Game Hacking: WinXP Minesweeper

dtm
Law Abiding Citizen

27d

Jan 17

Jan 17

What's good, peeps? Before I start this topic, I'd like to sincerely apologise for the *huge* delay. At the time (before my exams started), I was hyped to deliver a second instalment of game hacking to the forums but I had lost interest due to the exhaustion gathered from exam preparation and participation. Let's finally get this topic started.

Welcome to my second topic on game hacking where I will be discussing how to hack Windows XP's Minesweeper. The Windows XP version I will be using is SP3 which (starting with SP2) includes **DEP**. On the contrary, Windows XP does not support **ASLR** which was only introduced in Vista. The addresses I will be using will be hardcoded so keep in mind that this may not work in environments where ASLR is enabled. In those situations, dynamically retrieving the process's module base is a necessity but I will not be covering in this paper, but will do so if I ever present another game hacking write up.

Pre-requisites and Recommendations

For those who are seeking an introduction to game hacking and wish to follow through and understand the paper, it is recommended that they be familiar with the assembly language (preferably x86 Intel) and a high level language which is able to access and manipulate memory such as C or C++. Unlike the first game hacking topic, this will include source code (written in MSVC++) to apply the hacks on demand so knowledge of the **WinAPI** is also recommended.

For everyone else, all that's required is some time to dedicate and a curious mind.

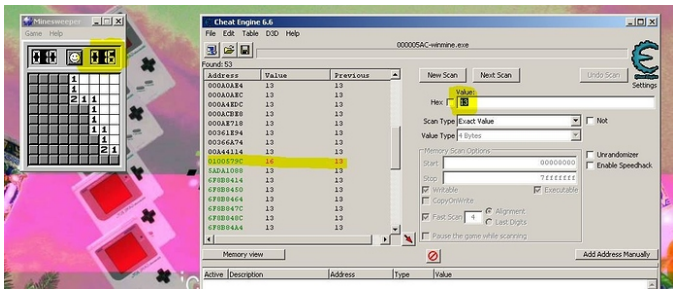
24d ago

Freezing the Timer

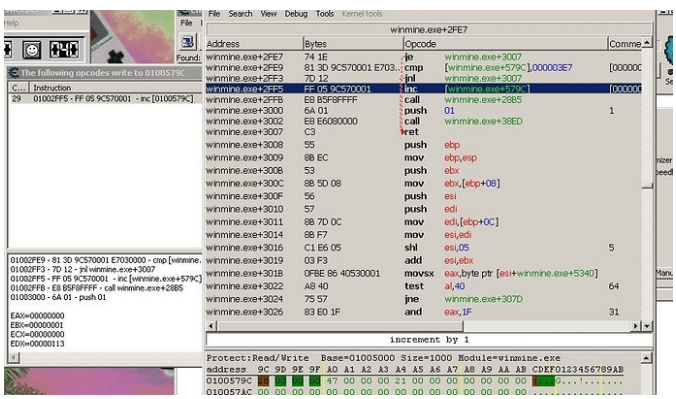
Debugging and Disassembly: Locating the Time

For those who are familiar with Minesweeper, the high scores are determined by the *fastest* sweep time so, clearly the timer is definitely something we want to target. First of all, we need to fire up our trusty Cheat Engine (CE) (mine is version 6.6 but that shouldn't matter) and Minesweeper. In CE, go to `File -> Open Process`, select the `winmine.exe` process from the list and press `Open`.

Now that Minesweeper is open and is loaded into CE, to be able to modify the time(r), we must locate where the value exists in memory. To do this, we have to let the timer run, then using CE, we attempt to find the value of the time by scanning the Minesweeper's process's memory. Start the game by pressing one of the cells and let it run until the time of, say `13` and while we wait, switch to CE and under `First Scan`, enter `13` into the edit box and prepare to press `First Scan` when the game's timer hits `13`. After first scanning, a list of address and their values will flood the left panel with the value of `13`, these are all of the existing values of `13` (of 4 bytes) in the process at the moment of scanning. If we scroll down a bit while the timer continues, we can see one of the addresses with red text.



