FIRST EDITION – CHAPTER 2 REV 1

Kevin Thomas

Forward

A long time ago there existed a time and space where the 6502 processor was everywhere. There was no internet, there was no cell phone and the personal computer was that of a creation of pure majesty which had a target market of a few enthusiasts.

On November 20, 1985, Microsoft introduced the Windows operating environment which was nothing more than a graphical operating shell for MS-DOS.

I will spare you the rest of the history as we know how this game played out. Today, Windows is the most used desktop and laptop OS having a 76% share followed by Apple’s macOS at 16% and the remaining ChromeOS and other Linux variants.

Like it or not Windows is the major player and throughout the years I have focused on teaching Reverse Engineering in the Linux environment so that we could focus on a more thinner and efficient development and communication with the processor.

Today we begin our journey into the Win32API. This book will take you step-by-step writing very simple Win32API’s in both x86 and x64 platforms in C++ and then reversing them both very carefully using the world’s most popular Hey Rays IDA Free tool which is a stripped down version of the IDA Pro tool used in more professional Reverse Engineering environments.

Let’s begin...

Table Of Contents

Chapter 1: Hello World

We begin our journey with programming a very simple hello world program in Windows Assembly language. We will ONLY write in pure Assembly in this chapter as we will focus on development in C++ which almost all Windows development occurs so you have a greater understanding of how these applications are put together and THEN reversing the entire app in Assembly Language both in x86 and x64.

Let's first download Visual Studio which we will use as our integrated development environment. Select the Visual Studio 2019 Community edition at the link below. Make SURE you select all of the C++ and Windows options during the setup to ensure the build environment has all the tools necessary. When in doubt, check the box to include it during install.

https://visualstudio.microsoft.com/downloads

Once installed, let's create a new project and get started by following the below steps.

```
Create a new project
Empty Project
Next
Project name: 0x0001-hello_world-x86
CHECK Place solution and project in the same directory
Create

RT CLICK on the 0x0001-hello_world-x86 in Solutions Explorer
Add
New Item…
main.asm
RT CLICK 0x0001-hello_world-x86
Build Dependencies
Build Customizations
CHECK masm
OK

RT CLICK on main.asm
Properties
Configuration Properties
General
Item Type: Microsoft Macro Assembler
OK
```

Now let's populate our **main.asm** file with the following.

```
.686
.model flat, stdcall
.stack 4096
```

```
extrn  ExitProcess@4: proc        ;1 param  1x4
extrn  MessageBoxA@16: proc       ;4 params 4x4

.data
    msg_txt      db "Hello World", 0
    msg_caption  db "Hello World App", 0

.code
    main:
        push  0                   ;UINT uType
        lea   eax, msg_caption    ;LPCSTR lpCaption
        push  eax
        lea   eax, msg_txt        ;LPCSTR lpText
        push  eax
        push  0                   ;HWND hWnd
        call  MessageBoxA@16

        push  0                   ;UINT uExitCode
        call  ExitProcess@4
    end main
```

Congratulations! You just created your first hello world code in x86 Windows Assembly. Time for cake!

We are going to spend the majority of our time in the Win32API documentation throughout this course.

Let's take a moment and review. To begin we designate a *.686* which means enable the assembly of non-privileged instructions for the Pentium Pro+ style architecture in 32-bit MASM.

(VISIT https://docs.microsoft.com/en-us/cpp/assembler/masm/dot-686?view=msvc-160)

We then set up a *flat* memory model which uses no combined segment or offset addressing. We also use the *stdcall* Win32 callign convention which we push args in reverse order onto the stack and then call the procedure.

Our first Win32API that we will call is the *ExitProcess* which simply exits the application and frees up the operation to the Windows OS.

(VISIT https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitprocess)

We see that the function is a void function which returns nothing and has one param UINT uExitCode which simply retrieves the process's exit value.

You might have noticed a very strange *@4* after the function. This is to designate that the function has 1 param. We multiply each param by 4 to get this designation.

Our next Win32API is the *MessageBoxA* function which simply displays a modal dialog box with a title and a message.

(VISIT https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxa)

We have 4 params here so we know we will have an @16 at the end of the function.

