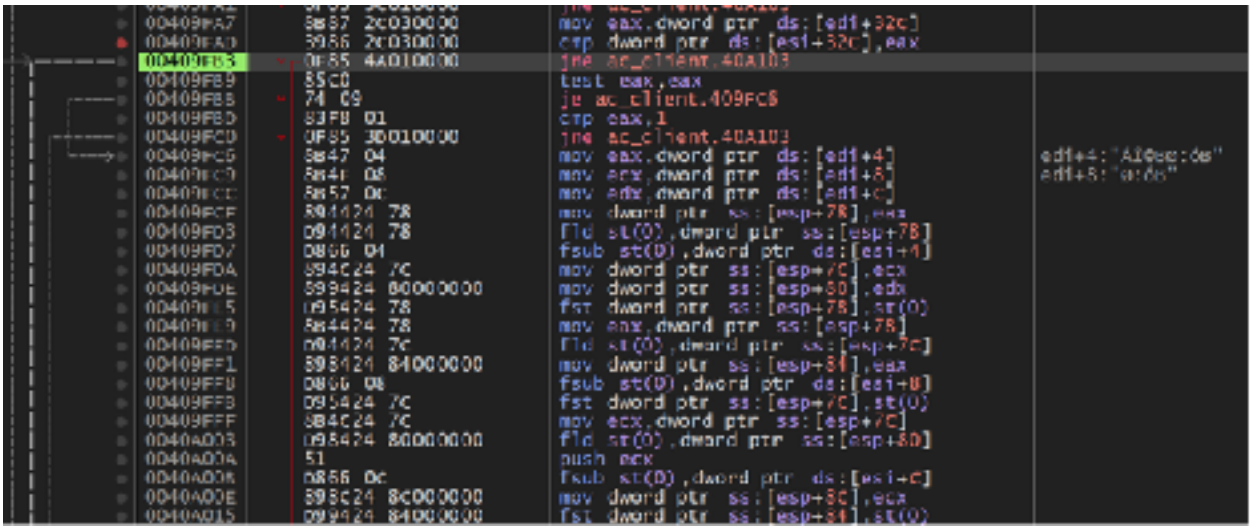
From this string constant, we can assume that this code has something to do with drawing the voice chat (or communication) symbols on the radar. Now we will continue on to the next location:



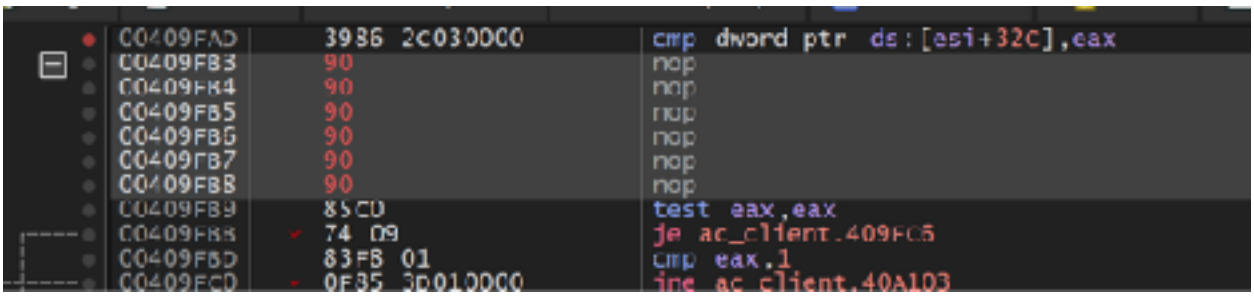
For our immediate reversing purpose, testing a register against itself is the same as comparing the register to 0. Here, we see that the code executes a branch if the player's team is set to 0, or the CLA team. The radar drawing operation should execute according to the value stored in the player's structure, not a static value. Now we can move on to the next location:



Examining this code, we see that it is doing an operation similar to the previous code. After loading in the value of the player's team to **ecx**, the code compares this value to 1, or the RSVF team with the **test cl, 1** instruction. The same logic applies here as it does in the paragraph above, so let's examine the next location:



Like our first location, this looks like a promising candidate. The **cmp** instruction at the top compares our current player's team against **eax**, which appears to be loading the same team offset from another data structure, potentially another player. We also see several floating point operations that may be responsible for placing the icon on the radar. Let's see what happens if we **nop** out the **jne** instruction:



If you go back into Assault Cube, you should notice that you can now see every player on the radar, including the ones that are not on your team. We have found our responsible radar code.

5.8.6 Changing the Code

Since this hack only requires us to write bytes to a memory address, we can use the same technique as discussed in Chapter 5.7:

```
#include <Windows.h>

unsigned char new_bytes[3] = { 0x90, 0x90, 0x90, 0x90, 0x90 };
```

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
	DWORD old_protect;
	unsigned char* hook_location = (unsigned char*)0x409FB3;

	if (fdwReason == DLL_PROCESS_ATTACH) {
		VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
		for (int i = 0; i < sizeof(new_bytes); i++) {
			*(hook_location + i) = new_bytes[i];
		}
	}

	return true;
}
```

5.9 ESP

5.9.1 Target

Our target for this chapter will be Assault Cube 1.2.0.2.

5.9.2 Identify

Our goal in this chapter is to create an ESP hack, a type of hack that displays player information above their heads. This information includes the player's health, name, or current weapon in use, and it is also displayed through walls.

5.9.3 Understand

In Chapter 5.6, we created an aimbot, which worked by calculating the angle between our player and an enemy and then setting our player's current view angle to that calculated angle. For games where a camera is always bound to our player's view, such as an FPS, we can use these same angles to create an ESP.

Instead of setting our player's view angle, we will use the difference in these angles to convert the enemy's 3D location in the world to a 2D position relative to our player's view. We will then draw text at this position.

The method discussed in this chapter has several drawbacks, but it demonstrates the basic concepts used in an ESP. A more accurate approach is to use the game's viewmatrix.

5.9.4 Viewports

Take the following scene from Assault Cube:

Playrix Ex. 1024, page 266



We know from the previous chapter that the enemy we see in the scene above has a 3D position in the world represented by X, Y, and Z coordinates. However, when we are playing Assault Cube, the game needs to display this enemy on a monitor, which is two-dimensional. To do this, the game will choose a static view of the world, called a viewport. In this case, the viewport is tied to the player model that we are controlling. The game will then use this viewport to determine where 3D objects in the world should be displayed.

A good way to visualize a viewport is to imagine a movie set. When filming a movie, the set has actors, sound fixtures, lighting fixtures, and people and fixtures responsible for practical effects. However, none of this extra information is shown when you watch the movie, as the only view of this 3D world (the set) that you can access is the camera filming a specific section. In this analogy, the camera is acting as your viewport into the movie set's 3D world.

Playrix Ex. 1024, page 267

By moving around in the world, we are adjusting our viewport's position. For example, by moving to the right of the position shown in the scene above, we have the following scene:



We can see that the enemy has not moved, but his model is now being displayed on the left side of our screen. This is because when we moved our player, we also moved our viewport into this world to a different position.

5.9.5 World to Screen

Like we did when developing our aimbot, we will simplify our ESP development by first isolating the X (or left and right) value. After we have figured out how to calculate this value, we can add our Y (up and down value). We will also develop our hack for a single enemy and then add support for multiple enemies.

In this chapter, we will assume that you are running Assault Cube in a window of 1024x768. This means that our window is 1024 pixels wide and 768 pixels high. Depending on where your viewport is in the world, the enemy will appear at certain pixel values when the scene is rendered. For example, take the following scene where we are looking at the enemy:



In this case, the enemy is in the middle of our screen, or 1024 / 2. This means that the enemy is at (roughly) the 512th pixel. When we move our player left, the enemy will now appear on the far right of our screen:



It is hard to identify the exact pixel that the enemy is at here, but we can assume it is roughly 1000. Likewise, if we move right, the enemy will appear on the far left of our screen:



Here the enemy starts at roughly 100 pixels.

We can represent these different scenarios in a series of equations. Since the default view has the middle of our viewport lining up to the middle of our screen, we want to find a value S such that all these equations below will be satisfied:

```
512 = 512 + S
100 = 512 + S
1000 = 512 + S
```

There is no constant value of S that will make all these equations true. We need a way for S to be both negative and positive and represent values from roughly -400 to 0 to 400. To achieve this, we can expand S out into a multiplication of two values, as shown below:

```
512 = 512 + (A * F)
100 = 512 + (A * F)
1000 = 512 + (A * F)
```

In these equations, A will be tied to how far the enemy is from our viewport's center, and F will be a static scaling value. If we are looking directly at the enemy, A will be 0, making our first equation true. If we are looking to the right of the enemy, (A * F) will produce a negative value to subtract from 512. Likewise, if we are looking to the left, (A * F) will produce a positive value to add to 512.

5.9.6 Scaling Values

Next, we need to determine the values of A and F. When writing our aimbot, we determined our current player's yaw as well as the yaw needed to aim at an enemy. In that case, we then set our current player's yaw to the latter yaw. However, for this hack, we can use the difference between these values as a value for A above. The larger the difference between these values is, the farther away the enemy is from the center of our screen.

We can use our aimbot code to determine what these values look like in the game. In our aimbot code, we calculated the yaw via:

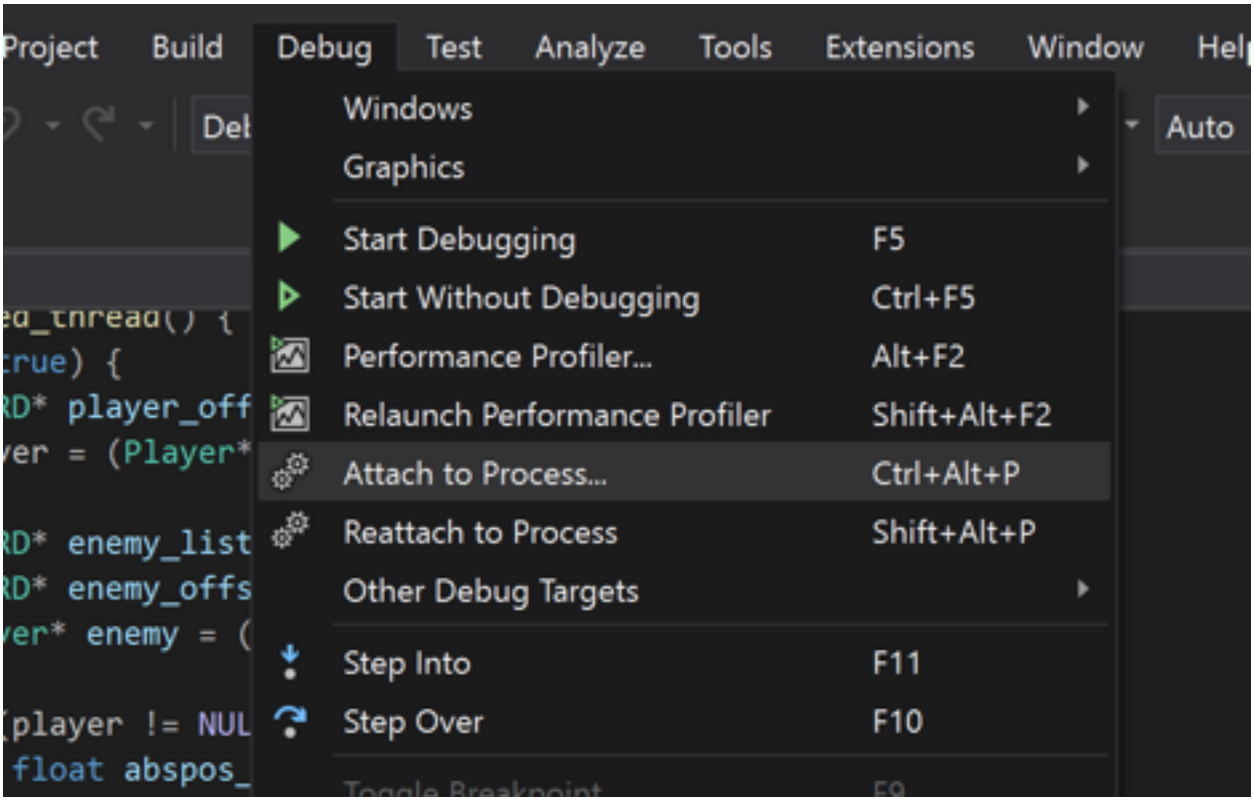
```
float abspos_x = enemy->x - player->x;
float abspos_y = enemy->y - player->y;
```

```
float azimuth_xy = atan2f(abspos_y, abspos_x);
float yaw = (float)(azimuth_xy * (180.0 / M_PI));
yaw += 90;
```

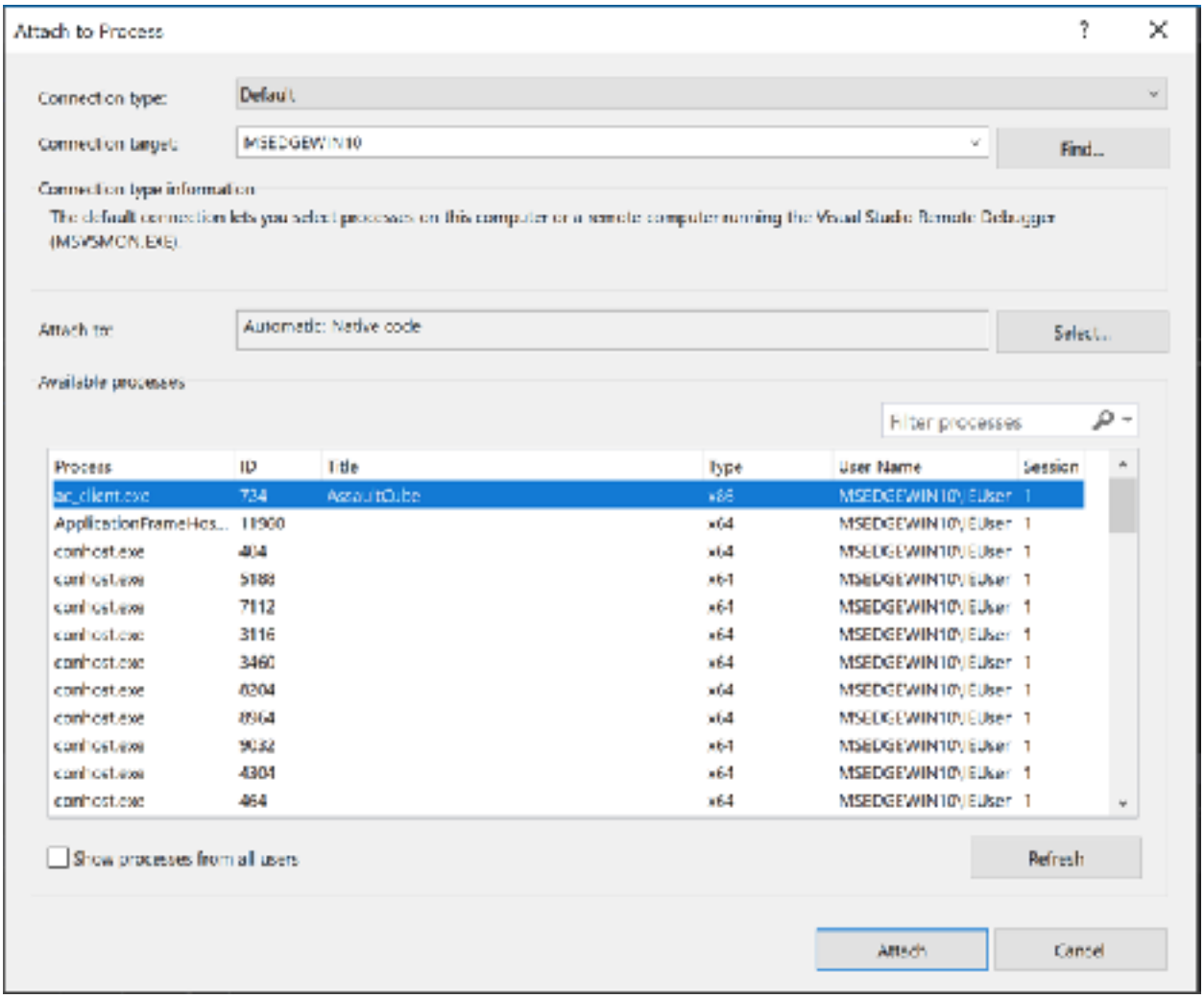
Unlike in the aimbot, where we set **player->yaw = yaw**, we will instead calculate the difference between these yaws:

```
float yaw_dif = player->yaw - yaw;
```

We can use Visual Studio's built-in debugger to see what this value is. First, build your DLL as normal. Then, open up Assault Cube, create a two-player game just like we did in Chapter 5.6, and inject the DLL with a DLL injector. With our DLL injected, go into Visual Studio and choose *Debug -> Attach to Process*:

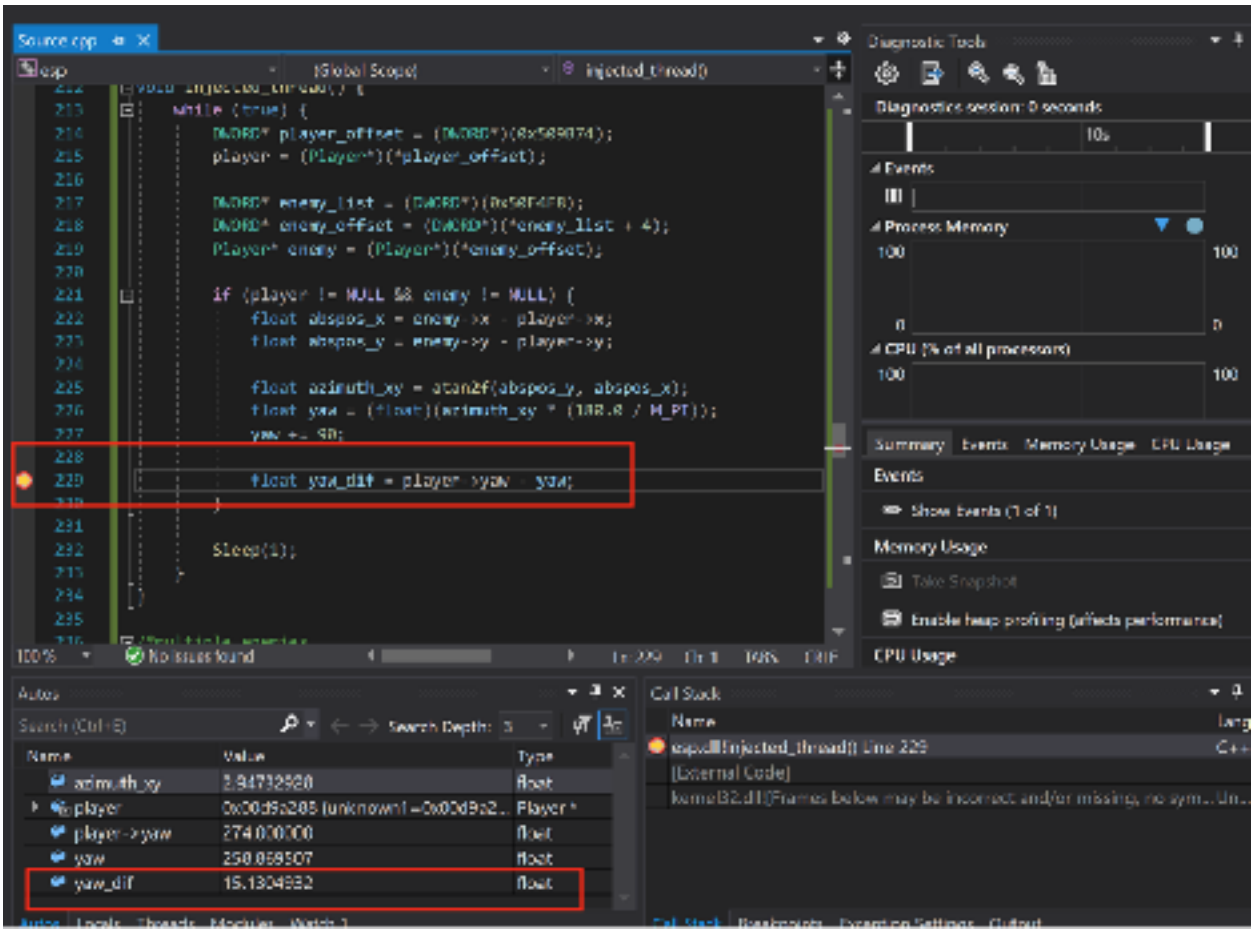


Next, choose the Assault Cube process, *ac_client*:



With this done, you can set breakpoints on your DLL code in an identical manner to how we did previously for regular executable code. If you put a breakpoint on the line assigning **yaw_dif**, you can see its value in the *Autos* window at the bottom of Visual Studio:

Playrix Ex. 1024, page 273



With this set up, we can now get our **yaw_dif** values. Repeat the same scenarios that we discussed above (looking at, far left, and far right) and get the corresponding **yaw_dif** values for each:

```
Looking at enemy:
      yaw_dif     -0.307769775

Enemy on far left of screen:
      yaw_dif     34.9015427

Enemy on far right of screen:
      yaw_dif     -39.5185280
```

Depending on where you stand in the map, your values may be different. For the sake of this chapter, we will use the values above for our equations. Let's plug these values into our equations as the value for A:

```
Looking at enemy:
      512 = 512 + (-0.307769775 * F)

Enemy on far left of screen:
      <100 = 512 + (34.9015427 * F)

Enemy on far right of screen:
      >950 = 512 + (-39.5185280 * F)
```

Since we are roughly estimating, we will round these values to the closest whole number:

```
Looking at enemy:
      512 = 512 + (0 * F)

Enemy on far left of screen:
      100 = 512 + (35 * F)

Enemy on far right of screen:
      1000 = 512 + (-40 * F)
```

We can see that **yaw_dif** will satisfy our first equation regardless of the value we choose for F. Using some basic algebra, we can solve for F using the far left and far right equations:

```
Enemy on far left of screen:
      F = -11.771428571

Enemy on far right of screen:
      F = -12.2
```

Since these were approximations, we will take the loose average and choose -12 as our value for F. Our initial equation to convert an enemy's position to a 2D screen coordinate for the X dimension is:

```
screen_x = 512 + (yaw_dif * -12)
```

We will have to make adjustments to this equation, but it gives us a good starting point.

Playrix Ex. 1024, page 275

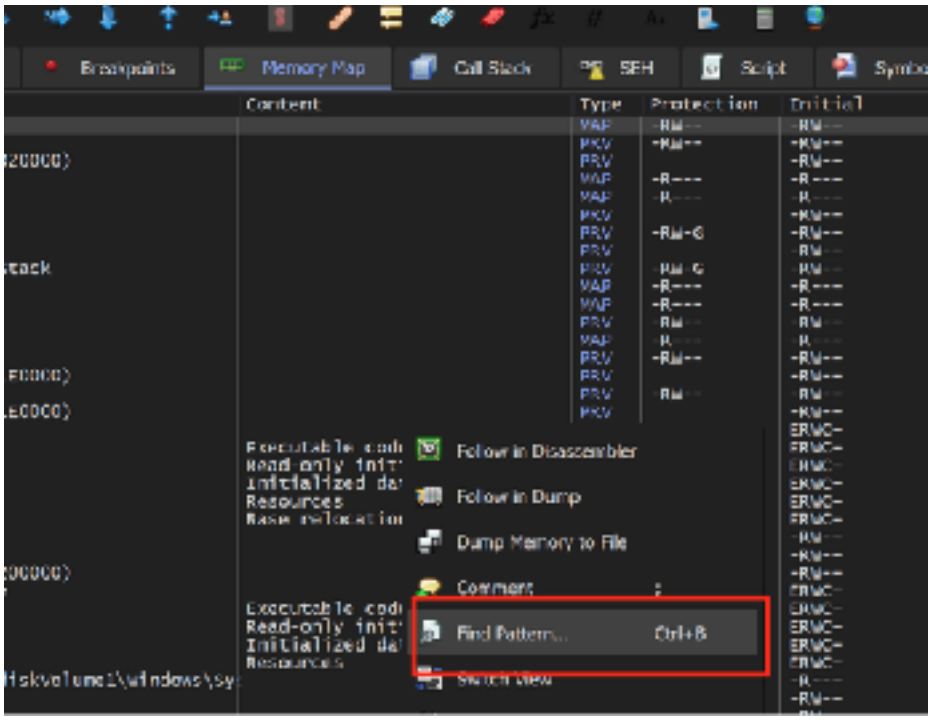
5.9.7 Locating Print Text

To continue testing our equation, we need to find a way to print text on the screen. We talked about how to find and hook a text printing function in Chapter 3.5. We will use a similar approach here.

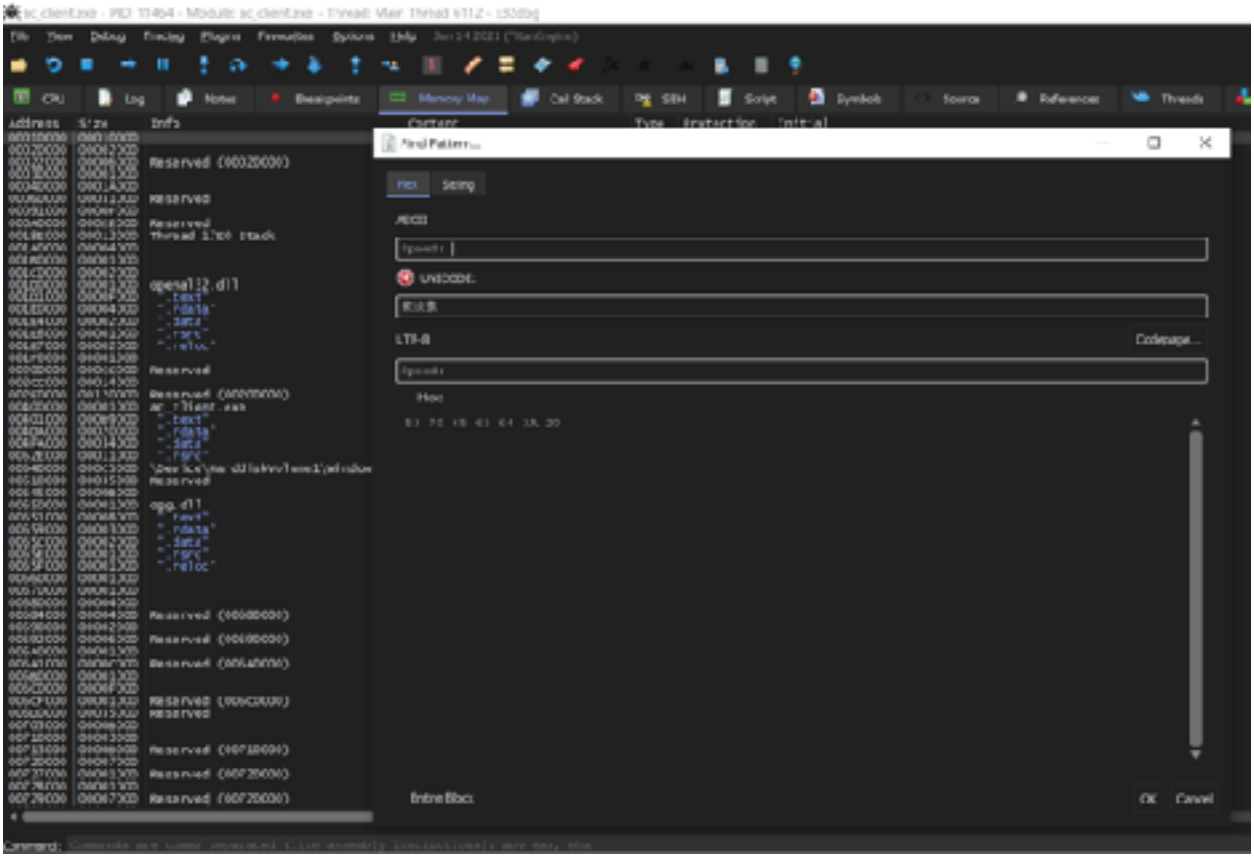
We want to identify some text that looks like it can be easily displayed anywhere on the screen. After investigating some of the documentation, we can determine that the *showspeed* command text is a good candidate, as it displays in the exact middle of the screen:



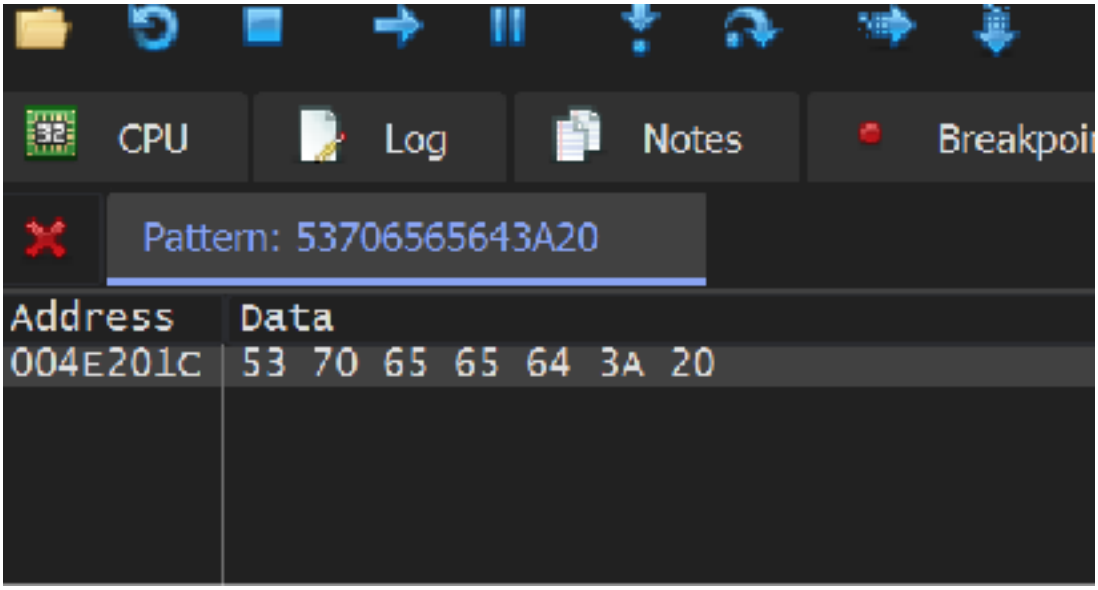
Since this text is static, we can search for its pattern in x64dbg and find where it resides in memory. Attach x64dbg to Assault Cube, then navigate to the *Memory Map* tab and right-click. Choose *Find Pattern*:



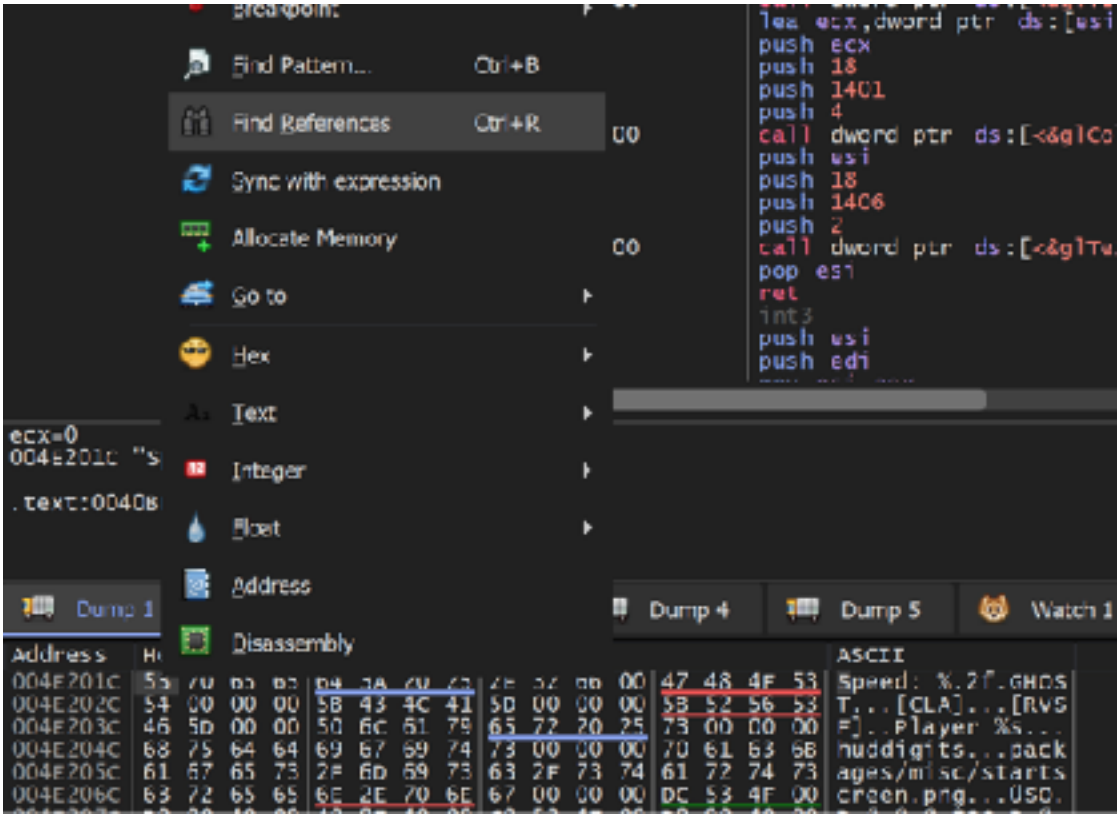
This will allow us to search all the active memory in the game for whatever pattern of bytes we specify. In this case, we will search for the start of this text, *Speed:* :



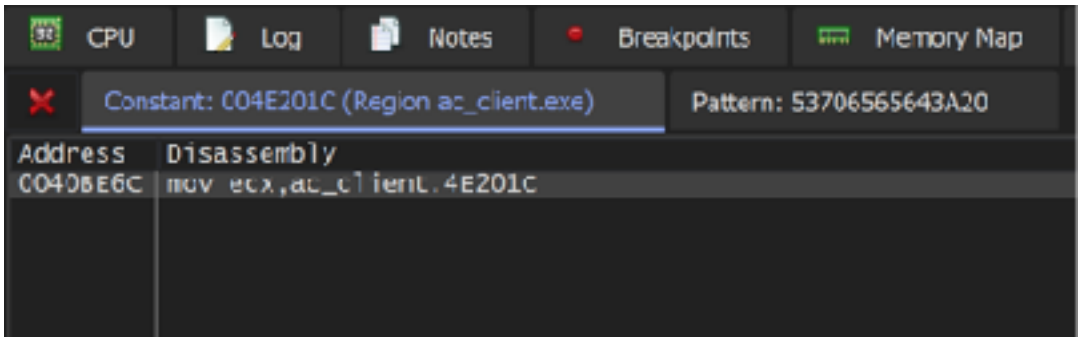
The search should find a single pattern:



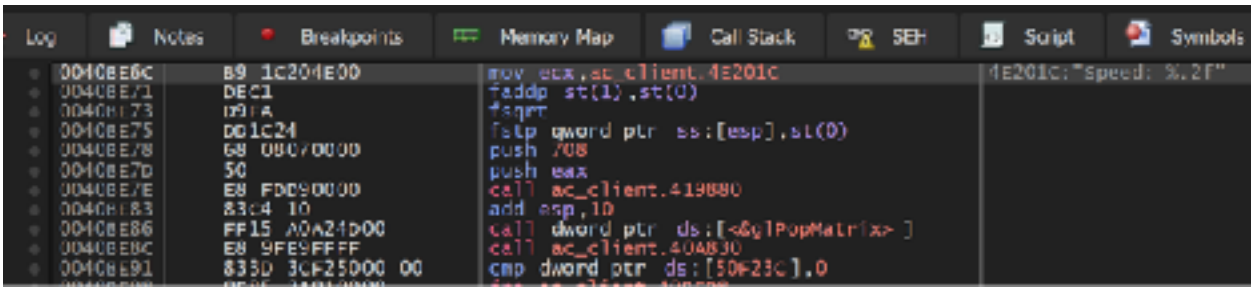
Double-click on it to show the address in the dump. Next, we want to see where this memory address is referenced in code. Select the first letter and right-click. Choose *Find references*:



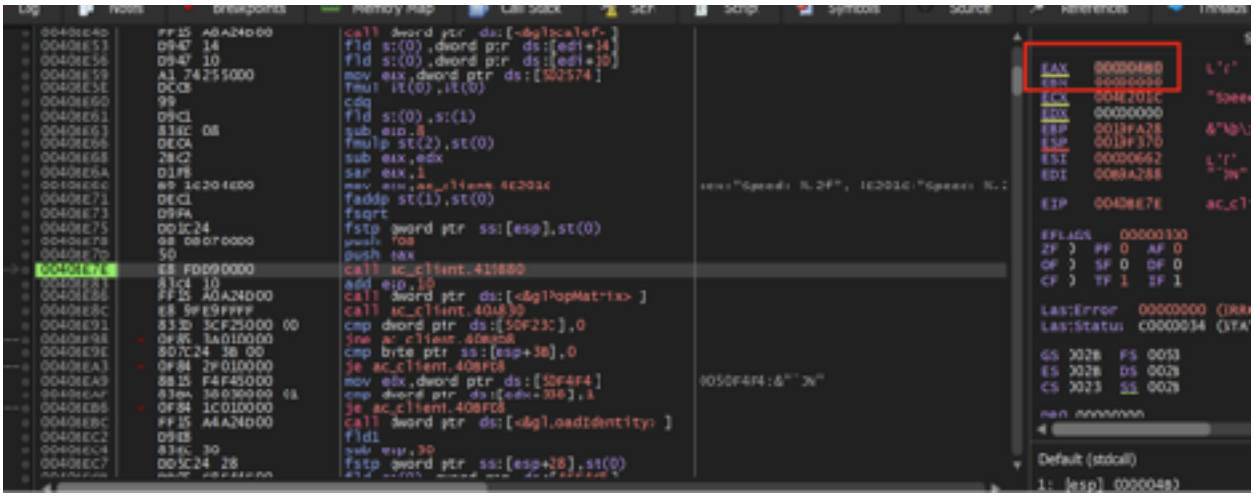
This will return a single reference, or place where this address was referenced by the code of the game:



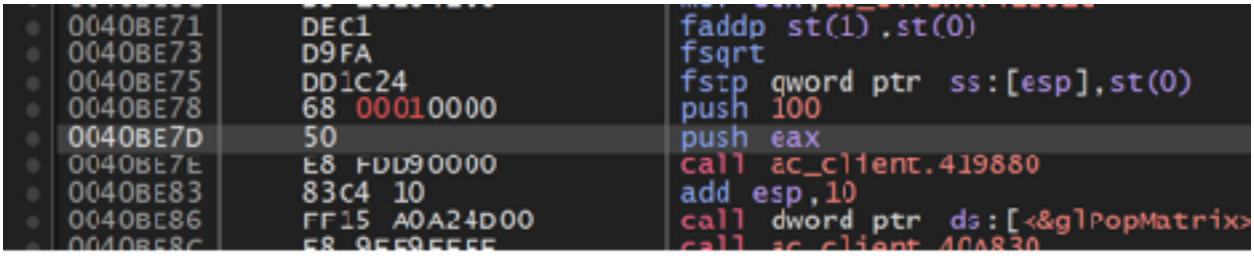
Double-click again on this reference to be brought to the code responsible for accessing this memory:



Quickly analyzing this code, we see that we move a text string into **ecx** (in this case, our *Speed:* string) and then push two values on the stack before calling `0x419880`. We can see one value is `0x708`, or 1800 decimal. If we set a breakpoint on the **call**, we can see the value of **eax**:



`0x4B0`, or 1200, and 1800 seem like reasonable X and Y values. Combined with a text string in **ecx**, this function is most likely responsible for printing the *Speed:* text. We can verify this behavior by modifying the **push 0x708** to another value, like **push 0x100**:



Upon doing this, our *Speed:* text will appear near the top of the screen:



From this, we can see that the method responsible for printing text expects Y to be pushed first, followed by X.

5.9.8 Print Text Code Cave

To nail down our equation, we will use a code cave that will modify the *showspeed* print text call to draw our enemy text. We will hook at the first **push** so we can push our desired values on the stack:

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    DWORD old_protect;
      unsigned char* hook_location = (unsigned char*)0x0040BE78;

    if (fdwReason == DLL_PROCESS_ATTACH) {
            CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread,
NULL, 0, NULL);

            VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
            *hook_location = 0xE9;
            *(DWORD*)(hook_location + 1) = (DWORD)&codecave -
((DWORD)hook_location + 5);
            *(hook_location + 5) = 0x90;
    …
```

Our code cave itself will push our currently assigned X and Y values on the stack, as well as move some generic **Enemy** text into **ecx**. After we do this, we will jump back to the original call to have it print out our text:

```
DWORD ret_address = 0x0040BE7E;
const char *text = "Enemy";

DWORD x = 0;
DWORD y = 0;

__declspec(naked) void codecave() {
    __asm {
        mov ecx, text
        push y
        push x
        jmp ret_address
    }
}
```

We can verify that this code is working by setting our X and Y to `0x100` in our thread, after we calculate the **yaw_dif**:

```
x = 0x100;
y = 0x100;
```

If we go into a game and show the speed, you will see our text appearing in the upper-left corner of the screen:



For now, we can use this to nail down our ESP. We will come back later and adjust this approach so that we can write multiple text strings for multiple players.

5.9.9 Refining Equation

With a text function, we can now start working on the ESP. However, we first need to adjust our equations. When we initially modeled the screen, we assumed that center would be 512. However, from the speed function, we saw that center was `0x4B0`, or 1200. Games will often make use of a "virtual" screen that will always be an identical size regardless of the resolution. That way, developers only have to convert the resolution into the virtual screen size once, but they can use consistent coordinates in the rest of the code.

In this case, it looks like the game's virtual screen is 2400x1800. We can go back to our original equations and update them with these new numbers:

```
Looking at enemy:
      1200 = 1200 + (0 * F)

Enemy on far left of screen:
      100 = 1200 + (35 * F)

Enemy on far right of screen:
      2400 = 1200 + (-40 * F)
```

Calculating with these new values, we will get a different value for F:

```
Enemy on far left of screen:
    F = -31.428571429

Enemy on far right of screen:
    F = -30
```

Since these are again approximations, we will choose a value of -30 as our value for F, making our new equation:

```
screen_x = 1200 + (yaw_dif * -30)
```

We can implement this equation in our main thread like so:

```
float yaw_dif = player->yaw - yaw;
```

```
x = (DWORD)(1200 + (yaw_dif * -30));
y = 0x200;
```

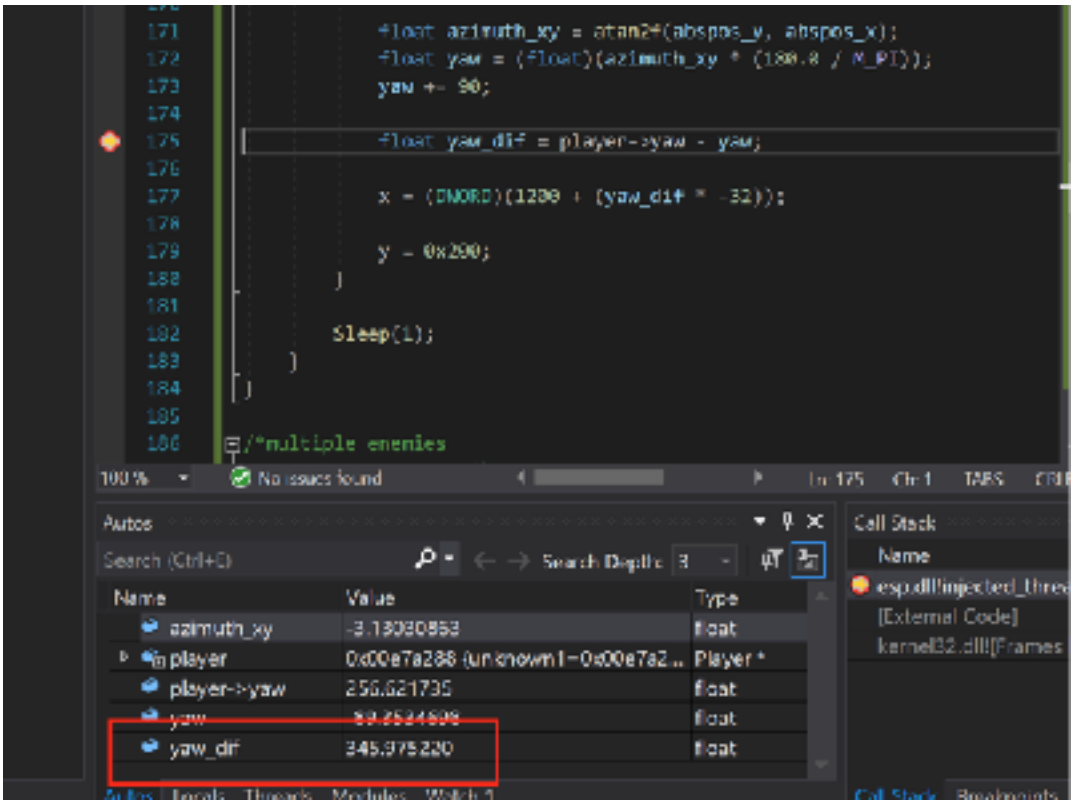
If you go into a game, the *Enemy* text will appear on the same X axis as the enemy from certain angles, as we expect:



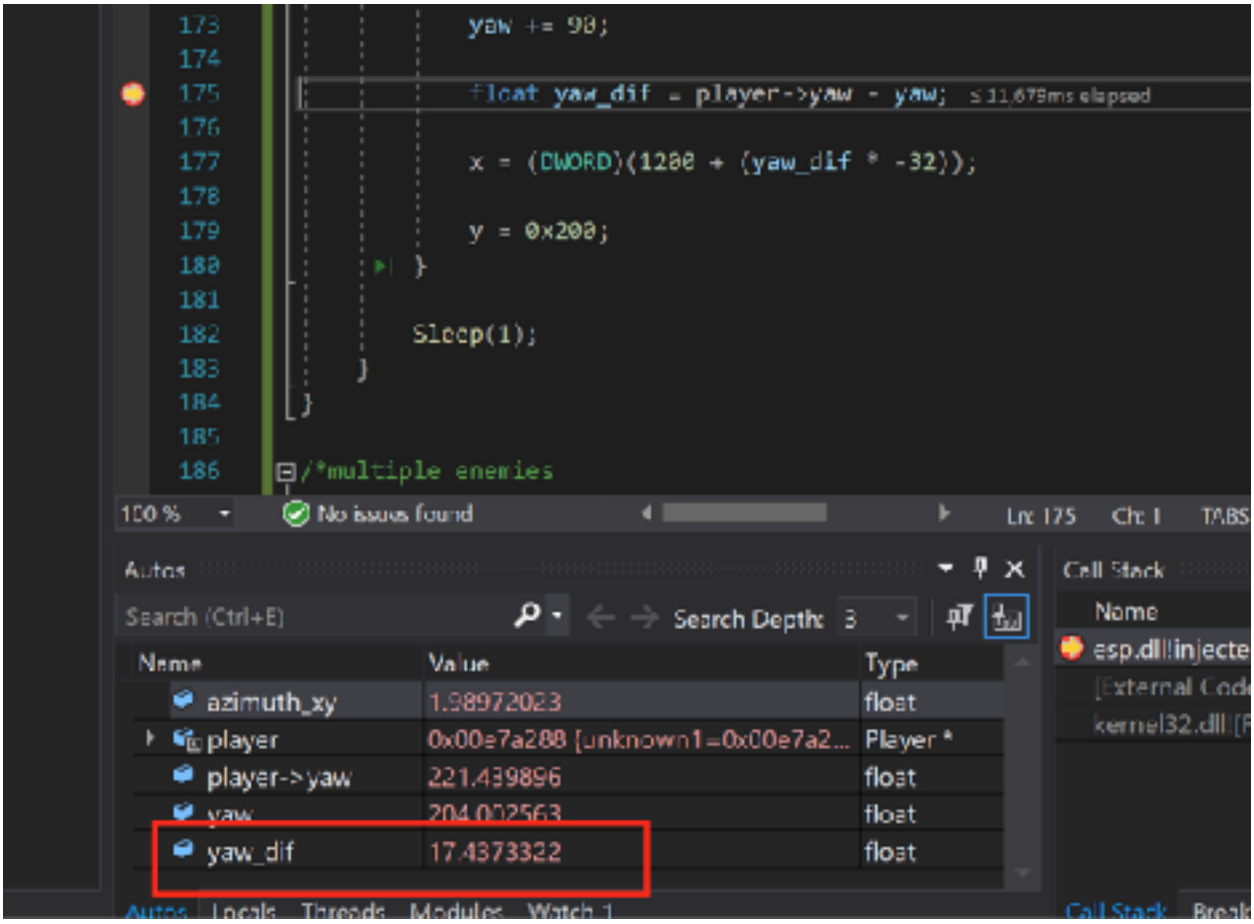
However, depending on what angle we are looking at, the text will appear far off to the side:



If we attach a debugger, we can see that when the text is not correct, the **yaw_dif** value is over 180:

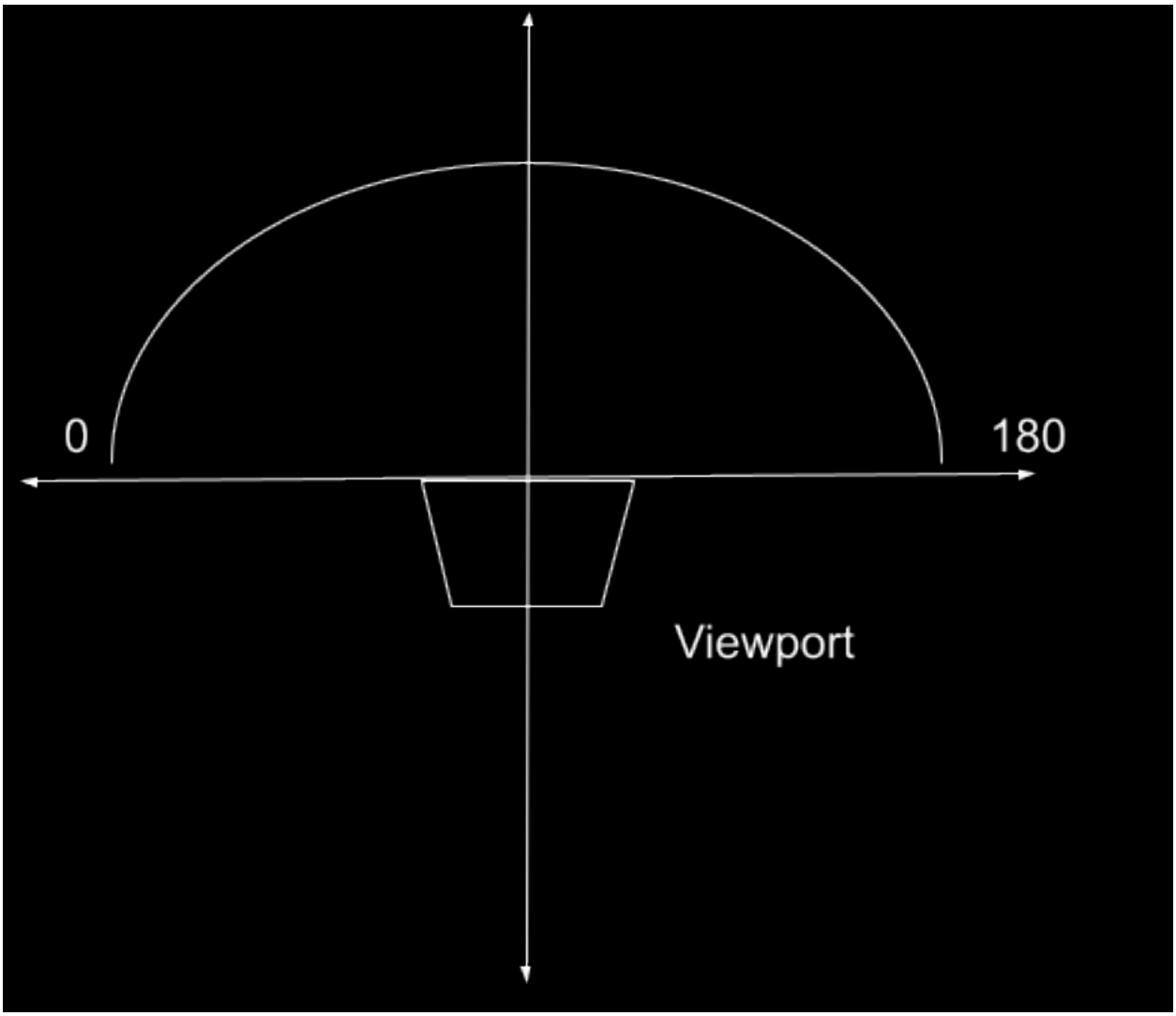


Just as a contrast, we can see that correct text values always have a **yaw_dif** value under 180:



This situation appears to occur when our player's yaw and the calculated yaw for the enemy are between 275 and 360/0. When subtracting to get our difference, the equations produce artificially high values that do not work with our scaling factor. For example, if our yaw difference is 5, our text will be correctly displayed. Likewise, if our difference is -5, the text will be displayed correctly in the opposite direction. However, if our difference is -355, the text will be displayed incorrectly, as the equation's result will now be 11,850, causing the text to wrap over to the other side of the screen.

Regardless of the viewport we choose, our viewport can never show more than 180 degrees of the screen. Any more would result in us seeing behind our player.



To fix the case of -355 that we described above, we can subtract (or add, in the case of negative) 360:

```
if (yaw_dif > 180)
      yaw_dif = yaw_dif - 360;

if (yaw_dif < -180)
      yaw_dif = yaw_dif + 360;

x = (DWORD)(1200 + (yaw_dif * -30));
```

With this in place, our text will always correctly display, regardless of the angle.

Playrix Ex. 1024, page 287

5.9.10 Up and Down

To calculate the Y dimension for our aimbot, we had the following code:

```
float abspos_z = enemy->z - player->z;
            ...
if (abspos_y < 0) {
    abspos_y *= -1;
}
if (abspos_y < 5) {
    if (abspos_x < 0) {
        abspos_x *= -1;
    }
    abspos_y = abspos_x;
}
float azimuth_z = atan2f(abspos_z, abspos_y);
float pitch = (float)(azimuth_z * (180.0 / M_PI));
```

We can use the same approach as above to calculate our Y dimension. We learned from the speed function that `0x708`, or 1800, was the bottom of the virtual screen. We can perform the same series of equations as above to get the scaling factor for Y:

```
looking above the enemy (enemy at 1800)
         pitch_dif      25.4983654
        F = 35.4
looking at enemy, 1800/2
         pitch_dif      -4.36527729
        F = 0
looking below the enemy (enemy at 100)
         pitch_dif      -41.1258888
        F = 19.464720195
```

From these values, we will choose a value of 25 as our scaling factor:

```
float pitch_dif = player->pitch - pitch;
y = (DWORD)(900 + ((pitch_dif) * 25));
```

With this in place, text will now correctly display in the Y axis:



5.9.11 Final Adjustments

Right now, our text will always display, even if the enemy is behind us. To prevent this, we will add a check into our print text code cave. In this check, we will set our string to empty if the enemy is not visible in our viewport:

```
__declspec(naked) void codecave() {
    if (x > 2400 || x < 0 || y < 0 || y > 1800) {
        text = "";
    }
    else {
        text = "Enemy";
    }
```

We can also resolve the issue of text always appearing slightly to the left of the enemy. For this, we will simply always add 200 to whatever X value we calculated:

```
else {
    text = "Enemy";
}

x += 200;
```

Our text is now displaying correctly for a single enemy.



