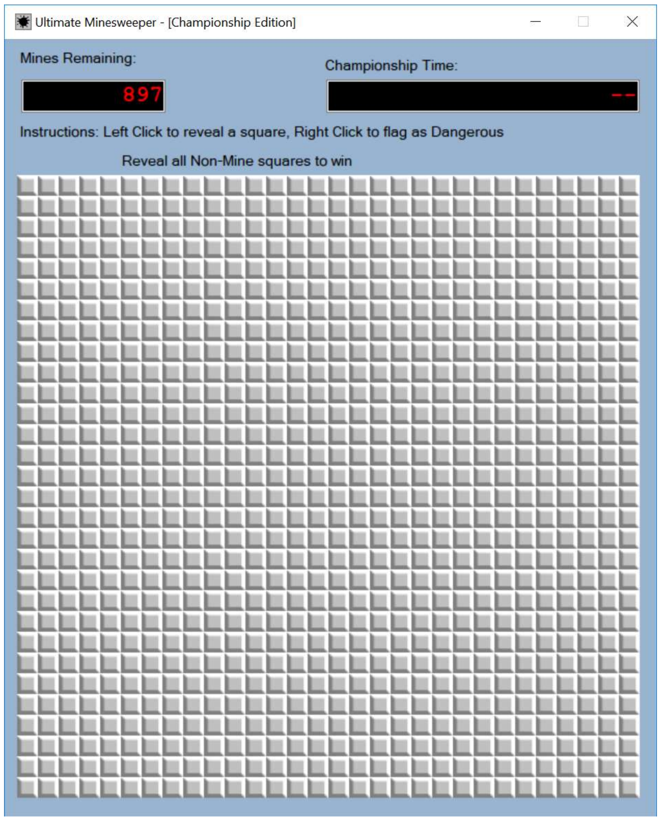
Flare-On 5: Challenge 2 Solution - UltimateMinesweeper.exe

Challenge Author: Nick Harbour

Description

You are presented with a Minesweeper game board as shown in Figure 1 below. It is a 30x30 grid containing 900 total squares. The objective of minesweeper is to left click on every non-mine square. You can right-click on a square to mark it as a mine. As you can see from the "Mines Remaining" counter, there are total of 897 mines, leaving only 3 non-mine squares.



Instructions: Left Click to reveal a square, Right Click to flag as Dangerous

Reveal all Non-Mine squares to win

Figure 1: Game Board

You could attempt to solve this challenge by systematically clicking each square, but it would be extremely frustrating and time-consuming. When an incorrect square is clicked, the player is presented with the following dialog and the game exits.

Figure 2: Fail Dialog

If you left-click on the three empty squares you will instead be presented with a success dialog screen. Determining exactly which three squares (they are the same each time) will require reverse engineering the binary.

Solution

The tool of choice for most .NET reverse engineering tasks is dnSpy. This tool allows decompilation, debugging and code editing. Loading this program into dnSpy you will see that there are only a few classes to examine: the `MainForm` class which represents the base GUI application; a class named `Program` which is standard in .NET binaries and is the starting point for execution; a `MineField` class which acts as a data model for our virtual mine field; a `MineFieldControl` class which contains the functionality for displaying and receiving input; classes named `SuccessPopup` and `FailurePopup` which are dialog windows that are displayed if the player won or lost the game.

Analysis of the `Program` class shows that it simply loads and displays the `MainForm`. The decompiled code is shown in Figure 3 below.

```
// UltimateMinesweeper.Program
// Token: 0x06000034 RID: 52 RVA: 0x00003219 File Offset: 0x0000141
9
[STAThread]
private static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new MainForm());
}
```

Figure 3: Program.Main()

