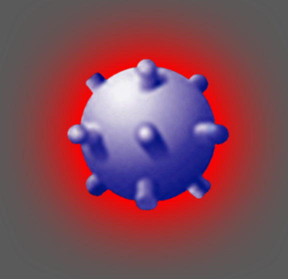
EECS 499
Introduction to Malware Analysis



# MINESWEEPER TRAINER

project 2

NOVEMBER 2017

Neil Orans
University of Michigan | College of Engineering
norans@umich.edu

# Contents

## Introduction

*"Game trainers are programs made to modify memory of a computer game thereby modifying its behavior using addresses and values, in order to allow cheating"*
- Wikipedia

I built a Minesweeper trainer as part of an independent study on malware analysis, which is modeled after a university course on malware analysis taught at RPI in 2015. The course resources can be found at this link: https://github.com/RPISEC/Malware.

This document describes the technical details of the trainer, with a focus on the five primary functions

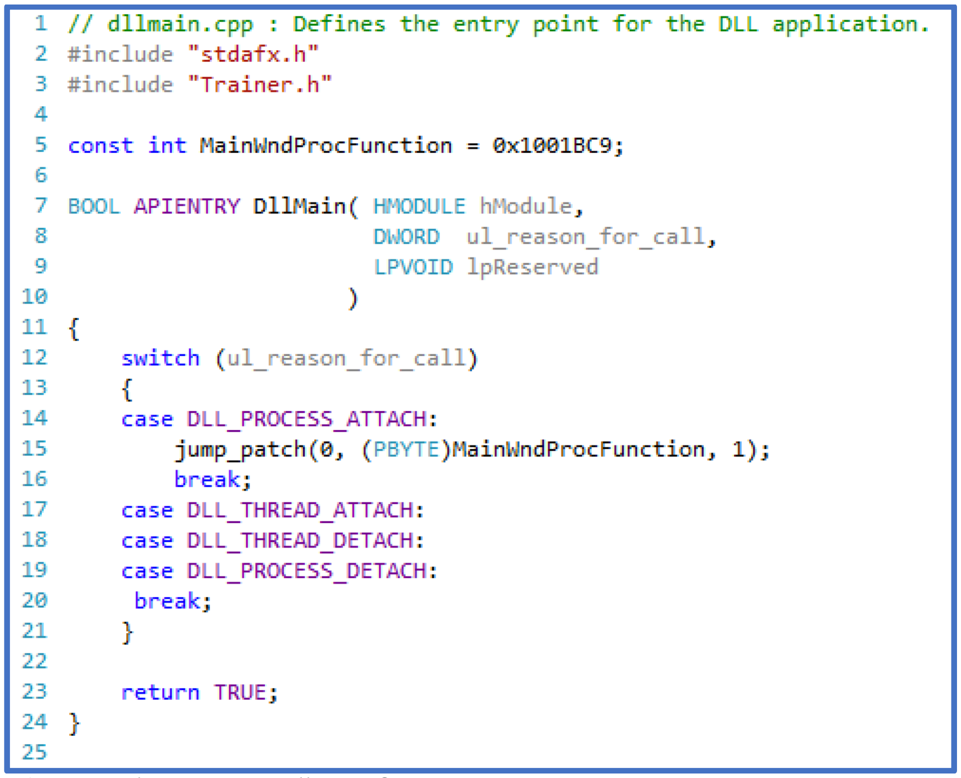
1. *Display game board* - prints the mine map in ASCII
2. *Show mines* - reveals all mines in the graphical interface
3. *Toggle timer* – stops or resumes timer
4. *Neutralize mines* – stepping on mine will not end game
5. *Auto-win* – automatically solve the current game board

The trainer is implemented as a DLL that gets injected into the game process. The appendix contains the download link for a DLL injector application and instructions for how to run the trainer. All code can be found at my personal GitHub page at http://bit.ly/minesweeper_trainer. Finally, I have made a short video displaying the functionality of the trainer. This video can be found at the following link: https://youtu.be/-htJzodr6s4.