The first param is *HWND hWnd* which is a handle to the owner of the window of the message box to be created and in our case it is *NULL* meaning the message bhas has no owner.

We then have the *LPCSTR lpText* which will display our text inside the message box.

We then have the *LPCSTR lpCaption* which will be the caption text on the message box.

Finally we have the UINT uType which is simply the combo of flags from the table located in the docs. In our case it will be NULL.

Remember in stdcall we push the params in REVERSE order onto the stack as you see in the code above.

At this point we can run our code by clicking on the green arrow next to the Local Windows Debugger.

HOORAY our hello world modal dialog box pops up.

Let’s now create our x64 version of this code.

```
Create a new project
Empty Project
Next
Project name: 0x0001-hello_world-x64
CHECK Place solution and project in the same directory
Create

RT CLICK on the 0x0001-hello_world-x64 in Solutions Explorer
Add
New Item…
main.asm
RT CLICK 0x0001-hello_world-x64
Build Dependencies
```

Build Customizations
CHECK masm
OK

RT CLICK on the 0x0001-hello_world-x64 in Solutions Explorer

RT CLICK on main.asm
Properties
Configuration Properties
Linker
Advanced
Entry Point: main
OK

Select x64 to the right of Debug and the left of Local Windows Debugger menu bar

Now let's populate our **main.asm** file with the following.

```
extrn MessageBoxA: proc
extrn ExitProcess: proc

.data
	msg_txt     db 'Hello World', 0
	msg_caption db 'Hello World App', 0

.code
	main proc
		sub    rsp, 20h           ;shadow stack

		mov    r9, rax            ;UINT uType
		lea    r8, msg_caption    ;LPCSTR lpCaption
		lea    rdx, msg_txt       ;LPCSTR lpText
		xor    rcx, rcx           ;HWND hWnd
		call   MessageBoxA

		add rsp, 20h              ;restore shadow stack

		mov rcx, rax              ;UNIT uExitCode
		call ExitProcess

		ret
	main endp
end
```

Congratulations! You just created your first hello world code in x64 Windows Assembly. Time for cake, again!

Let's take a moment and review. We first need to understand the x64 calling convention.

(VISIT https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160)

What we see here under the *Parameter passing* section is by default, the x64 calling convention passes the first four arguments to a function in registers. The registers used for these arguments depend on the position and type of the argument. Remaining arguments get pushed on the stack in right-to-left order.

Integer valued arguments in the leftmost four positions are passed in left-to-right order in RCX, RDX, R8, and R9, respectively. The fifth and higher arguments are passed on the stack as previously described. All integer arguments in registers are right-justified, so the callee can ignore the upper bits of the register and access only the portion of the register necessary.

Any floating-point and double-precision arguments in the first four parameters are passed in XMM0 - XMM3, depending on position. Floating-point values are only placed in the integer registers RCX, RDX, R8, and R9 when there are varargs arguments. For details, see Varargs. Similarly, the XMM0 - XMM3 registers are ignored when the corresponding argument is an integer or pointer type.

According to the x64 calling convention we need to provide a shadow stack for memory cells for each QWORD and the stack has to be aligned to 16 bytes for the next instruction.

The shadow space is the mandatory 32 bytes (4x8 bytes) which we must reserve for the called procedure. We provide 32 bytes on the stack before calling. This space can be left uninitialized.

In this calling convention, arguments after the 4th are pushed on the stack, which are on top of this shadow space (pushed before the 32 bytes).

We then setup and call our *MessageBoxA* Win32API again. We do not need to review the params as we have handled this earlier in our x86 example.

We then restore the shadow stack and then call *ExitProcess*.

At this point we can run our code by clicking on the green arrow next to the Local Windows Debugger.

HOORAY our hello world modal dialog box pops up.

This will be the only example where we write in all Assembly as I want to teach using the official Win32API which is natively in C++

however I wanted to first show you EXACTLY what is going on under the hood when it is in fact compiled.

Chapter 2: Debugging Hello World x86

Today we debug our Hello World x86 version within Ida Free. We first need to download Ida Free which is the free version of the most popular Ida Pro tool.

https://hex-rays.com/ida-free/#download

Once installed let's copy our **0x0001-hello_world-x86.exe**, which is inside the Debug folder within **0x0001-hello_world-x86** folder to a new folder called **0x0001-hello_world-x86-debug**.

