FIRST EDITION

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)

Our first Win32API that we call is the MessageBoxA which provides a Windows Message Box to appear. We then set up a *flat* memory model which uses no combined segment or offset addressing. We also use the *stdcall* Win32 calling convention which we push args in reverse order onto the stack and then call the procedure. The *calle* clears the stack after the call.

Our second 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.

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
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)

The *Microsoft x64 calling convention, fastcall*, is what we use in x64. 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. The *caller* cleans up the stack after the call.

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
```

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 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.

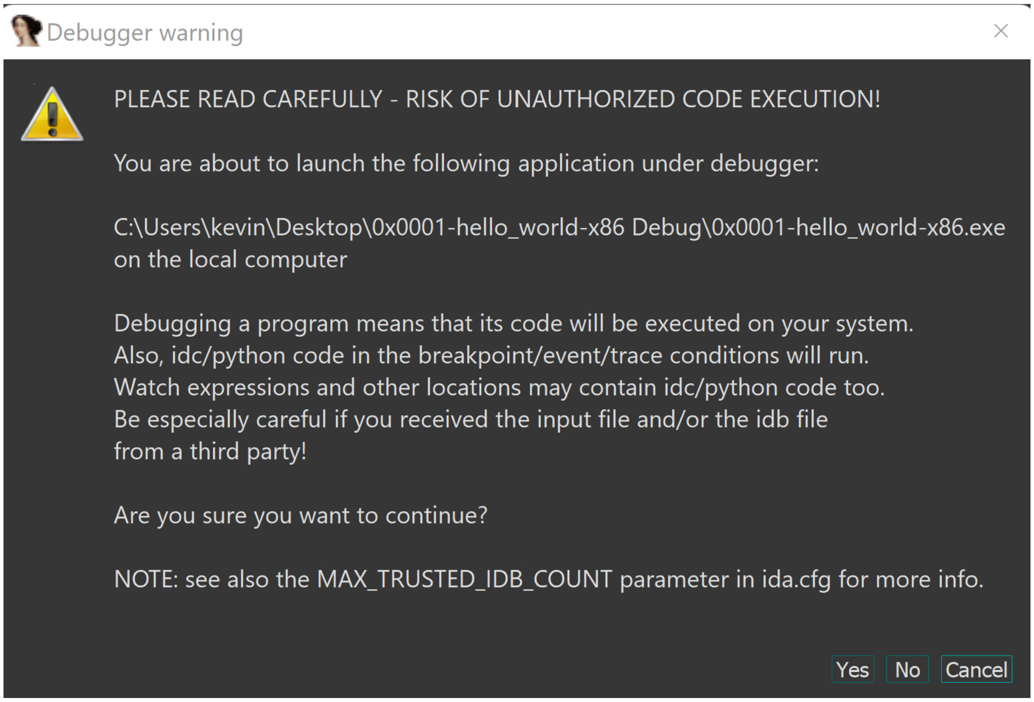
```
public _main
_main proc near

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.

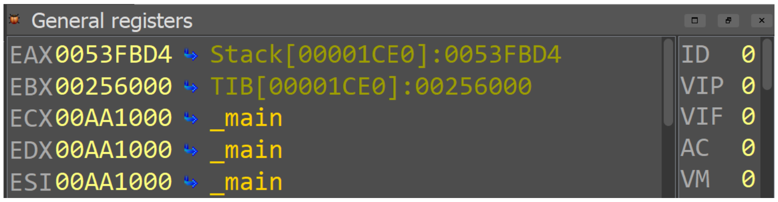


We see it load up our source code window which is quite handy as we can see that it broke on the *push 0* instruction.

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.

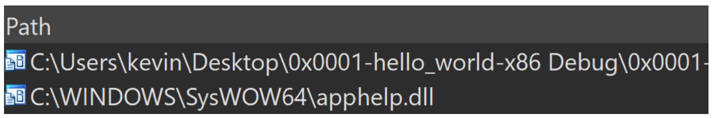
```
.text:00AA1000 _main proc near
.text:00AA1000
.text:00AA1000 argc= dword ptr  4
.text:00AA1000 argv= dword ptr  8
.text:00AA1000 envp= dword ptr  0Ch
.text:00AA1000
.text:00AA1000 push    0                   ; uType
.text:00AA1002 lea     eax, msg_caption
.text:00AA1008 push    eax              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

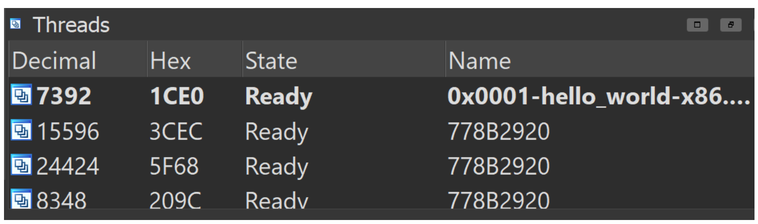


This is only a partial view of the registers as you have to scroll bars to work with. On the right hand side you see the values of the *eflags* register as it displays each bit.

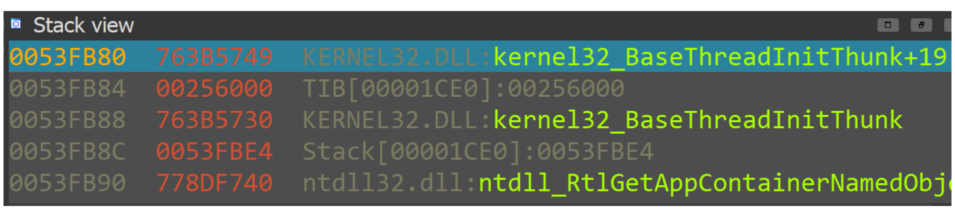
The next window is the *Modules* window which shows the application and all of the respective .dll libs it is using. Like the registers window you will need to scroll.



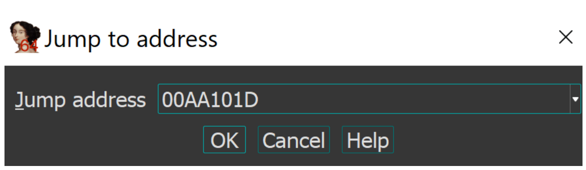
We have our *Threads* window.



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

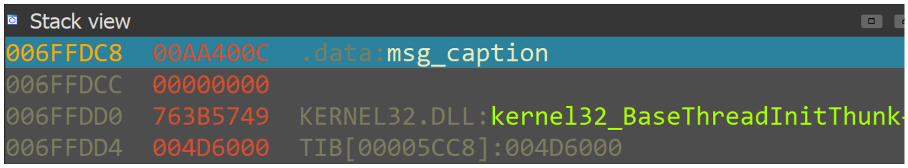
Let’s click on the first blue icon with the two arrows to single-step. Let’s single-step twice.

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
```

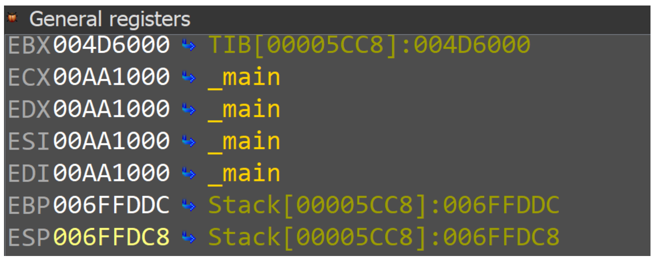
```
General registers
EAX 00AA400C ↳ .data:msg_caption
EBX 004D6000 ↳ TIB[00005CC8]:004D6000
ECX 00AA1000 ↳ _main
EDX 00AA1000 ↳ _main
ESI 00AA1000 ↳ _main
```

