KTH Computer Science
and Communication

Learning Reverse Engineering

PETTER DJUPFELDT
LUCAS TAUBERT

Bachelor's Thesis at NADA
Supervisor: Alexander Baltatzis
Examiner: Mårten Björkman

Abstract

Reverse engineering is the process of translating compiled programs to source code, and analyzing the resulting code. It is useful, since without knowing the inside of a program it is very difficult to build onto it, create software that can interact well with it, or simply create a similar program of your own.

The challenge of reverse engineering lies within the fact that a lot of information contained within the source code of a program is destroyed in the compilation process, and re-obtaining it is done through different kinds of analyses, of which some are discussed within.

To delve into this subject, we tried to reverse a few applications and games with different approaches, tools and methods, to find out how the most information could be acquired.

We found a set of methods that were optimal to us, and allowed us to modify a computer game both in runtime, and to edit the compiled bytecode to completely change the game's behaviour. The methods require some knowledge about software engineering, but they provide a good framework for a beginner to start reversing on their own.

Referat

Att lära sig reverse engineering

Reverse engineering är processen att översätta kompilerad kod till källkod, och att analysera denna. Det är ett användbart ämne, eftersom det är väldigt svårt att bygga vidare på 1 pt eller att återskapa ett program, om man inte vet hur det ser ut på insidan. Att veta det underlättar även om man ska skapa nya program som ska interagera med det äldre programmet.

Utmaningen med reverse engineering är att mycket av den information som finns i ett programs källkod går förlorad när den kompileras, och att återskapa denna genom olika sorters analys, av vilka en del diskuteras i rapporten.

För att lära oss om det här ämnet gav vi oss på att reverse:a några applikationer och spel. Vi använde olika infallsvinklar, verktyg och metoder för att ta reda på hur vi skulle få ut så mycket information som möjligt.

Vi hittade en mängd metoder som var optimala för oss och lät oss modifiera ett datorspel både i runtime och dess kompilerade byte-kod, för att helt ändra spelets beteende. Metoderna kräver viss kunskap om programmering, men de ger en god grund för en nybörjare att börja reverse:a på egen hand.

Statement of collaboration

Report writing and literature study

We contributed equally to writing the report. Most of it is written by us in collaboration, and we cannot really attribute any single part to one person. The literature study was also done by both of us in collaboration where both of us read most of the material, but L. Taubert had more focus on the ownedcore forums, while P. Djupfeldt focused on the paper about Deobfuscation.

Block Attacks

Analysis

The analysis of Block Attacks is primarily done by L. Taubert. He did all the analysis with Cheat Engine and furthermore the debugging with OllyDbg. P. Djupfeldt was responsible for the parts where we used TSearch.

Coding

Gaden and the hacked executable file were written by L. Taubert.

Spelunky

The analysis of Spelunky was primarily done by P. Djupfeldt, in both Cheat Engine and TSearch. L. Taubert did some minor research with Cheat Engine.

Minesweeper

The Minesweeper analysis was done by P. Djupfeldt.

Contents

Contents

Chapter 1

Introduction

Reverse engineering is the art of translating compiled software to source code. Often though, you only search for specific key parts of the software, to gain some knowledge of either how to modify it, or in order to learn more about the software.

Our goal with this project is to learn how to perform some basic reverse engineering. We also want to evaluate different methods and tools for reverse engineering, and see what you can do with the knowledge of a reversed program.

Our primary focus will be on reversing games. We will try to gain as much information as possible about the construction of the game by using different tools and methods, and see how much we can manipulate the games behaviour with that newfound knowledge.

1.1 Purpose

One of the prime examples of where reverse engineering is used is cheating in games. By first analyzing how a game works, and then using methods to modify this code at runtime or even permanently modify the program, you can gain significant advantages within the game.

Outside the domain of computer games, you can also gain insight in how to create your own software, if you for example find an obscure program which does interesting things, but not enough for your goals. With knowledge of reverse engineering you could then analyze it, and use its methods to create your own solution with pieces of code or ideas from that software.

It is also possible to use reverse engineering to achieve interoperability. If there is no public API to some software, but you still want to use it, it is possible to reverse the software to create some kind of outside interface to it. A famous case of this was in 1990 where a game developer named Accolade reversed Sega's Genesis platform to create games for it without having to negotiate with Sega about the proper licences. [3, p. 18][5]

The list of possibilities goes on, we recommend continued reading in Reversing - Secrets of Reverse Engineering for the interested, Eilam provides a really good

briefing on the subject in the first chapter of the book.

1.2 Problem statement

The main problem is that translating from machine readable byte code to human readable source code is not as simple as the other way around. The high level code contains a lot more information, which is lost in the translation, for instance names of classes, variables, functions, et cetera. The structure is lost as well, and all the comments and documentation. Code gets squeezed together, and if the coder is really trying to make the final compiled code unreadable, there can be clusters of pointers and memory allocations that switches locations at different launches of the software.

On the other hand, since a computer will have to be able to read the program, it logically has to be possible for a human to read it as well. It can be hard, but it is always possible.

Because a compiled program is so large, and contains so much information, it is impossible to just manually look through the code if you actually want to gather information. What you have to do instead is using different tools to understand the code. We will discuss some of them, and how we used them, in this paper.

Chapter 2

Background

Since reverse engineering is a quickly evolving subject, where new methods arise quickly, we have split our research into two parts. First the basic research, which includes reading up on how programs are stored in a computer, how compilers work, and how the assembly language works. This is our primary literature study. The second part is experimenting, where our primary focus lies, that will be described in later chapters.

To give the reader a good overview of our starting point, we will also include information about some terms that will be used throughout this paper, and a reference to the tools we will use.

2.1 Terms

Reversing: Performing reverse engineering on software. **Memory editing**: Changing the behaviour of running software by editing the allocated memory for that program during runtime. [**Assembly/DLL**] **Injection**: Changing or evaluating the behaviour of running software by programming your own code which you insert into the running program, forcing it to run that code. **Static program analysis**: Analysing a program without executing it; can be reading source code, byte code, etc. **Dynamic program analysis**: Analysing a program during runtime, e.g. with a debugger.

2.2 Literature study

2.2.1 x86 Disassembly

This book in an online wikibook (open source book), which goes in-depth about disassembly programs written for the x86 assembly language. The level of expertise expected by the reader is decent knowledge about the assembly language, and some understanding of how computers and operating systems works.

