Introduction to Reverse Engineering Win32 Applications

trew

trew@exploit.us

Contents

Chapter 1

Foreword

Abstract: During the course of this paper the reader will be (re)introduced to many concepts and tools essential to understanding and controlling native Win32 applications through the eyes of Windows Debugger (WinDBG). Throughout, WinMine will be utilized as a vehicle to deliver and demonstrate the functionality provided by WinDBG and how this functionality can be harnessed to aid the reader in reverse engineering native Win32 applications. Topics covered include an introductory look at IA-32 assembly, register significance, memory protection, stack usage, various WinDBG commands, call stacks, endianness, and portions of the Windows API. Knowledge gleaned will be used to develop an application designed to reveal and/or remove bombs from the WinMine playing grid.

Thanks: The author would like to thank thief, skape, arachne, H D Moore, h1kari, Peter, warlord, west, and everyone else that participated in the initial release of the Uninformed Journal.

Chapter 2

Introduction

Games can often times be very frustrating. This frustration stems from the inherent fact that games, by design, present many unknowns to the player. For example, how many monsters are lurking behind door number three, and are these eight clips of 90 50 caliber rounds going to be enough to kill this guy? Ten lives and a broken keyboard later, acquiring the ability to not only level the playing field, but get even, grows extremely attractive, at any cost. Some people risk reputational and karma damage to acquire that edge – by cheating.

Many develop cheats for this very reason, to obtain an unfair advantage. Others, however, have an entirely different motivation – the challenge it involves. Motivations aside, the purpose of this document is to familiarize the reader with basic methodologies and tools available that aid in the practice of reverse engineering native Windows applications. Throughout, the reader will be introduced to WinDBG, IA-32 assembler, and portions of the Windows API. These concepts will be demonstrated by example, via a step by step navigation through the portions of WinMine that are pivotal in obtaining the coveted unfair advantage.

Chapter 3

Getting Started

Although this document is designed to speak at an introductory level, it is expected that the reader satisfies the following prerequisites:

1. `Understanding of hexadecimal number system`
2. `The ability to develop basic C applications`
3. `The ability to install and properly configure WinDBG`
4. `Access to a computer running Windows XP with WinMine installed`[1]

The following are suggested materials to have available while reading this document:

1. `IA-32 Instruction Set Reference A-M [7]`
2. `IA-32 Instruction Set Reference N-Z [7]`
3. `IA-32 Volume 1 - Basic Architecture [7]`
4. `Microsoft Platform SDK [4]`
5. `Debugger Quick Reference [8]`

First, WinDBG and the Symbol Packages[2] need to be properly installed and configured. WinDBG is part of The Debugging Tools Windows[3] package.

[1]The version of WinMine varies between Windows release.
[2]http://msdl.microsoft.com/download/symbols/packages/windowsxp/WindowsXP-KB835935-SP2-slp-Symbols.exe
[3]http://msdl.microsoft.com/download/symbols/debuggers/dbg_x86_6.4.7.2.exe

While these download, the time will be passed by identifing potential goals, articulating what a debugger is, what abilities they provide, what symbols are, and how they are useful when debugging applications.

3.1 Identifying Goals

The basic strategy behind WinMine is to identify the location of bombs within a given grid and clear all unmined blocks in the shortest duration of time. The player may track identified bomb locations by placing flags or question marks upon suspect blocks. With this in mind, one can derive the following possible goals:

1. `Control or modify time information`
2. `Verify the accuracy of flag and question mark placement`
3. `Identify the location of bombs`
4. `Remove bombs from the playing grid`

In order to achieve these goals, the reader must first determine the following:

1. `The location of the playing grid within the WinMine process`
2. `How to interpret the playing grid`
3. `The location of the clock within the WinMine process`

For the scope of this paper, the focus will be on locating, interpreting, and revealing and/or removing bombs from the playing grid.

3.2 Symbols and Debuggers

A debugger is a tool or set of tools that attach to a process in order to control, modify, or examine portions of that process. More specifically, a debugger provides the reader with the ability to modify execution flow, read or write process memory, and alter register values. For these reasons, a debugger is essential for understanding how an application works so that it can be manipulated in the reader's favor.

Typically, when an application is compiled for release, it does not contain debugging information, which may include source information and the names of functions and variables. In the absence of this information, understanding the application while reverse engineering becomes more difficult. This is where symbols come in, as they provide the debugger, amongst other things, the previously unavailable names of functions and variables. For more information on symbols, the reader is encouraged to read the related documents in the reference section.[3]

3.3 Symbol Server

Hopefully by now both the Debugging Tools for Windows and the Symbols Packages have finished downloading. Install them in either order but take note of the directory the symbols are installed to. Once both are installed, begin by executing WinDBG, which can be found under `Debugging Tools for Windows`, beneath `Programs`, within the Start Menu. Once WinDBG is running click on `File`, `Symbol File Path`, and type in the following:

```
SRV*<path to symbols>*http://msdl.microsoft.com/download/symbols
```

For example, if symbols were installed to:

```
C:\Windows\Symbols
```

then the reader should enter:

```
SRV*C:\WINDOWS\Symbols*http://msdl.microsoft.com/download/symbols
```

This configuration tells WinDBG where to find the previously installed symbols, and if needed symbols are unavailable, where to get them from – the Symbol Server. For more information on Symbol Server, the reader is encouraged to read the information in the reference section.[2]

Chapter 4

Getting Familiar with WinDBG

Whether the reader points and clicks their way through applications or uses shortcut keys, the WinDBG toolbar will briefly act as a guide for discussing some basic debugging terminology that will be used throughout this document. From left to right, the following options are available:

1. `Open Source Code` Open associated source code for the debugging session.
2. `Cut` Move highlighted text to the clipboard
3. `Copy` Copy highlighted text to the clipboard
4. `Go` Execute the debugee
5. `Restart` Restart the debugee process[1]
6. `Stop Debugging` Terminate the debugging session[2]
7. `Break` Pause the currently running debugee process

The next four options are used after the debugger has been told to break. The debugger can be issued a break via the previous option, or the user may specify `breakpoints`. Breakpoints can be assigned to a variety of conditions. Most common are when the processor executes instructions at a specific address, or when certain areas of memory have been accessed. Implementing breakpoints will be discussed in more detail later in this document.

[1]This will cause the debugee to terminate.
[2]This will cause the debugee to terminate.

Once a breakpoint has been reached, the process of executing individual instructions or function calls is referred to as `stepping` through the process. WinDBG has a handful of methods for stepping, four of which will be immediately discussed.

