# Exercise 3 – Reverse Engineering

## Deadline: 24.04.2013 15:00

Note: Please make sure you're using the supplied Minesweeper version and not the version that comes with Windows. There are a few minor changes in the supplied version.

## Part 1 – Getting Acquainted (3 points)

The first step in every reverse engineering project is to get acquainted with the target program.

- Run `winmine.exe` (Minesweeper).
- Play around with the menus, settings and all program features.
    - Note that you can right-click the board while playing. Also try to click both mouse buttons at the same time. What does that do? (it's a useful feature, understand it).
    - It's good practice, while playing, to put yourself in the developer's shoes. How would you implement such features? (No need to answer this in the README).
- Finish the game in expert mode. You can get inspiration from this. Report your result in the submitted README file. ☺

## Part 2 – Manipulating Time (42 points)

As you probably noticed, all Minesweeper games are timed. In this part you'll manipulate the game's timer.

### Question 1

In this question you'll freeze the timer. Note that with the `winmine.exe` executable we've supplied the `.pdb` file, which means that you have the debug symbols of the executable. Make sure you tell IDA to load the supplied `.pdb` file. If done correctly, you should see meaningful function names.

A good place to start will be to explore the Names window in IDA, and try to locate interesting functions. Once you find where timer value is increased, you need to create a patched `winmine.exe`, named `winmine_freezed.exe`. Also please write what you did in the README file. We're not looking for a long answer but we're expecting to understand how you approached the problem and to see that you understand what you did. A good answer will consist of two parts – the first explaining how you found the interesting locations in the code and the second explaining exactly what your patch does.

Some notes:

- The timer value must remain 0 all the time (and not 1). Hint: you might need to patch more than one location.
- When sounds are on, a 'tick' is played whenever the timer increases. In addition to stopping the timer you'll also need to mute this 'tick' (consider bringing headphones to the lab).

## Question 2

Read a bit about the Windows Timer API. Create a patch (from the original `winmine.exe`, not from the previous question) in which the time goes twice as fast. Submit a file called `winmine_extreme.exe`. Again, besides the patched file, we also expect an explanation in the README.

## Question 3

In this question you'll modify the behavior of the time counter. Instead of showing the number of seconds elapsed, it will start at 120 seconds, and count down until it reaches zero. The following should hold:

- The timer will not advance below zero.
- The timer will advance only when the game is active (first square was already revealed).
- When the timer reaches zero, the game should be over by preventing the user from doing any action on the board. He should be able to restart the game from the menu.

In order to solve this question, you'll probably need to locate the Window Procedure in the minesweeper code. There are many ways to do so. Here is one of them:

- Set a breakpoint on the `DispatchMessageW()` function.
- Step into the function.
- Set a breakpoint on `.text` access of `winmine.exe`.
- The `switch` of the window procedure is right beneath you.

Explain in the README file why this method worked.

Now that you found the window procedure, you'll need to hook the `WM_LBUTTONDOWN and WM_RBUTTONDOWN` messages. Note that you'll need to add code. The way to do it is to add the code to an empty location at the end of the file, and jump to it and back.

Submit a patched executable named `winmine_countdown.exe`. As always, you'll need to explain what you did in the README file.

## Part 3 – Registry (5 points)

### Question 4

Minesweeper uses the registry to store its highscores. Please answer the following questions in your README file:

- Where exactly are the scores kept? (Full path in the registry)
- Where in the code are those scores updated? (Write the name of the function which is responsible to dump all of Minesweeper's settings to the registry)

## Part 4 – Understanding the board (50 points)

The Minesweeper program saves a representation of the board in memory. In this section you'll locate and investigate this representation.

### Question 5

- Locate the board in memory. Note that there are many ways to do so. Please submit the memory location (hex value) and how you found it.
- There are several entities on the game board (mine, unrevealed square, etc.). Investigate the board in memory and understand how these entities are represented. Submit a detailed table of all the entities in the game and their value (hint: there are some entities which are rarer than others, you must find them all. Yet another hint: each cell on the board is a single byte in memory).