It goes through the subjects of different compilers, different assembly languages, discusses tools for disassembly and decompiling, and several problems you can run into while decompiling or disassembling programs, and often how to solve them.

This is a wikibook, which means that anyone can modify it. On the other hand, anyone can also correct errors contained within, so in most cases the reliability can be trusted. If we found anything that did not make sense, we looked to another source for confirmation.

2.2.2 Reversing: Secrets of Reverse Engineering

Reversing is a book containing a lot of information of reversing. The author brings up information of why and when reverse engineering is useful, what can be achieved with it, and explains on a highly technical level different methods of reverse engineering.

The book is written by a professional software engineer and reverser, and the target audience is a person with extensive knowledge of software development, and skills in lower level languages, and program structures.

It contained very good information, but it was also aimed at people with knowledge sometimes quite far outside our own expertise, which made some of the information difficult to absorb. The primary information we gathered from this book was the more introductory information of how reversing is used.

2.2.3 Deobfuscation - Reverse Engineering Obfuscated Code

This is a scientific paper about how to deal with obfuscation techniques. It describes a few techniques for obfuscating code and how to deal with them as a reverser, and evaluates the results of these deobfuscation techniques. It concludes that obfuscation is possible to bypass.

As it is a scientific paper on a narrow subject within reverse engineering, it is very formally written and requires at least some understanding of the subject beforehand. A grasp of formal logic is also needed to absorb the entirety of the paper, but not necessary to understand the basic ideas.

2.2.4 Ownedcore Memory Editing Forums

This is a section of a large cheating forums for computer games, primarily massive multiplayer online games like World of Warcraft. In this section, the reversing parts of game cheating is explained, discussed and researched in depth. You can find texts written by complete beginners asking for help, but also texts by very skilled programmers who explain methods and contribute to discussions with their knowledge.

In here we found more substantial methods of what you can do with knowledge from reversing. The focus within this forum lies more on the part after you have reversed a piece of software to gain information about it, and what you can do with

this knowledge. In here we also found really good references to further reading, and information about tools for reversing.

While reading the information here you have to keep in mind that anyone could have written it. There is a lot of good information, but there is also a lot of junk, and people who think they know more than they actually do. Before trusting any information written here, we cross-referenced with more reliable sources, or simply tested the claims ourselves to confirm or reject them.

2.3 Tools

2.3.1 OllyDbg 2.01 alpha

OllyDbg is a debugger. This software grants the possibility to debug compiled programs, with all that comes with it. Setting breakpoints, monitor values, and everything else that you can expect from a debugger is possible with this.

Editing the program memory to change program functionality is also possible with this tool, if you only want your changes to apply within the scope of one runtime.[6]

2.3.2 HxD 1.7.7.0

HxD is a freeware hex editor. The only end we use this program for is editing compiled files. It opens an .exe file, to show its bytecode in hexadecimal form, all of which is open for editing.

When a certain piece of code that should be changed is isolated within a program, this software is excellent for the purpose of just entering the file and replacing that code, with its good navigational controls and search methods.[7]

2.3.3 TSearch 1.6b

TSearch is a freeware tool for scanning and editing memory addresses, performing code injections and debugging. It also contains functionality for generating trainers, and some other functions such as a built-in calculator and hexadecimal to decimal converter.

TSearch primary use is for cheating in games by editing them during run time or generating trainers for them, but it can open and edit any program currently running in memory, not just games.[8]

2.3.4 Cheat Engine 6.1

Cheat Engine is an open source tool designed for modifying single-player games during runtime, to modify the difficulty by changing some key parameters. It has matured a lot since its release in 2000 though, and can now be used for a lot of advanced reversing, not only in games.

It comes with a scanning feature for finding variables or code, a debugger, a disassembler, and additional useful tools for reversing programs. It also contains an interesting tool for especially cheating: A trainer generator. Based on the memory values that you have saved within Cheat Engine after scanning, you can automatically create a trainer, which modifies these values when you press buttons or hotkeys.[10]

2.3.5 BlackMagic and C#

BlackMagic is a library written in C# which uses Windows API calls to gain access to running processes under Windows. With this library, reading and writing to memory allocated to specified processes becomes very straightforward.

BlackMagic also contains other tools, such as injecting assembly code into a running process, and other functions.[9]

2.4 Problems

2.4.1 Obfuscated code

A measure a developer might take to protect its code is obfuscating it. Code obfuscation is an umbrella term for a number of techniques for making code harder to reverse. These include changing the structure of the program, its logic, data and layout, without changing its functionality. This can be done either manually, by renaming functions and variables or making confusing function calls, or automatically, by using an obfuscation tool.

Obfuscated code is a problem for a reverser because it is harder to read for a human. Changing the layout of the code is no problem for a computer, it can still read it perfectly. But for a human, not having a clear, structured path to follow turns confusing very quickly. Likewise, adding pointless algorithms will make it much harder for a human to understand what the program does. Adding these pointless functions of course comes with a cost: the program might require more processing power and become slower. [3, p. 327]

The methods above are used for obfuscating the code during runtime, but might not be as big a problem during a static program analysis. A method for making static program analysis harder is introducing execution paths that are not authentic. While they will never be executed during runtime, they will introduce false information into the result of a static analysis. [2]

2.4.2 Optimized code

When a program is written in a high-level language, like C++, the programmer often goes through a lot of effort to structure and organize the program code. Code is put into classes and structs, with thought-out variable names and extensive documentation. Function names are selected with great care, so that you know

exactly what the different functions actually do, even if you did not write them yourself.

When this is compiled, most of it goes away. All the variable names, class names and comments disappear completely. But it does not end there. Most high-level compilers do not stop at just translating code to its low-level counterpart, they also optimize the code, to increase the speed at runtime. They might change methods using tail recursion to iteration, reform boolean expressions within if-clauses, and generally just change snippets of code to mathematically equivalent pieces that run faster on a machine, but is a lot harder for a human to read.

So when you compile and decompile a program, you do not only lose all the structural conveniences, but you also receive a result that is structured different from the original result. [1, p. 63]

2.4.3 Pointers

Pointers are variables which does not explicitly store a value, but instead an address in the memory. They can become a problem for reversers when they are used extensively, and in chains (pointer pointing to a pointer, pointing to a pointer, etc.). The reason is that a value can reside at a location in memory, but when this value is edited, it is instantly overwritten (or even ignored) since the real information was stored at a pointer at a completely different place.

