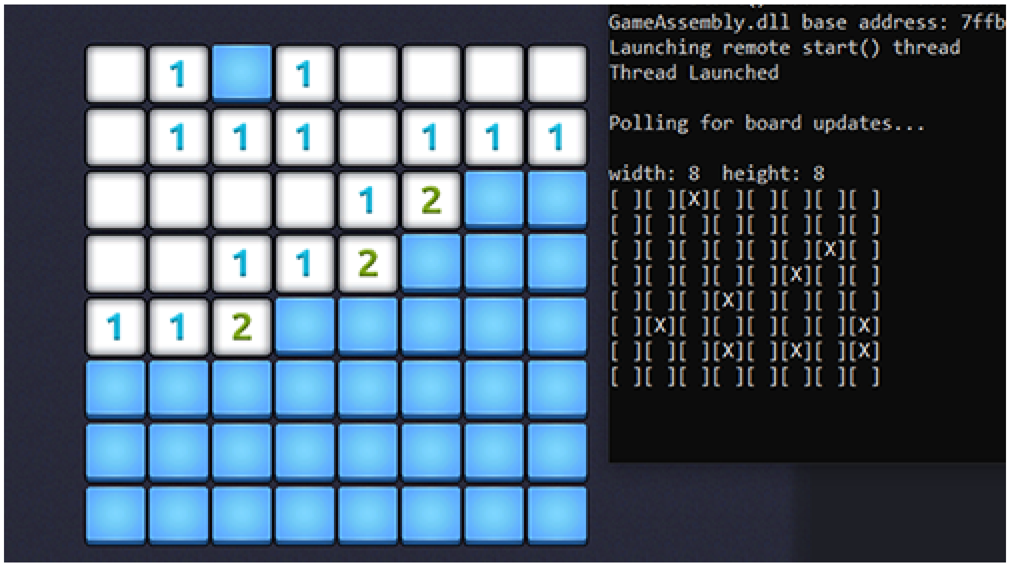
Minesweeper x64 Hacking

Nathan Smith
https://github.com/charlesnathansmith/minehack



```
GameAssembly.dll base address: 7ffb
Launching remote start() thread
Thread Launched

Polling for board updates...

width: 8  height: 8
[ ][ ][X][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ][ ][X][ ]
[ ][ ][ ][ ][ ][X][ ][ ]
[ ][ ][ ][X][ ][ ][ ][ ]
[ ][X][ ][ ][ ][ ][ ][X]
[ ][ ][ ][X][ ][X][ ][X]
[ ][ ][ ][ ][ ][ ][ ][ ]
```

Reverse engineering, DLL injection, and code hooking in a Windows App

There are countless articles and videos focused on reverse engineering and modifying the classic Windows XP-era Minesweeper game. While I found a few interesting stabs at the 64-bit version such as [1] and [2], I couldn't find anything that went into much detail about how to approach something like this and some of the issues that come up when reversing and hooking Windows Apps.

We'll look at what goes into reverse engineering the 64-bit version of Minesweeper to acquire the board data, and write a working hook and DLL injector that works around some of the issues that arise when manipulating Windows Apps.

Code for the final hook DLL and injector are available at [3].

Getting the Game

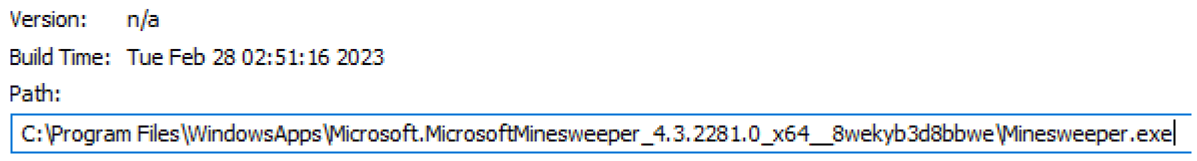
Minesweeper on XP was a simple game that came bundled with the OS and clocked in at around 120 kB. The 64-bit version requires a 120+ MB download from the App store and comes buried in banner and interstitial ads. Ignoring those, it's a lot more visually appealing than the original, and it introduces Challenge and Adventure modes, which are fun twists on the original Classic.

We're only going to focus on Classic mode here, but you should be able to figure out how to extend what we do here to modifying the other modes. And we aren't going anywhere near messing with the ads. Microsoft owns Github now and I'd really rather not have my account suspended.

Look for Microsoft Minesweeper in the Windows Store, and it should be the first thing that comes up.

Where Did It Go?