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

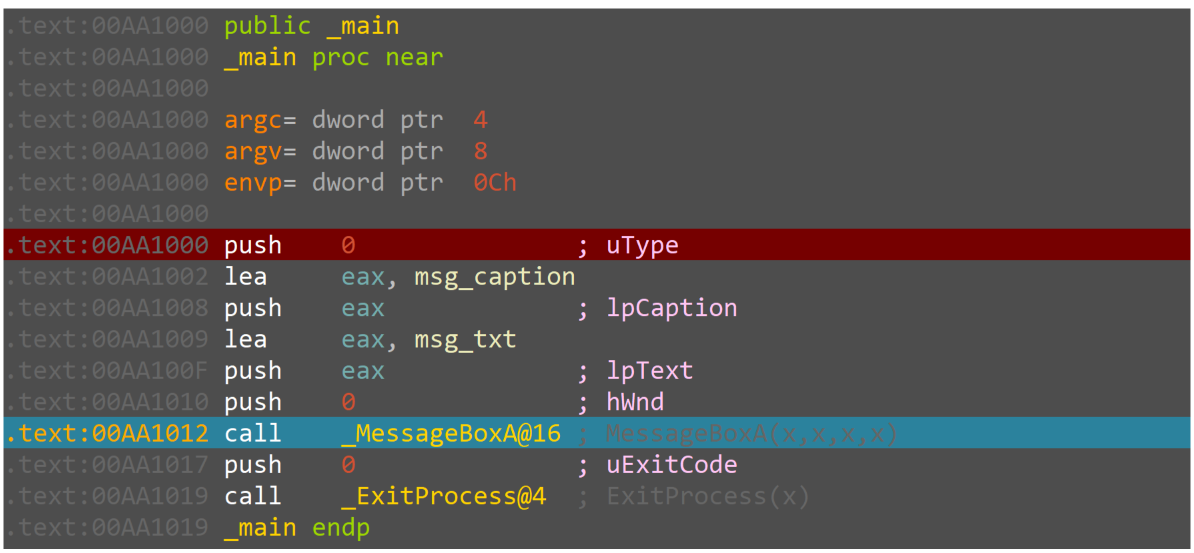


We see the msg_caption moved to the top of the stack as it was just pushed from eax.

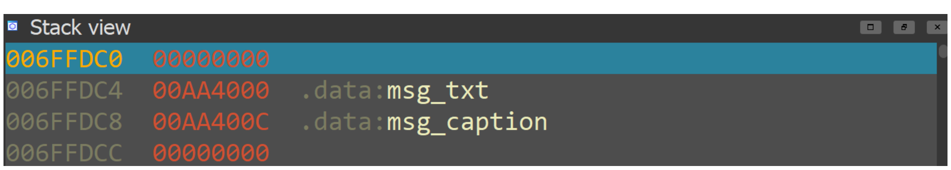
Take immediate note of the value in esp as that is the top of the stack.



Let’s step and stop right before the call.



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.

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 ; MessageBoxA(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
aElloWorld      db 'ello World',0
; CHAR msg_caption
msg_caption     db 48h                  ; DATA XREF: _main+2↑o
aElloWorldApp   db 'ello World App',0
                align 1000h
_data           ends
```

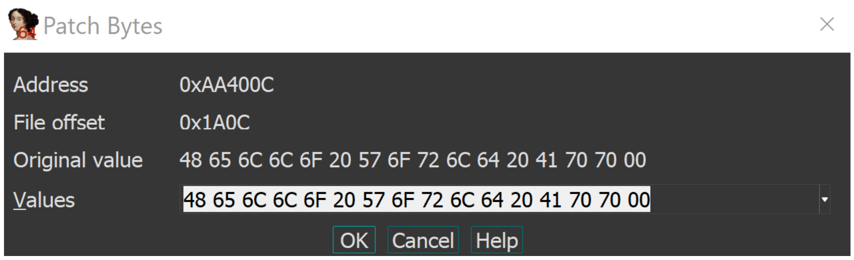
Click on the *Hex View-1* tab.

```
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

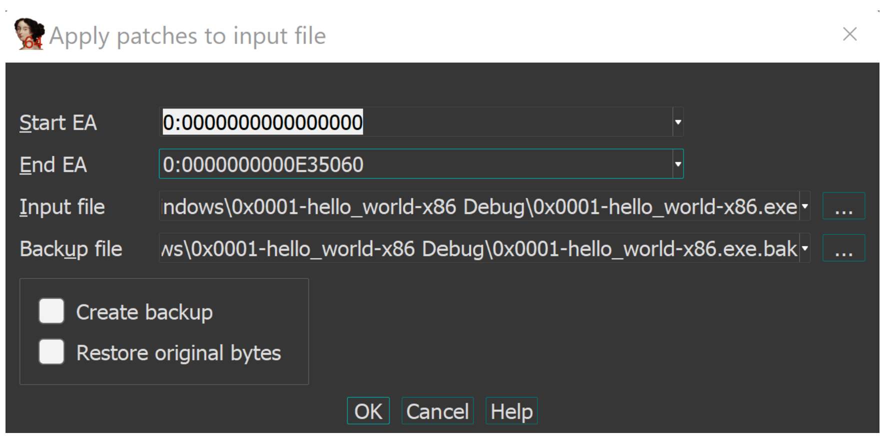
Let’s change the caption to ‘Hacky World’.

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.

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+7↑o
.data:000000014000400D aElloWorldApp   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*.

```
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.

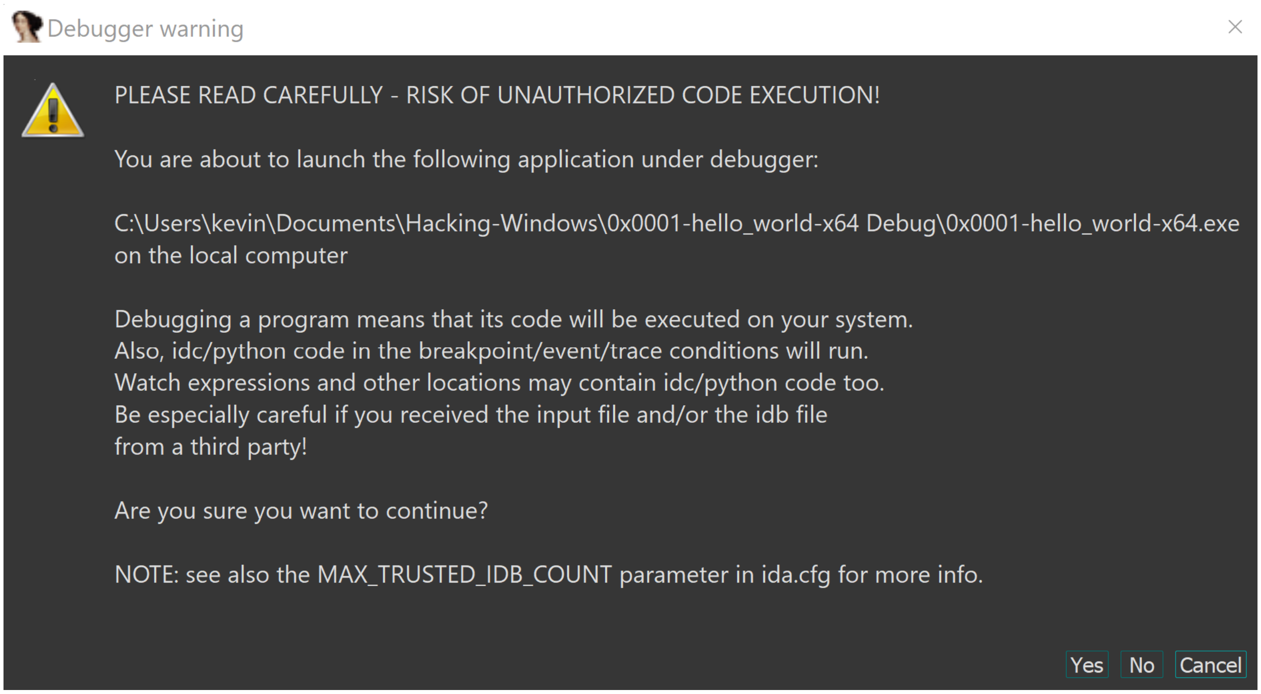
Let's his the esc button and go back to our main function.

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.

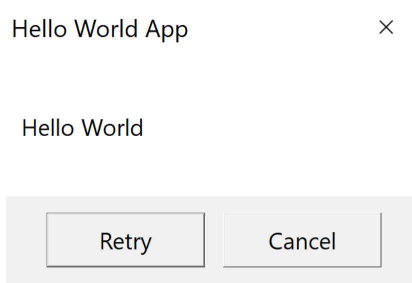
This is likely a compiler optimization as we did not code this in Assembly.

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

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

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

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*.

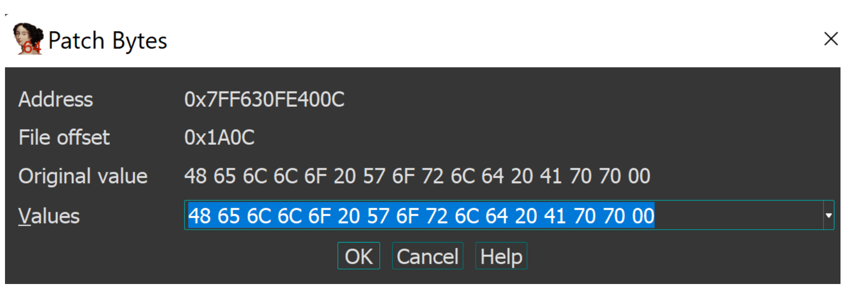
```
; CHAR msg_txt
msg_txt         db 48h                  ; DATA XREF: main+E↑o
aElloWorld      db 'ello World',0
; CHAR msg_caption
msg_caption     db 48h                  ; DATA XREF: main+7↑o
aElloWorldApp   db 'ello World App',0
                align 1000h
_data           ends
```

Click on the Hex View-1 tab.