It also poses a problem for when you want to use the memory addresses within a trainer software, since the base pointer could change its address, which results in the interesting data being stored at different places in the memory at different launches, even related to the base address of the program.

2.5 Solution approaches

A lot of these approaches are not found in our literature, and hence no sources are provided. Instead, we have found these solutions by experimenting, in many cases with guidance from our literature, in others just by using the functions provided by the tool, and in all by using our knowledge about software engineering and problem solving; and our common sense.

2.5.1 Already known data

When you want to reverse a well-known program or a program which is built upon a well-known platform, you rarely have to do all the work alone. Even if the structure of programs like, for instance, popular games and operating systems can be very complex, there are often resources available by people who have found bits and pieces of the code, and made it publicly available. This can include the structure of object management, key memory locations within the program, or bits of the engine that can be used and modified.

A good example of this is within the ownedcore forums, where you can find extensive amounts of information about most popular games, especially World of Warcraft. With all the information available there, it takes a programmer only a few hours to set up a running trainer for that game, which can perform tasks, or display normally hidden data, within the game automatically.

2.5.2 Debugging

A debugger is a very useful, perhaps even mandatory, tool for performing dynamic program analysis. Debugging is done by loading the program you wish to reverse into the debugger. Using the debugger you then set a breakpoint instruction in the code of the program. When it executes this instruction it is paused and control is handed to the debugger. You can now use the debugger to go through the program step by step, instruction by instruction. A good debugger will not only show you the instructions, but also the relevant registers and program stack, and what they contain at that point of execution.

All this information is very valuable for understanding exactly what a program does. By observing what happens both in the program and in memory when a certain instruction is called with a certain value, you get an idea of what that segment of code is for. With enough time this will let you piece together the entirety of the program.

While it is possible to glean some of the same information from just reading disassembled code as from debugging, it would be a very arduous task. A debugger helps the reverser keep track of exactly what is happening, with no risk of jumping to the wrong instruction. A good debugger will also give the reverser access to information not normally available, such as relevant system statistics and data, and data only available at runtime. [3, p. 116]

2.5.3 Scanning

Another useful method for finding interesting data within a program is scanning. Instead of stepping through the code, scanning is used for finding specific pieces and values of the code. If you know something about how data or code will behave, you can scan an open process for that behaviour. We will provide some examples of how scanning with Cheat Engine and TSearch works.

Finding specific value that changes

This is the most simple scan you can do. You know exactly what value you are looking for. To perform this scan, you simply put in the value and data type, and scan the process for that value. Since an open process contains a lot of data, it is probable that this value is found several times. You then trigger a change of this value (for example type a new row in a document handler, or trigger something that yields score in a game) and scan for the new value within the results. This process is repeated until the value is found.

Other types of value scans

If you do not know the value, you will have to use other kinds of scans. For example, to find your location within a 3d game world, you might know when you move to different places, but you have no idea of knowing your exact position.

For this event, we scan for an unknown initial value, which we then filter with a lot of "value changed", "value increased", "value decreased" and "value unchanged" scans after each other. We can for example repeatedly scan for value unchanged when we stand still to remove a lot of values which naturally change (enemies moving, time ticking, and so on). We can move forward, and search for an increase, move backwards and search for a decrease, and so on. This process is more time consuming, and often harder, but with some patience it can yield very interesting results.

Code scans

Sometimes you are interested more in some piece of code instead of a stored value. In Cheat Engine, you can search for this too. Two scans perfect for this are "find what accesses this address" and "find what writes to this address". Their functionality is exactly what it sounds like. They lock onto a memory location, and finds pieces of code which accesses or writes to this address.

In TSeach, this is called "AutoHack". You enable the debugger and select a memory address you wish to AutoHack. The debugger then checks which instructions access that address and displays them in the AutoHack window.

This code can then be debugged, replaced, changed, analyzed or used for any other ends.

Chapter 3

Method

Here we will begin to cover the second part of our research, the experimenting. We will go through our experiences, to give a hint of what worked out well for us, and what did not work out so well at all.

3.1 Our approach

When we started out with this project, neither of us knew much about reverse engineering. Both of us could use high level programming language - such as Java and C++ - and we had basic knowledge about lower level language - such as assembly and C.

The first method we used was debugging some simple programs with OllyDbg. We attempted to follow the code, and look at the program's output while stepping through the debugger. This was not very viable though, and only really worked if you had a clue of where to begin, or if the program was very simple, such as something just printing "Hello World!".

After that we attempted to find values with Cheat Engine and TSearch instead. We experimented with these tools to utilize their functionality, to make some sense of the programs we were trying to reverse. This yielded a lot more results than the debugging, and made the debugging so much more relevant with more knowledge about the program. We will explain our results in detail below.

3.2 Practical results

3.2.1 Spelunky

Spelunky is a randomized adventure game written in GML (Game maker language). It is a 2d game with the player controlled character running around in dungeons, collecting treasures and defeating enemies.

Scanning

By using Cheat Engine, we found the locations of various variables, for example health, number of bombs, current level, et cetera. Of note was that all these values were stored in 8 byte double types, even though many of them only could contain values 1-99.

After discovering the x and y coordinates for the player we also found, by experimenting and looking at the assembly code, that it appears that the instructions for storing these coordinates are always stored at the same space in the memory. Although, after further digging, we realized that these instructions were just the end of a long chain of events. What actually happened was that these lines of code were called every time some data should be written. In other words, it was used for pretty much everything.

Runtime memory editing

During an attempt to hack the game to give you infinite health by replacing the address where health is stored with a NOP instruction something interesting was discovered. Understandably, replacing a value with null made the game crash. The interesting part is that the crash report showed each line where an error occurred in plain source code. This could help with understanding just how the game works. That it shows plain source code might be specific to GML, but it shows that something as simple as crashing a program can reveal information about how it works.

Debugging

When we attempted to use TSearch's debugger to look at the disassembled code and find the instructions for writing to the address where health is stored, Spelunky crashed. This was a recurring problem for TSearch with Spelunky, and prevented any deeper study of memory addresses using this tool, as we would need the debugger to investigate any pointers to the address.