If you launch Minesweeper then go to its properties in Task Manager to try to find the executable, the way the location is given is a bit confusing, to me at least. The easiest thing to do is open up Process Explorer or attach a debugger to it and get the full path that way. The path should look something like this, though probably not exactly the same on your system:



```
Version:     n/a
Build Time:  Tue Feb 28 02:51:16 2023
Path:
C:\Program Files\WindowsApps\Microsoft.MicrosoftMinesweeper_4.3.2281.0_x64__8wekyb3d8bbwe\Minesweeper.exe
```

Go ahead and make a new folder, open up the exe's folder and copy everything out of there. Windows Apps folders have exotic restrictive permissions that are difficult to restore if you mess them up and have to be set exactly right or apps will refuse to run (ask me how I know.) We'll hook and do our debugging on a running instance of the original later, but during static analysis I found it easier just to make a copy somewhere rather than changing IDA's configuration to create it's databases elsewhere.

Il2CppDumper

Minesweeper (and many games like it) was built on top of the Unity engine using C# for the game logic. It's put through a backend called Il2Cpp which effectively converts it into C++ and generates definition files for objects, functions, and strings, then is compiled into one massive library called *GameAssembly.dll*. I haven't really worked with Unity, so I'm not going to pretend to know a whole lot of specifics about the translation and packing process.

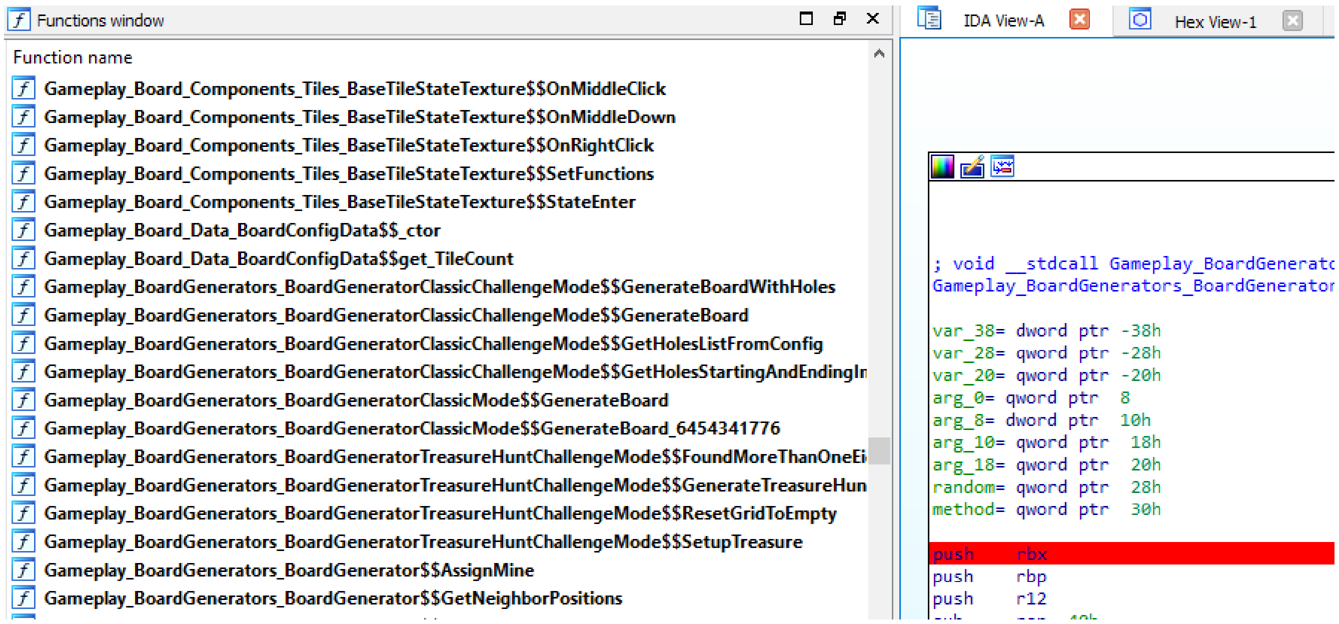
The definitions aren't in human readable form, but can be extracted using a tool called Il2CppDumper [4]. You just feed in the files it needs, then it produces several output files, the most important to us being *il2cpp.h* which contains structs and typedefs used in the program, and *script.json* which has function and string literal definitions and addresses. If you encounter strange errors when trying to run it, you probably need to install the .NET runtime library that matches the version of Il2CppDumper you downloaded.

There are scripts available in the repository to load the definitions into IDA, Ghidra, and other tools. I had some issues getting their IDA scripts to work in version 7.0+ and ended up writing my own version that is implemented as a plugin, gives progress information (loading all of the definitions can take quite a while,) and fixes some issues I was having with missing standard types. [5] But you might have more luck with one of theirs. Just try them out and see which one works for you. They don't do anything that complicated, but the APIs for parsing and applying types differ somewhat between IDA versions.

Everything we care about is in *GameAssembly.dll*. That's the file we will disassemble and then apply the script or plugin to.

