



FIRST EDITION – CHAPTER 8 REV 1

Kevin Thomas
Copyright © 2021 My Techno Talent

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

Chapter 2: Debugging Hello World x86

Chapter 3: Hacking Hello World x86

Chapter 4: Debugging Hello World x64

Chapter 5: Hacking Hello World x64

Chapter 6: Directories

Chapter 7: Debugging Directories x86

Chapter 8: Hacking Directories x86

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 box 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.

HOOORAY 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 PostQuitMessage: 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

        xor    rcx, rcx           ;int nExitCode
        call   PostQuitMessage

        ret
    main endp
end
```

We also see a call to PostQuitMessage which has an int nExitCode as a param.

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

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 ; MessageBox(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 aEHelloWorldApp db 'ello World App',0
```

In the *msg_text* we also 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 hit 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.

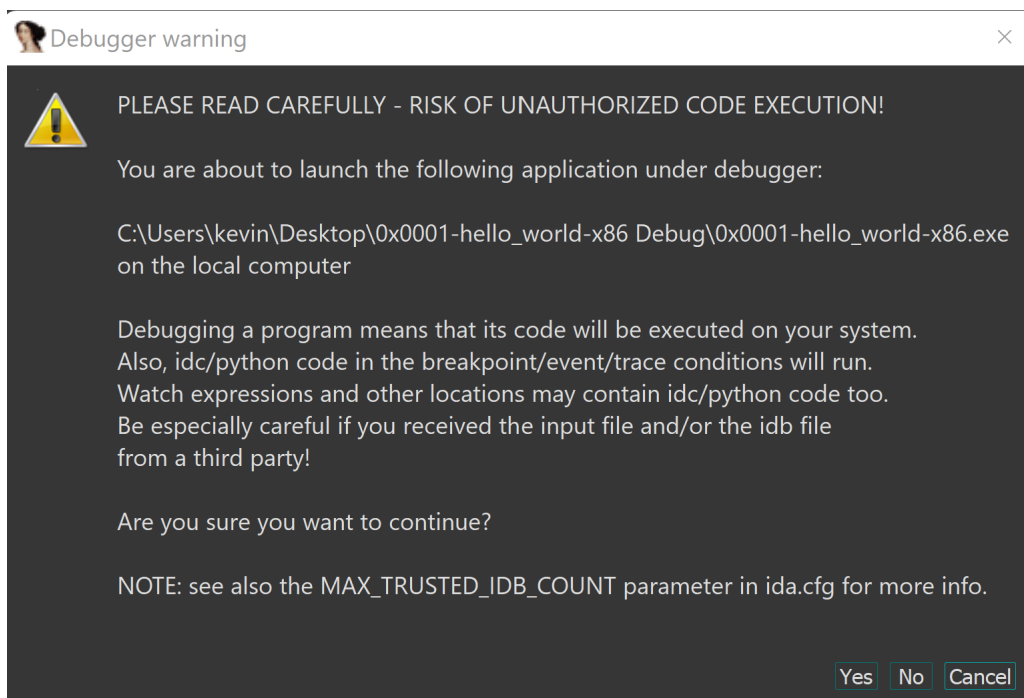
```
public _main
_main proc near

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.



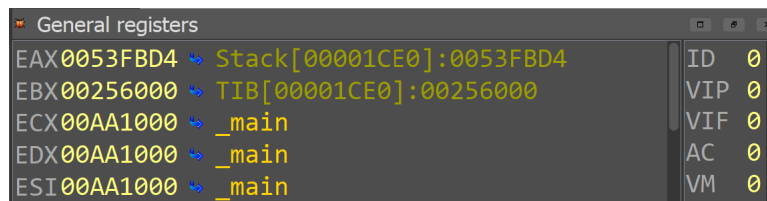
We see it load up our source code window which is quite handy as we can see that it broke on the *push 0* instruction.

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.

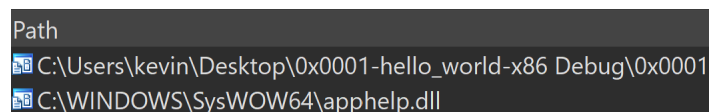
```
.text:00AA1000 _main proc near
.text:00AA1000
.text:00AA1000 argc= dword ptr 4
.text:00AA1000 argv= dword ptr 8
.text:00AA1000 envp= dword ptr 0Ch
.text:00AA1000
.text:00AA1000 push 0 ; uType
.text:00AA1002 lea eax, msg_caption
.text:00AA1008 push eax ; lpCaption
.text:00AA1009 lea eax, msg_txt
.text:00AA100F 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.