After loading Ida Free, click Go Work on your own and drag-and-drop the **0x0001-hello_world-x86.exe** into it.

When the *Load a new file* modal pops up click *OK*.

When *The input file was linked with debug information* modal pops up select *Yes* as we will use the symbols in our reversing as we learn the Win32API.

Immediately it shows the disassembly and drops us into the *_main* function.

```
public _main
_main proc near

argc= dword ptr  4
argv= dword ptr  8
envp= dword ptr  0Ch

push    0               ; uType
lea     eax, msg_caption
push    eax             ; lpCaption
lea     eax, msg_txt
push    eax             ; lpText
push    0               ; hWnd
call    _MessageBoxA@16 ; MessageBoxA(x,x,x,x)
push    0               ; uExitCode
call    _ExitProcess@4  ; ExitProcess(x)
_main endp
```

Here we see a clean disassembly of our source as we wrote it in Assembly.

Let's first examine what is inside *msg_caption* so the first step is to double-click on the *msg_caption* text which will take us into the *.data* section of the code.

```
.data:0040400C ; CHAR msg_caption
.data:0040400C msg_caption     db 48h                 ; DATA XREF: _main+2↑o
.data:0040400D aElloWorldApp   db 'ello World App',0
```

Here we notice a strange *db 48h* at offset *4000* and another at offset 4001 of *db `ello World`,0*.

The first *48h* is ascii. Let's load up an ascii table and do some simple investigation.

https://www.asciitable.com

Here we see *0x48* or *48h* as *H*. This makes sense as our *msg_caption* begins with a capital *H*.

We are currently in the *IDA View*-A tab. Let's click on the *48h* value and the click on the *Hex View-1* tab to the right of *IDA View-A*.

```
00404000  48 65 6C 6C 6F 20 57 6F  72 6C 64 00 48 65 6C 6C  Hello·World.Hell
00404010  6F 20 57 6F 72 6C 64 20  41 70 70 00 00 00 00 00  o·World·App.....
```

Here we see our string represented in hex ascii. If we refer back to our table we can easily see how everything matches up. These letters, each representing a byte in the *.data* section are in fact the letters that will display in our *msg_caption*.

If we click back on the IDA View-A tab we can follow the same procedure and as the above images indicate we can see our *msg_txt* section as well following the same pattern.

Let's his the esc button and go back to our _main function.

Let's click on the first *push 0* instruction and hit f2 to set a breakpoint. You will notice a red box highlight that line.

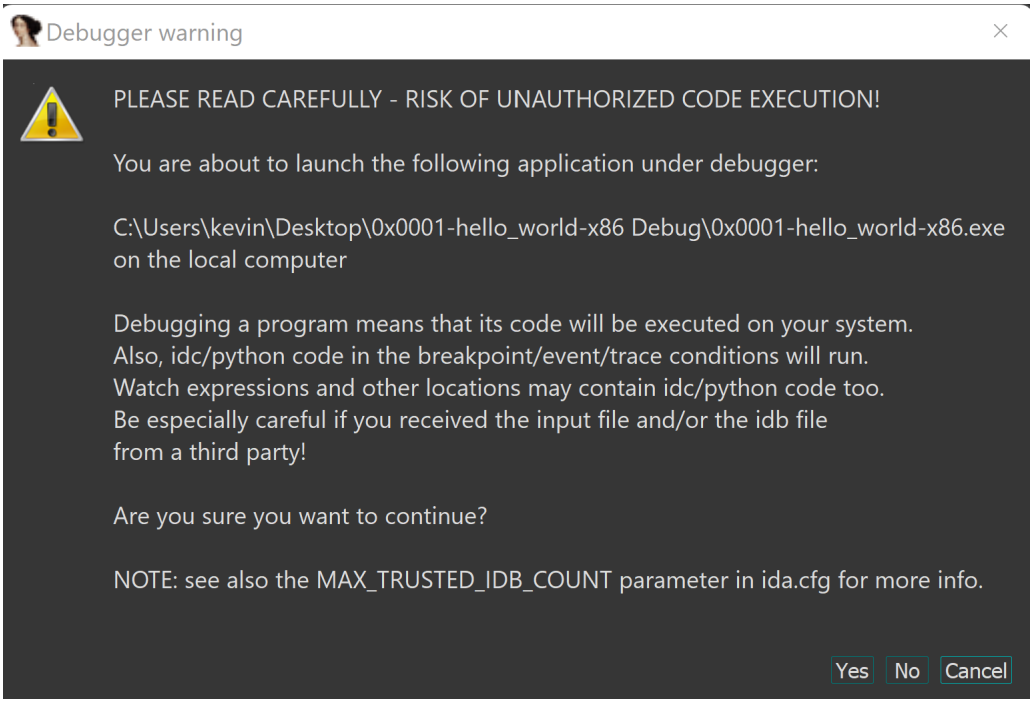
```
public _main
_main proc near

argc= dword ptr  4
argv= dword ptr  8
envp= dword ptr  0Ch

push    0               ; uType
lea     eax, msg_caption
push    eax             ; lpCaption
lea     eax, msg_txt
push    eax             ; lpText
push    0               ; hWnd
call    _MessageBoxA@16 ; MessageBoxA(x,x,x,x)
push    0               ; uExitCode
call    _ExitProcess@4  ; ExitProcess(x)
_main endp
```