Getting our Bearings

Once all of our definitions are loaded and autoanalysis completes, we can start looking around and see we now have a ton of really helpful symbol data (make sure autoanalysis completes before searching or sorting functions, or it will slow to a crawl continually "Caching function window..." – Just close the function window until analysis completes if that happens.)



```
Gameplay_Board_Components_Tiles_BaseTileStateTexture$$OnMiddleClick
Gameplay_Board_Components_Tiles_BaseTileStateTexture$$OnMiddleDown
Gameplay_Board_Components_Tiles_BaseTileStateTexture$$OnRightClick
Gameplay_Board_Components_Tiles_BaseTileStateTexture$$SetFunctions
Gameplay_Board_Components_Tiles_BaseTileStateTexture$$StateEnter
Gameplay_Board_Data_BoardConfigData$$_ctor
Gameplay_Board_Data_BoardConfigData$$get_TileCount
Gameplay_BoardGenerators_BoardGeneratorClassicChallengeMode$$GenerateBoardWithHoles
Gameplay_BoardGenerators_BoardGeneratorClassicChallengeMode$$GenerateBoard
Gameplay_BoardGenerators_BoardGeneratorClassicChallengeMode$$GetHolesListFromConfig
Gameplay_BoardGenerators_BoardGeneratorClassicChallengeMode$$GetHolesStartingAndEndingIn
Gameplay_BoardGenerators_BoardGeneratorClassicMode$$GenerateBoard
Gameplay_BoardGenerators_BoardGeneratorClassicMode$$GenerateBoard_6454341776
Gameplay_BoardGenerators_BoardGeneratorTreasureHuntChallengeMode$$FoundMoreThanOneEi
Gameplay_BoardGenerators_BoardGeneratorTreasureHuntChallengeMode$$GenerateTreasureHun
Gameplay_BoardGenerators_BoardGeneratorTreasureHuntChallengeMode$$ResetGridToEmpty
Gameplay_BoardGenerators_BoardGeneratorTreasureHuntChallengeMode$$SetupTreasure
Gameplay_BoardGenerators_BoardGenerator$$AssignMine
Gameplay_BoardGenerators_BoardGenerator$$GetNeighborPositions
```

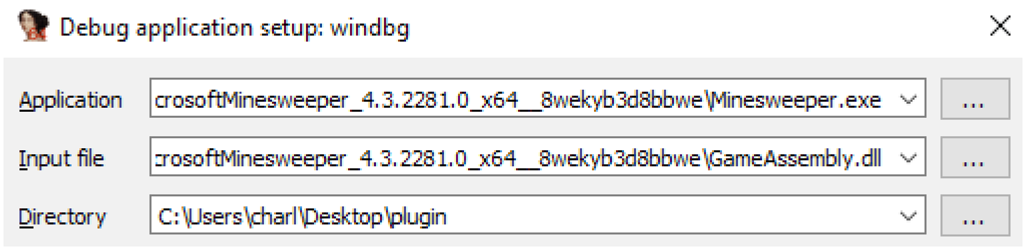
```
; void __stdcall Gameplay_BoardGenerat
Gameplay_BoardGenerators_BoardGenerator

var_38= dword ptr -38h
var_28= qword ptr -28h
var_20= qword ptr -20h
arg_0= qword ptr  8
arg_8= dword ptr  10h
arg_10= qword ptr  18h
arg_18= qword ptr  20h
random= qword ptr  28h
method= qword ptr  30h

push    rbx
push    rbp
push    r12
```

We can start searching for things like "board" or "mine" to find functions we might be interested in. A word of caution, though: different game modes use different board generators and mine functions. I'll walk through the ones relevant to Classic mode, then let you try to work out the others if you want, but the best thing to do is to set breakpoints at the beginning of functions you think might be involved then launch Minesweeper and attach to it and play around to see which ones are actually hit before wasting time studying the wrong ones.

Debugging

Debugging Windows Apps is a bit tricky. You can't just launch the exe, there's a whole loading framework it has to go through. The easiest thing to do is to just launch it from the Start Menu, then attach to it.

If you built your IDA database off of a copy like I did, you'll have to go to "Process Options..." under the Debugger menu, then point "Application" and "Input file" to the originals before trying to attach.



Debug application setup: windbg

Application	crosoftMinesweeper_4.3.2281.0_x64__8wekyb3d8bbwe\Minesweeper.exe
Input file	crosoftMinesweeper_4.3.2281.0_x64__8wekyb3d8bbwe\GameAssembly.dll
Directory	C:\Users\charl\Desktop\plugin

Using IDA to do the debugging would be ideal since it has all of our symbol infomation loaded in it, but I found it to be cumbersome in practice. It has to rebase everything in the database so it can follow along accurately, which isn't normally a problem, but *GameAssembly.dll* is just so massive it was painfully slow at least on my machine. The game engine also relies heavily on user-defined exceptions. I use Windbg as the debugging engine with IDA, and I don't know of a straightforward way to just blanket pass on all exceptions.

Maybe one of the other engines it can use would work better, but I just ended up doing the debugging with x64dbg attached to the running process in one window, and the static disassembly open in IDA in another beside it. It's nice having the big picture static view up in one window while stepping through line by line in the other, but it's not the best setup and easy to lose your place and you could probably do much better.