The `Value` value is the same as the timer with a `Previous` value of what we first scanned. And still, while the timer on Minesweeper keeps increasing, so does the `Value` of address `0x0100579C`. We don't see any other addresses with the same property so we can safely assume that this is the address of where the time is stored. If we want to modify this time value, we have to open it up for debugging. Right-click the line and select `Find out what writes to this address`, agree to open the debugger and we will see one instruction `inc [0x0100579C]` in a new window. Select the line and click `Show disassembler` and we'll see the instructions at (and around) the address of where it is located in memory (`0x01002FF5`) in another new window. At the bottom, we can see the value of the time in a memory dump at `0x0100579C`.



If we want to freeze the timer, we can right-click the highlighted line and select `Replace with code that does nothing`, AKA noping the instruction. At this point, the timer in the Minesweeper window will immediately freeze. If we wanted to restore the instruction, right-click the first `nop` and select `Restore with original code` and the timer will start incrementing again. From here, we can do whatever we wanted. We can set it to decrement the value by right-clicking,

selecting `Assemble` and replacing `inc` with `dec` or change it directly in the memory dump to `42`, `69`, `420`, or whatever other significant integer in human history that lies between `0` and `999` inclusive.

Coding the Hack

Before we can implement the hack, we need to be able to write to the process's memory. To write to a process, its handle must be obtained to enable **`WriteProcessMemory`** but there are a few different ways of doing this. The method I chose retrieves the process handle through finding the window's handle which seemed to be a bit more simpler and appropriate so I opted for that instead of the standard process list enumeration.

```
HANDLE GetWindowProcessHandle(LPWSTR lpWindowName, DWORD dwDesiredAccess) {
    DWORD dwProcessId = 0;
    HWND hFindWnd = FindWindow(NULL, lpWindowName);
    if (hFindWnd == NULL)
        return NULL;
    GetWindowThreadProcessId(hFindWnd, &dwProcessId);
    return OpenProcess(dwDesiredAccess, FALSE, dwProcessId);
}
```

Now that we can get the process's handle, we can write to it using `WriteProcessMemory`:

```
BOOL WriteMemory(LPVOID lpBaseAddress, LPVOID lpBuffer, SIZE_T nSize) {
    HANDLE hWinMineProc = GetWindowProcessHandle(L"Minesweeper", PROCESS_VM_WRITE | PROCESS_VM_READ | PROCESS_VM_OPERATION);
    if (hWinMineProc == NULL)
        return Error(L"Find Minesweeper window"), FALSE;

    DWORD dwWritten = 0;
    BOOL bRet = WriteProcessMemory(hWinMineProc, lpBaseAddress, lpBuffer, nSize, &dwWritten);

    CloseHandle(hWinMineProc);

    return bRet;
}
```

We have everything we need to write our time-freezing function which should be pretty simple. All we have to do is overwrite the instruction at `0x01002FF5` with nops.

```
#define ADDR_TIMER 0x1002FF5

__declspec(naked) VOID FreezeTimerShellcodeStart(VOID) {
    __asm {
        nop
        nop
        nop
        nop
        nop
        nop
    }
}

BOOL FreezeTimer(BOOL bFreeze) {
    DWORD dwShellcodeSize = 6;
    LPBYTE lpShellcode = malloc(dwShellcodeSize);
    if (lpShellcode == NULL)
        return FALSE;

    if (bFreeze == TRUE)
        memcpy(lpShellcode, FreezeTimerShellCodeStart, dwShellcodeSize);
    else
        memcpy(lpShellcode, UnfreezeTimerShellcodeStart, dwShellcodeSize);

    BOOL bSuccess = WriteMemory((LPVOID)ADDR_TIMER, lpShellcode, dwShellcodeSize);
    if (bSuccess == FALSE)
        Error(L"Inject timer shellcode");

    free(lpShellcode);

    return bSuccess;
}
```

If we want to restore the instruction, all we have to do is rewrite the original opcodes back into the same memory address:

```
#define db(x) __emit(x)

__declspec(naked) VOID UnfreezeTimerShellcodeStart(VOID) {
    // MSVC++ producing incorrect opcodes
    // so I hardcoded the original bytes
    __asm {
        db(0xFF)
        db(0x05)
        db(0x9C)
        db(0x57)
```

```
            db(0x00)
            db(0x01)
        }
    }
```