1. `Step Into` Execute a single instruction. When a function is called, this will cause the debugger to step into that function and break, instead of executing the function in its entirety.
2. `Step Over` Execute one or many instructions. When a function is called, this will cause the debugger to execute the called function and break after it has returned.
3. `Step Out` Execute one or many instructions. Causes the debugger to execute instructions until it has returned from the current function.
4. `Run to Cursor` Execute one or many instructions. Causes the debugger to execute instructions until it has reached the addresses highlighted by the cursor.

Next, is `Modify Breakpoints` which allows the reader to add or modify breakpoints. The remainder of the toolbar options is used to make visible and customize various windows within WinDBG.

4.1 WinDBG Windows

WinDBG provides a variety of windows, which are listed beneath the `View` toolbar option, that provide the reader with a variety of information. Of these windows, we will be utilizing `Registers`, `Disassembly`, and `Command`. The information contained within these three windows is fairly self describing.

The `Registers` window contains a list of all processor registers and their associated values. Note, as register values change during execution the color of this value will turn red as a notification to the reader. For the purpose of this document, we will briefly elaborate on only the following registers: `eip, ebp, esp, eax, ebx, ecx, edx, esi, and edi`.

1. `eip` Contains the address of the next instruction to be executed
2. `ebp` Contains the address of the current stack frame
3. `esp` Contains the address of the top of the stack[3]

[3]This will be discussed in greater detail further in the document.

The remaining listed registers are for general use. How each of these registers are utilized is dependant on the specific instruction. For specific register usage on a per instruction basis, the reader is encouraged to reference the IA-32 Command References [7].

The `Disassembly` window will contain the assembly instructions residing at a given address, defaulting at the value stored within the `eip` register.

The `Command` window will contain the results of requests made of the debugger. Note, at the bottom of the `Command` window is a text box. This is where the user issues commands to the debugger. Additionally, to the left of this box is another box. When this box is blank the debugger is either detached from a process, processing a request, or the debugee is running. When debugging a single local process in user-mode, this box will contain a prompt that resembles "`0:001>`". For more information on interpreting this prompt, the reader is encouraged to read the related documentation in the reference section [9].

There exists three classes of commands that we can issue in the `Command` window; `regular, meta, and extension`. `Regular` commands are those commands designed to allow the reader to interface with the debugee. `Meta` commands are those commands prefaced with a period (.) and are designed to configure or query the debugger itself. `Extension` commands are those commands prefaced with an exclamation point (!) and are designed to invoke WinDBG plug-ins.

Chapter 5

Locating the WinMine Playing Grid

Let's begin by firing up WinMine, via Start Menu ->Run ->WinMine. Ensure WinMine has the following preferences set:

```
Level: Custom
    Height: 900
    Width:  800
    Mines:  300
Marks: Enabled
Color: Enabled
Sound: Disabled
```

Once this is complete, compile and execute the supplemental `SetGrid` application[12] found in the reference section. This will ensure that the reader's playing grid mirrors the grid utilized during the writing of this paper. Switch over to WinDBG and press F6. This will provide the reader with a list of processes. Select `winmine.exe` and press `Enter`. This will attach WinDBG to the WinMine process. The reader will immediately notice the `Command`, `Registers`, and `Disassembly` windows now contain values.

5.1 Loaded Modules

If the reader directs attention to the `Command` window it is noticed that a series of modules are loaded and the WinMine process has been issued a break.

```
ModLoad: 01000000 01020000   C:\WINDOWS\System32\winmine.exe
ModLoad: 77f50000 77ff7000   C:\WINDOWS\System32\ntdll.dll
ModLoad: 77e60000 77f46000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 77c10000 77c63000   C:\WINDOWS\system32\msvcrt.dll
...
ModLoad: 77c00000 77c07000   C:\WINDOWS\system32\VERSION.dll
ModLoad: 77120000 771ab000   C:\WINDOWS\system32\OLEAUT32.DLL
ModLoad: 771b0000 772d4000   C:\WINDOWS\system32\OLE32.DLL
(9b0.a2c): Break instruction exception - code 80000003 (first chance)
eax=7ffdf000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004
eip=77f75a58 esp=00cfffcc ebp=00cffff4 iopl=0   nv up ei pl zr na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000   efl=00000246
ntdll!DbgBreakPoint:
77f75a58 cc               int     3
```

The two 32-bit addresses following ”ModLoad:” represent the virtual memory address range the corresponding module is mapped to. These loaded modules contain functionality that WinMine is dependant upon. To get a list of loaded modules, the reader may issue either of the following commands: `lm, !lm, !dlls`

The reader should also notice that WinDBG, by default, articulates register values within the `Command` window upon reaching a breakpoint or at the completion of each step.

5.2 Loaded Symbols

Time was spent to download and install the `Symbols Packages`, so let's see what hints they provide. Issue the following within the `Command` window get a list of all available symbols for WinMine.

```
x WinMine!*
```

The e(x)amine command interprets everything to the left of the exclamation point as a regular expression mask for the module name, and everything to the right as a regular expression mask for the symbol name. For more information on regular expression syntax, the reader is encouraged to read the related documents in the reference section [10].

A list of symbols will scroll within the `Command` window.

```
...
01003df6 winmine!GetDlgInt = <no type information>
```

```
010026a7 winmine!DrawGrid = <no type information>
0100263c winmine!CleanUp = <no type information>
01005b30 winmine!hInst = <no type information>
01003940 winmine!Rnd = <no type information>
01001b81 winmine!DoEnterName = <no type information>
...
```

From this listing, it is not possible to positively ascertain which symbols represent functions or variables. This is due, as WinDBG has pointed out, to the absence of type information. This is typical of public symbol files. Thankfully, methodologies exist that allow the reader to, at a minimum, distinguish functions from non-functions. Assuming the reader is not well versed reading assemblies, methods requiring that skill set will be for a short time avoided. An alternative technique, examining virtual memory protections, will be investigated that is relatively easy to comprehend and apply,

5.3 Memory Protection

Thus far, discussions related to application memory have been sufficiently neglected, until now. This is not to say the interworkings of Windows memory management are about to be revealed, vice, a fairly pigeon holed approach will be taken for the sake of brevity and to satisfy our immediate utilitarian needs.