If you do end up using a separate debugger like that, then to find function addresses in the debugger, look at the address in IDA, subtract the base address to get the relative virtual address (RVA) then add that to the base address *GameAssembly.dll* is loaded at in the running process. Like I said, I can't really recommend it, but it was good enough at the time for no more than I needed to do with it. Label things as you find them in x64dbg and it will keep up with them during the next session.

Digging In

These function names and definitions are extremely long so bear with me on the awful formatting.

Classic mode generates a new board by calling:
`Gameplay_BoardGenerators_BoardGeneratorClassicMode$$GenerateBoard`

Which is defined as:

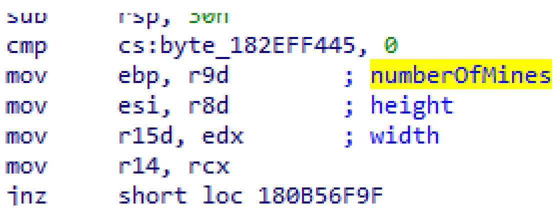
```
void __stdcall Gameplay_BoardGenerators_BoardGeneratorClassicMode__GenerateBoard
    (Gameplay_BoardGenerators_BoardGeneratorClassicMode_o* this,
    int32_t width, int32_t height, int32_t numberOfMines,
    System_Collections_Generic_List_TilePosition__o* mineExclusions,
    System_Action_ReadOnlyBoardDataClassicMode__o* onBoardGenerated,
    const MethodInfo* method)
```

This was found just by breakpointing on this and a couple of other functions that sounded like they were probably what we were looking for and seeing what was hit. It's nice having symbol information.

Interestingly, this actually isn't called right when the game starts, but after you choose the first tile to clear. It does this to ensure you don't hit a mine with your first click, and the tile you choose is passed in via **`mineExclusions`**.

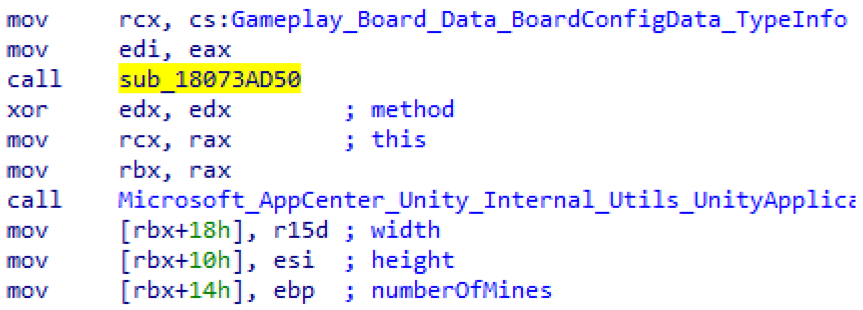
Just looking at the arguments, some things pop out, notably **`numberOfMines`**, which is going to have to get used by a mine setting routine somewhere, and **`width`** and **`height`** which we'll want to know to display the board data properly, but we'll come back to those because there's a more interesting way we can get those later.

Let's see where **`numberOfMines`** takes us.



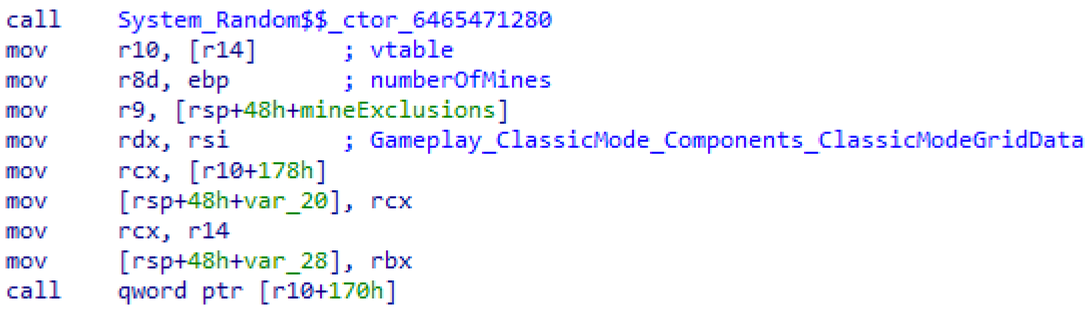
```
sub     rsp, 50h
cmp     cs:byte_182EFF445, 0
mov     ebp, r9d        ; numberOfMines
mov     esi, r8d        ; height
mov     r15d, edx       ; width
mov     r14, rcx
jnz     short loc_180B56F9F
```