```
07FF630FE4000  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

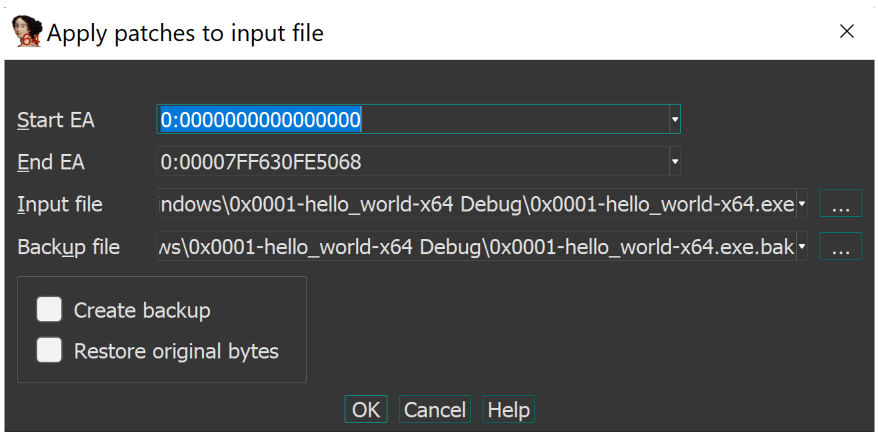
Let’s change the caption to ‘Hacky World’.

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

Click OK.

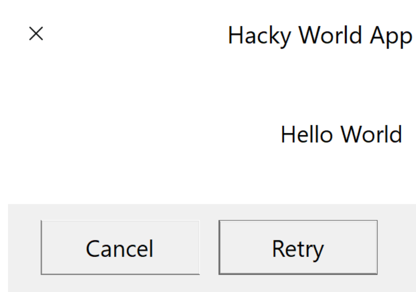
Click *Edit – Patch program – Apply patches to input file…*

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 %ul\n", GetLastError());
    }
    else
    {
        printf("CreateDirectory Success!\n");
    }

    bDir = RemoveDirectory(
        L"C:\\mydir"
    );
    if (bDir == FALSE)
    {
        printf("RemoveDirectory failed & error no %ul\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 mean *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 created.

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.

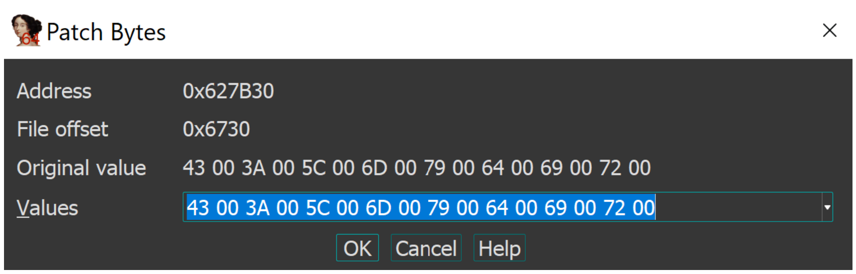
In this chapter we will hack the directory name this will continue to build our experience on custom hacking binaries.

```
.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.

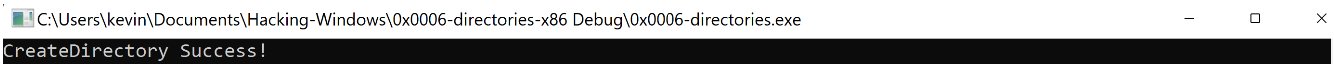
Click *Edit – Patch program – Apply patches to input file…*

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.

Chapter 9: Debugging Directories x64

We are going to debug the 64-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:000000014001187B                 xor     edx, edx        ; lpSecurityAttributes
.text:000000014001187D                 lea     rcx, PathName   ; "C:\\mydir"
.text:0000000140011884                 call    cs:__imp_CreateDirectoryW
```

Here we are simply putting the security attribute into *edx*, which is *0* and then we load the effective address of PathName into *rcx* and call our 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 created.

Let’s stop execution and delete our breakpoint.

We then see our RemoveDirectoryW function.

```
.text:00007FF75AE618B5                 lea     rcx, PathName   ; "C:\\mydir"
.text:00007FF75AE618BC                 call    cs:__imp_RemoveDirectoryW
```

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.

Bingo! Another debug victory!

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

Chapter 10: Hacking Directories x64

We are going to debug the 64-bit version of our Directories program.

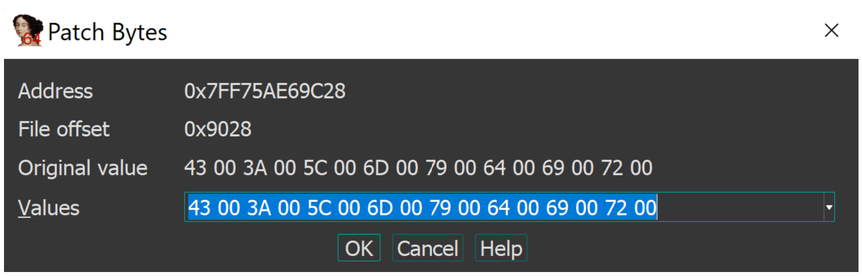
In this chapter we will hack the directory name in an x64 environment.

```
5AE6187D                lea     rcx, PathName   ; "C:\\mydir"
```

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

```
75AE69C28               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.

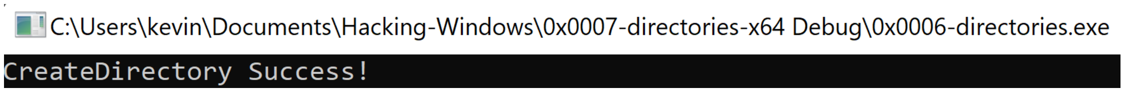
Click *Edit – Patch program – Apply patches to input file…*

Click OK.

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

```
5AE618A9                lea     rcx, aCreatedirector_0 ; "CreateDirectory Success!\n"
5AE618B0                call    j_printf
5AE618B5
5AE618B5 loc_7FF75AE618B5:                     ; CODE XREF: main+47↑j
5AE618B5                lea     rcx, PathName  ; "C:\\hacky"
```

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.

In our next chapter we discuss CopyFile.

Chapter 11: CopyFile

We continue with a simple app that handles the Windows CopyFile API which simply copies the contents of one file into a new file.

Let's create a new project

Create a new project
Empty Project
Next
Project name: 0x000b-copyfile
CHECK Place solution and project in the same directory
Create

RT CLICK on the 0x000b-copyfile 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 bFile;

	bFile = CopyFile(
		L"C:\\temp\\test1.txt",
		L"C:\\temp\\test2.txt",
		TRUE
	);
	if (bFile == FALSE)
	{
		printf("CopyFile failed & error no %ul\n", GetLastError());
	}
	else
	{
		printf("CopyFile Success!\n");
	}

	return 0;
}
```

Let's review the *CopyFileW API* below.

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

Here we see we have 3 parameters. The first, *lpExistingFileName*, is simply the existing file we want to copy. The second, *lpNewFileName*, is the name of the new file to which we will create and copy the contents of the original file to. The third, *bFailIfExists*, is the flag to indicate if the new file already exists and if it does fail the operation if TRUE.

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

We need to manually create the file **test1.txt** within C:\temp so you can use Notepad to do so now. Simply create the file and put any contents you like inside.

