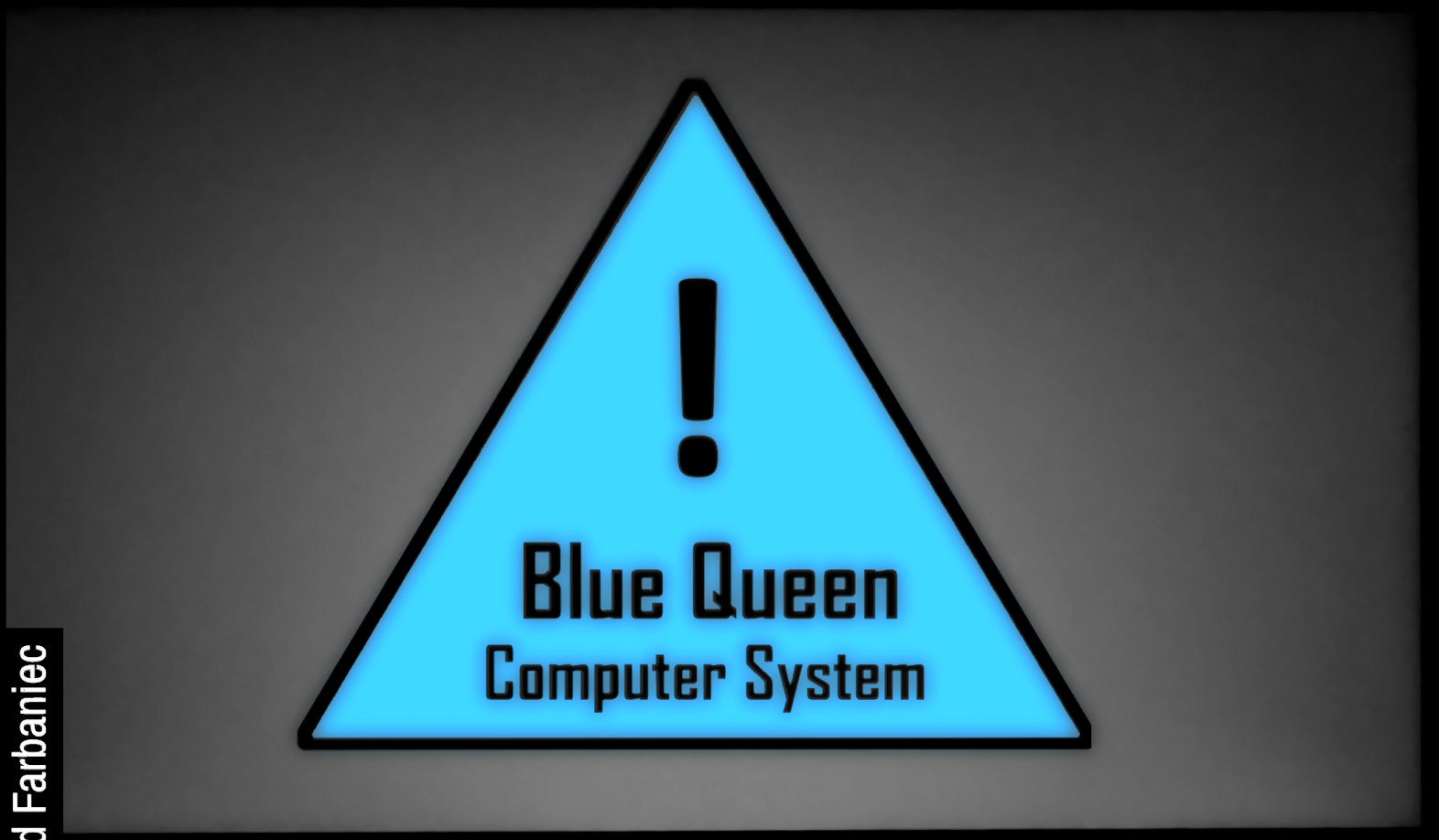
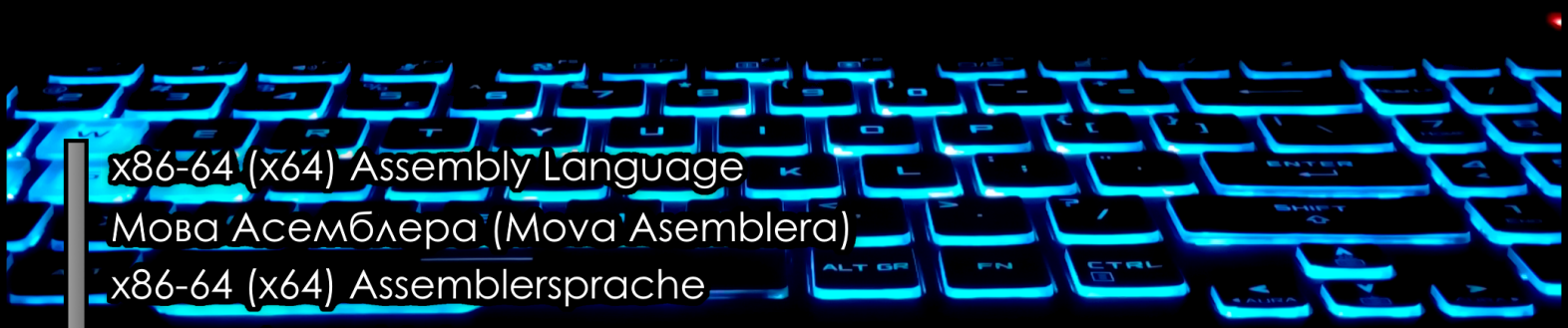
ethical.blue Magazine