When we click on the green *play* button next to *Local Windows Debugger* it will then begin the debugging session.

We immediately see a warning message as we are going to run the code dynamically however we wrote it so we can then click *Yes* at the bottom right.

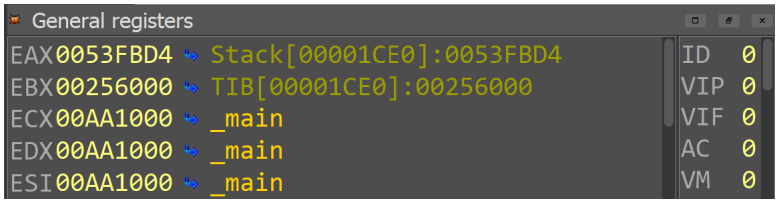


We see it load up our source code window which is quite handy as we can see that it broke on the *push 0* instruction.

Let’s ignore this window for now and click on the *IDA View-EIP* window to the left.
Here we see a number of different windows. We see our *Code* window.

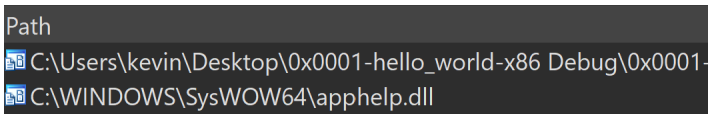
```
.text:00AA1000 _main proc near
.text:00AA1000
.text:00AA1000 argc= dword ptr  4
.text:00AA1000 argv= dword ptr  8
.text:00AA1000 envp= dword ptr  0Ch
.text:00AA1000
.text:00AA1000 push    0               ; uType
.text:00AA1002 lea     eax, msg_caption
.text:00AA1008 push    eax             ; lpCaption
.text:00AA1009 lea     eax, msg_txt
.text:00AA100F push    eax             ; lpText
.text:00AA1010 push    0               ; hWnd
.text:00AA1012 call    _MessageBoxA@16 ; MessageBoxA(x,x,x,x)
.text:00AA1017 push    0               ; uExitCode
.text:00AA1019 call    _ExitProcess@4  ; ExitProcess(x)
.text:00AA1019 _main endp
```

There is a *General registers* window.

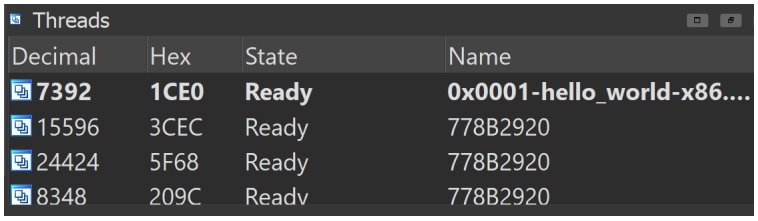


This is only a partial view of the registers as you have to scroll bars to work with. On the right hand side you see the values of the *eflags* register as it displays each bit.