# Injection and Start-up

## DllMain & Byte Patching

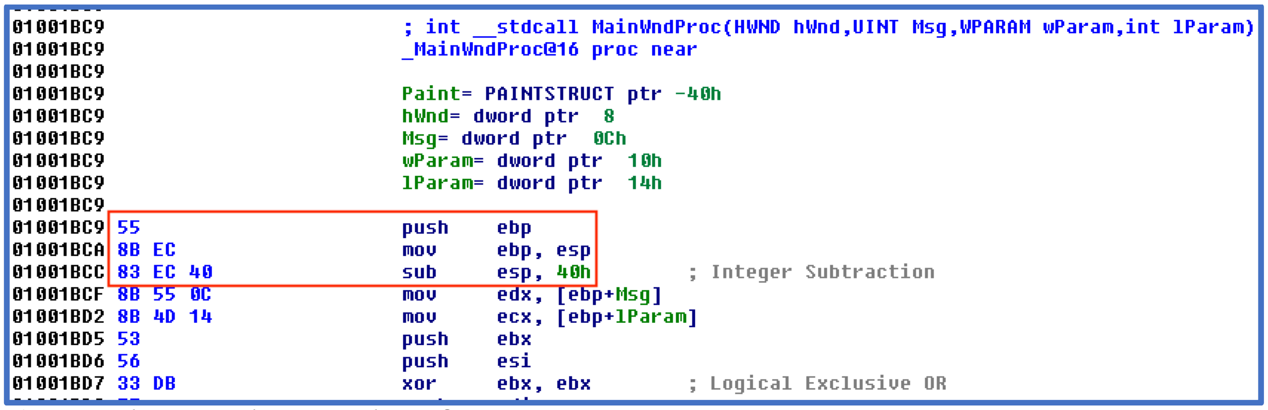
The trainer is a DLL file that gets injected into a running Windows XP Minesweeper process. Its DllMain function is very basic, and consists of only a single function call. The function call is to *jump_patch(void* dest, void* src)*, a function which places a JMP [dest] call at memory location specified by [src]. I chose to place this byte patch at the entry point to the *MainWndProc* function, which handles messages sent to the program's main window. I chose this location because it gives me control as soon as the user’s mouse hovers over the Minesweeper process, regardless if they have clicked any buttons yet.



```
// dllmain.cpp : Defines the entry point for the DLL application.
#include "stdafx.h"
#include "Trainer.h"

const int MainWndProcFunction = 0x1001BC9;

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        jump_patch(0, (PBYTE)MainWndProcFunction, 1);
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
     break;
    }

    return TRUE;
}

```

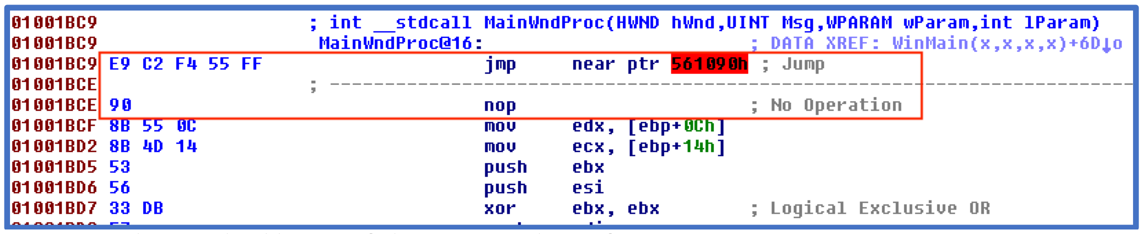
**Figure 1** The trainer’s DllMain function.

The entry point of the *MainWndProc* function is at 0x1001BC9 in the Minesweeper PE. Figure 2 shows the first few bytes of the *MainWndProc* function before the byte patching, and Figure 3 shows the patched bytes.



```
01001BC9                         ; int __stdcall MainWndProc(HWND hWnd,UINT Msg,WPARAM wParam,int lParam)
01001BC9                         _MainWndProc@16 proc near
01001BC9
01001BC9                         Paint= PAINTSTRUCT ptr -40h
01001BC9                         hWnd= dword ptr  8
01001BC9                         Msg= dword ptr  0Ch
01001BC9                         wParam= dword ptr  10h
01001BC9                         lParam= dword ptr  14h
01001BC9
01001BC9 55                      push    ebp
01001BCA 8B EC                   mov     ebp, esp
01001BCC 83 EC 40                sub     esp, 40h        ; Integer Subtraction
01001BCF 8B 55 0C                mov     edx, [ebp+Msg]
01001BD2 8B 4D 14                mov     ecx, [ebp+lParam]
01001BD5 53                      push    ebx
01001BD6 56                      push    esi
01001BD7 33 DB                   xor     ebx, ebx        ; Logical Exclusive OR
```

**Figure 2** The original MainWndProc function.



```
01001BC9                         ; int __stdcall MainWndProc(HWND hWnd,UINT Msg,WPARAM wParam,int lParam)
01001BC9                          MainWndProc@16:                ; DATA XREF: WinMain(x,x,x,x)+6D↓o
01001BC9 E9 C2 F4 55 FF                         jmp     near ptr 561090h ; Jump
01001BCE                         ; ---------------------------------------------------------------------
01001BCE 90                                     nop              ; No Operation
01001BCF 8B 55 0C                               mov     edx, [ebp+0Ch]
01001BD2 8B 4D 14                               mov     ecx, [ebp+14h]
01001BD5 53                                     push    ebx
01001BD6 56                                     push    esi
01001BD7 33 DB                                  xor     ebx, ebx ; Logical Exclusive OR
```

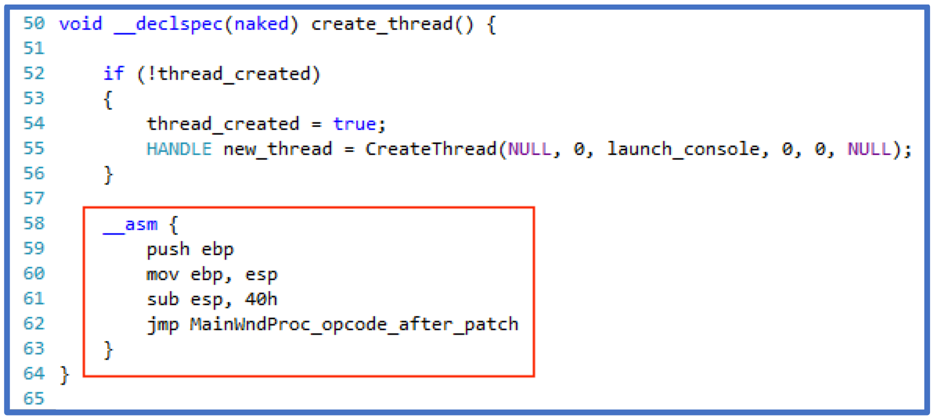
**Figure 3** The patched bytes of the MainWndProc function.