When we run the program it shows the following input.

```
CopyFile Success!

C:\Users\kevin\Documents\Hacking-Windows\0x000b-copyfile\0x000b-copyfile\Debug\0x000b-copyfile.exe (process 22464) 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 12: Debugging CopyFile x86

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

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

```
.text:00411877                 push    1               ; bFailIfExists
.text:00411879                 push    offset NewFileName ; "C:\\temp\\test2.txt"
.text:0041187E                 push    offset ExistingFileName ; "C:\\temp\\test1.txt"
.text:00411883                 call    ds:__imp__CopyFileW@12 ; CopyFileW(x,x,x)
```

Here we are simply pushing the *bFailIfExists* onto the stack followed by the *lpNewFileName* and finally the *lpExistingFileName*.

BEFORE we run make sure we delete the file **test2.txt** within C:\temp so we can proceed as if this was being run the first time.

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

NOTICE we see that **test2.text** was created.

This was a very simple debug as I have to take the time again to clearly state that our objective is to take SMALL steps so you can not get overwhelmed and have a full appreciation for what is going on at every step of these very popular Win32API calls.

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

Chapter 13: Hacking CopyFile x86

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

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

```
.text:00AA1879                 push    offset NewFileName ; "C:\\temp\\test2.txt"
```

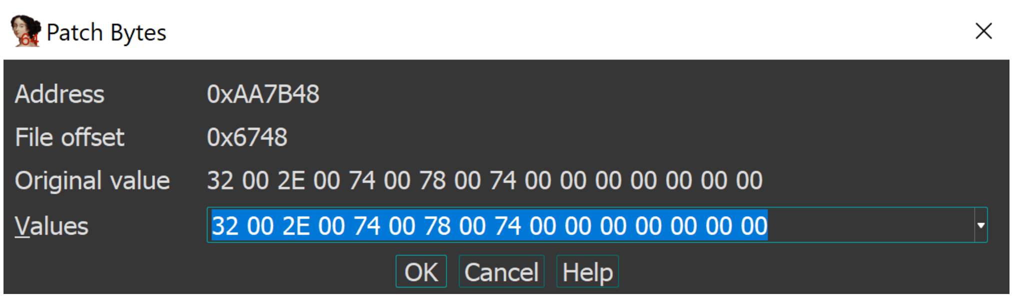
Here we see the *PathName* of *“C:\\temp\\test2.txt”*. Double-click to get to the .rdata section.

```
.rdata:00AA7B30                 text "UTF-16LE", 'C:\temp\test2.txt',0
```

Select the Hex View-1 tab. Click on the *32*.

```
00AA7B30  43 00 3A 00 5C 00 74 00  65 00 6D 00 70 00 5C 00  C.:.\.t.e.m.p.\.
00AA7B40  74 00 65 00 73 00 74 00  32 00 2E 00 74 00 78 00  t.e.s.t.2...t.x.
00AA7B50  74 00 00 00 00 00 00 00  00 00 00 00 43 00 3A 00  t...........C.:.
00AA7B60  5C 00 74 00 65 00 6D 00  70 00 5C 00 74 00 65 00  \.t.e.m.p.\.t.e.
00AA7B70  73 00 74 00 31 00 2E 00  74 00 78 00 74 00 00 00  s.t.1...t.x.t...
00AA7B80  00 00 00 00 00 00 00 00  43 6F 70 79 46 69 6C 65  ........CopyFile
00AA7B90  20 66 61 69 6C 65 64 20  26 20 65 72 72 6F 72 20  ·failed·&·error·
00AA7BA0  6E 6F 20 25 75 6C 0A 00  00 00 00 00 00 00 00 00  no·%ul..........
00AA7BB0  43 6F 70 79 46 69 6C 65  20 53 75 63 63 65 73 73  CopyFile·Success
```

Click *Edit – Patch program – Change byte …*



32 00 2E 00 74 00 78 00 74 00 00 00 00 00 00 00

Let’s change the file to ‘test3’.

33 00 2E 00 74 00 78 00 74 00 00 00 00 00 00 00

Click OK.

Click *Edit – Patch program – Apply patches to input file…*

Click OK.

Back in the IDA View0A tab, let's set a breakpoint on the next instruction after the call to *CopyFileW*.

```
.text:00AA1877                 push    1               ; bFailIfExists
.text:00AA1879                 push    offset NewFileName ; "C:\\temp\\test3.txt"
.text:00AA187E                 push    offset ExistingFileName ; "C:\\temp\\test1.txt"
.text:00AA1883                 call    ds:__imp__CopyFileW@12 ; CopyFileW(x,x,x)
.text:00AA1889                 cmp     esi, esp
```

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

test3   9/27/2021 7:26 AM   Text Document   0 KB

Hooray! We have hacked our simple program and altered the new file name.

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

Chapter 14: Debugging CopyFile x64

We are going to debug the 64-bit version of our CopyFile program.

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

```
.text:000000014001187B                 mov     r8d, 1          ; bFailIfExists
.text:0000000140011881                 lea     rdx, NewFileName ; "C:\\temp\\test2.txt"
.text:0000000140011888                 lea     rcx, ExistingFileName ; "C:\\temp\\test1.txt"
.text:000000014001188F                 call    cs:__imp_CopyFileW
```

Here we are simply putting the value of *bFailIfExists* into *r8d* followed by the *NewFileName* into *rdx* and finally the *ExistingFileName* into *rcx*.

BEFORE we run make sure we delete the file **test2.txt** within C:\temp so we can proceed as if this was being run the first time.

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

NOTICE we see that **test2.text** was created.

This was a very simple debug as I have to take the time again to clearly state that our objective is to take SMALL steps so you can not get overwhelmed and have a full appreciation for what is going on at every step of these very popular Win32API calls.

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

Chapter 15: Hacking CopyFile x64

We are going to hack the 64-bit version of our CopyFile program.

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

```
.text:00007FF79C8C1881                 lea     rdx, NewFileName ; "C:\\temp\\test2.txt"
```

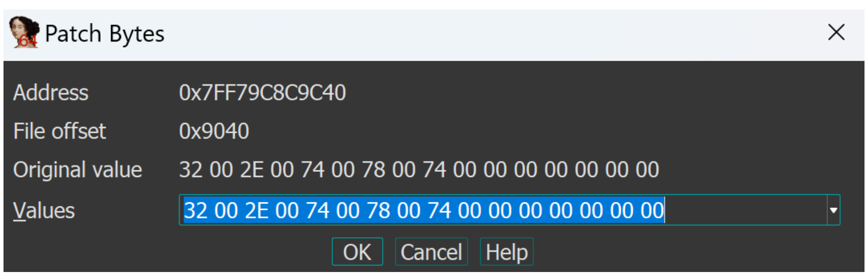
Here we see the *PathName* of *"C:\\temp\\test2.txt"*. Double-click to get to the .rdata section.

```
.rdata:00007FF79C8C9C28                 text "UTF-16LE", 'C:\temp\test2.txt',0
```

Select the Hex View-1 tab. Click on the *32*.

```
00007FF79C8C9C20  00 00 00 00 00 00 00 00  43 00 3A 00 5C 00 74 00  ........C.:.\.t.
00007FF79C8C9C30  65 00 6D 00 70 00 5C 00  74 00 65 00 73 00 74 00  e.m.p.\.t.e.s.t.
00007FF79C8C9C40  32 00 2E 00 74 00 78 00  74 00 00 00 00 00 00 00  2...t.x.t.......
00007FF79C8C9C50  00 00 00 00 00 00 00 00  43 00 3A 00 5C 00 74 00  ........C.:.\.t.
00007FF79C8C9C60  65 00 6D 00 70 00 5C 00  74 00 65 00 73 00 74 00  e.m.p.\.t.e.s.t.
00007FF79C8C9C70  31 00 2E 00 74 00 78 00  74 00 00 00 00 00 00 00  1...t.x.t.......
00007FF79C8C9C80  00 00 00 00 00 00 00 00  43 6F 70 79 46 69 6C 65  ........CopyFile
00007FF79C8C9C90  20 66 61 69 6C 65 64 20  26 20 65 72 72 6F 72 20  ·failed·&·error·
00007FF79C8C9CA0  6E 6F 20 25 75 6C 0A 00  00 00 00 00 00 00 00 00  no·%ul..........
00007FF79C8C9CB0  43 6F 70 79 46 69 6C 65  20 53 75 63 63 65 73 73  CopyFile·Success
```

Click *Edit – Patch program – Change byte …*



32 00 2E 00 74 00 78 00 74 00 00 00 00 00 00 00

Let's change the file to 'test3'.

33 00 2E 00 74 00 78 00 74 00 00 00 00 00 00 00

Click OK.

Click *Edit – Patch program – Apply patches to input file…*

Click OK.

Back in the IDA View0A tab, let's set a breakpoint on the next instruction after the call to *CopyFileW*.