Turning now to look at the `MainForm` class. The decompiled code of the initializer function shown in Figure 4 below. This function provides a good view of how the class functions, it creates a `MineField` object, calls the `AllocateMemory()` function with the minefield as a parameter, then sets a number of properties on the `MainForm` class and creates a `StopWatch` object and a List of revealed cells.

```
// Token: 0x06000009 RID: 9 RVA: 0x00002204 File Offset: 0x00000404
public MainForm()
{
    this.InitializeComponent();
    this.MineField = new MineField(MainForm.VALLOC_NODE_LIMIT);
    this.AllocateMemory(this.MineField);
    this.mineFieldControl.DataSource = this.MineField;
    this.mineFieldControl.SquareRevealed += this.SquareRevealedCall
back;
    this.mineFieldControl.FirstClick += this.FirstClickCallback;
    this.stopwatch = new Stopwatch();
    this.FlagsRemaining = this.MineField.TotalMines;
    this.mineFieldControl.MineFlagged += this.MineFlaggedCallback;
    this.RevealedCells = new List<uint>();
}
```

Figure 4: MainForm()

When a player clicks on a cell the `MineFieldControl` object calls the callback function supplied here named `SquareRevealedCallback`. The decompilation for this function is provided below in

```
// Token: 0x0600000C RID: 12 RVA: 0x00002348 File Offset: 0x0000054
private void SquareRevealedCallback(uint column, uint row)
{
    if (this.MineField.BombRevealed)
    {
        this.stopwatch.Stop();
        Application.DoEvents();
        Thread.Sleep(1000);
        new FailurePopup().ShowDialog();
        Application.Exit();
    }
    this.RevealedCells.Add(row * MainForm.VALLOC_NODE_LIMIT + colum
n);
    if (this.MineField.TotalUnrevealedEmptySquares == 0)
    {
        this.stopwatch.Stop();
        Application.DoEvents();
        Thread.Sleep(1000);
        new SuccessPopup(this.GetKey(this.RevealedCells)).ShowDialog
();
        Application.Exit();
    }
}
```

Figure 5.