The jmp call only needs 5 bytes, but it replaces 3 instructions for a total of 6 bytes, so a single NOP is added. The destination of the jmp call destination changes each time the trainer gets injected because of DLL relocation. Once these bytes are patched, DllMain exits.

## Thread Creation

The destination of the byte-patched jmp call is a function in the trainer DLL that creates a thread. Since Minesweeper is a single-threaded process, in order to create a separate console interface for the user, a second thread must be started.

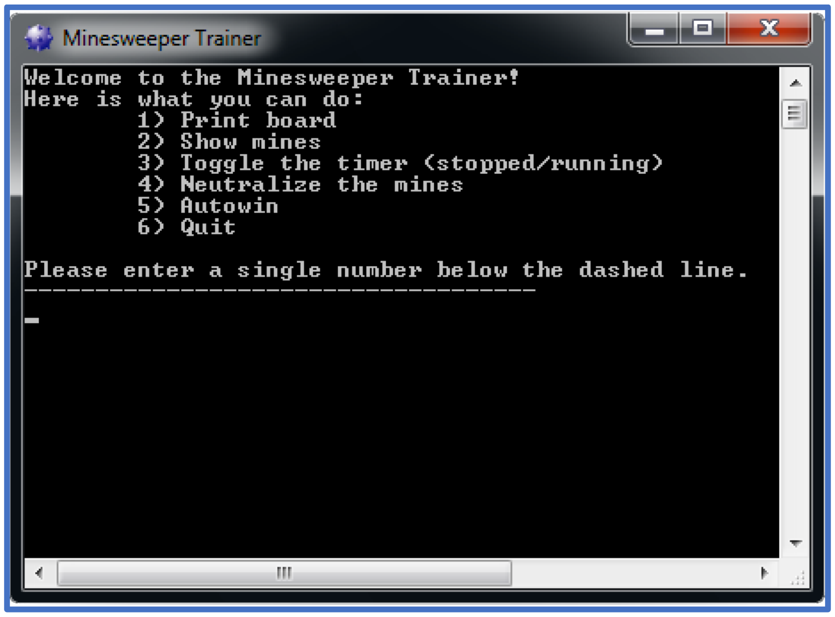
Recall that the location of the byte patch is the entry point of the function that handles all events sent to the programs main window. Consequently, this function runs multiple times per second. In order to only create **one** extra thread, I keep a global boolean variable in the DLL called 'thread_created' which gets set after the first thread launches. Every time program execution gets redirected after this first thread has been created, my code does nothing but rerun the instructions it patched and return execution to Minesweeper.



```
void __declspec(naked) create_thread() {

    if (!thread_created)
    {
        thread_created = true;
        HANDLE new_thread = CreateThread(NULL, 0, launch_console, 0, 0, NULL);
    }

    __asm {
        push ebp
        mov ebp, esp
        sub esp, 40h
        jmp MainWndProc_opcode_after_patch
    }
}

```

**Figure 4** The thread creation function. The first three instructions of the inline assembly (red box) are the exact same instructions from Figure 2. The fourth instruction returns execution to the instruction right below the red box in Figure 2.

The second thread starts a function called *launch_console*, which creates a console interface where the user receives instructions/output and types input.



```
Welcome to the Minesweeper Trainer!
Here is what you can do:
        1) Print board
        2) Show mines
        3) Toggle the timer (stopped/running)
        4) Neutralize the mines
        5) Autowin
        6) Quit

Please enter a single number below the dashed line.
------------------------------------
_
```

**Figure 5** The trainer interface.

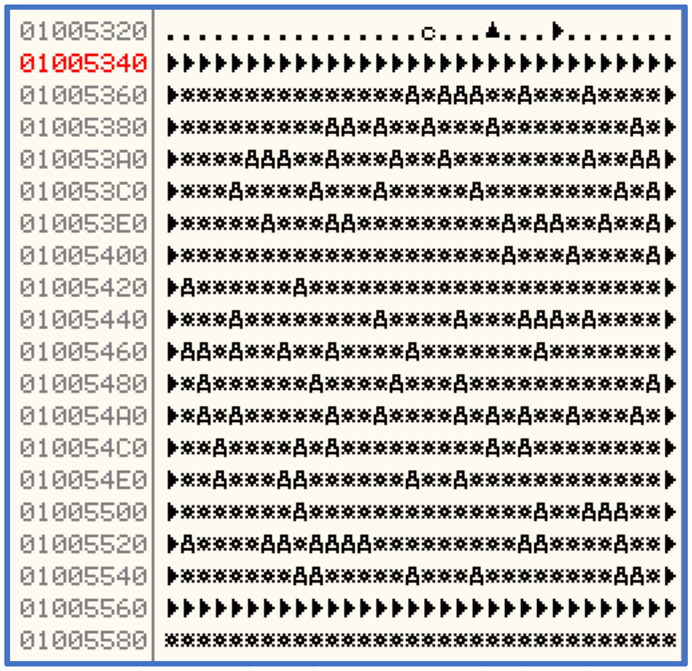
This thread runs indefinitely, and the console will stay as long as the Minesweeper process is running.