Difficulties

At different runs of the program, the variables we had discovered appeared at different locations in the memory. Our first thought was that they were simply offsets, and our next approach was to see them as offsets from the base address of the game. This, however, did not prove successful either. Not only did they appear at different memory locations, they also appeared at different offsets seen from the base address of the game.

Solutions

The way variables appeared at different locations in memory might hint to them being stored in a similar way as more advanced games, like World of Warcraft (WoW). As found in numerous threads in the ownedcore forum, WoW uses an

object manager to store data about objects (for instance the player). This object manager appears at different locations in the memory space at different launches, but there are pointers at a fixed offset that points towards this manager object. We attempted to find such a pointer, but with no success.

We then decided to abandon the research of this game, since it was compiled in a way that made advanced reversing very difficult, and beyond our scope.

3.2.2 Minesweeper for Windows 7

Minesweeper is a staple game for Windows. The goal of the game is to find all the mines without clicking on any of them. You discover mines by clicking on tiles. Tiles either hide mines, numbers or empty space. If you click on a mine, you lose. The numbers tell you how many mines are adjacent to that tile. To help you, you can tag a tile with either a flag or a question mark.

Scanning

By using Cheat Engine, we discovered the value for deciding how many mines should appear on a custom field. We also discovered that each tile is an object, and an un-clicked tile has one of several states: blank underneath, a number underneath, un-flagged, flagged or question marked.

TSearch was unable to detect Minesweeper as a running process, and thus unable to open it.

Runtime memory editing

on the field, but by directly editing the memory address for mines we discovered while scanning, we managed to fill the entire field with mines. This revealed something interesting about this version of Minesweeper: You cannot lose on the first click. No matter where you clicked, you won immediately, as that tile was always safe and every other was a mine.

We also edited the state of one of the tiles using the data we found while scanning. The states are all decided by the values 0-11 at the memory address; 0 being blank, 1-8 being a number, 9 unflagged, 10 flagged and 11 question marked . We were unable to find the value that decided if there was a mine hidden underneath, though. This might be connected to the fact that mine locations are not decided until after the first click, as shown above.

It was, however, possible to turn a mine into something else. By playing the game normally, you can become certain that a tile must hide a mine. Once this has been ascertained, it is possible to change the state of the tile, and there would be no mine. Once a tile had been revealed though, it was no longer possible to change its value with this method.

Debugging

We did not attempt any in-depth debugging of Minesweeper.

Difficulties

The greatest difficulty with Minesweeper was that it has very little visible data, and thus makes it a bit difficult to get an entry point for scanning.

Solutions

We had to start from what little information was available, such as the number of mines. To get any more interesting data, careful use of the scanning tools was required. We scanned for a likely range of values, and then performed an action to influence the values we were looking for, and repeated this method until we found something interesting.

3.2.3 Block Attacks - Rise of the Blocks

Block attacks is a puzzle game, quite similar to Tetris. The player sees a 6x14 matrix of different colored blocks, and the objective is to match them in lines or columns with three or more by swapping blocks, two at a time. You control a ”crosshair” with the keyboard, which will aim on two horizontally adjacent blocks, and by clicking the ”fire”-button, those two blocks swap places.

You receive score when you remove blocks and when a new row of blocks is pushed up. The speed also increases as you progress.

Scanning

We started out by finding the score value. Since the game displays the value, and it changes in a controllable manner, it was very easy to just search for this value using the methods of Cheat Engine.

To see how memory allocation worked, we restarted the game after finding this value, and attempted to access the same memory location. It resulted in the score value again, and through this we learned that the game uses static memory allocation within its assigned virtual space.

This was very good news, since it meant that we could use the same address values each time we ran the program. All the values we found had this kind of storage, which would allow us to modify pretty much the whole program without using any advanced scanning methods during runtime.

Using this knowledge, we performed more advanced scans, to find even more interesting values. One of the most key pieces of memory was where the game field itself is stored. Since the game is relatively simple, and the idea makes sense, we assumed that the blocks would be stored in a matrix of integers, where the number in the integer was 0 for an empty field, and another number for each of the colors.

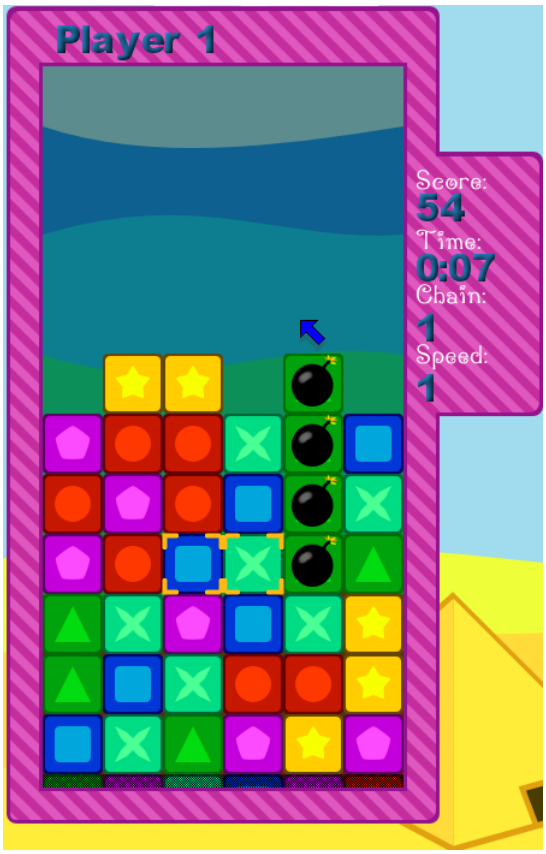

Figure 3.1. The main playing window of Block Attacks. This scene shows how four green blocks soon will be removed because they have been lined up.

Based on this assumption we first scanned for zeros, then we narrowed down the results by performing multiple scans for unchanged values while the game field had not pushed up a new row, since the game field should look identical until it did. When the game field had been pushed up, we scanned for a changed value, and then started the process over. This scan resulted in some strange addresses, but actually gave results. We received three addresses, which corresponded to three randomly located blocks on the field.

We also used TSearch in an attempt to replicate the results of the scan for block addresses we did with Cheat Engine. Once we had found one block address, it was easy to find the addresses for the first and last block in the matrix: by first making a list of all possible block states and their values we searched for the values of the first blocks in two adjacent columns. By finding these, we could calculate how far apart these addresses were in memory and thus how many rows each column contains. By applying some simple math we quickly found the start and end points of the matrix.