When an application requests memory, a region is allocated provided the requested amount is available. If the allocation is successful, this region of memory can, amongst other things, be protected. More specifically, the region has psuedo access control lists applied to it that deny or permit certain access types. A couple examples of these access types are the ability to read information from, write information to, and execute instructions at, the given region. It is these access types that will provide the ability to quickly determine with relatively high probability whether a symbol is a function or non-function. By virtue of being a function, these memory regions allow execution. Conversely, memory regions allocated for classic variables do not allow instruction execution[1]. Conveniently, WinDBG is shipped with an extension that allows the user the retrieve memory protection attributes for a given address. This extension command is `!vprot`. Let's select aptly named symbols to demonstrate this functionality. Type the following in the `Command` window:

```
!vprot WinMine!ShowBombs
```

`ShowBombs` was chosen as the name implies (to me) that it's a function. Let's see what `!vprot` says:

[1]All memory pages on the IA-32 architecture are executable at the hardware level despite memory protections.

```
BaseAddress:       01002000
AllocationBase:    01000000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        00003000
State:             00001000  MEM_COMMIT
Protect:           00000020  PAGE_EXECUTE_READ
Type:              01000000  MEM_IMAGE
```

At first glance this might appear contradictory. However, the `AllocationProtect` field denotes the default protection for the entire memory region. The `Protect` field speaks to the current protections on the specific region specified in the first argument. This, as one would expect, is set to execute and read as denoted by `PAGE_EXECUTE_READ`. Next, look at the memory protection for a region allocated for a suspected variable, such as `WinMine!szClass`.

```
!vprot WinMine!szClass
```

The expectation is `!vprot` will return page protection that only allows read and write access to this region.

```
BaseAddress:       01005000
AllocationBase:    01000000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              01000000  MEM_IMAGE
```

So be it. Considering the naming convention (sz preface), which implies a string type, one could easily validate the assumption by examining the data at this memory location. To do this, the display memory command can be utilized. Type the following in the `Command` window:

```
du WinMine!szClass
```

The 'u' modifier tells the (d)isplay memory command to interpret the string as Unicode. The results of this are:

```
01005aa0  "Minesweeper"
```

I'm convinced.

5.4 Understanding Assemblies

The goal for this chapter is to simply locate where the playing grid resides. With that in mind, revisit the previously identified `ShowBombs` function. Logically, it wouldn't be that long of a jump to assume this function will lead to the playing grid. Set a breakpoint on this function by issuing the following command:

```
bp WinMine!ShowBombs
```

WinDBG provides no positive feedback that the breakpoint was successfully set. However, WinDBG will alert the user if it is unable to resolve the name or address being requested. To obtain a list of set breakpoints, issue the following command:

```
bl
```

The `Command` window should reflect:

```
0 e 01002f80     0001 (0001)  0:*** WinMine!ShowBombs
```

The leading digit represents the breakpoint number, which can be used as a reference for other breakpoint aware commands. The next field depicts the status of the breakpoint, as either (e)nabled or (d)isabled. This is followed by the virtual address of the breakpoint. The next four digits speak to the number of passes remaining until this breakpoint will activate. Adjacent, in parentheses, is the initial pass count. Next, is the process number, not to be confused with process ID, a colon acting as a separator between what would be a thread ID, if this was a thread specific breakpoint. It's not, hence three asterisks. Lastly, at least in the above example, is the module name and symbol/function where WinDBG will break at.

Set WinMine in motion by hitting F5 (Go) or type 'g' in the `Command` window. The reader should notice that WinDBG informs the user the "Debugee is running". Switch to the WinMine window and click on the upper left box, which should reveal the number two. Next, click the box to the right and it will be very apparent that a bomb has been selected, as the reader will no longer be able to interact with WinMine. This is due to the fact that WinDBG recognized that a breakpoint condition has been met and is waiting for instruction. When back in the WinDBG, the `Command` window has highlighted the following instruction:

```
01002f80 a138530001       mov     eax,[WinMine!yBoxMac (01005338)]
```

This is the first instruction within the `ShowBombs` function, which corresponds to the address previously identified when current breakpoints were listed. Before attempting to understand this instruction, let's first cover a few functional and syntactical aspects of IA-32 assembly. It is recommended that the reader make available the aforementioned supplemental material mentioned in Chapter 2.

Each line in the disassembly describes an instruction. Use the above instruction to identify the major components of an instruction without getting distracted by how this instruction relates to the `ShowBombs` function. If distilled, the previous instruction can be abstracted and represented as:

```
<address> <opcodes> <mnemonic> <operand1>, <operand2>
```

The `<address>` represents the virtual location of the `<opcodes>`. Opcodes are literal instructions that the processor interprets to perform work. Everything to the right of `<opcodes>` represents a translation of these opcodes into assembly language. The `<mnemonic>` can be thought of as a verb or function that treats each operand as an argument. It's of importance to note that in Intel[2] style assembly, these operations move from right to left. That is, when performing arithmetic or moving data around, the result typically finds its destination at `<operand1>`.

Looking back at the original instruction one can determine that the 32-bit value, or word, located at 0x01005338 is being copied into the `eax` register. Brackets ([]) are used to deference the address contained in an operand, much like an asterisk (*) does in C. Let's focus on the opcodes for a moment. If the reader looks up the opcode for a `mov` instruction into the `eax` register, the value 0xa1 will be found.

```
Opcode Instruction      Description
...
A0     MOV AL,moffs8*   Move byte at (seg:offset) to AL.
A1     MOV AX,moffs16*  Move word at (seg:offset) to AX.
A1     MOV EAX,moffs32* Move doubleword at (seg:offset) to EAX.
...
```

It is not by coincidence that the first byte of `<opcodes>` is also 0xa1. This leaves `<operand2>` for the remainder, which brings us to the short discussion in endianness.

[2]Opposed to AT&T style, which is utilized by GCC

5.5 Endianness

Endianness refers to the order by which any multi-byte data is stored. There exists two commonly referred conventions: little and big. Little endian systems, which includes the IA-32 architecture, store data starting with the least significant byte, through the most significant byte. Big endian systems do the opposite, storing the most significant byte first. For example, the value 0x11223344 would be stored as 0x44332211 on a little endian system, and 0x11223344 on a big endian system.

Notice the value in `<operand2>` is 0x01005338 and the remainder of `<opcodes>` is 0x38530001. If `<operand2>` is rewritten and expressed in little endian order one can see these values are equal.

```
0x01005338, rewritten for clarity: 0x01 0x00 0x53 0x38
                                      |    |    |    |
                                      |    |    +----|-+
                                      |     +--------|-|-+
                                      +--------------|-|-|-+
                                                     V V V V
                                                  0x38530001
```

With this information, one can see exactly how the processor is instructed to move the value stored at 0x01005338 into the `eax` register. For more information on endianness, the reader is encouraged to read the related documents in the reference section [5].