Exposing the Mine Field

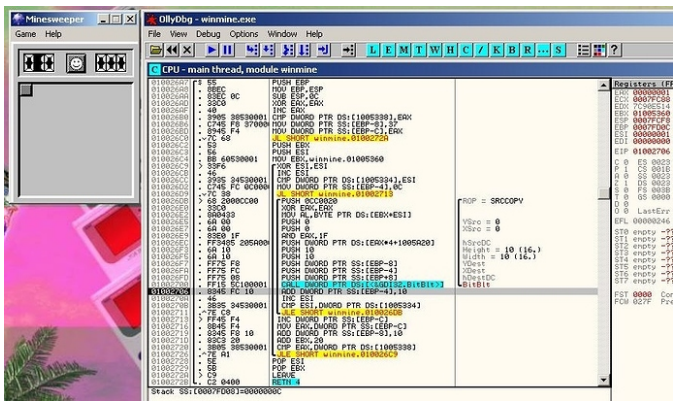
Debugging and Disassembly: Locating the Mine Field

The main goal of this game is to utilize hints to seek out and seek out all of the *safe* cells on the given field. While these hints are great and all that, let's make our *own* hints. This one requires a little bit more effort to trace and locate since it is not as straight-forward as the previous hack and needs a little knowledge of the WinAPI.

To start things off, CE might not be able to help us out in this situation so we need a proper debugger for this job. I'll be using Ollydbg but any debugger should be fine. Open up the `winmine.exe` in the debugger, place a breakpoint at the entry point, run it until it hits the breakpoint. When an application wants to draw and use graphics, it must use the `User32.dll` along with the **`Windows GDI (GDI32.dll)`** both which export functions to enable such tasks. One of these functions is **`BeginPaint`** which sets up a particular window for painting. We know that Minesweeper has to draw something so let's search for the function in the imports list (`Ctrl+N` on Ollydbg) and find references to `BeginPaint` (right-click and select `Find references`). Select the line and set a breakpoint on it so the debugger will stop when the game starts to lay out its graphical components.

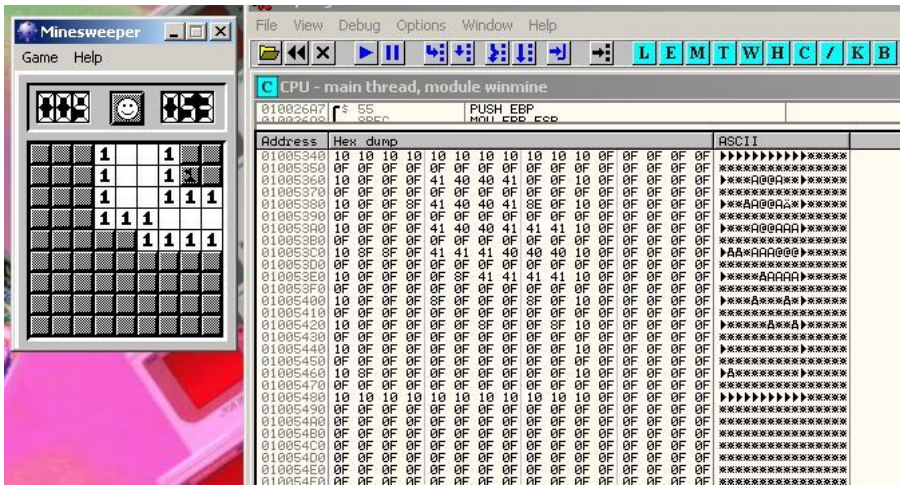
Run the program and the debugger will pause with the Minesweeper showing a blank window. Now, we should follow a typical reverse engineering approach to swiftly find the set of procedures which draw out the cells. Carefully begin to step *over* each instruction until we see something occur on the application, in our case, when we see the cells start to or have already been drawn out, stop stepping, place a breakpoint on the function call which produced the outcome, remove the previous breakpoint and restart the application, this time, stepping *into* the function with the new breakpoint. Repeat this process until we pinpoint the location of what we want.