5.9.12 Enemy Name

With our coordinates nailed down, we can work on getting the enemy's name to display above their head. Like before, we can find the player's name in the game's Player structure and add it to our code's Player structure. Looking at the player structure in memory, we see the player's name a bit after the yaw element we identified before:



If we try to change our own player's name, we find that this can be a maximum of 16 characters. Subtracting the offset from our yaw value, we can create padding with an unknown element, like we did in Chapter 5.6. Our player structure with the name now looks like:

```
struct Player {
	char unknown1[4];
	float x;
	float y;
	float z;
	char unknown2[0x30];
	float yaw;
	float pitch;
	char unknown3[0x1DD];
	char name[16];
};
```

We can now modify our code to use this enemy name instead of the generic **Enemy** text. First, remove the **else** condition in the code cave which set the text to **Enemy**:

Playrix Ex. 1024, page 291

```
const char* text = "";
...
__declspec(naked) void codecave() {
      if (x > 2400 || x < 0 || y < 0 || y > 1800) {
            text = "";
      }

    x += 200;
    …
```

Next, after we calculate our X and Y, we want to set the text member to the enemy's name by assigning the pointer:

```
text = enemy->name;
```

With this change, enemy names will now appear above their head:



In this chapter, we will only display the enemy's name. A similar approach can be used to display the enemy's weapon, health, and other information.

5.9.13 Multiple Enemies

Now that we have a working ESP for a single enemy, we can expand it to include multiple enemies. Like we did when creating our aimbot, we can use the same code we nailed down above and include it in a loop.

First, instead of one X, Y, and name value, we will create an array. The maximum amount of players in an Assault Cube game is 32, so we will use this as the size of our array. For our loops, we will use the current number of players we identified previously, so any extra array elements will not cause an issue. Since we will need to use this current player element in both our calculation loop and draw loop, we will create a global variable for it as well:

```
#define MAX_PLAYERS 32

DWORD x_values[MAX_PLAYERS] = { 0 };
DWORD y_values[MAX_PLAYERS] = { 0 };
char* names[MAX_PLAYERS] = { NULL };

int* current_players;
```

Next, we will modify our thread to iterate over all enemies in an identical manner to the aimbot. When we calculate the X and Y values, we will store these values in an array instead of a single element:

```
current_players = (int*)(0x50F500);

for (int i = 1; i < *current_players; i++) {
    DWORD* enemy_list = (DWORD*)(0x50F4F8);
    DWORD* enemy_offset = (DWORD*)(*enemy_list + (i*4));

    ...
    x_values[i] = (DWORD)(1200 + (yaw_dif * -30));
    ...
    y_values[i] = (DWORD)(center_y + ((pitch_dif) * 25));
    ...
    names[i] = enemy->name;
```

Finally, we need to redo our text printing function so that we can print multiple enemy names. We will use the same location, but instead of replacing the pushed parameters, we will hook the call itself. In our code cave, we will replace the call with empty text, and then create a loop to call the print text function several times.

First, we will change our hook location to hook the call:

```
unsigned char* hook_location = (unsigned char*)0x0040BE7E;

if (fdwReason == DLL_PROCESS_ATTACH) {
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread, NULL, 0,
NULL);

    VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
    *hook_location = 0xE9;
    *(DWORD*)(hook_location + 1) = (DWORD)&codecave - ((DWORD)hook_location +
5);
}
```

We can then delete our previous code cave and create a new one. In it, we will first replace **ecx** with empty text, then call the print text function, and then save and restore everything as we have done before:

```
DWORD ret_address = 0x0040BE83;
DWORD text_address = 0x419880;

const char* empty_text = "";

__declspec(naked) void codecave() {

      __asm {
            mov ecx, empty_text
            call text_address
            pushad
      }

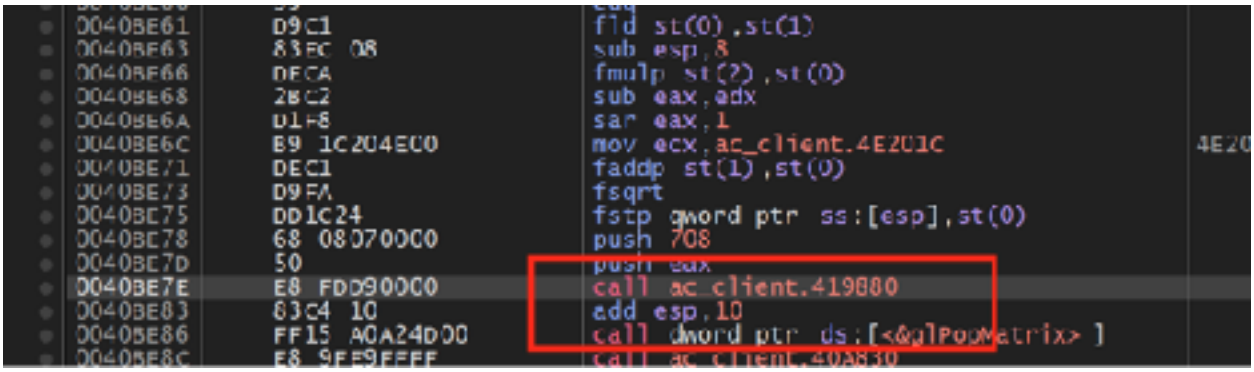
    //loop

    __asm {
            popad
            jmp ret_address
```

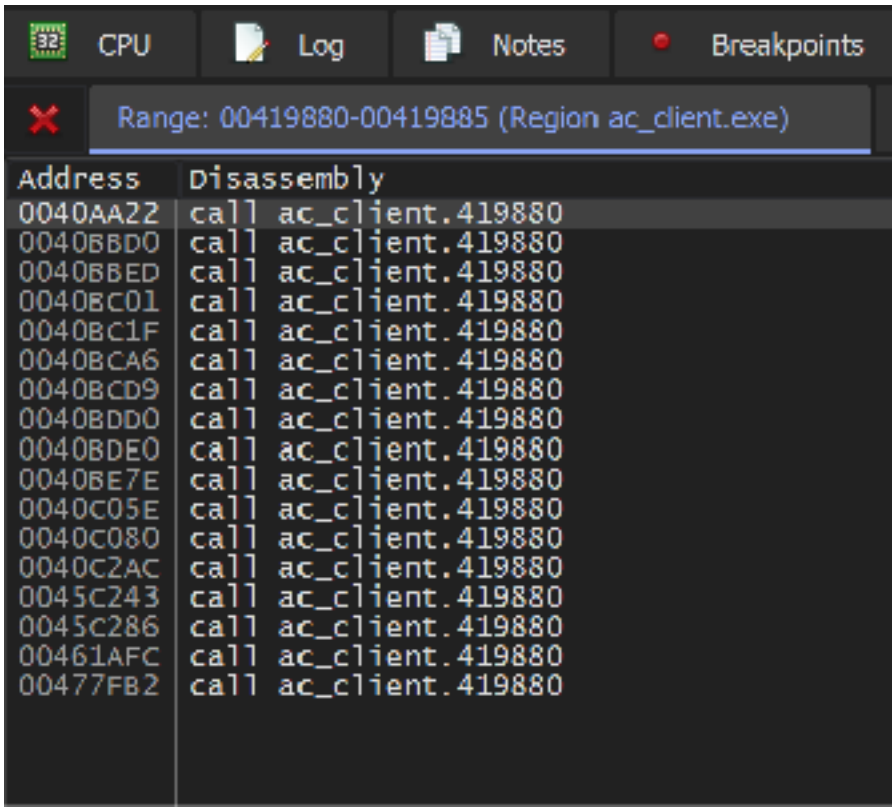

Playrix Ex. 1024, page 294

```
        }
}
```

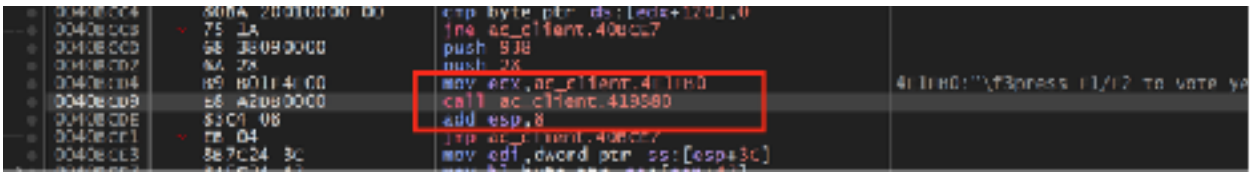
To call the print text function by ourselves, we need to figure out how to fix the stack. Remember the code we found:



When you see **add esp** or **sub esp** after a call, it means the called code expects you to balance the stack. The easiest way to determine how to balance the stack is to find the smallest value of **esp** being modified and use that call as a basis. This can be done by entering the print text call and finding the references:



After going through several of these references, you should find the following reference, which shows that **add esp, 8** is the lowest required value to balance the stack:



With this information, we can invoke the print text call ourselves. First, we will create a loop that will iterate over all the current players in the game:

```
current_players = (int*)(0x50F500);

for (int i = 1; i < *current_players; i++) {
```

To make our array of values easy to use inside the **asm** block, we will copy the current value into temporary variables:

```
const char* text = "";

DWORD x = 0;
DWORD y = 0;
...
for ...
    x = x_values[i];
    y = y_values[i];
    text = names[i];
```

Like before, we will then check the values to make sure that they can be displayed on screen:

```
if (x > 2400 || x < 0 || y < 0 || y > 1800) {
    text = "";
}
```

Finally, we will move the enemy's name into **ecx**, push our X and Y values, and call the print text function. After that, we will balance the stack:

```
__asm {
```

```
    mov ecx, text
    push y
    push x
    call text_address
    add esp, 8
}
```

With this in place, we now have all the parts we need to handle multiple enemies. If you build and inject the DLL into Assault Cube, you will see multiple enemy names appearing above their heads:



The full code is available for reference in Appendix A.

Playrix Ex. 1024, page 297

5.10 Multihack

5.10.1 Target

Our target for this chapter will be Assault Cube 1.2.0.2.

5.10.2 Identify

In the previous chapters, we created several hacks for Assault Cube, including a triggerbot, an aimbot, and ESP. In this chapter, we will create a multihack that combines these hacks together along with a wallhack. Then, we will add an interactive menu that will allow us to toggle all the hacks we have created at once.

5.10.3 Understand

We have written code in the previous chapters for a wallhack, triggerbot, aimbot, and ESP. However, if we try to combine all this code together, it will quickly become overwhelming to add new features. To create our multihack, we will use a software development technique known as refactoring.

When refactoring code, you take existing code and alter its structure without changing its behavior. There can be many goals when refactoring, but in our case, our goal will be to encapsulate certain functionality into classes so that the code can be separated out into logical sections. This will clean up the code and make it easier to maintain. Once this is done, we will build off this refactored code to add our menu.

This chapter will involve working with a lot of code. Each separate stage of the code will be available in Appendix A for this chapter. The final code will be in the "Finished" section.

5.10.4 Wallhack

In Chapter 5.3, we covered an approach for making a wallhack for games that used OpenGL. We can use the same technique for Assault Cube with some small modifications.

Playrix Ex. 1024, page 298

In the original target game (Urban Terror), we needed to check for counts and re-enable depth testing if the model's count was not large enough. If we did not do this, every item would have depth testing disabled. However, in Assault Cube, the rendering logic works differently, and this check is not required. Additionally, Assault Cube does not require us to worry about clipping planes. As a result, our **glDrawElements** code cave can be simplified:

```
__declspec(naked) void opengl_codecave() {
	__asm {
		pushad
	}

	(*glDepthFunc)(0x207);

	// Finally, restore the original instruction and jump back
	__asm {
		popad
		mov esi, dword ptr ds : [esi + 0xA18]
		jmp opengl_ret_address
	}
}
```

By using the same hooking technique described in Chapter 5.3, we now have a working wallhack for Assault Cube.

5.10.5 Combining

This combined code we are covering in this section is available in the “Combined” section in Appendix A.

Our first task is to combine all of our code from the previous chapters into one DLL. Our multihack will contain the following hacks:

- OpenGL Wallhack
- Triggerbot
- Aimbot
- ESP

We can combine this code by copying it all into a single main file and changing any conflicting variable or function names (like all the versions of **injected_thread**). The result of this can be seen in Appendix A.

Looking over this code, the first thing that should jump out is that it is over 300 lines long with 20 global variables. In addition, we can see that we have two threads being created (one for hooking OpenGL and one for our aimbot) and multiple code caves.

With our combined code, we can make two small changes to slightly improve the size of the code. First, we can combine together the aimbot and ESP code, since they use a majority of the same logic. Second, we can modify the thread for OpenGL to **break** out of its **while** loop once it hooks **glDrawElements**:

```
if (openGLHandle != NULL && glDepthFunc == NULL) {
    glDepthFunc = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glDepthFunc");
    ...
    // Since OpenGL is loaded dynamically, we need to dynamically calculate
the return address
    opengl_ret_address = (DWORD)(opengl_hook_location + 0x6);
}
else {
    break;
}
```

This will effectively exit the thread once we hook OpenGL, to prevent our hack from having so many open threads.

In its current form, this code could be built and used as a multihack; however, it is almost impossible to maintain. If we want to add a menu and a method to toggle functionality, we would need to thoroughly examine all 300+ lines of code and make sure our toggles do not introduce any unexpected behavior across the many threads. Furthermore, we do not currently have a good way to print text outside of our ESP.

5.10.6 First Refactor

The source code we are covering in this section is available in the “First Refactor” section in Appendix A.

There are multiple approaches that can be used to simplify our code. For our purposes, we will encapsulate major functionality inside classes. Our end goal is to create classes that can be easily reused in other FPS games. We will then call those classes from the main file.

Classes in C++ commonly have two components: the header, which describes what the class contains and is included by the caller, and the source, which contains all the class's code. Therefore, we will split our multihack's code into *Header* and *Source* folders for all the following refactoring.

A good place to start is the triggerbot. In its most basic form, our triggerbot sends a mouse down event whenever we are looking at a player. To make this code reusable, we will structure the triggerbot class to require the main hack to provide information on if we are looking at a player.

Let's start with the current triggerbot code:

```
__declspec(naked) void triggerbot_codecave() {
	__asm {
		call triggerbot_ori_call_address
		pushad
		mov edi_value, eax
	}

	if (edi_value != 0) {
		input.type = INPUT_MOUSE;
		input.mi.dwFlags = MOUSEEVENTF_LEFTDOWN;
		SendInput(1, &input, sizeof(INPUT));
	}
	else {
		input.type = INPUT_MOUSE;
		input.mi.dwFlags = MOUSEEVENTF_LEFTUP;
		SendInput(1, &input, sizeof(INPUT));
	}

	_asm {
		popad
		jmp triggerbot_ori_jump_address
	}
}
```

In Assault Cube, the **edi** register holds whether a player is being looked at. However, in other games, this will be different. Therefore, it makes sense to only abstract out the code between the **__asm** blocks. We can replace this code with a call to our triggerbot class:

```
__declspec(naked) void triggerbot_codecave() {
	__asm {
```


Playrix Ex. 1024, page 301

```
            call triggerbot_ori_call_address
            pushad
            mov edi_value, eax
      }

      triggerbot->execute(edi_value);

      _asm {
            popad
            jmp triggerbot_ori_jump_address
      }
}
```

Now that we know how the calling code will look, we can create the class. First, we can create a header that will contain the definition of our triggerbot class:

```
#pragma once

#include <Windows.h>

class Triggerbot {
private:
      INPUT input = { 0 };
public:
      Triggerbot();
      void execute(int isLookingAtEnemy);
};
```

Classes have both private and public members. Public members can be accessed by other code. For example, we can see that the **execute** method will be called directly by our main file. However, the main file will not have access to the **input** variable.

To implement the code for our triggerbot class, we will create the source file next. This file will include the header we defined above, but will contain the actual code of the class:

```
#include <Windows.h>

#include "Triggerbot.h"

Triggerbot::Triggerbot() {
      input = { 0 };
```


Playrix Ex. 1024, page 302

```
}

void Triggerbot::execute(int isLookingAtEnemy) {
	if (isLookingAtEnemy != 0) {
		input.type = INPUT_MOUSE;
		input.mi.dwFlags = MOUSEEVENTF_LEFTDOWN;
		SendInput(1, &input, sizeof(INPUT));
	}
	else {
		input.type = INPUT_MOUSE;
		input.mi.dwFlags = MOUSEEVENTF_LEFTUP;
		SendInput(1, &input, sizeof(INPUT));
	}
}
```

This is the same as the original triggerbot code, except now encapsulated into this one class. We could add this class to a hack for another game and it would work, assuming that the main source file in that hack provided the correct value for **isLookingAtEnemy**.

To use this class in our main code, we will need to include the header and create an instance of it:

```
#include "Triggerbot.h"
...
Triggerbot *triggerbot;
...
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{

	if (fdwReason == DLL_PROCESS_ATTACH) {
		triggerbot = new Triggerbot();
	}
	else if (fdwReason == DLL_PROCESS_DETACH) {
		delete triggerbot;
	}
```

By structuring the code this way, our triggerbot code is now simplified and can be easily modified. For example, to toggle the triggerbot on and off, we could simply add a single conditional, like:

```
if(triggerbot_enabled) {
    triggerbot->execute(edi_value);
```


Playrix Ex. 1024, page 303

```
}
```

We can also use this opportunity to remove the **input** global variable from the main source file.

5.10.7 Finish Refactor

The code for this section is in the "Refactor Finished" section in Appendix A.

Another major component we would like to separate out is the code responsible for the aimbot and ESP. Looking at the code, we see that it is responsible for setting the following values:

```
x_values[i] = (DWORD)(1200 + (yaw_dif * -30));
y_values[i] = (DWORD)(900 + ((pitch_dif) * 25));
names[i] = enemy->name;

player->yaw = closest_yaw;
player->pitch = closest_pitch;
```

The X, Y, and name values are used for the ESP whereas the player's yaw and pitch are used for the aimbot. To calculate these values, our aimbot and ESP require the player's base address, the enemy list's base address, and the current number of players in the game.

To encapsulate this behavior in a class, we will separate the functionality into two functions. The first function will be responsible for calculating all the X, Y, and name values for the ESP, as well as the closest yaw and pitch. The second function will be responsible for setting the player's view to the calculated location. Separating these functions will allow us to easily toggle both the ESP and aimbot.

Since this class is responsible for player geometry and the player's relation to the world, we will call it **PlayerGeometry**. Like we did with the triggerbot, we can change our aimbot thread to:

```
void aimbot_thread() {
      while (true) {
        playerGeometry->update();
        playerGeometry->set_player_view();
    }
```


Playrix Ex. 1024, page 304

```
}
```

Since this code is now easily maintainable, we can combine the two threads in the main file:

```
void injected_thread() {

      while (true) {
        ...
            if (openGLHandle != NULL && glDepthFunc == NULL) {
            ...
            }

            playerGeometry->update();
            playerGeometry->set_player_view();

            Sleep(1);
      }
}
```

Our **PlayerGeometry** class will contain all player-relevant functions. To handle printing the ESP in our main file, the class will expose the array of X, Y, and name values:

```
class PlayerGeometry {
private:
      DWORD player_offset_address;
      DWORD enemy_list_address;
      DWORD current_players_address;

      float closest_yaw;
      float closest_pitch;

      Player* player;

      float euclidean_distance(float, float);
public:
      DWORD x_values[MAX_PLAYERS] = { 0 };
      DWORD y_values[MAX_PLAYERS] = { 0 };
      char* names[MAX_PLAYERS] = { NULL };

      int* current_players;

      PlayerGeometry(DWORD, DWORD, DWORD);
```

```
      void update();
      void set_player_view();
};
```

Unlike the triggerbot class, which just needed a parameter, this class requires the player's base address, the enemy list's base address, and the current number of players in the game. We will pass these in the constructor of the class, which is a special function that executes when the class is created:

```
PlayerGeometry::PlayerGeometry(DWORD p_address, DWORD e_address, DWORD
cp_address) {
      player_offset_address = p_address;
      enemy_list_address = e_address;
      current_players_address = cp_address;
}
```

We can then use these values in the class's code:

```
void PlayerGeometry::update() {
      DWORD* player_offset = (DWORD*)(player_offset_address);
      player = (Player*)(*player_offset);
    ... rest of aimbot and ESP code ...
}

void PlayerGeometry::set_player_view() {
      player->yaw = closest_yaw;
      player->pitch = closest_pitch;
}
```

When we create this class in our main file, we will pass these values. In this way, we can reuse the aimbot code in any game that has a similar memory layout:

```
playerGeometry = new PlayerGeometry(0x509B74, 0x50F4F8, 0x50F500);
```

We will also need to adjust the ESP code to use the values from this class:

```
for (int i = 1; i < *playerGeometry->current_players; i++) {
    x = playerGeometry->x_values[i];
    y = playerGeometry->y_values[i];
    text = playerGeometry->names[i];
```


Playrix Ex. 1024, page 306

Finally, we can move some variables that never change to a constants header, just to separate the variables out from the main file.

5.10.8 Adding a Menu

The code for the rest of this chapter is in the "Finished" section in Appendix A.

With our code refactored, we can add a menu. First, we will extract out the text printing functionality to its own function:

```
void print_text(DWORD x, DWORD y, const char* text) {
	if (x > 2400 || x < 0 || y < 0 || y > 1800) {
		text = "";
	}

	x += 200;

	__asm {
		mov ecx, text
		push y
		push x
		call text_address
		add esp, 8
	}
}
```

Like we have done with our refactoring, we will place our menu functionality in its own class. Our menu needs to handle two things:

- Toggling items on and off
- Displaying a cursor and set of menu items

We will focus on displaying the menu first. To make the job of displaying the menu easier, we will create two arrays in our menu class definition: one that contains item display texts, and one that contains item states:

```
#define MAX_ITEMS 4

public:
const char* items[MAX_ITEMS] = { "Wallhack", "ESP", "Aimbot", "Triggerbot" };
```


Playrix Ex. 1024, page 307

```
bool item_enabled[MAX_ITEMS] = { false };
```

We will also need a way to return a string of **On** or **Off** depending on the item's state:

```
const char* Menu::get_state(int item) {
      return item_enabled[item] ? "On" : "Off";
}
```

With these pieces in place, we can now add a loop in the text code cave to display all the menu items:

```
for (int i = 0; i < MAX_ITEMS; i++) {
    print_text(50, 250 + (100 * i), menu->items[i]);
    print_text(500, 250 + (100 * i), menu->get_state(i));
}
```

With our items printed, we can move on to adding a cursor. Our cursor will need to have a character and a position, so we will add these in the class definition. We also need to create an external function to handle all input for our menu:

```
public:
int cursor_position;
const char* cursor = ">";

const char* get_state(int);
```

To handle our input, we will use **GetAsyncKeyState**, similar to what we did in previous chapters. First, we will handle up and down:

```
void Menu::handle_input() {

      if (GetAsyncKeyState(VK_DOWN) & 1) {
            cursor_position++;
      }
      else if (GetAsyncKeyState(VK_UP) & 1) {
            cursor_position--;
      }
```


Playrix Ex. 1024, page 308

The **&1** has the effect of only registering the key press a single time for a short period of time instead of spamming it. The API documentation discusses this behavior.

If we press left and right, we want to enable or disable an item. Since all the item states are either true or false, we can simply switch their current value with the not (**!**) operator:

```
else if ((GetAsyncKeyState(VK_LEFT) & 1) || (GetAsyncKeyState(VK_RIGHT) & 1))
{
    item_enabled[cursor_position] = !item_enabled[cursor_position];
}
```

If we navigate past the boundaries of our menu, we want the cursor to appear at the other end. We can do that by adding a few checks:

```
if (cursor_position < 0) {
    cursor_position = 3;
}
else if (cursor_position > 3) {
    cursor_position = 0;
}
```

We can now add our menu to our main file. First, we need the text code cave to also print the cursor. We will make it look like it's moving by offsetting the current position with a multiple of 100:

```
print_text(10, 250 + (100 * menu->cursor_position), menu->cursor);
```

In our thread, we also need to pass input to the menu to check for key presses:

```
menu->handle_input();

playerGeometry->update();
```


Playrix Ex. 1024, page 309

5.10.9 Toggling Features

Finally, we need to toggle features based on their menu state. We already created an array of all the item states. To determine the current value of one of the features, we can query this array via:

```
if (menu->item_enabled[0])
```

To make these entries more readable, we can create constants in our menu header that reference the positions for each item:

```
#define WALLHACK 0
#define ESP 1
#define AIMBOT 2
#define TRIGGERBOT 3
```

We can implement checks in our code by using these values. For example, to toggle the wallhack, we can change the code to:

```
if (menu->item_enabled[WALLHACK]) {
    (*glDepthFunc)(0x207);
}
```

This builds off of our refactoring efforts from before. To toggle our aimbot, we can easily do the following in the thread:

```
if (menu->item_enabled[AIMBOT]) {
    playerGeometry->set_player_view();
}
```

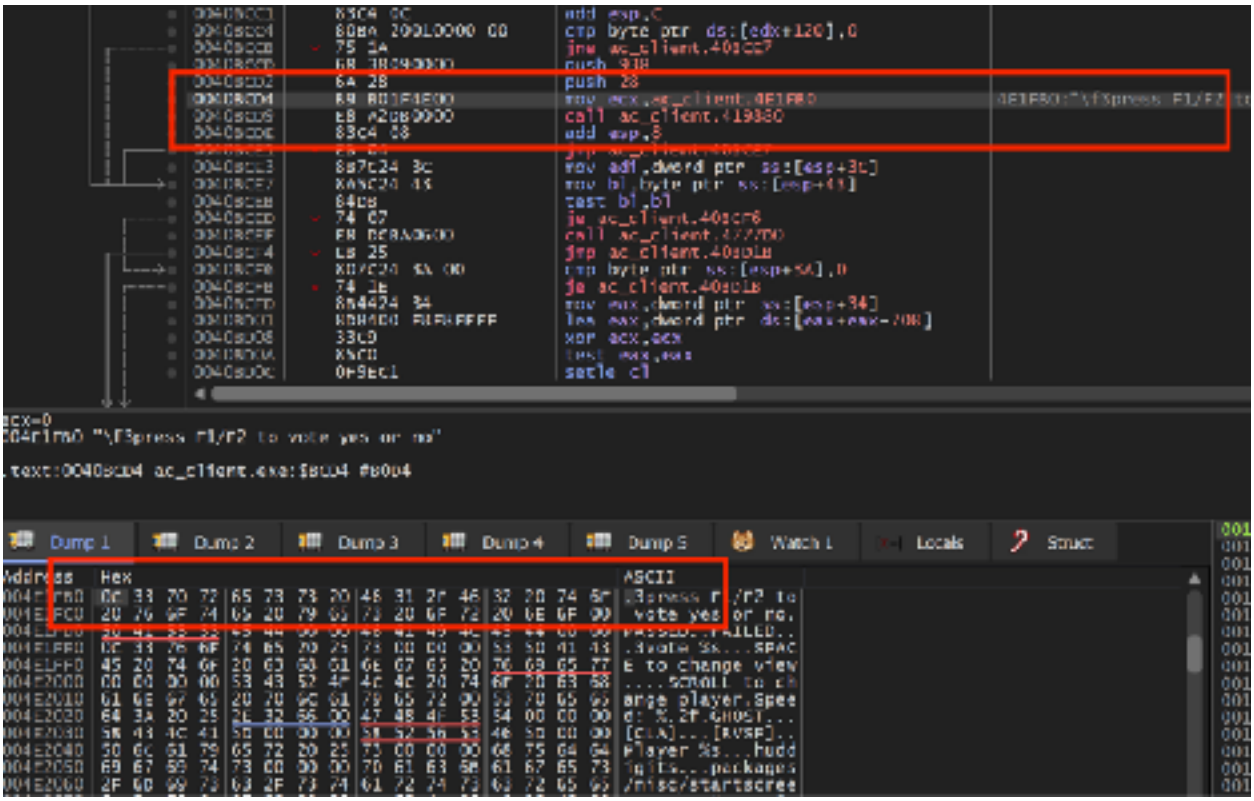
Similar checks can be done for the triggerbot and ESP.

5.10.10 Adding Colors

Just for some visual flair, we can add colors to the menu items to make them easier to read. By issuing votes in game, we can see that some text in the game already has

Playrix Ex. 1024, page 310

color, like the *Press F1 to vote* text. If we examine this string in x64dbg, we see that it has the following data:



It looks like strings prefixed with `0x0C 33` are given a color. If we look for other strings, we see that `0x0C` is always there, but occasionally the value of `0x33` will be different. Let's incorporate these bytes into our on/off strings:

```
class Menu {
private:
	const char on_text[5] = { 0xc, 0x33, 'O', 'N', 0 };
	const char off_text[6] = { 0xc, 0x33, 'O', 'F', 'F', 0 };
```

If you go into the game, you will see both our strings are now red. If you play around with the `0x33` and try different values (`0x31`, `0x38`, etc.), you will eventually see that `0x30` is green. Now we can modify our code to change the **On** text to green:

```
const char on_text[5] = { 0xc, 0x30, 'O', 'N', 0 };
```

With this, our multihack is complete.

Part 6
Multiplayer

Playrix Ex. 1024, page 312

6.1 Multiplayer Fundamentals

6.1.1 Peer-2-Peer

Imagine that two neighbors want to play a game of chess. One approach may be to have Neighbor A set up a chess board in his house and give Neighbor B the house keys. At any point during the day, Neighbor A could make a move. However, if Neighbor B wants to make a move, he has to walk over to Neighbor A's house. Additionally, if Neighbor B wanted to think on his move, he would need to take a picture or somehow record the copy of the chess board before he went back over to his own house.

This is an example of a Peer-2-Peer (P2P) model. In a P2P model, one player acts as the host and all other players act as guests. This is the model many console games use to handle multiplayer functionality. Its major downside is that the host will have an advantage in terms of response time. This is because all other players must connect to the host to retrieve and send updates, while the host can update his local copy.

6.1.2 Client-Server

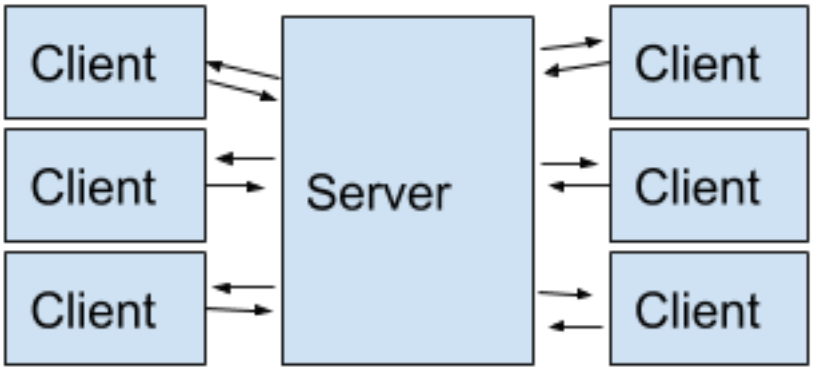
Imagine now that Neighbor A and B want to play chess, while another neighbor (C) wants to observe the chess game. This time, each neighbor has their own chess board and all players agree that an additional neighbor (D) will be the judge. The judge is the most trusted party of all involved and his roles are making sure no illegal moves happen and maintaining the “correct” version of the chess board.

To make a move, Neighbor A would write his move on an envelope and place the envelope in Neighbor D's mailbox. Neighbor D would then ensure that the move is legal, update his board, and then place letters in Neighbor B and C's mailboxes containing the move. Neighbors A, B, and C are all responsible for updating their boards to match the board of Neighbor D. If Neighbor A delivers a move that is

Playrix Ex. 1024, page 313

impossible, Neighbor D will warn him that it appears his board is not up-to-date and he cannot make that move.

This is an example of a client-server model, which we briefly discussed in Chapter 1.2.



In a client-server model, the server is a trusted entity that all clients connect to. When playing a multiplayer game, the server will not directly participate in the game, but it is responsible for keeping a trusted copy of the game's state. Each client will send updates to the server, and the server will distribute those updates to other clients. If a player sends too many updates that are not legal, the server will warn the client that it is desynchronized before kicking the client off.

6.1.3 Packets

In the client-server chess example, each neighbor placed an envelope with their move inside Neighbor D's mailbox. In networking, these envelopes are known as packets. Just like envelopes, packets contain who the packet is from, who the packet is going to, and the data itself. For example, if a player sends a chat message of **hello** in a multiplayer game like Wesnoth, the packet might contain the following information:

```
source: player
destination: server
data: hello
```

The larger the packet, the more time it will take to transmit from the client to the server. The more time it takes, the more lag is present in the client and server communication. To ensure that lag is at a minimum, packets contain the minimum amount of information possible. For example, if a client wants to say they fired a single shot from their weapon, the packet might look like:

```
source: player
destination: server
data: f1
```

In this example, both the client and server agree that **f** means fire and **1** means 1 round.

6.1.4 Network Protocols

If you want to tell someone, "I like this restaurant," you need to ensure that you are speaking the same language as the other person. Depending on their language, the syntax or structure of this sentence may differ, or certain words may be conjugated differently. This same logic applies when sending packets over a network. These communication rules are called protocols. These protocols determine how both the source and destination will communicate and how individual packets will look. The two main protocols you will encounter when looking at game network traffic are UDP and TCP.

Imagine you want to send a letter to the neighbor across the street, reminding him to water his plants. You do not expect a response to this letter, so you give it to your dog to take over to him. You would like this letter to get to him, but you will not be particularly upset if it does not. This is an example of UDP, in which packets are sent without any method to determine that they have arrived.

Now imagine you want to exchange multiple letters with your neighbor. Since you will be responding directly to what your neighbor says, you want to ensure that all letters are delivered. You and your neighbor agree to light your respective porch lights when you have received a letter. This is an example of TCP, in which an upfront connection is established and packets are acknowledged as delivered.

The data contained within TCP and UDP packets can be identical, but the packets will be different. This is because each protocol has a different header that is used by both the source and the destination to understand the data in the packet.

6.1.5 Sockets

Both TCP and UDP packets use the Internet Protocol (IP) to handle the process of routing the packet from the source to the destination. Each network device has an IP address that represents that device's “location". To differentiate between types of

Playrix Ex. 1024, page 315

traffic (such as web browsing, email, or video chat), packets also have a port number. For example, to browse a website over HTTP, you could visit 123.45.67.89 on port 80. While browsing, you could also connect to this machine using SSH, another service, on port 22. Both of these requests could be handled simultaneously as they are being handled by different programs listening on different ports.

An IP:Port pair is sometimes referred to as a socket. Sockets represent endpoints that can be communicated with. Windows has an API known as WinSock to enable programmers to quickly write programs that communicate to different destinations over TCP or UDP.

Playrix Ex. 1024, page 316

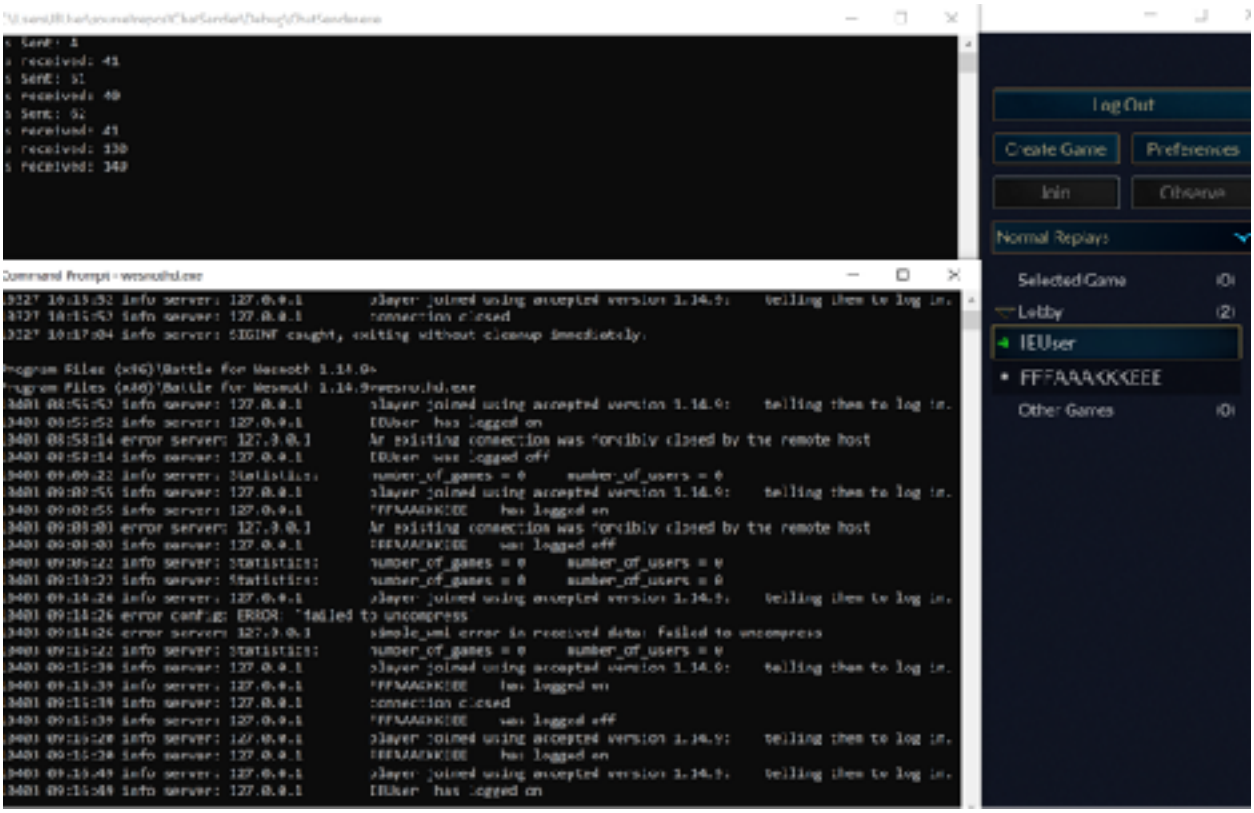
6.2 Packet Analysis

6.2.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

6.2.2 Identify

Like many games, Wesnoth has a multiplayer mode that allows multiple players to join a lobby, chat with each other, and play games against each other over a network. Our goal in this chapter is to analyze the packets used for connecting to the lobby and create a client that will connect to a lobby.



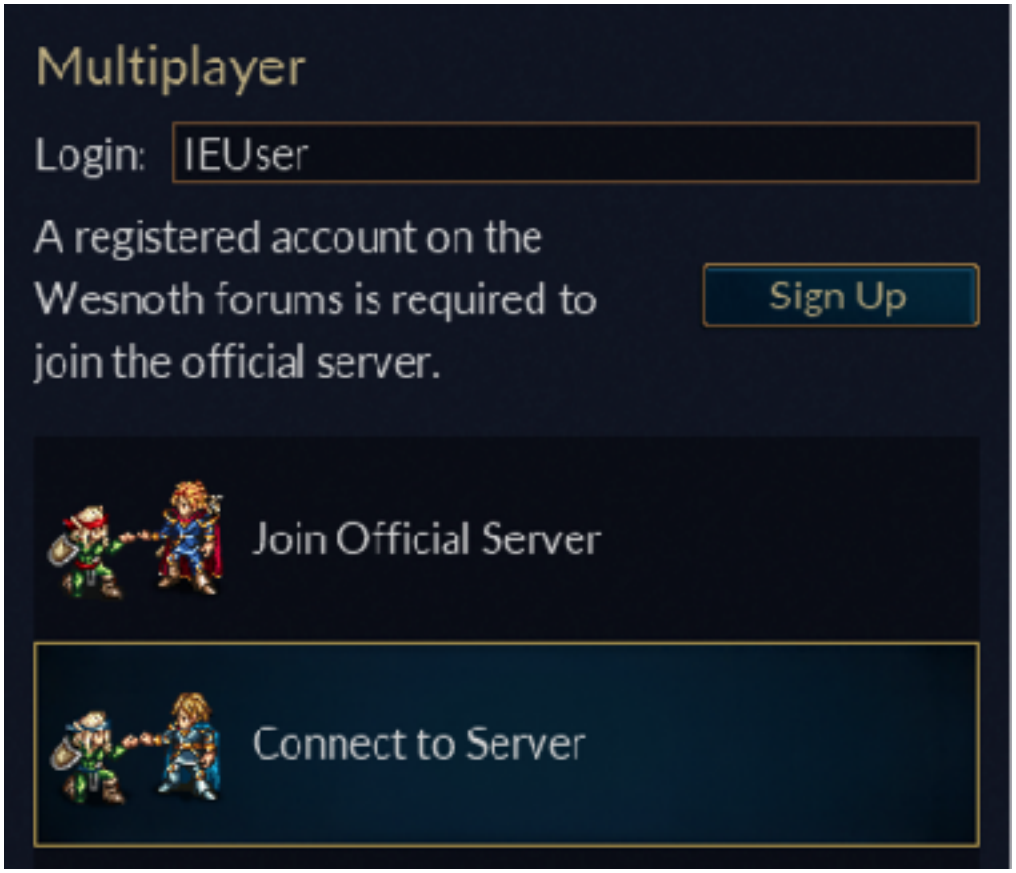
6.2.3 Understand

For multiple players to communicate over a network, all of the clients (in this case, the Wesnoth game executable) must agree on a network model and protocol. They also must agree on the data each packet will contain. If there is a server, the server must also agree with all of these components.

Since this data is structured, we can first identify the network model and protocol used by the game. Then, we can observe the packets being sent and reverse the data to determine what each packet is doing. We can then use the data in these packets to create our own client using the Windows' Socket API.

6.2.4 Local Server

If you start Wesnoth and click *Multiplayer*, you will see the following screen:



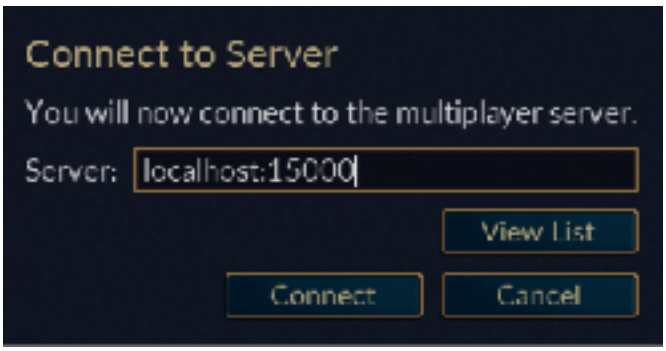
These entries indicate that Wesnoth is using a client-server model. If we explore the Wesnoth game folder in *C:\Program Files (x86)\Battle for Wesnoth 1.14.9*, you will find a program called *wesnothd.exe*. Reading the documentation on the developer's website, we know that this is a server daemon that allows you to host a server. It can be run by invoking it from the command prompt:



With the server running, we can now connect to it from Wesnoth. In the previous chapter, we discussed how clients need to know two pieces of information to connect to another host: the IP address and the port. In this case, the server is running on our local machine. There are several reserved IP address ranges that will never be used for normal network assignments. One of these is the range from 127.0.0.0 to 127.255.255.255, which is reserved for loopback addresses on the local machine. The loopback component indicates that the external network will not be able to access these IP ranges.

On all operating systems, 127.0.0.1 will always direct to your current host. In addition, localhost is a hostname that directs to 127.0.0.1. Therefore, we know that the IP for this server is 127.0.0.1 or localhost. From the documentation, we know that the server runs by default on port 15000. With these two pieces of information, we can connect to the server.

Choose *Connect to Server* and then enter in *localhost:15000*:



Playrix Ex. 1024, page 319

When you hit *Connect*, your client will join a multiplayer lobby. If you observe the server running in the command prompt, you should see that it has printed out the connection event:

```
C:\Program Files (x86)\Battle for Wesnoth 1.14.9>
C:\Program Files (x86)\Battle for Wesnoth 1.14.9>wesnothd.exe
20210403 08:55:52 info server: 127.0.0.1        player joined using accepted version 1.14.9:    telling them t
20210403 08:55:52 info server: 127.0.0.1        TFUser  has logged on
20210403 08:58:14 error server: 127.0.0.1       An existing connection was forcibly closed by the remote host
20210403 08:58:14 info server: 127.0.0.1        TFUser  was logged off
```

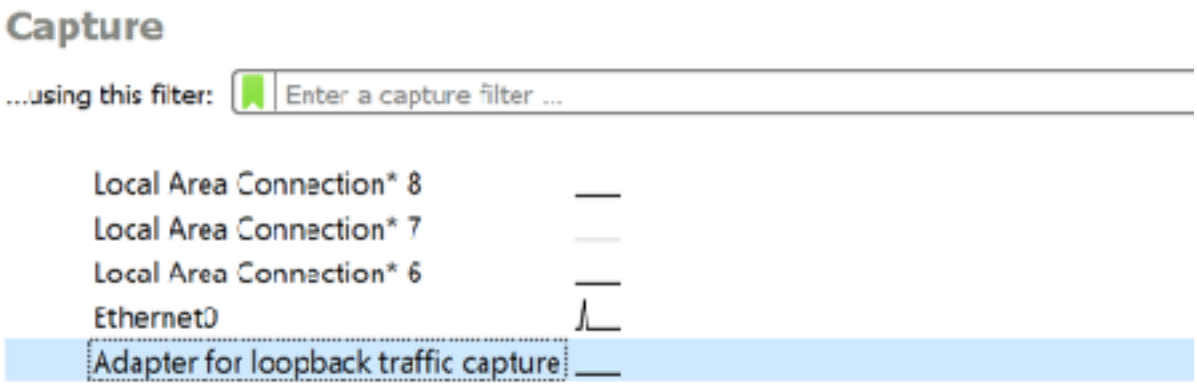
6.2.5 Observing Packets

With the server running, we can close the Wesnoth client and start the process of observing packets. There are many tools that can be used, but for these chapters, we will use Wireshark. This can be installed via:

```
choco install wireshark
```

The first time you use Wireshark, you will also need to install the WinPcap driver as instructed by the program.

With Wireshark and the driver installed, you can pick a network interface to observe on:

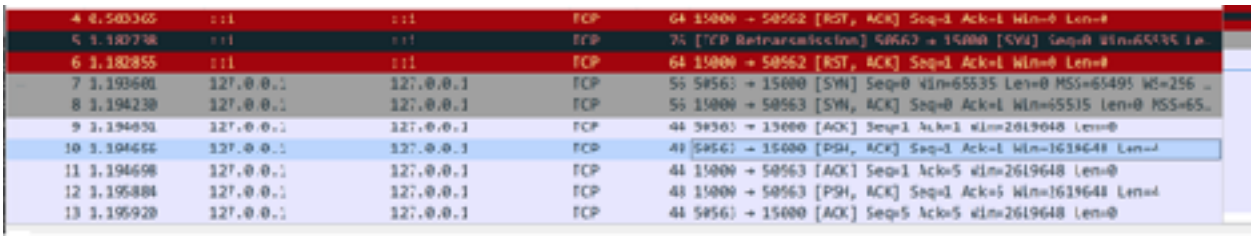


Each listed network interface represents a piece of software or hardware that connects to a public or private network. For example, the *Ethernet0* interface listed is the default

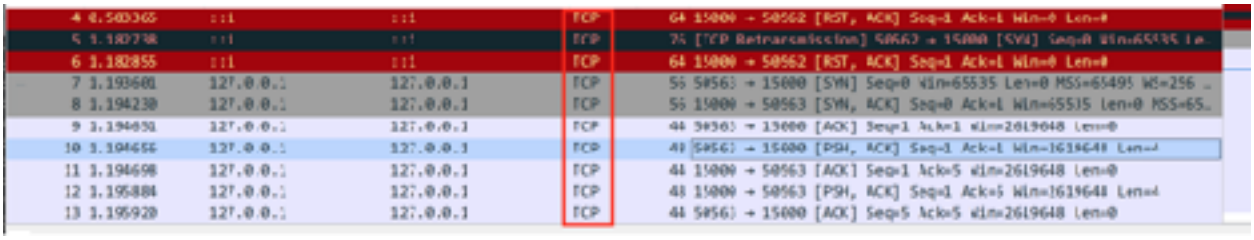
Playrix Ex. 1024, page 320

network card used to communicate over the Internet for this particular VM. Depending on your VM and computer, these interfaces may be different.

In this case, we know all of the traffic for our game will be flowing on the loopback interface, so select *Adapter for loopback traffic capture*. Upon selecting this, Wireshark will start monitoring for packets. Open up Wesnoth and connect to the local server. When connecting, you should see Wireshark log multiple packets:



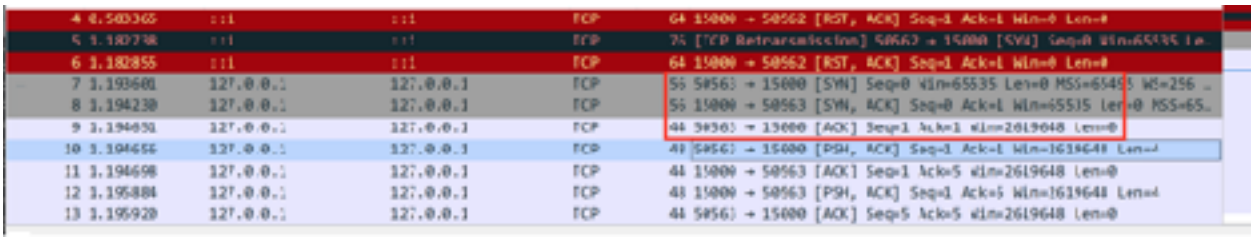
Initially, this can appear overwhelming, so let's break down what exactly we are seeing here and identify what we care about. In the protocol column, we can see that Wesnoth is using *TCP*:



We know from the previous chapter that TCP initiates an upfront connection and acknowledges when packets have been received. This initial negotiation is known as a three-way handshake, and has three parts:

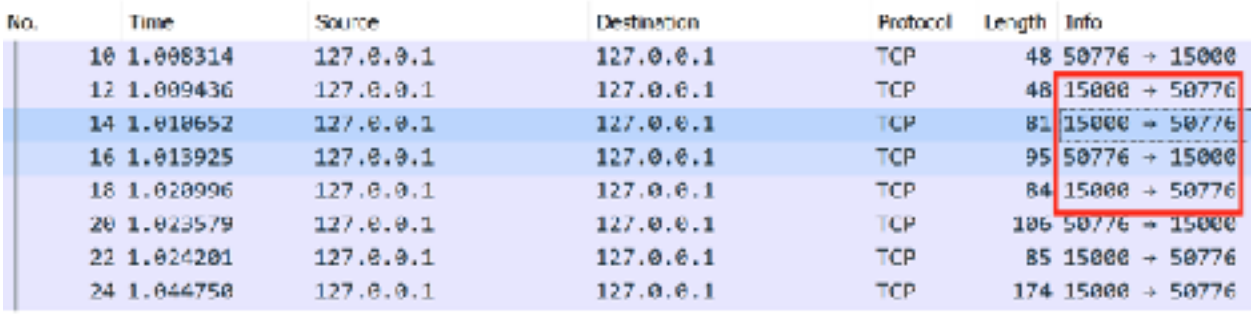
1. One side sends a packet with a *SYN* flag.
2. The other side responds with *SYN* and *ACK* flags.
3. The first side sends an *ACK* flag.

We can see this behavior in the first few packets highlighted below:



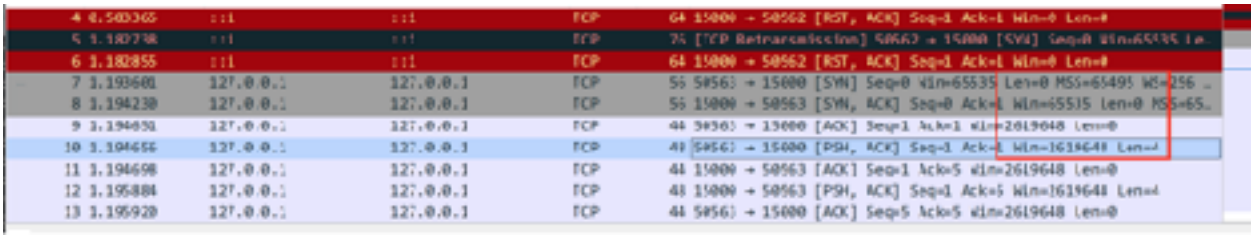
Playrix Ex. 1024, page 321

Since we know 15000 is the server's port, we can determine that 50563 is our client's port. However, this number will probably not match the number you are seeing. If we close Wesnoth and start it again, we will also see that this number changes:

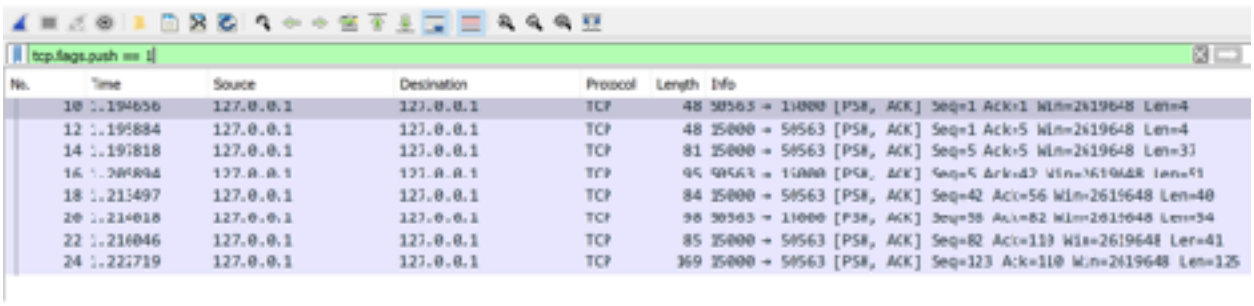


This is an example of an ephemeral, or short-lived, port. Since the Wesnoth client does not need to be discoverable by other users, it can choose a "random" available port each time it starts up. When the client is closed, it will free this port. This is in contrast to the server, which always needs to be discoverable on port 15000 by clients.