# Reversing Minesweeper and Trainer Functionality

## Extracting the game board

I started by launching the Minesweeper process in IDA. Thankfully the executable is linked with debugging information, and the function and variable names are pretty straightforward. There is a function called *DisplayBlk(x,x)*, which references a memory location with the symbol '__rgBlk'. There are several cross-references to this memory location from many different functions, further hinting that this location is important.

After opening the executable in OllyDbg, I set a breakpoint at this instruction. When the breakpoint hit, I examined the __rgBlk (0x1005340) area in memory. The game board is stored pretty apparently.



```
01005320 ................c...▲...►.......
01005340 ►►►►►►►►►►►►►►►►►►►►►►►►►►►►►►►►
01005360 ►☼☼☼☼☼☼☼☼☼☼☼☼☼☼Å☼ÅÅÅ☼☼Å☼☼☼Å☼☼☼☼►
01005380 ►☼☼☼☼☼☼☼☼☼ÅÅ☼Å☼☼Å☼☼☼Å☼☼☼☼☼☼☼☼Å☼►
010053A0 ►☼☼☼☼ÅÅÅ☼☼Å☼☼☼Å☼☼Å☼☼☼☼☼☼☼☼Å☼☼ÅÅ►
010053C0 ►☼☼☼Å☼☼☼☼Å☼☼☼Å☼☼☼☼☼Å☼☼☼☼☼☼☼☼Å☼Å►
010053E0 ►☼☼☼☼☼Å☼☼☼ÅÅ☼☼☼☼☼☼☼☼☼Å☼ÅÅ☼☼Å☼☼Å►
01005400 ►☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼Å☼☼☼Å☼☼☼☼Å►
01005420 ►Å☼☼☼☼☼☼Å☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼►
01005440 ►☼☼☼Å☼☼☼☼☼☼☼☼Å☼☼☼☼Å☼☼☼ÅÅÅ☼Å☼☼☼☼►
01005460 ►ÅÅ☼Å☼☼Å☼☼Å☼☼☼☼Å☼☼☼☼☼☼☼Å☼☼☼☼☼☼☼►
01005480 ►☼Å☼☼☼☼☼☼Å☼☼☼☼Å☼☼☼Å☼☼☼☼☼☼☼☼☼☼☼Å►
010054A0 ►☼Å☼Å☼☼☼☼☼Å☼☼Å☼☼☼☼Å☼Å☼Å☼☼Å☼☼☼Å☼►
010054C0 ►☼☼Å☼☼☼☼Å☼Å☼☼☼☼☼☼☼☼☼Å☼Å☼☼☼☼☼☼☼☼►
010054E0 ►☼☼Å☼☼☼ÅÅ☼☼☼☼☼☼Å☼☼Å☼☼☼☼☼☼☼☼☼☼☼☼►
01005500 ►☼☼☼☼☼☼☼Å☼☼☼☼☼☼☼☼☼☼☼☼☼☼Å☼☼ÅÅÅ☼☼►
01005520 ►Å☼☼☼☼ÅÅ☼ÅÅÅÅ☼☼☼☼☼☼☼☼☼ÅÅ☼☼☼☼Å☼☼►
01005540 ►☼☼☼☼☼☼☼ÅÅ☼☼☼☼☼Å☼☼☼Å☼☼☼☼☼☼☼☼ÅÅ☼►
01005560 ►►►►►►►►►►►►►►►►►►►►►►►►►►►►►►►►
01005580 ☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
```

**Figure 6** The game board in memory. This board is 16x30.

In memory, the board is stored with a border of bytes with the value 0x10 (the little triangles in Figure 6). The first byte of the actual board (1,1 on the game board) is stored at 0x1005361. Non-mine tiles are initially stored as 0x0F, and mines are 0x8F.