Dawid Farbaniec



x86-64 (x64) Assembly Language
Мова Асемблера (Mova Asemblera)
x86-64 (x64) Assemblersprache
Lingua Assemblatoria

0xc3a1

Table of Contents

Bit, Byte, and Machine Word

The smallest unit of information in classical computer memory is a bit, which can be in one of two states: zero or one. In contrast, quantum computers, expected to be more available in the post-quantum era, use quantum bits that can exist in superpositions, which, simply put, are combinations of zero and one simultaneously. [7]

Fascinating, but let's return to the classical technology.

A bit with its two states can be compared to a light bulb or a flag. On or off, set or cleared. In computer programs, we use digits, so a bit can be either 1 or 0.

The decimal number system uses ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In contrast, the binary number system uses only two digits: 0 and 1.

It's important to notice the repeating pattern presented below.

In the decimal system, we count from zero to nine. When we reach 9 (nine), which is the highest digit, then we reset it to 0 (zero) and add 1 (one) to the left, so 9 (nine) becomes 10 (ten).

In the binary system, we only use zero and one. After 1 (one), which is the highest digit, we reset to 0 (zero) and add 1 (one) to the left, so 1 (one) becomes 10 (two in binary). The pattern repeats: 0, 1, 10, 11, 100, 101, 110, 111, 1000, and so on. In the hexadecimal system, we count from zero to nine and from letter A to F, so we have sixteen symbols. The pattern is similar to the previous one, look: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, and so on. See Figure 1.

Binary	Decimal	Hexadecimal
0	0	0
1	1	1
10	2	2
...	...	...
1010	10	A
1011	11	B
...	...	...
1111	15	15
...	...	...
11000011	195	C3
...	...	...
11111111	255	FF
...	...	...

Figure 1. Binary, Decimal, and Hexadecimal

Over time, odd-looking hexadecimal numbers like C3, 1F4E, or FFFF become everyday and no longer seem like weird creatures.

Fundamental Data Types

In x64 core programs, a byte consists of eight bits. Two bytes are a machine word (16 bits). Two words are a doubleword (32 bits), and two doublewords are a quadword (64 bits). Two quadwords are a double quadword (128 bits, also called an octword).

Bit indexes start from zero, not one. The most significant bit is also called the sign bit.

A signed byte with value 11111111 in binary (hexadecimal FF) is -1 in decimal. An unsigned byte with value 11111111 in binary (hexadecimal FF) is 255 in decimal.

Signed values are represented in two's complement format. The conversion of positive number to its negative equivalent is simple. Invert all bits and add one to the result. [1] [6]

An example of converting one (+1) to minus one (-1) in two's complement.