During a first, failed, attempt to find the block addresses with TSearch we

discovered something interesting. By chance we found the memory address for the gameï¿½s crosshair y-position. From this and the way other data has been stored in the game memory we deduced that the x-position should be located at a nearby memory address. We assumed that it would be stored at the address just before the y-position, since this is how x- and y-positions are usually defined. This assumption turned out to be correct, and this knowledge let us gain full control of the crosshair.

Runtime memory editing

By changing the block values found during scanning, we discovered that zero actually was one of the colors, and that an empty field was represented by the maximum value or a 32-bit unsigned integer, equalling -1. Some tweaking of these values revealed the location of the first block, and how exactly they were stored.

The blocks were stored in a matrix, where the first value was the bottom left block. The next value was the block on top of it, so the columns were stored together. Our first attempts at modifying these values assumed that they used a matrix of the same size as the playing field: 6x14. We then proceeded to clear this matrix within Gaden, but received some strange results. Some further investigation revealed that the matrix was actually 30 blocks high, even though it never filled up that much by ï¿½naturalï¿½ means.

We used TSearch to locate the value for ï¿½chainsï¿½, the number of successively deleted lines with one move, in the game. Finding this value proved easy, but we were unable to do anything with it directly, as it is constantly rewritten while the game plays. In an attempt to change this we replaced a segment of code with no operations. This led to something odd occurring in the game. The blocks stopped rising and the score counter started going up at a startling rate. Undoing the code replacement did nothing. The game was still playable, though it would run out of blocks as no new ones would appear on the field. We tried to understand why this happened, but because of limitations in TSearch we were unable to fully study the entire code segment.

Debugging

When the scanning process was complete, a lot of memory locations were acquired. To proceed with the reversing we then used the debugging tools in both Cheat Engine and OllyDbg.

From our scanning process, we knew the location of the piece of code that changes the type of block that appears at the bottom row, and we used this memory location as starting point in our debug. By stepping through the code from this point, we found the piece of code that changes the type of block that will appear the next turn. This piece of code and its location were noted, to be used in hex modifications.

Hex editing

With the code for replacing the bottom blocks isolated, we modified it. By using the hex editor HxD we located this bytecode within the executable file and replaced it with bytecode representing assembly nop-operations (operations that does nothing at all). This modified executable was then saved as a different file. When launched and played, this game would randomize colors for each column, but never change these colors. This resulted in the game just going on forever, the rows could not get up since they removed themselves every three ï¿½popsï¿½.

Gaden

With all this information about the structure and storage within the game, we could proceed with creating a trainer of our own to this particular game. We named this trainer software "Gaden", an acronym for "Game Administration Engine". We wrote it in C# with references to BlackMagic. By using everything we learned about the game, we created an admin-like panel for the game. Here we could change scores, manually swap any two pieces we wanted to, change score, and use a "panic button", which fills up the whole game field with blocks of the same color, that then gets removed by the game as matches, and provides an empty field.

The second and more advanced part of Gaden consists of an Artificial Intelligence (AI), which simulates actual gameplay of this game. Even though we do have access to all of the playing field, we decided to create a program which actually follows the rules, and plays the game. To accomplish this, the engine scans the playing field for three blocks of the same color which is located in adjacent rows. It then simply goes through the two lower rows, aligning these blocks to the location of the topmost block. This simple strategy is repeated until there are no more triples to be found, and then Gaden switches over to randomly swapping pieces, to create new opportunities.

This method proved to be quite efficient, and with a delay between each move long enough for it to still seem humanly slow it could solve puzzles at very high speeds. And when the delay was set to zero, Gaden could, with decent luck, solve puzzles at any speed.

There are plenty of optimizations available for this AI, but the goal was not to create a perfect AI for this game. We were more interested in demonstrating what can be done when you have information about the structure and internal workings of a program.

Difficulties

We had very few difficulties with Block Attacks, since it was such a simple program with very little, if anything, done to obfuscate its code.

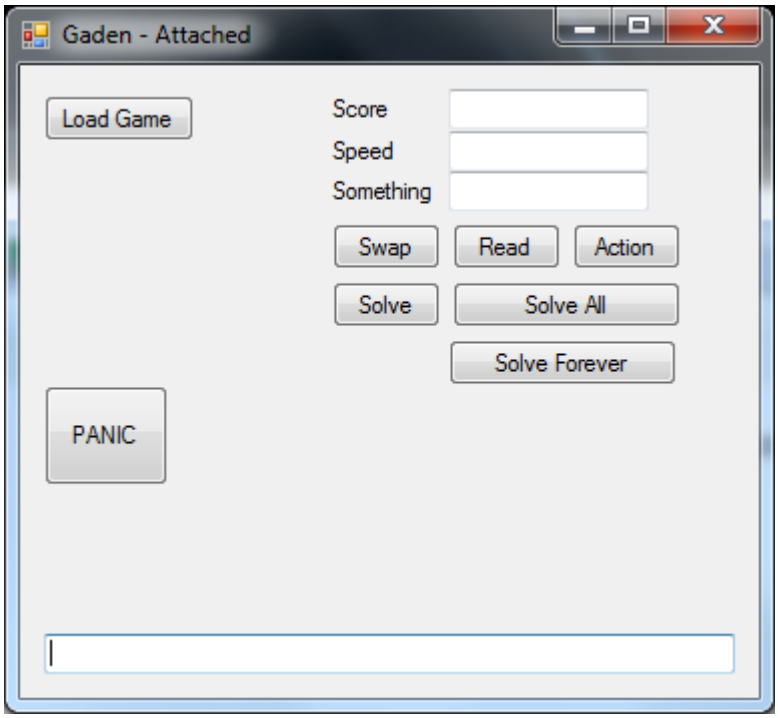

Figure 3.2. The GUI of our trainer when it is attached to the Block Attacks process. The buttons will cause Gaden to interact with the game in different manners.

Solutions

With a lack of difficulties, we simply applied what we had learned during the previous test on other games, and yielded satisfactory results.

Chapter 4

Discussion

4.1 Tool evaluation

4.1.1 Cheat Engine and TSearch

Cheat Engine

In addition to performing its expected duties with grace, Cheat Engine surprised in a positive manner by having very advanced features for reversing, coming with a debugger equipped with automatic assembly injection, and other very useful reversing tools.

The graphical user interface (GUI) is also well matured, and all buttons can be hovered for an explanation of what they do.