`numberOfMines` is the fourth argument, and since this is a 64-bit application, that means it starts out in **`r9`**. In the first block of this function, it gets moved into **`ebp`** (the lower 32-bits at least, but we're not going to have billions of mines.) The **`*`**`this` that was passed in to this function is moved to **`r14`**.



```
mov     rcx, cs:Gameplay_Board_Data_BoardConfigData_TypeInfo
mov     edi, eax
call    sub_18073AD50
xor     edx, edx        ; method
mov     rcx, rax        ; this
mov     rbx, rax
call    Microsoft_AppCenter_Unity_Internal_Utils_UnityApplica
mov     [rbx+18h], r15d ; width
mov     [rbx+10h], esi  ; height
mov     [rbx+14h], ebp  ; numberOfMines
```

The next time we see **`ebp`**, it's being used along with the **`width`** and **`height`** to fill out what we can probably figure is a `BoardConfigData` object with the unnamed **`sub_18073AD50`** function being some sort of creator or accessor for it. We'll take a mental note of this, but then immediately find:



```
call    System_Random$$_ctor_6465471280
mov     r10, [r14]      ; vtable
mov     r8d, ebp        ; numberOfMines
mov     r9, [rsp+48h+mineExclusions]
mov     rdx, rsi        ; Gameplay_ClassicMode_Components_ClassicModeGridData
mov     rcx, [r10+178h]
mov     [rsp+48h+var_20], rcx
mov     rcx, r14
mov     [rsp+48h+var_28], rbx
call    qword ptr [r10+170h]
```

Now this looks promising.

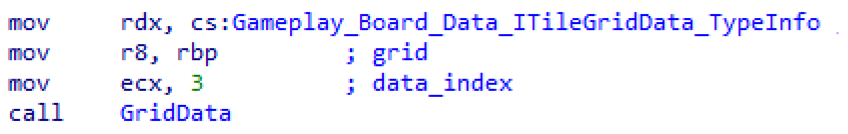
`r14` still holds our **`*`**`this`, so it looks like the head of its vtable is being loaded into **`r10`**, and the entry at **`r10`**`+170h` is getting called with lots of arguments that a mine layer would likely want.

We fire up the debugger to see where this call actually takes us, and we meet this guy:

```
Gameplay_BoardGenerators_BoardGenerator$$SetupMines
```

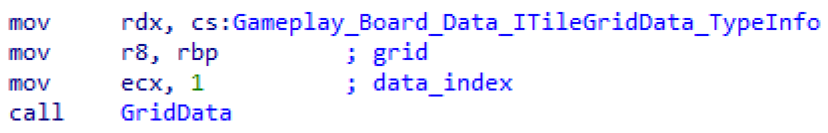
```
void __stdcall Gameplay_BoardGenerators_BoardGenerator__SetupMines
  (Gameplay_BoardGenerators_BoardGenerator_o* this,
   Gameplay_Board_Data_ITileGridData_o* grid,
   int32_t numberOfMines,
   System_Collections_Generic_List_TilePosition__o* exclusions,
   System_Random_o* random,
   const MethodInfo* method)
```

Carrying on this way, we'll skip to the interesting parts of this function.



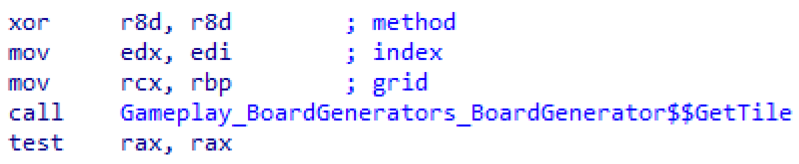
```
mov     rdx, cs:Gameplay_Board_Data_ITileGridData_TypeInfo
mov     r8, rbp         ; grid
mov     ecx, 3          ; data_index
call    GridData
```

This part calls some sort of generic attribute or member value getter to get the total number of tiles on the board (the name is mine, it was just another `SUB_xxxx` function.)



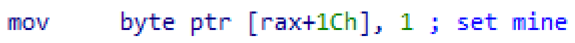
```
mov     rdx, cs:Gameplay_Board_Data_ITileGridData_TypeInfo
mov     r8, rbp         ; grid
mov     ecx, 1          ; data_index
call    GridData
```

This calls the same function with a different member index to get the number of colums.



```
xor     r8d, r8d        ; method
mov     edx, edi        ; index
mov     rcx, rbp        ; grid
call    Gameplay_BoardGenerators_BoardGenerator$$GetTile
test    rax, rax
```

This gets a pointer to an individual tile object by index (it actually already knew this information, but there are some weird artifacts throughout, likely introduced by the original Il2Cpp translation.)



```
mov     byte ptr [rax+1Ch], 1 ; set mine
```

And this is how a tile object is marked as a mine

The reason we care about being able to access the board size data this way is because the **`width`** and **`height`** arguments that were passed to **`GenerateBoard`** earlier don't get passed to **`SetupMines`** and we'd really like to avoid having to setup hooks in multiple places if we can avoid it.