5.6 Conditions

Let's see if this new information can be applied to aid in reaching the goal of locating the playing grid. Start by hitting F10, or by typing 'p' in the `Command` window, to execute the current instruction and break. There are a couple of things to notice. First, the previously magenta colored bar that highlighted the examined instruction from above is now red and the instruction just below this is now highlighted blue. WinDBG, by default, denotes instructions that satisfy a breakpoint condition with a red highlight and the current instruction with a blue highlight. Additionally, a handful of values in the `Registers` window have been highlighted in red. Remember from Chapter 4 that this signifies an updated register value. As one would expect, the `eax` register has been updated, but what does its new value represent? 0x18, which now resides in `eax`, can be expressed as 24 in decimal. Note that our playing grid, even though previously specified at 800x900, was rendered at 30x24. Coincidence? This can be validated by restarting WinMine with varying grid sizes, but for the sake of brevity let the following statement evaluate as true:

```
winmine!yBoxMac == Height of Playing Grid
```

The following instructions:

```
01002f85 83f801 cmp     eax,0x1
01002f88 7c4    jl      winmine!ShowBombs+0x58 (01002fd8)
```

compare this value, the maximum height, to the literal numeric 0x1. If the reader visits the description of the `cmp` instruction in the reference material it can be determined that this command sets bits within `EFLAGS`[6] based on the result of the comparison. Logically, the next instruction is a conditional jump. More specifically, this instruction will jump to the address 0x01002fd8 if `eax` is "Less, Neither greater nor equal" than 0x1. One can come to this conclusion by first recognizing that any mnemonic starting with the letter 'j' and is not `jmp` is a conditional jump. The condition by which to perform the jump is represented by the following letter or letters. In this case an 'l', which signifies "Jump short if less" per the definition of this instruction found in the instruction reference and the previously mentioned `EFLAGS` definition. This series of instructions can be expressed in more common terms of:

```
if(iGridHeight < 1) {
        //jmp winmine!ShowBombs+0x58
}
```

Translating assembly into pseudo code or C may be helpful when attempting to understand large or complex functions. One can make the prediction that the conditional jump will fail, as `eax` is currently valued at 0x18. But, for the mere academics of it, one can determine what would happen by typing the following in the `Command` window:

```
u 0x1002fd8
```

This will show the reader the instructions that would be executed should the condition be met.

5.7 Stacks and Frames

The 'u' command instructs WinDBG to (u)nassemble, or translate from opcodes to mnemonics with operands, the information found at the specified address.

```
01002fd8 e851f7ffff      call    WinMine!DisplayGrid (0100272e)
01002fdd c20400          ret     0x4
```

From this, one can see that the `DisplayGrid` function is called and the `ShowBombs` function subsequently returns to the caller. But what is `call` actually doing? Can one tell where `ret` is really returning to and what does the 0x4 represent? The IA-32 Command Reference states that `call` "Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure." The reader may notice that the command reference has variable behaviors for the `call` instruction depending on the type of call being made. To further identify what `call` is doing, the reader can examine the opcodes, as previously discussed, and find 0xe8. 0xe8, represents a near call. "When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) onto the stack (for use later as a return-instruction pointer)." This is the first step in building a frame. Each time a function call is made, another frame is created so that the called function can access arguments, create local variables, and provide a mechanism to return to calling function. The composition of the frame is dependant on the function calling convention. For more information on calling conventions, the reader is encouraged to read the relevant documents in the reference section[1]. To view the current call stack, or series of linked frames, use the 'k' command.

```
ChildEBP RetAddr
0006fd34 010034b0 winmine!ShowBombs
0006fd40 010035b0 winmine!GameOver+0x34
0006fd58 010038b6 winmine!StepSquare+0x9e
0006fd84 77d43b1f winmine!DoButton1Up+0xd5
0006fdb8 77d43a50 USER32!UserCallWinProcCheckWow+0x150
0006fde4 77d43b1f USER32!InternalCallWinProc+0x1b
0006fe4c 77d43d79 USER32!UserCallWinProcCheckWow+0x150
0006feac 77d43ddf USER32!DispatchMessageWorker+0x306
0006feb8 010023a4 USER32!DispatchMessageW+0xb
0006ff1c 01003f95 winmine!WinMain+0x1b4
0006ffc0 77e814c7 winmine!WinMainCRTStartup+0x174
0006fff0 00000000 kernel32!BaseProcessStart+0x23
```

With this, the reader can track the application flow in reverse order. The reader may find it easier to navigate the call stack by pressing Alt-F6, which will bring up the `Call Stack` window. Here, the call stack information is digested a bit more and displayed in a tabular manner. With the call stack information, one can answer the second question regarding where `ret` is really headed. For example, once `ShowBombs` returns, `eip` will be set to 0x010034b0. Finally, the significance of 0x4 can be learned by reading the `ret` instruction definition, which states that this value represents "...the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed." More specifically, when the processor encounters a `ret` instruction it pops the address stored at the top of the stack, where `esp` is pointing, and places that value in the `eip` register. If the `ret` instruction has an operand, that operand represents how many additional bytes should be removed from the top of the stack. With this information, and knowing that 32-bit addresses are four bytes long, one can determine that the `ShowBombs` function accepts one argument[3].

Returning to the task at hand, the following represents the current picture of what the `ShowBombs` function is doing:

```
if(iGridHeight < 1) {
        DisplayGrid();
        return;
} else {
        //do stuff
}
```

Continuing on, one can see the following in the `Disassembly` window, which will take the place of "//do stuff".

```
01002f8a 53              push    ebx
01002f8b 56              push    esi
01002f8c 8b3534530001    mov     esi,[winmine!xBoxMac (01005334)]
01002f92 57              push    edi
01002f93 bf60530001      mov     edi,0x1005360
```

Begin by stepping WinDBG twice (by pressing 'p' twice) so that `eip` is set to 0x1002f8a. The next two instructions are storing the `ebx` and `esi` registers on the stack. This can be demonstrated by first viewing the memory referenced by `esp`, identifying the value stored in `ebx`, pressing 'p' to execute `push ebx`, and revisiting the value stored at `esp`. The reader will find the value of `ebx` stored at `esp`.

```
0:000> dd esp
0006fd38  010034b0 0000000a 00000002 010035b0
0006fd48  00000000 00000000 00000200 0006fdb8
...
0:000> r ebx
```

[3]This is dependant upon calling convetion, which will be discussed later in this document

```
ebx=00000001
0:000> p
eax=00000018 ebx=00000001 ecx=0006fd14 edx=7ffe0304 esi=00000000 edi=00000000
eip=01002f8b esp=0006fd34 ebp=0006fdb8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
winmine!ShowBombs+0xb:
01002f8b 56               push    esi
0:000> dd esp
0006fd34  00000001 010034b0 0000000a 00000002
0006fd44  010035b0 00000000 00000000 00000200
```