TSearch

One of the first things one notices when running TSearch is that it is not immediately apparent what every button does. Some are obvious, such as the big Open Process button, but others are near impossible to tell what they do just from looking at them. All buttons in the main window give a short description upon mouse over, but none in one of the other windows, the AutoHack window, where one looks at the disassembled code and performs code injections. These buttons required looking up in the TSearch manual to understand.

A limit in TSearch is that it, despite what the manual says, cannot open any running process. What decides if a process can be opened in TSearch is uncertain, as it cannot open any active program running and only a few passively running processes.

Conclusion

In conclusion, it is readily apparent that Cheat Engine is a better tool than TSearch. There was nothing TSearch could do that Cheat Engine could not do just as well, or better, except having a built-in calculator and hexadecimal to decimal converter.

These tools are however not very necessary, as the standard Windows calculator can perform these functions.

4.1.2 OllyDbg 2.01 alpha

OllyDbg is a very competent debugger. We did not use it overly much, mostly because the debugging step comes when you have pretty significant knowledge about the software already, and want to go in-depth. For our purposes when we needed a debugger, we found no flaws. The user interface was excellent, loading times were relatively fast and breakpoint management smooth.

4.1.3 HxD 1.7.7.0

We only used this tool for replacing source code, and for that end it performed great. It opened up, modified and changed the program instantly (as opposed to OllyDbg) since no debugger has to be attached, and the navigation was excellent.

4.1.4 C# with BlackMagic

BlackMagic was a great library for managing running processes. The compilation of the provided source code was quick, and linking to the completed DLLs was easy. We did not delve into advanced coding, and any high-level programming language could probably have sufficed. Although, since we worked in Windows, C# had relatively straightforward access to the necessary API-calls, which made this choice a good one, and removed a lot of configuration time from our project.

4.2 Method evaluation

4.2.1 Static analysis

We have used very little static analysis during this project, partially because it is a very daunting task to go through pure decompiled code, but also because the method did not lend itself very well to what we tried to do, i.e. editing games during run time. It is likely to be more useful for fully reversing a program, as that requires a deeper understanding of the program structure than what we did.

4.2.2 Dynamic analysis

Scanning

Scanning worked very well for us, and is the method we used the most throughout the project. We used it a lot in conjunction with debugging, where we would scan for a value we were interested in and then using a debugging tool to see how the value was changed by the program. Experimenting with values and memory addresses we found through scanning often gave us a better understanding of the program

we were working on. An example is when we discovered the true size of the block matrix in Block Attacks. We had first assumed that the matrix was 6x14 blocks large, the same size as the visible playing field, but by scanning and changing values we discovered that it was in fact 6x30 blocks large.

We used scanning as an entry point method. Even with no knowledge at all about a program, scanning based on qualified guesses provided very usable results, which we noted and built more research on.

Debugging

Debugging is very useful for finding details about how code executes. With knowledge about the program's structure, and some detailed information about where information is stored, breakpoints can be set in the program, and code analyzed.

Debugging worked out great when we had information to base the process on, and often yielded very interesting results. For example, we learned how variables are accessed and modified by debugging. By debugging, we could also find memory locations that we needed, by starting the debug process in a place we had already found by scanning, and following its trace.

We also attempted debugging as an entry point for reversing, which did not work out well. Programs are simply too large, debugging without any information about where to start becomes too much of a tedious and futile process.

4.2.3 Hex modifications

The hex modifications we performed were very simple. They were the last step in our chain, and when we got to it we already knew what code we had, and what code we needed. We had already analyzed the variables, debugged the program, and tried replacing some assembly lines during runtime. Only the final modifications were left to do here.

A pitfall can be to attempt to perform hex modification too early. Modifying even one assembly instruction to something invalid might completely break the program, so you have to possess the exact knowledge required before attempting this step, it is no method of analysis.

Chapter 5

Conclusions

5.1 Tool evaluation summary

The art of reverse engineering is a vast subject, and we have only touched the surface of it in this project. There are many tools available and we have tested and evaluated only a few of them, most notably Cheat Engine and TSearch.

At first glance the two programs appeared to be equally good, with only aesthetic differences and some very minor functional differences, TSearch having some tools for performing calculations in decimal and hexadecimal. But once we pushed their limits a bit more we noticed some serious flaws in TSearch, and became more and more positively disposed towards Cheat Engine.

OllyDbg, HxD and BlackMagic all worked as expected, and we think they are all excellent tools in the repository for a beginning reverser.

5.2 Method evaluation summary

The method that yielded best results for us was using a chain of tools in order, proceeding to the next step after sufficient knowledge had been acquired. The chain which worked out best for us is explained below.

The first thing to do was always scanning. This was done to isolate pieces of data and/or code within a running program, which could then be analyzed further. If the scan was thorough, knowledge about the structure within the program, some source code and methods of storage were all obtained.

The second step was debugging the running program. With the knowledge from scanning, breakpoints could be set, and code further analyzed. Particularly interesting pieces of code were read through, and knowledge of details not revealed in the scanning process could be obtained.

The third step was the modification. This could be done in two ways, either modifying the program itself with an hex editor, or creating an external program which would attach to the process in order to modify and/or read the program data in runtime.

5.3 Final conclusion

Reverse engineering is a very deep and interesting field, and we have learned a lot about the subject throughout the course of this project. We are, however, still no experts, as the subject is far too complex to fully absorb during the relatively short time we had for this project.

There are several applications for the art of reversing, many of them in the field of computer security. By using reversing techniques you can find out how a program works on a more fundamental level than by just using it and looking at what is presented to you while you use it normally. This can of course reveals holes in a program's security, and this can be used for a number of purposes by a reverser: from cheating in games, such as we have done in our experiments and shown in this report, to attacking businesses that rely on secure software. By using reversing techniques to detect these security holes a company will be able to deliver a safer product.

Bibliography

[1] *x86 Disassembly*, (2011). Available from: http://en.wikibooks.org/wiki/X86_Disassembly; [Accessed 31-01-2012]

[2] Udupa, S., Debray, S., Madou, M. *Deobfuscation - Reverse Engineering Obfuscated Code*, (2006). Tucson: The University of Arizona. Available from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1566145 [Accessed 04-04-2012]

[3] E. Eilam, *Reversing - Secret of Reverse Engineering*, (2005). Indianapolis, Wiley Publishing.