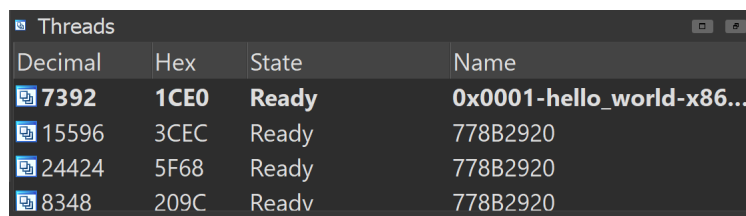


This is only a partial view of the registers as you have to scroll bars to work with. On the right hand side you see the values of the *eflags* register as it displays each bit.

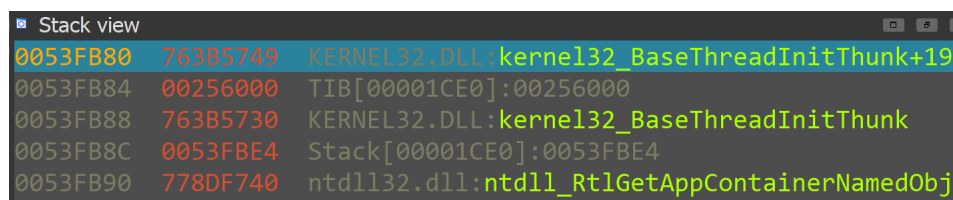
The next window is the *Modules* window which shows the application and all of the respective .dll libs it is using. Like the registers window you will need to scroll.



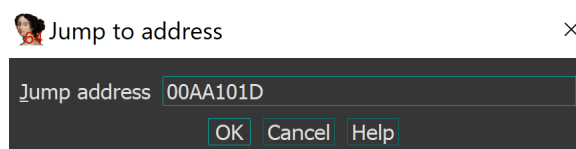
We have our *Threads* window.



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....@.P...@.P
00AA1010  6A 00 E8 14 00 00 00 6A 00 E8 07 00 00 00 CC CC  j.è....j.è....ïï
00AA1020  CC CC CC CC CC FF 25 00 50 AA 00 FF 25 30 50 AA  ïïïïÿ%.P.ÿ%P.
00AA1030  00 CC CC CC CC CC CC CC CC CC CC CC CC CC CC  .ïïïïïïïïïïïïïï
00AA1040  CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC  ïïïïïïïïïïïïïï
```

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 DownDisk: 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 ; MessageBox(x,x,x,x)
.text:00AA1017  push 0 ; uExitCode
.text:00AA1019  call _ExitProcess@4 ; ExitProcess(x)
.text:00AA1019  _main endp
```

```
General registers
EAX00AA400C  ↳ .data:msg_caption
EBX004D6000  ↳ TIB[00005CC8]:004D6000
ECX00AA1000  ↳ _main
EDX00AA1000  ↳ _main
ESI00AA1000  ↳ _main
```

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

```
Stack view
006FFDC8 00AA400C .data:msg_caption
006FFDCC 00000000
006FFDD0 763B5749 KERNEL32.DLL:kernel32_BaseThreadInitThunk
006FFDD4 004D6000 TIB[00005CC8]:004D6000
```

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.

```
General registers
EBX004D6000 ↪ TIB[00005CC8]:004D6000
ECX00AA1000 ↪ _main
EDX00AA1000 ↪ _main
ESI00AA1000 ↪ _main
EDI00AA1000 ↪ _main
EBP006FFDDC ↪ Stack[00005CC8]:006FFDDC
ESP006FFDC8 ↪ Stack[00005CC8]:006FFDC8
```

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 ; MessageBox(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.

```
Stack view
006FFDC0 00000000
006FFDC4 00AA4000 .data:msg_txt
006FFDC8 00AA400C .data:msg_caption
006FFDCC 00000000
```

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.

Chapter 3: Hacking Hello World x86

Today we hack our Hello World x86 version within Ida Free. Let's fire up our session in Ida Free and begin.