None of the packets we have examined so far contain any data. We can determine this by looking at the *len* member:



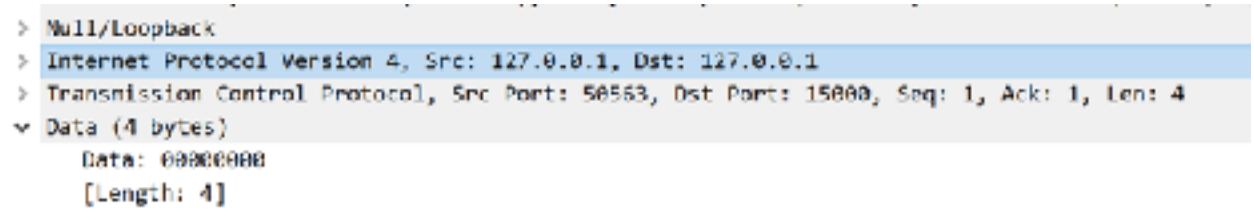
Looking at all the packets, we can see that the only packets with data have the *PSH* flag. This flag tells the TCP connection to immediately send whatever data is inside the packet to the associated application instead of placing it in a buffer. We can filter for this flag in Wireshark to only see the packets that we care about via *tcp.flags.push == 1*:

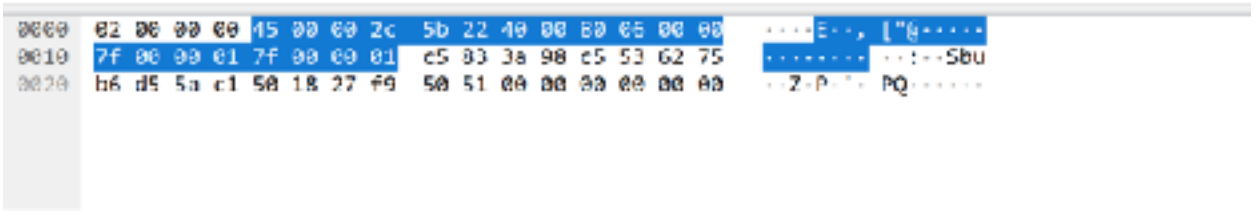


Playrix Ex. 1024, page 322

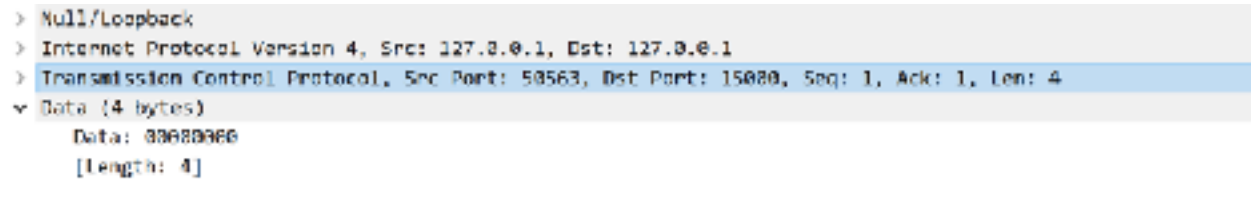
6.2.6 Packet Structure

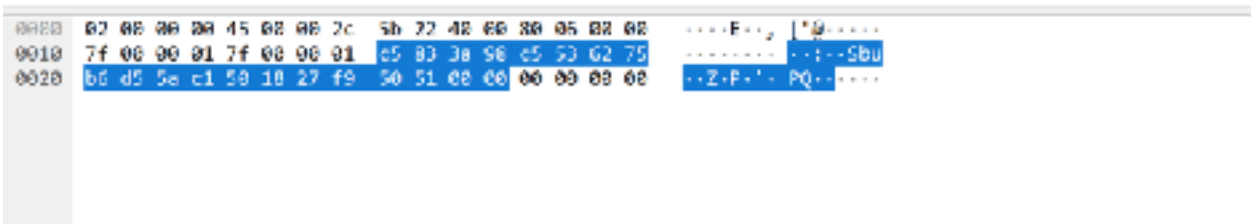
When you select a packet in Wireshark, it displays the full packet broken down in different segments. For example, if we select the first packet of *len* 4, we will see the following view:



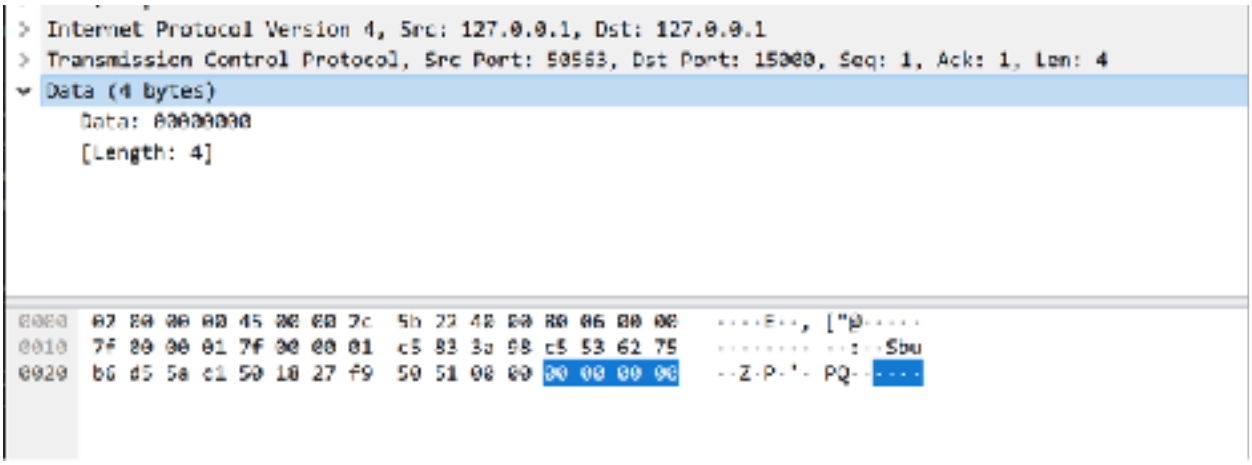


As you select different components (like *IP* and *TCP*), the associated parts of the packet will be highlighted in the bottom section. For example, by selecting the *TCP* component, the following section is highlighted:

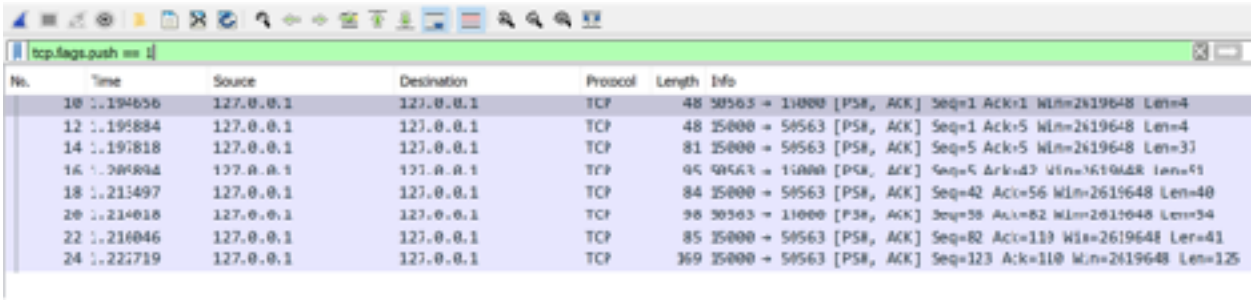




For the purposes of this chapter, we can ignore the IP and TCP components and focus on the data component for all the packets:



Given that data in packets is often compressed, it is difficult to determine the purpose of a single packet in isolation. Instead, it is easier to look at the overall flow of network traffic and determine the role of each packet. In our case, the flow of network traffic for connecting looks like:



We know that packets from *50563 -> 15000* represent communication sent from our client to the server, and *15000 -> 50563* represent communication sent from the server to our client. As such, the network traffic looks like:

1. Client -> Server Packet 1
2. Server -> Client
3. Client -> Server Packet 2
4. Server -> Client
5. Client -> Server Packet 3

Since we are writing a client, we will only need to reverse the three packets being sent from the client to the server.

Playrix Ex. 1024, page 324

6.2.7 Sockets

Each OS will have its own set of API's that allows you to interact with the networking stack. In Windows, this API is known as Winsock. Microsoft has comprehensive documentation available on how to use this API, including the process to establish a socket. This is available here.

Microsoft also provides a complete example using these API's here. This example will create a TCP connection to a provided IP on port 27015 and send a single packet containing the data **this is a test**. It will then continuously wait for packets from the server. We will base our code on this example.

A good starting place is to see how the server responds when we simply use the code as is. Since we are targeting a specific IP, we will remove the following code from the example:

```
if (argc != 2) {
    printf("usage: %s server-name\n", argv[0]);
    return 1;
}
```

Next, we can modify the **getaddrinfo** function to use the values for our server:

```
iResult = getaddrinfo("127.0.0.1", "15000", &hints, &result);
```

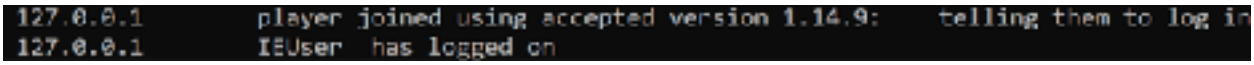
With these changes, compile the code and run the executable. If you have Wireshark still logging, you should notice your host sending a packet with the data **this is a test** to the server. If you look at the server process, you will see the following message:

```
20210403 09:34:08 error server: 127.0.0.1       incorrect handshake
```

6.2.8 Reversing Packets

This message indicates that the first packets sent by our client must initiate some sort of handshake. Let's compare the first message logged by the server for a valid connection:

Playrix Ex. 1024, page 325



Going back to our Wireshark capture, we can see that the first packet sent by the client contains `00 00 00 00`. The server then responds with data. From this, we can assume that the data `00 00 00 00` is interpreted by the Wesnoth server as the start of a handshake.

We can modify our socket example to perform this behavior. First, remove the following code since we will be writing our own sending code:

```
// Send an initial buffer
iResult = send( ConnectSocket, sendbuf, (int)strlen(sendbuf), 0 );
if (iResult == SOCKET_ERROR) {
    printf("send failed with error: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

printf("Bytes Sent: %ld\n", iResult);

// shutdown the connection since no more data will be sent
iResult = shutdown(ConnectSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
    printf("shutdown failed with error: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}
```

Next, create a buffer that will hold our `00 00 00 00` data:

```
const unsigned char buff_handshake_p1[] = {
    0x00, 0x00, 0x00, 0x00
};
```

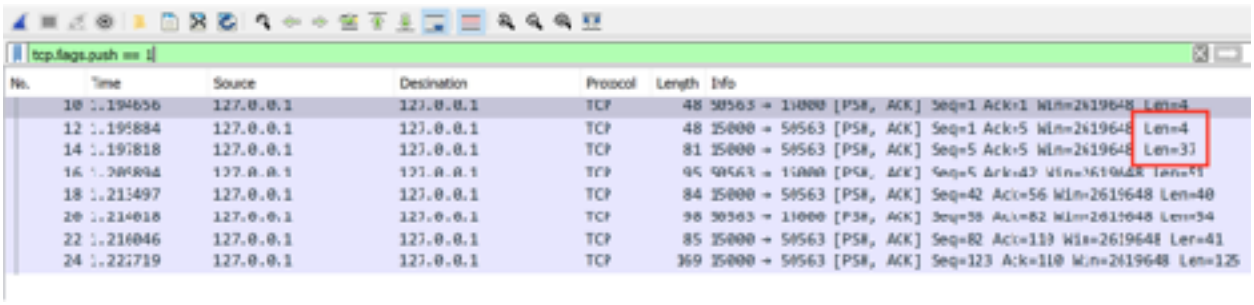
Finally, add the following code to send this data and receive a single packet back:

```
iResult = send(ConnectSocket, (const char*)buff_handshake_p1,
(int)sizeof(buff_handshake_p1), 0);
printf("Bytes Sent: %ld\n", iResult);
```


Playrix Ex. 1024, page 326

```
iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
printf("Bytes received: %d\n", iResult);
```

If you run this program, you will receive 41 bytes back. This is equal to the two responses sent by the server in Wireshark, indicating that the first packet sent by the client initiates the handshake:



From the server messages, we can see that the next packet the client is responsible for is sending their current version. An example of this packet's data is shown below:

```
0x00, 0x00, 0x00, 0x2f, 0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xff, 0x8b, 0x2e, 0x4b, 0x2d, 0x2a, 0xce,
0xcc, 0xcf, 0x8b, 0xe5, 0xe2, 0x84, 0xb2, 0x6c, 0x95, 0x0c,
0xf5, 0x0c, 0x4d, 0xf4, 0x2c, 0x95, 0xb8, 0xa2, 0xf5, 0xe1,
0x92, 0x5c, 0x00, 0xc0, 0x38, 0xd3, 0xd7, 0x28, 0x00, 0x00,
0x00
```

Even when converted into ASCII, our game version (1.14.9) does not appear in this data. This is because, like most games, Wesnoth compresses all data by default. In future chapters, we will examine the compression scheme used so that we can create packets with custom data. However, in this chapter, we will not need to do this since this data does not change. You can verify that by joining the same server multiple times with Wireshark running.

Let's add this packet to our program to send as well:

```
const unsigned char buff_handshake_p2[] = {
    0x00, 0x00, 0x00, 0x2f, 0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0xff, 0x8b, 0x2e, 0x4b, 0x2d, 0x2a, 0xce,
    0xcc, 0xcf, 0x8b, 0xe5, 0xe2, 0x84, 0xb2, 0x6c, 0x95, 0x0c,
    0xf5, 0x0c, 0x4d, 0xf4, 0x2c, 0x95, 0xb8, 0xa2, 0xf5, 0xe1,
    0x92, 0x5c, 0x00, 0xc0, 0x38, 0xd3, 0xd7, 0x28, 0x00, 0x00,
```


Playrix Ex. 1024, page 327

```
    0x00
};

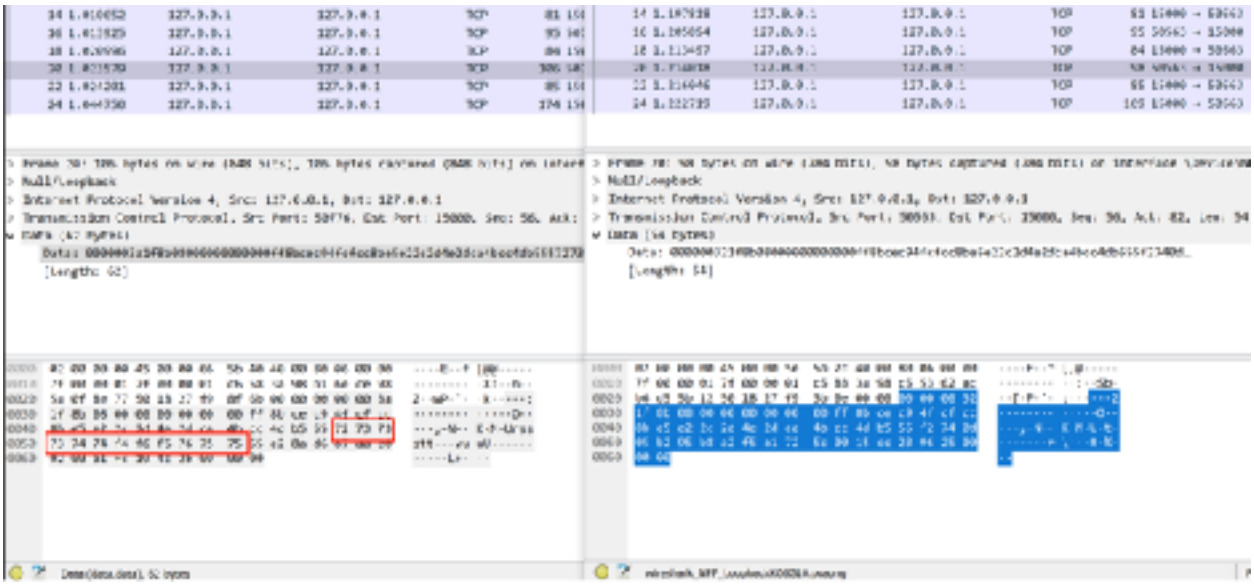
iResult = send(ConnectSocket, (const char*)buff_handshake_p2,
(int)sizeof(buff_handshake_p2), 0);
printf("Bytes Sent: %ld\n", iResult);

iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
printf("Bytes received: %d\n", iResult);
```

With this additional packet, the Wesnoth server will now think that a client is sending them a game's version before closing the connection:

```
info server: 127.0.0.1        player joined using accepted version 1.14.9:    telling them to log in.
info server: 127.0.0.1        connection closed
```

Finally, we can add in the name that our client will send. Since we have control over this field, we can use it to observe the compression scheme in use. With Wireshark running, connect to a server with two usernames, one short and one long. In the long username, make sure multiple characters repeat in a row. This will allow us to detect patterns. In this chapter, we will use the examples of *FFFAAAKKKEEE* and *IEUser*. Their related packets look like:

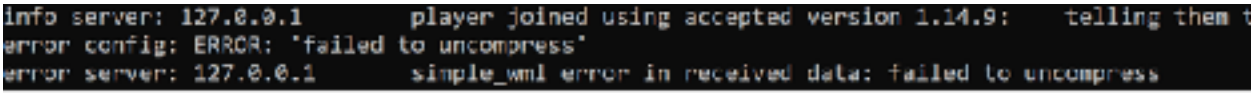


The highlighted areas most likely represent the compressed name of the user. Let's try sending the data from the *FFFAAAKKKEEE* request, but slightly modifying the bytes for the name:

```
const unsigned char buff_send_name[] = {
    0x00, 0x00, 0x00, 0x3a, 0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0xff, 0x8b, 0xce, 0xc9, 0x4f, 0xcf, 0xcc,
    0x8b, 0xe5, 0xe2, 0x2c, 0x2d, 0x4e, 0x2d, 0xca, 0x4b, 0xcc,
    0x4d, 0xb5, 0x55, 0x72, 0x74, 0x74, 0x74, 0x74, 0x74, 0xf4,
    0xf6, 0xf6, 0x76, 0x75, 0x75, 0x55, 0xe2, 0x8a, 0xd6, 0x87,
    0xaa, 0xe0, 0x02, 0x00, 0xa1, 0xfc, 0x19, 0x4c, 0x2b, 0x00,
    0x00, 0x00
};

iResult = send(ConnectSocket, (const char*)buff_send_name,
(int)sizeof(buff_send_name), 0);
printf("Bytes Sent: %ld\n", iResult);
```

If we observe the server, we see the following error:

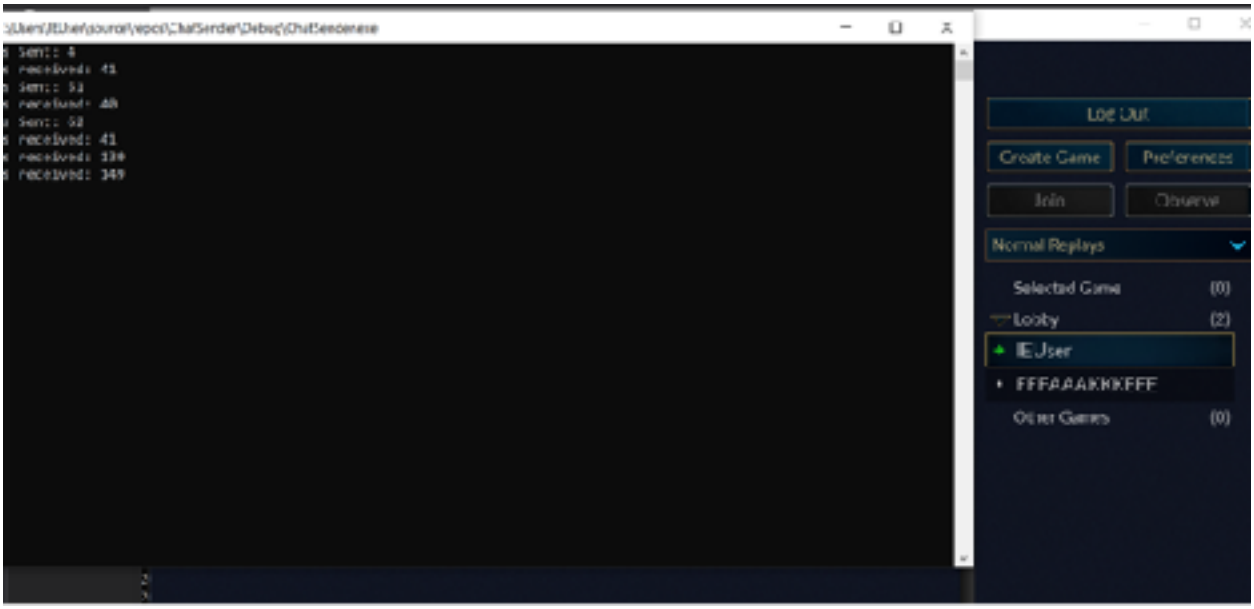


This verifies that the packet is being compressed, and even indicates the compression scheme (*simple_wml*). We can use this information in future chapters when we want to create our own packet. For this chapter, we can just modify **buff_send_name** to contain the original data:

```
const unsigned char buff_send_name[] = {
    0x00, 0x00, 0x00, 0x3a, 0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0xff, 0x8b, 0xce, 0xc9, 0x4f, 0xcf, 0xcc,
    0x8b, 0xe5, 0xe2, 0x2c, 0x2d, 0x4e, 0x2d, 0xca, 0x4b, 0xcc,
    0x4d, 0xb5, 0x55, 0x72, 0x73, 0x73, 0x73, 0x74, 0x74, 0xf4,
    0xf6, 0xf6, 0x76, 0x75, 0x75, 0x55, 0xe2, 0x8a, 0xd6, 0x87,
    0xaa, 0xe0, 0x02, 0x00, 0xa1, 0xfc, 0x19, 0x4c, 0x2b, 0x00,
    0x00, 0x00
};
```

With this change, our client will now connect to the server using the name *FFFAAAKKKEEE*. If you join the lobby with a legitimate client, you will notice that our client is also connected.

Playrix Ex. 1024, page 329



The full code for this client is available in Appendix A.

Playrix Ex. 1024, page 330

6.3 Reversing Packets

6.3.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

6.3.2 Identify

In the previous chapter, we identified the packets used for connecting to a Wesnoth server. We then wrote a client that would replay these packets. In this chapter, we will identify the packets used for sending chat messages and reverse their structure. This will allow us to create our own legitimate packets instead of only replaying packets.

6.3.3 Understand

For clients and servers to communicate over a network, both sides must agree on how to structure the data in each packet. Since this structure must be reversible for each side, we can also reverse it. Once the data is reversed, we can modify the data and do the opposite of the reversing process to create a new packet.

6.3.4 Chat Packets

Similar to reversing an executable, it is helpful to have a context when reversing packets. With executables, this context is often a string that we have observed inside the executable. When it comes to packets, this context is some type of data we can control in the packet.

Typically, you control several pieces of data in a packet. For example, most games allow you to set your name. Other games will allow you to connect with multiple versions or certain mods. Both of these pieces would allow you to associate certain

Playrix Ex. 1024, page 331

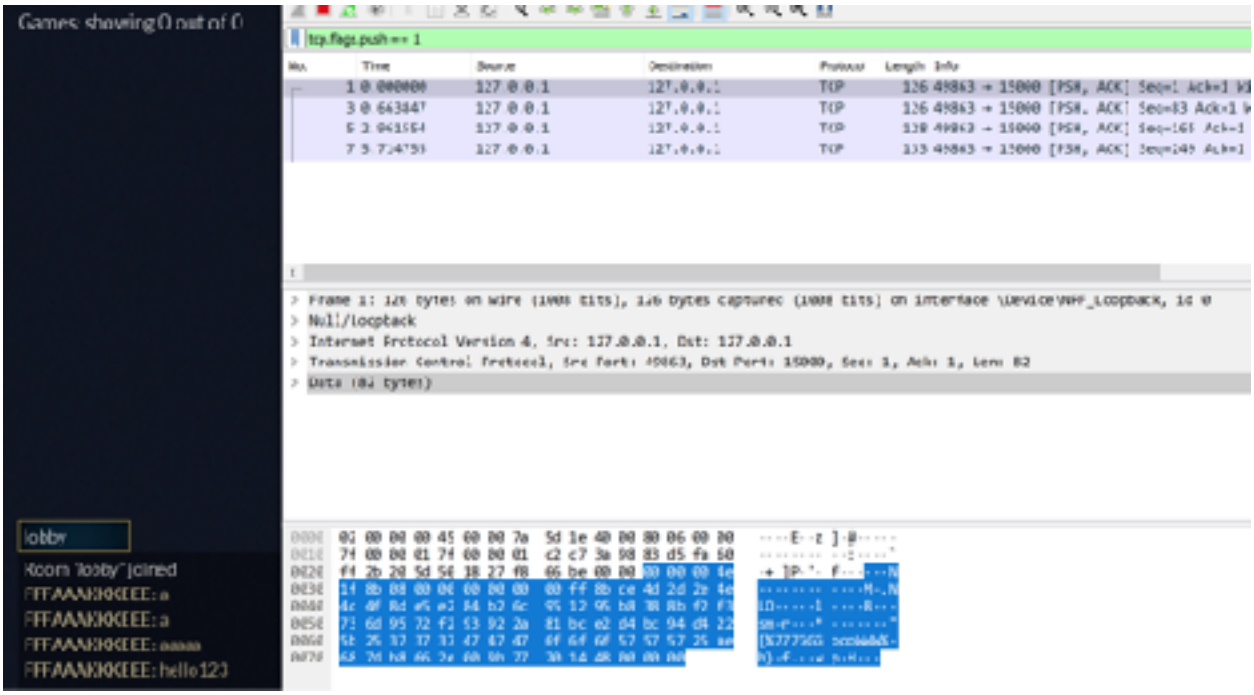
data with certain packets. However, one of the easiest pieces of data to control is a game's chat messages. Since Wesnoth allows players to send chat messages, this is the context we will use for this chapter.

Start Wesnoth and connect to your local server with a user named *FFFAAAKKKEEE*, identically to the last chapter. Once connected, start Wireshark to log packets. To help reverse the game's packets, we want to answer a few questions:

1. Is there any randomness or time element encoded in the packets?
2. Can we observe patterns of letters in the packets?
3. Can we observe any human-readable characters in the packets?

To answer these questions, we can send the following four chat messages:

- *a*
- *a*
- *aaaaa*
- *hello123*

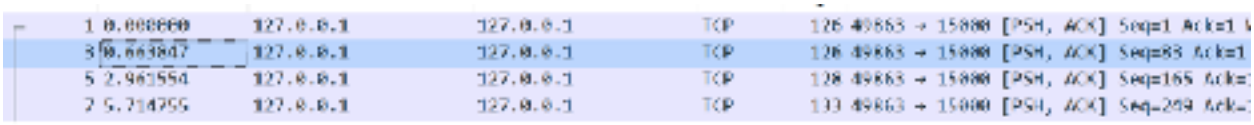


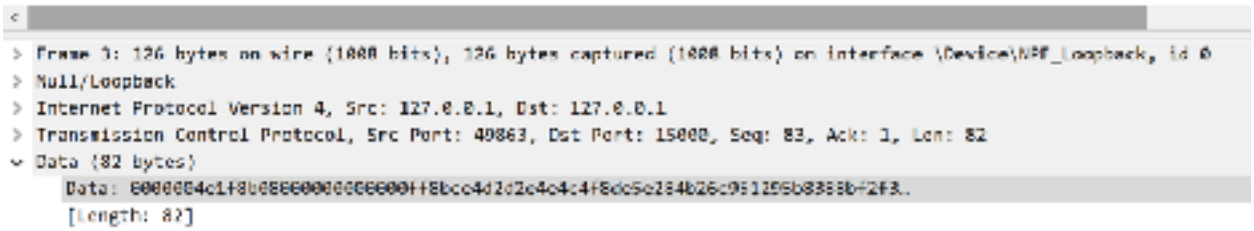
The first two messages are identical and will help us determine if identical messages result in identical packets. The third message repeats *a* several times, allowing us to observe any type of data patterns. Finally, the last message will immediately tell us if it appears readable in a packet.

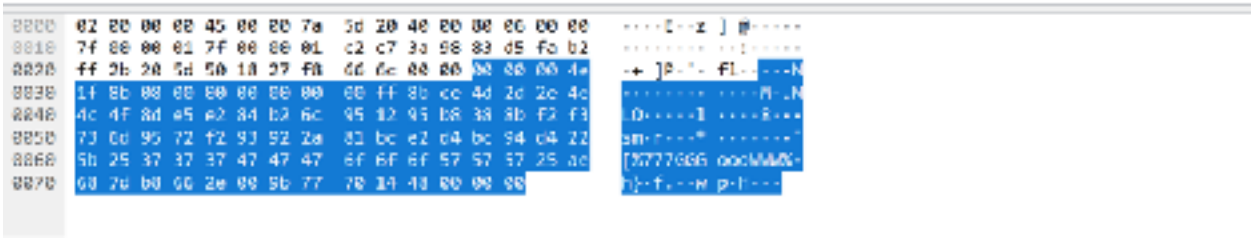
Playrix Ex. 1024, page 332

6.3.5 Test Cases

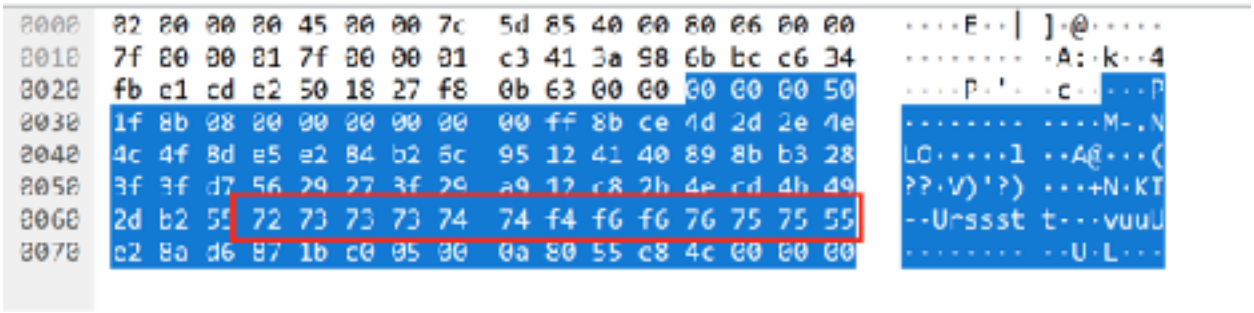
Examining the data of the first two packets (*a* and *a*), we can see that their data is identical. This means that the packets do not contain a timestamp or any other uniqueness factor:







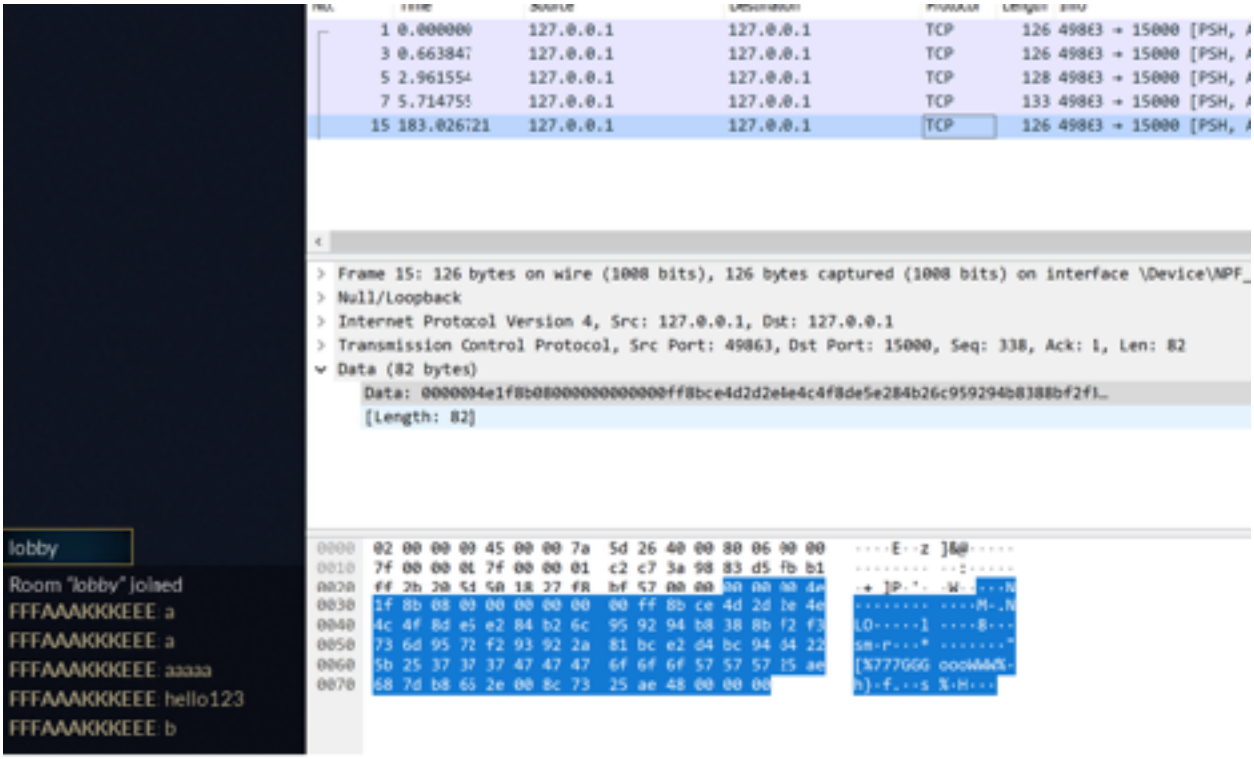
Next, examining the data of the third packet (*aaaaa*), we see a pattern toward the end:



This pattern is too long for *aaaaa*, and its structure of 3-3-3-3 is closest to our player's name (*FFFAAAKKKEEE*). We can see that this pattern also occurs in all of our messages, indicating that one element of this packet must contain our player's name.

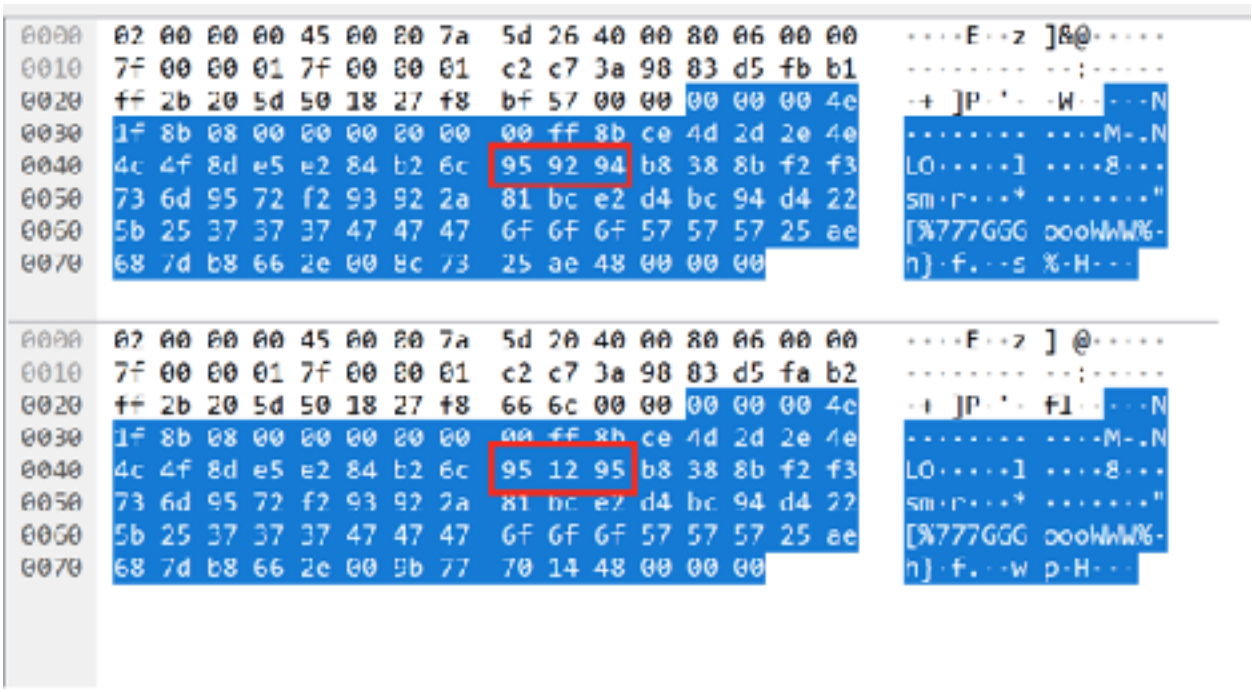
Finally, examining the data of the last packet, we cannot observe anything that would resemble *hello123*.

Next, let's observe the difference between the packets for the message *a* and a new message *b* to help determine how single characters are handled:



Comparing these two packets side by side, we find that the single-character modification resulted in 2 of the bytes being changed in the middle of the packet, and several bytes being changed at the end:

Playrix Ex. 1024, page 334



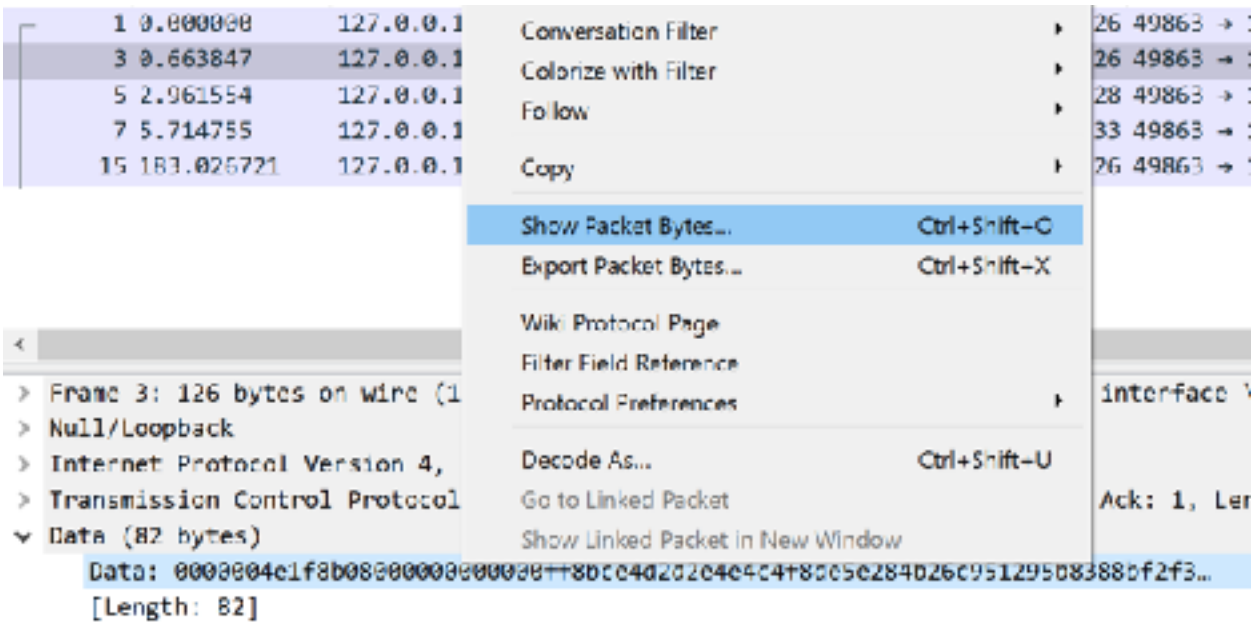
This demonstrates that our text is not being mapped one-to-one into a packet, and additional processing is taking place. With this information, we can close Wesnoth and stop logging packets in Wireshark. However, make sure to keep Wireshark running so we can grab packet data as we analyze it further.

6.3.6 Packet Modification

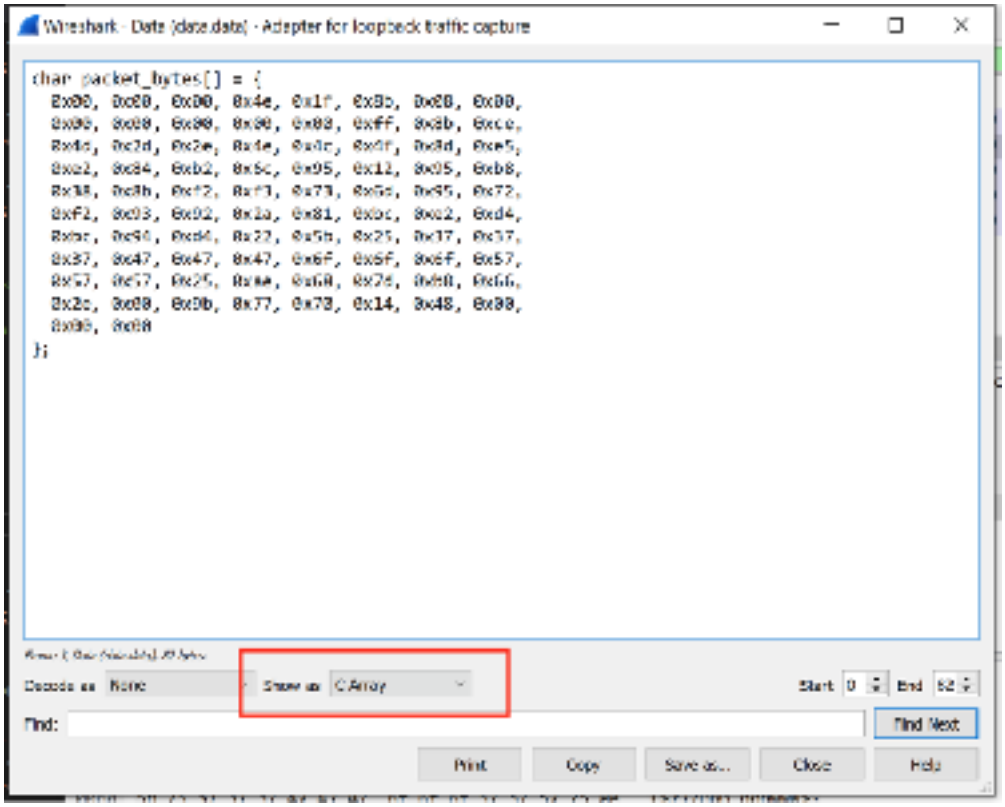
Now, let's see how the server responds if we modify a packet. In the last chapter, we wrote a client that would connect to the server with the username *FFFAAAKKKEEE*. We can expand on this code to send a chat message *a* after we connect.

In Wireshark, click on the first or second packet (the *a* messages) and select the data. Right-click and choose *Show Packet Bytes*:

Playrix Ex. 1024, page 335



In the window that appears, choose *C Array* in the *Show As* section. This will format the packet's data into an array that can be dropped directly into your code.



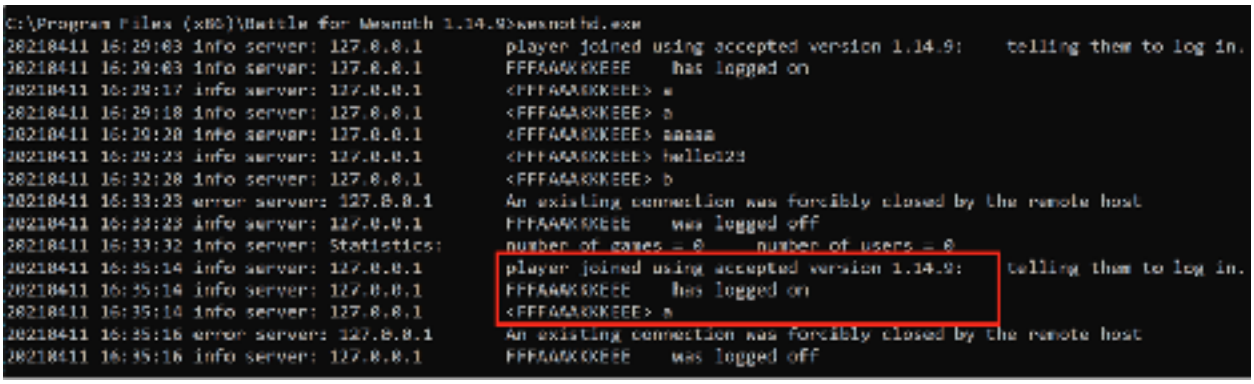
Playrix Ex. 1024, page 336

After we have sent the handshake, version, and username packets, we can add the following code to send the chat message:

```
const unsigned char packet_bytes[] = {
    0x00, 0x00, 0x00, 0x4e, 0x1f, 0x8b, 0x08, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0x8b, 0xce,
    0x4d, 0x2d, 0x2e, 0x4e, 0x4c, 0x4f, 0x8d, 0xe5,
    0xe2, 0x84, 0xb2, 0x6c, 0x95, 0x12, 0x95, 0xb8,
    0x38, 0x8b, 0xf2, 0xf3, 0x73, 0x6d, 0x95, 0x72,
    0xf2, 0x93, 0x92, 0x2a, 0x81, 0xbc, 0xe2, 0xd4,
    0xbc, 0x94, 0xd4, 0x22, 0x5b, 0x25, 0x37, 0x37,
    0x37, 0x47, 0x47, 0x47, 0x6f, 0x6f, 0x6f, 0x57,
    0x57, 0x57, 0x25, 0xae, 0x68, 0x7d, 0xb8, 0x66,
    0x2e, 0x00, 0x9b, 0x77, 0x70, 0x14, 0x48, 0x00,
    0x00, 0x00
};

iResult = send(ConnectSocket, (const char*)packet_bytes,
(int)sizeof(packet_bytes), 0);
printf("Bytes Sent: %ld\n", iResult);
```

Executing this code will send a chat message just as if we were connected:



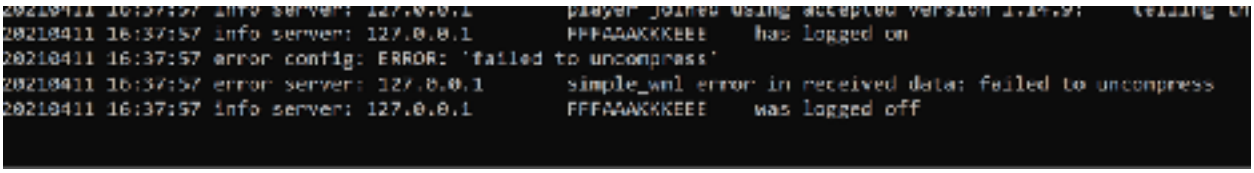
In the section above, we saw that the difference between the *a* and *b* chat messages was a change of 2 bytes. Let's change only the 2 bytes above and see how the server responds:

```
const unsigned char packet_bytes[] = {
    0x00, 0x00, 0x00, 0x4e, 0x1f, 0x8b, 0x08, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0x8b, 0xce,
    0x4d, 0x2d, 0x2e, 0x4e, 0x4c, 0x4f, 0x8d, 0xe5,
    0xe2, 0x84, 0xb2, 0x6c, 0x95, 0x92, 0x94, 0xb8,
```


Playrix Ex. 1024, page 337

```
    0x38, 0x8b, 0xf2, 0xf3, 0x73, 0x6d, 0x95, 0x72,
    0xf2, 0x93, 0x92, 0x2a, 0x81, 0xbc, 0xe2, 0xd4,
    0xbc, 0x94, 0xd4, 0x22, 0x5b, 0x25, 0x37, 0x37,
    0x37, 0x47, 0x47, 0x47, 0x6f, 0x6f, 0x6f, 0x57,
    0x57, 0x57, 0x25, 0xae, 0x68, 0x7d, 0xb8, 0x66,
    0x2e, 0x00, 0x9b, 0x77, 0x70, 0x14, 0x48, 0x00,
    0x00, 0x00
};
```

Executing the code with this packet will result in the following error from the server:



This error message, plus how a single letter change results in multiple modifications to the bytes in the packet, indicates that at least some part of the packet is compressed.

6.3.7 Compression

Compression is the process of taking input data and reducing its size. One of the most simplistic compression techniques is combining repeating data. For example, the string *AAAAAAAAAA* could become *10A*. When decompressed, the decompressor would know to expand *10A* back to *AAAAAAAAAA*. There are multiple ways to compress data, with some popular formats being ZIP and RAR. Just like executables are always distinct from other data (like pictures), different compression formats are distinct from each other.

Wesnoth is a multi-platform game that supports Windows and Linux. Therefore, we know that whatever format being used must run on both OS's. On Linux-based systems, two of the most popular compression formats are gzip and bzip2. We will start our investigation with these formats.

Windows' default command prompt does not have good support for data operations. To help us investigate, we will use another terminal emulator called Cmder. We will also install the gzip package. Both of these can be installed using Chocolatey in Powershell:

```
choco install cmder -y
choco install gzip -y
```


Playrix Ex. 1024, page 338

To test out the two compression techniques, create a text file named test.txt. In this file, add a single line of text. Next, open up Cmder (*C:\Tools\Cmder.exe*) and navigate to the directory with this text file. Run the following command to create a gzip'd version of the file:

```
gzip test.txt
```

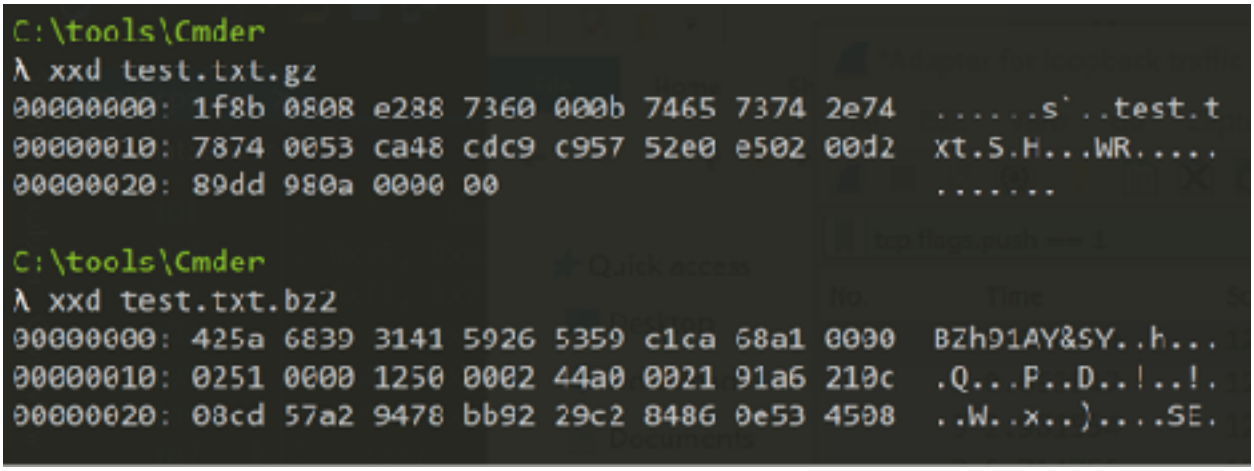
By default, gzip will remove the original file. Recreate test.txt in the same way so that you can then create a bzip'd version:

```
bzip2 test.txt
```

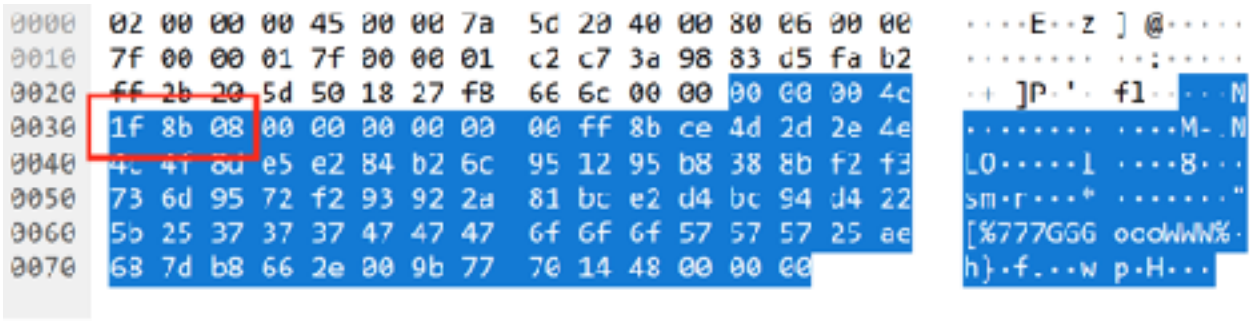
You should now have two files: test.txt.gz and test.txt.bz2. We next want to examine what the bytes of these files look like. To do this, we can use a tool called **xxd**:

```
xxd test.txt.gz
xxd test.txt.bz2
```

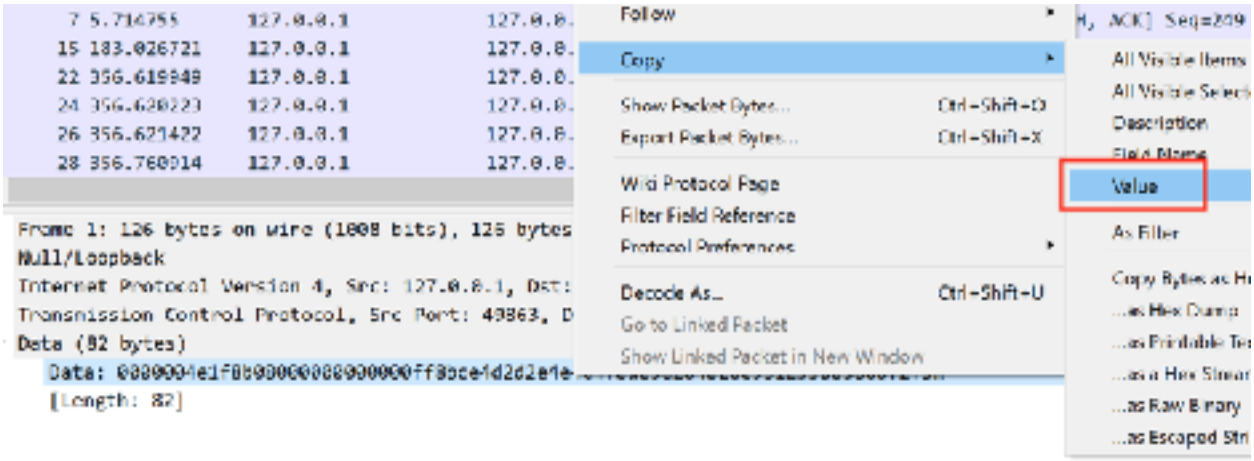
Your results should look similar to the following:



Comparing these against one of the packets, you should immediately notice that the beginning bytes (`0x1f8b`) jump out in the packet:



This indicates that part of the packet may be compressed with gzip. We can validate this by attempting to decompress an actual packet. In Wireshark, select a packet, right-click on the data, and choose *Copy -> Value*:



We can use xxd again to turn this data into a gzip'd file. To do this, we will first print the data to the terminal using the command *echo*. However, instead of only printing, we will pass this printed text to **xxd** via the pipe operator (**|**). We can then use the **-r** switch to tell xxd to reverse the operation (or create a file from the hex), and the **-p** switch to tell xxd to read from whatever is typed in, in this case the echo command. Finally, we use the redirection operator (**>**) to save this to a compressed file:

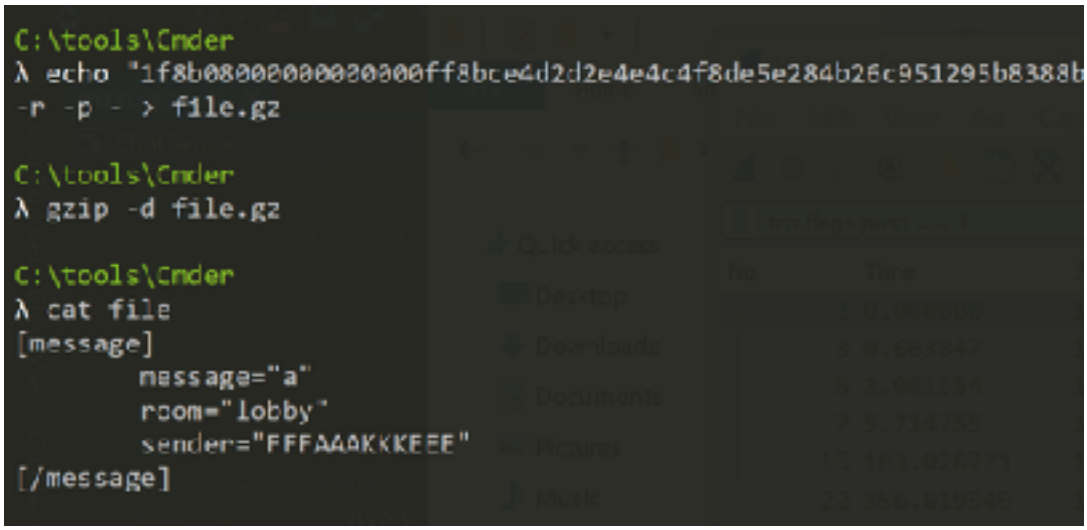
```
echo "1f8b08....." | xxd -r -p - > file.gz
```

We can then decompress this file using gzip:

```
gzip -d file.gz
```

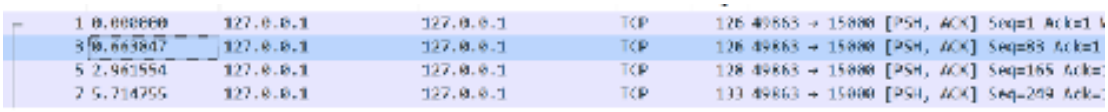

Playrix Ex. 1024, page 340

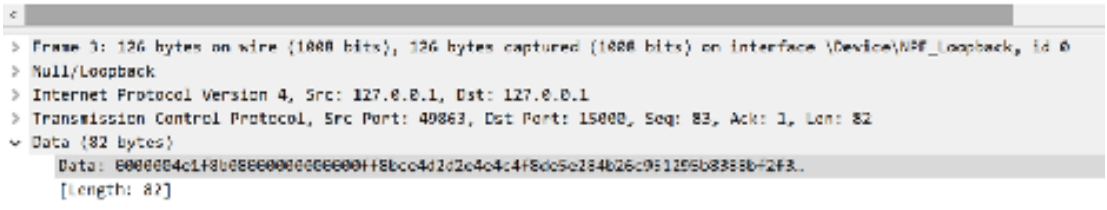
This will produce a text file named *file*. Viewing this file, we can see structured data representing our chat message:

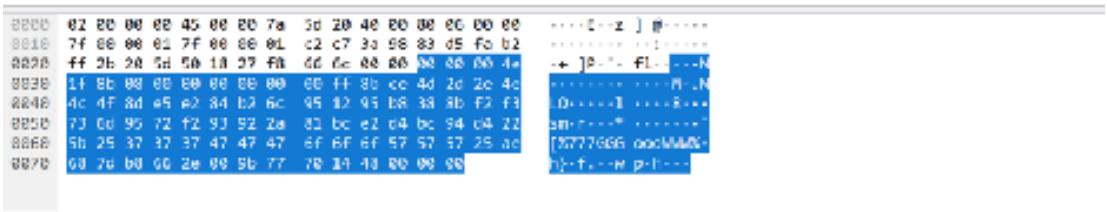


6.3.8 Packet Structure

We now know that the majority of the packet contains compressed data, but there is still one piece of the packet we have not reversed yet. Looking at the packet again, we can see that the data `0x00 00 00 4e` comes before the compressed section:





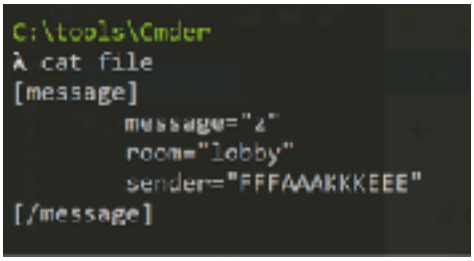


Playrix Ex. 1024, page 341

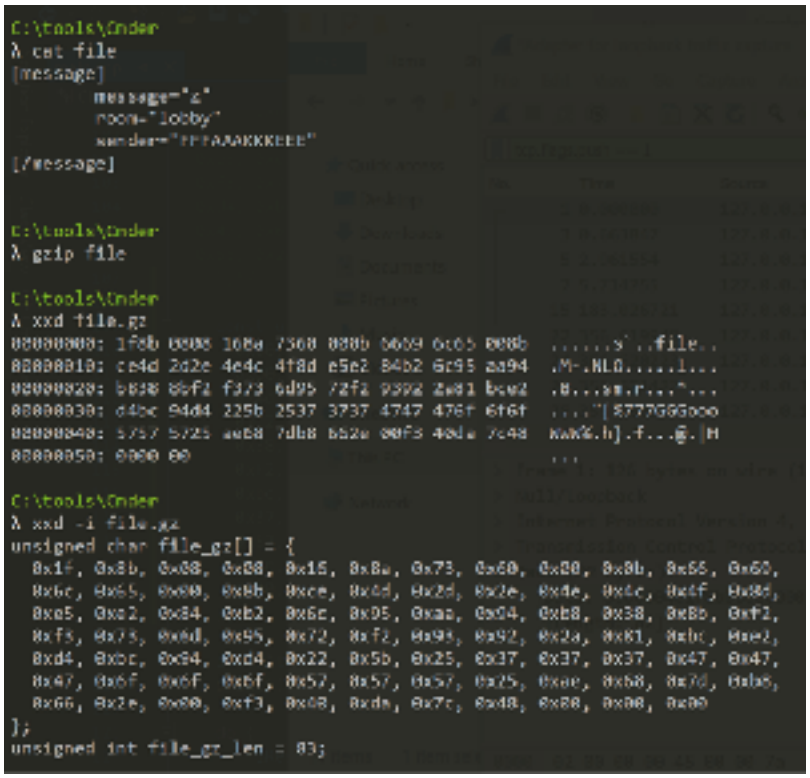
If we convert this `0x4E` into decimal, we get the value 78. Examining the length of the data section of the packet, we see that it is 82. From this we can deduce that the first 4 bytes of the packet are responsible for holding the size of the compressed data. With this, we have all the information we need to create our own packet.