[4] *[Guide] How to make a Wow bot for complete newbs!*, (2011). Available from: http://tinyurl.com/ownedcore-wow-bot-how-to; [Accessed 15-02-2012]

[5] United States Court of Appeals *977 F.2d 1510*, (1993). Available from: http://bulk.resource.org/courts.gov/c/F2/977/977.F2d.1510.92-15655.html; [Accessed 11-04-2012]

[6] *OllyDbg*, (2011). Available from: http://www.ollydbg.de/; [Accessed 31-01-2012]

[7] *HxD - Freeware Hex Editor and Disk Editor*, (2011). Available from: http://mh-nexus.de/en/hxd/; [Accessed 04-04-2012]

[8] *TSearc Manual*, (2002). Available from: http://www.timsvault.com/cheattools/tsearch.htm; [Accessed 20-02-2012]

[9] *BlackMagic – Managed Memory Manipulation*, (2008). Available from: http://tinyurl.com/blackmagic-mem-manipulation; [Accessed 17-03-2012]

[10] *Cheat Engine*, (2011). Available from: http://www.cheatengine.org/; [Accessed 20-02-2012]

Appendix

AI code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

using Magic;

namespace BlockAtHACKs
{
   class AI
   {
       private uint[,] boardMatrix;
       private int xSize, ySize;
       private BlackMagic BMage;
       public static int SLEEPTIME = 100;
       private static int SLEEP_BETWEEN_CLEAR = 1000;

       public void init(int xsize, int ysize, BlackMagic BM)
       {
           BMage = BM;
           xSize = xsize;
           ySize = ysize;

           boardMatrix = new uint[xSize, ySize];
           UpdateBoard();
       }

       public void Scramble(int times)
       {
           SLEEPTIME = SLEEPTIME / 4;
           Random rng = new Random();
           for (int i = 0; i < times; i++)
           {
               MoveCursorSlowly((uint)rng.Next(12), (uint)rng.Next(5))
                   ;
               SwapAtCursor();
           }
           SLEEPTIME = SLEEPTIME * 4;
       }
```

```
public bool ClearRow()
{
    uint targetx = 11;
    uint targety = 0;
    bool found = false;
    while (targetx >= 2 && !found)
    {
        targety = 0;
        while (targety <= 5 && !found)
        {
            if (IsTopOfTriple(targetx, targety))
            {
                found = true;
                System.Console.WriteLine("Found " + BMage.
                    ReadUInt(GetMatrixMemPos(targetx,targety))
                    + " at x: " + targetx + ", y: " + targety);
            }
            targety++;
        }
        targetx--;
    }
    targety--;
    targetx++;
    System.Console.WriteLine("Target X: " + targetx + ", Target
         Y: " + targety);
    if (!found)
        return false;
    // x, y = FindMatching() >= 2
    // color = Read(x,y)
    uint color = BMage.ReadUInt(GetMatrixMemPos(targetx,targety
        ));
    // location1 = FindLocation(x-1,y,color)
    uint firsty = FindBlockOnRow(targetx-1, color);
    // location2 = FindLocation(x-2,y,color)
    uint secondy = FindBlockOnRow(targetx-2, color);
    System.Console.WriteLine("First Y: " + firsty + ", Second Y
        : " + secondy + ", Color: " + color);
    MoveCursorSlowly(targetx-1, targety);
    System.Console.WriteLine("Movement complete to place of
        origin");
    MoveCursorDir('d');
    Thread.Sleep(SLEEPTIME);
    MoveBlockBetween(firsty, targety);
    System.Console.WriteLine("Movement complete from first row
        .");
    MoveCursorDir('d');
    Thread.Sleep(SLEEPTIME);
    MoveBlockBetween(secondy, targety);
    System.Console.WriteLine("Movement complete from second row
        .");
    Thread.Sleep(SLEEP_BETWEEN_CLEAR);
    return true;
}
```

```
public bool IsTopOfTriple(uint x, uint y)
{
    if (NumberMatchingBelow(x, y) >= 2)
        return true;
    return false;
}

public void UpdateBoard()
{
    //for (uint x = 0; x < xSize; x++)
    //{
    //    for (uint y = 0; y < ySize; y++)
    //    {
    //        boardMatrix[x, y] = BMage.ReadUInt(
        GetMatrixMemPos(x, y));
    //    }
    //}
}

public void MoveCursorDir(char dir)
{
    uint curx = BMage.ReadUInt((uint)Form1.Globals.XHAIR_X);
    uint cury = BMage.ReadUInt((uint)Form1.Globals.XHAIR_Y);
    switch (dir)
    {
        case 'u':
            MoveCursor(curx + 1, cury);
            break;
        case 'd':
            MoveCursor(curx - 1, cury);
            break;
        case 'l':
            MoveCursor(curx, cury - 1);
            break;
        case 'r':
            MoveCursor(curx, cury + 1);
            break;
    }
}

private uint FindBlockOnRow(uint row, uint block)
{
    for (uint i = 0; i <= 6; i++)
    {
        if (BMage.ReadUInt(GetMatrixMemPos(row, i)) == block)
        {
            return i;
        }
    }
    return 100;
}

public void MoveBlockBetween(uint from, uint to)
```

```
{
    // from 6 to 5
    if (from > to) // We're going to move the block left. Align
         cursor 1 step to the right of the pos.
        MoveCursorSlowly(0, from - 1, false);
    else // Moving right, align on the spot.
        MoveCursorSlowly(0, from, false);
    while (to > from) // Move the block some steps right
    {
        from+=1;
        SwapAtCursor();
        MoveCursorDir('r');
        Thread.Sleep(SLEEPTIME);
    }
    while (to < from) // Move the block some steps left
    {
        from-=1;
        SwapAtCursor();
        MoveCursorDir('l');
        Thread.Sleep(SLEEPTIME);
    }
}

public void MoveCursorSlowly(uint x, uint y, bool movex = true)
{
    if (x > 12)
        x = 12;
    if (y > 4)
        y = 4;
    uint curx = BMage.ReadUInt((uint)Form1.Globals.XHAIR_X);
    uint cury = BMage.ReadUInt((uint)Form1.Globals.XHAIR_Y);
    while (curx != x && movex)
    {
        if (curx > x)
        {
            MoveCursorDir('d');
            curx--;
        }
        else if (curx < x)
        {
            MoveCursorDir('u');
            curx++;
        }
        Thread.Sleep(SLEEPTIME);
    }
    while ( cury != y )
    {
        if (cury > y)
        {
            MoveCursorDir('l');
            cury--;
        }
        else if (cury < y)
        {
```