- 00000001 (binary) = 1 (decimal)
- NOT 00000001 = 11111110 (invert bits)
- 11111110 + 1 = 11111111 (add one)
- 11111111 (binary, two's complement) = -1 (decimal)

Numeric Limits

The numeric limits of specified data types are presented below.

- Signed Byte
  -128 .. 127
- Unsigned Byte
  0 .. 255
- Signed Word
  -32 768 .. 32 767
- Unsigned Word
  0 .. 65 535
- Signed Doubleword
  -2 147 483 648 .. 2 147 483 647
- Unsigned Doubleword
  0 .. 4 294 967 295
- Signed Quadword
  -9 223 372 036 854 775 808
  .. 9 223 372 036 854 775 807
- Unsigned Quadword
  0 .. 18 446 744 073 709 551 615

Byte Ordering

The x86 and x64 processors use little-endian (LE) byte ordering. This means the least significant byte is stored at the lowest byte address.

Verify SystemType with PowerShell

```
зона://ethical.blue
PS C:\> Get-CimInstance Win32_ComputerSystem | Select-Object SystemType

SystemType
----------
x64-based PC

PS C:\>
```

Assembling and Linking

Main.asm

```
; ...
xor r9, r9
lea r8, Caption
lea rdx, Text
xor rcx, rcx
call MessageBoxW
xor rcx, rcx
call ExitProcess
; ...
```

ML64.exe
Assembler Tool

Assembling and Linking to Generate an Executable

Mnemonics, Directives, and Source Code are Translated into Raw Machine Code

Program.exe

```
...
4D 33 C9
4C 8D 05 CB 2F 00 00
48 8D 15 C4 31 00 00
48 33 C9
E8 36 00 00 00
48 33 C9
E8 28 00 00 00
...
```


Author: Dawid Farbaniec

Microsoft Macro Assembler x64

Assembly is a programming language for a specific platform, and an assembler is a tool that translates source code (mnemonics, directives, etc.) into machine code and finally into an executable file. Machine code consists of raw bytes that a processor can execute. Each instruction includes an operation code (opcode) that tells the processor which specific function to perform. The exact format and encoding of these instructions are defined in the processor documentation. [1] [6]

When programming in assembly language, one could embed raw opcodes directly into the source code. However, this approach makes the code extremely difficult to read and maintain. Instead, there are mnemonics used that are textual representations of those opcodes.

The Microsoft Macro Assembler (x64) comes with Visual Studio, which you can install by running the following command in PowerShell.

```
winget install --id Microsoft.
   VisualStudio.Community --override
    "--passive --force --wait --
   locale en-us --add Microsoft.
   VisualStudio.Workload.
   NativeDesktop --
   includeRecommended"
```