Planning the Hook

We need our hook to get called after **`SetupMines`** so we can read the board it generates. Using the information gathered above, we can collect all of the information we need about the board just by passing **`grid`** to different functions. Ignoring error checking for now, the basic method to copy data to some **`board`** object that we come up with will look roughly like:

```
void __stdcall SetupMines_hook(void* grid)
{
    size_t num_tiles = GridData(3, *pTypeInfo, grid);

    board.width = GridData(1, *pTypeInfo, grid);
    board.height = num_tiles / board.width;

    for (size_t i = 0; i < num_tiles; i++)
        board.tiles[i] = ((uint8_t*)GetTile(grid, i, 0))[0x1c];
}
```

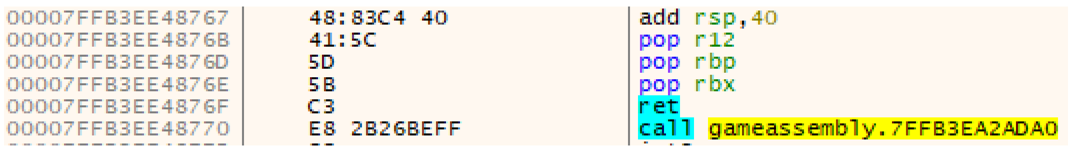
Where everything is at a known position in memory except for **`grid`**.

From here to there

Our hook function will reside within our injected DLL. The original **SetupMines** we want to hook the end of is all the way on the other side of the 64-bit world inside of *GameAssembly.dll*. We also need to pass **grid** to our hook in a correctly formatted call.

For a detailed look at the considerations involved when designing 64-bit hook trampolines, see [6].

We need an initial jumping off point that has access to all of the data we need.



```
00007FFB3EE48767 | 48:83C4 40        | add rsp,40
00007FFB3EE4876B | 41:5C             | pop r12
00007FFB3EE4876D | 5D                | pop rbp
00007FFB3EE4876E | 5B                | pop rbx
00007FFB3EE4876F | C3                | ret
00007FFB3EE48770 | E8 2B26BEFF       | call gameassembly.7FFB3EA2ADA0
```

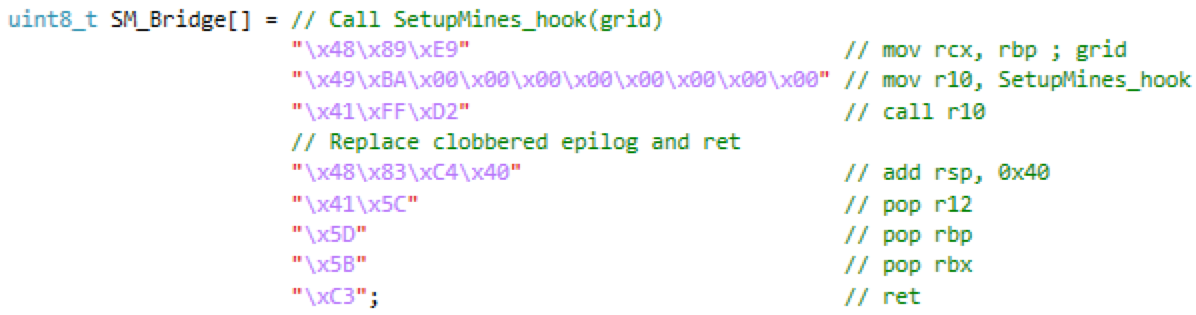
At the end of **SetupMines**, all of the mines have been set, and **grid** is still sitting inside of **rbp** until the `pop` instruction destroys it. We need to hook somewhere before that happens. Since we plan on calling into our code, it probably makes the most sense to jump before the `add rsp, 40h` instruction executes so we're not having to readjust the stack for parameter homing [7].

Some other code follows the `ret` instruction here, which means we only have 9 bytes we can alter. We'll have to find or allocate some useable space nearby that we can do a 32-bit relative jmp to, since that only takes up 5 bytes and is about all that will fit here:

```
E9 WW XX YY ZZ      jmp (start of next ins + 0xZZYYXXWW)
```

This will replace the `add rsp, 40h` instruction, as well as clobber the first byte of the `pop r12` instruction, but we can just complete the epilog and `ret` from our call trampoline when we're done.

Let's see what our bridge code is going to have to look like to make the call to `SetupMines_hook` and return properly:

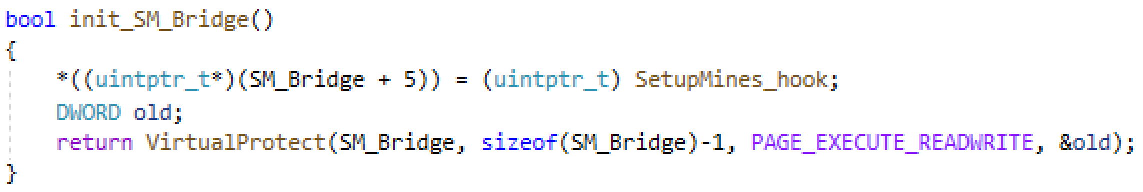


```
uint8_t SM_Bridge[] = // Call SetupMines_hook(grid)
                      "\x48\x89\xE9"                         // mov rcx, rbp ; grid
                      "\x49\xBA\x00\x00\x00\x00\x00\x00\x00\x00" // mov r10, SetupMines_hook
                      "\x41\xFF\xD2"                         // call r10
                      // Replace clobbered epilog and ret
                      "\x48\x83\xC4\x40"                     // add rsp, 0x40
                      "\x41\x5C"                             // pop r12
                      "\x5D"                                 // pop rbp
                      "\x5B"                                 // pop rbx
                      "\xC3";                                // ret
```