We start with our `_main` proc.

```
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 ; MessageBox(x,x,x,x)
push 0 ; uExitCode
call _ExitProcess@4 ; ExitProcess(x)
_main endp
```

Double-click on `msg_caption`.

```
; CHAR msg_txt
msg_txt db 48h ; DATA XREF: _main+9↑o
aEloWorld db 'ello World',0
; CHAR msg_caption
msg_caption db 48h ; DATA XREF: _main+2↑o
aEloWorldApp db 'ello World App',0
align 1000h
_data ends
```

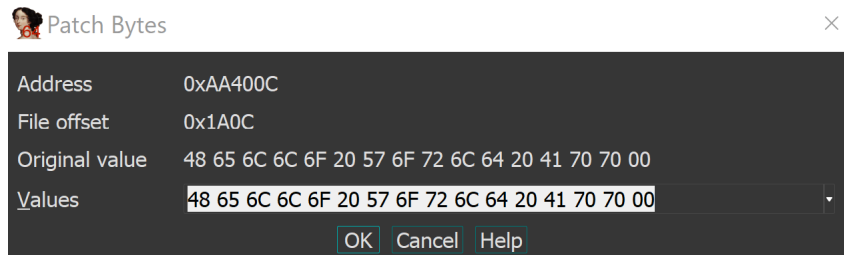
Click on the *Hex View-1* tab.

```
00AA4000 48 65 6C 6C 6F 20 57 6F 72 6C 64 00 48 65 6C 6C Hello·World·Hell
00AA4010 6F 20 57 6F 72 6C 64 20 41 70 70 00 00 00 00 00 o·World·App·....
```

We noticed in the last chapter that `0x48` begins the string as we know in the ascii table that `0x48` is in fact `'H'`.

<https://www.asciitable.com>

Click *Edit – Patch program – Change byte ...*



48 65 6C 6C 6F 20 57 6F 72 6C 64 20 41 70 70 00

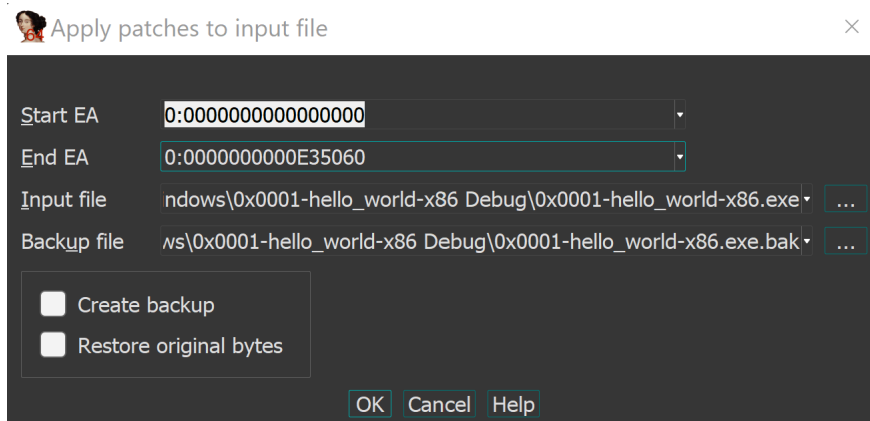
Let's change the caption to 'Hacky World'.

48 61 63 6B 79 20 57 6F 72 6C 64 20 41 70 70 00

Click OK.

Click *Edit - Patch program - Apply patches to input file...*

Click OK.

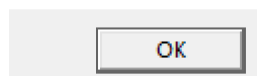


Click the green play button. We notice two warning windows which we can ignore stating that the binary has changed.

We broke on our first break point. Let's hit the play button again.

Hacky World App ×

Hello World



Hooray! Time for cake! We saw that we were able to successfully hack our *msg_caption* correctly.

You could also take it a step further and hack the actual *msg_txt* if you so chose.

This is the first of many small hacks. The purpose of this book it to take SMALL steps. Take very careful analysis on exactly what is happening at the assembly level and understanding have to have absolute control over the process.

Chapter 4: Debugging Hello World x64

Today we debug our Hello World x64 version within Ida Free.

Let's copy our **0x0001-hello_world-x64.exe**, which is inside the **Debug** folder within **0x0001-hello_world-x64** folder to a new folder called **0x0001-hello_world-x64-debug**.