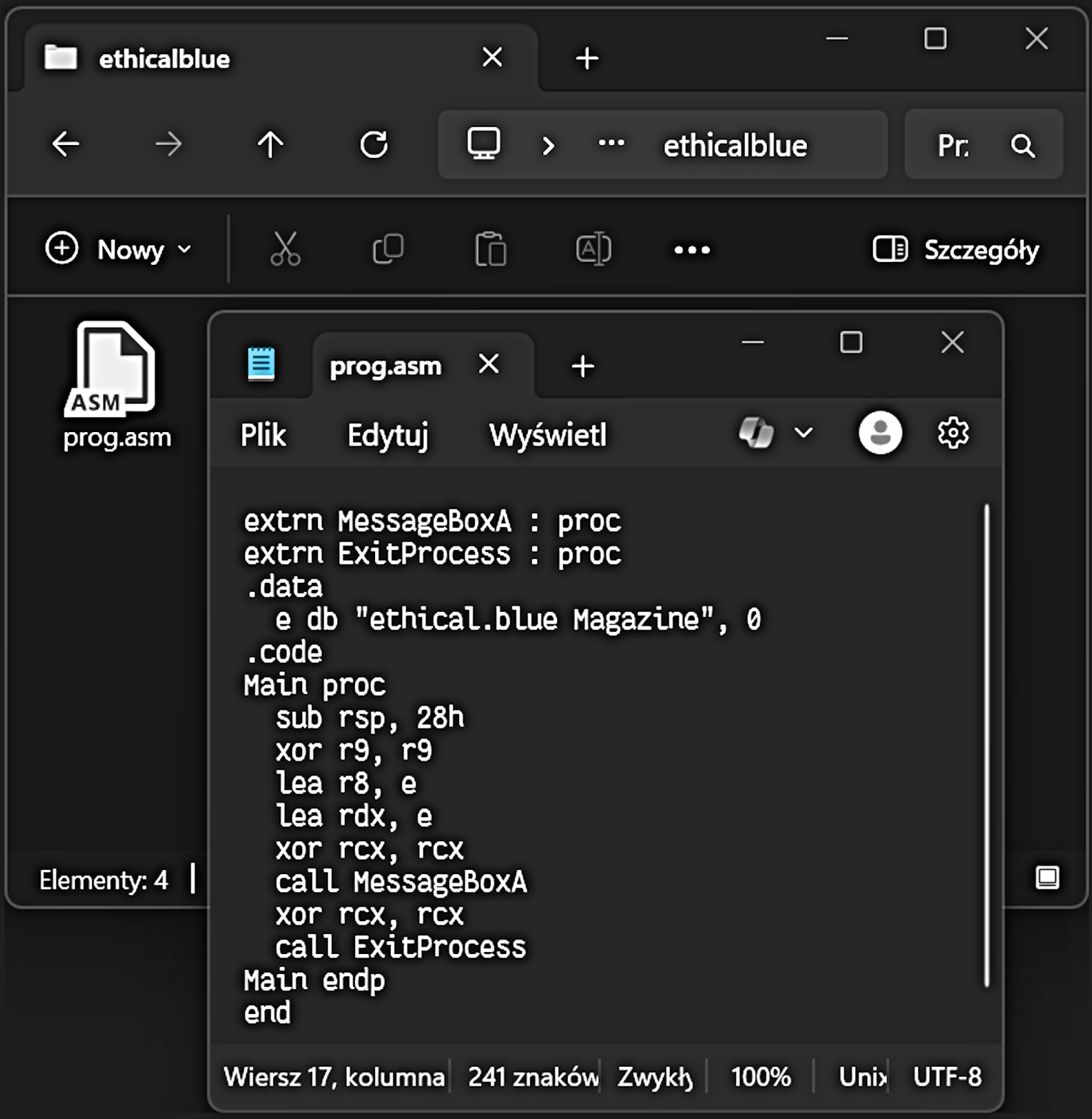
Now, let's create a sample program in MASM x64 Assembly Language. Locate the x64 Native Tools Command Prompt for Visual Studio shortcut in the Programs menu and launch it.

The source code below presents a sample program in MASM x64 syntax.

```
extrn MessageBoxA : proc
extrn ExitProcess : proc
.data
  e db "ethical.blue Magazine", 0
.code
  Main proc
    sub rsp, 28h
    xor r9, r9
    lea r8, e
    lea rdx, e
    xor rcx, rcx
    call MessageBoxA
    xor rcx, rcx
    call ExitProcess
  Main endp
end
```

Select an accessible folder on your Microsoft Windows operating system and create a text file with a .asm extension. The file should contain the example source code. Paste the code using a text editor like Notepad (notepad.exe).

Example Program in MASM x64



Author: Dawid Farbaniec

In the x64 Native Tools Command Prompt for VS, navigate to the directory with the sample code using the CD (change directory) command.

For example:

```
CD "C:\ethicalblue"
```

Next, execute the command provided below to assemble the prog.asm source code into the prog.exe executable (Figure 2).

Build command:

```
ml64.exe /quiet prog.asm /link /
   subsystem:windows /defaultlib:
   kernel32.lib /defaultlib:user32.
   lib /entry:Main /out:prog.exe
```

After a successful build, a new executable file (prog.exe) appears in the specified folder.

Execute the program by double-clicking prog.exe icon to verify its functionality.

The MessageBox dialog with the specified text then appears. Clicking OK terminates the program and returns control to Microsoft Windows operating system. See Figure 3.