Figure 5: SquareRevealedCallback

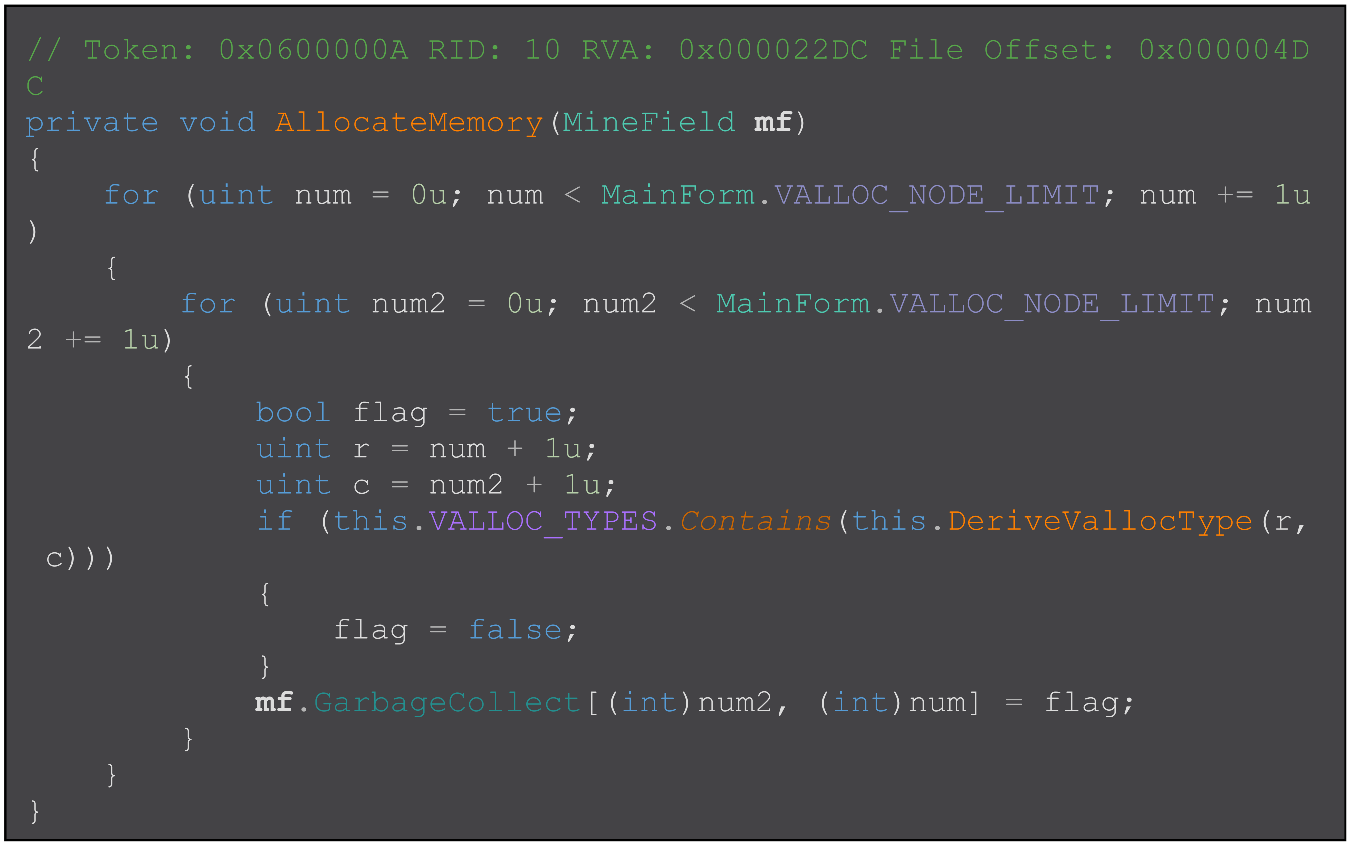
We can see that this callback function checks the value of a Boolean property within the `MineField` object named `BombRevealed` that will be true if any of the revealed squares were a bomb. In practicality, there could only be one revealed bomb as the game would end as soon as the first bomb is revealed. You may be tempted to just modify the code at this point to always show the success dialog or skip the `BombRevealed` check, but as we will see later on this would lead you to not getting the correct key. The `BombRevealed` property decompilation is provided in Figure 6 below.

```
// Token: 0x17000009 RID: 9
// (get) Token: 0x0600001E RID: 30 RVA: 0x00002BBC File Offset: 0x0
0000DBC
public bool BombRevealed
{
    get
    {
        int num = 0;
        while ((long)num < (long)((ulong)this.Size))
        {
            int num2 = 0;
            while ((long)num2 < (long)((ulong)this.Size))
            {
                if (this.MinesPresent[num2, num] && this.MinesVisib
le[num2, num])
                {
                    return true;
                }
                num2++;
            }
            num++;
        }
        return false;
    }
}
```

Figure 6: BombRevealed Property

It would be possible at this point to inspect the `MinesPresent` array at runtime to determine which values in the two-dimensional array were set to false. You could inject code using dnSpy to iterate over the array and notify you of the free cells with console output or a message box.

The program uses some stealthy function and variable naming to obscure the initial cell population function and list of free cells. The `AllocateMemory()` function we saw in the `MainForm` initializer in Figure 4 is actually the function responsible for populating the `MineField` with true and false values depending on if a mine is in each cell, it does not allocate memory.



```
// Token: 0x0600000A RID: 10 RVA: 0x000022DC File Offset: 0x000004D
C
private void AllocateMemory(MineField mf)
{
    for (uint num = 0u; num < MainForm.VALLOC_NODE_LIMIT; num += 1u
)
    {
        for (uint num2 = 0u; num2 < MainForm.VALLOC_NODE_LIMIT; num
2 += 1u)
        {
            bool flag = true;
            uint r = num + 1u;
            uint c = num2 + 1u;
            if (this.VALLOC_TYPES.Contains(this.DeriveVallocType(r,
 c)))
            {
                flag = false;
            }
            mf.GarbageCollect[(int)num2, (int)num] = flag;
        }
    }
}
```