```
.text:00007FF79C8C187B        mov     r8d, 1              ; bFailIfExists
.text:00007FF79C8C1881        lea     rdx, NewFileName ; "C:\\temp\\test3.txt"
.text:00007FF79C8C1888        lea     rcx, ExistingFileName ; "C:\\temp\\test1.txt"
.text:00007FF79C8C188F        call    cs:__imp_CopyFileW
.text:00007FF79C8C1895        mov     [rbp+0F0h+var_EC], eax
```

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

test3	9/27/2021 7:26 AM	Text Document	0 KB

Hooray! We have hacked our simple program and altered the new file name.

In our next chapter we discuss MoveFile.

Chapter 16: MoveFile

We continue with a simple app that handles the Windows MoveFile API which simply moves (renames) one file.

Let's create a new project

Create a new project
Empty Project
Next
Project name: 0x0010-movefile
CHECK Place solution and project in the same directory
Create

RT CLICK on the 0x0010-movefile 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 bFile;

	bFile = MoveFile(
		L"C:\\temp\\test1.txt",
		L"C:\\temp\\test2.txt"
	);
	if (bFile == FALSE)
	{
		printf("MoveFile failed and error no %ul\n", GetLastError());
	}
	else
	{
		printf("MoveFile Success!");
	}
}
```

Let's review the *MoveFileW* API below.

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

Here we see we have 2 parameters. The first, *lpExistingFileName*, is simply the existing file we want to copy. The second, *lpNewFileName*, is the name of the new file to which we will move the contents of the original file to.

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

We need to manually create the file **test1.txt** within C:\temp so you can use Notepad to do so now. Simply create the file and put any contents you like inside.

When we run the program it shows the following input.

```
MoveFile Success!

C:\Users\kevin\Documents\Hacking-Windows\0x0010-movefile\Debug\0x0010-movefile.exe (process 10480) 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 17: Debugging MoveFile x86

We are going to debug the 32-bit version of our MoveFile 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 a *_KERNEL32_NULL_THUNK_DATA* function.

```
.text:00411887                 push    offset NewFileName ; "C:\\temp\\test2.txt"
.text:0041188C                 push    offset ExistingFileName ; "C:\\temp\\test1.txt"
.text:00411891                 call    ds:_KERNEL32_NULL_THUNK_DATA
```

Wait! What?

Let's double-click and do more inspection.

```
.idata:0041B064 ; BOOL (__stdcall *KERNEL32_NULL_THUNK_DATA)(LPCWSTR lpExistingFileName, LPCWSTR lpNewFileName)
.idata:0041B064                 extrn _KERNEL32_NULL_THUNK_DATA:dword
.idata:0041B064                                         ; CODE XREF: _main+31↑p
.idata:0041B064                                         ; DATA XREF: MoveFileW(x,x)↑r ...
```

Here we do see this is actually calling MoveFileW as expected.

In our last chapter we reviewed the API in C. Here we first push the *lpNewFileName* param to the stack followed by the *lpExistingFileName* 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 **test2.txt** file has been created.

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

Chapter 18: Hacking MoveFile x86

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

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

```
.text:00A21887                 push    offset NewFileName ; "C:\\temp\\test2.txt"
```

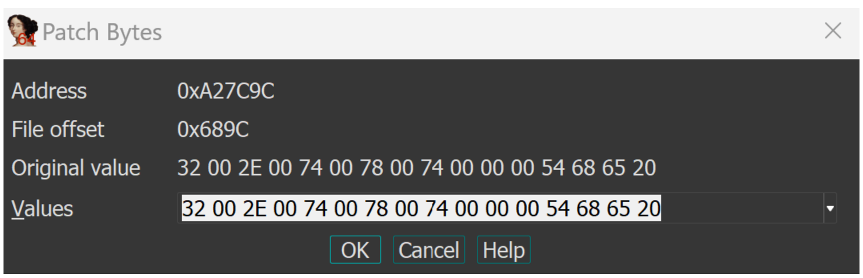
Here we see the *PathName* of *"C:\\temp\\test2.txt"*. Double-click to get to the .rdata section.

```
.rdata:00A27C84                 text "UTF-16LE", 'C:\temp\test2.txt',0
```

Select the Hex View-1 tab. Click on the *32*.

```
00A27C80  64 2E 00 00 43 00 3A 00  5C 00 74 00 65 00 6D 00  d...C.:.\.t.e.m.
00A27C90  70 00 5C 00 74 00 65 00  73 00 74 00 32 00 2E 00  p.\.t.e.s.t.2...
00A27CA0  74 00 78 00 74 00 00 00  54 68 65 20 76 61 6C 75  t.x.t...The·valu
```

Click *Edit – Patch program – Change byte …*



32 00 2E 00 74 00 78 00 74 00 00 00 00 00 00 00

Let's change the file to 'test3'.

33 00 2E 00 74 00 78 00 74 00 00 00 00 00 00 00

Click OK.

Click *Edit – Patch program – Apply patches to input file…*

Click OK.

Back in the IDA View0A tab, let's set a breakpoint on the next instruction after the call to *_KERNEL32_NULL_THUNK_DATA*.

```
.text:00A21887          push    offset NewFileName ; "C:\\temp\\test3.txt"
.text:00A2188C          push    offset ExistingFileName ; "C:\\temp\\test1.txt"
.text:00A21891          call    ds:_KERNEL32_NULL_THUNK_DATA
.text:00A21897          cmp     esi, esp
```

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

test3	9/27/2021 7:26 AM	Text Document	0 KB

Hooray! We have hacked our simple program and altered the new file name.

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

Chapter 19: Debugging MoveFile x64

We are going to debug the 64-bit version of our MoveFile program.

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

```
.text:000000014001187B                 lea     rdx, NewFileName ; "C:\\temp\\test2.txt"
.text:0000000140011882                 lea     rcx, ExistingFileName ; "C:\\temp\\test1.txt"
.text:0000000140011889                 call    cs:__imp_MoveFileW
```

Here we are simply putting the value of *NewFileName* into *rdx* and the *ExistingFileName* into *rcx*.

BEFORE we run make sure we rename the file **test2.txt** to **test1.txt** within C:\temp so we can proceed as if this was being run the first time.

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

NOTICE we see that **test2.txt** was the final renamed file.

This was a very simple debug as I have to take the time again to clearly state that our objective is to take SMALL steps so you can not get overwhelmed and have a full appreciation for what is going on at every step of these very popular Win32API calls.

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

Chapter 20: Hacking MoveFile x64

We are going to hack the 64-bit version of our MoveFile program.

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

```
.text:00007FF7227D187B                 lea     rdx, NewFileName ; "C:\\temp\\test2.txt"
```

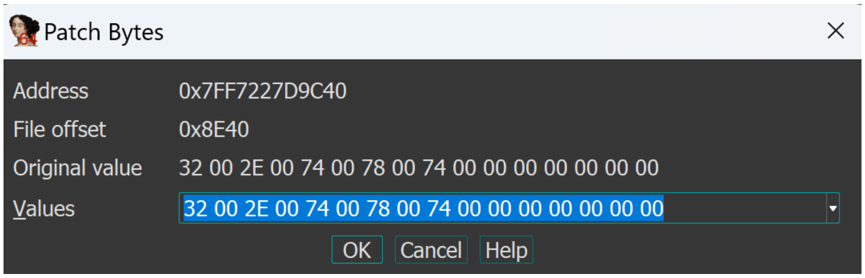
Here we see the *PathName* of *“C:\\temp\\test2.txt”*. Double-click to get to the .rdata section.

```
.rdata:00007FF79C8C9C28                 text "UTF-16LE", 'C:\temp\test2.txt',0
```

Select the Hex View-1 tab. Click on the *32*.