After loading Ida Free, click Go Work on your own and drag-and-drop the **0x0001-hello_world-x64.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.

```
main proc near
sub     rsp, 20h
mov     r9, rax           ; uType
lea     r8, msg_caption   ; lpCaption
lea     rdx, msg_txt       ; lpText
xor     rcx, rcx          ; hWnd
call    MessageBoxA_0
add     rsp, 20h
xor     rcx, rcx          ; nExitCode
call    PostQuitMessage_0
retn
main endp
```

Take note and re-read Chapter 2. Unlike x86 where we push params to the stack we are moving the params into *rcx*, *rdx*, *r8*, *r9*. This is how x64 handles their function calls at the Assembly level.

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:000000014000400C ; CHAR msg_caption
..data:000000014000400C msg_caption db 48h ; DATA XREF: main+710
..data:000000014000400D aEHelloWorldApp db 'ello World App',0
```

In the *msg_text* we also 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 then click on the `Hex View-1` tab to the right of `IDA View-A`.

```
0000000140004000 48 65 6C 6C 6F 20 57 6F 72 6C 64 00 48 65 6C 6C Hello·World·Hell
0000000140004010 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.

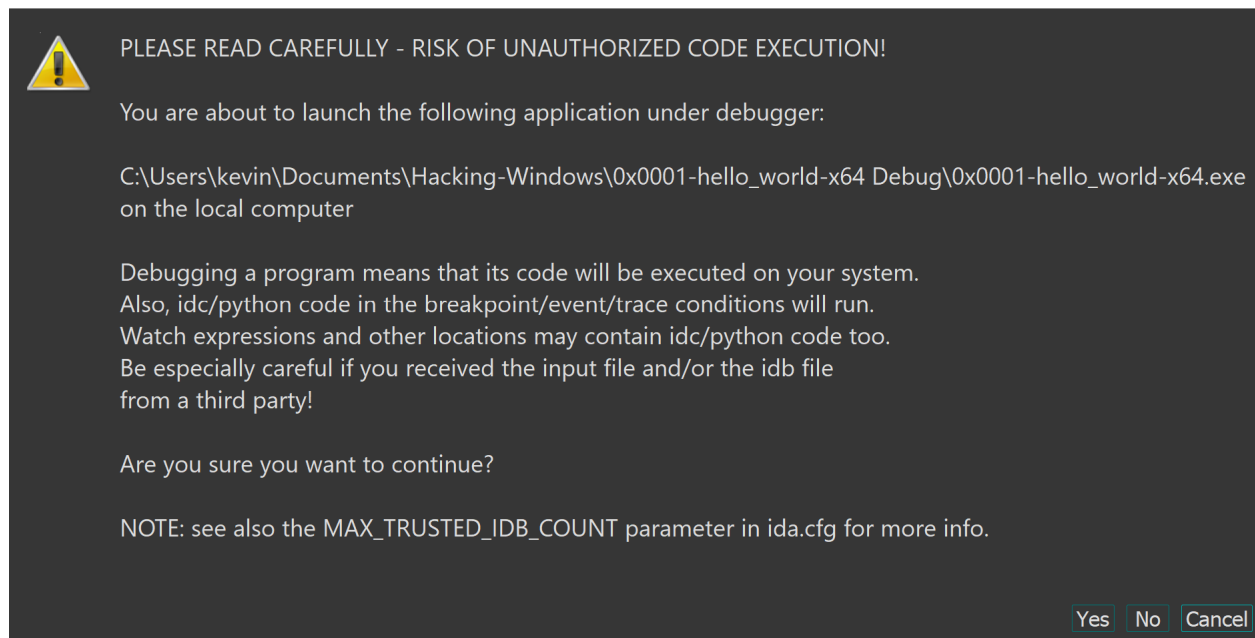
Let's hit the `esc` button and go back to our main function.