6.3.9 Creating a Packet

Now that we have reversed a packet, we can use the opposite steps to create our own. In this case, let's create a chat message that says *z* from our chat message that said *a*. Take the file produced from our steps above, and change the message to *z*:



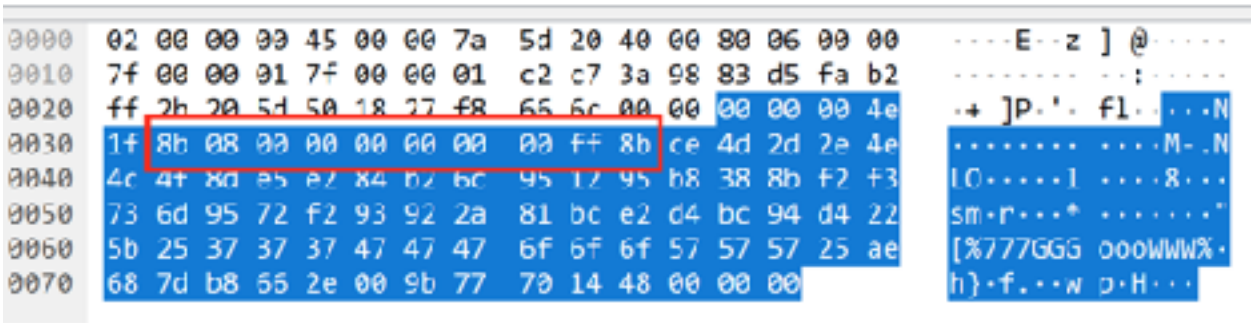
Next, we are going to gzip this file. We can then use xxd to print out its byte representation. By using the **-i** flag, xxd will display this data in a format that we can use in our code:



We can place this data into our code like before:

```
const unsigned char packet_bytes[] = {
    0x1f, 0x8b, 0x08, 0x08, 0x16, 0x8a, 0x73, 0x60, 0x00, 0x0b, 0x66, 0x69,
    0x6c, 0x65, 0x00, 0x8b, 0xce, 0x4d, 0x2d, 0x2e, 0x4e, 0x4c, 0x4f, 0x8d,
    0xe5, 0xe2, 0x84, 0xb2, 0x6c, 0x95, 0xaa, 0x94, 0xb8, 0x38, 0x8b, 0xf2,
    0xf3, 0x73, 0x6d, 0x95, 0x72, 0xf2, 0x93, 0x92, 0x2a, 0x81, 0xbc, 0xe2,
    0xd4, 0xbc, 0x94, 0xd4, 0x22, 0x5b, 0x25, 0x37, 0x37, 0x37, 0x47, 0x47,
    0x47, 0x6f, 0x6f, 0x6f, 0x57, 0x57, 0x57, 0x25, 0xae, 0x68, 0x7d, 0xb8,
    0x66, 0x2e, 0x00, 0xf3, 0x40, 0xda, 0x7c, 0x48, 0x00, 0x00, 0x00
};
```

If you examine the chat messages so far and our newly generated message, you may notice that there is a difference:



All the other chat messages used by the game have `0x00…ff` in between `0x08` and `0x8b`. By contrast, our message has what appears to be random data. To fix this, we can simply replace these bytes with values that we know work from the game:

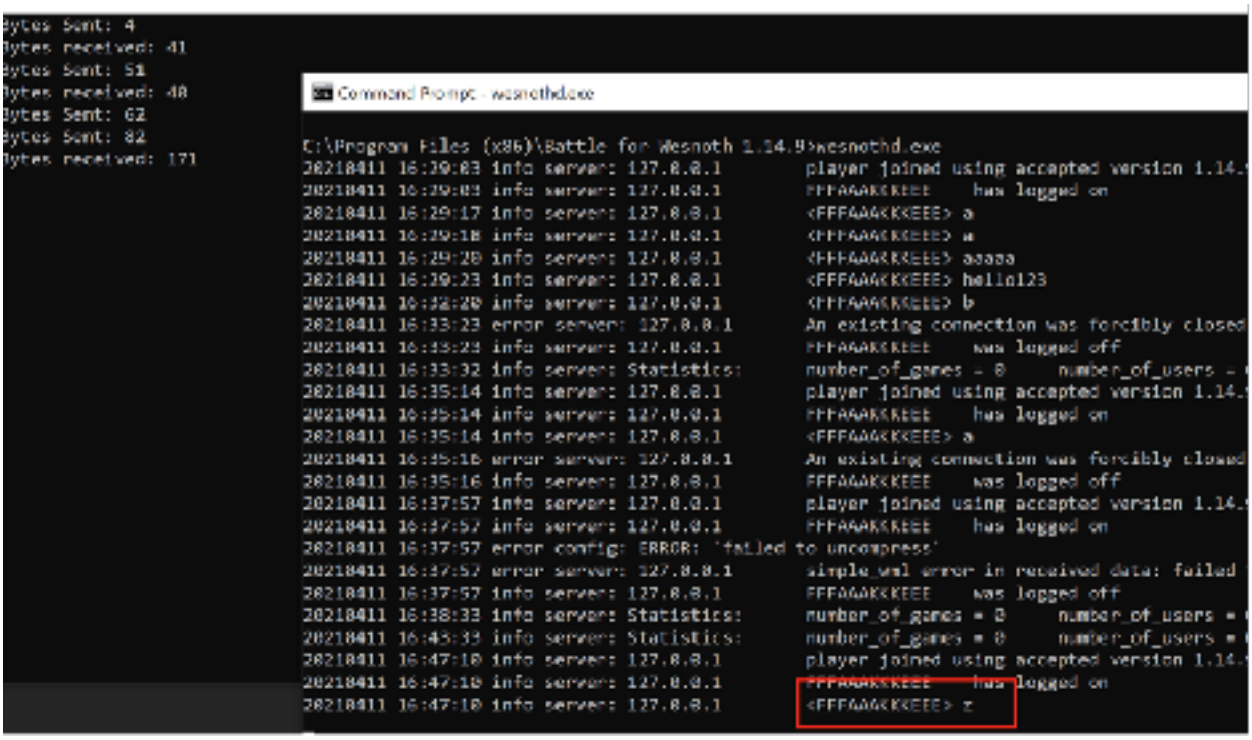
```
const unsigned char packet_bytes[] = {
    0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0x8b, 0xce,
    0x4d, 0x2d, 0x2e, 0x4e, 0x4c, 0x4f, 0x8d, 0xe5, 0xe2, 0x84, 0xb2, 0x6c,
    0x95, 0xaa, 0x94, 0xb8, 0x38, 0x8b, 0xf2, 0xf3, 0x73, 0x6d, 0x95, 0x72,
    0xf2, 0x93, 0x92, 0x2a, 0x81, 0xbc, 0xe2, 0xd4, 0xbc, 0x94, 0xd4, 0x22,
    0x5b, 0x25, 0x37, 0x37, 0x37, 0x47, 0x47, 0x47, 0x6f, 0x6f, 0x6f, 0x57,
    0x57, 0x57, 0x25, 0xae, 0x68, 0x7d, 0xb8, 0x66, 0x2e, 0x00, 0xf3, 0x40,
    0xda, 0x7c, 0x48, 0x00, 0x00, 0x00
};
```

This also has the effect of shortening the data. Finally, we need to add the length to the front of the packet. Since it is a single letter, we know that it will be `0x4e`:

Playrix Ex. 1024, page 343

```
const unsigned char packet_bytes[] = {
    0x00, 0x00, 0x00, 0x4e, 0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0xff, 0x8b, 0xce, 0x4d, 0x2d, 0x2e, 0x4e, 0x4c, 0x4f, 0x8d, 0xe5,
    0xe2, 0x84, 0xb2, 0x6c, 0x95, 0xaa, 0x94, 0xb8, 0x38, 0x8b, 0xf2, 0xf3,
    0x73, 0x6d, 0x95, 0x72, 0xf2, 0x93, 0x92, 0x2a, 0x81, 0xbc, 0xe2, 0xd4,
    0xbc, 0x94, 0xd4, 0x22, 0x5b, 0x25, 0x37, 0x37, 0x37, 0x47, 0x47, 0x47,
    0x6f, 0x6f, 0x6f, 0x57, 0x57, 0x57, 0x25, 0xae, 0x68, 0x7d, 0xb8, 0x66,
    0x2e, 0x00, 0xf3, 0x40, 0xda, 0x7c, 0x48, 0x00, 0x00, 0x00
};
```

With these changes, we can build and execute the code. The resulting program will connect to the server and send the chat message *z*, proving that we now know the structure and can create our own packets:



Playrix Ex. 1024, page 344

6.4 Creating an External Client

6.4.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

6.4.2 Identify

In the previous chapter, we reversed the structure of a Wesnoth chat packet and identified the steps needed to reverse and create chat packets for the game. In this chapter, we will expand on this work to create a chat bot, a type of bot that will wait for and respond to certain commands.

6.4.3 Understand

In the previous chapter, we determined that the process to reverse a packet went like this:

1. Retrieve the packet's data.
2. Split the data into two sections: size and compressed content.
3. Decompress the second section.

We used that technique on a chat packet and retrieved structured data that looked like:

```
[message]
    message="a"
    room="lobby"
    sender="FFFAAAKKKEEE"
[/message]
```

Once we retrieved this data, we could make modifications and use a similar process to create a new packet:

1. Compress the data section.
2. Add the section's length to the front of the data section.
3. Send the packet with the new data.

Since we can write both of the above processes out as a series of concrete steps, we can create a program to automatically perform them for us. In the retrieval process, we can analyze the content of each retrieved packet and look for certain characters. If we identify these characters, we can act on them.

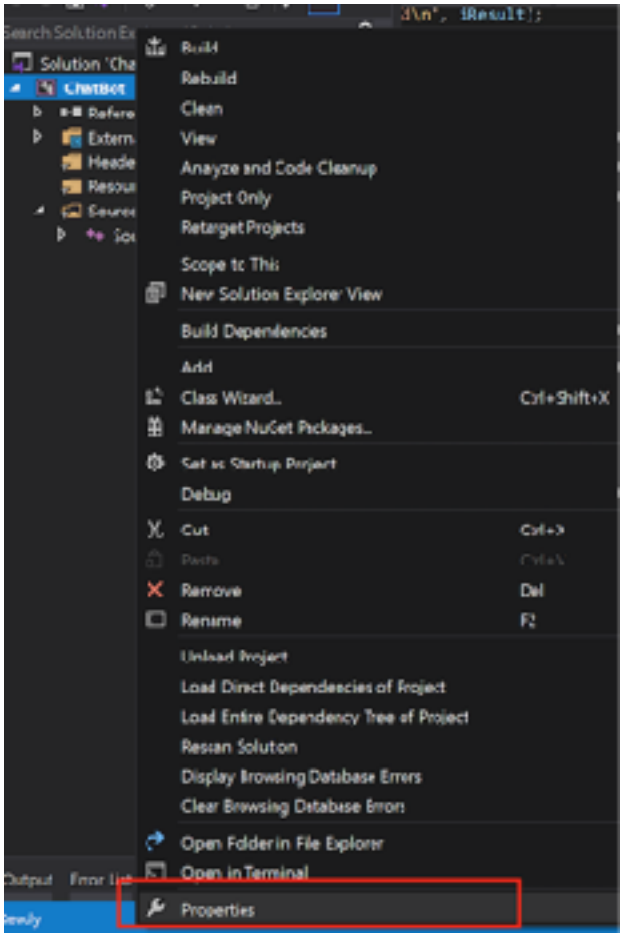
6.4.4 ZLib Installation

The data in the packets was compressed using gzip. While we could write our own functions to manage gzip'd data, there are external libraries that already provide functionality to compress and decompress gzip'd data in C++. External libraries generally contain two parts: header files to include in your code, and library files that contain the actual code. For this chapter, we will use a library called ZLib. Most of these libraries require additional installation steps to fully work, including ZLib.

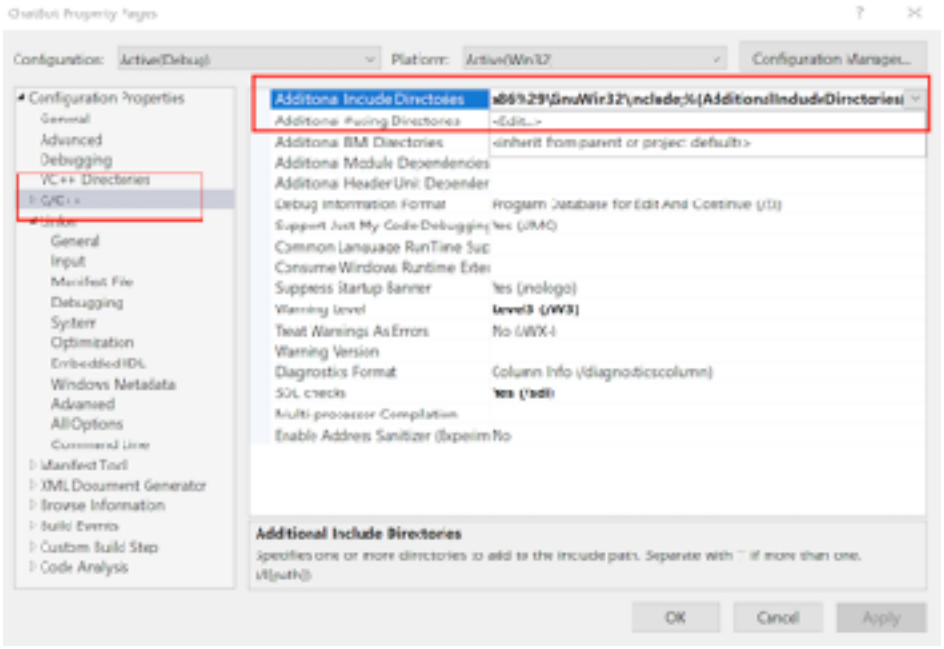
To set up ZLib, first download the *Complete package* installer from their site under the *zlib for Windows 9x/NT* entry. Once installed, it should create a directory at *C:\Program Files (x86)\GnuWin32*.

This installation placed several header and library files in this directory. To use these, we need to include them in our Visual Studio project. Open up Visual Studio and create a project. Once created, right-click on the project file and choose *Properties*:

Playrix Ex. 1024, page 346

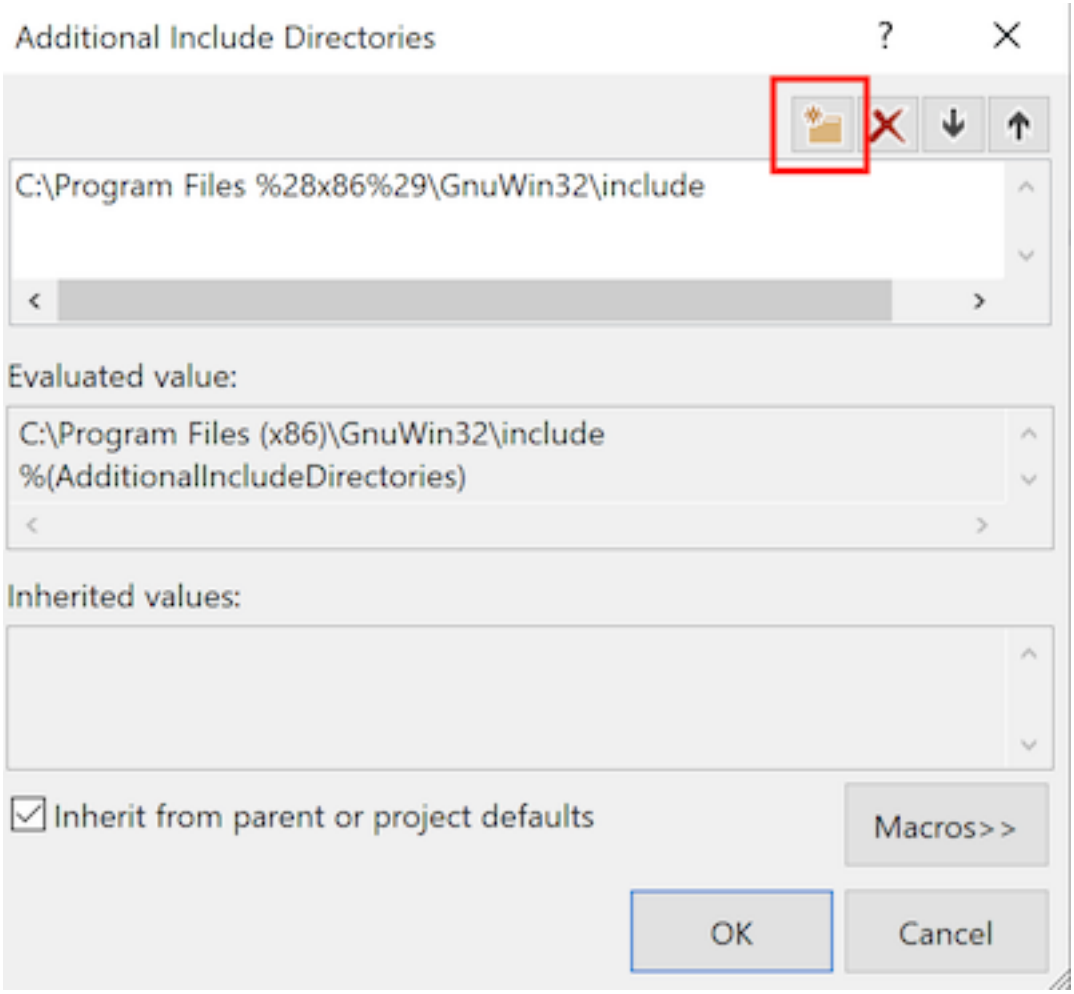


First, we will make sure that our project can find the ZLib include files. In the properties dialog, choose *C/C++ -> Additional Include Directories -> Edit*:



Playrix Ex. 1024, page 347

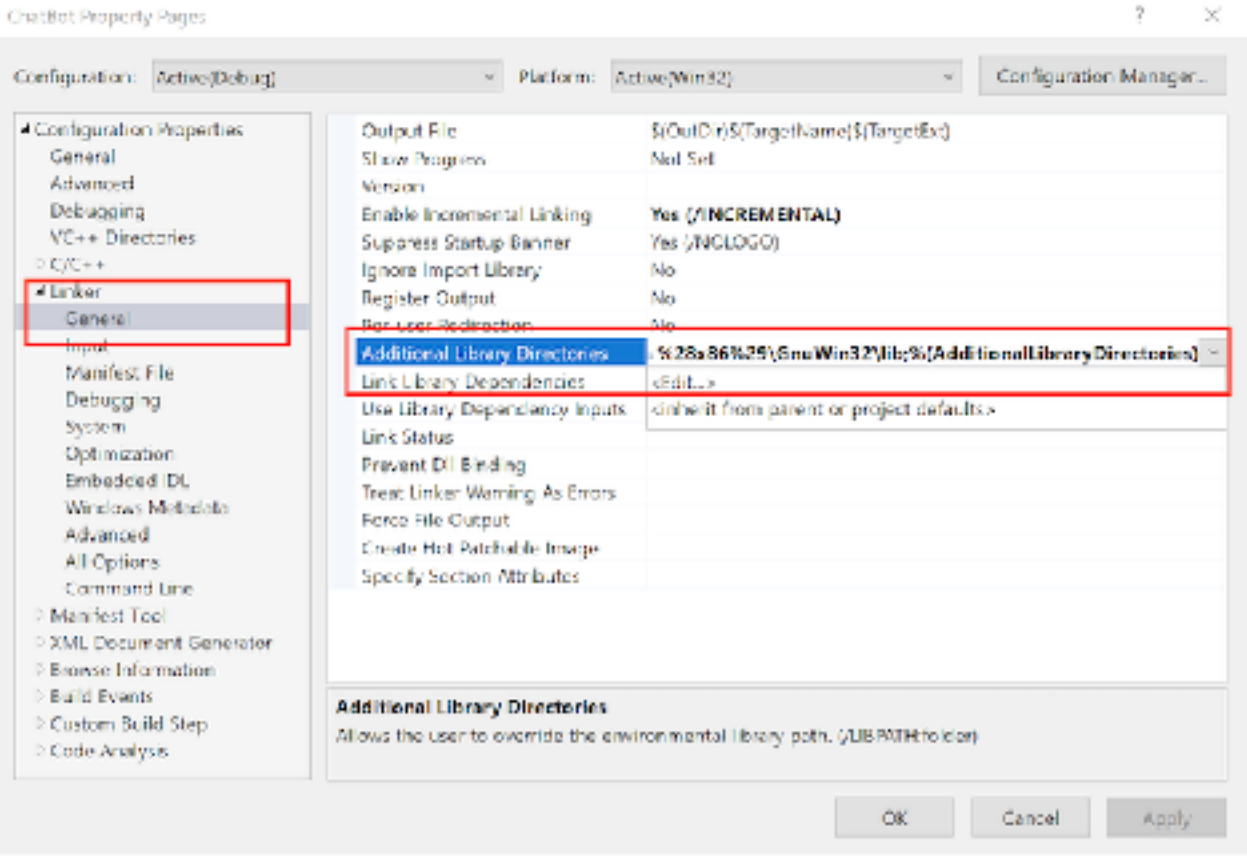
In the dialog box that appears, choose the *New folder* icon and browse to *C:\Program Files (x86)\GnuWin32\include*:



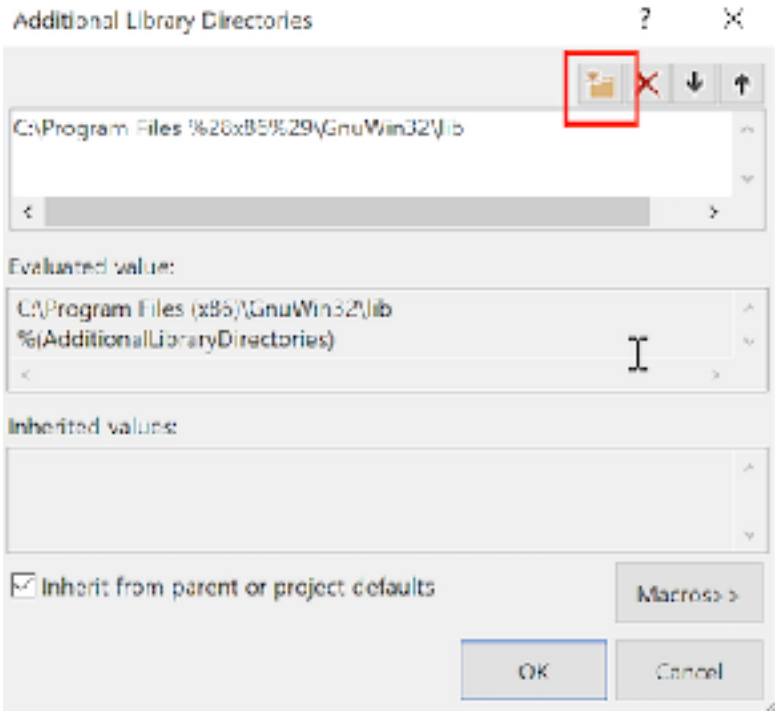
We can now include *zlib.h* inside our code, and the build process will be able to locate the file. However, as we have seen in previous chapters, header files generally only contain function definitions and do not contain the majority of code. In ZLib, this code is stored in library files. We will include these library files in a similar manner.

Select *Linker -> General* and then select *Additional Library Directories -> Edit*:

Playrix Ex. 1024, page 348

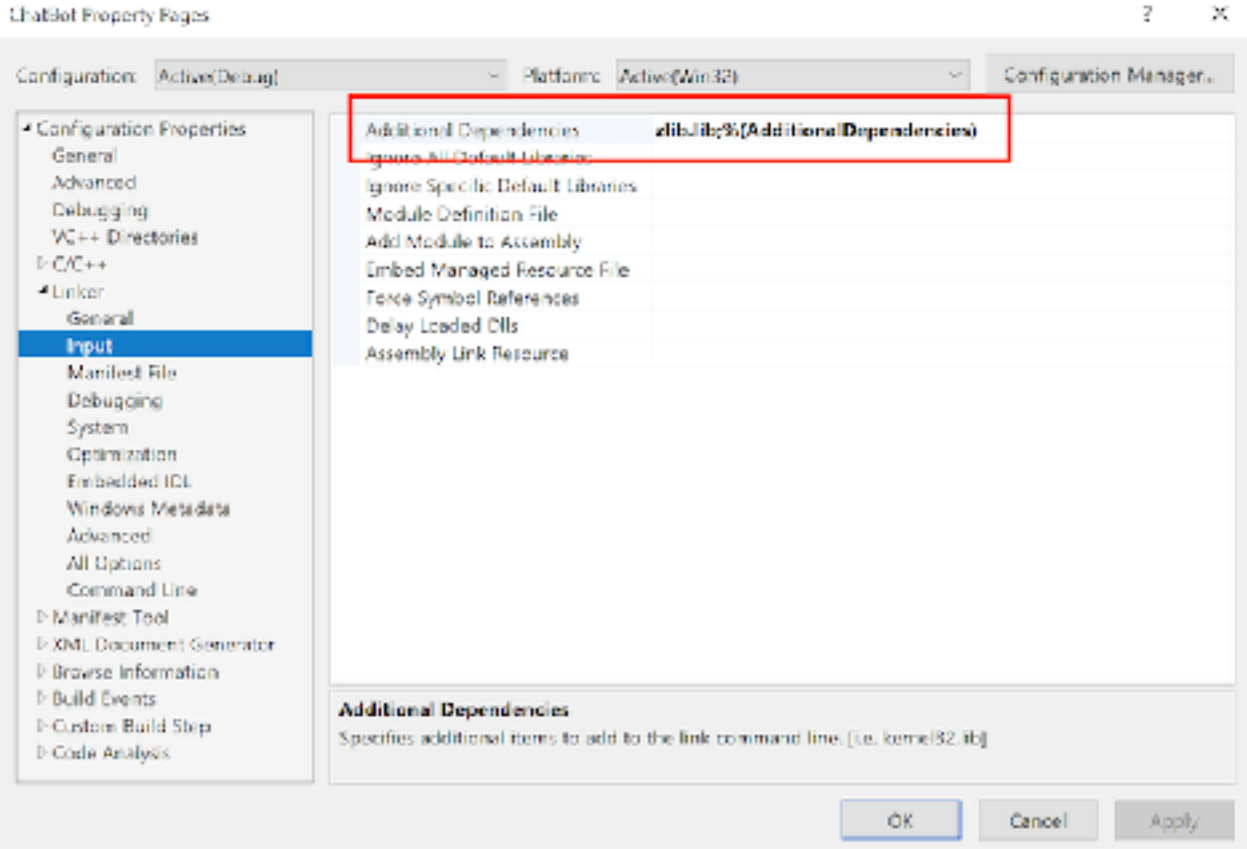


Like we just did, choose the *New folder* icon. This time, supply the path to the library folder at *C:\Program Files (x86)\GnuWin32\lib*:



Playrix Ex. 1024, page 349

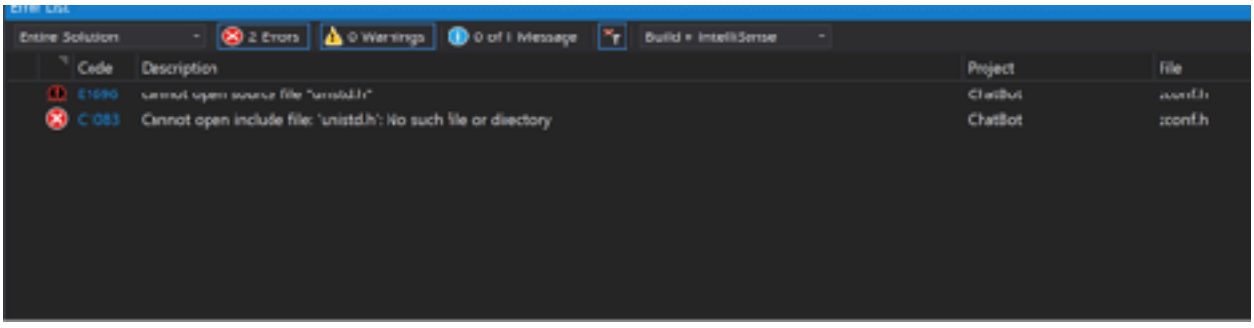
Now that we have loaded the library folder, we need to include the actual .lib file. In *Linker -> Input*, select *Additional Dependencies* and add *zlib.lib* in the dialog that appears:



With all this setup done, we can now include ZLib in our code like a regular header:

```
#include <zlib.h>
```

However, if we try to build this code, we will get the following build error:

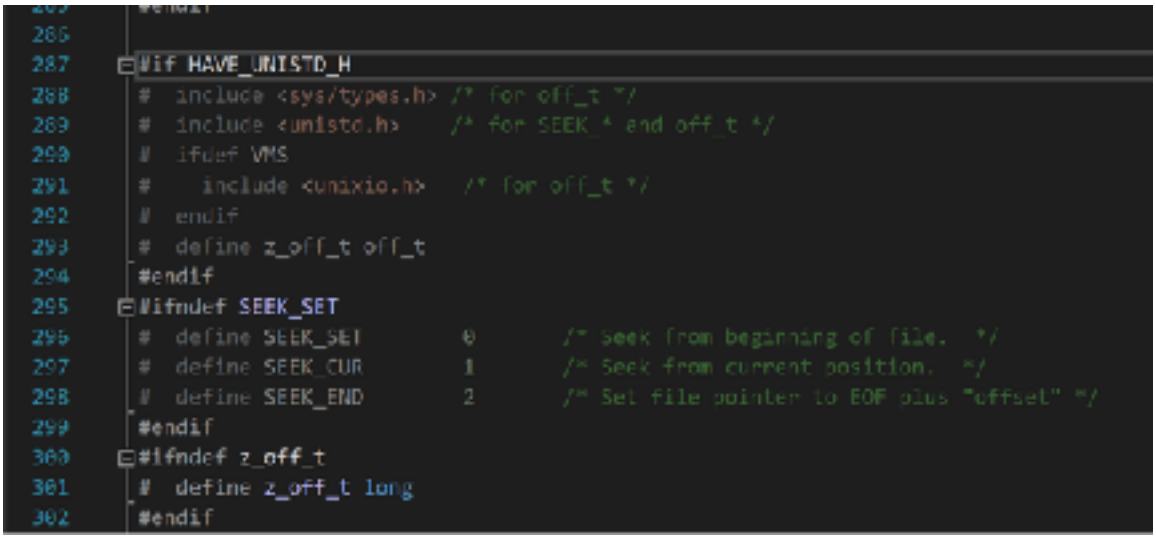


Playrix Ex. 1024, page 350

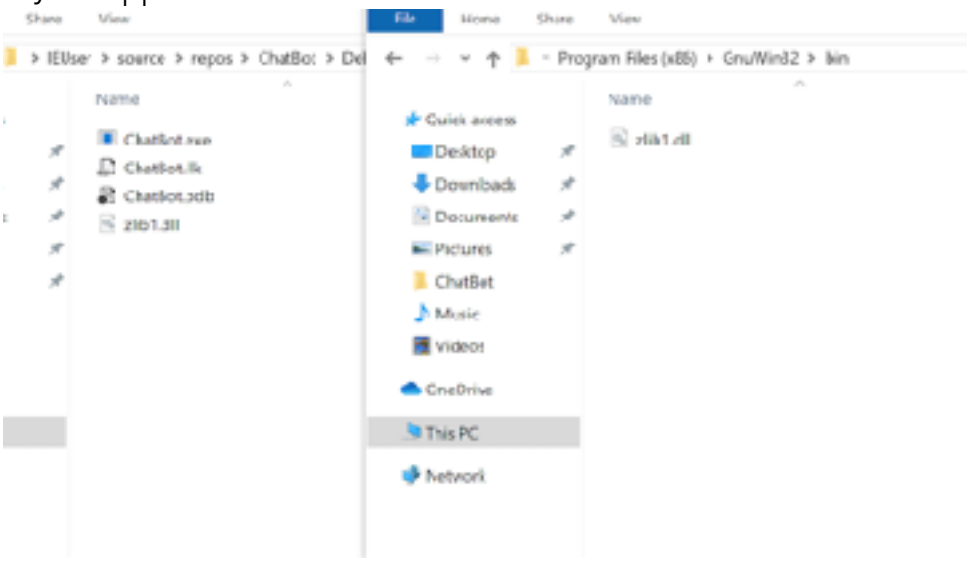
If we double-click on this error, we are directed to the following code block:

```
#if 1           /* HAVE_UNISTD_H -- this line is updated by ./configure */
#  include <sys/types.h> /* for off_t */
#  include <unistd.h>    /* for SEEK_* and off_t */
#  ifdef VMS
```

unistd.h is a Unix specific file. To fix this error, we can change the **#if 1** to **#if HAVE_UNISTD_H**, like so:



With these changes, our program will now build. However, if you run the program, you will encounter a missing DLL error. To fix this, copy over the *zlib1.dll* file to the running directory of your application:



Playrix Ex. 1024, page 351

We can now build and run programs that contain ZLib functionality.

6.4.5 Sending Data

Now that we can compress data in our code, we can create a function to send a Wesnoth-structured packet with whatever data we would like. To create the packet, this function will need our data and the length of this data. To send the packet, we will need a socket. With these requirements, we can create our function definition:

```
void send_data(const unsigned char *data, size_t len, SOCKET s) {
```

Based on the documentation, the easiest way to use ZLib to produce gzip'd data is to create a file with the compressed data. We can do this using the **gzopen**, **gzwrite**, and **gzclose** functions. These are similar to the regular file functions **fopen**, **fwrite**, and **fclose**. In our case, we will create a single compressed file, **packet.gz**, and write whatever data is passed into this file:

```
gzFile temp_data = gzopen("packet.gz", "wb");
gzwrite(temp_data, data, len);
gzclose(temp_data);
```

We can test our current implementation via:

```
const unsigned char version[] = "[test]hello[/test]";
send_data(version, sizeof(version), ConnectSocket);
```

Running this code will produce a packet.gz file in the same directory that you ran the program from. If you use gzip to decompress the packet.gz file, you will find that it contains **[test]hello[/test]**, showing that our code works so far.

Our packet needs to contain the byte representation of this file. To retrieve this, we can read the file as a binary file:

```
#define DEFAULT_BUFLEN 512
...
FILE* temp_file = NULL;
fopen_s(&temp_file, "packet.gz", "rb");
```


Playrix Ex. 1024, page 352

```
if (temp_file) {
    size_t compress_len = 0;
    unsigned char buffer[DEFAULT_BUFLEN] = { 0 };
    compress_len = fread(buffer, 1, sizeof(buffer), temp_file);
    fclose(temp_file);
```

If you run this code and set a breakpoint after **fclose**, you will see that the buffer now contains the byte representation of the compressed file. We can use this to build our packet. We know that the first 4 bytes (a **DWORD**) of the packet represent the packet's size. Since all of the chat packets are small, we only need to write the size to the last byte of this **DWORD**. Since buffers start at 0 in C, we reference this position with **+3**. We will write the buffer containing the compressed data after that:

```
unsigned char buff_packet[DEFAULT_BUFLEN] = { 0 };
memcpy(buff_packet + 3, &compress_len, sizeof(compress_len));
memcpy(buff_packet + 4, buffer, compress_len);
```

Next, we will use the code we have seen before to send a packet containing this data:

```
int iResult = send(s, (const char*)buff_packet, compress_len + 4, 0);
printf("Bytes Sent: %ld\n", iResult);
```

To verify that this method works, we will build off the code we wrote in the previous chapter. In this code, we sent three packets to connect to the server: an initial negotiation packet that contained 0's, a packet containing our client's version, and a packet containing our username.

We can use the same technique that we used to decode chat messages to decode these packets. For example, the packet containing our version looks like this after decoding:

```
[version]
    version="1.14.9"
[/version]
```

Instead of sending the packet's bytes like we were doing, we can use our new function:

```
const unsigned char version[] = "[version]\nversion=\"1.14.9\"\n[/version]";
send_data(version, sizeof(version), ConnectSocket);
```


Playrix Ex. 1024, page 353

If you build and run this code, you will see that our bot will connect in the same way, verifying that our function works. We can build on this approach to send a custom username:

```
const unsigned char name[] = "[login]\nusername=\"ChatBot\"\n[/login]";
send_data(name, sizeof(name), ConnectSocket);
```

Finally, we can use the same method to send an initial chat message when we connect:

```
const unsigned char first_message[] = "[message]\nmessage=\"ChatBot
connected\"\nroom=\"lobby\"\nsender=\"ChatBot\"\n[/message]";
send_data(first_message, sizeof(first_message), ConnectSocket);
```

This is the same chat message structure that we observed in the previous chapter.

6.4.6 Retrieving Data

We can now send chat messages. For our chatbot to work, we also need to retrieve messages from the server and parse them for certain text. For this chapter, we will have our bot respond to any message that contains **\wave** with a chat message that says **hello** back.

At the bottom of the code from Microsoft is a loop that continuously checks for new packets. We can modify this to send retrieved packets to our own function, and, depending on the contents, send a chat message:

```
do {
    iResult = recv(ConnectSocket, (char*)recvbuf, recvbuflen, 0);
    if (iResult > 0)
        printf("Bytes received: %d\n", iResult);
    else if (iResult == 0)
        printf("Connection closed\n");
    else
        printf("recv failed with error: %d\n", WSAGetLastError());

    if (parse_data(recvbuf, iResult)) {
        const unsigned char message[] = "[message]\nmessage=\"Hello!
\"\nroom=\"lobby\"\nsender=\"ChatBot\"\n[/message]";
        send_data(message, sizeof(message), ConnectSocket);
    }
} while (iResult > 0);
```


Playrix Ex. 1024, page 354

Our **parse_data** function will take a data buffer and its length, and return true if **\wave** is found.

```
bool parse_data(unsigned char *buff, int buff_len) {
```

Like we described above, this function will do the same steps as the **send_data** function, but in the opposite order. First, we will extract the compressed data from the packet and write it to a file:

```
unsigned char data[DEFAULT_BUFLEN] = { 0 };
memcpy(data, buff + 4, buff_len - 4);

FILE* temp_file = NULL;
fopen_s(&temp_file, "packet_recv.gz", "wb");

if (temp_file) {
    fwrite(data, 1, sizeof(data), temp_file);
    fclose(temp_file);
}
```

With the compressed data saved, we can use the **gzopen** and **gzread** to read the decompressed data into a variable. We will then write this variable to the terminal using **fwrite**:

```
gzFile temp_data_in = gzopen("packet_recv.gz", "rb");
unsigned char decompressed_data[DEFAULT_BUFLEN] = { 0 };
gzread(temp_data_in, decompressed_data, DEFAULT_BUFLEN);
fwrite(decompressed_data, 1, DEFAULT_BUFLEN, stdout);
gzclose(temp_data_in);
```

Finally, we will check if the data contains the text **\wave** using **strstr**. This function returns a positive value if the second parameter is included in the first parameter. It returns 0 if not. Because of this, we can return the value of the search and use that to signify to our calling code that the text was found:

```
return strstr((const char*)decompressed_data, (const char*)”\\wave");
```

The full source code for this chapter is available in Appendix A for comparison.

Playrix Ex. 1024, page 355

6.5 Proxying TCP Traffic

6.5.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

6.5.2 Overview

In the previous chapter, we created an external client that would connect to a Wesnoth server and listen for and respond to specific chat messages. A major downside to this approach is that we had to reverse the entire authentication process so that our client could connect to the server. Our goal in this chapter is to create a proxy. This will allow us to use a regular game client and only intercept and modify the traffic we care about from the client.

6.5.3 Reason for Proxying

The best way to understand our purpose for creating a proxy is to observe the network traffic when connecting to a server with two clients on the same host. On your lab machine, start a Wesnoth server and connect to it with a legitimate copy of the game. Next, modify the client that we wrote in the previous chapter and remove all the authentication. Instead, have it simply send a chat message:

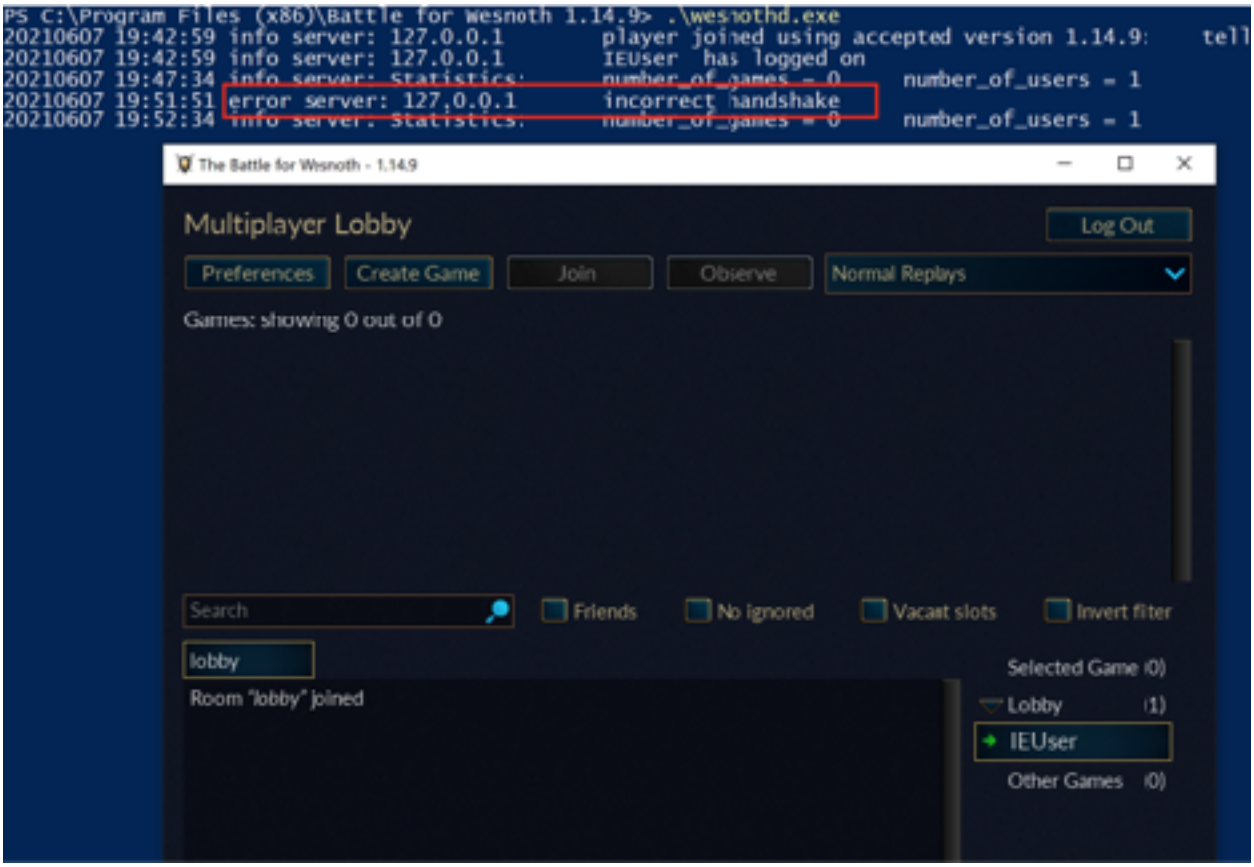
```
...
freeaddrinfo(result);

if (ConnectSocket == INVALID_SOCKET) {
    printf("Unable to connect to server!\n");
    WSACleanup();
    return 1;
}
```


Playrix Ex. 1024, page 356

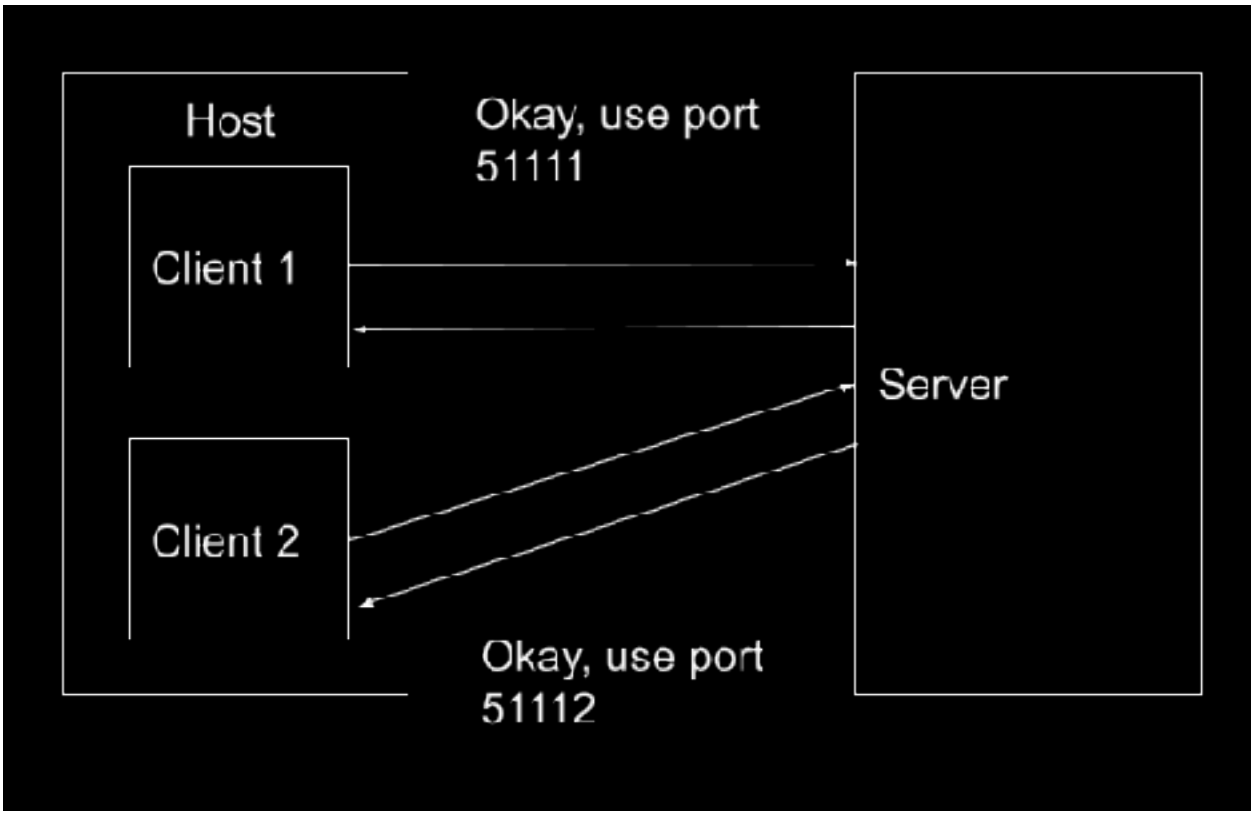
```
const unsigned char first_message[] = "[message]\nmessage=\"ChatBot
connected\"\nroom=\"lobby\"\nsender=\"ChatBot\"\n[/message]";
send_data(first_message, sizeof(first_message), ConnectSocket);
…
```

Finally, open up Wireshark and start monitoring for traffic on the local adapter, as we have discussed in previous chapters. When you start the modified external client, you should see the following error on the server:

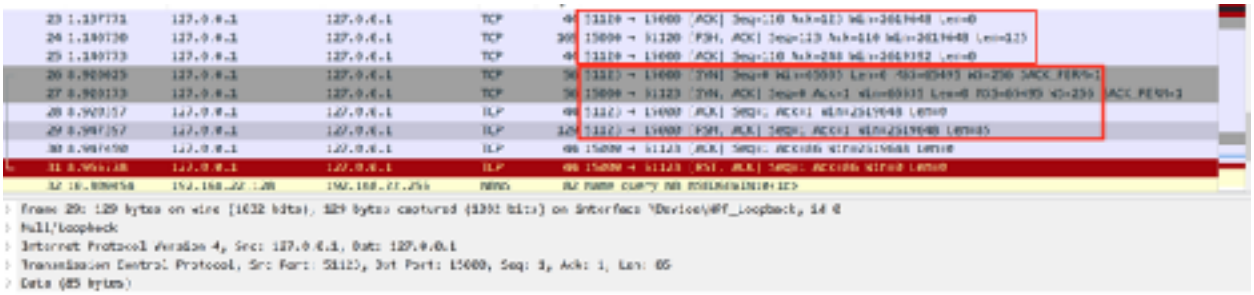


We are familiar with this error from our initial analysis, so we know that it occurs when we do not provide a valid handshake. Even though both our game and external client are running on the same machine, they have different sockets and are treated as completely separate connections by the server:

Playrix Ex. 1024, page 357



If we examine the Wireshark traffic from the game client and our external client, we can observe this behavior. While the game is busy sending traffic on port 51120, we can see the TCP handshake of our external client on port 51123:

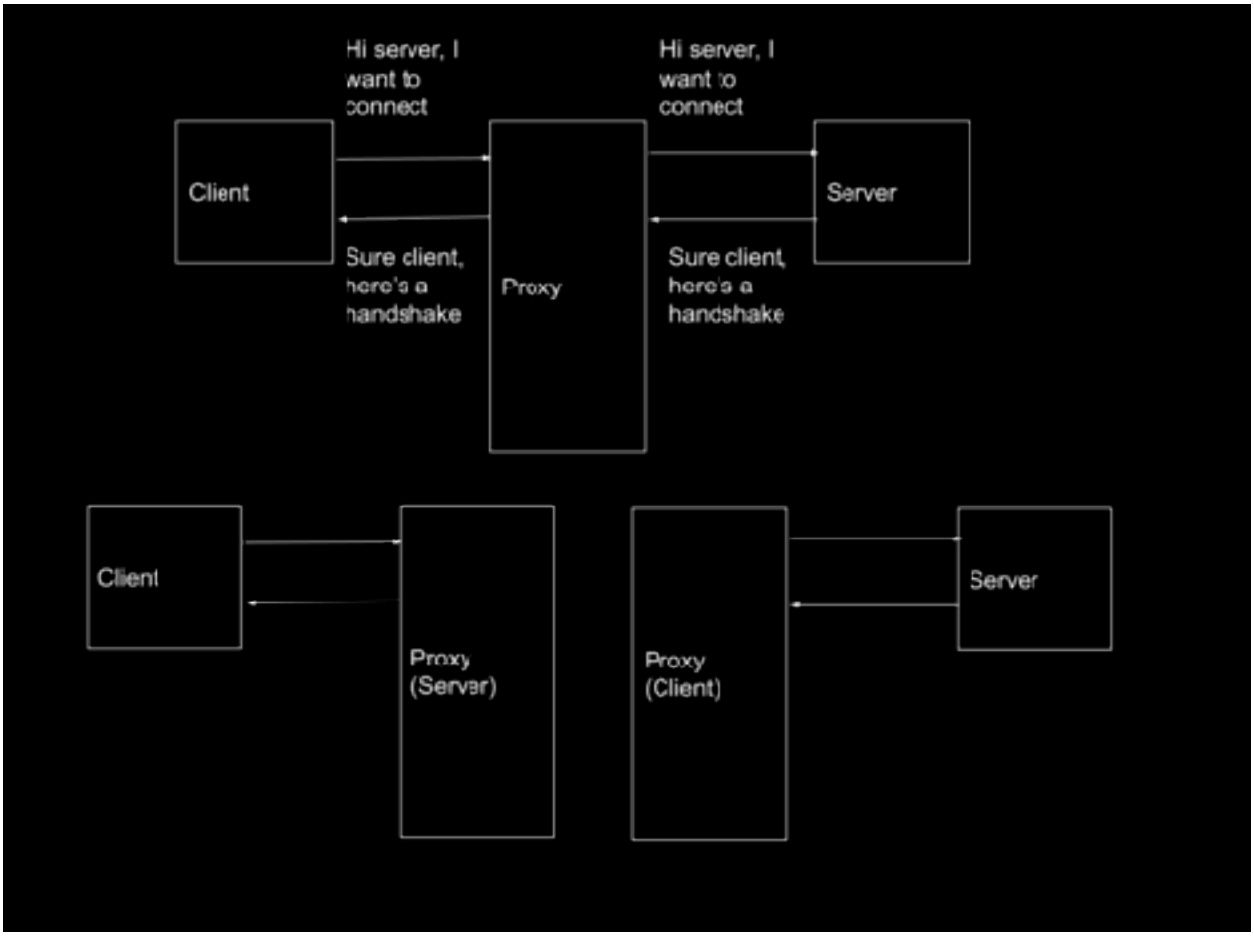


If we want to intercept and modify a game's client traffic, we will need to use a different approach.

6.5.4 Proxying Traffic

When using TCP, we know that a connection is established between a client and server after completing a handshake. To intercept and inject our own traffic into this connection, the easiest approach is to be a man-in-the-middle (MitM) of this connection.

At this position, we can modify requests from the client before they are sent to the server, as well as responses from the server before they are sent to the client. This is commonly known as a proxy, or an agent that simply relays traffic from a source to a destination. A visualization of this model is shown below:



The key to this model is that our proxy is acting as the server to the client and the client to the server. This means that the server completes a handshake with our proxy, allowing us to inject whatever traffic we want. Since we are forwarding legitimate traffic from the client, we do not need to reverse traffic (such as the authentication mechanism) because the client will handle that for us.