Figure 7: AllocateMemory()

We can see here that this function iterates twice over a value named `VALLOC_NODE_LIMIT`, which you can see in dnSpy is just the number 30 (remember that our minefield is 30x30). Each time through the loop it will set a bool value in an array named `GarbageCollect`, within the `MineField` object, at the current cell coordinates to true unless the list object named `VALLOC_TYPES` contains the value returned by the function `DeriveVallocType()`, which is given the current coordinates to compute. The `GarbageCollect` array is property that simply obfuscates access to the `MineField` array. The `VALLOC_TYPES` array and related constants are shown in the decompilation fragment in

Figure 8 below.

```
// Token: 0x04000007 RID: 7
private static uint VALLOC_TYPE_HEADER_PAGE = 4294966400u;

// Token: 0x04000008 RID: 8
private static uint VALLOC_TYPE_HEADER_POOL = 4294966657u;

// Token: 0x04000009 RID: 9
private static uint VALLOC_TYPE_HEADER_RESERVED = 4294967026u;

// Token: 0x0400000A RID: 10
private uint[] VALLOC_TYPES = new uint[]
{
    MainForm.VALLOC_TYPE_HEADER_PAGE,
    MainForm.VALLOC_TYPE_HEADER_POOL,
    MainForm.VALLOC_TYPE_HEADER_RESERVED
};
```

Figure 8: VALLOC_TYPES Array

The `VALLOC_TYPES` array contains three values that are named constants defined in the code above the array. These three values are shown in dnSpy as decimal values but correspond to the hex values `0xFFFFFC80`, `0xFFFFFD81`, and `0xFFFFFEF2`. In order for the `AllocateMemory()` function to set a given cell to be clear of a mine, the value computed by the `DeriveVallocType()` function must match one of these three values. Let's now examine the `DeriveVallocType()` function as shown in Figure 9 below.