It just passes **grid** to the hook we wrote earlier, then finishes the epilog we're going to have to destroy and returns to whatever called the original **SetupMines** function.

We would like to just fill in the missing address for hook and paste this thing close enough for our relative near jmp to reach, but at 25 bytes, there isn't likely to be room for it in the address space near the function we're jumping out of.

We could use VirtualAlloc to try to allocate some memory near it, but we’re going to take a different approach.

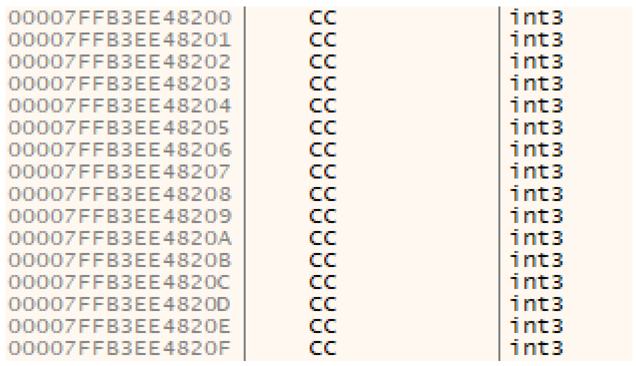


```
bool init_SM_Bridge()
{
    *((uintptr_t*)(SM_Bridge + 5)) = (uintptr_t) SetupMines_hook;
    DWORD old;
    return VirtualProtect(SM_Bridge, sizeof(SM_Bridge)-1, PAGE_EXECUTE_READWRITE, &old);
}
```

We fill in the address of **`SetupMines_hook`** that the bridge needs, then make it executable. Now if we can get from where we want to hook **`SetupMines_hook`** over to **`SM_Bridge`**, it will do what we want.

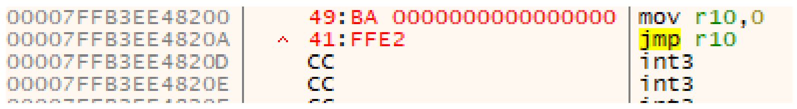
This solved the size problem, but it’s still too far away for a near `jmp` to reach. Thankfully, while there aren’t 25 bytes free near the end of **`SetupMines`**, there are more than 13 bytes free that I happened to notice back near **`GenerateBoard`**, which is enough room for a long `jmp` we can use to trampoline over to our **`SM_Bridge`**.

Here is the new space we found before:



```
00007FFB3EE48200 | CC | int3
00007FFB3EE48201 | CC | int3
00007FFB3EE48202 | CC | int3
00007FFB3EE48203 | CC | int3
00007FFB3EE48204 | CC | int3
00007FFB3EE48205 | CC | int3
00007FFB3EE48206 | CC | int3
00007FFB3EE48207 | CC | int3
00007FFB3EE48208 | CC | int3
00007FFB3EE48209 | CC | int3
00007FFB3EE4820A | CC | int3
00007FFB3EE4820B | CC | int3
00007FFB3EE4820C | CC | int3
00007FFB3EE4820D | CC | int3
00007FFB3EE4820E | CC | int3
00007FFB3EE4820F | CC | int3
```

And after:



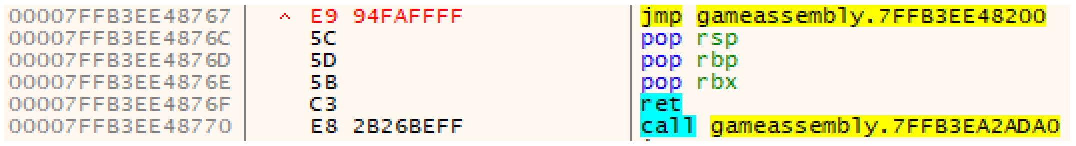
```
00007FFB3EE48200 |   49:BA 0000000000000000 | mov r10,0
00007FFB3EE4820A | ^ 41:FFE2                | jmp r10
00007FFB3EE4820D |   CC                     | int3
00007FFB3EE4820E |   CC                     | int3
```

Where the 00s will be replaced by the address of our **`SM_Bridge`** above.

Now everything is in place except for our near `jmp` out of the end of **`SetupMines`** over to the trampoline we just built.