Notice, that `esp` has been decremented by four (the size of a 32-bit pointer) and the value of `ebx` is at that location. The behavior can be observed again by stepping to execute `push esi`. Again, the reader will notice the value of `esp` decrement by four and the value within `esi` is at this new location. This is the basic principal of how the stack works. The stack pointer is decremented and the value being push'd onto the stack is placed at the new address `esp` is pointing at. It is also important to note that the stack grows down. That is, as values are placed on the stack, the stack pointer decreases to make room. Which begs the question, what are the upper and lower limits of the stack? It can't keep on growing for ever, can it? The short answer is no, the stack has a floor and ceiling. Which can be identified by examining the `Thread Environment Block` or TEB. Luckily, WinDBG comes with an extension command to accomplish this, `!teb`.

```
0:000> !teb
TEB at 7ffde000
    ExceptionList:        0006fe3c
    StackBase:            00070000
    StackLimit:           0006c000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffde000
    EnvironmentPointer:   00000000
    ClientId:             00000ff4 . 00000ff8
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffdf000
    LastErrorValue:       183
    LastStatusValue:      c0000008
    Count Owned Locks:    0
    HardErrorMode:        0
```

Note the values for `StackBase` and `StackLimit`, which refer to the stack's ceiling and floor, respectively. For more information on the TEB, the reader is encouraged to read the related documents in the reference section [11]. That was an exciting tangent. Circling back, the reader is found at the following instruction:

```
01002f8c 8b3534530001      mov     esi,[winmine!xBoxMac (01005334)]
```

This, if convention holds true, will store the width of the playing grid in `esi`. By single stepping ('p'), the reader will notice the `esi` register is denoted in red within the `Registers` window and now contains the value 0x1e. 0x1e is 30 in decimal, which, if the reader recalls, is the width of the current playing grid. Hence, one can make the educated determination that `xBoxMac` represents the width of the playing grid. The next instruction, `push edi`, is saving the value in the `edi` register on the stack in preparation for the subsequent instruction; `mov edi,0x1005360`. This is were things get a bit more interesting, as this instruction begs the question; what is the significance of 0x1005360? Considering the previous instructions gathered prerequisite information about the playing grid, perhaps this address is indeed the playing grid itself! To determine this, the reader should examine some aspects of this memory address. The aforementioned `!vprot` extension command will provide information regarding the type of access permitted to this memory address, which is `PAGE_READWRITE`. This information isn't overly valuable but is favorable in the sense that this address does not reside within an executable portion of the application space and is therefore likely a variable allocation. If this area is truly the playing grid one should be able to identify a pattern while viewing the memory. To accomplish this, type 0x1005360 in to the `Memory` window. The following should appear:

```
01005360 10 42 cc 8f 8f 8f 8f 0f 8f 8f 8f 8f 0f 0f 8f 0f  .B..............
01005370 0f 8f 8f 8f 8f 8f 8f 0f 0f 0f 8f 0f 0f 8f 8f 10  ................
01005380 10 8f 0f 0f 8f 8f 0f 0f 0f 0f 0f 8f 0f 0f 8f 8f  ................
01005390 0f 8f 0f 0f 0f 8f 8f 8f 0f 0f 8f 8f 8f 8f 8f 10  ................
010053a0 10 0f 0f 8f 0f 0f 8f 0f 0f 0f 0f 0f 8f 0f 0f 0f  ................
010053b0 8f 0f 0f 0f 8f 8f 0f 0f 8f 0f 8f 0f 8f 8f 0f 10  ................
010053c0 10 0f 0f 8f 0f 0f 8f 0f 0f 0f 8f 0f 0f 8f 0f 0f  ................
010053d0 8f 0f 0f 8f 0f 0f 0f 8f 0f 0f 0f 8f 0f 0f 0f 10  ................
010053e0 10 0f 0f 8f 0f 8f 8f 0f 0f 8f 8f 0f 0f 8f 0f 0f  ................
010053f0 0f 0f 0f 0f 8f 0f 0f 0f 0f 0f 0f 0f 8f 0f 0f 10  ................
01005400 10 8f 0f 0f 0f 0f 0f 0f 8f 8f 0f 8f 8f 0f 0f 8f  ................
01005410 0f 8f 0f 0f 0f 0f 0f 0f 0f 0f 0f 0f 8f 8f 0f 10  ................
01005420 10 8f 0f 8f 8f 0f 8f 8f 0f 0f 0f 8f 8f 0f 8f 0f  ................
```

Chapter 6

Interpreting the Playing Grid

The reader may make the immediate observation that this portion of memory is littered with a limited set of values. Most notably are 0x8f, 0x0f, 0x10, 0x42, 0xcc. Additionally, one may notice the following repeating pattern:

```
0x10 <30 bytes> 0x10.
```

The number 30 may ring familiar to the reader, as it was encountered earlier when discovering the grid width. One may speculate that each pattern repetition represents a row of the playing grid. To aid in confirming this, switch to WinDBG and resume WinMine by pressing 'g' in the `Command` window. Switch to WinMine and mentally overlay the information in the `Memory` window with the playing grid. A correlation between these can be identified such that each bomb on the playing grid corresponds to 0x8f and each blank position on the playing grid corresponds to 0x0f. Furthermore, one may notice the blown bomb on the playing grid is represented by 0xcc and the number two is represented by 0x42.

To confirm this is indeed the playing grid, it is essential to test the lower bound by performing simple arithmetic and exercising the same technique employed to identify the suspected beginning. The current hypothesis is that each aforementioned pattern represents a row on the playing grid. If this is true, one can multiply 32, the length of our pattern, by the number of rows in the playing grid, 24. The product of this computation is 768, or 0x300 in hexadecimal. This value can be added to the suspected beginning of the grid, which is located at 0x01005360, to derive an ending address of 0x01005660. Restart WinMine by clicking the yellow smiley face, rerun the `SetGrid` helper application, and click the bottom right square on the playing grid. Coincidentally, the number two will appear. Next, click on the position to the immediate left of the number two. This position contains a bomb and will trigger a breakpoint in WinDBG. Switch over to WinDBG and direct attention to the `Memory` window. Press 'Next' in the Memory window twice to bring this range into focus.

```
01005640 10 8f 0f 0f 0f 8f 0f 0f 8f 0f 8f 0f 0f 0f 0f 8f  ................
01005650 0f 8f 0f 0f 0f 8f 0f 0f 0f 0f 0f 0f 0f cc 42 10  ..............B.
01005660 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10  ................
01005670 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10  ................
```

Following the same overlay as before, the reader will notice that the previous correlations can be made between the last row of the playing grid and the information located at 0x01006540, the start of the previously identified 32 byte pattern. Notice, again, each bomb is represented by 0x8f. With this information the reader can reasonably conclude that this is indeed the playing grid.