### Question 6

In this question you'll write a program with two objectives:

- Help the user "cheat" by showing him the layout of the board including mines' locations.
- Save the user the trouble of clicking – solve the game automatically without user interaction.

The first part is quite straightforward. You'll need to write an application that reads Minesweeper's memory (at the location from question 4) and outputs the game board to the user. The output should conform to the following legend:

| Cell Type | Output Character |
|---|---|
| **Unrevealed cell without underlying mine** | `.` |
| **Unrevealed cell with underlying mine** | `*` |
| **Revealed cell with d neighboring mines** | `d` |

For example, let's say a (hypothetical) 10x10 board consists of mines only and it's fully unrevealed. Your program should output a 10x10 square of '*'.

The above table is **not complete**. Add more characters to mark the rest of the cell types. Please submit in the README the legend to your output.

Guidance:

- The output should match the structure of the board. Each board line should correspond to one output line.
- Your program must work with various board sizes.
- You may assume that only one Minesweeper is running and that it's running before your code is executed.
- Use `CreateToolhelp32Snapshot` to find Minesweeper's process ID from its process name (`winmine.exe`).
- Use `ReadProcessMemory` to read Minesweeper's relevant memory locations. Find out for yourself how to get the process handle that you'll need for this function call.
- Make sure your program works in various stages of the Minesweeper game.
- As always, document your code well and handle errors correctly.

For the second part, you'll need to send window messages to Minesweeper. We'd like to mimic the user interaction with Minesweeper by sending those messages from our program.

Guidance:

- Note that in the previous section you've understood the board, so all you need to do now is click on all unrevealed cells without a mine. Traverse the board from right to left and bottom to top, one column at a time. Wait 50 milliseconds between clicks.
- You can use Spy++ to find out which messages you need to send.
- Use `FindWindow` to find a handle to Minesweeper's window. You might need to know the name of the window. Again, Spy++ is your friend.
- Use the relevant message function to deliver your messages to the window.
- Note that before solving you'll have to simulate a click to a button on the board to start a Minesweeper game and generate the board.

Putting it all together:

Your program should behave as follows. When running your program with no arguments the usage should be printed:

```
Usage: trainer.exe -[v|s]
```

If your program received `-v` as its first argument, it should display the board (first part of the question). If your program received `-s` as its first argument, it should solve the board (second part of the question). You may assume that the user input is valid (`argv[1]` is either empty, `-v` or `-s`).

Submit two files named `trainer.c` and `trainer.h`. The files should compile with no error and warnings under default Visual Studio settings with compiler warning level 4 (Project settings, C/C++, Warning Level).

## Bonus Part – Resource hacking (up to 5 points)

In this section you'll need to show your artistic side. Use Resource Hacker to redesign Minesweeper. You'll be judged according to the amount of work put into your version of Minesweeper. You may swap whichever resources you want (Icons, Sounds, Menus, Bitmaps, etc.).

Submit your new version as `wackymine.exe`. Summarize your changes in your README file.

## Bonus Part – Find the cheat (up to 13 points)

In this section you'll find and change the cheat that is implemented in versions of Minesweeper prior to Windows Vista:

- Find and explain the cheat that's built into Minesweeper that allows you to know whether a square has a mine or not.
- Find all relevant code parts that implement this. Submit an .idb (IDA database) file with detailed remarks. Supply a list of all relevant memory addresses in the README.
- Change the key sequence to be g, o, d, shift + enter.
- Extra credit: name a game in which g,o,d was indeed the cheat sequence.

Submit your new version as `godlymine.exe`.

## Submission

All solutions should work under the Windows machines in Ross 19.

The following files should be included in the submitted tarball file:

- `winmine_freezed.exe`
- `winmine_extreme.exe`
- `winmine_countdown.exe`
- `trainer.c`
- `trainer.h`
- `wackymine.exe` (if you chose to submit the bonus)
- `godlymine.exe` (if you chose to submit the bonus)
- `winmine.idb` (if you chose to submit the bonus)
- README – Formatted according to the course guidelines.

A single TAR file named `ex3.tar` should be submitted. Make sure you run the pre-submit script before submitting the exercise.

We hold the right to randomly summon people for interviews, in which we will go over your solutions.

Good luck!