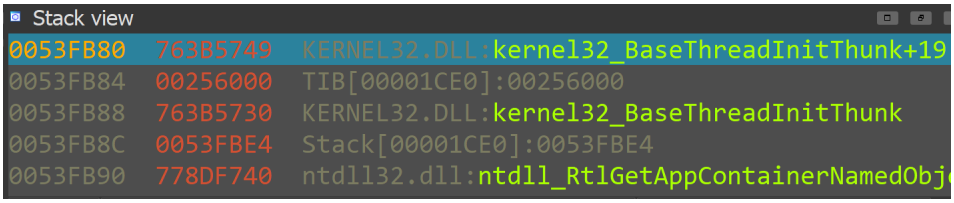
The next window is the *Modules* window which shows the application and all of the respective .dll libs it is using. Like the registers window you will need to scroll.



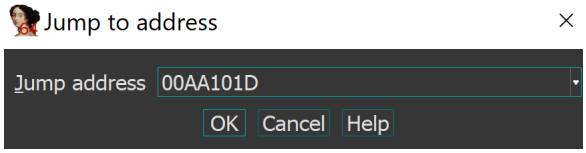
We have our *Threads* window.



We then have our *Stack view* window which the top of the stack is highlighted in blue. Like all of the others it is scrollable.



We have our *Hex View-1* window where if you type g within the window you can seek to that given memory address within the hex.



Let’s jump to *00aa101d* and look at the *Hex View-1*.

```
Hex View-1
00AA1000  6A 00 8D 05 0C 40 AA 00  50 8D 05 00 40 AA 00 50  j....@ª.P...@ª.P
00AA1010  6A 00 E8 14 00 00 00 6A  00 E8 07 00 00 00 CC CC  j.è....j.è....ÌÌ
00AA1020  CC CC CC CC CC FF 25 00  50 AA 00 FF 25 30 50 AA  ÌÌÌÌÌÿ%.Pª.ÿ%0Pª
00AA1030  00 CC CC CC CC CC CC CC  CC CC CC CC CC CC CC CC  .ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
00AA1040  CC CC CC CC CC CC CC CC  CC CC CC CC CC CC CC CC  ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
```

Finally we have our *Output* window.

```
Output
75F10000: loaded C:\WINDOWS\SysWOW64\IMM32.DLL
PDBSRC: loading symbols for 'C:\Users\kevin\Desktop\0x0001-hello_world-x86 Debug\0x0001-hello_world-x86.exe'...
PDB: using PDBIDA provider
PDB: loading C:\Users\kevin\Documents\Hacking-Windows\0x0001-hello_world-x86\Debug\0x0001-hello_world-x86.pdb
PDB: There is no type information
PDB: There is no IPI stream
IDC
AU:  idle  Down Disk: 41GB
```

Let’s step through the code. Let’s enable the debugger menu.

View – Toolbars – Debugger commands

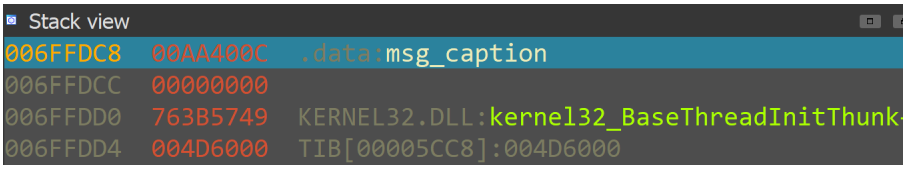
Let’s click on the first blue icon with the two arrows to single-step. Let’s single-step twice.

We are now about to execute the first *push eax* instruction. We see *msg_caption* moved into *eax*. Before we step take note of the *Stack view* window as well.

```
.text:00AA1000 argv= dword ptr  8
.text:00AA1000 envp= dword ptr  0Ch
.text:00AA1000
.text:00AA1000 push    0               ; uType
.text:00AA1002 lea     eax, msg_caption
.text:00AA1008 push    eax             ; lpCaption
.text:00AA1009 lea     eax, msg_txt
.text:00AA100F push    eax             ; lpText
.text:00AA1010 push    0               ; hWnd
.text:00AA1012 call    _MessageBoxA@16 ; MessageBoxA(x,x,x,x)
.text:00AA1017 push    0               ; uExitCode
.text:00AA1019 call    _ExitProcess@4  ; ExitProcess(x)
.text:00AA1019 _main endp
```