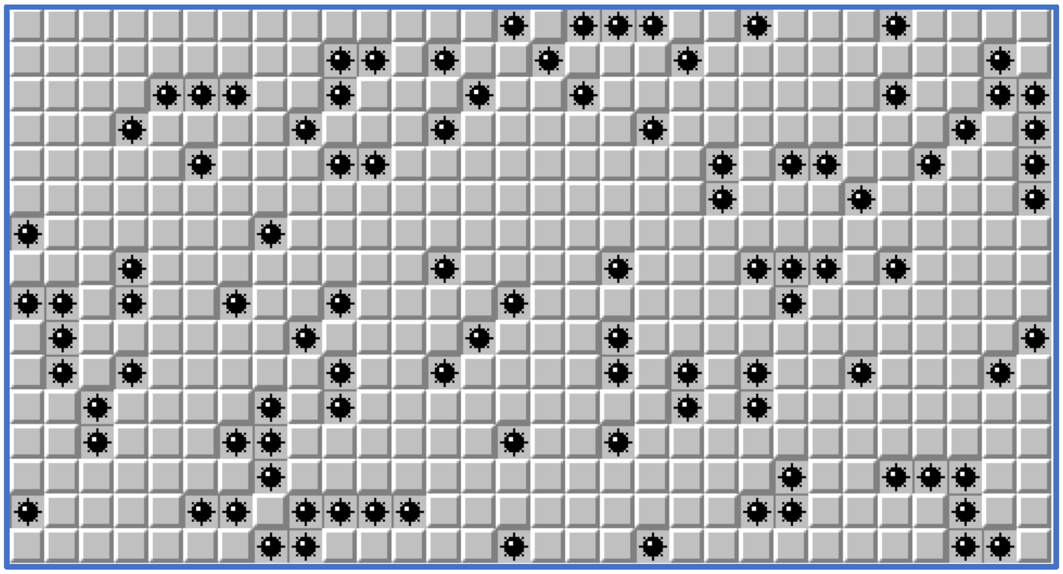
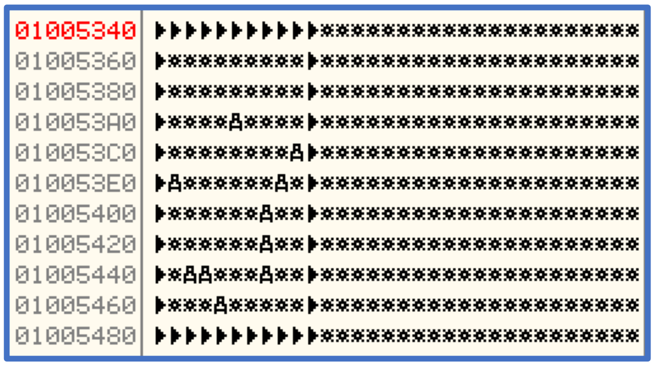


**Figure 7** The graphic representation of the game board from Figure 6.

Game boards that do not have a width of 30 (maximum) are still row-aligned by 32 bytes. For example, in the 9x9 board below, row one starts at 0x1005360, and row two starts 32 bytes later at 0x1005380.



```
01005340 ▶▶▶▶▶▶▶▶▶▶▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
01005360 ▶☼☼☼☼☼☼☼☼☼▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
01005380 ▶☼☼☼☼☼☼☼☼☼▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
010053A0 ▶☼☼☼☼Å☼☼☼☼▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
010053C0 ▶☼☼☼☼☼☼☼☼Å▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
010053E0 ▶Å☼☼☼☼☼☼Å☼▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
01005400 ▶☼☼☼☼☼☼Å☼☼▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
01005420 ▶☼☼☼☼☼☼Å☼☼▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
01005440 ▶☼ÅÅ☼☼☼Å☼☼▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
01005460 ▶☼☼☼Å☼☼☼☼☼▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
01005480 ▶▶▶▶▶▶▶▶▶▶▶☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼☼
```

**Figure 8** A 9x9 game board with 10 mines.

## Showing the mines

When a user loses or wins Minesweeper, the mines expose themselves. For a loss, the mines that were hidden appear as a bomb with a gray background, and the mine that killed the user is displayed with a red background. For a win, all the mines are shown as flags.

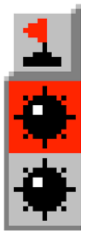
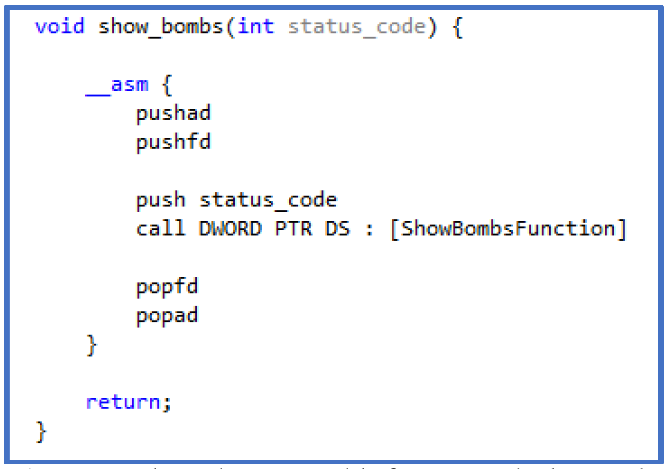


**Figure 9** From top to bottom; marked mine, exploded mine, revealed mine.

The subroutine responsible for this changing all of the mine values after a win or loss is (appropriately) titled *ShowBombs(x)*, and begins at 0x1002F80.

I set a breakpoint at this function to observe the argument passed into the function. I discovered that when I won (by cheating and looking at the game board in memory), the value 0xA was passed in, and when I lost, the value 0xE was passed in. There is some shifting and byte manipulation going on with this value that ultimately effects what is written to the game board in memory. This is not important – all that we need is the 'lose' status code.

Using this information, I could write in-line assembly code to call this function with the 'lose' status code, which shows the mines as bombs.