Here is the result:



The **`BitBlt`** function is what draws out each cell of the mine field. If we analyze this set of instructions, we will see that it looks like some form of while loop where `esi` is a counter starting at `0` and `ebx` is the array beginning at `0x1005340`. It also seems like the cell's source image (a blank cell, a flag, a mine, etx.) is dependent on the value at `ds:[ebx+esi]`.

To satisfy our hypotheses, we will continue running the game normally and this time, we will dump the memory starting at `0x1005340` and see what it looks like.



Clearly, there is a very similar layout in the game and in the memory dump. We can match the `1`s with the `41`s and the flag with the `8E`. There's a bit of an issue with correctly formatting width the dump because we can slightly recognize that the border of the mine field is represented by the `10`s and the empty/unused cells are `0F`s.

Coding the Hack

In the previous hack, we learned how to write to the process, but this time, we need to read from the process in order to get the information about the array. There is a reading counterpart called **`ReadProcessMemory`** that we can use for this job and it is used exactly the same way we used `WriteProcessMemory`.

```
BOOL ReadMemory(LPCVOID lpBaseAddress, LPVOID lpBuffer, SIZE_T nSize) {
    HANDLE hWinMineProc = GetWindowProcessHandle(L"Minesweeper", PROCESS_VM_READ);
    if (hWinMineProc == NULL)
        return Error(L"Find Minesweeper window"), FALSE;

    DWORD dwRead = 0;
    BOOL bRet = ReadProcessMemory(hWinMineProc, lpBaseAddress, lpBuffer, nSize, &dwRead);

    CloseHandle(hWinMineProc);

    return bRet;
}
```

It is now just a simple matter of looping and reading the memory of Minesweeper and printing it out with some nice visual formatting. Keep in mind that whenever a cell is activated, its value in memory will change so the loop must be continuous to update these values.

```
#define CELL_EMPTY L" "
#define CELL_NOT_MINE L"o"
#define CELL_MINE L"x"
#define CELL_FLAG L"!"
#define CELL_UNKNOWN L"?"
#define CELL_BORDER L"*"

// console colors
#define CELL_DIGIT_COLOUR FOREGROUND_GREEN | FOREGROUND_INTENSITY
#define CELL_NOT_MINE_COLOUR FOREGROUND_GREEN
#define CELL_MINE_COLOUR FOREGROUND_RED | FOREGROUND_INTENSITY
#define CELL_FLAG_COLOUR FOREGROUND_GREEN | FOREGROUND_RED | FOREGROUND_INTENSITY
#define CELL_UNKNOWN_COLOUR FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_INTENSITY
#define CELL_BORDER_COLOUR FOREGROUND_BLUE | FOREGROUND_GREEN | FOREGROUND_RED | FOREGROUND_INTENSITY

VOID PrintDigit(int nDigit) {
    CONSOLE_SCREEN_BUFFER_INFO csbi;
    GetConsoleScreenBufferInfo(GetStdHandle(STD_OUTPUT_HANDLE), &csbi);
    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), CELL_DIGIT_COLOUR);

    wprintf(L"%d", nDigit);

    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), csbi.wAttributes);
}

VOID PrintCell(LPWSTR c, WORD wAttributes) {
    CONSOLE_SCREEN_BUFFER_INFO csbi;
    GetConsoleScreenBufferInfo(GetStdHandle(STD_OUTPUT_HANDLE), &csbi);
    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), wAttributes);

    wprintf(L"%s", c);

    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), csbi.wAttributes);
}

VOID PrintMineField(VOID) {
    BYTE bCell = 0;
    DWORD dwMineFieldSize = ADDR_MINE_FIELD_END - ADDR_MINE_FIELD_START;
    LPBYTE lpMineFieldBuffer = malloc(dwMineFieldSize);
    if (lpMineFieldBuffer == NULL)
        return;

    // I have opted to threading this function
    // since my hacktool includes a GUI
    while (bThreadActive == TRUE) {
        BOOL bSuccess = ReadMemory((LPVOID)ADDR_MINE_FIELD_START, lpMineFieldBuffer, dwMineFieldSize);
        if (bSuccess == FALSE) {
            // ignore this, this is also part of the GUI
            SendMessage(hWnd, WM_COMMAND, ID_HIDE_MINES, 0);
            return;
        }

        for (DWORD j = 0, i = 0, bCell = lpMineFieldBuffer[i]; i < dwMineFieldSize; i++, j++, bCell = lpMineFieldBuffer[i]) {
            // new line on the mine field
            if (j == 0x20) {
                wprintf(L"\n");
                j = 0;
            }

            // these are most (if not all) of the values of the cells
            int nDigit = bCell & 0x1F;
            if (bCell == 0x8F || bCell == 0x8D)
                PrintCell(CELL_MINE, CELL_MINE_COLOUR);
            else if (bCell == 0x10)
                PrintCell(CELL_BORDER, CELL_BORDER_COLOUR);
            else if (bCell == 0x8E || bCell == 0x0E)
```

```
                PrintCell(CELL_FLAG, CELL_FLAG_COLOUR);
            else if (bCell == 0x0D)
                PrintCell(CELL_UNKNOWN, CELL_UNKNOWN_COLOUR);
            else if (nDigit == 0x0)
                PrintCell(CELL_EMPTY, CELL_NOT_MINE_COLOUR);
            else if (nDigit >= 0x1 && nDigit <= 0x9)
                PrintDigit(nDigit);
            else
                PrintCell(CELL_NOT_MINE, CELL_NOT_MINE_COLOUR);
        }

        Sleep(1000);
        ////////////////////////////////
        ///////    YUCK!!!!    ////////
        ////////////////////////////////
        system("cls");
    }

    free(lpMineFieldBuffer);
}
```