```
00007FF7227D9C20  00 00 00 00 00 00 00 00  43 00 3A 00 5C 00 74 00  ........C.:.\.t.
00007FF7227D9C30  65 00 6D 00 70 00 5C 00  74 00 65 00 73 00 74 00  e.m.p.\.t.e.s.t.
00007FF7227D9C40  32 00 2E 00 74 00 78 00  74 00 00 00 00 00 00 00  2...t.x.t.......
00007FF7227D9C50  00 00 00 00 00 00 00 00  43 00 3A 00 5C 00 74 00  ........C.:.\.t.
00007FF7227D9C60  65 00 6D 00 70 00 5C 00  74 00 65 00 73 00 74 00  e.m.p.\.t.e.s.t.
00007FF7227D9C70  31 00 2E 00 74 00 78 00  74 00 00 00 00 00 00 00  1...t.x.t.......
00007FF7227D9C80  00 00 00 00 00 00 00 00  4D 6F 76 65 46 69 6C 65  ........MoveFile
00007FF7227D9C90  20 66 61 69 6C 65 64 20  61 6E 64 20 65 72 72 6F  ·failed·and·erro
00007FF7227D9CA0  72 20 6E 6F 20 25 75 6C  0A 00 00 00 00 00 00 00  r·no·%ul........
00007FF7227D9CB0  4D 6F 76 65 46 69 6C 65  20 53 75 63 63 65 73 73  MoveFile·Success
```

Click *Edit – Patch program – Change byte …*



32 00 2E 00 74 00 78 00 74 00 00 00 00 00 00 00

Let’s change the file to ‘test3’.

33 00 2E 00 74 00 78 00 74 00 00 00 00 00 00 00

Click OK.

Click *Edit – Patch program – Apply patches to input file…*

Click OK.

Back in the IDA View0A tab, let’s set a breakpoint on the next instruction after the call to *CopyFileW*.

```
.text:00007FF7227D187B        lea     rdx, NewFileName ; "C:\\temp\\test3.txt"
.text:00007FF7227D1882        lea     rcx, ExistingFileName ; "C:\\temp\\test1.txt"
.text:00007FF7227D1889        call    cs:__imp_MoveFileW
.text:00007FF7227D188F        mov     [rbp+0F0h+var_EC], eax
```

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

test3	9/27/2021 7:26 AM	Text Document	0 KB

Hooray! We have hacked our simple program and altered the new file name.

In our next chapter we discuss CreateFile.

Chapter 21: CreateFile

We continue with a simple app that handles the Windows CreateFile API which simply creates one file.

Let’s create a new project

Create a new project
Empty Project
Next
Project name: 0x0011-createfile
CHECK Place solution and project in the same directory
Create

RT CLICK on the 0x0011-createfile 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)
{
    HANDLE hFile;

    hFile = CreateFile(
        L"C:\\temp\\test.txt",
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ,
        NULL,
        CREATE_NEW,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );
    if (hFile == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed and error no %ul\n", GetLastError());
    }
    else
    {
        printf("CreateFile Success!");
    }

    CloseHandle(hFile);
}
```

Let’s review the *CreateFileW* API below.

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

Here we see we have 7 parameters. The first, *lpFileName*, is simply the file we want to create. The second, *dwDesiredAccess*, is the requested access to the file or device which will be read, write, zero or neither zero. The third, *dwShareMode*, is the requested sharing mode of the file or device which is read, write, both, delete, all of these or none. The fourth, *lpSecurityAttributes*, is a pointer to a SECURITY_ATTRIBUTES structure that contains two separate data members, this is an optional param. The fifth, *dwCreationDisposition*, is an action to take on a file or device that exists or does not exist. The sixth, *dwFlagsAndAttributes*, is the file or device attributes and flags. The seventh, hTemplateFile, is a valid handle to a template file with the GENERIC_READ access right. This is optional.

The return value is an open handle to the specified file, device, named pipe, or mail slot or if fails, the return value is INVALID_HANDLE_VALUE which you can get with *GetLastError*.

When we run the program it shows the following input.

```
CreateFile Success!

C:\Users\kevin\Documents\Hacking-Windows\0x0011-createfile\0x0011-createfile\x64\Debug\0x0011-createfile.exe (process 6488) 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 22: Debugging CreateFile x86

We are going to debug the 32-bit version of our CreateFile 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 a *__imp_CreateFileW@28* function.

```
.text:00411887                 push    0               ; hTemplateFile
.text:00411889                 push    80h ; '€'       ; dwFlagsAndAttributes
.text:0041188E                 push    1               ; dwCreationDisposition
.text:00411890                 push    0               ; lpSecurityAttributes
.text:00411892                 push    1               ; dwShareMode
.text:00411894                 push    0C0000000h      ; dwDesiredAccess
.text:00411899                 push    offset FileName ; "C:\\temp\\test.txt"
.text:0041189E                 call    ds:__imp__CreateFileW@28 ; CreateFileW(x,x,x,x,x,x,x)
```

Here we do see this is actually calling CreateFileW as expected.

In our last chapter we reviewed the API in C. If you are not familiar with the parameters please review the last chapter.

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

NOTICE we see that our **test.txt** file has been created.

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

Chapter 23: Hacking CreateFile x86

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

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

```
.text:008E1899                 push    offset FileName ; "C:\\temp\\test.txt"
```

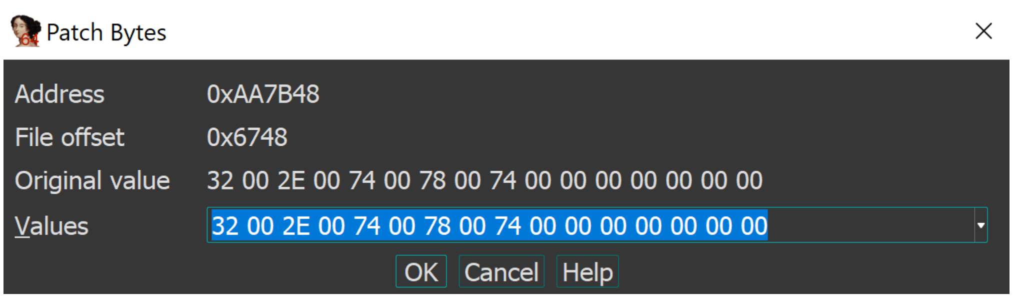
Here we see the *PathName* of *"C:\\temp\\test.txt"*. Double-click to get to the .rdata section.

```
.rdata:008E7B30                 text "UTF-16LE", 'C:\temp\test.txt',0
```

Select the Hex View-1 tab. Click on the second *74*.

```
008E7B30  43 00 3A 00 5C 00 74 00  65 00 6D 00 70 00 5C 00  C.:.\.t.e.m.p.\.
008E7B40  74 00 65 00 73 00 74 00  2E 00 74 00 78 00 74 00  t.e.s.t...t.x.t.
008E7B50  00 00 00 00 00 00 00 00  43 72 65 61 74 65 46 69  ........CreateFi
008E7B60  6C 65 20 66 61 69 6C 65  64 20 61 6E 64 20 65 72  le·failed·and·er
008E7B70  72 6F 72 20 6E 6F 20 25  75 6C 0A 00 00 00 00 00  ror·no·%ul......
```

Click *Edit – Patch program – Change byte …*



74 00 65 00 73 00 74 00 2E 00 74 00 78 00 74

Let's change the file to 'hest'.

68 00 65 00 73 00 74 00 2E 00 74 00 78 00 74

Click OK.

Click *Edit – Patch program – Apply patches to input file…*

Click OK.

Back in the IDA View0A tab, let’s set a breakpoint on the next instruction after the call to *CreateFileW*.