Playrix Ex. 1024, page 359

A proxy consists of three main sections of code:

1. A socket to listen for client traffic
2. A socket to send traffic to the server
3. Logic to relay traffic from the client to the server and the server to the client

To simplify this chapter, we will create a proxy that will respond to the *\wave* event, identical to the external client we wrote in the previous chapter. For these operations, the Wesnoth client must send data to the server for the server to respond. When using a proxy for other operations, additional logic will need to be added in to pass server traffic to the client.

The full code for this chapter is available in Appendix A.

6.5.5 Listening for Client Traffic

To listen for client traffic, we will create a listen socket. Once we receive a connection, we will establish a connection with the client. To do this, we can build off the Microsoft server example:

```
#define DEFAULT_PORT "27015"

WSADATA wsaData;
int iResult;

SOCKET ListenSocket = INVALID_SOCKET;
SOCKET ClientSocket = INVALID_SOCKET;

struct addrinfo* result = NULL,
    hints;

iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;

iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
ListenSocket = socket(result->ai_family, result->ai_socktype, result-
>ai_protocol);
```


Playrix Ex. 1024, page 360

```
iResult = bind(ListenSocket, result->ai_addr, (int)result->ai_addrlen);
freeaddrinfo(result);

iResult = listen(ListenSocket, SOMAXCONN);
ClientSocket = accept(ListenSocket, NULL, NULL);
closesocket(ListenSocket);
```

This will create a socket on port 27015 that will accept a single connection.

6.5.6 Sending Traffic to Server

For sending traffic to the server, we can build off the code that we already discussed in the previous chapter:

```
SOCKET ServerSocket = INVALID_SOCKET;

ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;

iResult = getaddrinfo("127.0.0.1", "15000", &hints, &result);
ServerSocket = socket(result->ai_family, result->ai_socktype, result-
>ai_protocol);
iResult = connect(ServerSocket, result->ai_addr, (int)result->ai_addrlen);
freeaddrinfo(result);
```

Like we discussed above, our proxy will forward client packets to the server and server packets to the client. However, not all client packets will require a response from the server. For example, in Wesnoth, sending a chat message does not require a response back. To ensure that our proxy does not get stuck waiting for a server response, we need to set a timeout on the server socket. This timeout will cause any **recv** calls to fail after a set amount of time:

```
DWORD timeout = 1000;
setsockopt(ServerSocket, SOL_SOCKET, SO_RCVTIMEO, (char*)&timeout,
sizeof(timeout));
```


Playrix Ex. 1024, page 361

6.5.7 Relaying Traffic

With both of our sockets created, we can now focus on relaying the traffic between the client and server. Since Wesnoth does not send out server responses unless a client sends a request, our proxy can be simplified to the following events:

1. Wait for a request from the client.
2. Send that request to the server.
3. Wait for a response from the server.
4. If a response comes back, send to the client. Otherwise, start waiting for the next client request.

After each event, our program will sleep for a short period to ensure that the traffic between the client and server does not get desynchronized. First, we will wait for a request from the client:

```
#define DEFAULT_BUFLEN 512

int iSendResult;
unsigned char recvbuf[DEFAULT_BUFLEN];
int recvbuflen = DEFAULT_BUFLEN;

do {
        iResult = recv(ClientSocket, (char*)recvbuf, recvbuflen, 0);
        Sleep(100);
```

If we retrieve a request, we pass this data to the server:

```
if (iResult > 0) {
    printf("Bytes received: %d\n", iResult);

    iSendResult = send(ServerSocket, (char*)recvbuf, iResult, 0);
    Sleep(100);
    printf("Bytes sent: %d\n", iSendResult);
```

Next, we wait for a response from the server. If a response actually comes back, we forward this on to the client:

```
iResult = recv(ServerSocket, (char*)recvbuf, recvbuflen, 0);
Sleep(100);
```


Playrix Ex. 1024, page 362

```
if (iResult != SOCKET_ERROR) {
    iSendResult = send(ClientSocket, (char*)recvbuf, iResult, 0);
    Sleep(100);
}
```

Finally, we continue the loop while we have a result, or if we have a timeout from the server:

```
} while (iResult > 0 || WSAGetLastError() == WSAETIMEDOUT);
```

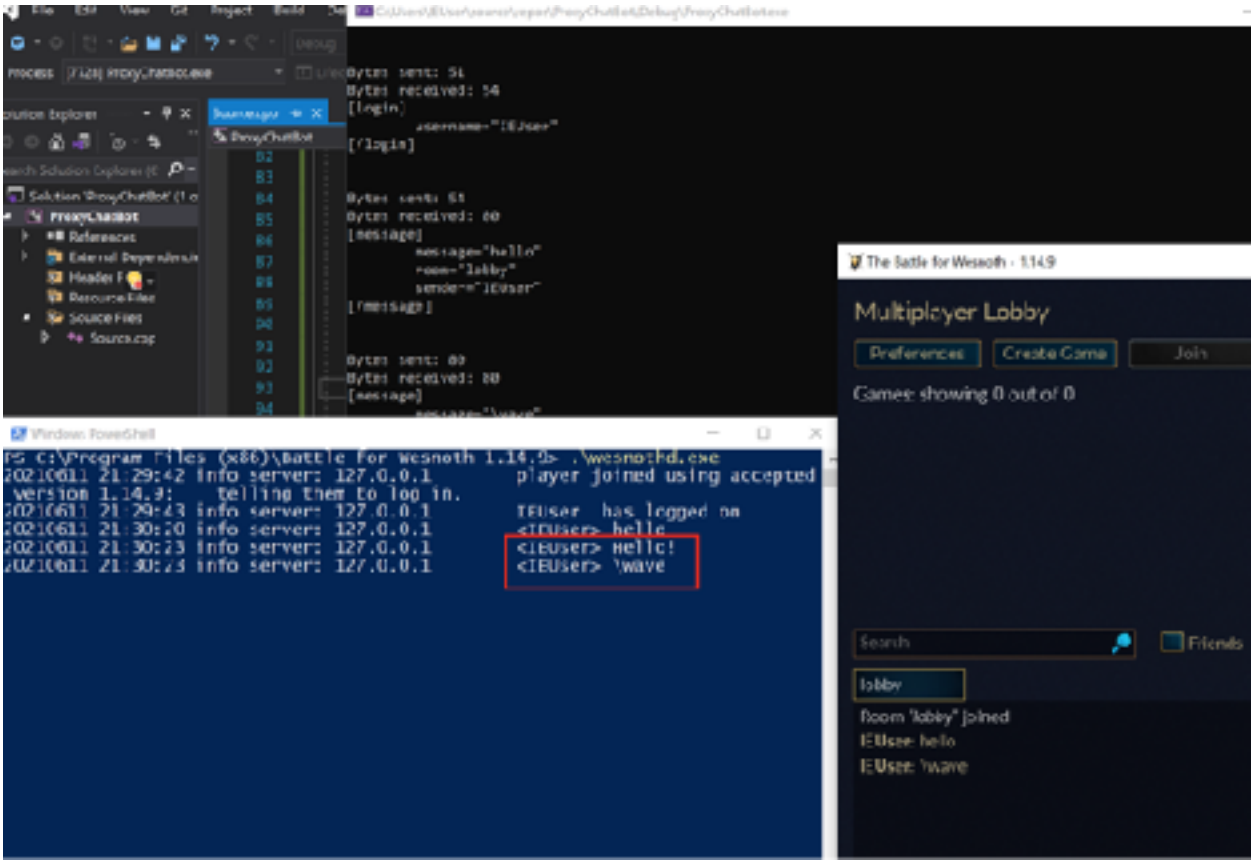
At this point, our proxy will properly pass traffic from a client to a server. We can verify this by running the proxy, connecting to it in Wesnoth by setting the server to localhost:27015, and confirming that we can connect to the actual server running on localhost:15000.

As a proof-of-concept, we can now add in logic to intercept and modify traffic. First, we can import our **parse_data** and **send_data** functions from our last chapter. Next, we will modify our main loop to check any client requests and see if they contain the chat message **\wave**. If so, we will send an additional packet with a **Hello!** message:

```
if (iResult > 0) {
    printf("Bytes received: %d\n", iResult);
    if (parse_data(recvbuf, iResult)) {
        const unsigned char message[] = "[message]\nmessage=\"Hello!
\"\nroom=\"lobby\"\nsender=\"ChatBot\"\n[/message]";
        send_data(message, sizeof(message), ServerSocket);
        Sleep(100);
    }
```

If you connect to the proxy now and send the chat message **\wave**, you will see that an additional message appears on the server, indicating that we successfully injected traffic into the connection:

Playrix Ex. 1024, page 363


Multiplayer Lobby
Preferences
Create Game
Join
Games showing 0 out of 0
Search
Friends
lobby
Room 'lobby' joined
EUser: hello
EUser: \wave
<EUser> Hello!
<EUser> \wave

Part 7
Tool Development

Playrix Ex. 1024, page 365

7.1 DLL Injector

7.1.1 Target

When writing a DLL injector, it is helpful to have an already working DLL for a particular target. For this chapter, we will use the memory wallhack we produced in Chapter 5.2. While our injector will be built for the game Urban Terror, we will be able to easily modify it for other targets in the future.

7.1.2 Overview

In previous chapters, we used Windows' AppInit functionality to inject DLL's into game executables. While this approach works well for testing, it has several drawbacks:

- AppInit_DLLs needs to be updated for each new DLL.
- AppInit_DLLs are injected into every started process.
- Secure Boot has to be disabled.
- AppInit_DLLs will only be injected into processes that load user32.dll.
- DLL's are loaded into the process at a set time, outside of our control.

To get around these drawbacks, we will write an injector, which will manually load our DLL into the game executable.

7.1.3 Concepts

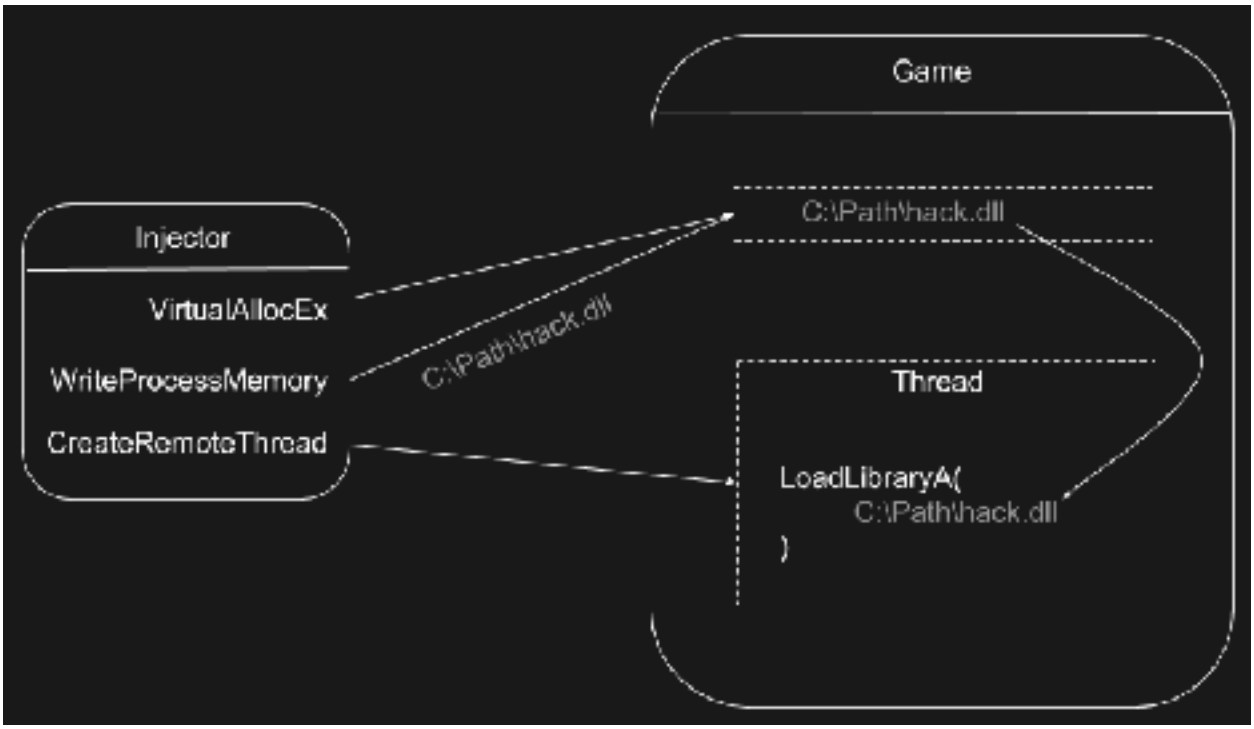
To load static and dynamic libraries, Windows executables can use the LoadLibraryA API function. This function takes a single argument, which is a full path to the library to load.

```
HMODULE LoadLibraryA(
    LPCSTR lpLibFileName
);
```

If we call **LoadLibraryA** in our injector's code, the DLL will be loaded into our injector's memory. Instead, we want our injector to force the game to call **LoadLibraryA**. To do

this, we will use the CreateRemoteThread API to create a new thread in the game. This thread will then execute **LoadLibraryA** inside the game's running process.

However, since the thread is running inside the game's memory, **LoadLibraryA** will not be able to find the path of our DLL specified in our injector. To get around this, we have to write our DLL's path into the game's memory. To ensure that we do not corrupt any other memory, we will also need to allocate additional memory inside the game using VirtualAllocEx. The full breakdown of this interaction looks like:



As we know from previous chapters, we will need a process handle to interact with an external process. For example, in Chapter 3.2, we used **FindWindow** and **GetWindowThreadProcessId** to retrieve a process identifier. This approach has many drawbacks and is not recommended beyond quick testing. Instead, we will use **CreateToolhelp32Snapshot**.

7.1.4 Process Identifier

To use **WriteProcessMemory**, we will need a handle to the Urban Terror process. Instead of using **FindWindow** like we did previously, we will use CreateToolhelp32Snapshot. This API takes a snapshot of all the currently running processes on the machine. Each process in this snapshot can then be examined using

Process32First and Process32Next. Microsoft provides a good example of how to do this here.

While Microsoft's example iterates all processes and dumps their loaded modules, we are only interested in finding a single process and retrieving its process identifier. Therefore, we can simplify their example code:

```
#include <windows.h>
#include <tlhelp32.h>

int main(int argc, char** argv) {
      HANDLE snapshot = 0;
      PROCESSENTRY32 pe32 = { 0 };

      pe32.dwSize = sizeof(PROCESSENTRY32);
      snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
      Process32First(snapshot, &pe32);

      do {

      } while (Process32Next(snapshot, &pe32));

    return 0;
}
```

Each process entry contains two fields that we care about: **szExeFile** and **th32ProcessID**. The former contains the name of the process, like svchost.exe or notepad.exe. The latter contains the process identifier of the process that we can pass to **OpenProcess**.

The process name of Urban Terror is **Quake3-UrT.exe**. This can be identified by viewing the process list in Task Manager while Urban Terror is running. To compare this value to the value in **szExeFile**, we can use the function **strcmp**. This function takes two strings and returns 0 if they match:

```
do {
    if (wcscmp(pe32.szExeFile, L"Quake3-UrT.exe") == 0) {

    }
}
```

7.1.5 Process Handle

When these strings match, we know that **pe32.th32ProcessID** must contain the process identifier for the running instance of Urban Terror. We can pass this value to **OpenProcess** just like we did in previous chapters:

```
HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, true, pe32.th32ProcessID);
```

7.1.6 Allocating Memory

Next, we need to allocate memory inside of Urban Terror to store the full path of our DLL. To do this, we will use **VirtualAllocEx**, which is defined as:

```
LPVOID VirtualAllocEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

Going through the arguments, **hProcess** will be the process handle we obtained from **OpenProcess**. **lpAddress** will be **NULL**, since we do not care where the address is allocated. **dwSize** will be the length of the path to our DLL. Since we want to allocate memory and have it be usable, we will choose **MEM_COMMIT** as the allocation type. Finally, since we want to write to the allocated memory, we will specify the protection as **PAGE_READWRITE**.

VirtualAllocEx will return a **void** pointer containing the address that our memory is allocated at. Since we will need this value for our next call to **WriteProcessMemory**, we will have to create a variable for it. We will also need to create a variable for the full path of our DLL. Due to how C++ interprets backslashes, we need to use two \'s for each single backslash. With all these parameters worked out, we can add the following code:

```
const char *dll_path = "C:\\Users\\IEUser\\source\\repos\\wallhack\\Debug\
\\wallhack.dll";
...
```


Playrix Ex. 1024, page 369

```
void *lpBaseAddress = VirtualAllocEx(process, NULL, strlen(dll_path) + 1,
MEM_COMMIT, PAGE_READWRITE);
```

7.1.7 Writing the DLL Name

With our memory now allocated, we can write our DLL name into Urban Terror's memory using **WriteProcessMemory**. The base address for writing will be the address that we retrieved from **VirtualAllocEx**:

```
WriteProcessMemory(process, lpBaseAddress, dll_path, strlen(dll_path) + 1,
NULL);
```

7.1.8 Creating the Thread

With our DLL's path written into the game's memory, we can create a thread to execute **LoadLibraryA** to load the DLL into the game. We will use **CreateRemoteThread** to create the thread, but first, we need to obtain the address of **LoadLibraryA**.

LoadLibraryA exists inside kernel32.dll. Windows takes care of loading this DLL into all processes that need any API contained inside kernel32.dll. To obtain the address of **LoadLibraryA**, we can use GetProcAddress. This API requires a handle to the DLL that contains the function, in this case kernel32.dll. We can get this handle using GetModuleHandle:

```
HMODULE kernel32base = GetModuleHandle(L”kernel32.dll");
```

Now we can use **CreateRemoteThread** to load our DLL. CreateRemoteThread's definition looks like:

```
HANDLE CreateRemoteThread(
  HANDLE                 hProcess,
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  SIZE_T                 dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID                 lpParameter,
  DWORD                  dwCreationFlags,
  LPDWORD                lpThreadId
);
```


Playrix Ex. 1024, page 370

Let's step through each parameter required. The process will be the process handle for Urban Terror, identical to **WriteProcessMemory**. The next two parameters we do not need, so we can pass **NULL** and **0** for them. Our start address will be the address of **LoadLibraryA** that we retrieve through **GetProcAddress**. Finally, we need to pass a single parameter to **LoadLibraryA**, our DLL path, which we know from our call to **VirtualAllocEx**. For the purpose of our injector, we can ignore the last two parameters as well. With all of this down, our call ends up looking like:

```
HANDLE thread = CreateRemoteThread(process, NULL, 0,
(LPTHREAD_START_ROUTINE)GetProcAddress(kernel32base, "LoadLibraryA"),
lpBaseAddress, 0, NULL);
```

We have some additional operations we need to do with our thread, so we will save a handle to the thread. Before exiting, we want our injector to wait until the thread has been created and finished executing. We can do this via WaitForSingleObject and GetExitCodeThread:

```
WaitForSingleObject(thread, INFINITE);
GetExitCodeThread(thread, &exitCode);
```

7.1.9 Clean Up

Finally, we can free up the memory we allocated and close the open handles we have after our DLL has been injected:

```
VirtualFreeEx(process, lpBaseAddress, 0, MEM_RELEASE);
CloseHandle(thread);
CloseHandle(process);
break;
```

The final **break** exits the loop that we created to scan through each process.

With all of this done, we can start Urban Terror, enter a game, and then run our injector. If everything went successfully, players will start appearing through walls, indicating that our DLL was injected. If it fails, make sure to run the injector with administrator permissions.

The full code for the injector is available in Appendix A.

Playrix Ex. 1024, page 371

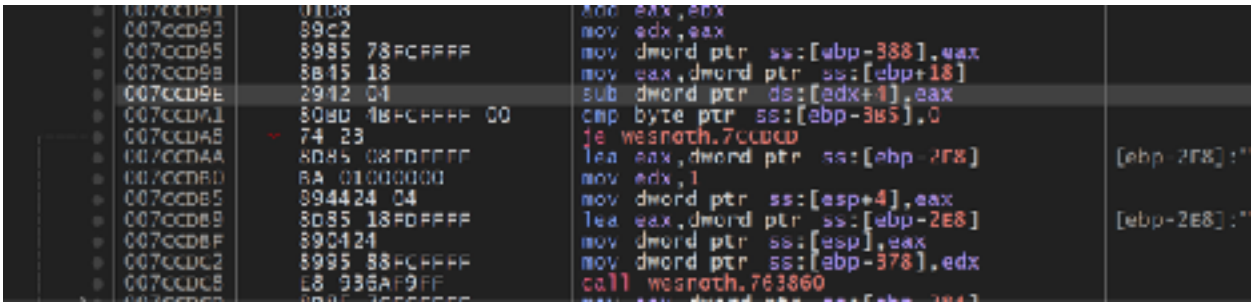
7.2 Pattern Scanner

7.2.1 Target

Our targets in this chapter will be Wesnoth 1.14.9 and Wesnoth 1.14.12.

7.2.2 Overview

In Chapter 2.3, we located the **sub** instruction responsible for subtracting gold from our player when we recruited a unit. In the 1.14.9 version of the game, we located this instruction at `0x7CCD9E`:

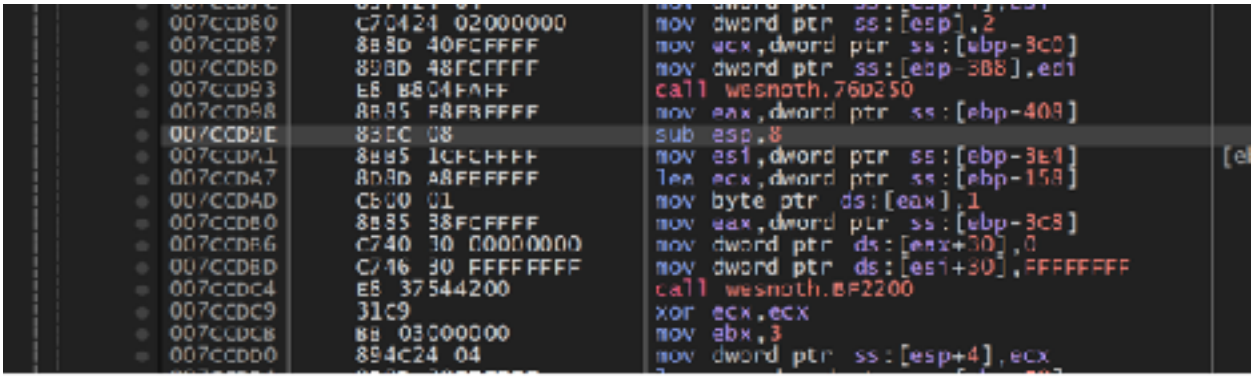


If a game's code is not loaded dynamically, addresses for instructions will not change. As such, we can consistently use them when programming. We used this behavior across several targets to build code caves, such as in Chapter 3.4.

However, newer versions of Wesnoth have been released, like 1.14.12. This version can be installed using Chocolatey in the same way we installed version 1.14.9:

```
choco install wesnoth --version=1.14.12 -y
```

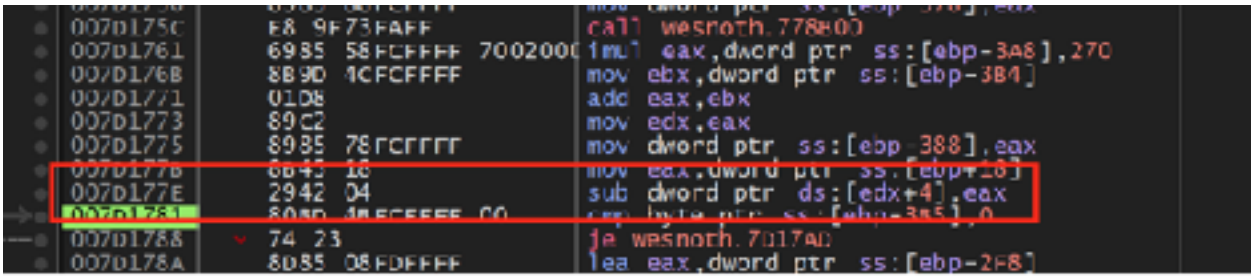
Most games will continually release additional versions and require updates to continue playing on multiplayer servers. If we examine `0x7CCD9E` in Wesnoth 1.14.12, we see that the **sub** instruction is no longer there:



When developers introduce new features or fix bugs in each release, they modify the game's code. They then compile these changes to produce a new executable for the game. Since this new executable has different code, the location of all code in the game will change. This is why the **sub** instruction is no longer present at `0x7CCD9E` in version 1.14.12.

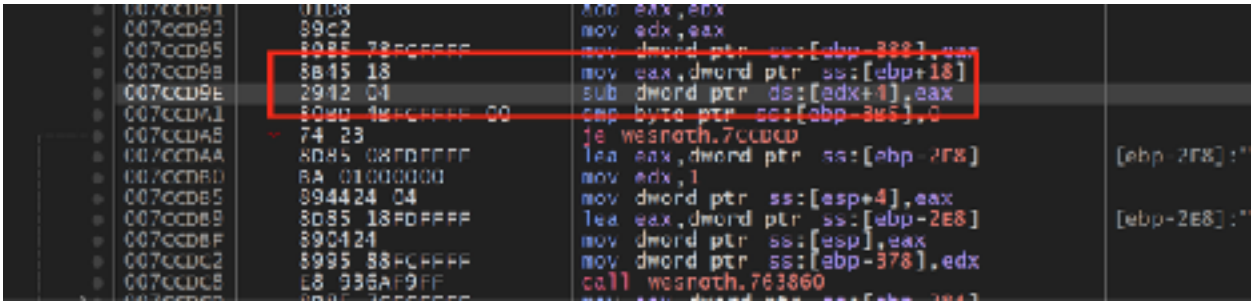
7.2.3 Opcodes

If we wanted to find the new address of the **sub** instruction, one approach is to repeat the exact same method we used in Chapter 2.3. If we do this, we can identify that the sub instruction in 1.14.12 is located at `0x7D177E`:



However, if we wanted to then upgrade our hack to a newer version, like 1.14.15, we would have to repeat this process again. This is a time-intensive process, especially for more complex tasks, like locating a player's base pointer.

Back in Chapter 1.1, we covered operation codes, or opcodes. Each opcode represents an instruction to execute. x64dbg displays the opcode for each instruction in the column to the left of the instruction:

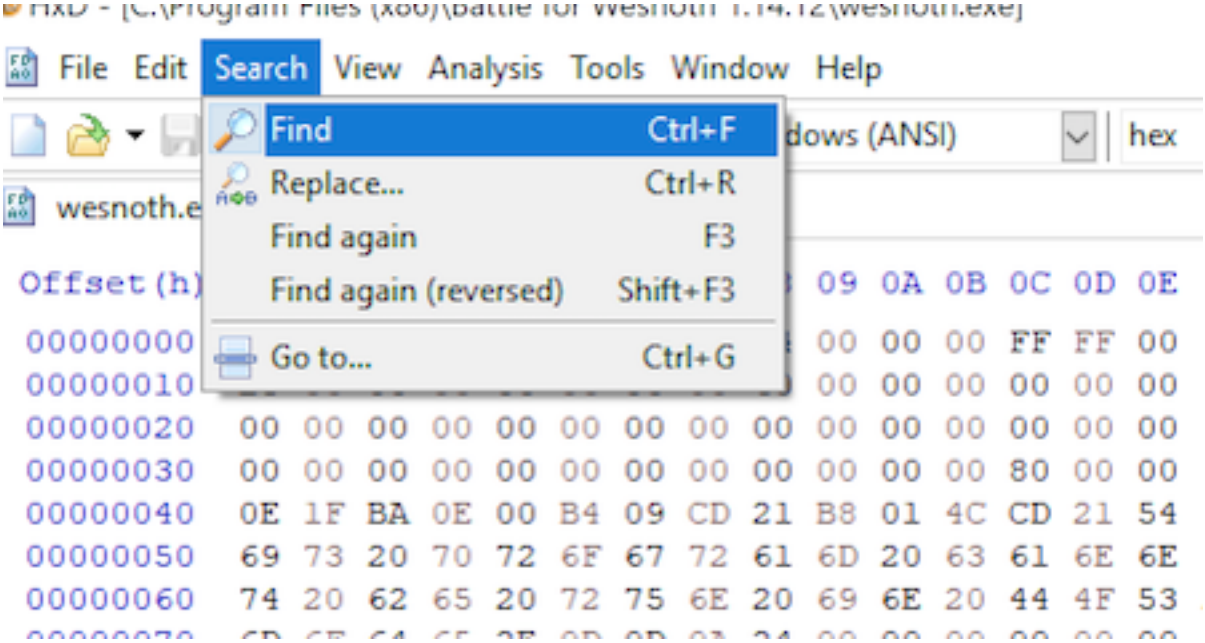


For example, the opcode for the sub instruction we identified is `0x2942 04`.

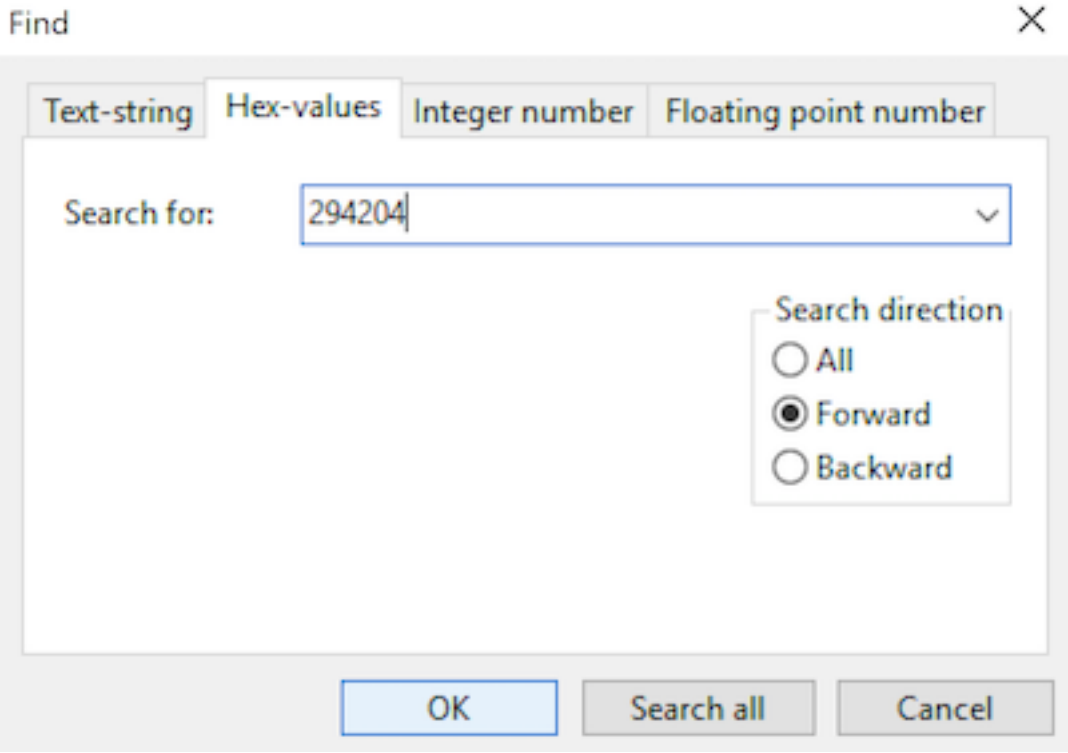
Executables do not store their code as assembly instructions. Rather, they store it as opcodes. The disassembly observed in x64dbg is reconstructed from these opcodes. We can verify this by opening up wesnoth.exe in a hex editor, a type of program that displays the hexadecimal bytes of a chosen file. In this chapter, we will use HxD:

```
choco install hxd
```

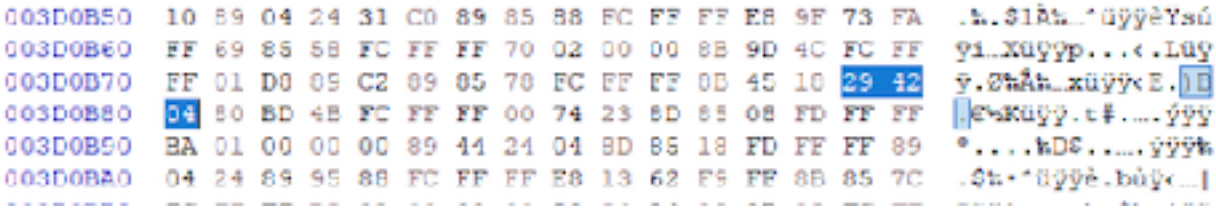
After opening wesnoth.exe, we can search for the opcode identified above via *Search -> Find*:



In the dialog that appears, we want to search for the hex value of our opcode:



Searching should highlight the opcode value:



Looking at the values near the highlighted value, we can see that they represent the other opcodes near the sub instruction.

7.2.4 Scanning

If we compare the opcodes for the **sub** instruction between 1.14.9 and 1.14.12, we can see that they are identical. This is because the opcodes for a particular instruction will always be the same. Since these opcodes do not change, we can scan for these bytes to locate the instruction we care about. This is known as pattern scanning.

Playrix Ex. 1024, page 375

Due to how Windows loads PE files into virtual memory, the address for the instruction differs between the hex editor and x64dbg. Since we want to locate and alter the running code, we are interested in identifying the latter address.

To accomplish this, we need to read the memory from a running instance of Wesnoth and then search that memory for a series of bytes. In this chapter, we will write an external program to demonstrate the concept, but this same behavior can be used inside a DLL to automatically update offsets.

Since we want to locate a running process and retrieve a process handle, we can start with the base that we already discussed in Chapter 7.1:

```
#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

int main(int argc, char** argv) {
      HANDLE snapshot = 0;
      PROCESSENTRY32 pe32 = { 0 };

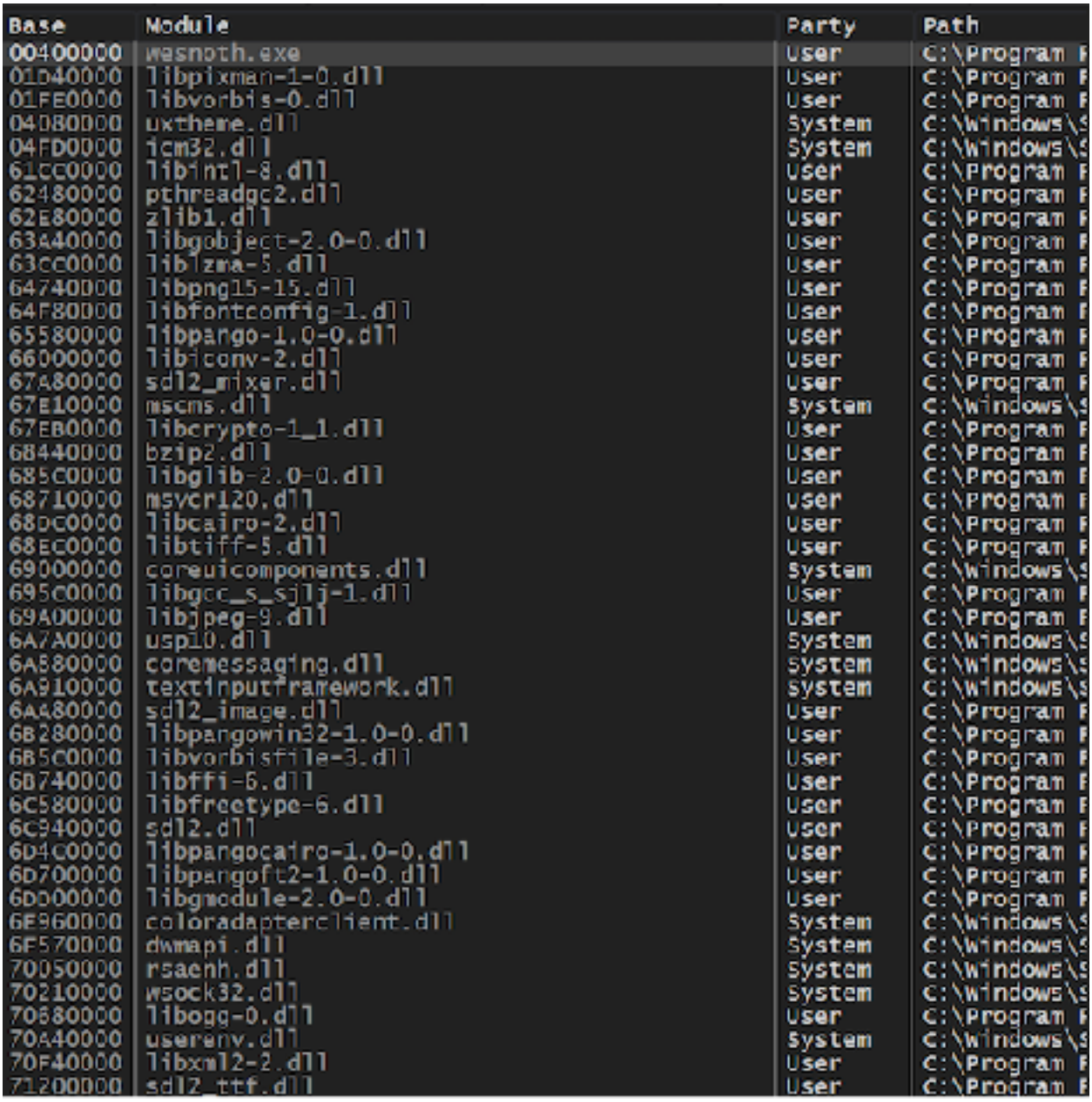
      pe32.dwSize = sizeof(PROCESSENTRY32);
      snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
      Process32First(snapshot, &pe32);

      do {
        if (wcscmp(pe32.szExeFile, L"wesnoth.exe") == 0) {
            HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);

            CloseHandle(process);
                  break;
        }
      } while (Process32Next(snapshot, &pe32));

    return 0;
}
```

With the process handle to Wesnoth, our next goal is to read the process's memory into a buffer that we can scan. However, processes are made up of many modules, or sections of code. For example, the Wesnoth process has modules for the main game code (wesnoth.exe), compression code (zlib1.dll), and graphics code (sdl.dll). We can observe all the modules loaded into the process using x64dbg's *Symbol* tab:



Base	Module	Party	Path
00400000	wesnoth.exe	User	C:\Program F
01D40000	libpixman-1-0.dll	User	C:\Program F
01FE0000	libvorbis-0.dll	User	C:\Program F
04D80000	uxtheme.dll	System	C:\Windows\S
04FD0000	icm32.dll	System	C:\Windows\S
61CC0000	libintl-8.dll	User	C:\Program F
62480000	pthreadgc2.dll	User	C:\Program F
62E80000	zlib1.dll	User	C:\Program F
63A40000	libgobject-2.0-0.dll	User	C:\Program F
63CC0000	liblzma-5.dll	User	C:\Program F
64740000	libpng15-15.dll	User	C:\Program F
64F80000	libfontconfig-1.dll	User	C:\Program F
65580000	libpango-1.0-0.dll	User	C:\Program F
66000000	libiconv-2.dll	User	C:\Program F
67A80000	sdl2_mixer.dll	User	C:\Program F
67E10000	mscms.dll	System	C:\Windows\S
67EB0000	libcrypto-1_1.dll	User	C:\Program F
68440000	bzip2.dll	User	C:\Program F
685C0000	libglib-2.0-0.dll	User	C:\Program F
68710000	msvcr120.dll	User	C:\Program F
68DC0000	libcairo-2.dll	User	C:\Program F
68EC0000	libtiff-5.dll	User	C:\Program F
69000000	coreuicomponents.dll	System	C:\Windows\S
695C0000	libgcc_s_sjlj-1.dll	User	C:\Program F
69A00000	libjpeg-9.dll	User	C:\Program F
6A7A0000	usp10.dll	System	C:\Windows\S
6A880000	coremessaging.dll	System	C:\Windows\S
6A910000	textinputframework.dll	System	C:\Windows\S
6A480000	sdl2_image.dll	User	C:\Program F
6B280000	libpangowin32-1.0-0.dll	User	C:\Program F
6B5C0000	libvorbisfile-3.dll	User	C:\Program F
6D740000	libffi-6.dll	User	C:\Program F
6C580000	libfreetype-6.dll	User	C:\Program F
6C940000	sdl2.dll	User	C:\Program F
6D4C0000	libpangocairo-1.0-0.dll	User	C:\Program F
6D700000	libpangoft2-1.0-0.dll	User	C:\Program F
6DD00000	libgmodule-2.0-0.dll	User	C:\Program F
6E960000	coloradapterclient.dll	System	C:\Windows\S
6F570000	dwmapi.dll	System	C:\Windows\S
70050000	rsaenh.dll	System	C:\Windows\S
70210000	wsock32.dll	System	C:\Windows\S
70680000	libogg-0.dll	User	C:\Program F
70A40000	userenv.dll	System	C:\Windows\S
70F40000	libxml2-2.dll	User	C:\Program F
71200000	sdl2_ttf.dll	User	C:\Program F

Since our **sub** instruction is in the wesnoth.exe module, we only want to scan this memory. To do this, we want to identify the base address of the module and its size. The **CreateToolhelp32Snapshot** API also allows us to iterate over a process's modules using **Module32First** and **Module32Next**:

```
if (wcscmp(pe32.szExeFile, L"wesnoth.exe") == 0) {
    HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);

    HANDLE module_snapshot = 0;
```

```
    MODULEENTRY32 me32;

    me32.dwSize = sizeof(MODULEENTRY32);
    module_snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE,
pe32.th32ProcessID);
    Module32First(module_snapshot, &me32);
    do {
          if (wcscmp(me32.szModule, L"wesnoth.exe") == 0) {

            break;
        }
    } while (Module32Next(module_snapshot, &me32));

    CloseHandle(process);
      break;
}
```

At this point, the **me32** structure will hold a few members that we care about: **modBaseAddr**, the base address of the module, and **modBaseSize**, the size of the module. We will use these two members to allocate a buffer and read the module's memory into the buffer:

```
unsigned char *buffer = (unsigned char*)calloc(1, me32.modBaseSize);
DWORD bytes_read = 0;

ReadProcessMemory(process, (void*)me32.modBaseAddr, buffer, me32.modBaseSize,
&bytes_read);

//scanning code

free(buffer);
```

At this point, our buffer contains the content of the memory from the wesnoth.exe module base to the end of the module. This memory contains the opcodes for the game's code. We can now scan over this memory to look for our pattern of bytes.

For each byte in the buffer, we will see if the pattern exists starting at the byte. If not, we will continue on to the next byte. If all the bytes match, we will print the offset in the buffer combined with the wesnoth.exe module base:

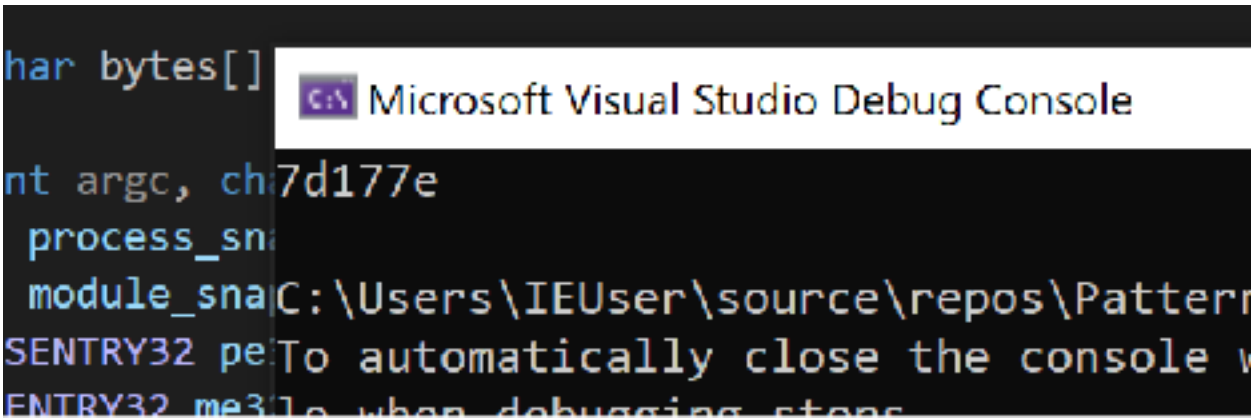
```
unsigned char bytes[] = { 0x29, 0x42, 0x04 };
```


Playrix Ex. 1024, page 378

```
for (unsigned int i = 0; i < me32.modBaseSize - sizeof(bytes); i++) {
    for (int j = 0; j < sizeof(bytes); j++) {
        if (bytes[j] != buffer[i + j]) {
            break;
        }

        if (j + 1 == sizeof(bytes)) {
            printf("%x\n", i + (DWORD)me32.modBaseAddr);
        }
    }
}
```

If we start Wesnoth 1.14.12 and then run our scanner, it will correctly print out the location of the **sub** instruction:



We can use this on any version of Wesnoth to locate the **sub** instruction we care about.

The full code for this chapter is available in Appendix A for comparison.

7.3 Memory Scanner

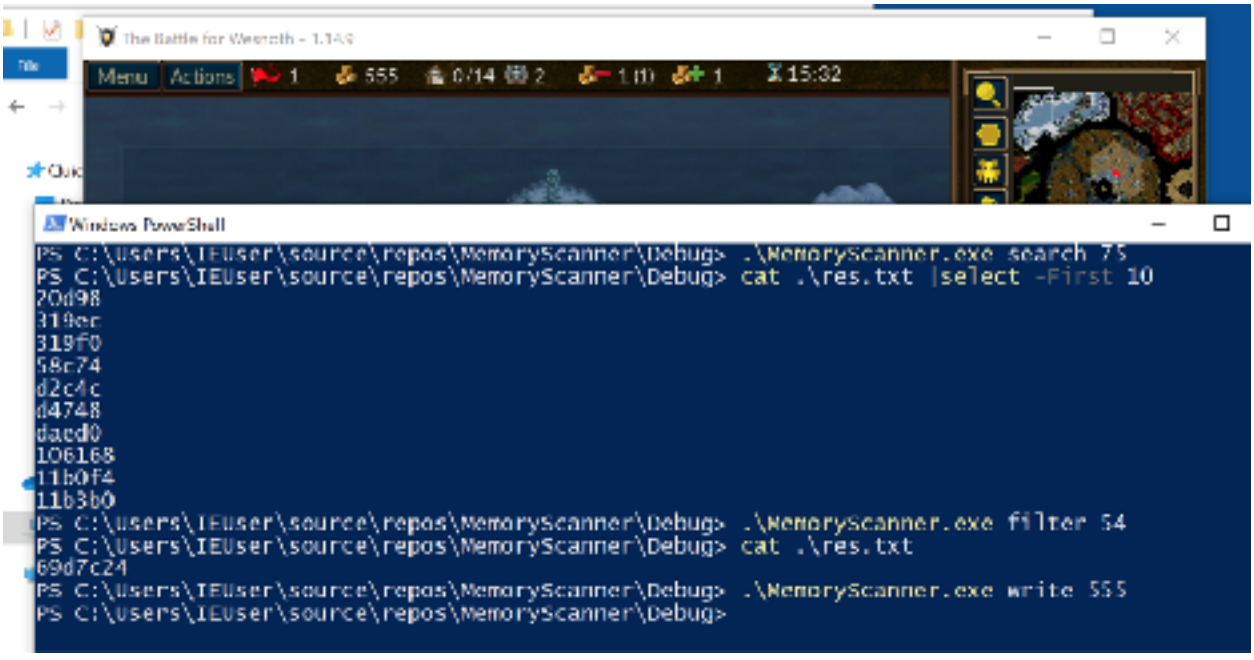
7.3.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

7.3.2 Overview

In previous chapters, we used Cheat Engine to search for memory addresses and change their values. Cheat Engine is a type of program known as a memory scanner. Memory scanners allow you to search for and edit memory inside a process.

Our goal in this chapter is to create a memory scanner that will operate on DWORD values for the game Wesnoth.



7.3.3 Understand

Memory scanners have three main operations:

1. Search all memory for a certain value.
2. Filter previously identified addresses against a new value.
3. Set a memory address to a certain value.

In the previous chapter, we created a pattern scanner that would search the main Wesnoth module for a series of bytes. We can use the same technique to search memory for a value. However, in this case, we will scan all memory from `0x00000000` to `0x7FFFFFFF`. This range of addresses represents all the virtual address space that a 32-bit Windows executable has access to. When scanning, we will save any address that is set to a certain value.

To filter these addresses, we will perform the same scan operation described above with one major difference: instead of scanning from `0x00000000` to `0x7FFFFFFF`, we will only scan saved addresses identified from the previous scan step. Any addresses that still match a provided value will again be saved. In this way, we can continue to filter down the list of valid addresses.

Finally, to write to an address, we can use the same **WriteProcessMemory** technique identified in Chapter 3.2.

7.3.4 Program Structure

Before we write our program, we need to determine how we will handle the multiple operations and passing data from one operation to another.

Since we have three distinct operations for our memory scanner to perform, we need to determine how to handle these cases. One approach is to create a separate program for each operation and then transfer data between the three programs. However, this approach would require us to duplicate logic between multiple programs, such as the logic to open a process handle.

For this chapter, we will use command-line arguments to designate which operation we want to perform. For example, if we want to search for the value 50, we will call our program like:

```
MemoryScanner.exe search 50
```

Since we need to call our scanner multiple times, we need a way to pass results from one operation to the next. The easiest way to accomplish this is to use a file. For example, if we search for a value, the file will be filled with all addresses that match this

value. When we filter, addresses will be read from this file, and then new addresses are placed in the file if they still match.

7.3.5 Process Handle

To read and write memory from Wesnoth, we need a process handle. We will use the same approach we used in the previous chapter to accomplish this:

```
#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

int main(int argc, char** argv) {
	HANDLE process_snapshot = 0;
	PROCESSENTRY32 pe32 = { 0 };

	pe32.dwSize = sizeof(PROCESSENTRY32);

	process_snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
	Process32First(process_snapshot, &pe32);

	do {
		if (wcscmp(pe32.szExeFile, L"wesnoth.exe") == 0) {
			HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);

		// handle operations

			CloseHandle(process);
			break;
		}
	} while (Process32Next(process_snapshot, &pe32));

	return 0;
}
```

We will pass this process handle to all of our operations.

Playrix Ex. 1024, page 382

7.3.6 Operations

Next, we can add in our operations. To access command-line arguments passed to our program, we can use the **argv** argument. **argv[0]** will always hold our program's name on the command-line (**MemoryScanner.exe**), with **argv[1]** representing the first argument.

All arguments are passed in as strings. We want our program to search for **DWORD** values. To convert from a string to a value that we can use to search for **DWORD**'s, we will use **strtol**, or (str)ing (to) (l)ong:

```
// handle operations
char* p;
long value = strtol(argv[2], &p, 10);

if(strcmp(argv[1], "search") == 0) {
    search(process, value);
}
else if(strcmp(argv[1], "filter") == 0) {
    filter(process, value);
}
else if (strcmp(argv[1], "write") == 0) {
    write(process, value);
}
```

With the base in place, we can now write each of these functions.

7.3.7 Search

We will start with our search function:

```
void search(const HANDLE process, const int passed_val) {
```

Like we discussed in the previous section, we will store the results of the search in a text file. Like we did back in Chapter 6.4, we will use **fopen_s** to create a text file we can write to:

```
FILE* temp_file = NULL;
fopen_s(&temp_file, "res.txt", “w");
```

As we know, memory does not have a particular structure. For example, the memory from `0x12345678` to `0x1234567C` could hold the values `0x44 0x45 0x41 0x45`. If read as a DWORD, this memory would hold the value 1145389381. However, if each byte is read as a char, this memory would hold the value DEAD. In memory scanners like Cheat Engine, you can select the type of data to scan for. In this chapter, we will scan all memory as if it was a DWORD. This will allow us to search for values that are numbers, such as gold.

Our search operation will scan all memory from `0x00000000` to `0x7FFFFFFF` and compare each 4 bytes to the value passed in the second argument. Previously, we used **ReadProcessMemory** to read a single 4-byte **DWORD**. However, **ReadProcessMemory** allows us to read any size of memory into any type of allocated buffer.

While Wesnoth can use all memory from `0x00000000` to `0x7FFFFFFF`, it first needs to request access via several API's, like **VirtualAlloc**. If Wesnoth has not requested access to a certain piece of memory, it will not be able to read or write data to it. We are using Wesnoth's handle to read memory, so we will need to account for this behavior.

If we try to read all memory from `0x00000000` to `0x7FFFFFFF` with one **ReadProcessMemory** call, the call will fail. This is because **ReadProcessMemory**'s behavior is to immediately fail and place a **NULL** value in our buffer if we encounter a section of memory we do not have access to. As a result, we will need to split our read requests up into blocks. That way, if we attempt to scan a block that Wesnoth has not allocated, only that block's read will fail.

We can choose any value for our block size, but there is a trade-off between speed and accuracy. The larger each block is, the faster the scan process will take, but more areas of memory may not be read successfully due to part of the block being inaccessible. For this chapter, we will choose a block size of 2056, or `0x808`:

```
#define size 0x00000808
```

We will then allocate a buffer that can hold a block-size worth of data:

```
unsigned char* buffer = (unsigned char*)calloc(1, size);
```

Next, we will loop through each block of memory from `0x00000000` to `0x7FFFFFFF` and read that block into the buffer:

```
DWORD bytes_read = 0;

for (DWORD i = 0x00000000; i < 0x7FFFFFFF; i += size) {
      ReadProcessMemory(process, (void*)i, buffer, size, &bytes_read);
```

Finally, we will cast each 4 bytes of our buffer as a **DWORD** and determine if its value equals the argument passed. If so, we will write its location to our results file:

```
for (int j = 0; j < size - 4; j += 4) {
    DWORD val = 0;
    memcpy(&val, &buffer[j], 4);
    if (val == passed_val) {
        fprintf(temp_file, "%x\n", i + j);
    }
}
```

If a read fails, our buffer will contain nothing but 0's and this final step will find nothing. After we finish with our **ReadProcessMemory** loop, we will close the file and free the buffer's memory:

```
fclose(temp_file);
free(buffer);
```

Our search function is now finished. If you build this code and search for a gold value inside Wesnoth, you will see that **res.txt** now contains a several addresses:

```
MemoryScanner.exe search 75
```

7.3.8 Filtering

The next operation we will focus on is filtering. The filtering operation will take a list of addresses produced by the search operation and check to see if those addresses equal a new value. If the address does equal the value, it will be saved. If it does not, it will be deleted:

```
void filter(const HANDLE process, const int passed_val) {
```

We will conduct the filtering operation in two parts:

1. Read each memory address from **res.txt** and if it matches the new value, save it to **res_fil.txt**.
2. Copy **res_fil.txt** to **res.txt** and delete **res_fil.txt**.

The end result will be a new **res.txt** file that contains only the filtered addresses. This model will allow us to filter multiple times. First, we will open **res.txt** for reading (**r**) and **res_fil.txt** for writing (**w**):

```
FILE* temp_file = NULL;
FILE* temp_file_filter = NULL;
fopen_s(&temp_file, "res.txt", "r");
fopen_s(&temp_file_filter, "res_fil.txt", “w");
```

We will then read each address from **res.txt** line by line and read Wesnoth's memory at that address. If the value matches our argument, we will write the address to **res_fil.txt**:

```
DWORD address = 0;
while (fscanf_s(temp_file, "%x\n", &address) != EOF) {
    DWORD val = 0;
    DWORD bytes_read = 0;

    ReadProcessMemory(process, (void*)address, &val, 4, &bytes_read);
    if (val == passed_val) {
        fprintf(temp_file_filter, "%x\n", address);
    }
}
```

With all the filtered addresses in **res_fil.txt**, we will then close both **res.txt** and **res_fil.txt**. Then, we will open up these files in the opposite order from above, with **res.txt** for writing and **res_fil.txt** for reading:

```
fclose(temp_file);
fclose(temp_file_filter);

fopen_s(&temp_file, "res.txt", "w");
fopen_s(&temp_file_filter, "res_fil.txt", “r");
```

Next, we will loop through each address in **res_fil.txt** and copy it to **res.txt**:

Playrix Ex. 1024, page 386

```
while (fscanf_s(temp_file_filter, "%x\n", &address) != EOF) {
    fprintf(temp_file, "%x\n", address);
}
```

With **res.txt** now containing our addresses, we will close each file and delete **res_fil.txt**:

```
fclose(temp_file);
fclose(temp_file_filter);

remove(“res_fil.txt");
```

We can now search for and filter addresses. If you search for your gold in Wesnoth, buy a unit, and then filter your gold value, you should be left with a single value. If you open up Cheat Engine and repeat these steps, you can verify that the address you identified and the address from Cheat Engine match. This shows that our scanner is properly finding memory addresses.