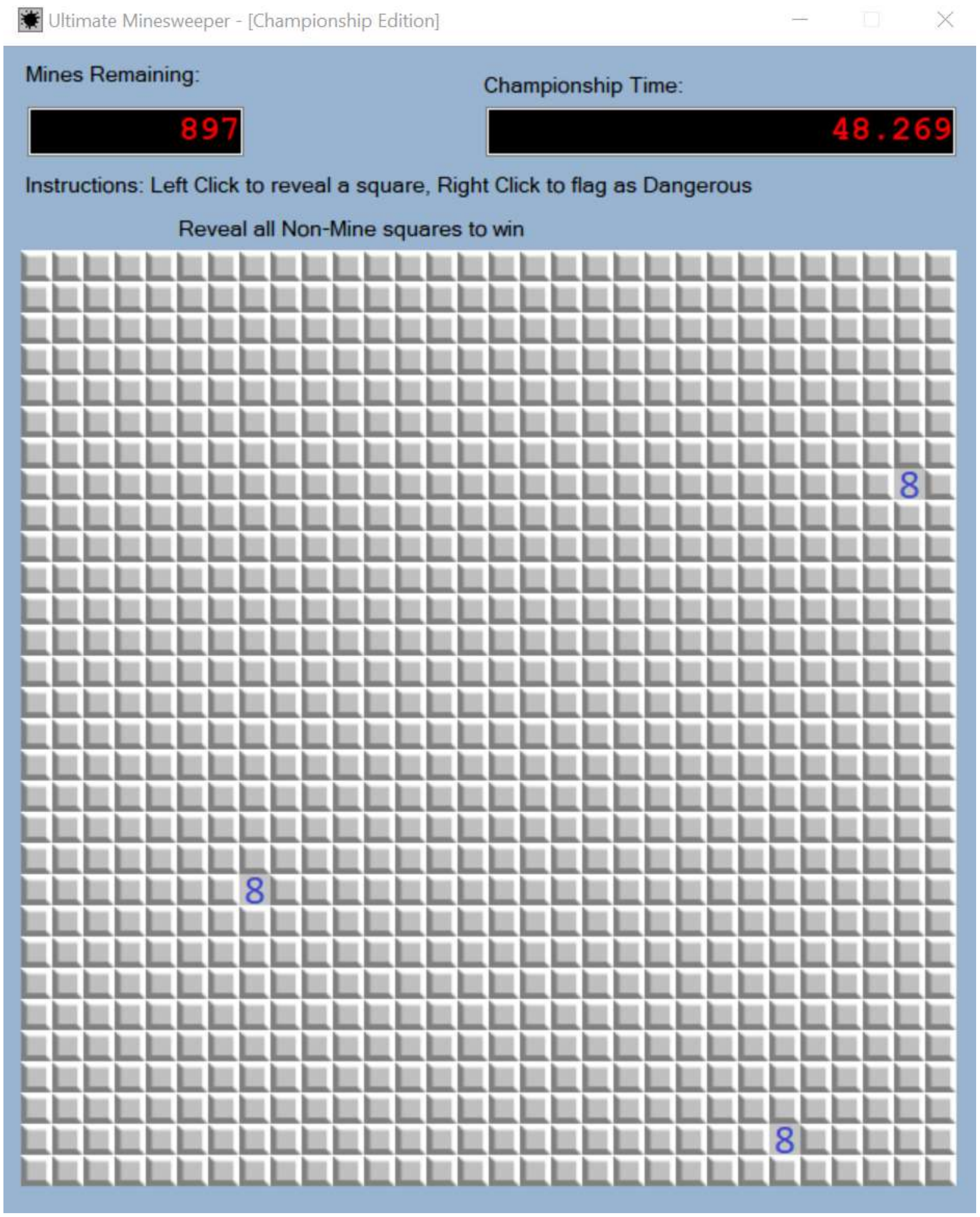
```
// Token: 0x0600000B RID: 11 RVA: 0x0000233A File Offset: 0x0000053
A
private uint DeriveVallocType(uint r, uint c)
{
    return ~(r * MainForm.VALLOC_NODE_LIMIT + c);
}
```

Figure 9: DeriveVallocType()

This function takes the row and column values (r and c) and forms them into an absolute cell index value by multiplying the row by the number of cells per row (`VALLOC_NODE_LIMIT`, 30) and then adding the column number to it. This function then returns the bitwise inverse of that value. The bitwise inverses of the three `VALLOC_TYPE` values described before are as follows:

- ~(0xFFFFFD81) = 0x27E = 638 = (21 * 30) + 8 = Row 21 Column 8
- ~(0xFFFFFC80) = 0x37F = 895 = (29 * 30) + 25 = Row 29 Column 25
- ~(0xFFFFFEF2) = 0x10D = 269 = (8 * 30) + 29 = Row 8 Column 29

You may have noticed that the `SuccessDialog` constructor takes a string argument. This argument is a key that will be displayed as the actual flag for the challenge. The key is derived from the `GetKey()` function which takes as input the list of revealed cells. The construction of this function does not need to be examined here but its values are set such that only a list of revealed cell values that exactly matches the above listed cells will properly produce a key value in the proper format (ending with “`@flare-on.com`”). If you had reverse engineered the binary to figure out which were the empty cells and revealed all three empty cells then this function would produce the key properly for it to be displayed in the success screen. If you modified the code or the minefield and got to the success screen without exactly these three cells then the value in your key field would be incorrect and most likely illegible garbage.