Chapter 7

Removing Mines

Before venturing into a programmatic method of instrumenting the playing grid the reader will first be introduced to tools provided by WinDBG, more specifically, the (e)nter values command. This command allows the reader to manipulate specific portions of virtual memory and can be utilized to, amongst other things, remove bombs from the WinMine playing grid. First, reset the grid by resuming the WinMine process in WinDBG, clicking on the yellow smiley face in WinMine, and running the `SetGrid` application. Next, click on the top left position to expose the two and break the WinMine process within WinDBG by pressing Control+Break. The reader should recall that the address 0x01005362, to the immediate right of the two, contains a bomb. To demonstrate the enter values command perform the following in the `Command` window.

```
eb 0x01005362 0x0f
```

Resume WinMine in WinDBG and click on the position to the right of the two. Notice, instead of a bomb being displayed, the number two is revealed. The reader could perform the tedious task of performing this function manually throughout the grid, or, one could develop an application to reveal and/or remove the bombs.

7.1 Virtual Mine Sweeper

In this section, the reader will be introduced to portions of the Windows API that will allow one to develop an application that will perform the following:

1. `Locate and attach to the WinMine process`

2. `Read the WinMine playing grid`
3. `Manipulate the grid to either reveal or remove hidden bombs`
4. `Write the newly modified grid back into WinMine application space`

To accomplish the first task, one can enlist the services of the `Tool Help Library`, which is exposed via `Tlhelp32.h`. A snapshot of running processes can be obtained by calling `CreateToolhelp32Snapshot`, which has the following prototype[1]:

```
HANDLE WINAPI CreateToolhelp32Snapshot(
  DWORD dwFlags,
  DWORD th32ProcessID
);
```

This function, when called with `dwFlags` set to `TH32CS_SNAPPROCESS` will provide the reader with a handle to the current process list. To enumerate this list, the reader must first invoke the `Process32First` function, which has the following prototype:

```
BOOL WINAPI Process32First(
  HANDLE hSnapshot,
  LPPROCESSENTRY32 lppe
);
```

Subsequent iterations through the process list are accessible via the `Process32Next` function, which has the following prototype:

```
BOOL WINAPI Process32Next(
  HANDLE hSnapshot,
  LPPROCESSENTRY32 lppe
);
```

As the reader surely noticed, both of these functions return a `LPPROCESSENTRY32`, which includes a variety of helpful information:

```
typedef struct tagPROCESSENTRY32 {
  DWORD dwSize;
  DWORD cntUsage;
  DWORD th32ProcessID;
```

[1]This information can be obtained by referencing the Platform SDK

```
  ULONG_PTR th32DefaultHeapID;
  DWORD th32ModuleID;
  DWORD cntThreads;
  DWORD th32ParentProcessID;
  LONG pcPriClassBase;
  DWORD dwFlags;
  TCHAR szExeFile[MAX_PATH];
} PROCESSENTRY32, *PPROCESSENTRY32;
```

Most notably of which is `szExeFile`, which will allow the reader to locate the WinMine process, and `th32ProcessID`, which provides the process ID to attach to once the WinMine process is found. Once the WinMine process is located, it can be attached to via the `OpenProcess` function, which has the following prototype:

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  DWORD dwProcessId
);
```

Once the WinMine process has been opened, the reader may read the current playing grid from its virtual memory via the `ReadProcessMemory` function, which has the following prototype:

```
BOOL ReadProcessMemory(
  HANDLE hProcess,
  LPCVOID lpBaseAddress,
  LPVOID lpBuffer,
  SIZE_T nSize,
  SIZE_T* lpNumberOfBytesRead
);
```

After the grid is read into the buffer, the reader may loop through it replacing all instances of 0x8f with either 0x8a to reveal bombs, or 0x0f to remove them. This modified buffer can then be written back into the WinMine process with the `WriteProcessMemory` function, which has the following prototype:

```
BOOL WriteProcessMemory(
  HANDLE hProcess,
  LPVOID lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T nSize,
  SIZE_T* lpNumberOfBytesWritten
);
```

With this information, the reader has the tools necessary to develop an application that to reach the ultimate goal of this paper, to reveal and/or remove bombs from the WinMine playing grid. The source code for a functioning demonstration of this can be found in the reference section.[13]

Chapter 8

Conclusion

Throughout this document the reader has been exposed to portions of many concepts required to successfully locate, comprehend, and manipulate the WinMine playing grid. As such, many details surrounding these concepts were neglected for the sake of brevity. In order to obtain a more holistic view of the covered concepts, the reader is encouraged to read those items articulated in the reference section and seek out additional works.

Chapter 9

References

1. Calling Conventions
   http://www.unixwiz.net/techtips/win32-callconv-asm.html
2. Symbol Server
   http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx
3. Symbols
   ms-help://MS.PSDK.1033/debug/base/symbol_files.htm
4. Platform SDK
   www.microsoft.com/msdownload/platformsdk/sdkupdate/
5. Endianness
   http://www.intel.com/design/intarch/papers/endian.pdf
6. EFLAGS
   ftp://download.intel.com/design/Pentium4/manuals/25366514.pdf
   Appendix B
7. Intel Command References
   http://www.intel.com/design/pentium4/manuals/index_new.htm
8. Debugger Quick Reference
   http://www.tonyschr.net/debugging.htm
9. WinDBG Prompt
   Reference WinDBG Help, Search, Command Window Prompt
10. Regular Expressions
    Reference WinDBG Help, Search, Regular Expression Syntax
11. TEB
    http://msdn.microsoft.com/library/en-us/dllproc/base/teb.asp

12. SetGrid.cpp

```
/*******************************************************************
 * SetGrid.cpp - trew@exploit.us
 *
 * This is supplemental code intended to accompany 'Introduction to
 * Reverse Engineering Windows Applications' as part of the Uninformed
 * Journal.  This application sets the reader's playing grid in a
 * deterministic manner so that demonstrations made within the paper
 * correlate with what the reader encounters in his or her instance of
 * WinMine.
 *
 *******************************************************************/

#include <stdio.h>
#include <windows.h>
#include <tlhelp32.h>

#pragma comment(lib, "advapi32.lib")