```
MemoryScanner.exe filter 54
```

7.3.9 Writing

The final main operation of a memory scanner is writing values to identified memory addresses:

```
void write(const HANDLE process, const int passed_val) {
```

This operation is identical to the approach we used in Chapter 3.2. For each address in **res.txt**, we will use **WriteProcessMemory** to write the provided argument value to the address:

```
FILE* temp_file = NULL;
fopen_s(&temp_file, "res.txt", "r");

DWORD address = 0;
while (fscanf_s(temp_file, "%x\n", &address) != EOF) {
    DWORD bytes_written = 0;
```

```
    WriteProcessMemory(process, (void*)address, &passed_val, 4,
&bytes_written);
}

fclose(temp_file);
```

With this code, we can now write whatever value we want to the previously searched for and filtered addresses:

```
MemoryScanner.exe write 555
```

The full code for this chapter is available in Appendix A for comparison.

Playrix Ex. 1024, page 388

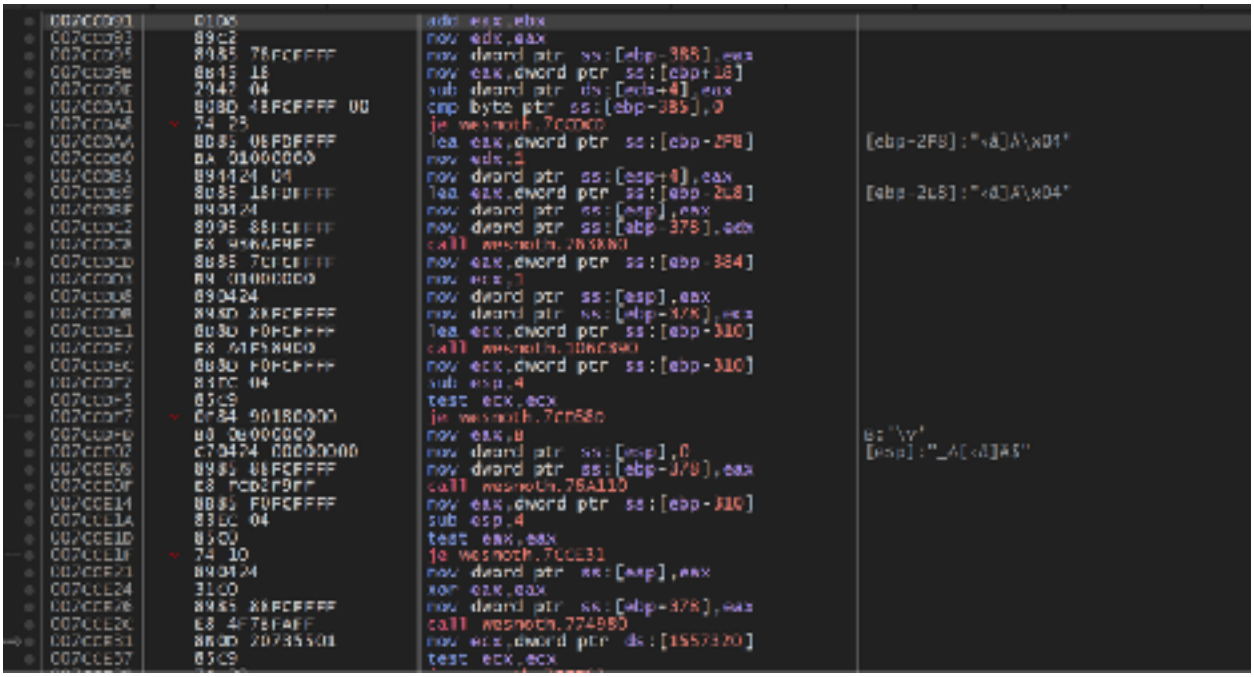
7.4 Disassembler

7.4.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

7.4.2 Overview

In previous chapters, we used x64dbg to debug and reverse games. When viewing these games in x64dbg, we are able to see the instructions that the games are executing. For example, we saw that the following instructions were responsible for decreasing a player's gold when recruiting a unit in Wesnoth:



From Chapter 7.2, we know that these instructions are all stored as opcodes, which are byte values. The process of converting these opcodes to instructions is known as disassembly. In this chapter, we will cover how to create a limited disassembler.

The full source code discussed in this chapter is available in Appendix A.

7.4.3 Disclaimer

Writing a disassembler is a complex task that takes a large amount of time. Even supporting a single instruction set in an efficient way takes many weeks of reading specifications and implementation. The approach covered here should be used as a starting point, but with the caveat that the approach will not scale. The main goal for this chapter is to explain how these concepts work. For an example of a feature-complete disassembler, check out the Capstone Engine.

7.4.4 Instructions

For a CPU to understand and execute each opcode encountered, these opcodes must have a consistent format. Each opcode must be assigned a specific instruction. For example, we have seen from previous chapters that the opcode `0xE8` always represents a **call** instruction. This mapping of opcodes to instructions is known as a processor's instruction set.

Each CPU can implement a unique instruction set. However, most Windows-based games are compiled with the expectation that they will be running on 32-bit, Intel-based processors. These processors typically implement a version of the x86 instruction set. For Intel processors specifically, this is referred to as IA-32.

The x86 instruction set is complex and has many different operations. These operations can also be a different length. For example, in the screenshot on the page above, we see that the **mov** instruction on the second line (`0x7ccd93`) is 2 bytes (`0x89C2`), whereas the **mov** instruction on the third line is 6 bytes (`0x8985 78FCFFFF`). For the CPU to understand the length of the instruction, this data must be encoded in the bytes in some way.

7.4.5 Instruction Set Reference

Imagine you want to create a compiler that will take the following C++ code and produce a binary that can run on an x86-compatible processor:

```
int x = 2;
```

This code could be converted into assembly in multiple ways, such as:

```
mov [x], 2
```

or

```
mov eax, 2
mov [x], eax
```

We have seen that there are multiple forms of the **mov** instruction, with different lengths. As the compiler developer, we need to know which form to use to produce our binary code.

To solve this problem, companies like Intel release instruction set references. These contain a full listing of all public instructions and their associated opcodes, along with other architectural information, such as how to encode the length of the instruction. The IA-32 reference is available here.

As we build our disassembler, we will use that reference to understand instructions. In addition, we will use another reference (here) to help figure out unknown opcodes and which instruction they are associated with.

7.4.6 Dumping a Process's Opcodes

Like in Chapter 7.2, our target in this chapter will be Wesnoth. We will use the same code from that chapter to locate, attach, and read the game's opcodes into a buffer:

```
int main(int argc, char** argv) {
	HANDLE process_snapshot = 0;
	HANDLE module_snapshot = 0;
	PROCESSENTRY32 pe32 = { 0 };
	MODULEENTRY32 me32;

	DWORD exitCode = 0;

	pe32.dwSize = sizeof(PROCESSENTRY32);
	me32.dwSize = sizeof(MODULEENTRY32);

	process_snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
	Process32First(process_snapshot, &pe32);

	do {
```


Playrix Ex. 1024, page 391

```
            if (wcscmp(pe32.szExeFile, L"wesnoth.exe") == 0) {
                  module_snapshot =
CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pe32.th32ProcessID);

                  HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);

                  Module32First(module_snapshot, &me32);
                  do {
                        if (wcscmp(me32.szModule, L"wesnoth.exe") == 0) {
                              unsigned char* buffer = (unsigned
char*)calloc(1, me32.modBaseSize);
                              DWORD bytes_read = 0;

                              ReadProcessMemory(process,
(void*)me32.modBaseAddr, buffer, me32.modBaseSize, &bytes_read);

                    // buffer contains the game's opcodes

                              free(buffer);
                              break;
                        }

                  } while (Module32Next(module_snapshot, &me32));

                  CloseHandle(process);
                  break;
            }
      } while (Process32Next(process_snapshot, &pe32));

      return 0;
}
```

We will validate our disassembler on the same instruction set seen in Section 7.4.2 (starting at the address `0x7ccd91`). We will also only disassemble `0x50` bytes' worth of instructions. First, we will simply dump all the opcodes:

```
#define START_ADDRESS 0x7ccd91
...

unsigned int i = START_ADDRESS - (DWORD)me32.modBaseAddr;
while (i < START_ADDRESS + 0x50 - (DWORD)me32.modBaseAddr) {
    printf("%x", buffer[i]);
      i++;
```


Playrix Ex. 1024, page 392

```
    printf("\n");
}
```

Our code above needs to offset **i** in this manner due to how the opcodes are read into our buffer. Like we saw in Chapter 7.2, the game's main module is loaded at address `0x400000`. However, this instruction is stored at position 0 in our buffer. To gain access to the opcodes starting at `0x7ccd91`, we need to determine the distance from `0x7ccd91` to `0x400000` and use that position in our buffer.

When executed, this code will produce the following result:



We can see that these opcodes line up with the values observed in x64dbg.

7.4.7 The add Instruction

Starting at the very top, we see that the first opcode is `0x01`. Looking at our reference site here, we see that this is an **add** instruction:

pf	0F	po	so	o	proc	st	m	rl	x	mnemonic	op1	op2
		00		r					L	ADD	**r/m8**	r8
		01		r					L	ADD	**r/m16/32/64**	r16/32/64
		02		r						ADD	**r8**	r/m8

If we look in section 3.2 of the reference, we can see that this instruction adds a 32-bit register to a 32-bit register:

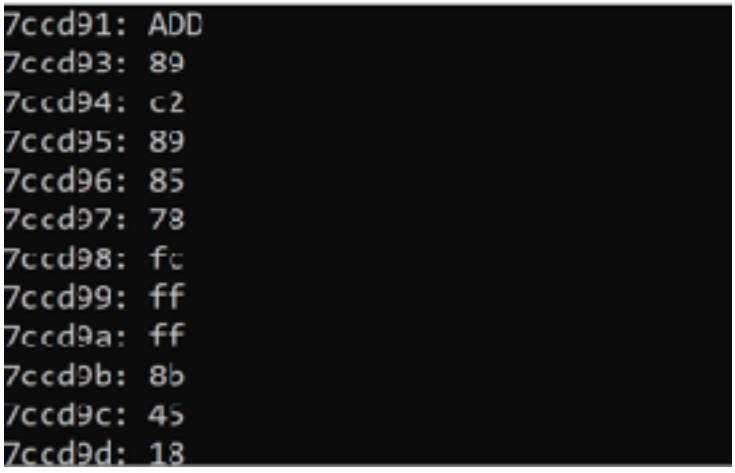
01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.

We still need to figure out how these registers are encoded, but for now, we can modify our main loop to print out an **add** instruction whenever we encounter `0x01`. Since we know that this instruction is 2 bytes, we will increment past the next opcode when we encounter it as well. We can also add code to print out the current address of the instruction:

```
while (i < START_ADDRESS + 0x50 - (DWORD)me32.modBaseAddr) {
    printf("%x:\t", i + (DWORD)me32.modBaseAddr);
    switch (buffer[i]) {
    case 0x1:
        printf("ADD ");
        i++;
        i++;
        break;
    default:
        printf("%x", buffer[i]);
        i++;
        break;
    }

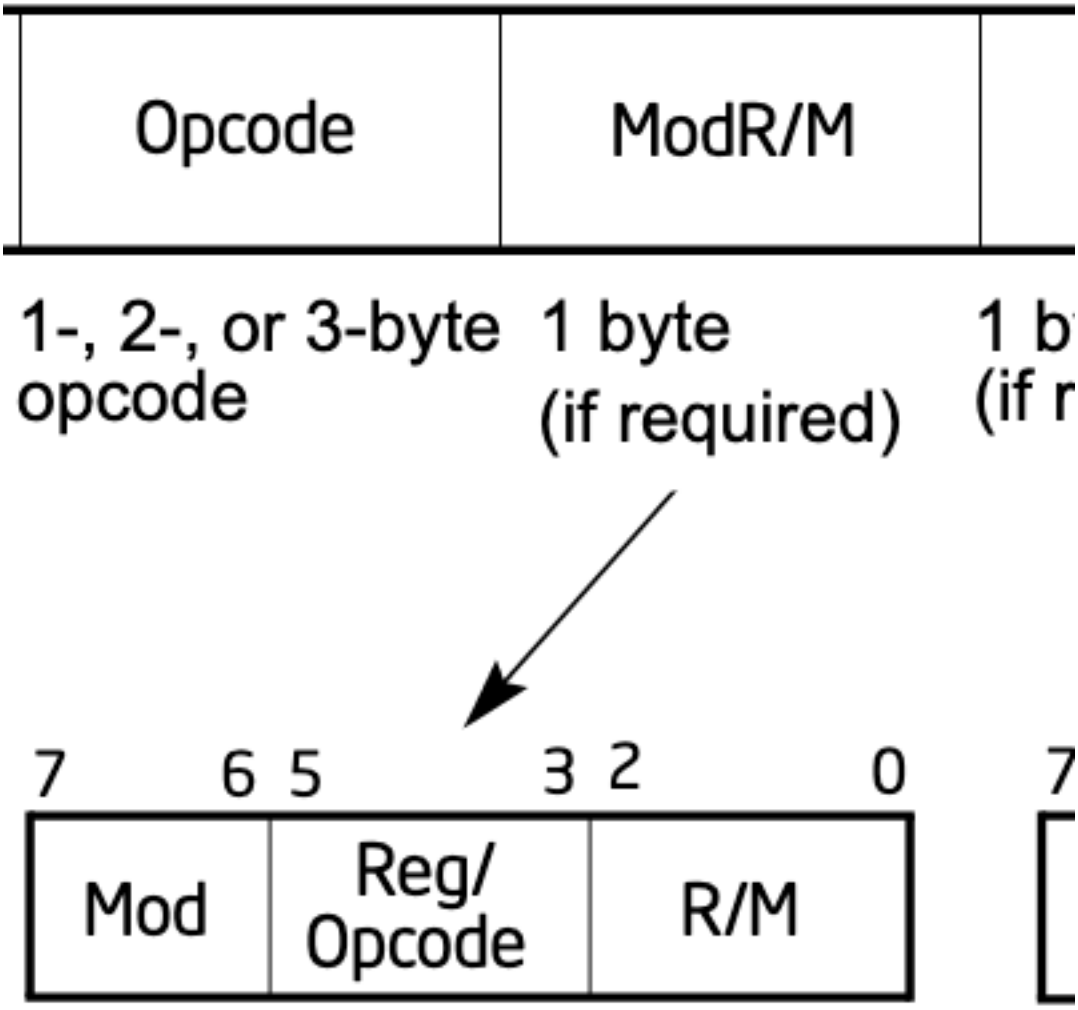
    printf("\n");
}
```

When running this code, we will now see that the first **add** instruction is correctly disassembled:



7.4.8 Decoding Operands

After `0x01`, the next opcode is `0xD8`. We know that this `0xD8` is somehow responsible for encoding the value of **eax, ebx**. Just like with opcodes, the exact method to decode this value must exist somewhere in this manual. If we look at the reference manual's section 2.1, we see that directly following the opcode is a *ModR/M* value that is 1 byte long:



If we scroll down to table 2-2, we can see how this value is laid out:

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)	AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)	AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)	0	1	2	3	4	5	6	7
(In binary) REG =	000	001	010	011	100	101	110	111

Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Finding the value of `0xD8`, we see that it is in the **eax** row and **ebx** column. Since this value is stored in a consistent manner, we can write a function to retrieve it:

```
int decode_operand(unsigned char* buffer, int location) {

    return 1;
}
```

So far, we have seen that operands are 1 byte long, so we will return a value of 1 to correctly increment the loop. We can call this function from our main loop:

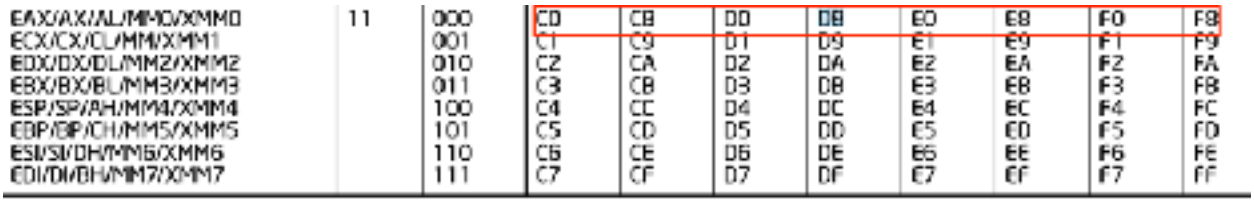
```
case 0x1:
    printf("ADD ");
    i++;
```

```
    i += decode_operand(buffer, i);
    break;
```

Going back to the table, we can see that we have 8 possible values: **eax**, **ecx**, **edx**, **ebx**, **esp**, **ebp**, **esi**, and **edi**. We can lay these out in an array of character arrays to reference in our function:

```
const char modrm_value[8][4] = {
      "eax",
      "ecx",
      "edx",
      "ebx",
      "esp",
      "ebp",
      "esi",
      "edi"
};
```

If we look at the table, we can see that **eax** will be the first operand whenever the byte value ends in 0 or 8:

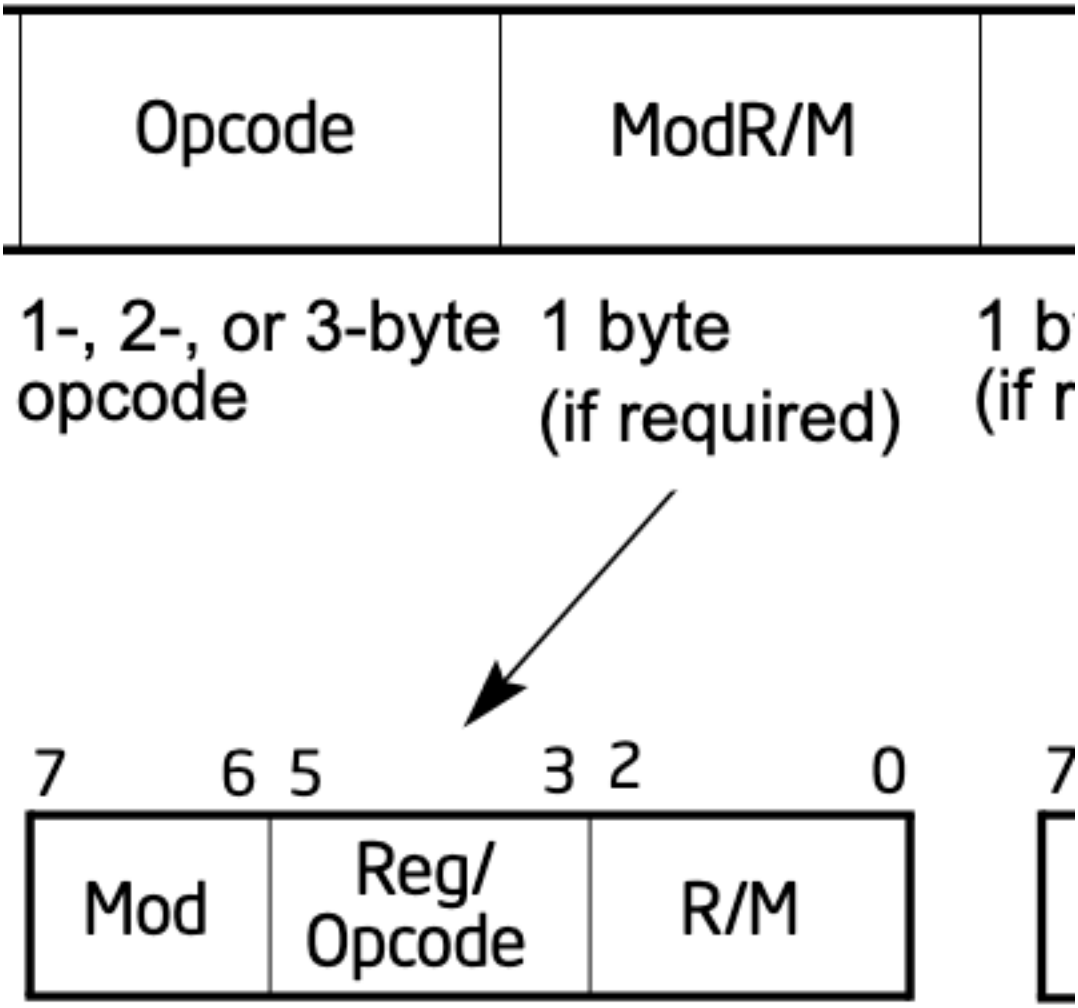


EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

This pattern continues for the other registers as well. For example, **ecx** always ends in 1 or 9, and **edx** in 2 or A. We can see that these values line up with the remainder when we divide the operand value by 8. Therefore, we can use the modulo operator to retrieve our first operand from the *ModR/M* value:

```
modrm_value[buffer[location] % 8]
```

To retrieve the second operand, we can use a similar operation. If we look at the *ModR/M* structure, we can see that the first value is stored at bits 0, 1, and 2:



For example, when converted to binary, `0xD8` is represented as:

```
1101 1000
```

For our first operand, we see that the three 000 bits are associated with the **eax** row. If we then shift these bits to the right, we get the following value:

```
0001 1011
```

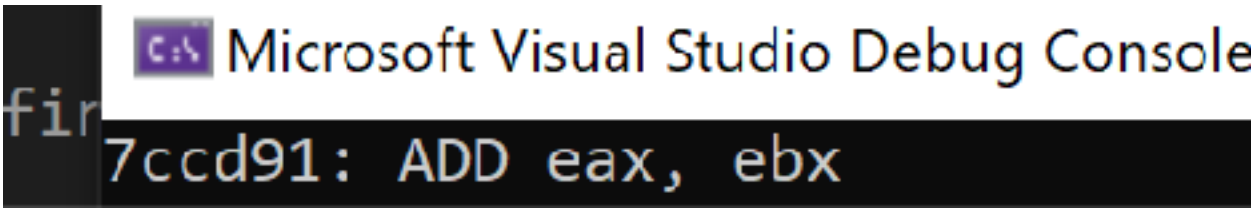
If we look at the columns on the top of the table, 011 is associated with the **ebx** column, in the same way as the first operand. As such, we can use the same approach once we shift the bits to retrieve the second operand via the modulo operator:

```
modrm_value[(buffer[location] >> 3) % 8]
```

With these two pieces, we can implement our function:

```
if (buffer[location] >= 0xC0 && buffer[location] <= 0xFF) {
    printf("%s, %s", modrm_value[buffer[location] % 8],
modrm_value[(buffer[location] >> 3) % 8]);
    return 1;
}
```

Running this code will correctly print the operands for the **add** operation:

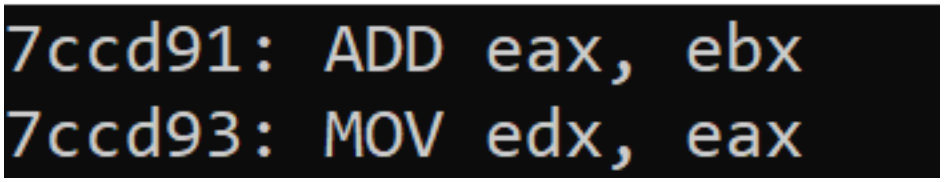


7.4.9 Other Instructions

Now that we can disassemble the **add** instruction, we can begin working on other instructions. First, let's implement the **mov** instruction at `0x7ccd93`:

```
case 0x89:
    printf("MOV ");
    i++;
    i += decode_operand(buffer, i);
    break;
```

Running this, we can verify that our operand decoding is working correctly, as `0xC2` (the operand associated with the second move) correctly decodes to **edx, eax**:



However, the next **mov** instruction does not decode correctly. Despite being the same opcode (`0x89`), it has an operand that we have not seen before, `0x85`. If we look at the table, we see that this is associated with **[ebp] + displacement**, or an offset. If we look at the x64dbg version, we can see that this offset is `-0x388`. We know that this value must be encoded somewhere in the instruction. Since `0x89 85` are already accounted for, this value must be in the `0x78fcffff` bytes.

In previous chapters, we talked about endianness, or the order of bytes. We identified that bytes are stored in a little-endian format. As a result, we need to reverse these bytes:

```
FF FF FC 78
```

This value does not match `0x388`. This is due to the signed nature of the value. Since this is a negative value, we need to subtract the maximum value of an integer (`0xFF FF FF FF`) to get the correct value:

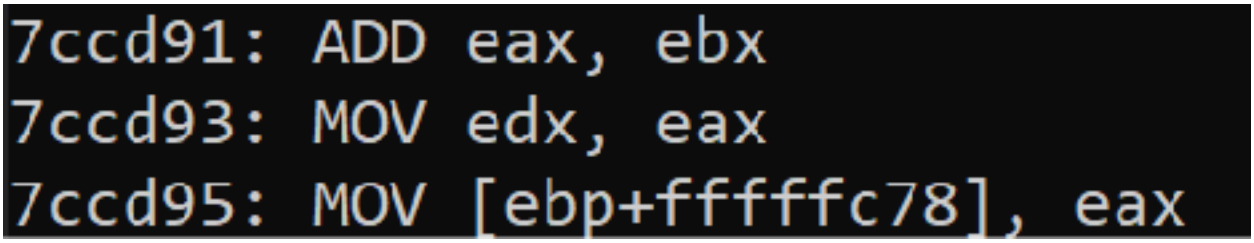
```
FF FF FF FF -
FF FF FC 78 =
387
```

We then need to add 1 to account for the sign change, resulting in the correct value of `-0x388`.

Since we now understand how this is working, we can add this to our decode operation:

```
else if (buffer[location] >= 0x80 && buffer[location] <= 0xBF) {
    DWORD displacement = buffer[location + 1] | (buffer[location + 2] << 8) |
(buffer[location + 3] << 16) | (buffer[location + 4] << 24);
    printf("[%s+%x], %s", modrm_value[buffer[location] % 8], displacement,
modrm_value[(buffer[location] >> 3) % 8]);
    return 5;
}
```

Like we saw with the first decoding operation, we can use bit shifting to retrieve each of the bytes in the displacement. With this included, the third operation now correctly decodes:



7.4.10 Calls and Jumps

In previous chapters, we covered how the opcode for a **call** or **jmp** used the following formula:

```
E8/E9 (new_location - original_location + 5)
```

We can reverse this operation to retrieve the address of a **call** from an opcode:

```
case 0xE8:
    printf("CALL ");
    i++;
    loc = buffer[i] | (buffer[i+1] << 8) | (buffer[i+2] << 16) | (buffer[i+3]
<< 24);
    printf("%x", loc + (i + (DWORD)me32.modBaseAddr) + 4);
    i += 4;
    break;
```

We add 4 instead of 5 to account for the fact that our parser is past the `0xE8` byte.

We also have a short relative jump if equal (**je**) instruction in our selected example. In this case, we can observe that the second byte of the opcode contains the amount to offset by:

```
7ccda8    74 23   je 7ccdcd
7ccda8 + 23 = 7ccdcd
```

We can add this logic to our main loop as well:

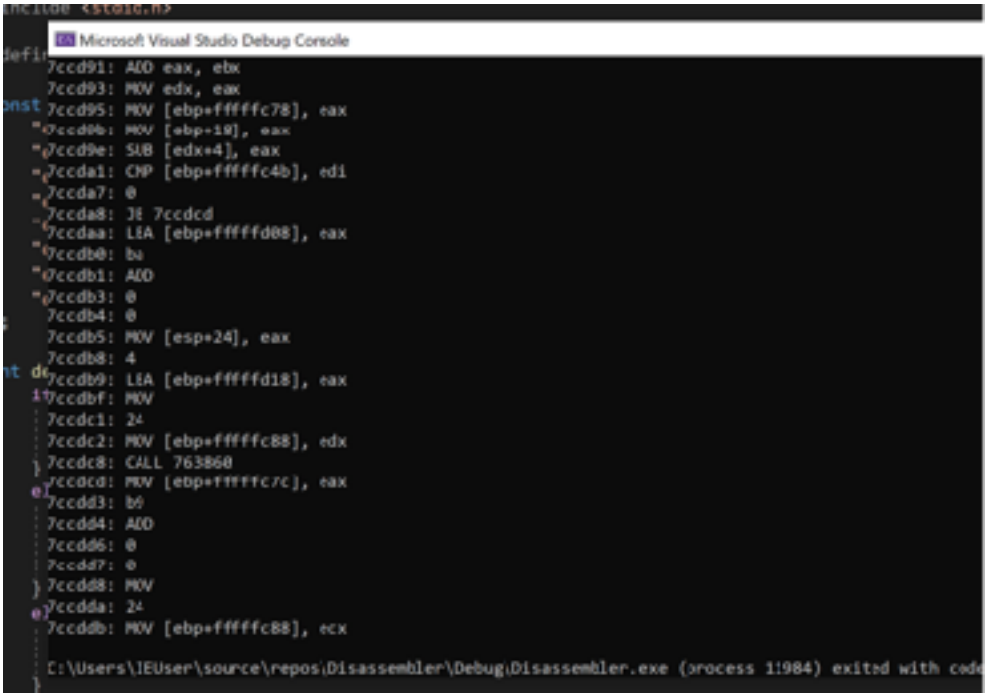
```
case 0x74:
    printf("JE ");
    printf("%x", i + (DWORD)me32.modBaseAddr + 2 + buffer[i + 1]);
    i += 2;
    break;
```

7.4.11 Final Result

As stated in the disclaimer, this was not a comprehensive disassembler. In the source shown in Appendix A, the following opcodes are implemented:

- **ADD** (0x01)
- **MOV** (0x89, 0x8B)
- **SUB** (0x29)
- **JE** (0x74)
- **CALL** (0xE8)
- **CMP** (0x80)
- **LEA** (0x8D)

With these instructions, we retrieve back the following result:



7.5 Debugger

7.5.1 Target

Our target for this chapter will be Assault Cube 1.2.0.2.

7.5.2 Overview

In previous chapters, we used x64dbg to debug and reverse games. After attaching x64dbg to these games, we were able to set breakpoints on game instructions. When the game executed these instructions, our breakpoints would pop and program execution would pause. We could then observe the values of all the registers and step through individual instructions.

In this chapter, we will explore how to create a debugger for Windows utilizing the Windows API. We will confirm that this debugger is working by using Assault Cube as an example. In Chapter 5.7, we identified that the **mov** instruction at `0x0046366C` was only executed when the player was firing. After we create our debugger, we will place a breakpoint on this instruction and verify that it is only hit when we fire.

7.5.3 Windows Debugger API's

Windows has a collection of API's that allow for a process to attach to and debug another process. These are detailed in several short articles available on MSDN. For our purposes, we mainly care about the following API's:

- **DebugActiveProcess**, which is used to attach to a target process
- **WaitForDebugEvent**, which is used to wait for debugging events, as described in this MSDN article
- **ContinueDebugEvent**, which is used to continue execution after a debug event is triggered

When using these API's, we are attaching to a process and waiting for it to trigger one of several debug events, such as creating a thread or encountering an exception. However, when debugging a target we do not have the source code to, this will limit us to only breaking on thread and process creation events.

To be able to trigger a breakpoint on an address, we will need to use an **interrupt** instruction. Interrupt instructions are a special set of software instructions that invoke a special interrupt handler on the CPU. One of these instructions, **int 3**, will trigger a breakpoint when executed. Its opcode is `0xCC`.

We can utilize this behavior to set a breakpoint on any instruction. Before we attach a debugger to a process, we will use **WriteProcessMemory** to write `0xCC` to the instruction we wish to break on. We will then listen for debug events like normal. When we get a breakpoint event, we will restore the instruction to its original form and continue execution. By doing this, we can set breakpoints on any instruction in targets that we do not have the source control to.

The full source code for the debugger discussed in this chapter is available in Appendix A.

7.5.4 Writing the Int 3 Instruction

To write our **int 3** instruction into the target, we will use an approach covered in previous chapters. First, we will iterate over all processes in the system using **CreateToolhelp32Snapshot** and locate the Assault Cube process (**ac_client.exe**). Then, we will open a handle to the process, and use that handle to write `0xCC` (the opcode for **int 3**) over the instruction at `0x0046366C`:

```
HANDLE process_snapshot = NULL;
HANDLE process_handle = NULL;

DWORD pid;
DWORD bytes_written = 0;

BYTE instruction_break = 0xcc;

PROCESSENTRY32 pe32 = { 0 };

pe32.dwSize = sizeof(PROCESSENTRY32);

process_snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
Process32First(process_snapshot, &pe32);

do {
    if (wcscmp(pe32.szExeFile, L"ac_client.exe") == 0) {
        pid = pe32.th32ProcessID;
```

```
        process_handle = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);
        WriteProcessMemory(process_handle, (void*)0x0046366C,
&instruction_break, 1, &bytes_written);
    }
} while (Process32Next(process_snapshot, &pe32));
```

Since we will need the process identifier (or pid) of Assault Cube for the **DebugActiveProcess** API, we also store the **pid** for later use.

7.5.5 Main Debugger Loop

Next, we can use an identical model discussed on MSDN to attach to the target and handle debugger events. The code provided on MSDN enters a permanent loop that checks for debugging events and then continues execution when encountering an event.

```
DEBUG_EVENT debugEvent = { 0 };

DWORD continueStatus = DBG_CONTINUE;

DebugActiveProcess(pid);

for (;;) {
    continueStatus = DBG_CONTINUE;

    if (!WaitForDebugEvent(&debugEvent, INFINITE))
        return 0;

    switch (debugEvent.dwDebugEventCode) {
    case EXCEPTION_DEBUG_EVENT:
        switch (debugEvent.u.Exception.ExceptionRecord.ExceptionCode)
        {
        case EXCEPTION_BREAKPOINT:
            continueStatus = DBG_CONTINUE;
            break;
        default:
            continueStatus = DBG_EXCEPTION_NOT_HANDLED;
            break;
        }
        break;
    default:
```

```
        continueStatus = DBG_EXCEPTION_NOT_HANDLED;
        break;
    }

    ContinueDebugEvent(debugEvent.dwProcessId, debugEvent.dwThreadId,
continueStatus);
}

CloseHandle(process_handle);
```

7.5.6 Handling the Breakpoint

With this structure setup, we can now begin handling debugger events. First, let's verify that our **int 3** breakpoint actually worked with a print statement:

```
case EXCEPTION_BREAKPOINT:
    printf("Breakpoint hit");

    continueStatus = DBG_CONTINUE;
    break;
```

Make sure Assault Cube is running and run the debugger we have built so far. It should immediately print out **Breakpoint hit**. If you then fire, it will print out **Breakpoint hit** again before the game crashes. This indicates that our breakpoint was set successfully.

However, crashing the target is not ideal. To fix this, we will need to adjust two things:

1. Only trigger our breakpoint when the instruction is executed and not when we first run our program.
2. Restore the original instruction after our breakpoint is executed.

When we first attach to a process, a breakpoint exception is triggered. Since we only want to handle our breakpoint on the instruction, we will ignore this first exception:

```
bool first_break_has_occurred = false;
case EXCEPTION_BREAKPOINT:
    if (first_break_has_occurred) {
        //only handle breakpoint events after the first exception
    }
```

```
    first_break_has_occurred = true;
```

Next, we can handle the crash that occurs after our breakpoint is triggered. This crash occurs because we have replaced the original **mov** opcode (`0x8b`) with our interrupt. After executing our interrupt and our handling of the debug event, the game tries to execute the next opcode, which is not valid. To resolve this, we need to restore the **mov** instruction after handling our debug event.

The **EIP** (extended instruction pointer) register is used to track the current instruction executing. Each time an instruction is executed, it is changed to reflect the next instruction address to execute. When we execute our **int 3** instruction, it is increased by 1. To restore the **mov** instruction, we need to first decrease it.

We can do this by opening the thread responsible for triggering the breakpoint and retrieving the context (registers) of the thread. We can then decrease the **EIP** register and set the thread's context to our new values:

```
HANDLE thread_handle = NULL;
CONTEXT context = { 0 };

thread_handle = OpenThread(THREAD_ALL_ACCESS, true, debugEvent.dwThreadId);
if (thread_handle != NULL) {
    context.ContextFlags = CONTEXT_ALL;
    GetThreadContext(thread_handle, &context);

    context.Eip--;

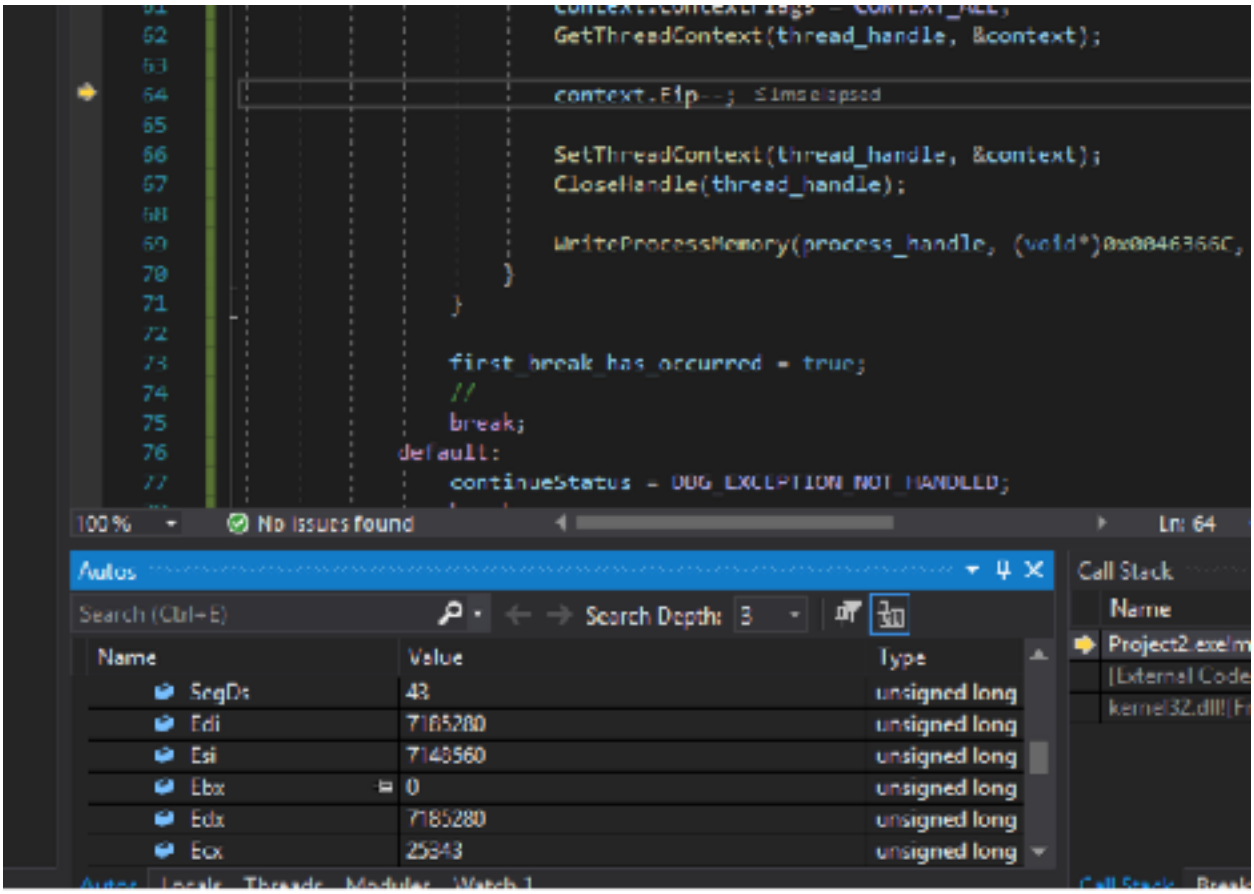
    SetThreadContext(thread_handle, &context);
    CloseHandle(thread_handle);
```

EIP will now point to the original **mov** instruction address again (`0x0046366C`). However, the instruction at this location will still be **int 3**. To fix this, we can use **WriteProcessMemory** to write the original opcode back to the address:

```
WriteProcessMemory(process_handle, (void*)0x0046366C, &instruction_normal, 1,
&bytes_written);
```

With this change, Assault Cube will no longer crash when our breakpoint is triggered. In addition, we can set a breakpoint on the **context.Eip--** line of code and verify that we can view the contents of all registers when our breakpoint is triggered:

Playrix Ex. 1024, page 407



The same approach used to modify **EIP** can be used to modify other registers as well.

Playrix Ex. 1024, page 408

7.6 Call Logger

7.6.1 Target

Our target in this chapter will be Wesnoth 1.14.9.

7.6.2 Overview

When reversing complex applications like video games, one of the most difficult steps is establishing a context inside the application. While there are many techniques to establish a context, one approach is to create a modified debugger that logs all **call** instructions executed by the application. Actions can then be executed in the game, such as clicking a button, and all the related calls can be observed. The logged calls can then be used to establish a context and begin reversing the target.

Our goal in this chapter is to modify the debugger we created in the previous chapter to log all **call** instructions made by the target. The full code for this chapter is available in Appendix A.

7.6.3 Locating the Main Module

In the previous chapter, we wrote a break instruction to a single location inside Assault Cube. Our target for this chapter will be Wesnoth. Therefore, we will modify the code responsible for locating the process's pid to find the Wesnoth process and remove the code responsible for writing the single breakpoint:

```
do {
    if (wcscmp(pe32.szExeFile, L"wesnoth.exe") == 0) {
        pid = pe32.th32ProcessID;

        process_handle = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);
    }
} while (Process32Next(process_snapshot, &pe32));
```

For this tool, we will only log calls in the main game module and not in external DLL's, such as user32.dll. To determine the beginning and end address of the main module, we will first use the EnumProcessModules API to retrieve a list of all loaded modules. Then, we will use the GetModuleInformation API to retrieve the address space of the first module, which always represents the main game module. We will execute this code in the first debug event that occurs in the target (when we attach our debugger to the process):

```
HMODULE modules[128] = { 0 };
MODULEINFO module_info = { 0 };

DWORD bytes_read = 0;

if (!first_break_has_occurred) {
    EnumProcessModules(process_handle, modules, sizeof(modules),
&bytes_read);
    GetModuleInformation(process_handle, modules[0], &module_info,
sizeof(module_info));
```

The **GetModuleInformation** API will fill **module_info.SizeOfImage** with the size of the main module, and **module_info.lpBaseOfDll** with the base address of the main module. With this range, we can begin searching for **call** instructions.

Like we have done previously, we will use **ReadProcessMemory** to read the instructions into a buffer. While we would like to read the entire memory of the whole process, this approach will not work. This is because different memory sections of the process have different memory protections. If the section does not allow reading, the call to **ReadProcessMemory** will fail. If we try to read the entire memory of the process in one call, we will encounter a section that fails, and then the entire read will fail.

To deal with this, we will instead read the memory in sections. These sections are called memory pages, and the default memory page size in Windows is 4096 bytes. As such, we will create a loop to read 4096 bytes of instructions at a time. We will use the **bytes_read** parameter to determine how many bytes of the page were actually read:

```
#define READ_PAGE_SIZE 4096

unsigned char instructions[READ_PAGE_SIZE] = { 0 };

for (DWORD i = 0; i < module_info.SizeOfImage; i += READ_PAGE_SIZE) {
    ReadProcessMemory(process_handle, (LPVOID)((DWORD)module_info.lpBaseOfDll
+ i), &instructions, READ_PAGE_SIZE, &bytes_read);
```


Playrix Ex. 1024, page 410

```
    for (DWORD c = 0; c < bytes_read; c++) {

    }
}
```

7.6.4 Locating Calls

Next, we will locate the **call** instructions in each page of memory. We know that the opcode for the **call** instruction is `0xe8`. While iterating over each instruction, we will check to see if it is `0xe8`:

```
BYTE instruction_call = 0xe8;
...
for (DWORD c = 0; c < bytes_read; c++) {
    if (instructions[c] == instruction_call) {

    }
}
```

However, not all `0xe8`'s represent **call** instructions. For example, the opcode for the **add eax, ebp** instruction is `0x01 e8`. We need to ensure that we do not identify these random `0xe8`'s as calls. The easiest way to do that is to read the 4 bytes after the **call**.

As we know from Chapter 2.6, these 4 bytes encode the location of the **call**. By retrieving this location, we can check if the calculated location of these bytes is valid. If not, we can assume that the `0xe8` is not a **call** and use the **continue** instruction to escape this check:

```
DWORD offset = 0;
DWORD call_location = 0;
DWORD call_location_bytes_read = 0;
...
if (instructions[c] == instruction_call) {
    offset = (DWORD)module_info.lpBaseOfDll + i + c;
    ReadProcessMemory(process_handle, (LPVOID)(offset + 1), &call_location,
4, &call_location_bytes_read);

    call_location += offset + 5;
    if (call_location < (DWORD)module_info.lpBaseOfDll || call_location
>(DWORD)module_info.lpBaseOfDll + module_info.SizeOfImage)
```

```
        continue;
```

Finally, we will write a break instruction (`0xcc`) to the location. In addition to **WriteProcessMemory**, we will use the FlushInstructionCache API to make sure our changes are done immediately to the target:

```
BYTE instruction_break = 0xcc;
...
WriteProcessMemory(process_handle, (void*)offset, &instruction_break, 1,
&bytes_written);
FlushInstructionCache(process_handle, (LPVOID)offset, 1);
```

Writing thousands of break instructions to a process can cause the program to crash. To avoid this, we will only write 2000 breakpoints:

```
int breakpoints_set = 0;
...
if (breakpoints_set < 2000) {
    WriteProcessMemory...
    breakpoints_set++;
}
```

7.6.5 Handling Breakpoints

Now that we have written breakpoints to all the calls, we need to handle the breakpoint events. We will start with the same approach that we used in the previous chapter:

```
else {
    thread_handle = OpenThread(THREAD_ALL_ACCESS, true,
debugEvent.dwThreadId);
    if (thread_handle != NULL) {
        context.ContextFlags = CONTEXT_ALL;
        GetThreadContext(thread_handle, &context);

        context.Eip--;

        SetThreadContext(thread_handle, &context);
        CloseHandle(thread_handle);
```


Playrix Ex. 1024, page 412

```
        WriteProcessMemory(process_handle, (void*)context.Eip,
&instruction_call, 1, &bytes_written);
        FlushInstructionCache(process_handle, (LPVOID)context.Eip, 1);
    }
}
```

Like we saw before, this code will decrease **EIP** and restore the original **call** instruction. Then, execution will resume at the **call** and the program will continue execution normally. The downside with this approach is that each breakpoint is only hit once. For our call logger, we want to log each time a **call** is executed. To achieve this behavior, we will use single-step mode.

Single-stepping is a special type of debug event that executes a single instruction before triggering an exception again. To enable single-step mode, we modify the **EFlags** of the current thread like so:

```
context.Eip--;
context.EFlags |= 0x100;
```

Next, we need to handle the single-step event. We will introduce another case for this:

```
case EXCEPTION_SINGLE_STEP:
```

When we receive our exception here, it means that the **call** has finished executing. Ultimately, our goal in this event is to restore the break instruction. We can do this via **WriteProcessMemory** in an identical way to restoring the **call** instruction:

```
thread_handle = OpenThread(THREAD_ALL_ACCESS, true, debugEvent.dwThreadId);
if (thread_handle != NULL) {
    context.ContextFlags = CONTEXT_ALL;
    GetThreadContext(thread_handle, &context);
    CloseHandle(thread_handle);

    WriteProcessMemory(process_handle, (void*)last_call_location,
&instruction_break, 1, &bytes_written);
    FlushInstructionCache(process_handle, (LPVOID)last_call_location, 1);
}
```

With this code in place, our breakpoints will be restored after being triggered.

Playrix Ex. 1024, page 413

7.6.6 Adding Logging

Finally, we will add logging to this code so that we can see the triggered breakpoints. In our debug event, we will store the current location of **EIP**:

```
DWORD last_call_location = 0;
...
last_call_location = context.Eip;
```

Next, in the single-step event, we will add the logging code. We know at this point that we have executed the **call** and we are at the **call**'s location. Now we can use the following print statement to print the **call**'s address and the location called:

```
printf("0x%08x: call 0x%08x\n", last_call_location, context.Eip);
last_call_location = 0;
```

In this chapter, we are only logging the calls as they happen. However, it is possible to modify this code to also hook **ret** instructions. This would allow you to build out a graph showing all calls made by the process and which calls call other calls.

Appendix A

Playrix Ex. 1024, page 415

A.1 Lab VM Setup Script

Referenced in Chapter 1.4.

```
Set-WindowsExplorerOptions -EnableShowHiddenFilesFoldersDrives
-EnableShowProtectedOSFiles -EnableShowFileExtensions
Enable-RemoteDesktop

cinst cheatengine
cinst x64dbg.portable
```

A.2 Wesnoth External Gold Hack

Referenced in Chapter 3.2.

An external memory hack for Wesnoth 1.14.9 that modifies the player's gold.

This code will create a console application that sets the player's gold in Wesnoth 1.14.9 to the value of 555 when run. It makes use of **ReadProcessMemory** and **WriteProcessMemory** to achieve this. The address `0x017EED18` represents the player's base pointer in Wesnoth.

This program must be run as an administrator.

```
// FindWindow, GetWindowThreadProcessId, OpenProcess, ReadProcessMemory, and
WriteProcessMemory are all contained inside windows.h
#include <Windows.h>

int main(int argc, char** argv) {
```

```
    /*
        To use ReadProcessMemory and WriteProcessMemory, we require a handle
to the Wesnoth process.

        To get this handle, we require a process id. The quickest way to get
a process id for a particular
        process is to use GetWindowThreadProcessId.

        GetWindowThreadProcessId requires a window handle (different than a
process handle). To get this
        window handle, we use FindWindow.
    */

    // Find our Wesnoth window. Depending on your language settings, this
might be different.
    HWND wesnoth_window = FindWindow(NULL, L"The Battle for Wesnoth -
1.14.9");

    // Get the process id for the Wesnoth process. GetWindowThreadProcessId
does not return a process id, but
    // rather fills a provided variable with its value, hence the &.
    DWORD process_id = 0;
    GetWindowThreadProcessId(wesnoth_window, &process_id);

    // Open our Wesnoth process. PROCESS_ALL_ACCESS means we can both read
and write to the process. However,
    // it also means that this program must be executed as an administrator
to work.
    HANDLE wesnoth_process = OpenProcess(PROCESS_ALL_ACCESS, true,
process_id);

    // Read the value at 0x017EED18 and place its value into the variable
gold_value.
    DWORD gold_value = 0;
    DWORD bytes_read = 0;
    ReadProcessMemory(wesnoth_process, (void*)0x017EED18, &gold_value, 4,
&bytes_read);

    // Add 0xA90 to the value read from the last step and then read the value
at that new address. These
    // offsets are covered in https://gamehacking.academy/lesson/13
    gold_value += 0xA90;
    ReadProcessMemory(wesnoth_process, (void*)gold_value, &gold_value, 4,
&bytes_read);
```

```
    // Add 4 to the gold_value, which will then be pointing at the player's
current gold address.
    // Write the value of new_gold_value (555) into this address
    gold_value += 4;
    DWORD new_gold_value = 555;
    DWORD bytes_written = 0;
    WriteProcessMemory(wesnoth_process, (void*)gold_value, &new_gold_value,
4, &bytes_written);

    return 0;
}
```

A.3 Wesnoth Internal Gold Hack

Referenced in Chapter 3.3.

An internal memory hack for Wesnoth 1.14.9 that modifies the player's gold.

This is an example of a DLL that needs to be injected into Wesnoth. Once injected, it creates a thread within the game. This thread waits for a player to hit the “M” key and then uses a series of pointers to directly set the player's gold value to 999.

This must be injected into the Wesnoth process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
// CreateThread and GetAsyncKeyState are defined within windows.h
#include <Windows.h>

// Our injected thread. Since we want to monitor for the user's key presses,
// we use a while loop to ensure that this thread never exits. Inside the
thread, we
// check if the "M" key is being held down. If so, we directly access the
game's memory
// through the use of pointers. We use these pointers to set our player's
gold value.
void injected_thread() {
```


Playrix Ex. 1024, page 418

```
	while (true) {
		if (GetAsyncKeyState('M')) {
			DWORD* player_base = (DWORD*)0x017EED18;
			DWORD* game_base = (DWORD*)(*player_base + 0xA90);
			DWORD* gold = (DWORD*)(*game_base + 4);
			*gold = 999;
		}

		// So our thread doesn't constantly run, we have it pause
execution for a millisecond.
		// This allows the processor to schedule other tasks.
		Sleep(1);
	}
}

// When injected, the parent process looks for the DLL's DllMain, similar to
the main function in regular executables.
// There are several events that can occur, the most important one for us
being DLL_PROCESS_ATTACH. This occurs when the
// DLL is fully loaded into the process' memory.
//
// Once loaded, we create a thread. This thread will run in the background of
the game as long as the process remains open.
// The code that this thread will execute is shown above.
BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpvReserved ) {
	if (fdwReason == DLL_PROCESS_ATTACH) {
		CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread,
NULL, 0, NULL);
	}

	return true;
}
```

A.4 Wesnoth Code Cave DLL

Referenced in Chapter 3.4.

A DLL that redirects the *Terrain Description* function in Wesnoth 1.14.9 to a custom function that sets the player's gold to 888.

This custom function then recreates the *Terrain Description* function and returns execution to the program.

This is done through the use of a code cave. When injected, the DLL modifies the function that displays the terrain description and changes the code to jump to the code cave function defined in the DLL. The code cave function then saves the registers, sets the gold to 888, and restores the original modified instructions before returning to the original calling code.