```
.text:008E1885                mov     esi, esp
.text:008E1887                push    0               ; hTemplateFile
.text:008E1889                push    80h ; '€'       ; dwFlagsAndAttributes
.text:008E188E                push    1               ; dwCreationDisposition
.text:008E1890                push    0               ; lpSecurityAttributes
.text:008E1892                push    1               ; dwShareMode
.text:008E1894                push    0C0000000h      ; dwDesiredAccess
.text:008E1899                push    offset FileName ; "C:\\temp\\hest.txt"
.text:008E189E                call    ds:__imp__CreateFileW@28 ; CreateFileW(x,x,x,x,x,x,x)
.text:008E18A4                cmp     esi, esp
```

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

hest   5/5/2022 12:05 PM   Text Document   0 KB

Hooray! We have hacked our simple program and altered the new file name.

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

Chapter 24: Debugging CreateFile x64

We are going to debug the 64-bit version of our CreateFile program.

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

```
.text:000000014001188B                 mov     [rsp+130h+hTemplateFile], 0 ; hTemplateFile
.text:0000000140011894                 mov     [rsp+130h+dwFlagsAndAttributes], 80h ; '€' ; dwFlagsAndAttributes
.text:000000014001189C                 mov     [rsp+130h+dwCreationDisposition], 1 ; dwCreationDisposition
.text:00000001400118A4                 xor     r9d, r9d        ; lpSecurityAttributes
.text:00000001400118A7                 mov     r8d, 1          ; dwShareMode
.text:00000001400118AD                 mov     edx, 0C0000000h ; dwDesiredAccess
.text:00000001400118B2                 lea     rcx, FileName   ; "C:\\temp\\test.txt"
.text:00000001400118B9                 call    cs:__imp_CreateFileW
```

Here we are putting the value of *hTemplateFile, dwFlagsAndAttributes, dwCreationDisposition* onto the stack and the *lpSecurityAttributes* into *r9, dwShareMode* into *r8, dwDesireAccess* into *rdx* and *FileName* into *rcx*.

BEFORE we run make sure we remove the file **test.txt** within C:\temp so we can proceed as if this was being run the first time.

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

NOTICE we see that **test.txt** was created.

This was a very simple debug as I have to take the time again to clearly state that our objective is to take SMALL steps so you can not get overwhelmed and have a full appreciation for what is going on at every step of these very popular Win32API calls.

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

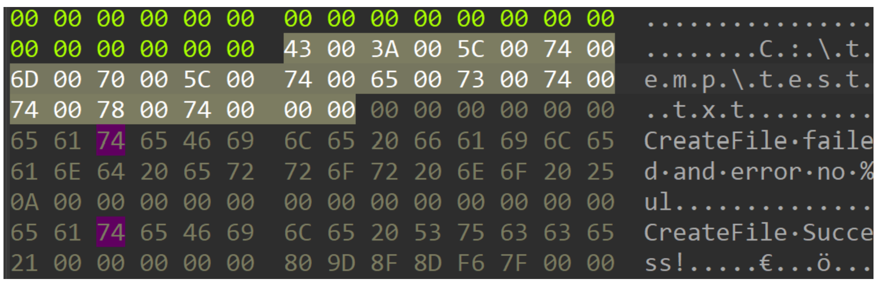
Chapter 25: Hacking CreateFile x64

We are going to hack the 64-bit version of our CreateFile program.

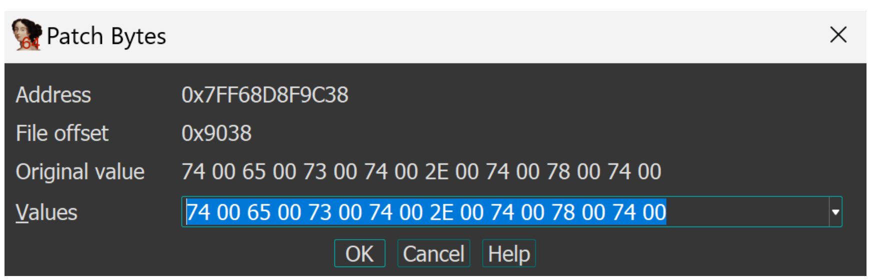
In this chapter we will hack the file name this will continue to build our experience on custom hacking binaries.

Here we see the *FileName* of *"C:\\temp\\test.txt"*. Double-click to get to the .rdata section.

Select the Hex View-1 tab. Click on the 2nd *74*.



Click *Edit – Patch program – Change byte …*



74 00 65 00 73 00 74 00 2E 00 74 00 78 00 74 00

Let's change the file to 'fest'.

74 00 65 00 73 00 66 00 2E 00 74 00 78 00 74 00

Click OK.

Click *Edit – Patch program – Apply patches to input file…*

Click OK.

Back in the IDA View0A tab, let’s set a breakpoint on the next instruction after the call to *CreateFileW*.

```
.text:00007FF68D8F188B    mov     [rsp+130h+hTemplateFile], 0 ; hTemplateFile
.text:00007FF68D8F1894    mov     [rsp+130h+dwFlagsAndAttributes], 80h ; '€' ; dwFlagsAndAttributes
.text:00007FF68D8F189C    mov     [rsp+130h+dwCreationDisposition], 1 ; dwCreationDisposition
.text:00007FF68D8F18A4    xor     r9d, r9d        ; lpSecurityAttributes
.text:00007FF68D8F18A7    mov     r8d, 1          ; dwShareMode
.text:00007FF68D8F18AD    mov     edx, 0C0000000h ; dwDesiredAccess
.text:00007FF68D8F18B2    lea     rcx, FileName   ; "C:\\temp\\fest.txt"
.text:00007FF68D8F18B9    call    cs:__imp_CreateFileW
.text:00007FF68D8F18BF    mov     [rbp+0F0h+hObject], rax
```

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

fest	5/16/2022 9:43 AM	Text Document	0 KB

Hooray! We have hacked our simple program and altered the new file name.

In our next chapter we discuss WriteFile.

Chapter 26: WriteFile

We continue with a simple app that handles the Windows WriteFile API which simply populates data in one file.

Let's create a new project

Create a new project
Empty Project
Next
Project name: 0x0012-writefile
CHECK Place solution and project in the same directory
Create

RT CLICK on the 0x0012-writefile 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)
{
    HANDLE hFile;
    BOOL bFile;
    char lpBuffer[] = "Reversing is my life!";
    DWORD nNumberOfBytesToWrite = strlen(lpBuffer);
    DWORD lpNumberOfBytesWritten = 0;

    hFile = CreateFile(
        L"C:\\temp\\test.txt",
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ,
        NULL,
        CREATE_NEW,
        FILE_ATTRIBUTE_NORMAL,
        NULL
    );
    if (hFile == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed and error no %ul\n", GetLastError());
    }
    else
    {
        printf("CreateFile Success!\n");
    }

    bFile = WriteFile(
        hFile,
```

```
        lpBuffer,
        nNumberOfBytesToWrite,
        &lpNumberOfBytesWritten,
        NULL
    );
    if (bFile == INVALID_HANDLE_VALUE)
    {
        printf("WriteFile failed and error no %ul\n", GetLastError());
    }
    else
    {
        printf("WriteFile Success!");
    }

    CloseHandle(hFile);
}
```

Let's review the *WriteFile* API below.

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

Here we see we have 5 parameters. The first, *hFile*, is simply the file we created. The second, *lpBuffer*, is a pointer to the buffer containing the data to be written to the file or device. The third, *nNumberOfBytesToWrite*, is the number of bytes to be written to the file or device. The fourth, *lpNumberOfBytesWritten*, is a pointer to the variable that receives the number of bytes written when using a synchronous hFile param and WriteFile sets this value to zero before doing any work or error checking and use NULL for this param if this is an async operation to avoid erroneous results, this is an optional param. The fifth, *lpOverlapped*, is a pointer to an OVERLAPPED structure if the hFile param was opened with FILE_FLAG_OVERLAPPED otherwise NULL. This is optional.

The return value is nonzero TRUE or if fails, the return value is INVALID_HANDLE_VALUE which you can get with *GetLastError*.

When we run the program it shows the following input.

```
CreateFile Success!
WriteFile Success!

C:\Users\kevin\Documents\Hacking-Windows\0x0012-writefile\0x0012-writefile\Debug\0x0012-writefile.exe (process 7964) 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 27: Debugging WriteFile x86

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

In the IDA View-A text view we first see a *__imp_WriteFile@20* function.