Let's click on the `mov r9, rax` instruction and hit `f2` to set a breakpoint. You will notice a red box highlight that line.

```
main proc near
sub     rsp, 20h
mov     r9, rax      ; uType
lea     r8, msg_caption ; lpCaption
lea     rdx, msg_txt   ; lpText
xor     rcx, rcx      ; hWnd
call    MessageBoxA_0
add     rsp, 20h
xor     rcx, rcx      ; nExitCode
call    PostQuitMessage_0
retn
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. As with the x86 version as we wrote this in Assembly we can ignore and click on the *IDA View-RIP* tab.

Enable 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 once.

We see the value of *rax* moved into *r9* which holds the value of *start*. This is the fourth param in reverse order.

This is likely a compiler optimization as we did not code this in Assembly.

```
R9 00007FF688F21005 ↗ start
```

We then load the effective address of *msg_caption* into *r8* the third param in reverse order after we step again.

```
R8 00007FF688F2400C ↗ .data:msg_caption
```

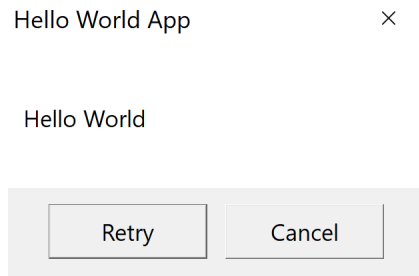
We then load the effective address of *msg_txt* into *rdx* the second param in reverse order after we step again.

```
RDX00007FF688F24000 ↗ .data:msg_txt
```

We then zero out or *xor rcx, rcx* to put a 0 in *rcx*.

```
RCX0000000000000000 ↗
```

Finally we call *MessageBoxA_0* and display our caption and message.



We then called *PostQuitMessage_0* and exit the program.

Chapter 5: Hacking Hello World x64

Today we hack our Hello World x64 version within Ida Free. Let's fire up our session in Ida Free and begin.

We start with our *main* proc.

```
main proc near
sub     rsp, 20h
mov     r9, rax      ; uType
lea     r8, msg_caption ; lpCaption
lea     rdx, msg_txt   ; lpText
xor     rcx, rcx      ; hWnd
call    MessageBoxA_0
add     rsp, 20h
xor     rcx, rcx      ; nExitCode
call    PostQuitMessage_0
retn
main endp
```

Double-click on *msg_caption*.

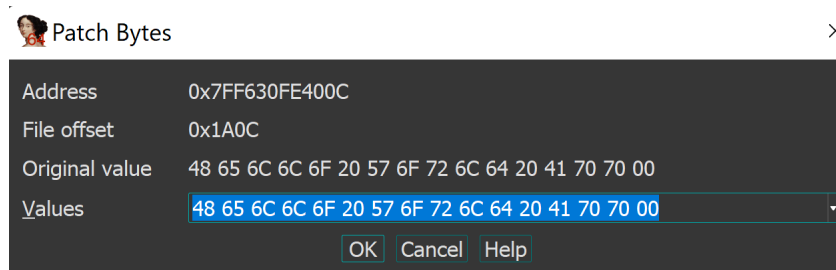
```
; CHAR msg_txt
msg_txt      db 48h          ; DATA XREF: main+Eto
aElloWorld   db 'ello World',0
; CHAR msg_caption
msg_caption  db 48h          ; DATA XREF: main+7to
aElloWorldApp db 'ello World App',0
align 1000h
_data       ends
```

Click on the Hex View-1 tab.