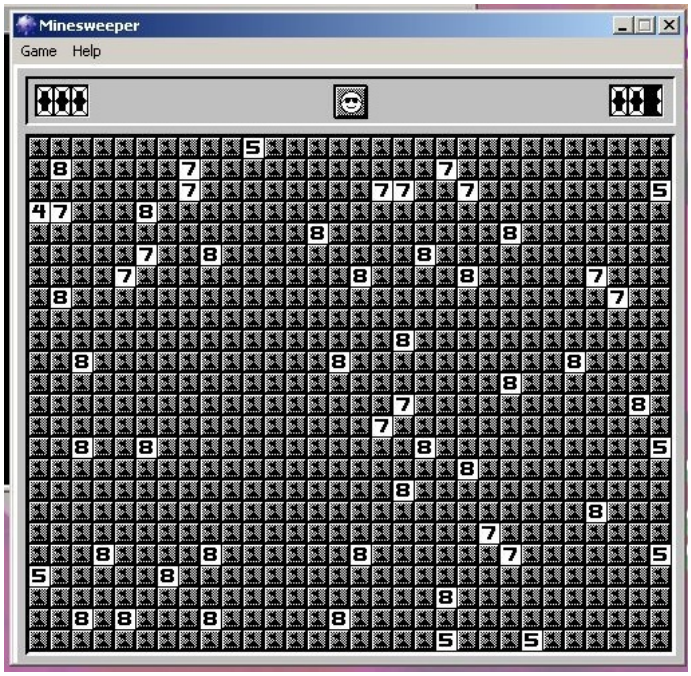
Here is an example of what it should look like:

Challenge

I have provided the basic fundamentals on how to hack a game such as memory scanning with CE and reading from and writing to an external process so for those who want in, see if you can mess around with the number of flags and implement it into your own hacktool. Be creative.

Conclusion

From what we've seen, game hacking is on par with reverse engineering. In fact, I think of it as a subcategory of reverse engineering as it exercises the same skillset and mindset (also see **previous game hacking topic**). In contrast, game hacking is probably more fun than just the typical reverse engineering crackme because we get to physically play around with our hacks even *after* successfully breaking open the innards. It is truly an amusing and satisfying feeling.



#nohacks #legit #minesweepergod

Complete source code on **GitLab** (includes spoilers to the challenge!).

-- dtm

Luigi **26d**

You sir will destroy my childhood hahaha
Waiting for the full topic

SmartOne **21d**
Not a N00b, but still learning

@dtm When will this be out? Waiting for it 😛

---

dtm **21d**
Law Abiding Citizen

It'll be out in this or next week.

---

dtm **24d**
Law Abiding Citizen

Bump. Updated. WOOP.

Reply