#define GRID_ADDRESS     0x1005360
#define GRID_SIZE        0x300

int main(int argc, char *argv[]) {

    HANDLE    hProcessSnap        = NULL;
    HANDLE    hWinMineProc        = NULL;

    PROCESSENTRY32 peProcess      = {0};

    unsigned int procFound        = 0;
    unsigned long bytesWritten    = 0;

    unsigned char grid[] =

    "\x10\x0f\x8f\x8f\x8f\x8f\x8f\x0f\x8f\x8f\x8f\x8f\x0f\x0f\x8f\x0f"
    "\x0f\x8f\x8f\x8f\x8f\x8f\x8f\x0f\x0f\x0f\x8f\x0f\x0f\x8f\x8f\x10"
    "\x10\x8f\x0f\x0f\x8f\x8f\x0f\x0f\x0f\x0f\x0f\x8f\x0f\x0f\x8f\x8f"
    "\x0f\x8f\x0f\x0f\x0f\x8f\x8f\x8f\x0f\x0f\x8f\x8f\x8f\x8f\x8f\x10"
    "\x10\x0f\x0f\x8f\x0f\x0f\x8f\x0f\x0f\x0f\x0f\x0f\x8f\x0f\x0f\x0f"
    "\x8f\x0f\x0f\x0f\x8f\x8f\x0f\x0f\x8f\x0f\x8f\x0f\x8f\x8f\x0f\x10"
    "\x10\x0f\x0f\x8f\x0f\x0f\x8f\x0f\x0f\x0f\x8f\x0f\x0f\x8f\x0f\x0f"
    "\x8f\x0f\x0f\x8f\x0f\x0f\x0f\x8f\x0f\x0f\x0f\x8f\x0f\x0f\x0f\x10"
    "\x10\x0f\x0f\x8f\x0f\x8f\x8f\x0f\x0f\x8f\x8f\x0f\x0f\x8f\x0f\x0f"
    "\x0f\x0f\x0f\x0f\x8f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x8f\x0f\x0f\x10"
```

```
"\x10\x8f\x0f\x0f\x0f\x0f\x0f\x0f\x8f\x8f\x0f\x8f\x8f\x0f\x0f\x8f"
"\x0f\x8f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x8f\x8f\x0f\x10"
"\x10\x8f\x0f\x8f\x8f\x0f\x8f\x8f\x0f\x0f\x0f\x8f\x8f\x0f\x8f\x0f"
"\x0f\x0f\x0f\x8f\x0f\x8f\x0f\x8f\x0f\x0f\x8f\x8f\x0f\x8f\x0f\x10"
"\x10\x8f\x8f\x0f\x0f\x0f\x8f\x0f\x0f\x0f\x0f\x8f\x8f\x8f\x8f\x0f"
"\x0f\x0f\x0f\x0f\x0f\x8f\x8f\x8f\x0f\x0f\x0f\x0f\x8f\x8f\x8f\x10"
"\x10\x8f\x0f\x8f\x8f\x8f\x0f\x0f\x0f\x0f\x0f\x8f\x0f\x8f\x0f\x0f"
"\x8f\x8f\x0f\x0f\x0f\x8f\x0f\x8f\x0f\x8f\x0f\x0f\x0f\x0f\x0f\x10"
"\x10\x0f\x0f\x8f\x8f\x0f\x8f\x8f\x8f\x8f\x0f\x0f\x0f\x0f\x0f\x0f"
"\x0f\x0f\x0f\x0f\x0f\x8f\x0f\x8f\x8f\x8f\x8f\x8f\x8f\x8f\x8f\x10"
"\x10\x0f\x0f\x0f\x8f\x8f\x8f\x0f\x8f\x8f\x0f\x0f\x0f\x8f\x0f\x0f"
"\x0f\x8f\x0f\x8f\x0f\x0f\x0f\x8f\x8f\x0f\x0f\x0f\x0f\x8f\x8f\x10"
"\x10\x0f\x8f\x8f\x0f\x8f\x0f\x8f\x0f\x8f\x0f\x8f\x8f\x0f\x0f\x8f"
"\x0f\x0f\x0f\x0f\x0f\x0f\x8f\x8f\x0f\x0f\x8f\x0f\x8f\x0f\x0f\x10"
"\x10\x0f\x0f\x8f\x8f\x0f\x8f\x0f\x0f\x0f\x8f\x0f\x0f\x0f\x8f\x0f"
"\x8f\x0f\x8f\x8f\x8f\x0f\x0f\x8f\x0f\x8f\x0f\x8f\x8f\x8f\x8f\x10"
"\x10\x8f\x8f\x0f\x0f\x0f\x0f\x0f\x0f\x8f\x0f\x8f\x0f\x0f\x8f\x0f"
"\x0f\x0f\x8f\x8f\x8f\x8f\x8f\x0f\x0f\x8f\x8f\x0f\x0f\x8f\x8f\x10"
"\x10\x8f\x0f\x0f\x0f\x8f\x0f\x8f\x8f\x8f\x8f\x0f\x0f\x8f\x8f\x0f"
"\x0f\x8f\x0f\x0f\x8f\x8f\x8f\x8f\x0f\x8f\x0f\x8f\x0f\x8f\x8f\x10"
"\x10\x0f\x8f\x8f\x0f\x0f\x8f\x8f\x8f\x0f\x8f\x0f\x0f\x0f\x0f\x0f"
"\x0f\x8f\x8f\x8f\x0f\x0f\x8f\x0f\x8f\x8f\x8f\x0f\x8f\x8f\x0f\x10"
"\x10\x8f\x0f\x0f\x8f\x8f\x8f\x8f\x0f\x0f\x8f\x0f\x0f\x0f\x8f\x8f"
"\x8f\x8f\x0f\x0f\x0f\x0f\x0f\x8f\x0f\x8f\x8f\x0f\x0f\x8f\x0f\x10"
"\x10\x0f\x8f\x8f\x0f\x0f\x0f\x0f\x8f\x0f\x8f\x0f\x8f\x0f\x0f\x0f"
"\x0f\x0f\x0f\x8f\x0f\x0f\x0f\x8f\x0f\x0f\x0f\x8f\x0f\x8f\x0f\x10"
"\x10\x0f\x0f\x0f\x0f\x8f\x8f\x8f\x8f\x8f\x0f\x0f\x0f\x8f\x0f\x0f"
"\x8f\x8f\x8f\x0f\x0f\x8f\x8f\x8f\x0f\x0f\x8f\x0f\x0f\x8f\x0f\x10"
"\x10\x8f\x8f\x0f\x8f\x8f\x0f\x8f\x8f\x0f\x0f\x0f\x0f\x8f\x8f\x8f"
"\x8f\x0f\x8f\x0f\x0f\x0f\x8f\x0f\x8f\x8f\x8f\x0f\x8f\x0f\x0f\x10"
"\x10\x0f\x8f\x8f\x0f\x0f\x8f\x8f\x8f\x0f\x0f\x8f\x0f\x0f\x0f\x0f"
"\x0f\x0f\x8f\x8f\x0f\x8f\x0f\x0f\x0f\x0f\x0f\x0f\x8f\x0f\x8f\x10"
"\x10\x0f\x0f\x8f\x0f\x8f\x0f\x8f\x8f\x0f\x0f\x0f\x0f\x0f\x0f\x0f"
"\x0f\x8f\x0f\x0f\x0f\x0f\x0f\x0f\x8f\x0f\x0f\x0f\x0f\x0f\x8f\x10"
"\x10\x0f\x8f\x8f\x8f\x0f\x8f\x0f\x8f\x0f\x0f\x8f\x0f\x0f\x8f\x0f"
"\x0f\x8f\x8f\x0f\x0f\x0f\x0f\x8f\x0f\x8f\x8f\x0f\x0f\x0f\x8f\x10"
"\x10\x8f\x0f\x0f\x0f\x8f\x0f\x0f\x8f\x0f\x8f\x0f\x0f\x0f\x0f\x8f"
"\x0f\x8f\x0f\x0f\x0f\x8f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x8f\x8f\x10";