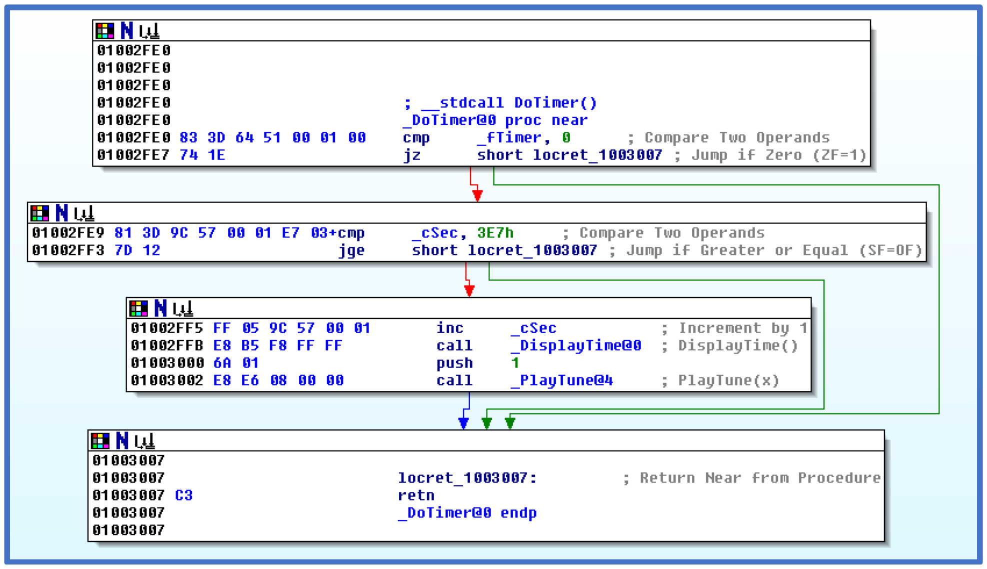


```
void show_bombs(int status_code) {

    __asm {
        pushad
        pushfd

        push status_code
        call DWORD PTR DS : [ShowBombsFunction]

        popfd
        popad
    }

    return;
}
```

**Figure 10** The in-line assembly function which reveals the mines.

## Toggling the timer

My analysis for this task started with searching the list of function names. There is a function called *DoTimer*, which looked to be a good starting point.

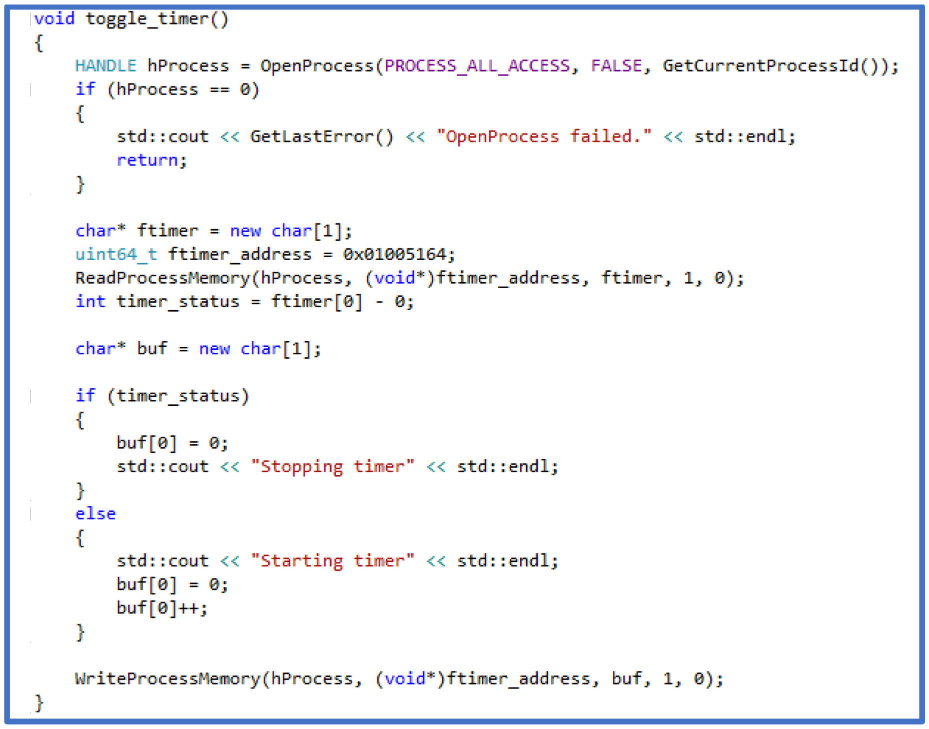


```
01002FE0
01002FE0
01002FE0
01002FE0                      ; __stdcall DoTimer()
01002FE0                      _DoTimer@0 proc near
01002FE0 83 3D 64 51 00 01 00 cmp     _fTimer, 0       ; Compare Two Operands
01002FE7 74 1E                jz      short locret_1003007 ; Jump if Zero (ZF=1)

01002FE9 81 3D 9C 57 00 01 E7 03+cmp     _cSec, 3E7h       ; Compare Two Operands
01002FF3 7D 12                   jge     short locret_1003007 ; Jump if Greater or Equal (SF=OF)

01002FF5 FF 05 9C 57 00 01       inc     _cSec             ; Increment by 1
01002FFB E8 B5 F8 FF FF          call    _DisplayTime@0    ; DisplayTime()
01003000 6A 01                   push    1
01003002 E8 E6 08 00 00          call    _PlayTune@4       ; PlayTune(x)

01003007
01003007                      locret_1003007:         ; Return Near from Procedure
01003007 C3                   retn
01003007                      _DoTimer@0 endp
01003007
```

**Figure 11** The DoTimer function

The function starts out by comparing a variable called _fTimer to the value 0. If it is zero, it jumps to the end of the function and returns without incrementing the _cSec variable. After further analysis, I discovered that the timer displayed in the top right of the game is the _cSec variable, so by setting the _fTimer variable to 0, we can skip the part of the game's code that increments the timer.