This must be injected into the Wesnoth process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>

DWORD* player_base;
DWORD* game_base;
DWORD* gold;
DWORD ret_address = 0xCCAF90;

// Our code cave that program execution will jump to. The declspec naked
attribute tells the compiler to not add any function
// headers around the assembled code
__declspec(naked) void codecave() {
      // Asm blocks allow you to write pure assembly
      // In this case, we use it to save all the registers
      __asm {
            pushad
      }

      // Set the player's gold in the same method discussed in https://
gamehacking.academy/lesson/16
      player_base = (DWORD*)0x017EED18;
      game_base = (DWORD*)(*player_base + 0xA90);
      gold = (DWORD*)(*game_base + 4);
      *gold = 888;

      // Restore the registers and then recreate the original instructions
that we overwrote
      // After those, jump back to the instruction after the one we overwrote
      _asm {
            popad
```


Playrix Ex. 1024, page 420

```
            mov eax, dword ptr ds:[ecx]
            lea esi,dword ptr ds:[esi]
            jmp ret_address
      }
}

// When our DLL is attached, unprotect the memory at the code we wish to
write at
// Then set the first opcode to E9, or jump
// Caculate the location using the formula: new_location -
original_location+5
// Finally, since the original instructions totalled 6 bytes, NOP out the
last remaining byte
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      DWORD old_protect;
      unsigned char* hook_location = (unsigned char*)0x00CCAF8A;

      if (fdwReason == DLL_PROCESS_ATTACH) {
            VirtualProtect((void*)hook_location, 6, PAGE_EXECUTE_READWRITE,
&old_protect);
            *hook_location = 0xE9;
            *(DWORD*)(hook_location + 1) = (DWORD)&codecave -
((DWORD)hook_location + 5);
            *(hook_location + 5) = 0x90;
      }

      return true;
}
```

A.5 Wesnoth Stathack

Referenced in Chapter 4.1.

A stathack for Wesnoth 1.14.9 that displays the second player's gold whenever the *Terrain Description* box is shown.

This is done through the use of a code cave. When injected, the DLL modifies the function that displays the terrain description and changes the code to jump to the code cave function defined in the DLL. The code cave function then saves the registers, gets the second player's gold, and writes the value into the buffer used by the game to display the *Terrain Description* text. It then jumps back to the *Terrain Description* method and displays the original description with the gold prepended to it.

This must be injected into the Wesnoth process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>
#include <stdio.h>

DWORD* player_base;
DWORD* game_base;
DWORD* gold;

// Original address called by the game
DWORD ori_call_address = 0x5E9630;

DWORD ret_address = 0x5ED12E;

// Buffer to hold the second player's gold value
char gold_byte_array[4] = { 0 };

// Our code cave that program execution will jump to. The declspec naked
attribute tells the compiler to not add any function
// headers around the assembled code
__declspec(naked) void codecave() {
      // Asm blocks allow you to write pure assembly
      // In this case, we use it to save all the registers
      __asm {
            pushad
      }

      // Get the second player's gold value based off the base pointer
      player_base = (DWORD*)0x017EED18;
      game_base = (DWORD*)(*player_base + 0xA90);
      gold = (DWORD*)(*game_base + 0x274);

      // Convert the gold value to its ASCII representation
      sprintf_s(gold_byte_array, 4, "%d", *gold);

      // Restore the registers corrupted by sprintf and save them again
```

```
	// Then, load the buffer from edx, and place each byte of the second
player's gold
	// value into the buffer
	__asm {
		popad
		pushad
		mov eax, dword ptr ds:[edx]
		mov bl, gold_byte_array[0]
		mov byte ptr ds:[eax], bl
		mov bl, gold_byte_array[1]
		mov byte ptr ds:[eax + 1], bl
		mov bl, gold_byte_array[2]
		mov byte ptr ds:[eax + 2], bl
	}

	// Restore the registers and then recreate the original instructions
that we overwrote
	// After those, jump back to the instruction after the one we overwrote
	_asm {
		popad
		call ori_call_address
		jmp ret_address
	}
}

// When our DLL is attached, unprotect the memory at the code we wish to
write at
// Then set the first opcode to E9, or jump
// Calculate the location using the formula: new_location - original_location
+ 5
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
	DWORD old_protect;
	unsigned char* hook_location = (unsigned char*)0x5ED129;

	if (fdwReason == DLL_PROCESS_ATTACH) {
		VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
		*hook_location = 0xE9;
		*(DWORD*)(hook_location + 1) = (DWORD)&codecave -
((DWORD)hook_location + 5);
	}

	return true;
}
```

A.6 Wesnoth Map Hack

Referenced in Chapter 4.2.

A map hack for Wesnoth 1.14.9 that reveals the entire map by removing in-game fog-of-war.

This is done by modifying the game's code responsible for re-setting all tiles to a hidden state at the start of a player's turn. This code is modified to set all tiles to a visible state (-1, or `0xFFFFFFFF` in Wesnoth). To fit in the space of the previous instructions, this is done through the use of an **or dword ptr ds:[esi],0xFFFFFFFF** instruction (opcode `0x830EFF`), along with several **nop**'s (**0x90**).

This must be injected into the Wesnoth process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>

// The new opcodes to write into the game's code
unsigned char new_bytes[8] = { 0x90, 0x90, 0x90, 0x83, 0x0E, 0xFF, 0x90, 0x90
};

// When our DLL is attached, first unprotect the memory responsible for
resetting the tiles in the game
// Then, write our new opcodes into that memory location
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      DWORD old_protect;
      unsigned char* hook_location = (unsigned char*)0x6CD519;

      if (fdwReason == DLL_PROCESS_ATTACH) {
            VirtualProtect((void*)hook_location, 8, PAGE_EXECUTE_READWRITE,
&old_protect);
            for (int i = 0; i < sizeof(new_bytes); i++) {
                  *(hook_location + i) = new_bytes[i];
            }
      }
```

```
      return true;
}
```

A.7 Wyrmsun Macrobot

Referenced in Chapter 4.3.

A hack for Wyrmsun version 5.0.1 that will automatically create worker units out of the currently selected structure when a player's gold is over 3000.

It accomplishes this by filling the current unit buffer with worker data and then calling the create unit function in the game.

After injecting this hack, go in game and recruit a worker. Then select a structure as you collect gold. You will notice workers being queued automatically. Due to the way Wyrmsun handles recruitment, it is possible to create units out of whatever is selected, including other units.

This must be injected into the Wyrmsun process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>

HANDLE wyrmsun_base;

DWORD* base;
DWORD* unitbase;
DWORD recruit_unit_ret_address;
DWORD recruit_unit_call_address;
unsigned char unitdata[0x110];
bool init = false;

DWORD gameloop_ret_address;
DWORD gameloop_call_address;
DWORD *gold_base, *gold;
```


Playrix Ex. 1024, page 425

```
// The recruit unit code cave hooks the game's recruit unit function
// It's main job is to copy a valid buffer of data for a worker unit
// instead of having to reverse the structure
__declspec(naked) void recruit_unit_codecave() {
    __asm {
        pushad
        mov base, ecx
    }

    unitbase = (DWORD*)(*base);
    memcpy(unitdata, unitbase, 0x110);
    init = true;

    _asm {
        popad
        push ecx
        mov ecx, esi
        call recruit_unit_call_address
        jmp recruit_unit_ret_address
    }
}

// In the main game loop, our code cave will check the current player's gold
// If it is over 3000, and we have a valid worker buffer, call the recruit
unit function
// with worker data.
__declspec(naked) void gameloop_codecave() {
    __asm {
        pushad
    }

    gold_base = (DWORD*)((DWORD)wyrmsun_base + 0x0061A504);
    gold = (DWORD*)(*gold_base + 0x78);
    gold = (DWORD*)(*gold + 4);
    gold = (DWORD*)(*gold + 8);
    gold = (DWORD*)(*gold + 4);
    gold = (DWORD*)(*gold);
    gold = (DWORD*)(*gold + 0x14);

    if (init && *gold > 3000) {
        memcpy(unitbase, unitdata, 0x110);
        __asm {
            mov ecx, base
            push ecx
```

```
            call recruit_unit_call_address
        }
    }


    __asm {
        popad
        call gameloop_call_address
        jmp gameloop_ret_address
    }
}

// When our DLL is attached, unprotect the memory at the code we wish to
write at
// Then set the first opcode to E9, or jump
// Caculate the location using the formula: new_location -
original_location+5
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      DWORD old_protect;

      if (fdwReason == DLL_PROCESS_ATTACH) {
            // Since Wyrmsun loads code dynamically, we need to calculate
offsets based of the base address of the main module
            wyrmsun_base = GetModuleHandle(L"wyrmsun.exe");

            unsigned char* hook_location = (unsigned char*)
((DWORD)wyrmsun_base + 0x223471);
            recruit_unit_ret_address = (DWORD)hook_location + 8;
            recruit_unit_call_address = (DWORD)wyrmsun_base + 0x2CF7;

            VirtualProtect((void*)hook_location, 8, PAGE_EXECUTE_READWRITE,
&old_protect);
            *hook_location = 0xE9;
            *(DWORD*)(hook_location + 1) = (DWORD)&recruit_unit_codecave -
((DWORD)hook_location + 5);
            *(hook_location + 5) = 0x90;
            *(hook_location + 6) = 0x90;
            *(hook_location + 7) = 0x90;

            hook_location = (unsigned char*)((DWORD)wyrmsun_base + 0x385D34);
            gameloop_ret_address = (DWORD)hook_location + 5;
            gameloop_call_address = (DWORD)wyrmsun_base + 0xDBCA;
```

```
            VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
            *hook_location = 0xE9;
            *(DWORD*)(hook_location + 1) = (DWORD)&gameloop_codecave -
((DWORD)hook_location + 5);
      }

      return true;
}
```

A.8 Urban Terror Memory Wallhack

Referenced in Chapter 5.2.

A wallhack for Urban Terror 4.3.4 that reveals entities through walls by disabling depth testing.

This is done by modifying each entity's render flag, which is responsible for determining how the entity should be rendered. By setting this value to the in-game value for disabled depth testing (`0xD`), entities will be drawn whether or not they should be visible. The code hooked is a **mov** instruction, which occurs after **ebx** is loaded with a valid entity structure.

This must be injected into the Urban Terror process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>

DWORD ret_address = 0x0052D303;

// Our code cave that program execution will jump to. The declspec naked
attribute tells the compiler to not add any function
// headers around the assembled code
__declspec(naked) void codecave() {
      // Asm blocks allow you to write pure assembly
      // In this case, we use it to save all the registers
```

```
	// Then set the entity's render value at [ebx+4] to disabled depth
testing (0xD)
	// Then we restore the registers, recreate the original instruction,
and jump back to the program code
	__asm {
		pushad
		mov dword ptr ds:[ebx+4], 0xD
		popad
		mov dword ptr ds:[0x102AE98], ebx

		jmp ret_address
	}
}

// When our DLL is attached, unprotect the memory at the code we wish to
write at
// Then set the first opcode to E9, or jump
// Caculate the location using the formula: new_location -
original_location+5
// Finally, since the original instructions totalled 6 bytes, NOP out the
last remaining byte
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
	DWORD old_protect;
	unsigned char* hook_location = (unsigned char*)0x0052D2FD;

	if (fdwReason == DLL_PROCESS_ATTACH) {
		VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
		*hook_location = 0xE9;
		*(DWORD*)(hook_location + 1) = (DWORD)&codecave -
((DWORD)hook_location + 5);
		*(hook_location + 5) = 0x90;
	}

	return true;
}
```

A.9 Urban Terror OpenGL Wallhack

Referenced in Chapter 5.3.

A wallhack for Urban Terror 4.3.4 that reveals entities through walls by hooking the game's OpenGL function **glDrawElements** and disabling depth testing for OpenGL.

This is done by locating the **glDrawElements** function inside the OpenGL library and creating a code cave at the start of the function. In the code cave, we check the number of vertices associated with the element. If it is over 500, we call **glDepthRange** to clear the depth clipping plane and **glDepthFunc** to disable depth testing. Otherwise, we call these same functions to re-enable the depth clipping plane and re-enable depth testing.

This DLL must be injected into the Urban Terror process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>

HMODULE openGLHandle = NULL;

// Function pointers for two OpenGL functions that we will dynamically
populate
// after injecting our DLL
void (__stdcall *glDepthFunc)(unsigned int) = NULL;
void (__stdcall* glDepthRange)(double, double) = NULL;

unsigned char* hook_location;

DWORD ret_address = 0;
DWORD old_protect;
DWORD count = 0;

// Code cave that runs before glDrawElements is called
__declspec(naked) void codecave() {
      // First, we retrieve the count parameter from the original call.
```

```
	// Then, we retrieve the value of the count parameter, which specifies
the amount
	// of indicies to be rendered
	__asm {
		pushad
		mov eax, dword ptr ds : [esp + 0x10]
		mov count, eax
		popad
		pushad
	}

	// If the count is over 500, we clear the depth clipping plane and then
	// set the depth function to GL_ALWAYS
	if (count > 500) {
		(*glDepthRange)(0.0, 0.0);
		(*glDepthFunc)(0x207);
	}
	else {
		// Otherwise, restore the depth clipping plane to the game's
default value and then
		// set the depth function to GL_LEQUAL
		(*glDepthRange)(0.0, 1.0);
		(*glDepthFunc)(0x203);
	}

	// Finally, restore the original instruction and jump back
	__asm {
		popad
		mov esi, dword ptr ds : [esi + 0xA18]
		jmp ret_address
	}
}

// The injected thread responsible for creating our hooks
void injected_thread() {
	while (true) {
		// Since OpenGL will be loaded dynamically into the process, our
thread needs to wait
		// until it sees that the OpenGL module has been loaded.
		if (openGLHandle == NULL) {
			openGLHandle = GetModuleHandle(L"opengl32.dll");
		}

		// Once loaded, we first find the location of the two depth
functions we are using in our
```

```
            // code caves above
            if (openGLHandle != NULL && glDepthFunc == NULL) {
                  glDepthFunc = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glDepthFunc");
                  glDepthRange = (void(__stdcall*)(double,
double))GetProcAddress(openGLHandle, "glDepthRange");

                  // Then we find the location of glDrawElements and offset
to an instruction that is easy to hook
                  hook_location = (unsigned
char*)GetProcAddress(openGLHandle, "glDrawElements");
                  hook_location += 0x16;

                  // For the hook, we unprotect the memory at the code we
wish to write at
                  // Then set the first opcode to E9, or jump
                  // Caculate the location using the formula: new_location -
original_location+5
                  // And finally, since the first original instructions
totalled 6 bytes, NOP out the last remaining byte
                  VirtualProtect((void*)hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
                  *hook_location = 0xE9;
                  *(DWORD*)(hook_location + 1) = (DWORD)&codecave -
((DWORD)hook_location + 5);
                  *(hook_location + 5) = 0x90;

                  // Since OpenGL is loaded dynamically, we need to
dynamically calculate the return address
                  ret_address = (DWORD)(hook_location + 0x6);
            }

            // So our thread doesn't constantly run, we have it pause
execution for a millisecond.
            // This allows the processor to schedule other tasks.
            Sleep(1);
      }
}

// When our DLL is loaded, create a thread in the process to create the hook
// We need to do this as our DLL might be loaded before OpenGL is loaded by
the process
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      if (fdwReason == DLL_PROCESS_ATTACH) {
```

```
            CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread,
NULL, 0, NULL);
      }

      return true;
}
```

A.10 Urban Terror OpenGL Chams

Referenced in Chapter 5.4.

A chams hack for Urban Terror 4.3.4 that both reveals entities through walls and changes these models to a bright red color. It works by hooking the game's OpenGL function **glDrawElements** and disabling depth testing and textures for OpenGL.

This is done by locating the **glDrawElements** function inside the OpenGL library and creating a code cave at the start of the function. In the code cave, we check the number of vertices associated with the element. If it is over 500, we call **glDepthRange** to clear the depth clipping plane and **glDepthFunc** to disable depth testing. We then disable texture and color arrays and enable color material before setting the color to red with **glColor**.

Otherwise, we call these same functions to re-enable the depth clipping plane, re-enable depth testing, and re-enable textures.

This DLL must be injected into the Urban Terror process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>
#include <vector>

HMODULE openGLHandle = NULL;

// Function pointers for two OpenGL functions that we will dynamically
populate
```


Playrix Ex. 1024, page 433

```
// after injecting our DLL
void(__stdcall* glDepthFunc)(unsigned int) = NULL;
void(__stdcall* glDepthRange)(double, double) = NULL;

void(__stdcall* glColor4f)(float, float, float, float) = NULL;
void(__stdcall* glEnable)(unsigned int) = NULL;
void(__stdcall* glDisable)(unsigned int) = NULL;
void(__stdcall* glEnableClientState)(unsigned int) = NULL;
void(__stdcall* glDisableClientState)(unsigned int) = NULL;

unsigned char* hook_location;

DWORD ret_address = 0;
DWORD old_protect;
DWORD count = 0;

// Code cave that runs before glDrawElements is called
__declspec(naked) void codecave() {
      // First, we retrieve the count parameter from the original call.
      // Then, we retrieve the value of the count parameter, which specifies
the amount
      // of indicies to be rendered
      __asm {
            pushad
            mov eax, dword ptr ds : [esp + 0x10]
            mov count, eax
            popad
            pushad
      }

      // If the count is over 500, we clear the depth clipping plane and then
      // set the depth function to GL_ALWAYS
      // We then disable color and texture arrays and enable color materials
before setting
      // the color to red
      if (count > 500) {
            (*glDepthRange)(0.0, 0.0);
            (*glDepthFunc)(0x207);

            (*glDisableClientState)(0x8078);
            (*glDisableClientState)(0x8076);
            (*glEnable)(0x0B57);
            (*glColor4f)(1.0f, 0.6f, 0.6f, 1.0f);
      }
      else {
```

```
            // Otherwise, restore the depth clipping plane to the game's
default value and then
            // set the depth function to GL_LEQUAL and restore textures
            (*glDepthRange)(0.0, 1.0);
            (*glDepthFunc)(0x203);

            (*glEnableClientState)(0x8078);
            (*glEnableClientState)(0x8076);
            (*glDisable)(0x0B57);
            (*glColor4f)(1.0f, 1.0f, 1.0f, 1.0f);
      }

      // Finally, restore the original instruction and jump back
      __asm {
            popad
            mov esi, dword ptr ds : [esi + 0xA18]
            jmp ret_address
      }
}

// The injected thread responsible for creating our hooks
void injected_thread() {
      while (true) {
            // Since OpenGL will be loaded dynamically into the process, our
thread needs to wait
            // until it sees that the OpenGL module has been loaded.
            if (openGLHandle == NULL) {
                  openGLHandle = GetModuleHandle(L"opengl32.dll");
            }

            // Once loaded, we first find the location of the functions we
are using in our
            // code caves above
            if (openGLHandle != NULL && glDepthFunc == NULL) {
                  glDepthFunc = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glDepthFunc");
                  glDepthRange = (void(__stdcall*)(double,
double))GetProcAddress(openGLHandle, "glDepthRange");
                  glColor4f = (void(__stdcall*)(float, float, float,
float))GetProcAddress(openGLHandle, "glColor4f");
                  glEnable = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glEnable");
                  glDisable = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glDisable");
```

```
                  glEnableClientState = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glEnableClientState");
                  glDisableClientState = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glDisableClientState");

                  // Then we find the location of glDrawElements and offset
to an instruction that is easy to hook
                  hook_location = (unsigned
char*)GetProcAddress(openGLHandle, "glDrawElements");
                  hook_location += 0x16;

                  // For the hook, we unprotect the memory at the code we
wish to write at
                  // Then set the first opcode to E9, or jump
                  // Caculate the location using the formula: new_location -
original_location+5
                  // And finally, since the first original instructions
totalled 6 bytes, NOP out the last remaining byte
                  VirtualProtect((void*)hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
                  *hook_location = 0xE9;
                  *(DWORD*)(hook_location + 1) = (DWORD)&codecave -
((DWORD)hook_location + 5);
                  *(hook_location + 5) = 0x90;

                  // Since OpenGL is loaded dynamically, we need to
dynamically calculate the return address
                  ret_address = (DWORD)(hook_location + 0x6);
            }

            // So our thread doesn't constantly run, we have it pause
execution for a millisecond.
            // This allows the processor to schedule other tasks.
            Sleep(1);
      }
}

// When our DLL is loaded, create a thread in the process to create the hook
// We need to do this as our DLL might be loaded before OpenGL is loaded by
the process
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      if (fdwReason == DLL_PROCESS_ATTACH) {
            CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread,
NULL, 0, NULL);
```

```
	}

	return true;
}
```

A.11 Assault Cube Triggerbot

Referenced in Chapter 5.5.

A triggerbot for Assault Cube 1.2.0.2 that fires the player's weapon whenever the crosshair goes over another player.

This works by hooking the game's feature that displays nametags when you hover over a player. Whenever a player is hovered over, our code cave will send a mouse down event to the game. Otherwise, it will send a mouse up event to stop firing.

This must be injected into the Assault Cube process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>

DWORD ori_call_address = 0x4607C0;
DWORD ori_jump_address = 0x0040ADA2;

INPUT input = { 0 };
DWORD edi_value = 0;

// Our code cave that program execution will jump to. The declspec naked
attribute tells the compiler to not add any function
// headers around the assembled code
__declspec(naked) void codecave() {
	// Asm blocks allow you to write pure assembly
	// In this case, we use it to call the function we hooked and save all
the registers
```

```
	// After we make the call, we move its return value in eax into a
variable
	__asm {
		call ori_call_address
		pushad
		mov edi_value, eax
	}

	// If the result of the call is not zero, then we are looking at a
player
	// Create a mouse event to simulate the left mouse button being pressed
down and send it to the game
	// Otherwise, raise the mouse button up so we stop firing
	if (edi_value != 0) {
		input.type = INPUT_MOUSE;
		input.mi.dwFlags = MOUSEEVENTF_LEFTDOWN;
		SendInput(1, &input, sizeof(INPUT));
	}
	else {
		input.type = INPUT_MOUSE;
		input.mi.dwFlags = MOUSEEVENTF_LEFTUP;
		SendInput(1, &input, sizeof(INPUT));
	}

	// Restore the registers and jump back to original code
	_asm {
		popad
		jmp ori_jump_address
	}
}

// When our DLL is attached, unprotect the memory at the code we wish to
write at
// Then set the first opcode to E9, or jump
// Caculate the location using the formula: new_location -
original_location+5
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
	DWORD old_protect;
	unsigned char* hook_location = (unsigned char*)0x0040AD9D;

	if (fdwReason == DLL_PROCESS_ATTACH) {
		VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
		*hook_location = 0xE9;
```

```
            *(DWORD*)(hook_location + 1) = (DWORD)&codecave -
((DWORD)hook_location + 5);
      }

      return true;
}
```

A.12 Assault Cube Aimbot

Referenced in Chapter 5.6.

An aimbot for Assault Cube 1.2.0.2 that automatically aims at enemy players.

It works by iterating over the enemy list and picking the closest enemy through Euclidean distance. The yaw and pitch required to aim at that enemy are then calculated using arctangents.

This must be injected into the Assault Cube process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>
#include <math.h>

// The atan2f function produces a radian. To convert it to degrees, we need
the value of pi
#define M_PI 3.14159265358979323846

// The player structure for every player in the game
struct Player {
      char unknown1[4];
      float x;
      float y;
      float z;
      char unknown2[0x30];
      float yaw;
      float pitch;
```

```
	char unknown3[0x2f0];
	int dead;
};

// Our player
Player *player = NULL;

// Function to calculate the euclidean distance between two points
float euclidean_distance(float x, float y) {
	return sqrtf((x * x) + (y * y));
}

// This thread contains all of our aimbot code
void injected_thread() {

	while (true) {
		// First, grab the current position and view angles of our player
		DWORD* player_offset = (DWORD*)(0x509B74);
		player = (Player*)(*player_offset);

		// Then, get the current number of players in the game
		int* current_players = (int*)(0x50F500);

		// These variables will be used to hold the closest enemy to us
		float closest_player = -1.0f;
		float closest_yaw = 0.0f;
		float closest_pitch = 0.0f;

		// Iterate through all active enemies
		for (int i = 0; i < *current_players; i++) {
			DWORD* enemy_list = (DWORD*)(0x50F4F8);
			DWORD* enemy_offset = (DWORD*)(*enemy_list + (i*4));
			Player* enemy = (Player*)(*enemy_offset);

			// Make sure the enemy is valid and alive
			if (player != NULL && enemy != NULL && !enemy->dead) {

				// Calculate the absolute position of the enemy away
from us to ensure that our future calculations are correct and based
				// around the origin
				float abspos_x = enemy->x - player->x;
				float abspos_y = enemy->y - player->y;
				float abspos_z = enemy->z - player->z;

				// Calculate our distance from the enemy
```

```
                        float temp_distance = euclidean_distance(abspos_x,
abspos_y);
                        // If this is the closest enemy so far, calculate the
yaw and pitch to aim at them
                        if (closest_player == -1.0f || temp_distance <
closest_player) {
                              closest_player = temp_distance;

                              // Calculate the yaw
                              float azimuth_xy = atan2f(abspos_y, abspos_x);
                              // Convert to degrees
                              float yaw = (float)(azimuth_xy * (180.0 /
M_PI));
                              // Add 90 since the game assumes direct north
is 90 degrees
                              closest_yaw = yaw + 90;

                              // Calculate the pitch
                              // Since Z values are so limited, pick the
larger between x and y to ensure that we
                              // don't look straight at the air when close to
an enemy
                              if (abspos_y < 0) {
                                    abspos_y *= -1;
                              }
                              if (abspos_y < 5) {
                                    if (abspos_x < 0) {
                                          abspos_x *= -1;
                                    }
                                    abspos_y = abspos_x;
                              }
                              float azimuth_z = atan2f(abspos_z, abspos_y);
                              // Covert the value to degrees
                              closest_pitch = (float)(azimuth_z * (180.0 /
M_PI));
                        }
                  }
            }

            // When our loop ends, set our yaw and pitch to the closst values
            player->yaw = closest_yaw;
            player->pitch = closest_pitch;

            // So our thread doesn't constantly run, we have it pause
execution for a millisecond.
```

```
            // This allows the processor to schedule other tasks.
            Sleep(1);
      }
}

// When our DLL is loaded, create a thread in the process that will handle
the aimbot code
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      if (fdwReason == DLL_PROCESS_ATTACH) {
            CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread,
NULL, 0, NULL);
      }

      return true;
}
```

A.13 Assault Cube No Recoil

Referenced in Chapter 5.7.

A hack for Assault Cube 1.2.0.2 that removes all recoil when firing a weapon.

This is done by modifying the game's code responsible for setting the player's recoil. By changing the final instruction, which changes the value of the player's yaw, to instead pop a value that is ignored, the player's yaw is never modified.

This must be injected into the Assault Cube process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>

// The new opcodes to write into the game's code
unsigned char new_bytes[3] = { 0xDD, 0xD8, 0x90 };
```

```
// When our DLL is attached, first unprotect the memory responsible for
adding recoil in the game
// Then, write our new opcodes into that memory location
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
	DWORD old_protect;
	unsigned char* hook_location = (unsigned char*)0x45BAAD;

	if (fdwReason == DLL_PROCESS_ATTACH) {
		VirtualProtect((void*)hook_location, 3, PAGE_EXECUTE_READWRITE,
&old_protect);
		for (int i = 0; i < sizeof(new_bytes); i++) {
			*(hook_location + i) = new_bytes[i];
		}
	}

	return true;
}
```

A.14 Assault Cube ESP

Referenced in Chapter 5.9.

An ESP for Assault Cube 1.2.0.2 that displays information about enemy players above their heads.

It works by iterating over the enemy list and calculating the yaw and pitch required to aim at that enemy using arctangents. This part of the code is taken from the aimbot code.

The difference between the calculated yaw and pitch and our player's yaw and pitch is then used to derive the screen coordinates of the enemy. This is done by adding the difference multiplied by a scaling factor to the middle of the screen.

This must be injected into the Assault Cube process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

```
#include <Windows.h>
#include <math.h>

// The atan2f function produces a radian. To convert it to degrees, we need
the value of pi
#define M_PI 3.14159265358979323846
// The maximum amount of players in an Assault Cube
#define MAX_PLAYERS 32

// The player structure for every player in the game
struct Player {
	char unknown1[4];
	float x;
	float y;
	float z;
	char unknown2[0x30];
	float yaw;
	float pitch;
	char unknown3[0x1DD];
	char name[16];
};

// Our player
Player* player = NULL;

DWORD ret_address = 0x0040BE83;
DWORD text_address = 0x419880;

// Our temporary variables for our print text code cave
const char* text = "";
const char* empty_text = "";

DWORD x = 0;
DWORD y = 0;

// List of calculated ESP values
DWORD x_values[MAX_PLAYERS] = { 0 };
DWORD y_values[MAX_PLAYERS] = { 0 };
char* names[MAX_PLAYERS] = { NULL };

int* current_players;
```

```
// Our code cave responsible for printing text
__declspec(naked) void codecave() {
	current_players = (int*)(0x50F500);

	// First, recreate the original function we hooked but set the text to
empty
	__asm {
		mov ecx, empty_text
		call text_address
		pushad
	}

	// Next, loop through all the current players in the game
	for (int i = 1; i < *current_players; i++) {
		// Store the calculated screen positions in temporary variables
		x = x_values[i];
		y = y_values[i];
		text = names[i];

		// Make sure our text is on screen
		if (x > 2400 || x < 0 || y < 0 || y > 1800) {
			text = "";
		}

		x += 200;

		// Invoke the print text function to display the text
		__asm {
			mov ecx, text
			push y
			push x
			call text_address
			add esp, 8
		}
	}

	// Restore the registers and jump back to the original code
	__asm {
		popad
		jmp ret_address
	}
}

// This thread contains all of the code for calculating our ESP screen
positions
```

```
void injected_thread() {
	while (true) {
		// First, grab the current position and view angles of our player
		DWORD* player_offset = (DWORD*)(0x509B74);
		player = (Player*)(*player_offset);

		// Then, get the current number of players in the game
		current_players = (int*)(0x50F500);

		// Iterate through all active enemies
		for (int i = 1; i < *current_players; i++) {
			DWORD* enemy_list = (DWORD*)(0x50F4F8);
			DWORD* enemy_offset = (DWORD*)(*enemy_list + (i*4));
			Player* enemy = (Player*)(*enemy_offset);

			// Make sure the enemy is valid
			if (player != NULL && enemy != NULL) {
				// Calculate the absolute position of the enemy away
from us to ensure that our future calculations are correct and based
				// around the origin
				float abspos_x = enemy->x - player->x;
				float abspos_y = enemy->y - player->y;
				float abspos_z = enemy->z - player->z;

				// Calculate the yaw
				float azimuth_xy = atan2f(abspos_y, abspos_x);
				// Convert to degrees
				float yaw = (float)(azimuth_xy * (180.0 / M_PI));
				// Add 90 since the game assumes direct north is 90
degrees
				yaw += 90;

				// Calculate the difference between our current yaw
and the calculated yaw to the enemy
				float yaw_dif = player->yaw - yaw;

				// If we are near the 275 angle boundary, our yaw_dif
will be too large, causing our text to appear incorrectly
				// To compensate for that, subtract the yaw_dif from
360 if it is over 180, since our viewport can never show 180 degrees
				if (yaw_dif > 180) {
					yaw_dif = yaw_dif - 360;
				}

				if (yaw_dif < -180) {
```

```
                              yaw_dif = yaw_dif + 360;
                        }

                        // Calculate our X value by adding the yaw_dif times
a scaling factor to the center of the screen horizontally (1200)
                        x_values[i] = (DWORD)(1200 + (yaw_dif * -30));

                        // Calculate the pitch
                        // Since Z values are so limited, pick the larger
between x and y to ensure that we
                        // don't look straight at the air when close to an
enemy
                        if (abspos_y < 0) {
                              abspos_y *= -1;
                        }
                        if (abspos_y < 5) {
                              if (abspos_x < 0) {
                                    abspos_x *= -1;
                              }
                              abspos_y = abspos_x;
                        }
                        float azimuth_z = atan2f(abspos_z, abspos_y);
                        // Covert the value to degrees
                        float pitch = (float)(azimuth_z * (180.0 / M_PI));

                        // Same as above but for pitch
                        float pitch_dif = player->pitch - pitch;

                        // Calculate our Y value by adding the pitch_dif
times a scaling factor to the center of the screen vertically (900)
                        y_values[i] = (DWORD)(900 + ((pitch_dif) * 25));

                        // Set the name to the enemy name
                        names[i] = enemy->name;
                  }
            }

            // So our thread doesn't constantly run, we have it pause
execution for a millisecond.
            // This allows the processor to schedule other tasks.
            Sleep(1);
      }
}
```

```
// When our DLL is loaded, create a thread in the process that will handle
the aimbot code
// Then, create a code cave for our print text function
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
	DWORD old_protect;
	unsigned char* hook_location = (unsigned char*)0x0040BE7E;

	if (fdwReason == DLL_PROCESS_ATTACH) {
		CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread,
NULL, 0, NULL);

		VirtualProtect((void*)hook_location, 5, PAGE_EXECUTE_READWRITE,
&old_protect);
		*hook_location = 0xE9;
		*(DWORD*)(hook_location + 1) = (DWORD)&codecave -
((DWORD)hook_location + 5);
	}

	return true;
}
```

A.15 Assault Cube Multihack

Referenced in Chapter 5.10.

Combined

The initial starting point of the multihack code, in which we combine all the various source code we had from the previous lessons into one massive file.

```
#include <Windows.h>
#include <math.h>
```

```
#define M_PI 3.14159265358979323846
#define MAX_PLAYERS 32

HMODULE openGLHandle = NULL;
void(__stdcall* glDepthFunc)(unsigned int) = NULL;
unsigned char* opengl_hook_location;
DWORD opengl_ret_address = 0;

DWORD triggerbot_ori_call_address = 0x4607C0;
DWORD triggerbot_ori_jump_address = 0x0040ADA2;
INPUT input = { 0 };
DWORD edi_value = 0;

// The player structure for every player in the game
struct Player {
	char unknown1[4];
	float x;
	float y;
	float z;
	char unknown2[0x30];
	float yaw;
	float pitch;
	char unknown3[0x1DD];
	char name[16];
	char unknown4[0x103];
	int dead;
};

// Our player
Player* player = NULL;

DWORD esp_ret_address = 0x0040BE83;
DWORD text_address = 0x419880;

// Our temporary variables for our print text code cave
const char* text = "";
const char* empty_text = "";

DWORD x = 0;
DWORD y = 0;

// List of calculated ESP values
DWORD x_values[MAX_PLAYERS] = { 0 };
DWORD y_values[MAX_PLAYERS] = { 0 };
char* names[MAX_PLAYERS] = { NULL };
```

```
int* current_players;

DWORD old_protect;

// Function to calculate the euclidean distance between two points
float euclidean_distance(float x, float y) {
	return sqrtf((x * x) + (y * y));
}

// Code cave responsible for disabling depth testing on models
__declspec(naked) void opengl_codecave() {
	__asm {
		pushad
	}

	(*glDepthFunc)(0x207);

	// Restore the original instruction and jump back
	__asm {
		popad
		mov esi, dword ptr ds : [esi + 0xA18]
		jmp opengl_ret_address
	}
}

// The injected thread responsible for creating our hooks for OpenGL
void opengl_thread() {
	while (true) {
		// Since OpenGL will be loaded dynamically into the process, our
thread needs to wait
		// until it sees that the OpenGL module has been loaded.
		if (openGLHandle == NULL) {
			openGLHandle = GetModuleHandle(L"opengl32.dll");
		}

		if (openGLHandle != NULL && glDepthFunc == NULL) {
			glDepthFunc = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glDepthFunc");

			// Then we find the location of glDrawElements and offset
to an instruction that is easy to hook
			opengl_hook_location = (unsigned
char*)GetProcAddress(openGLHandle, "glDrawElements");
			opengl_hook_location += 0x16;
```

```
                  // For the hook, we unprotect the memory at the code we
wish to write at
                  // Then set the first opcode to E9, or jump
                  // Caculate the location using the formula: new_location -
original_location+5
                  // And finally, since the first original instructions
totalled 6 bytes, NOP out the last remaining byte
                  VirtualProtect((void*)opengl_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
                  *opengl_hook_location = 0xE9;
                  *(DWORD*)(opengl_hook_location + 1) =
(DWORD)&opengl_codecave - ((DWORD)opengl_hook_location + 5);
                  *(opengl_hook_location + 5) = 0x90;

                  // Since OpenGL is loaded dynamically, we need to
dynamically calculate the return address
                  opengl_ret_address = (DWORD)(opengl_hook_location + 0x6);
            }
            else {
                  break;
            }

            // So our thread doesn't constantly run, we have it pause
execution for a millisecond.
            // This allows the processor to schedule other tasks.
            Sleep(1);
      }
}

// Our triggerbot code cave
__declspec(naked) void triggerbot_codecave() {
      // Asm blocks allow you to write pure assembly
      // In this case, we use it to call the function we hooked and save all
the registers
      // After we make the call, we move its return value in eax into a
variable
      __asm {
            call triggerbot_ori_call_address
            pushad
            mov edi_value, eax
      }

      // If the result of the call is not zero, then we are looking at a
player
```

```
	// Create a mouse event to simulate the left mouse button being pressed
down and send it to the game
	// Otherwise, raise the mouse button up so we stop firing
	if (edi_value != 0) {
		input.type = INPUT_MOUSE;
		input.mi.dwFlags = MOUSEEVENTF_LEFTDOWN;
		SendInput(1, &input, sizeof(INPUT));
	}
	else {
		input.type = INPUT_MOUSE;
		input.mi.dwFlags = MOUSEEVENTF_LEFTUP;
		SendInput(1, &input, sizeof(INPUT));
	}

	// Restore the registers and jump back to original code
	_asm {
		popad
		jmp triggerbot_ori_jump_address
	}
}

// Our code cave responsible for printing text
__declspec(naked) void esp_codecave() {
	current_players = (int*)(0x50F500);

	// First, recreate the original function we hooked but set the text to
empty
	__asm {
		mov ecx, empty_text
		call text_address
		pushad
	}

	// Next, loop through all the current players in the game
	for (int i = 1; i < *current_players; i++) {
		// Store the calculated screen positions in temporary variables
		x = x_values[i];
		y = y_values[i];
		text = names[i];

		// Make sure our text is on screen
		if (x > 2400 || x < 0 || y < 0 || y > 1800) {
			text = "";
		}
```

```
            // Invoke the print text function to display the text
            __asm {
                  mov ecx, text
                  push y
                  push x
                  call text_address
                  add esp, 8
            }
      }

      // Restore the registers and jump back to the original code
      __asm {
            popad
            jmp esp_ret_address
      }
}

// This thread contains all of our aimbot and ESP code
void aimbot_thread() {

      while (true) {
            // First, grab the current position and view angles of our player
            DWORD* player_offset = (DWORD*)(0x509B74);
            player = (Player*)(*player_offset);

            // Then, get the current number of players in the game
            int* current_players = (int*)(0x50F500);

            // These variables will be used to hold the closest enemy to us
            float closest_player = -1.0f;
            float closest_yaw = 0.0f;
            float closest_pitch = 0.0f;

            // Iterate through all active enemies
            for (int i = 0; i < *current_players; i++) {
                  DWORD* enemy_list = (DWORD*)(0x50F4F8);
                  DWORD* enemy_offset = (DWORD*)(*enemy_list + (i * 4));
                  Player* enemy = (Player*)(*enemy_offset);

                  // Make sure the enemy is valid and alive
                  if (player != NULL && enemy != NULL) {

                        // Calculate the absolute position of the enemy away
from us to ensure that our future calculations are correct and based
                        // around the origin
```

```
                        float abspos_x = enemy->x - player->x;
                        float abspos_y = enemy->y - player->y;
                        float abspos_z = enemy->z - player->z;

                        // Calculate our distance from the enemy
                        float temp_distance = euclidean_distance(abspos_x,
abspos_y);
                        // If this is the closest enemy so far, calculate the
yaw and pitch to aim at them

                        float azimuth_xy = atan2f(abspos_y, abspos_x);
                        float yaw = (float)(azimuth_xy * (180.0 / M_PI));
                        yaw += 90;

                        // Calculate the difference between our current yaw
and the calculated yaw to the enemy
                        float yaw_dif = player->yaw - yaw;

                        // If we are near the 275 angle boundary, our yaw_dif
will be too large, causing our text to appear incorrectly
                        // To compensate for that, subtract the yaw_dif from
360 if it is over 180, since our viewport can never show 180 degrees
                        if (yaw_dif > 180) {
                              yaw_dif = yaw_dif - 360;
                        }

                        if (yaw_dif < -180) {
                              yaw_dif = yaw_dif + 360;
                        }

                        // Calculate our X value by adding the yaw_dif times
a scaling factor to the center of the screen horizontally (1200)
                        x_values[i] = (DWORD)(1200 + (yaw_dif * -30));

                        // Calculate the pitch
                        // Since Z values are so limited, pick the larger
between x and y to ensure that we
                        // don't look straight at the air when close to an
enemy
                        if (abspos_y < 0) {
                              abspos_y *= -1;
                        }
                        if (abspos_y < 5) {
                              if (abspos_x < 0) {
                                    abspos_x *= -1;
```

```
                              }
                              abspos_y = abspos_x;
                        }
                        float azimuth_z = atan2f(abspos_z, abspos_y);
                        float pitch = (float)(azimuth_z * (180.0 / M_PI));
                        // Same as above but for pitch
                        float pitch_dif = player->pitch - pitch;

                        // Calculate our Y value by adding the pitch_dif
times a scaling factor to the center of the screen vertically (900)
                        y_values[i] = (DWORD)(900 + ((pitch_dif) * 25));

                        // Set the name to the enemy name
                        names[i] = enemy->name;

                        if ((closest_player == -1.0f || temp_distance <
closest_player) && !enemy->dead) {
                              closest_player = temp_distance;
                              closest_yaw = yaw;
                              closest_pitch = pitch;
                        }
                  }
            }

            // When our loop ends, set our yaw and pitch to the closst values
            player->yaw = closest_yaw;
            player->pitch = closest_pitch;

            // So our thread doesn't constantly run, we have it pause
execution for a millisecond.
            // This allows the processor to schedule other tasks.
            Sleep(1);
      }
}

// When our DLL is loaded, create a thread in the process to create the hook
// We need to do this as our DLL might be loaded before OpenGL is loaded by
the process
// Also create the aimbot and ESP thread and hook the locations for the
triggerbot and printing text
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      unsigned char* triggerbot_hook_location = (unsigned char*)0x0040AD9D;
      unsigned char* esp_hook_location = (unsigned char*)0x0040BE7E;
```

```
	if (fdwReason == DLL_PROCESS_ATTACH) {
		CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)opengl_thread,
NULL, 0, NULL);
		CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)aimbot_thread,
NULL, 0, NULL);

		VirtualProtect((void*)triggerbot_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
		*triggerbot_hook_location = 0xE9;
		*(DWORD*)(triggerbot_hook_location + 1) =
(DWORD)&triggerbot_codecave - ((DWORD)triggerbot_hook_location + 5);

		VirtualProtect((void*)esp_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
		*esp_hook_location = 0xE9;
		*(DWORD*)(esp_hook_location + 1) = (DWORD)&esp_codecave -
((DWORD)esp_hook_location + 5);
	}

	return true;
}
```

First Refactor

Our first refactor of the multihack code, in which we break out the triggerbot functionality to its own class.

Header/Triggerbot.h

```
#pragma once

#include <Windows.h>

class Triggerbot {
private:
	INPUT input = { 0 };
public:
	Triggerbot();
	void execute(int isLookingAtEnemy);
};
```

Source/Triggerbot.cpp

```
#include <Windows.h>

#include "Triggerbot.h"

Triggerbot::Triggerbot() {
	input = { 0 };
}

// If isLookingAtEnemy is not zero, then we are looking at a player
// Create a mouse event to simulate the left mouse button being pressed down
and send it to the game
// Otherwise, raise the mouse button up so we stop firing
void Triggerbot::execute(int isLookingAtEnemy) {
	if (isLookingAtEnemy != 0) {
		input.type = INPUT_MOUSE;
		input.mi.dwFlags = MOUSEEVENTF_LEFTDOWN;
		SendInput(1, &input, sizeof(INPUT));
	}
	else {
		input.type = INPUT_MOUSE;
		input.mi.dwFlags = MOUSEEVENTF_LEFTUP;
		SendInput(1, &input, sizeof(INPUT));
	}
}
```

Source/main.cpp

```
#include <Windows.h>
#include <math.h>

#include "Triggerbot.h"

#define M_PI 3.14159265358979323846
#define MAX_PLAYERS 32

// Our triggerbot class
Triggerbot *triggerbot;

HMODULE openGLHandle = NULL;
void(__stdcall* glDepthFunc)(unsigned int) = NULL;
unsigned char* opengl_hook_location;
DWORD opengl_ret_address = 0;

DWORD triggerbot_ori_call_address = 0x4607C0;
```

```
DWORD triggerbot_ori_jump_address = 0x0040ADA2;
DWORD edi_value = 0;

// The player structure for every player in the game
struct Player {
	char unknown1[4];
	float x;
	float y;
	float z;
	char unknown2[0x30];
	float yaw;
	float pitch;
	char unknown3[0x1DD];
	char name[16];
	char unknown4[0x103];
	int dead;
};

// Our player
Player* player = NULL;

DWORD esp_ret_address = 0x0040BE83;
DWORD text_address = 0x419880;

// Our temporary variables for our print text code cave
const char* text = "";
const char* empty_text = "";

DWORD x = 0;
DWORD y = 0;

// List of calculated ESP values
DWORD x_values[MAX_PLAYERS] = { 0 };
DWORD y_values[MAX_PLAYERS] = { 0 };
char* names[MAX_PLAYERS] = { NULL };

int* current_players;

DWORD old_protect;

// Function to calculate the euclidean distance between two points
float euclidean_distance(float x, float y) {
	return sqrtf((x * x) + (y * y));
}
```

```
// Our glDrawElements code cave responsible for our wallhack
__declspec(naked) void opengl_codecave() {
	__asm {
		pushad
	}

	(*glDepthFunc)(0x207);

	// Finally, restore the original instruction and jump back
	__asm {
		popad
		mov esi, dword ptr ds : [esi + 0xA18]
		jmp opengl_ret_address
	}
}

// The injected thread responsible for creating our OpenGL hooks
void opengl_thread() {
	while (true) {
		// Since OpenGL will be loaded dynamically into the process, our
thread needs to wait
		// until it sees that the OpenGL module has been loaded.
		if (openGLHandle == NULL) {
			openGLHandle = GetModuleHandle(L"opengl32.dll");
		}

		if (openGLHandle != NULL && glDepthFunc == NULL) {
			glDepthFunc = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glDepthFunc");

			// Then we find the location of glDrawElements and offset
to an instruction that is easy to hook
			opengl_hook_location = (unsigned
char*)GetProcAddress(openGLHandle, "glDrawElements");
			opengl_hook_location += 0x16;

			// For the hook, we unprotect the memory at the code we
wish to write at
			// Then set the first opcode to E9, or jump
			// Caculate the location using the formula: new_location -
original_location+5
			// And finally, since the first original instructions
totalled 6 bytes, NOP out the last remaining byte
			VirtualProtect((void*)opengl_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
```

```
                  *opengl_hook_location = 0xE9;
                  *(DWORD*)(opengl_hook_location + 1) =
(DWORD)&opengl_codecave - ((DWORD)opengl_hook_location + 5);
                  *(opengl_hook_location + 5) = 0x90;

                  // Since OpenGL is loaded dynamically, we need to
dynamically calculate the return address
                  opengl_ret_address = (DWORD)(opengl_hook_location + 0x6);
            }
            else {
                  break;
            }

            // So our thread doesn't constantly run, we have it pause
execution for a millisecond.
            // This allows the processor to schedule other tasks.
            Sleep(1);
      }
}

// Our triggerbot code cave
__declspec(naked) void triggerbot_codecave() {
      // Restore the original call and get the value of edi, which holds
whether we are looking at a player
      __asm {
            call triggerbot_ori_call_address
            pushad
            mov edi_value, eax
      }

      // Pass this information off to the triggerbot instance to determine
what to do
      triggerbot->execute(edi_value);

      // Restore the registers and jump back to original code
      _asm {
            popad
            jmp triggerbot_ori_jump_address
      }
}

// Our code cave responsible for printing text
__declspec(naked) void esp_codecave() {
      current_players = (int*)(0x50F500);
```

```
	// First, recreate the original function we hooked but set the text to
empty
	__asm {
		mov ecx, empty_text
		call text_address
		pushad
	}

	// Next, loop through all the current players in the game
	for (int i = 1; i < *current_players; i++) {
		// Store the calculated screen positions in temporary variables
		x = x_values[i];
		y = y_values[i];
		text = names[i];

		// Make sure our text is on screen
		if (x > 2400 || x < 0 || y < 0 || y > 1800) {
			text = "";
		}

		// Invoke the print text function to display the text
		__asm {
			mov ecx, text
			push y
			push x
			call text_address
			add esp, 8
		}
	}

	// Restore the registers and jump back to the original code
	__asm {
		popad
		jmp esp_ret_address
	}
}

// This thread contains all of our aimbot and ESP code
void aimbot_thread() {

	while (true) {
		// First, grab the current position and view angles of our player
		DWORD* player_offset = (DWORD*)(0x509B74);
		player = (Player*)(*player_offset);
```

```
            // Then, get the current number of players in the game
            int* current_players = (int*)(0x50F500);

            // These variables will be used to hold the closest enemy to us
            float closest_player = -1.0f;
            float closest_yaw = 0.0f;
            float closest_pitch = 0.0f;

            // Iterate through all active enemies
            for (int i = 0; i < *current_players; i++) {
                  DWORD* enemy_list = (DWORD*)(0x50F4F8);
                  DWORD* enemy_offset = (DWORD*)(*enemy_list + (i * 4));
                  Player* enemy = (Player*)(*enemy_offset);

                  // Make sure the enemy is valid and alive
                  if (player != NULL && enemy != NULL) {

                        // Calculate the absolute position of the enemy away
from us to ensure that our future calculations are correct and based
                        // around the origin
                        float abspos_x = enemy->x - player->x;
                        float abspos_y = enemy->y - player->y;
                        float abspos_z = enemy->z - player->z;

                        // Calculate our distance from the enemy
                        float temp_distance = euclidean_distance(abspos_x,
abspos_y);
                        // If this is the closest enemy so far, calculate the
yaw and pitch to aim at them

                        float azimuth_xy = atan2f(abspos_y, abspos_x);
                        float yaw = (float)(azimuth_xy * (180.0 / M_PI));
                        yaw += 90;

                        // Calculate the difference between our current yaw
and the calculated yaw to the enemy
                        float yaw_dif = player->yaw - yaw;

                        // If we are near the 275 angle boundary, our yaw_dif
will be too large, causing our text to appear incorrectly
                        // To compensate for that, subtract the yaw_dif from
360 if it is over 180, since our viewport can never show 180 degrees
                        if (yaw_dif > 180) {
                              yaw_dif = yaw_dif - 360;
                        }
```

```
                        if (yaw_dif < -180) {
                              yaw_dif = yaw_dif + 360;
                        }

                        // Calculate our X value by adding the yaw_dif times
a scaling factor to the center of the screen horizontally (1200)
                        x_values[i] = (DWORD)(1200 + (yaw_dif * -30));

                        // Calculate the pitch
                        // Since Z values are so limited, pick the larger
between x and y to ensure that we
                        // don't look straight at the air when close to an
enemy
                        if (abspos_y < 0) {
                              abspos_y *= -1;
                        }
                        if (abspos_y < 5) {
                              if (abspos_x < 0) {
                                    abspos_x *= -1;
                              }
                              abspos_y = abspos_x;
                        }
                        float azimuth_z = atan2f(abspos_z, abspos_y);
                        float pitch = (float)(azimuth_z * (180.0 / M_PI));
                        // Same as above but for pitch
                        float pitch_dif = player->pitch - pitch;

                        // Calculate our Y value by adding the pitch_dif
times a scaling factor to the center of the screen vertically (900)
                        y_values[i] = (DWORD)(900 + ((pitch_dif) * 25));

                        // Set the name to the enemy name
                        names[i] = enemy->name;

                        if ((closest_player == -1.0f || temp_distance <
closest_player) && !enemy->dead) {
                              closest_player = temp_distance;
                              closest_yaw = yaw;
                              closest_pitch = pitch;
                        }
                  }
            }

            // When our loop ends, set our yaw and pitch to the closst values
```

```
            player->yaw = closest_yaw;
            player->pitch = closest_pitch;

            // So our thread doesn't constantly run, we have it pause
execution for a millisecond.
            // This allows the processor to schedule other tasks.
            Sleep(1);
      }
}

// When our DLL is loaded, create a thread in the process to create the hook
// We need to do this as our DLL might be loaded before OpenGL is loaded by
the process
// Also create the aimbot and ESP thread and hook the locations for the
triggerbot and printing text
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      unsigned char* triggerbot_hook_location = (unsigned char*)0x0040AD9D;
      unsigned char* esp_hook_location = (unsigned char*)0x0040BE7E;

      if (fdwReason == DLL_PROCESS_ATTACH) {
            // Create our triggerbot
            triggerbot = new Triggerbot();

            CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)opengl_thread,
NULL, 0, NULL);
            CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)aimbot_thread,
NULL, 0, NULL);

            VirtualProtect((void*)triggerbot_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
            *triggerbot_hook_location = 0xE9;
            *(DWORD*)(triggerbot_hook_location + 1) =
(DWORD)&triggerbot_codecave - ((DWORD)triggerbot_hook_location + 5);

            VirtualProtect((void*)esp_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
            *esp_hook_location = 0xE9;
            *(DWORD*)(esp_hook_location + 1) = (DWORD)&esp_codecave -
((DWORD)esp_hook_location + 5);
      }
      else if (fdwReason == DLL_PROCESS_DETACH) {
            delete triggerbot;
      }
```

```
	return true;
}
```

Refactor Finished

The complete code for the multihack after the refactor is finished.