Ultimate Minesweeper - [Championship Edition]

Mines Remaining:

897

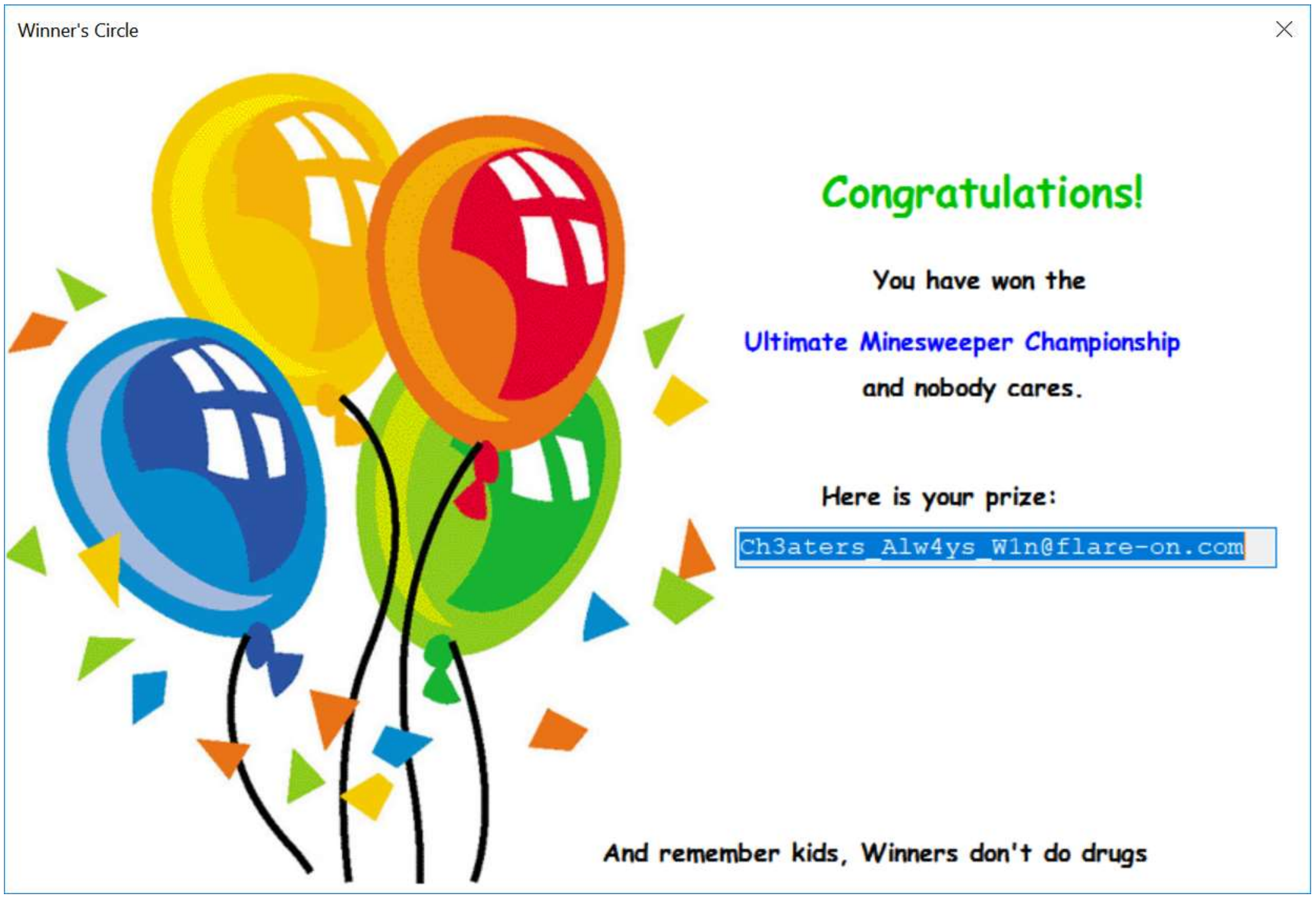
Championship Time:

48.269

Instructions: Left Click to reveal a square, Right Click to flag as Dangerous

Reveal all Non-Mine squares to win

Figure 10: Final Winning Minefield



Winner's Circle

Congratulations!

You have won the

Ultimate Minesweeper Championship

and nobody cares.

Here is your prize:

```
Ch3aters_Alw4ys_W1n@flare-on.com
```

And remember kids, Winners don't do drugs

Figure 11: Victory Screen and Final Key