```
07FF630FE400  48 65 6C 6C 6F 20 57 6F 72 6C 64 00 48 65 6C 6C  Hello·World·Hell
07FF630FE4010  6F 20 57 6F 72 6C 64 20 41 70 70 00 00 00 00 00  o·World·App·....
```

We know from our prior chapters that `0x48` is 'H' and the other bytes are the additional letters.

Click *Edit - Patch program - Change byte ...*



48 65 6C 6C 6F 20 57 6F 72 6C 64 20 41 70 70 00

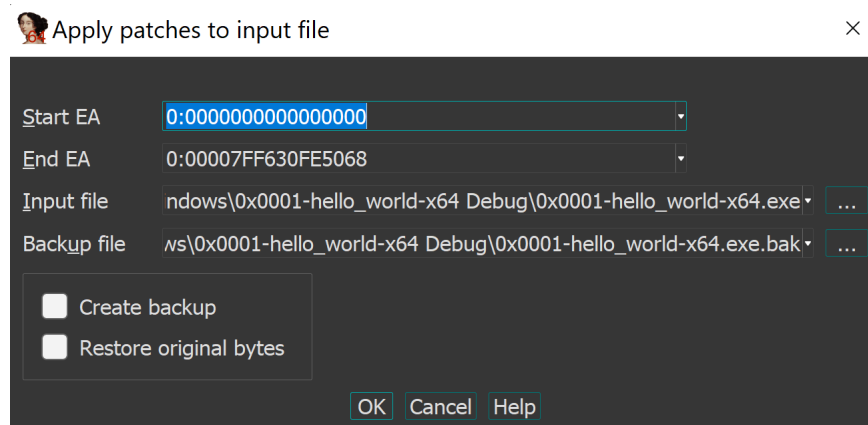
Let's change the caption to 'Hacky World'.

48 61 63 6B 79 20 57 6F 72 6C 64 20 41 70 70 00

Click OK.

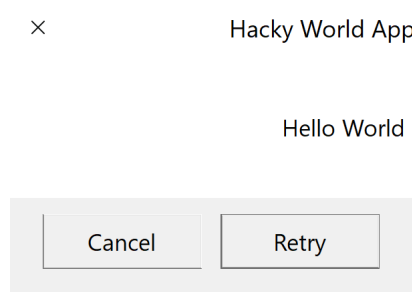
Click *Edit - Patch program - Apply patches to input file...*

Click OK.



Click the green play button. We notice two warning windows which we can ignore stating that the binary has changed.

We broke on our first break point. Let's hit the play button again.



Hooray! As in the previous hacking chapter you can further hack anything you wish. We are doing nothing more than taking small bite-sized building blocks so you have a full understanding of the Win32API.

Chapter 6: Directories

We continue with a simple app that handles Windows directory manipulation by creating and removing a directory.

Let's create a new project

Create a new project

Empty Project

Next

Project name: 0x0006-directories

CHECK Place solution and project in the same directory

Create

RT CLICK on the 0x0006-directories in Solutions Explorer

Add

New Item...

main.c

OK

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

```
#include <stdio.h>
#include <Windows.h>

int main(void)
{
    BOOL bDir;

    bDir = CreateDirectory(
        L"C:\\mydir",
        NULL
    );
    if (bDir == FALSE)
    {
        printf("CreateDirectory failed & error no %u!\n", GetLastError());
    }
    else
    {
        printf("CreateDirectory Success!\n");
    }

    bDir = RemoveDirectory(
        L"C:\\mydir"
    );
    if (bDir == FALSE)
    {
        printf("RemoveDirectory failed & error no %u!\n", GetLastError());
    }
    else
    {
        printf("RemoveDirectory Success!\n");
    }
}
```

```
    return 0;
}
```

Let's review the *CreateDirectoryW* API below.

(VISIT <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createdirectoryw>)

REMEMBER if you hover over *CreateDirectory* it expands to *CreateDirectoryW* in Visual Studio. This means *CreateDirectory* is an alias for *CreateDirectoryW*.

We see we have two params which are *lpPathName* which is the path of the directory to be created and *lpSecurityAttributes* which is a pointer to a *SECURITY_ATTRIBUTES* structure. In our case we are just using *NULL*.

The return value is non-zero if the function succeeds otherwise it will return the code *ERROR_ALREADY_EXISTS* or *ERROR_PATH_NOT_FOUND*.

Let's review the *RemoveDirectoryW* API below.

(VISIT <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-removedirectoryw>)

We see we have one param *lpPathName* which is the path of the directory to be created.

The return value is non-zero if the function succeeds otherwise it will return 0 and any relevant error information inside *GetLastError*.

When we run the program it shows the following output.

```
CreateDirectory Success!
RemoveDirectory Success!

C:\Users\kevin\Documents\Hacking-Windows\0x0006-directories\0x0006-directories\Debug\0x0006-
directories.exe (process 10204) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging-
>Automatically close the console when debugging stops.
Press any key to close this window . . .
```

In our next chapter we will debug this program in x86.

Chapter 7: Debugging Directories x86

We are going to debug the 32-bit version of our directories program.

Since we have created a few projects together I assume you know what you are doing in IDA at this point. If this process is unfamiliar to you please re-read the prior chapters.

In the IDA View-A text view we first see our *CreateDirectoryW* function.