//Get a list of running processes
hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

if(hProcessSnap == INVALID_HANDLE_VALUE) {
    printf("Unable to get process list (%d).\n", GetLastError());
    return 0;
}

peProcess.dwSize = sizeof(PROCESSENTRY32);

//Get first process in list
if(Process32First(hProcessSnap, &peProcess)) {

    do {
        //Is it’s winmine.exe?
        if(!stricmp(peProcess.szExeFile, "winmine.exe")) {

            printf("Found WinMine Process ID (%d)\n", peProcess.th32ProcessID);
            procFound = 1;

            //Get handle on winmine process
            hWinMineProc = OpenProcess(PROCESS_ALL_ACCESS,
                                       1,
                                       peProcess.th32ProcessID);

            //Make sure the handle is valid

            if(hWinMineProc == NULL) {
                printf("Unable to open minesweep process (%d).\n", GetLastError());
                return 0;
            }

            //Write grid
            if(WriteProcessMemory(hWinMineProc,
                                  (LPVOID)GRID_ADDRESS,
                                  (LPCVOID)grid,
                                  GRID_SIZE,
                                  &bytesWritten) == 0) {
                printf("Unable to write process memory (%d).\n", GetLastError());
                return 0;
            } else {
                printf("Grid Update Successful\n");
            }

            //Let go of minesweep
            CloseHandle(hWinMineProc);
            break;
        }

    //Get next process
    } while(Process32Next(hProcessSnap, &peProcess));
}
```

```
    if(!procFound)
        printf("WinMine Process Not Found\n");

    return 0;
}
```

13. MineSweeper.cpp

```
/*********************************************************************
 * MineSweeper.cpp - trew@exploit.us
 *
 * This is supplemental code intended to accompany 'Introduction to
 * Reverse Engineering Windows Applications' as part of the Uninformed
 * Journal.  This application reveals and/or removes mines from the
 * WinMine grid.  Note, this code only works on the version of WinMine
 * shipped with WinXP, as the versions differ between releases of
 * Windows.
 *
 *********************************************************************/

#include <stdio.h>
#include <windows.h>
#include <tlhelp32.h>

#pragma comment(lib, "advapi32.lib")

#define BOMB_HIDDEN      0x8f
#define BOMB_REVEALED    0x8a
#define BLANK            0x0f
#define GRID_ADDRESS     0x1005360
#define GRID_SIZE        0x300

int main(int argc, char *argv[]) {

    HANDLE    hProcessSnap        = NULL;
    HANDLE    hWinMineProc        = NULL;

    PROCESSENTRY32 peProcess      = {0};

    unsigned char procFound       = 0;
    unsigned long bytesWritten    = 0;
    unsigned char *grid           = 0;
    unsigned char replacement     = BOMB_REVEALED;
    unsigned int x                = 0;
```

```
grid = (unsigned char *)malloc(GRID_SIZE);

if(!grid)
    return 0;

if(argc > 1) {
    if(stricmp(argv[1], "remove") == 0) {
        replacement = BLANK;
    }
}

//Get a list of running processes
hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

//Ensure the handle is valid
if(hProcessSnap == INVALID_HANDLE_VALUE) {
    printf("Unable to get process list (%d).\n", GetLastError());
    return 0;
}

peProcess.dwSize = sizeof(PROCESSENTRY32);

//Get first process in list
if(Process32First(hProcessSnap, &peProcess)) {

    do {
        //Is it's winmine.exe?
        if(!stricmp(peProcess.szExeFile, "winmine.exe")) {

            printf("Found WinMine Process ID (%d)\n", peProcess.th32ProcessID);
            procFound = 1;

            //Get handle on winmine process
            hWinMineProc = OpenProcess(PROCESS_ALL_ACCESS,
                                       1,
                                       peProcess.th32ProcessID);

            //Make sure the handle is valid
            if(hWinMineProc == NULL) {
                printf("Unable to open minesweep process (%d).\n", GetLastError());
                return 0;
            }

            //Read Grid
            if(ReadProcessMemory(hWinMineProc,
                                 (LPVOID)GRID_ADDRESS,
                                  (LPVOID)grid, GRID_SIZE,
                                  &bytesWritten) == 0) {
                    printf("Unable to read process memory (%d).\n", GetLastError());
                    return 0;
                } else {
                    //Modify Grid
                    for(x=0;x<=GRID_SIZE;x++) {
                        if((*(grid + x) & 0xff) == BOMB_HIDDEN) {
                            *(grid + x) = replacement;
                        }
                    }
                }

                //Write grid
                if(WriteProcessMemory(hWinMineProc,
                                      (LPVOID)GRID_ADDRESS,
                                      (LPCVOID)grid,
                                      GRID_SIZE,
                                      &bytesWritten) == 0) {
                    printf("Unable to write process memory (%d).\n", GetLastError());
                    return 0;
                } else {
                    printf("Grid Update Successful\n");
                }

                //Let go of minesweep
                CloseHandle(hWinMineProc);
                break;
            }

        //Get next process
        } while(Process32Next(hProcessSnap, &peProcess));
    }

    if(!procFound)
        printf("WinMine Process Not Found\n");

    return 0;
}
```