```
            MoveCursorDir('r');
            cury++;
        }
        Thread.Sleep(SLEEPTIME);
    }
}

public void SwapAtCursor()
{
    uint x = BMage.ReadUInt((uint)Form1.Globals.XHAIR_X);
    uint y = BMage.ReadUInt((uint)Form1.Globals.XHAIR_Y);
    x++;

    uint oldleft = BMage.ReadUInt(GetMatrixMemPos(x, y));
    uint oldright = BMage.ReadUInt(GetMatrixMemPos(x, y + 1));

    if ((oldleft > 5 && oldleft < 4000000000) || (oldright > 5
        && oldright < 4000000000))
        return;

    BMage.WriteUInt(GetMatrixMemPos(x, y), oldright);
    BMage.WriteUInt(GetMatrixMemPos(x, y + 1), oldleft);
}

public void MoveSelectedToPos(uint x, uint y, uint block)
{
    // Finding closest block
    uint pos = 0;
    for (uint col = 0; col <= 6; col++)
    {
        if (BMage.ReadUInt(GetMatrixMemPos(col, y)) == block)
        {
            pos = col;
            break;
        }
    }
}

public int NumberMatchingBelow(uint x, uint y)
{
    int matching = 0;
    uint block = BMage.ReadUInt(GetMatrixMemPos(x, y));
    if (block > 100)
        return 0;
    for (uint row = x - 1; row >= 1 && row < 100; row--)
    {
        if (NumberInRow(row, block) > 0)
            matching++;
        else
            return matching;
    }
    return matching;
}
```

```
        private int NumberInRow(uint x, uint block)
        {
            int matching = 0;
            for (uint y = 0; y <= 6; y++)
            {
                matching += (BMage.ReadUInt(GetMatrixMemPos(x, y)) ==
                    block) ? 1 : 0;
            }
            return matching;
        }

        public void MoveCursor(uint x, uint y)
        {
            if (x < 0 || x > 12 || y < 0 || y > 4)
                return;
            BMage.WriteUInt((uint)Form1.Globals.XHAIR_X, x);
            BMage.WriteUInt((uint)Form1.Globals.XHAIR_Y, y);
        }

        public void Swap(uint x, uint y)
        {
            if (x < 0 || x > 12 || y < 0 || y > 4)
                return;
            MoveCursor(x, y);
            uint leftPos = GetMatrixMemPos(x + 1, y);
            uint rightPos = GetMatrixMemPos(x + 1, y + 1);
            uint oldLeft = BMage.ReadUInt(leftPos);
            uint oldRight = BMage.ReadUInt(rightPos);
            BMage.WriteUInt(leftPos, oldRight);
            BMage.WriteUInt(rightPos, oldLeft);
        }

        public uint GetMatrixMemPos(uint x, uint y)
        {
            return (uint)(Form1.Globals.BOARD_PTR) + x * 4 + y * 4 *
                30;
        }
    }
}
```

Gaden code

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Threading;

using Magic;

namespace BlockAtHACKs
{
   public partial class Form1 : Form
   {

       /* Form methods */

       public Form1()
       {
           InitializeComponent();
       }

       private void LoadButton_Click(object sender, EventArgs e)
       {
           if (IsLoaded)
           {
               return;
           }
           Initialize();
       }

       private void PanicButton_Click(object sender, EventArgs e)
       {
           ClearField();
       }


       private void SolveButton_Click(object sender, EventArgs e)
       {
           AInt.ClearRow();
       }


       private void SolveAllButton_Click(object sender, EventArgs e)
       {
           while (AInt.ClearRow()) ;
       }
```

```
private void SolveForeverButton_Click(object sender, EventArgs
    e)
{
    while (true)
    {
        if (!AInt.ClearRow())
            AInt.Scramble(1);
    }
}


private void SwapButton_Click(object sender, EventArgs e)
{
    //RandomSwap();
    try
    {
        AInt.Swap((uint)Convert.ToInt32(textBox1.Text), (uint)
            Convert.ToInt32(textBox2.Text));
    }
    catch (System.FormatException exc)
    {
        RandomSwap();
    }
}

private void ReadButton_Click(object sender, EventArgs e)
{
    textBox3.Text = "" + AInt.NumberMatchingBelow(Convert.
        ToUInt32(textBox1.Text), Convert.ToUInt32(textBox2.Text
        ));
    SetDebug(debug);
}

private void ActionButton_Click(object sender, EventArgs e)
{
    int newSpeed = Convert.ToInt32(textBox1.Text);
    AI.SLEEPTIME = newSpeed;
}

/* Editing part */

//Debug
public static string debug;


private BlackMagic BMage;
private AI AInt;
private bool IsLoaded;
private int[,] boardMatrix;

public enum Globals
{
    SCORE = 0x0028A538,
    SPEED = 0x0028A4F4,
```

```
            BOARD_PTR = 0x0028A1A8,
            XHAIR_X = 0x0028A520,
            XHAIR_Y = 0x0028A51C
        };

        public void Initialize()
        {
            BMage = new BlackMagic();
            if(!BMage.OpenProcessAndThread(SProcess.
                GetProcessFromProcessName("block_attack")))
            {
                Console.WriteLine("Yes you are fucked!");
                IsLoaded = false;
                return;
            }
            IsLoaded = true;
            AInt = new AI();
            AInt.init(30, 6, BMage);
            Console.WriteLine("Attached to le Blocks!");
            Form1.ActiveForm.Text = "Gaden - Attached at " + BMage.
                MainModule.BaseAddress;
        }

        private void RandomSwap()
        {
            Random random = new Random();
            uint x = (uint)random.Next(13);
            uint y = (uint)random.Next(5);
            AInt.Swap(x, y);
        }

        private void ClearField()
        {
            int offset = 0x0;
            for (int i = 0; i < 30; i++)
            {
                for (int j = 0; j < 6; j++)
                {
                    BMage.WriteInt((uint)(Globals.BOARD_PTR + offset),
                        0);
                    offset += 0x4;
                }
            }
        }

        public void SetDebug(string text)
        {
            Text_Debug.Text = text;
        }

    }
}
```