Our trampoline is at `00007FFB3EE48200` in this example, and our near `jmp` is going to replace the `add rsp, 40h` instruction back at `00007FFB3EE48767`. The near `jmp` is 5 bytes long, so the instruction right after it will then start at `00007FFB3EE4876C`.

The relative offset that goes in our near `jmp` is the start of trampoline minus the start of the instruction after the `jmp`, or `00007FFB3EE48200 - 00007FFB3EE4876C = FFFFFA94` (truncated to 32 bits.)



```
00007FFB3EE48767 | ^ E9 94FAFFFF     | jmp gameassembly.7FFB3EE48200
00007FFB3EE4876C |   5C              | pop rsp
00007FFB3EE4876D |   5D              | pop rbp
00007FFB3EE4876E |   5B              | pop rbx
00007FFB3EE4876F |   C3              | ret
00007FFB3EE48770 |   E8 2B26BEFF     | call gameassembly.7FFB3EA2ADA0
```

In practice, we will calculate it dynamically, but that hopefully gives a sense of what needs to happen.

Once that `jmp` is in place, the hook will be live, and the following will happen: when **`SetupMines`** gets called, it will setup the board with mines as usual, then right before its epilog, our relative near `jmp` will get executed. This will `jmp` to our nearby trampoline, which will long `jmp` to **`SM_Bridge`** in our DLL, which will call **`SetupMines_hook`**, finish **`SetupMines`**' epilog, then return to wherever originally called **`SetupMines`**.

A lot of the reason this is so convoluted is a consequence of 64-bit addressing and instructions. Absolute addresses take up 8 bytes, and there aren't any 64-bit jumps that work on immediate values, requiring extra steps loading them into registers. It's challenging to find unused space for all of this.

Some of it is due to design choice. We could have written all of our final hook logic in assembly within **`SM_Bridge`**, but it's convenient and easier to maintain if we can work our way to a nomal C++ function and do anything somewhat complex there. We also could have tried to allocate memory near **`SetupMines`** and avoided the trampoline. Different approaches might be better in different situations, and this isn't necessarily the absolute best approach to this one, but it's one that showcases some of the sorts of things we can do when need be.

Phoning Home

Part of the hook we haven't talked about is how it will transmit the data back out to whatever we want receiving it. This actually ends up being really tricky because many Windows Apps, including Minesweeper, are heavily sandboxed (few to no "Restricted Capabilities" are enabled.)

This makes it extremely difficult to establish any sort of interprocess communication between code running in the context of the App and other processes running on the system (no named pipes, socket connections to localhost, file access outside its home directory, etc..)

I'm sure there's a way we could patch this since we're on the outside of it, but that's getting beyond the scope of this project, so the external injector in the code example just polls for board updates at regular intervals, and our **`board`** structure maintains a counter so we know when there's new data to show.

For just displaying the mine locations like we're doing here, using something like AllocConsole as [2] did would be adequate, but in general there are going to be situations where we want to get information back out of the injected App, and this is an area that could use some more exploration.

Injection

DLL injection is nearly identical to how it would normally be carried out. The injection portion of the external loader I wrote borrows heavily from Kyle Halladay's [5] implementation, but it's just a classic CreateRemoteThread with LoadLibrary as the ThreadProc, with one major difference.

The restrictions on Apps' access to the outside world carries over to reading most DLL files outside of their home directories. Happily we can just copy the security descriptor from a DLL we know it can load from outside its home, like a core Windows library, to the one we want to inject and the problem goes away. If we ever try to hook an App that even this won't work for, we can of course always manually load the DLL, but that's a lot more error-prone work than is generally necessary.

Overall, the loader in our example code looks a lot more complicated that the actual injected code, but that's just because it's doing most of the work finding modules and addresses and there's nothing particularly novel going on there that hasn't already been covered to death elsewhere.

It's also overengineered and could use a rewrite, so there's that.

Au Revoir

Hopefully this was helpful to someone wanting to play with modifying these kinds of apps, or just learn more about 64-bit injection and hooking in general. I learned a lot more working on this than I ever did by reading anything, so go try something out. Exploring Adventure mode would probably be an interesting challenge that builds on what we did here.

Be aware that some game manufacturers will strip or obfuscate all of the symbolic infomation we were able to extract and take advantage of here, but working on programs like this that do have that information present can give some insight into how these sorts of things are usually structured so you can better recognize what's going on when you don't have it to rely on, just like reversing your own programs with debug information helps build your pattern recognition for working on other programs.

References

[1] https://fearlessrevolution.com/viewtopic.php?t=20123
[2] https://github.com/morsisko/MineSweeper-Bot
[3] https://github.com/charlesnathansmith/minehack
[4] https://github.com/Perfare/Il2CppDumper
[5] https://github.com/charlesnathansmith/Il2CppDumper/blob/master/Il2CppDumper/ida7.py
[6] http://kylehalladay.com/blog/2020/11/13/Hooking-By-Example.html
[7] https://codemachine.com/articles/x64_deep_dive.html#parameter_homing