```
.text:00621887      push     0                ; lpSecurityAttributes
.text:00621889      push     offset PathName ; "C:\\mydir"
.text:0062188E      call    ds:__imp__CreateDirectoryW@8 ; CreateDirectoryW(x,x)
```

In our last chapter we reviewed the API in C. Here we first push the *lpSecurityAttributes* param to the stack followed by the *PathName* param and then we call the function.

Let's set a breakpoint directly after the call and run the Local Windows debugger.

NOTICE we see that our **mydir** folder has been .

Let's stop execution and delete our breakpoint.

We then see our *RemoveDirectoryW* function.

```
.text:006218D2      push     offset PathName ; "C:\\mydir"
.text:006218D7      call    ds:__imp__RemoveDirectoryW@4 ; RemoveDirectoryW(x)
```

Here we see the first param of *PathName* and then the call.

Let's set a breakpoint directly after the call and run the Local Windows debugger.

NOTICE we see that our **mydir** folder has been deleted.

Let's stop execution and delete our breakpoint.

The flow of this series now that we have a basic familiarity with IDA will be a simple reversing of the binary such that we continue to reinforce how each Windows API looks like in both 32-bit and 64-bit Assembly as this will help us get a firm grasp on what is going on under the hood with any Windows binary.

I won't often keep repeating myself however I wanted to at this stage have a small retrospective.

There are TONS of good reversing resources out there however my aim is to take SMALL Win32 API's and reverse them step-by-step so that in the real world when you are dealing with obfuscated Windows binaries which might have dynamic resolution based on a complicated hash you will recognize patters that you may not have without going through these exercises.

Taking time and getting your hands dirty on these small but digestible exercises will help you master the domain!

In our next chapter we will hack this program in x86.

Chapter 8: Hacking Directories x86

We are going to hack the 32-bit version of our directories program.

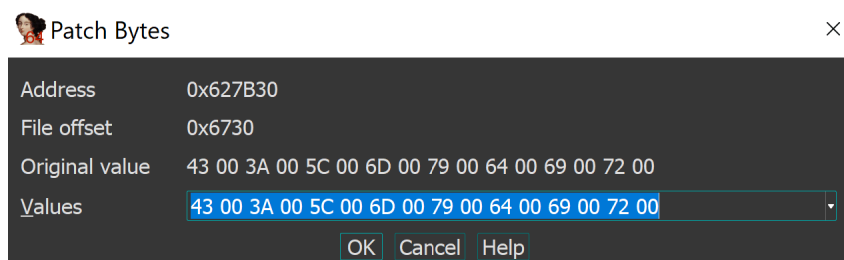
In this chapter we will hack the directory name this will continue to build our experience on custom hacking binaries.

```
.text:00621889      push     offset PathName ; "C:\\mydir"
```

Here we see the *PathName* of "C:\\mydir". Double-click to get to the *.rdata* section.

```
.rdata:00627B30      text     "UTF-16LE", 'C:\\mydir',0
```

Click *Edit - Patch program - Change byte ...*



43 00 3A 00 5C 00 6D 00 79 00 64 00 69 00 72 00

Let's change the path to 'hacky'.

43 00 3A 00 5C 00 68 00 61 00 63 00 6b 00 79 00

Click OK.

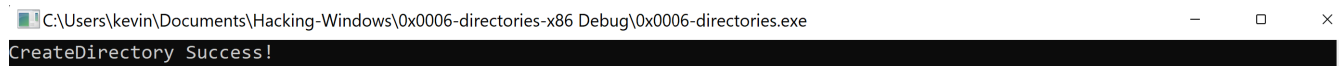
Click *Edit - Patch program - Apply patches to input file...*

Click OK.

Let's set a breakpoint on the next instruction after the call to *printf* indicating the *CreateDirectory Success!* Message.

```
.text:003B18C3      push     offset aCreatedirector_0 ; "CreateDirectory Success!\n"
.text:003B18C8      call     j__printf
.text:003B18CD      add     esp, 4
```

Click the green play button. We see the terminal indicating our CreateDirectory has been called successfully.



Let's look at the root of our hard drive.



Hooray! We have hacked our simple program and altered the creation of the directory name.

As I have said before these are small bite-sized lessons that help you to code, debug and hack in addition to researching each of the Win32API functions so we have a mastery of the process.

In our next chapter we will debug this program in x64.