```
.text:00415421                 push    0               ; lpOverlapped
.text:00415423                 push    0               ; lpNumberOfBytesWritten
.text:00415425                 mov     eax, [ebp+nNumberOfBytesToWrite]
.text:00415428                 push    eax             ; nNumberOfBytesToWrite
.text:00415429                 lea     ecx, [ebp+lpBuffer]
.text:0041542C                 push    ecx             ; lpBuffer
.text:0041542D                 mov     edx, [ebp+hFile]
.text:00415430                 push    edx             ; hFile
.text:00415431                 call    ds:__imp__WriteFile@20 ; WriteFile(x,x,x,x,x)
```

In our last chapter we reviewed the API in C. Here we first push the *lpOverlapped* param to the stack followed by the *lpNumberOfBytesWritten* param followed by the *nNumberOfBytesToWrite* param followed by the *lpBuffer* param followed by the *hFile* 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 **test.txt** file has been created and populated with, **Reversing is my life!**

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

Chapter 28: Hacking WriteFile x86

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

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

```
.text:000A53D8                push    offset FileName ; "C:\\temp\\test.txt"
```

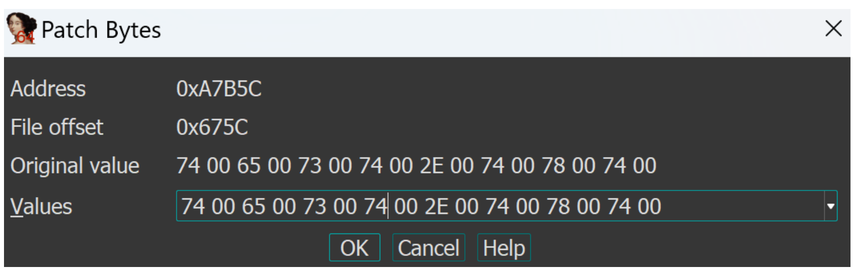
Here we see the *PathName* of *"C:\\temp\\test.txt"*. Double-click to get to the .rdata section.

```
.rdata:000A7B4C                text "UTF-16LE", 'C:\temp\test.txt',0
```

Select the Hex View-1 tab. Click on the second *74*.

```
000A7B40  73 21 00 00 00 00 00 00  00 00 00 00 43 00 3A 00  s!.........C.:.
000A7B50  5C 00 74 00 65 00 6D 00  70 00 5C 00 74 00 65 00  \.t.e.m.p.\.t.e.
000A7B60  73 00 74 00 2E 00 74 00  78 00 74 00 00 00 00 00  s.t...t.x.t.....
```

Click *Edit – Patch program – Change byte …*



74 00 65 00 6D 00 70 00 5C 00 74 00 65 00 73 00

Let's change the file to 'tesv'.

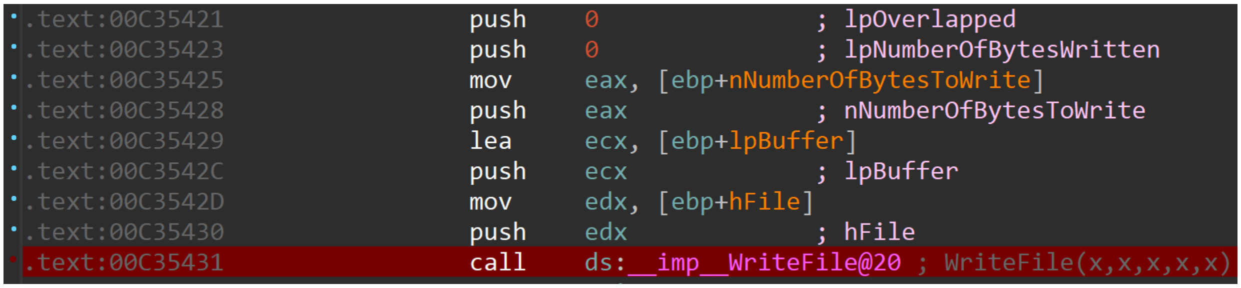
75 00 65 00 6D 00 70 00 5C 00 76 00 65 00 73 00

Click OK.

Click *Edit – Patch program – Apply patches to input file…*

Click OK.

Back in the IDA View0A tab, let's set a breakpoint on the next instruction after the call to *__imp__WriteFileW@20*.



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

Hooray! We have hacked our simple program and altered the new file name.

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

Chapter 29: Debugging WriteFile x64

We are going to debug the 64-bit version of our WriteFile program.

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

```
.text:0000000140011943                 mov     qword ptr [rsp+1A0h+dwCreationDisposition], 0 ; lpOverlapped
.text:000000014001194C                 xor     r9d, r9d        ; lpNumberOfBytesWritten
.text:000000014001194F                 mov     r8d, [rbp+160h+nNumberOfBytesToWrite] ; nNumberOfBytesToWrite
.text:0000000140011953                 lea     rdx, [rbp+160h+Str] ; lpBuffer
.text:0000000140011957                 mov     rcx, [rbp+160h+hFile] ; hFile
.text:000000014001195B                 call    cs:__imp_WriteFile
```

Here we first putting the *lpOverlapped* param to the stack, an offset of *rsp + 1a0 + dwCreationDisposition* followed by the *lpNumberOfBytesWritten* param into *r9* followed by the *nNumberOfBytesToWrite* param into *r8* followed by the *lpBuffer* param into *rdx* followed by the *hFile* param into *rcx* and then we call the function.

BEFORE we run make sure we remove the file **test.txt** within C:\temp so we can proceed as if this was being run the first time.

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

NOTICE we see that **test.txt** was created.

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

Chapter 30: Hacking WriteFile x64

We are going to hack the 64-bit version of our WriteFile program.

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

```
.text:00007FF7CE981909                 lea     rcx, FileName   ; "C:\\temp\\test.txt"
```

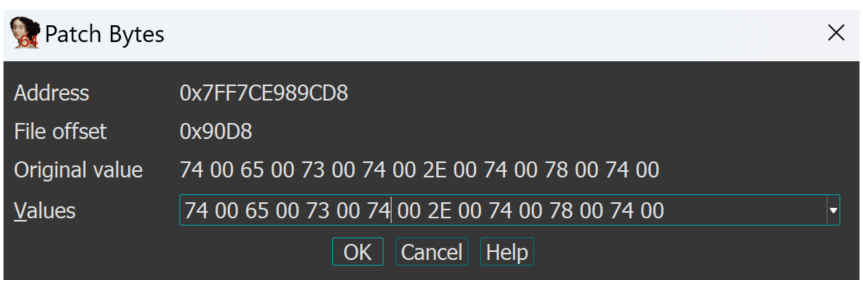
Here we see the *FileName* of *"C:\\temp\\test.txt"*. Double-click to get to the .rdata section.

```
.rdata:00007FF7CE989CC8                 text "UTF-16LE", 'C:\temp\test.txt',0
```

Select the Hex View-1 tab. Click on the 2nd *74*.

```
00007FF7CE989CC0  00 00 00 00 00 00 00 00  43 00 3A 00 5C 00 74 00  ........C.:.\.t.
00007FF7CE989CD0  65 00 6D 00 70 00 5C 00  74 00 65 00 73 00 74 00  e.m.p.\.t.e.s.t.
00007FF7CE989CE0  2E 00 74 00 78 00 74 00  00 00 00 00 00 00 00 00  ..t.x.t.........
```

Click *Edit – Patch program – Change byte …*



74 00 65 00 73 00 74 00 2E 00 74 00 78 00 74 00

Let's change the file to 'tesf'.

74 00 65 00 73 00 74 00 2E 00 66 00 78 00 74 00

Click OK.

Click *Edit – Patch program – Apply patches to input file…*

Click OK.

Back in the IDA View0A tab, let's set a breakpoint on the next instruction after the call to *CreateFileW*.

```
.text:00007FF7CE981943                 mov     qword ptr [rsp+1A0h+dwCreationDisposition], 0 ; lpOverlapped
.text:00007FF7CE98194C                 xor     r9d, r9d        ; lpNumberOfBytesWritten
.text:00007FF7CE98194F                 mov     r8d, [rbp+160h+nNumberOfBytesToWrite] ; nNumberOfBytesToWrite
.text:00007FF7CE981953                 lea     rdx, [rbp+160h+Str] ; lpBuffer
.text:00007FF7CE981957                 mov     rcx, [rbp+160h+hFile] ; hFile
.text:00007FF7CE98195B                 call    cs:__imp_WriteFile
.text:00007FF7CE981961                 mov     [rbp+160h+var_13C], eax
```

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

tesf	6/24/2022 7:46 AM	Text Document	1 KB

Hooray! We have hacked our simple program and altered the new file name.

I hope you have enjoyed this tutorial and have learned how to now take any remaining Win32API function and reverse engineer it either in x86 or x64.

Happy Hacking!