The function I wrote to toggle the timer checks the status of the timer in memory first (by reading the _fTimer variable). If it detects the clock is running, it freezes the clock. If the clock is frozen, it resumes the count.



```
void toggle_timer()
{
    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, GetCurrentProcessId());
    if (hProcess == 0)
    {
        std::cout << GetLastError() << "OpenProcess failed." << std::endl;
        return;
    }

    char* ftimer = new char[1];
    uint64_t ftimer_address = 0x01005164;
    ReadProcessMemory(hProcess, (void*)ftimer_address, ftimer, 1, 0);
    int timer_status = ftimer[0] - 0;

    char* buf = new char[1];

    if (timer_status)
    {
        buf[0] = 0;
        std::cout << "Stopping timer" << std::endl;
    }
    else
    {
        std::cout << "Starting timer" << std::endl;
        buf[0] = 0;
        buf[0]++;
    }

    WriteProcessMemory(hProcess, (void*)ftimer_address, buf, 1, 0);
}
```

**Figure 12** The toggle_timer function.

## Neutralizing the mines

To discover what happens when clicking on a mine, I followed the execution trace in OllyDbg. When any tile is selected, regardless of underlying value, the program first calls the function *StepSquare(row, col)*. *StepSquare* checks the underlying value, and if it detects a bomb, will direct program flow towards the *GameOver* function.

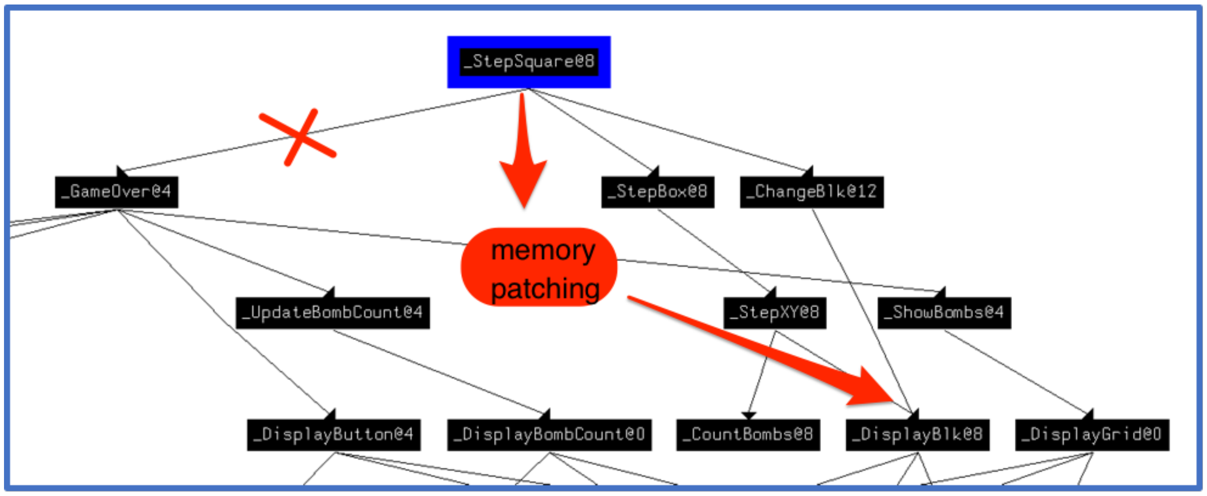
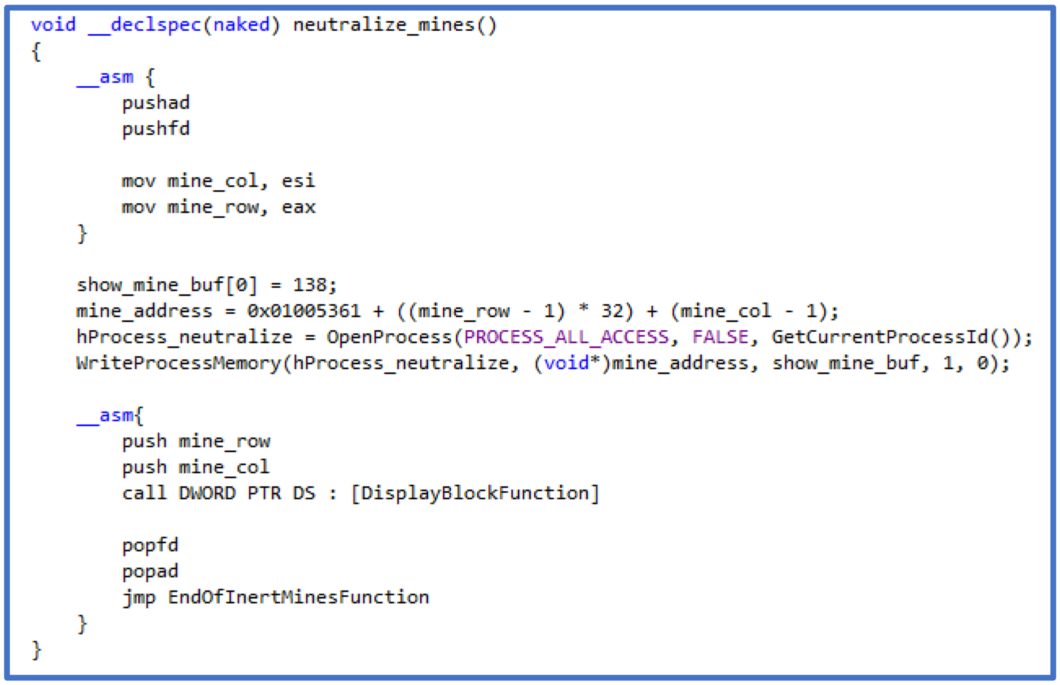