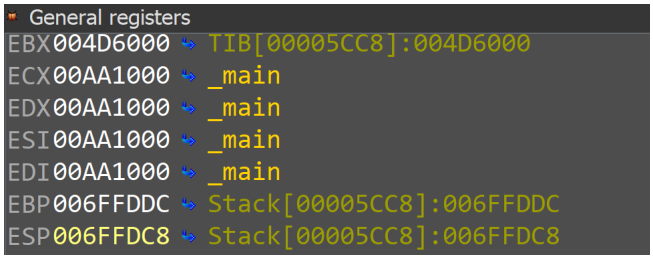
```
General registers
EAX 00AA400C ↳ .data:msg_caption
EBX 004D6000 ↳ TIB[00005CC8]:004D6000
ECX 00AA1000 ↳ _main
EDX 00AA1000 ↳ _main
ESI 00AA1000 ↳ _main
```

Now let’s step again. Let’s now examine the stack.



We see the msg_caption moved to the top of the stack as it was just pushed from eax.

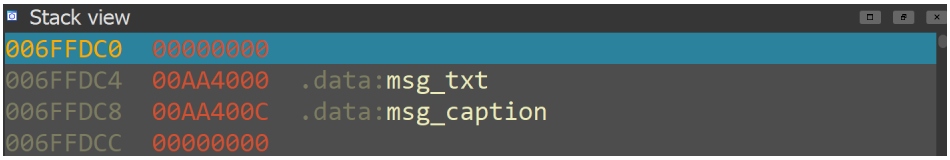
Take immediate note of the value in esp as that is the top of the stack.



Let’s step and stop right before the call.

```
.text:00AA1000 public _main
.text:00AA1000 _main proc near
.text:00AA1000
.text:00AA1000 argc= dword ptr  4
.text:00AA1000 argv= dword ptr  8
.text:00AA1000 envp= dword ptr  0Ch
.text:00AA1000
.text:00AA1000 push    0               ; uType
.text:00AA1002 lea     eax, msg_caption
.text:00AA1008 push    eax             ; lpCaption
.text:00AA1009 lea     eax, msg_txt
.text:00AA100F push    eax             ; lpText
.text:00AA1010 push    0               ; hWnd
.text:00AA1012 call    _MessageBoxA@16 ; MessageBoxA(x,x,x,x)
.text:00AA1017 push    0               ; uExitCode
.text:00AA1019 call    _ExitProcess@4  ; ExitProcess(x)
.text:00AA1019 _main endp
```

At this point take careful note on the Stack view.



It is CRITICAL that you take SPECIAL CARE to review the *Code* window above and compare it to the *Stack view* window.

Notice that the top of the stack, in this case *0x006ffdc0* holds the value of *0* which was the LAST, most recent value pushed to the stack.

Remember that the STACK GROWS DOWN in memory. The value of *ebp* which is the stack base pointer is HIGHER in memory as compared to *esp*. Please write this down.

As we push more items onto the stack *esp* will continue to grow DOWNWARD in memory and therefore the gap between *ebp* and *esp* grows larger as *esp* is growing downward toward the heap until either call occurs which will collapse the stack frame (*ebp* to *esp*) OR a pop operation will pop the value in *esp* into whatever you are popping it into and therefore moving esp UPWARD in memory.

At the +4 offset we see *msg_txt* which was the 2nd to the last thing pushed onto the stack.

At the +8 offset we see *msg_caption* was the 3rd to the last thing pushed onto the stack.

Finally at +12 or +0xc we see *0* which was the 4th to the last thing pushed onto the stack.

We can step over the call to _MessageBoxA@16 and it will load our modal window.

We can then step over the call to _ExitProcess@4 and it will terminate our binary.

If you single-step it will take you through the internal Win32API functions if you wanted to get a greater appreciation of what exactly is happening when these functions are in fact called.

When we continue execution we will see our program run and we now have a complete idea of how this simple programs works as we did a complete dynamic reversing analysis on this binary.