Header/Triggerbot.h - Unchanged from First Refactor

Header/constants.h

```
#pragma once

#include <Windows.h>

DWORD triggerbot_ori_call_address = 0x4607C0;
DWORD triggerbot_ori_jump_address = 0x0040ADA2;

DWORD esp_ret_address = 0x0040BE83;
DWORD text_address = 0x419880;

const char* empty_text = “";
```

Header/PlayerGeometry.h

```
#pragma once

#include <Windows.h>

#define M_PI 3.14159265358979323846
#define MAX_PLAYERS 32

// The player structure for every player in the game
struct Player {
	char unknown1[4];
	float x;
	float y;
	float z;
	char unknown2[0x30];
	float yaw;
	float pitch;
	char unknown3[0x1DD];
```


Playrix Ex. 1024, page 465

```
	char name[16];
	char unknown4[0x103];
	int dead;
};

class PlayerGeometry {
private:
	DWORD player_offset_address;
	DWORD enemy_list_address;
	DWORD current_players_address;

	float closest_yaw;
	float closest_pitch;

	Player* player;

	float euclidean_distance(float, float);
public:
	DWORD x_values[MAX_PLAYERS] = { 0 };
	DWORD y_values[MAX_PLAYERS] = { 0 };
	char* names[MAX_PLAYERS] = { NULL };

	int* current_players;

	PlayerGeometry(DWORD, DWORD, DWORD);
	void update();
	void set_player_view();
};
```

Source/Triggerbot.cpp - Unchanged from First Refactor

Source/PlayerGeometry.cpp

```
#include <Windows.h>
#include <math.h>

#include "PlayerGeometry.h"

PlayerGeometry::PlayerGeometry(DWORD p_address, DWORD e_address, DWORD
cp_address) {
	player_offset_address = p_address;
	enemy_list_address = e_address;
	current_players_address = cp_address;
}
```

```
// Function to calculate the euclidean distance between two points
float PlayerGeometry::euclidean_distance(float x, float y) {
	return sqrtf((x * x) + (y * y));
}

void PlayerGeometry::update() {
	// First, grab the current position and view angles of our player
	DWORD* player_offset = (DWORD*)(player_offset_address);
	player = (Player*)(*player_offset);

	// Then, get the current number of players in the game
	current_players = (int*)(0x50F500);

	float closest_player = -1.0f;
	closest_yaw = 0.0f;
	closest_pitch = 0.0f;

	// Iterate through all active enemies
	for (int i = 0; i < *current_players; i++) {
		DWORD* enemy_list = (DWORD*)(0x50F4F8);
		DWORD* enemy_offset = (DWORD*)(*enemy_list + (i * 4));
		Player* enemy = (Player*)(*enemy_offset);

		// Make sure the enemy is valid and alive
		if (player != NULL && enemy != NULL) {
			// Calculate the absolute position of the enemy away from
us to ensure that our future calculations are correct and based
			// around the origin
			float abspos_x = enemy->x - player->x;
			float abspos_y = enemy->y - player->y;
			float abspos_z = enemy->z - player->z;

			// Calculate our distance from the enemy
			float temp_distance = euclidean_distance(abspos_x,
abspos_y);
			// If this is the closest enemy so far, calculate the yaw
and pitch to aim at them

			float azimuth_xy = atan2f(abspos_y, abspos_x);
			float yaw = (float)(azimuth_xy * (180.0 / M_PI));
			yaw += 90;

			// Calculate the difference between our current yaw and the
calculated yaw to the enemy
```

```
                float yaw_dif = player->yaw - yaw;

                // If we are near the 275 angle boundary, our yaw_dif will
be too large, causing our text to appear incorrectly
                // To compensate for that, subtract the yaw_dif from 360 if
it is over 180, since our viewport can never show 180 degrees
                if (yaw_dif > 180) {
                      yaw_dif = yaw_dif - 360;
                }

                if (yaw_dif < -180) {
                      yaw_dif = yaw_dif + 360;
                }

                // Calculate our X value by adding the yaw_dif times a
scaling factor to the center of the screen horizontally (1200)
                x_values[i] = (DWORD)(1200 + (yaw_dif * -30));

                // Calculate the pitch
                // Since Z values are so limited, pick the larger between x
and y to ensure that we
                // don't look straight at the air when close to an enemy
                if (abspos_y < 0) {
                      abspos_y *= -1;
                }
                if (abspos_y < 5) {
                      if (abspos_x < 0) {
                            abspos_x *= -1;
                      }
                      abspos_y = abspos_x;
                }
                float azimuth_z = atan2f(abspos_z, abspos_y);
                float pitch = (float)(azimuth_z * (180.0 / M_PI));
                // Same as above but for pitch
                float pitch_dif = player->pitch - pitch;

                // Calculate our Y value by adding the pitch_dif times a
scaling factor to the center of the screen vertically (900)
                y_values[i] = (DWORD)(900 + ((pitch_dif) * 25));

                // Set the name to the enemy name
                names[i] = enemy->name;

                if ((closest_player == -1.0f || temp_distance <
closest_player) && !enemy->dead) {
```

```
                        closest_player = temp_distance;
                        closest_yaw = yaw;
                        closest_pitch = pitch;
                  }
            }
      }
}

void PlayerGeometry::set_player_view() {
      player->yaw = closest_yaw;
      player->pitch = closest_pitch;
}
```

Source/main.cpp

```
#include <Windows.h>

#include "constants.h"
#include "Triggerbot.h"
#include "PlayerGeometry.h"

Triggerbot *triggerbot;
PlayerGeometry *playerGeometry;

HMODULE openGLHandle = NULL;
void(__stdcall* glDepthFunc)(unsigned int) = NULL;
DWORD opengl_ret_address = 0;

DWORD edi_value = 0;

// Our temporary variables for our print text code cave
const char* text = "";

DWORD x;
DWORD y;

DWORD old_protect;

// Code cave responsible for disabling depth testing on models
__declspec(naked) void opengl_codecave() {
      __asm {
            pushad
      }
```

```
	(*glDepthFunc)(0x207);

	// Finally, restore the original instruction and jump back
	__asm {
		popad
		mov esi, dword ptr ds : [esi + 0xA18]
		jmp opengl_ret_address
	}
}

// Code cave responsible for our triggerbot
__declspec(naked) void triggerbot_codecave() {
	// Asm blocks allow you to write pure assembly
	// In this case, we use it to call the function we hooked and save all
the registers
	// After we make the call, we move its return value in eax into a
variable
	__asm {
		call triggerbot_ori_call_address
		pushad
		mov edi_value, eax
	}

	triggerbot->execute(edi_value);

	// Restore the registers and jump back to original code
	_asm {
		popad
		jmp triggerbot_ori_jump_address
	}
}

// Our code cave responsible for printing text
__declspec(naked) void text_codecave() {
	// First, recreate the original function we hooked but set the text to
empty
	__asm {
		mov ecx, empty_text
		call text_address
		pushad
	}

	// Next, loop through all the current players in the game
	for (int i = 1; i < *playerGeometry->current_players; i++) {
		// Store the calculated screen positions in temporary variables
```

```
            x = playerGeometry->x_values[i];
            y = playerGeometry->y_values[i];
            text = playerGeometry->names[i];

            // Make sure our text is on screen
            if (x > 2400 || x < 0 || y < 0 || y > 1800) {
                  text = "";
            }

            // Invoke the print text function to display the text
            __asm {
                  mov ecx, text
                  push y
                  push x
                  call text_address
                  add esp, 8
            }
      }

      // Restore the registers and jump back to the original code
      __asm {
            popad
            jmp esp_ret_address
      }
}

// This thread contains all of our aimbot, ESP, and OpenGL hooking code
void injected_thread() {

      while (true) {
            if (openGLHandle == NULL) {
                  openGLHandle = GetModuleHandle(L"opengl32.dll");
            }

            if (openGLHandle != NULL && glDepthFunc == NULL) {
                  glDepthFunc = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glDepthFunc");

                  // Then we find the location of glDrawElements and offset
to an instruction that is easy to hook
                  unsigned char *opengl_hook_location = (unsigned
char*)GetProcAddress(openGLHandle, "glDrawElements");
                  opengl_hook_location += 0x16;
```

```
                  // For the hook, we unprotect the memory at the code we
wish to write at
                  // Then set the first opcode to E9, or jump
                  // Caculate the location using the formula: new_location -
original_location+5
                  // And finally, since the first original instructions
totalled 6 bytes, NOP out the last remaining byte
                  VirtualProtect((void*)opengl_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
                  *opengl_hook_location = 0xE9;
                  *(DWORD*)(opengl_hook_location + 1) =
(DWORD)&opengl_codecave - ((DWORD)opengl_hook_location + 5);
                  *(opengl_hook_location + 5) = 0x90;

                  // Since OpenGL is loaded dynamically, we need to
dynamically calculate the return address
                  opengl_ret_address = (DWORD)(opengl_hook_location + 0x6);
            }

            playerGeometry->update();
            playerGeometry->set_player_view();

            // So our thread doesn't constantly run, we have it pause
execution for a millisecond.
            // This allows the processor to schedule other tasks.
            Sleep(1);
      }
}

// When our DLL is loaded, create a thread in the process to create the hook
// We need to do this as our DLL might be loaded before OpenGL is loaded by
the process
// Also create the aimbot and ESP thread and hook the locations for the
triggerbot and printing text
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      unsigned char* triggerbot_hook_location = (unsigned char*)0x0040AD9D;
      unsigned char* text_hook_location = (unsigned char*)0x0040BE7E;

      if (fdwReason == DLL_PROCESS_ATTACH) {
            triggerbot = new Triggerbot();
            playerGeometry = new PlayerGeometry(0x509B74, 0x50F4F8,
0x50F500);
```

```
            CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread,
NULL, 0, NULL);

            VirtualProtect((void*)triggerbot_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
            *triggerbot_hook_location = 0xE9;
            *(DWORD*)(triggerbot_hook_location + 1) =
(DWORD)&triggerbot_codecave - ((DWORD)triggerbot_hook_location + 5);

            VirtualProtect((void*)text_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
            *text_hook_location = 0xE9;
            *(DWORD*)(text_hook_location + 1) = (DWORD)&text_codecave -
((DWORD)text_hook_location + 5);
      }
      else if (fdwReason == DLL_PROCESS_DETACH) {
            delete triggerbot;
            delete playerGeometry;
      }

      return true;
}
```

Finished

A multihack for Assault Cube 1.2.0.2 that contains the following features:

- Wallhack
- ESP
- Aimbot
- Triggerbot

It also has an interactive menu that allows these features to be toggled on and off. Use the up and down arrows to change the selection, and the left and right arrows to toggle the features.

This must be injected into the Assault Cube process to work. One way to do this is to use a DLL injector. Another way is to enable AppInit_DLLs in the registry.

Header/PlayerGeometry.h - Unchanged from “Refactor Finished”

Header/Triggerbot.h - Unchanged from “Refactor Finished”

Header/constants.h - Unchanged from “Refactor Finished”

Header/Menu.h

```
#pragma once

#define WALLHACK 0
#define ESP 1
#define AIMBOT 2
#define TRIGGERBOT 3

#define MAX_ITEMS 4

class Menu {
private:
	const char on_text[5] = { 0xc, 0x30, 'O', 'N', 0 };
	const char off_text[6] = { 0xc, 0x33, 'O', 'F', 'F', 0 };
public:
	int cursor_position;

	const char* items[MAX_ITEMS] = { "Wallhack", "ESP", "Aimbot",
"Triggerbot" };
	bool item_enabled[MAX_ITEMS] = { false };

	const char* cursor = ">";

	Menu();
	void handle_input();
	const char* get_state(int);
};
```

Source/Triggerbot.cpp - Unchanged from “Refactor Finished”

Source/PlayerGeometry.cpp - Unchanged from “Refactor Finished”

Source/Menu.cpp

```
#include <Windows.h>

#include "Menu.h"

Menu::Menu() {
	cursor_position = 0;
}
```

```
void Menu::handle_input() {

	if (GetAsyncKeyState(VK_DOWN) & 1) {
		cursor_position++;
	}
	else if (GetAsyncKeyState(VK_UP) & 1) {
		cursor_position--;
	}
	else if ((GetAsyncKeyState(VK_LEFT) & 1) || (GetAsyncKeyState(VK_RIGHT)
& 1)) {
		item_enabled[cursor_position] = !item_enabled[cursor_position];
	}

	if (cursor_position < 0) {
		cursor_position = 3;
	}
	else if (cursor_position > 3) {
		cursor_position = 0;
	}
}

const char* Menu::get_state(int item) {
	return item_enabled[item] ? on_text : off_text;
}
```

Source/main.cpp

```
#include <Windows.h>

#include "constants.h"
#include "Triggerbot.h"
#include "PlayerGeometry.h"
#include "Menu.h"

Triggerbot *triggerbot;
PlayerGeometry *playerGeometry;
Menu *menu;

HMODULE openGLHandle = NULL;
void(__stdcall* glDepthFunc)(unsigned int) = NULL;
DWORD opengl_ret_address = 0;

DWORD edi_value = 0;
```

```
DWORD old_protect;

// Code cave responsible for disabling depth testing on models
__declspec(naked) void opengl_codecave() {
	__asm {
		pushad
	}

	if (menu->item_enabled[WALLHACK]) {
		(*glDepthFunc)(0x207);
	}

	// Finally, restore the original instruction and jump back
	__asm {
		popad
		mov esi, dword ptr ds : [esi + 0xA18]
		jmp opengl_ret_address
	}
}

// Our triggerbot code cave
__declspec(naked) void triggerbot_codecave() {
	// Asm blocks allow you to write pure assembly
	// In this case, we use it to call the function we hooked and save all
the registers
	// After we make the call, we move its return value in eax into a
variable
	__asm {
		call triggerbot_ori_call_address
		pushad
		mov edi_value, eax
	}

	if (menu->item_enabled[TRIGGERBOT]) {
		triggerbot->execute(edi_value);
	}

	// Restore the registers and jump back to original code
	_asm {
		popad
		jmp triggerbot_ori_jump_address
	}
}
```

```
// A helper function for printing text
void print_text(DWORD x, DWORD y, const char* text) {
	if (x > 2400 || x < 0 || y < 0 || y > 1800) {
		text = "";
	}

	x += 200;

	__asm {
		mov ecx, text
		push y
		push x
		call text_address
		add esp, 8
	}
}

// Our code cave responsible for printing text
__declspec(naked) void text_codecave() {
	// First, recreate the original function we hooked but set the text to
empty
	__asm {
		mov ecx, empty_text
		call text_address
		pushad
	}

	for (int i = 0; i < MAX_ITEMS; i++) {
		print_text(50, 250 + (100 * i), menu->items[i]);
		print_text(500, 250 + (100 * i), menu->get_state(i));
	}
	print_text(10, 250 + (100 * menu->cursor_position), menu->cursor);

	if (menu->item_enabled[ESP]) {
		// Next, loop through all the current players in the game
		for (int i = 1; i < *playerGeometry->current_players; i++) {
			print_text(playerGeometry->x_values[i], playerGeometry-
>y_values[i], playerGeometry->names[i]);
		}
	}

	// Restore the registers and jump back to the original code
	__asm {
		popad
		jmp esp_ret_address
```

```
	}
}

// This thread contains all of our aimbot, ESP, and OpenGL hooking code
void injected_thread() {

	while (true) {
		if (openGLHandle == NULL) {
			openGLHandle = GetModuleHandle(L"opengl32.dll");
		}

		if (openGLHandle != NULL && glDepthFunc == NULL) {
			glDepthFunc = (void(__stdcall*)(unsigned
int))GetProcAddress(openGLHandle, "glDepthFunc");

			// Then we find the location of glDrawElements and offset
to an instruction that is easy to hook
			unsigned char *opengl_hook_location = (unsigned
char*)GetProcAddress(openGLHandle, "glDrawElements");
			opengl_hook_location += 0x16;

			// For the hook, we unprotect the memory at the code we
wish to write at
			// Then set the first opcode to E9, or jump
			// Caculate the location using the formula: new_location -
original_location+5
			// And finally, since the first original instructions
totalled 6 bytes, NOP out the last remaining byte
			VirtualProtect((void*)opengl_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
			*opengl_hook_location = 0xE9;
			*(DWORD*)(opengl_hook_location + 1) =
(DWORD)&opengl_codecave - ((DWORD)opengl_hook_location + 5);
			*(opengl_hook_location + 5) = 0x90;

			// Since OpenGL is loaded dynamically, we need to
dynamically calculate the return address
			opengl_ret_address = (DWORD)(opengl_hook_location + 0x6);
		}

		menu->handle_input();

		playerGeometry->update();

		if (menu->item_enabled[AIMBOT]) {
```

```
                  playerGeometry->set_player_view();
            }

            // So our thread doesn't constantly run, we have it pause
execution for a millisecond.
            // This allows the processor to schedule other tasks.
            Sleep(1);
      }
}

// When our DLL is loaded, create a thread in the process to create the hook
// We need to do this as our DLL might be loaded before OpenGL is loaded by
the process
// Also create the aimbot and ESP thread and hook the locations for the
triggerbot and printing text
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
      unsigned char* triggerbot_hook_location = (unsigned char*)0x0040AD9D;
      unsigned char* text_hook_location = (unsigned char*)0x0040BE7E;

      if (fdwReason == DLL_PROCESS_ATTACH) {
            triggerbot = new Triggerbot();
            playerGeometry = new PlayerGeometry(0x509B74, 0x50F4F8,
0x50F500);
            menu = new Menu();

            CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)injected_thread,
NULL, 0, NULL);

            VirtualProtect((void*)triggerbot_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
            *triggerbot_hook_location = 0xE9;
            *(DWORD*)(triggerbot_hook_location + 1) =
(DWORD)&triggerbot_codecave - ((DWORD)triggerbot_hook_location + 5);

            VirtualProtect((void*)text_hook_location, 5,
PAGE_EXECUTE_READWRITE, &old_protect);
            *text_hook_location = 0xE9;
            *(DWORD*)(text_hook_location + 1) = (DWORD)&text_codecave -
((DWORD)text_hook_location + 5);
      }
      else if (fdwReason == DLL_PROCESS_DETACH) {
            delete triggerbot;
            delete playerGeometry;
            delete menu;
```

```
	}

	return true;
}
```

A.16 Wesnoth Multiplayer Bot

Referenced in Chapter 6.2.

An example client that will connect to a local Wesnoth 1.14.9 server with the username *FFFAAAKKKEEE*.

The majority of the code is based on the Winsock example provided by Microsoft: https://docs.microsoft.com/en-us/windows/win32/winsock/complete-client-code

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib, "Ws2_32.lib")

#define DEFAULT_BUFLEN 512

int main(int argc, char** argv) {
	WSADATA wsaData;
	SOCKET ConnectSocket = INVALID_SOCKET;
	struct addrinfo* result = NULL,
		* ptr = NULL,
		hints;
	char recvbuf[DEFAULT_BUFLEN];
	int iResult;
	int recvbuflen = DEFAULT_BUFLEN;

	// The handshake initiation request
	const unsigned char buff_handshake_p1[] = {
		0x00, 0x00, 0x00, 0x00
```


Playrix Ex. 1024, page 480

```
	};

	// Contains the version 1.14.9
	const unsigned char buff_handshake_p2[] = {
		0x00, 0x00, 0x00, 0x2f, 0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00,
		0x00, 0x00, 0x00, 0xff, 0x8b, 0x2e, 0x4b, 0x2d, 0x2a, 0xce,
		0xcc, 0xcf, 0x8b, 0xe5, 0xe2, 0x84, 0xb2, 0x6c, 0x95, 0x0c,
		0xf5, 0x0c, 0x4d, 0xf4, 0x2c, 0x95, 0xb8, 0xa2, 0xf5, 0xe1,
		0x92, 0x5c, 0x00, 0xc0, 0x38, 0xd3, 0xd7, 0x28, 0x00, 0x00,
		0x00
	};

	// Contains the username FFFAAAKKKEEE
	const unsigned char buff_send_name[] = {
		0x00, 0x00, 0x00, 0x3a, 0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00,
		0x00, 0x00, 0x00, 0xff, 0x8b, 0xce, 0xc9, 0x4f, 0xcf, 0xcc,
		0x8b, 0xe5, 0xe2, 0x2c, 0x2d, 0x4e, 0x2d, 0xca, 0x4b, 0xcc,
		0x4d, 0xb5, 0x55, 0x72, 0x73, 0x73, 0x73, 0x74, 0x74, 0xf4,
		0xf6, 0xf6, 0x76, 0x75, 0x75, 0x55, 0xe2, 0x8a, 0xd6, 0x87,
		0xaa, 0xe0, 0x02, 0x00, 0xa1, 0xfc, 0x19, 0x4c, 0x2b, 0x00,
		0x00, 0x00
	};

	iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
	if (iResult != 0) {
		printf("WSAStartup failed: %d\n", iResult);
		return 1;
	}

	ZeroMemory(&hints, sizeof(hints));
	hints.ai_family = AF_INET;
	hints.ai_socktype = SOCK_STREAM;
	hints.ai_protocol = IPPROTO_TCP;

	iResult = getaddrinfo("127.0.0.1", "15000", &hints, &result);
	if (iResult != 0) {
		printf("getaddrinfo failed: %d\n", iResult);
		WSACleanup();
		return 1;
	}

	ptr = result;

	ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr-
>ai_protocol);
```

```
	if (ConnectSocket == INVALID_SOCKET) {
		printf("Error at socket(): %ld\n", WSAGetLastError());
		freeaddrinfo(result);
		WSACleanup();
		return 1;
	}

	iResult = connect(ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
	if (iResult == SOCKET_ERROR) {
		closesocket(ConnectSocket);
		ConnectSocket = INVALID_SOCKET;
	}

	freeaddrinfo(result);

	if (ConnectSocket == INVALID_SOCKET) {
		printf("Unable to connect to server!\n");
		WSACleanup();
		return 1;
	}

	iResult = send(ConnectSocket, (const char*)buff_handshake_p1,
(int)sizeof(buff_handshake_p1), 0);
	printf("Bytes Sent: %ld\n", iResult);

	iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
	printf("Bytes received: %d\n", iResult);

	iResult = send(ConnectSocket, (const char*)buff_handshake_p2,
(int)sizeof(buff_handshake_p2), 0);
	printf("Bytes Sent: %ld\n", iResult);

	iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
	printf("Bytes received: %d\n", iResult);

	iResult = send(ConnectSocket, (const char*)buff_send_name,
(int)sizeof(buff_send_name), 0);
	printf("Bytes Sent: %ld\n", iResult);

	do {
		iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
		if (iResult > 0)
			printf("Bytes received: %d\n", iResult);
		else if (iResult == 0)
```

```
                printf("Connection closed\n");
            else
                printf("recv failed with error: %d\n", WSAGetLastError());

        } while (iResult > 0);

        closesocket(ConnectSocket);
        WSACleanup();

        return 0;
}
```

A.17 Wesnoth ChatBot

Referenced in Chapter 6.4.

An example chatbot that will connect to a local Wesnoth 1.14.9 server with the username **ChatBot** and respond to the **\wave** command.

The majority of the code is based on the Winsock example provided by Microsoft: https://docs.microsoft.com/en-us/windows/win32/winsock/complete-client-code

```
#include <stdio.h>
#include <winsock2.h>
#include <ws2tcpip.h>

#pragma comment(lib, "Ws2_32.lib")

#include <zlib.h>

#define DEFAULT_BUFLEN 512

void send_data(const unsigned char *data, size_t len, SOCKET s) {
    gzFile temp_data = gzopen("packet.gz", "wb");
    gzwrite(temp_data, data, len);
    gzclose(temp_data);
```


Playrix Ex. 1024, page 483

```
    FILE* temp_file = NULL;
    fopen_s(&temp_file, "packet.gz", "rb");

    if (temp_file) {
        size_t compress_len = 0;
        unsigned char buffer[DEFAULT_BUFLEN] = { 0 };
        compress_len = fread(buffer, 1, sizeof(buffer), temp_file);
        fclose(temp_file);

        unsigned char buff_packet[DEFAULT_BUFLEN] = { 0 };
        memcpy(buff_packet + 3, &compress_len, sizeof(compress_len));
        memcpy(buff_packet + 4, buffer, compress_len);

        int iResult = send(s, (const char*)buff_packet, compress_len + 4, 0);
        printf("Bytes Sent: %ld\n", iResult);
    }
}

bool parse_data(unsigned char *buff, int buff_len) {
    unsigned char data[DEFAULT_BUFLEN] = { 0 };
    memcpy(data, buff + 4, buff_len - 4);

    FILE* temp_file = NULL;
    fopen_s(&temp_file, "packet_recv.gz", "wb");

    if (temp_file) {
        fwrite(data, 1, sizeof(data), temp_file);
        fclose(temp_file);
    }

    gzFile temp_data_in = gzopen("packet_recv.gz", "rb");
    unsigned char decompressed_data[DEFAULT_BUFLEN] = { 0 };
    gzread(temp_data_in, decompressed_data, DEFAULT_BUFLEN);
    fwrite(decompressed_data, 1, DEFAULT_BUFLEN, stdout);
    gzclose(temp_data_in);

    return strstr((const char*)decompressed_data, (const char*)"\\wave");
}

int main(int argc, char** argv) {
    WSADATA wsaData;
    SOCKET ConnectSocket = INVALID_SOCKET;
    struct addrinfo* result = NULL,
        * ptr = NULL,
```

```
        hints;
    unsigned char recvbuf[DEFAULT_BUFLEN];
    int iResult;
    int recvbuflen = DEFAULT_BUFLEN;

    // The handshake initiation request
    const unsigned char buff_handshake_p1[] = {
        0x00, 0x00, 0x00, 0x00
    };

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    iResult = getaddrinfo("127.0.0.1", "15000", &hints, &result);
    if (iResult != 0) {
        printf("getaddrinfo failed: %d\n", iResult);
        WSACleanup();
        return 1;
    }

    ptr = result;

    ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr-
>ai_protocol);

    if (ConnectSocket == INVALID_SOCKET) {
        printf("Error at socket(): %ld\n", WSAGetLastError());
        freeaddrinfo(result);
        WSACleanup();
        return 1;
    }

    iResult = connect(ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        closesocket(ConnectSocket);
        ConnectSocket = INVALID_SOCKET;
    }
```

```
    freeaddrinfo(result);

    if (ConnectSocket == INVALID_SOCKET) {
        printf("Unable to connect to server!\n");
        WSACleanup();
        return 1;
    }

    iResult = send(ConnectSocket, (const char*)buff_handshake_p1,
(int)sizeof(buff_handshake_p1), 0);
    printf("Bytes Sent: %ld\n", iResult);

    iResult = recv(ConnectSocket, (char*)recvbuf, recvbuflen, 0);
    printf("Bytes received: %d\n", iResult);

    const unsigned char version[] = "[version]\nversion=\"1.14.9\"\n[/
version]";
    send_data(version, sizeof(version), ConnectSocket);

    iResult = recv(ConnectSocket, (char*)recvbuf, recvbuflen, 0);
    printf("Bytes received: %d\n", iResult);

    const unsigned char name[] = "[login]\nusername=\"ChatBot\"\n[/login]";
    send_data(name, sizeof(name), ConnectSocket);

    const unsigned char first_message[] = "[message]\nmessage=\"ChatBot
connected\"\nroom=\"lobby\"\nsender=\"ChatBot\"\n[/message]";
    send_data(first_message, sizeof(first_message), ConnectSocket);

    do {
        iResult = recv(ConnectSocket, (char*)recvbuf, recvbuflen, 0);
        if (iResult > 0)
            printf("Bytes received: %d\n", iResult);
        else if (iResult == 0)
            printf("Connection closed\n");
        else
            printf("recv failed with error: %d\n", WSAGetLastError());

        if (parse_data(recvbuf, iResult)) {
            const unsigned char message[] = "[message]\nmessage=\"Hello!
\"\nroom=\"lobby\"\nsender=\"ChatBot\"\n[/message]";
            send_data(message, sizeof(message), ConnectSocket);
        }
    } while (iResult > 0);
```

```
    closesocket(ConnectSocket);
    WSACleanup();

    return 0;
}
```

A.18 Wesnoth Proxy

Referenced in Chapter 6.5.

An example proxy for Wesnoth 1.14.9 that allows interception and modification of traffic from a Wesnoth game client to a Wesnoth server. In this case, any time the proxy sees the chat message **\wave**, it will send an additional chat message saying **Hello!**.

The majority of the code is based on the Winsock example provided by Microsoft: https://docs.microsoft.com/en-us/windows/win32/winsock/complete-client-code and https://docs.microsoft.com/en-us/windows/win32/winsock/complete-server-code

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment (lib, "Ws2_32.lib")

#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "27015"

#include <zlib.h>

void send_data(const unsigned char* data, size_t len, SOCKET s) {
    gzFile temp_data = gzopen("packet.gz", "wb");
    gzwrite(temp_data, data, len);
    gzclose(temp_data);

    FILE* temp_file = NULL;
    fopen_s(&temp_file, "packet.gz", "rb");

    if (temp_file) {
        size_t compress_len = 0;
        unsigned char buffer[DEFAULT_BUFLEN] = { 0 };
```


Playrix Ex. 1024, page 487

```
        compress_len = fread(buffer, 1, sizeof(buffer), temp_file);
        fclose(temp_file);

        unsigned char buff_packet[DEFAULT_BUFLEN] = { 0 };
        memcpy(buff_packet + 3, &compress_len, sizeof(compress_len));
        memcpy(buff_packet + 4, buffer, compress_len);

        int iResult = send(s, (const char*)buff_packet, compress_len + 4, 0);
        printf("Bytes Sent: %ld\n", iResult);
    }
}

bool parse_data(unsigned char* buff, int buff_len) {
    unsigned char data[DEFAULT_BUFLEN] = { 0 };

    memcpy(data, buff + 4, buff_len - 4);

    FILE* temp_file = NULL;
    fopen_s(&temp_file, "packet_recv.gz", "wb");

    if (temp_file) {
        fwrite(data, 1, sizeof(data), temp_file);
        fclose(temp_file);
    }

    gzFile temp_data_in = gzopen("packet_recv.gz", "rb");
    unsigned char decompressed_data[DEFAULT_BUFLEN] = { 0 };
    gzread(temp_data_in, decompressed_data, DEFAULT_BUFLEN);
    fwrite(decompressed_data, 1, DEFAULT_BUFLEN, stdout);
    gzclose(temp_data_in);

    return strstr((const char*)decompressed_data, (const char*)"\\wave");
}

int main(void)
{
    WSADATA wsaData;
    int iResult;

    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET ClientSocket = INVALID_SOCKET;
    SOCKET ServerSocket = INVALID_SOCKET;

    struct addrinfo* result = NULL,
        hints;
```

```
    int iSendResult;
    unsigned char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;

    DWORD timeout = 1000;

    // Client Socket
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;

    iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
    ListenSocket = socket(result->ai_family, result->ai_socktype, result-
>ai_protocol);
    iResult = bind(ListenSocket, result->ai_addr, (int)result->ai_addrlen);
    freeaddrinfo(result);

    iResult = listen(ListenSocket, SOMAXCONN);
    ClientSocket = accept(ListenSocket, NULL, NULL);
    closesocket(ListenSocket);

    // Server Socket
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    iResult = getaddrinfo("127.0.0.1", "15000", &hints, &result);
    ServerSocket = socket(result->ai_family, result->ai_socktype, result-
>ai_protocol);
    iResult = connect(ServerSocket, result->ai_addr, (int)result-
>ai_addrlen);
    freeaddrinfo(result);

    setsockopt(ServerSocket, SOL_SOCKET, SO_RCVTIMEO, (char*)&timeout,
sizeof(timeout));

    do {
        iResult = recv(ClientSocket, (char*)recvbuf, recvbuflen, 0);
        Sleep(100);
```

```
        if (iResult > 0) {
            printf("Bytes received: %d\n", iResult);
            if (parse_data(recvbuf, iResult)) {
                const unsigned char message[] = "[message]\nmessage=\"Hello!
\"\nroom=\"lobby\"\nsender=\"ChatBot\"\n[/message]";
                send_data(message, sizeof(message), ServerSocket);
                Sleep(100);
            }

            iSendResult = send(ServerSocket, (char*)recvbuf, iResult, 0);
            Sleep(100);
            printf("Bytes sent: %d\n", iSendResult);
            iResult = recv(ServerSocket, (char*)recvbuf, recvbuflen, 0);
            Sleep(100);
            if (iResult != SOCKET_ERROR) {
                iSendResult = send(ClientSocket, (char*)recvbuf, iResult, 0);
                Sleep(100);
            }
        }
        else if (iResult == 0)
            printf("Connection closing...\n");
        else
            printf("recv failed with error: %d\n", WSAGetLastError());

    } while (iResult > 0 || WSAGetLastError() == WSAETIMEDOUT);

    iResult = shutdown(ClientSocket, SD_SEND);

    closesocket(ClientSocket);
    closesocket(ServerSocket);
    WSACleanup();

    return 0;
}
```

A.19 DLL Injector

Referenced in Chapter 7.1.

A DLL injector that loads the specified DLL into Urban Terror.

To load static and dynamic libraries, Windows executables can use the **LoadLibraryA** API function. This function takes a single argument, which is a full path to the library to load.

```
HMODULE LoadLibraryA(
    LPCSTR lpLibFileName
);
```

If we call **LoadLibraryA** in our injector's code, the DLL will be loaded into our injector's memory. Instead, we want our injector to force the game to call **LoadLibraryA**. To do this, we will use the **CreateRemoteThread** API to create a new thread in the game. This thread will then execute **LoadLibraryA** inside the game's running process.

However, since the thread is running inside the game's memory, **LoadLibraryA** will not be able to find the path of our DLL specified in our injector. To get around this, we have to write our DLL's path into the game's memory. To ensure that we do not corrupt any other memory, we will also need to allocate additional memory inside the game using **VirtualAllocEx**.

```
#include <windows.h>
#include <tlhelp32.h>

// The full path to the DLL to be injected.
const char *dll_path = "C:\\Users\\IEUser\\source\\repos\\wallhack\\Debug\
\wallhack.dll";

int main(int argc, char** argv) {
      HANDLE snapshot = 0;
      PROCESSENTRY32 pe32 = { 0 };

      DWORD exitCode = 0;

      pe32.dwSize = sizeof(PROCESSENTRY32);
```

```
	// The snapshot code is a reduced version of the example code provided
by Microsoft at
	// https://docs.microsoft.com/en-us/windows/win32/toolhelp/taking-a-
snapshot-and-viewing-processes
	snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
	Process32First(snapshot, &pe32);

	do {
		// We only want to operate on the Urban Terror process
		if (wcscmp(pe32.szExeFile, L"Quake3-UrT.exe") == 0) {
			// First, we need to get a process handle to use for the
following calls
			HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);

			// So we don't corrupt any memory, allocate additional
memory to hold our DLL path
			void *lpBaseAddress = VirtualAllocEx(process, NULL,
strlen(dll_path) + 1, MEM_COMMIT, PAGE_READWRITE);

			// Write our DLL path into the memory we just allocated
inside the game
			WriteProcessMemory(process, lpBaseAddress, dll_path,
strlen(dll_path) + 1, NULL);

			// Create a remote thread inside the game that will execute
LoadLibraryA
			// To this LoadLibraryA call, we will pass the full path of
our DLL that we wrote into the game
			HMODULE kernel32base = GetModuleHandle(L"kernel32.dll");
			HANDLE thread = CreateRemoteThread(process, NULL, 0,
(LPTHREAD_START_ROUTINE)GetProcAddress(kernel32base, "LoadLibraryA"),
lpBaseAddress, 0, NULL);

			// To make sure that our DLL is injected, we can use the
following two calls to block program execution
			WaitForSingleObject(thread, INFINITE);
			GetExitCodeThread(thread, &exitCode);

			// Finally free the memory and clean up the process handles
			VirtualFreeEx(process, lpBaseAddress, 0, MEM_RELEASE);
			CloseHandle(thread);
			CloseHandle(process);
			break;
```

```
            }
      } while (Process32Next(snapshot, &pe32));

      return 0;
}
```

A.20 Pattern Scanner

Referenced in Chapter 7.2.

A pattern scanner that will search a running Wesnoth process for the bytes `0x29 42 04`. These bytes are the opcode for the **sub** instruction that is responsible for subtracting gold from a player when recruiting a unit.

The scanner works by using **CreateToolhelp32Snapshot** to find the Wesnoth process and the main Wesnoth module. Once located, a buffer is created and the module's memory is read into that buffer. The module's memory mainly contains opcodes for instruction. Once loaded, we loop through all the bytes in the buffer and search for our pattern. Once found, we print the offset.

```
#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

// Our opcode pattern to scan for inside the process
unsigned char bytes[] = { 0x29, 0x42, 0x04 };

int main(int argc, char** argv) {
      HANDLE process_snapshot = 0;
      HANDLE module_snapshot = 0;
      PROCESSENTRY32 pe32 = { 0 };
      MODULEENTRY32 me32;

      DWORD exitCode = 0;

      pe32.dwSize = sizeof(PROCESSENTRY32);
      me32.dwSize = sizeof(MODULEENTRY32);

      // The snapshot code is a reduced version of the example code provided
by Microsoft at
```


Playrix Ex. 1024, page 493

```
      // https://docs.microsoft.com/en-us/windows/win32/toolhelp/taking-a-
snapshot-and-viewing-processes
      process_snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
      Process32First(process_snapshot, &pe32);

      do {
            // Only scan for patterns inside the Wesnoth process
            if (wcscmp(pe32.szExeFile, L"wesnoth.exe") == 0) {
                  module_snapshot =
CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pe32.th32ProcessID);

                  // Retrieve a process handle so that we can read the game's
memory
                  HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);

                  Module32First(module_snapshot, &me32);
                  do {
                        // Wesnoth is made up of many modules. For our
example, we only want to scan the main executable module's code
                        if (wcscmp(me32.szModule, L"wesnoth.exe") == 0) {
                              // Due to the size of the code, dynamically
create a buffer after determining the size
                              unsigned char *buffer = (unsigned
char*)calloc(1, me32.modBaseSize);
                              DWORD bytes_read = 0;

                              // Read the entire code block into our buffer
                              ReadProcessMemory(process,
(void*)me32.modBaseAddr, buffer, me32.modBaseSize, &bytes_read);

                              // For each byte in the game's code, check to
see if the pattern of bytes starts at the byte
                              for (unsigned int i = 0; i < me32.modBaseSize -
sizeof(bytes); i++) {
                                    for (int j = 0; j < sizeof(bytes); j++) {
                                          // If so, continue to check if all
the bytes match. If one does not, exit the loop
                                          if (bytes[j] != buffer[i + j]) {
                                                break;
                                          }

                                          // If we are at the end of the
loop, the bytes must all match
                                          if (j + 1 == sizeof(bytes)) {
```

```
                                                printf("%x\n", i +
(DWORD)me32.modBaseAddr);
                                            }
                                        }
                                    }

                                    free(buffer);
                                    break;
                                }

                    } while (Module32Next(module_snapshot, &me32));

                    CloseHandle(process);
                    break;
                }
        } while (Process32Next(process_snapshot, &pe32));

        return 0;
}
```

A.21 Memory Scanner

Referenced in Chapter 7.3.

A memory scanner for Wesnoth that allows you to search, filter, and edit memory inside the process. This code can be adapted to any target and is intended to show how tools like Cheat Engine work.

The scanner has three main operations:

- search
- filter
- write

The search operation will scan all memory from `0x00000000` to `0x7FFFFFFF` and use **ReadProcessMemory** to determine if the address holds a certain value. Because

ReadProcessMemory fails if a process doesn't have access to an address, the memory is scanned in blocks. Any values that match are saved to **res.txt**.

The filter operation iterates over all addresses in **res.txt** to determine if they match the provided value. If so, they are saved to **res_fil.txt**. At the end, **res_fil.txt** is copied over to **res.txt**.

The write operation uses **WriteProcessMemory** to write a passed value to all addresses in **res.txt**

CreateToolhelp32Snapshot is used to find the Wesnoth process, and **OpenProcess** is used to retrieve a handle.

```
#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

#define size 0x00000808

// The search function scans all memory from 0x00000000 to 0x7FFFFFFF for the
passed value
void search(const HANDLE process, const int passed_val) {
	FILE* temp_file = NULL;
	fopen_s(&temp_file, "res.txt", "w");

	unsigned char* buffer = (unsigned char*)calloc(1, size);
	DWORD bytes_read = 0;

	for (DWORD i = 0x00000000; i < 0x7FFFFFFF; i += size) {
		ReadProcessMemory(process, (void*)i, buffer, size, &bytes_read);

		for (int j = 0; j < size - 4; j += 4) {
			DWORD val = 0;
			memcpy(&val, &buffer[j], 4);
			if (val == passed_val) {
				fprintf(temp_file, "%x\n", i + j);
			}
		}
	}

	fclose(temp_file);

	free(buffer);
}
```

```
// The filter function takes a list of addresses in res.txt and checks to see
// if they match the provided value. If so, they are written to res_fil.txt
// After the initial pass, filter writes all the addresses in res_fil.txt to
res.txt
void filter(const HANDLE process, const int passed_val) {
	FILE* temp_file = NULL;
	FILE* temp_file_filter = NULL;
	fopen_s(&temp_file, "res.txt", "r");
	fopen_s(&temp_file_filter, "res_fil.txt", "w");

	DWORD address = 0;
	while (fscanf_s(temp_file, "%x\n", &address) != EOF) {
		DWORD val = 0;
		DWORD bytes_read = 0;

		ReadProcessMemory(process, (void*)address, &val, 4, &bytes_read);
		if (val == passed_val) {
			fprintf(temp_file_filter, "%x\n", address);
		}
	}

	fclose(temp_file);
	fclose(temp_file_filter);

	fopen_s(&temp_file, "res.txt", "w");
	fopen_s(&temp_file_filter, "res_fil.txt", "r");
	while (fscanf_s(temp_file_filter, "%x\n", &address) != EOF) {
		fprintf(temp_file, "%x\n", address);
	}

	fclose(temp_file);
	fclose(temp_file_filter);

	remove("res_fil.txt");
}

// The write function writes a value to every address in res.txt
void write(const HANDLE process, const int passed_val) {
	FILE* temp_file = NULL;
	fopen_s(&temp_file, "res.txt", "r");

	DWORD address = 0;
	while (fscanf_s(temp_file, "%x\n", &address) != EOF) {
		DWORD bytes_written = 0;
```

```
            WriteProcessMemory(process, (void*)address, &passed_val, 4,
&bytes_written);
      }

      fclose(temp_file);
}

// The main function is retrieving a process handle to Wesnoth, parsing the
program's arguments and passing
// execution to the proper operation
int main(int argc, char** argv) {
      HANDLE process_snapshot = 0;
      PROCESSENTRY32 pe32 = { 0 };

      pe32.dwSize = sizeof(PROCESSENTRY32);


      // The snapshot code is a reduced version of the example code provided
by Microsoft at
      // https://docs.microsoft.com/en-us/windows/win32/toolhelp/taking-a-
snapshot-and-viewing-processes
      process_snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
      Process32First(process_snapshot, &pe32);

      do {
            // Only retrieve a process handle for Wesnoth
            if (wcscmp(pe32.szExeFile, L"wesnoth.exe") == 0) {
                  // Retrieve a process handle so that we can read and write
the game's memory
                  HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);

                  // Convert the second parameter to a DWORD-like value
                  char* p;
                  long value = strtol(argv[2], &p, 10);

                  // Depending on the first argument, pass execution to the
search, filter, or write operations
                  if(strcmp(argv[1], "search") == 0) {
                        search(process, value);
                  }
                  else if(strcmp(argv[1], "filter") == 0) {
                        filter(process, value);
                  }
                  else if (strcmp(argv[1], "write") == 0) {
```

```
                        write(process, value);
                }

                // Close the process handle
                CloseHandle(process);
                break;
            }
        } while (Process32Next(process_snapshot, &pe32));

        return 0;
}
```

A.22 Disassembler

Referenced in Chapter 7.4.

A limited disassembler that will search for a running Wesnoth process and then disassemble `0x50` bytes starting at `0x7ccd91`. These instructions are responsible for subtracting gold from a player when recruiting a unit.

The disassembler works by using **CreateToolhelp32Snapshot** to find the Wesnoth process and the main Wesnoth module. Once it is located, a buffer is created and the module's memory is read into that buffer. The module's memory mainly contains opcodes for instruction. Once they are loaded, we loop through all the bytes in the buffer and disassemble them based on the reference provided by Intel here.

```
#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

#define START_ADDRESS 0x7ccd91

// The 8 possible operand values
const char modrm_value[8][4] = {
        "eax",
        "ecx",
        "edx",
        "ebx",
        "esp",
        "ebp",
        "esi",
```

```
	"edi"
};

// Table 2-2 in the reference document describes how to retrieve the operands
from a ModR/M value
int decode_operand(unsigned char* buffer, int location) {
	if (buffer[location] >= 0xC0 && buffer[location] <= 0xFF) {
		printf("%s, %s", modrm_value[buffer[location] % 8],
modrm_value[(buffer[location] >> 3) % 8]);
		return 1;
	}
	else if (buffer[location] >= 0x80 && buffer[location] <= 0xBF) {
		DWORD displacement = buffer[location + 1] | (buffer[location + 2]
<< 8) | (buffer[location + 3] << 16) | (buffer[location + 4] << 24);
		printf("[%s+%x], %s", modrm_value[buffer[location] % 8],
displacement, modrm_value[(buffer[location] >> 3) % 8]);
		return 5;
	}
	else if (buffer[location] >= 0x40 && buffer[location] <= 0x7F) {
		printf("[%s+%x], %s", modrm_value[buffer[location] % 8],
buffer[location+1], modrm_value[(buffer[location] >> 3) % 8]);
		return 2;
	}

	return 1;
}

int main(int argc, char** argv) {
	HANDLE process_snapshot = 0;
	HANDLE module_snapshot = 0;
	PROCESSENTRY32 pe32 = { 0 };
	MODULEENTRY32 me32;

	DWORD exitCode = 0;

	pe32.dwSize = sizeof(PROCESSENTRY32);
	me32.dwSize = sizeof(MODULEENTRY32);

	// The snapshot code is a reduced version of the example code provided
by Microsoft at
	// https://docs.microsoft.com/en-us/windows/win32/toolhelp/taking-a-
snapshot-and-viewing-processes
	process_snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
	Process32First(process_snapshot, &pe32);
```


Playrix Ex. 1024, page 500

```
      do {
            // Only disassmble the Wesnoth process
            if (wcscmp(pe32.szExeFile, L"wesnoth.exe") == 0) {
                  module_snapshot =
CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pe32.th32ProcessID);

                  // Retrieve a process handle so that we can read the game's
memory
                  HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);

                  Module32First(module_snapshot, &me32);
                  do {
                        // Wesnoth is made up of many modules. For our
example, we only want to scan the main executable module's code
                        if (wcscmp(me32.szModule, L"wesnoth.exe") == 0) {
                              // Due to the size of the code, dynamically
create a buffer after determining the size
                              unsigned char* buffer = (unsigned
char*)calloc(1, me32.modBaseSize);
                              DWORD bytes_read = 0;

                              // Read the entire code block into our buffer
                              ReadProcessMemory(process,
(void*)me32.modBaseAddr, buffer, me32.modBaseSize, &bytes_read);

                              DWORD loc = 0;
                              unsigned int i = START_ADDRESS -
(DWORD)me32.modBaseAddr;

                              // For each byte in the game's code, attempt to
disassmble it
                              while (i < START_ADDRESS + 0x50 -
(DWORD)me32.modBaseAddr) {
                                    printf("%x:\t", i +
(DWORD)me32.modBaseAddr);
                                    switch (buffer[i]) {
                                    case 0x1:
                                          printf("ADD ");
                                          i++;
                                          i += decode_operand(buffer, i);
                                          break;
                                    case 0x29:
                                          printf("SUB ");
                                          i++;
```

```
                                        i += decode_operand(buffer, i);
                                        break;
                                  case 0x74:
                                        printf("JE ");
                                        printf("%x", i +
(DWORD)me32.modBaseAddr + 2 + buffer[i + 1]);
                                        i += 2;
                                        break;
                                  case 0x80:
                                        printf("CMP ");
                                        i++;
                                        i += decode_operand(buffer, i);
                                        break;
                                  case 0x8D:
                                        printf("LEA ");
                                        i++;
                                        i += decode_operand(buffer, i);
                                        break;
                                  case 0x8B:
                                  case 0x89:
                                        printf("MOV ");
                                        i++;
                                        i += decode_operand(buffer, i);
                                        break;
                                  case 0xE8:
                                        printf("CALL ");
                                        i++;
                                        loc = buffer[i] | (buffer[i+1] <<
8) | (buffer[i+2] << 16) | (buffer[i+3] << 24);
                                        printf("%x", loc + (i +
(DWORD)me32.modBaseAddr) + 4);
                                        i += 4;
                                        break;
                                  default:
                                        printf("%x", buffer[i]);
                                        i++;
                                        break;
                                  }

                                  printf("\n");
                            }

                            free(buffer);
                            break;
                      }
```

```
                } while (Module32Next(module_snapshot, &me32));

                CloseHandle(process);
                break;
            }
      } while (Process32Next(process_snapshot, &pe32));

      return 0;
}
```

A.23 Debugger

Referenced in Chapter 7.5.

An example of a Windows debugger that will attach to a running Assault Cube 1.2.0.2 process, change a specific instruction to an **int 3** instruction (`0xCC`), and then restore the original instruction when the breakpoint is hit. The instruction modified only executes when the player is firing, allowing us to verify that the debugger is working as intended.

```
#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>

int main(int argc, char** argv) {
    HANDLE process_snapshot = NULL;
    HANDLE thread_handle = NULL;
    HANDLE process_handle = NULL;

    PROCESSENTRY32 pe32 = { 0 };

    DWORD pid;
    DWORD continueStatus = DBG_CONTINUE;
    DWORD bytes_written = 0;

    BYTE instruction_break = 0xcc;
    BYTE instruction_normal = 0x8b;

    DEBUG_EVENT debugEvent = { 0 };
```


Playrix Ex. 1024, page 503

```
    CONTEXT context = { 0 };

    bool first_break_has_occurred = false;

    pe32.dwSize = sizeof(PROCESSENTRY32);

    // Iterate through all active processes and find the Assault Cube process
    process_snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    Process32First(process_snapshot, &pe32);

    do {
        if (wcscmp(pe32.szExeFile, L"ac_client.exe") == 0) {
            // Save the pid and write the int 3 instruction to 0x0046366C
            pid = pe32.th32ProcessID;

            process_handle = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);
            WriteProcessMemory(process_handle, (void*)0x0046366C,
&instruction_break, 1, &bytes_written);
        }
    } while (Process32Next(process_snapshot, &pe32));

    // Attach the debugger and enter the main debug loop
    DebugActiveProcess(pid);

    for (;;) {
        continueStatus = DBG_CONTINUE;

        if (!WaitForDebugEvent(&debugEvent, INFINITE))
            return 0;

        switch (debugEvent.dwDebugEventCode) {
        case EXCEPTION_DEBUG_EVENT:
            switch (debugEvent.u.Exception.ExceptionRecord.ExceptionCode)
            {
            case EXCEPTION_BREAKPOINT:
                printf("Breakpoint hit");

                // Our main breakpoint code
                // This will first be hit when attaching, so ignore the first
time we enter this condition
                if (first_break_has_occurred) {
                    // If we break, open a handle to the thread that
triggered the event and revert back eip to the previous instruction
```

```
                    thread_handle = OpenThread(THREAD_ALL_ACCESS, true,
debugEvent.dwThreadId);
                    if (thread_handle != NULL) {
                        context.ContextFlags = CONTEXT_ALL;
                        GetThreadContext(thread_handle, &context);

                        context.Eip--;

                        SetThreadContext(thread_handle, &context);
                        CloseHandle(thread_handle);

                        // Then, write back the previous mov instruction so
our breakpoint does not trigger again
                        WriteProcessMemory(process_handle, (void*)0x0046366C,
&instruction_normal, 1, &bytes_written);
                    }
                }

                first_break_has_occurred = true;
                continueStatus = DBG_CONTINUE;
                break;
            default:
                continueStatus = DBG_EXCEPTION_NOT_HANDLED;
                break;
            }
            break;
        default:
            continueStatus = DBG_EXCEPTION_NOT_HANDLED;
            break;
        }

        ContinueDebugEvent(debugEvent.dwProcessId, debugEvent.dwThreadId,
continueStatus);
    }

    CloseHandle(process_handle);

    return 0;
}
```


Playrix Ex. 1024, page 505

A.24 Call Logger

Referenced in Chapter 7.6.

An example of a modified Windows debugger that will attach to a running Wesnoth process, locate all **call** instructions, and change them to an **int 3** instruction. When the breakpoint is hit, the location will be logged and the instruction will be restored. Then, after the instruction is executed, an **int 3** instruction will be rewritten to the location.

```
#include <windows.h>
#include <tlhelp32.h>
#include <stdio.h>
#include <Psapi.h>

#define READ_PAGE_SIZE 4096

int main(int argc, char** argv) {
    HANDLE process_snapshot = NULL;
    HANDLE thread_handle = NULL;
    HANDLE process_handle = NULL;

    PROCESSENTRY32 pe32 = { 0 };

    DWORD pid;
    DWORD continueStatus = DBG_CONTINUE;
    DWORD bytes_written = 0;

    BYTE instruction_break = 0xcc;
    BYTE instruction_call = 0xe8;

    DEBUG_EVENT debugEvent = { 0 };

    CONTEXT context = { 0 };

    bool first_break_has_occurred = false;

    HMODULE modules[128] = { 0 };
    MODULEINFO module_info = { 0 };

    DWORD bytes_read = 0;
    DWORD offset = 0;
    DWORD call_location = 0;
```


Playrix Ex. 1024, page 506

```
    DWORD call_location_bytes_read = 0;
    DWORD last_call_location = 0;

    unsigned char instructions[READ_PAGE_SIZE] = { 0 };

    int breakpoints_set = 0;

    pe32.dwSize = sizeof(PROCESSENTRY32);

    // Iterate through all active processes and find the Wesnoth process
    process_snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    Process32First(process_snapshot, &pe32);

    do {
        if (wcscmp(pe32.szExeFile, L"wesnoth.exe") == 0) {
            // Save the pid and open a handle to the process
            pid = pe32.th32ProcessID;

            process_handle = OpenProcess(PROCESS_ALL_ACCESS, true,
pe32.th32ProcessID);
        }
    } while (Process32Next(process_snapshot, &pe32));

    // Attach the debugger and enter the main debug loop
    DebugActiveProcess(pid);

    for (;;) {
        continueStatus = DBG_CONTINUE;

        if (!WaitForDebugEvent(&debugEvent, INFINITE))
            return 0;

        switch (debugEvent.dwDebugEventCode) {
        case EXCEPTION_DEBUG_EVENT:
            switch (debugEvent.u.Exception.ExceptionRecord.ExceptionCode)
            {
            case EXCEPTION_BREAKPOINT:
                // On the initial attachment breakpoint, replace all calls
with breakpoints
                if (!first_break_has_occurred) {
                    thread_handle = OpenThread(THREAD_ALL_ACCESS, true,
debugEvent.dwThreadId);

                    printf("Attaching breakpoints\n");
```

```
                    // In this code, we will only log all calls in the main
game module and not DLLs
                    // To locate the address space of this module, retrieve
all the modules and then get the first
                    // module's address space
                    EnumProcessModules(process_handle, modules,
sizeof(modules), &bytes_read);
                    GetModuleInformation(process_handle, modules[0],
&module_info, sizeof(module_info));
                    // Next, loop through each section of memory and locate
the opcode for calls (0xe8)
                    for (DWORD i = 0; i < module_info.SizeOfImage; i +=
READ_PAGE_SIZE) {
                        // ReadProcessMemory will fail if the memory
permissions are not correct for the page
                        // To prevent a single failure from skipping all
memory, read a single page of memory at a time
                        ReadProcessMemory(process_handle, (LPVOID)
((DWORD)module_info.lpBaseOfDll + i), &instructions, READ_PAGE_SIZE,
&bytes_read);
                        for (DWORD c = 0; c < bytes_read; c++) {
                            // If we detect an 0xe8, determine if it is a
call instruction
                            // We do this by first reading the next four
bytes after the 0xe8
                            // We then use these bytes to calculate the call
location
                            // If this location is outside the address space
of the main module, we ignore the opcode
                            if (instructions[c] == instruction_call) {
                                offset = (DWORD)module_info.lpBaseOfDll + i +
c;
                                ReadProcessMemory(process_handle, (LPVOID)
(offset + 1), &call_location, 4, &call_location_bytes_read);
                                call_location += offset + 5;
                                if (call_location <
(DWORD)module_info.lpBaseOfDll || call_location
>(DWORD)module_info.lpBaseOfDll + module_info.SizeOfImage)
                                    continue;

                                // If the call location is valid, write a
break instruction (0xcc) at the address
                                // In this case, 0x0040e3d8 and 0x0040e3ea
are two commonly called addresses that contain low-level code
```

```
                                // To prevent them from clogging up the logs,
we don't log these locations
                                // In addition, having thousands of
breakpoints can cause the executing program to crash
                                // Therefore, we limit the amount of
breakpoints to less than 2000
                                if (offset != 0x0040e3d8 && offset !=
0x0040e3ea && breakpoints_set < 2000) {
                                    WriteProcessMemory(process_handle,
(void*)offset, &instruction_break, 1, &bytes_written);
                                    FlushInstructionCache(process_handle,
(LPVOID)offset, 1);
                                    breakpoints_set++;
                                }
                            }
                        }
                    }

                    printf("Done attaching breakpoints\n");
                }
                else {
                    // If we break, open a handle to the thread that
triggered the event and revert back eip to the previous instruction
                    // Next, we will set single-step mode so that we can
restore our breakpoint
                    // After, we will write back the call instruction and
continue execution of the program
                    thread_handle = OpenThread(THREAD_ALL_ACCESS, true,
debugEvent.dwThreadId);
                    if (thread_handle != NULL) {
                        context.ContextFlags = CONTEXT_ALL;
                        GetThreadContext(thread_handle, &context);

                        context.Eip--;
                        context.EFlags |= 0x100;

                        SetThreadContext(thread_handle, &context);
                        CloseHandle(thread_handle);

                        WriteProcessMemory(process_handle,
(void*)context.Eip, &instruction_call, 1, &bytes_written);
                        FlushInstructionCache(process_handle,
(LPVOID)context.Eip, 1);

                        last_call_location = context.Eip;
```

```
                    }
                }

                first_break_has_occurred = true;
                continueStatus = DBG_CONTINUE;
                break;
            case EXCEPTION_SINGLE_STEP:
                // This code will executed after we enter single-step mode in
the breakpoint statement above
                // Single-step mode executes a single instruction and then
triggers this debug event
                // Therefore, after we execute the call we broke on above,
restore the break instruction so that our breakpoints don't
                // only fire a single time
                thread_handle = OpenThread(THREAD_ALL_ACCESS, true,
debugEvent.dwThreadId);
                if (thread_handle != NULL) {
                    context.ContextFlags = CONTEXT_ALL;
                    GetThreadContext(thread_handle, &context);
                    CloseHandle(thread_handle);

                    WriteProcessMemory(process_handle,
(void*)last_call_location, &instruction_break, 1, &bytes_written);
                    FlushInstructionCache(process_handle,
(LPVOID)last_call_location, 1);

                    printf("0x%08x: call 0x%08x\n", last_call_location,
context.Eip);
                    last_call_location = 0;
                }

                continueStatus = DBG_CONTINUE;
                break;
            default:
                continueStatus = DBG_EXCEPTION_NOT_HANDLED;
                break;
            }
            break;
        default:
            continueStatus = DBG_EXCEPTION_NOT_HANDLED;
            break;
        }

        ContinueDebugEvent(debugEvent.dwProcessId, debugEvent.dwThreadId,
continueStatus);
```

```
    }

    CloseHandle(process_handle);

    return 0;
}
```