**Figure 13** A graph of function cross references, displaying the function hooking used to prevent an exploded mine from ending a game.

I hooked this control-flow by patching the bytes in memory right before the call to *GameOver* to redirect to a trainer function. The trainer function manually edits the selected mine in memory to be 0x8A which is the value needed for the graphic-drawing function *BitBlt* to draw an unexploded mine on the game board. After this location in memory is edited, the function calls the *DisplayBlk* function to draw the graphic, and then returns to the end of the *StepSquare* function.

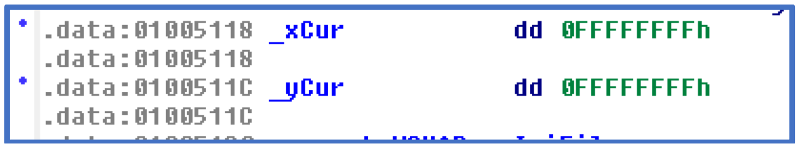


```
void __declspec(naked) neutralize_mines()
{
    __asm {
        pushad
        pushfd

        mov mine_col, esi
        mov mine_row, eax
    }

    show_mine_buf[0] = 138;
    mine_address = 0x01005361 + ((mine_row - 1) * 32) + (mine_col - 1);
    hProcess_neutralize = OpenProcess(PROCESS_ALL_ACCESS, FALSE, GetCurrentProcessId());
    WriteProcessMemory(hProcess_neutralize, (void*)mine_address, show_mine_buf, 1, 0);

    __asm{
        push mine_row
        push mine_col
        call DWORD PTR DS : [DisplayBlockFunction]

        popfd
        popad
        jmp EndOfInertMinesFunction
    }
}
```

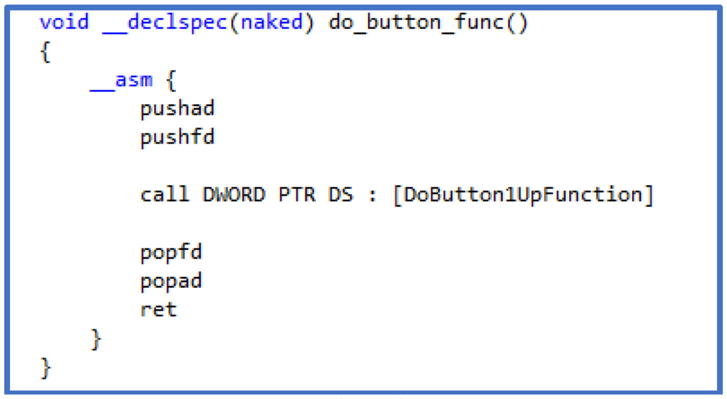
**Figure 14** The neutralize_mines function.

## Auto-win

The easiest way to auto-win the game is to simulate the press of each non-mine tile. The pressing down of a button in the game triggers a function called *DoButton1Up*. This function starts by fetching the value of the pressed tile's coordinates, which are stored by a different function at (0x1005118, 0x100511C).


```
.data:01005118 _xCur            dd 0FFFFFFFFh
.data:01005118
.data:0100511C _yCur            dd 0FFFFFFFFh
.data:0100511C
```
**Figure 15** The programs names for the tile's coordinates, xCur and yCur.

I wrote a for-loop that steps through each tile on the board, and if it detects a non-mine tile, will write that tile's coordinates into this location in memory. After the memory has been written, the button press function is called using in-line assembly.


```
void __declspec(naked) do_button_func()
{
    __asm {
        pushad
        pushfd

        call DWORD PTR DS : [DoButton1UpFunction]

        popfd
        popad
        ret
    }
}
```
**Figure 16** The do_button_func function, which simulates the pressing of a button.

When all the tiles are successfully marked, the game automatically calls its *GameOver(x)* function with an argument of '1', signifying a win.

## References

[1] https://stackoverflow.com/questions/17648966/handling-non-ascii-chars-in-c

[2] https://stackoverflow.com/questions/191842/how-do-i-get-console-output-in-c-with-a-windows-program

[3] http://www.rohitab.com/discuss/topic/35537-cc-reverse-engineering-tutorial-for-newbies/

# Appendix

DLL Injector is a free application that is used to inject DLLs into processes on Windows machines. It can be downloaded from https://www.dllinjector.com/. Instructions on how to inject a DLL using DLL Injector can be found at https://www.dllinjector.com/dll-injector-tutorials/.

Once the DLL has been injected into the Minesweeper process, simply follow the instructions in the console window that appears.

Electrical Engineering and Computer Science
2017