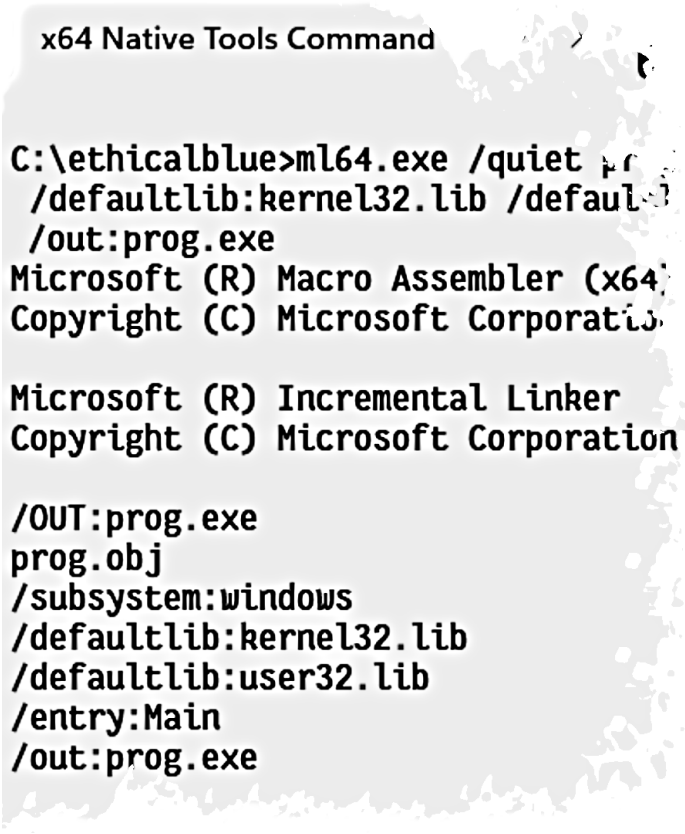


Figure 2. ML64.exe (MASM for x64)

Figure 3. Program with a Simple MessageBox

General Purpose Registers

A register is a small, high-speed storage element with a fixed size, such as 8, 16, 32, or 64 bits, or, in some architectures, even 2048 bits. General Purpose Registers (GPRs) and their default use are presented below as a simple reference. [1] [6]

RAX (Accumulator)

Common use: Accumulator for operands and results.

In Windows x64, RAX is the return value register. This means volatile, or potentially changed by a callee on return.

The RAX register (64 bits) can be divided into smaller parts.

- EAX (32 bits, low doubleword)
- AX (16 bits, low word)
- AH (8 bits, high byte)
- AL (8 bits, low byte)

RBX (Base)

Common use: Address generation in old 16-bit code.

In Windows x64, RBX register is nonvolatile and must be preserved by callee.

The RBX register (64 bits) can be divided into smaller parts.

- EBX (32 bits, low doubleword)
- BX (16 bits, low word)
- BH (8 bits, high byte)
- BL (8 bits, low byte)

RCX (Counter)

Common use: Iteration count for loops. Bit index in shift and rotate instructions.

In Windows x64, RCX register is volatile, or potentially changed by a callee on return.

The RCX register (64 bits) can be divided into smaller parts.

- ECX (32 bits, low doubleword)
- CX (16 bits, low word)
- CH (8 bits, high byte)
- CL (8 bits, low byte)

RDX (Data)

Common use: Operand for arithmetic instructions.

In Windows x64, RDX register is volatile, or potentially changed by a callee on return.

The RDX register (64 bits) can be divided into smaller parts.

- EDX (32 bits, low doubleword)
- DX (16 bits, low word)
- DH (8 bits, high byte)
- DL (8 bits, low byte)

RSI (Source Index)

Common use: Memory address of source operand for string instructions.

In Windows x64, RSI register is nonvolatile and must be preserved by callee.

The RSI register (64 bits) can be divided into smaller parts.

- ESI (32 bits, low doubleword)
- SI (16 bits, low word)
- SIL (8 bits, low byte)

RDI (Destination Index)

Common use: Memory address of destination operand for string instructions.

In Windows x64, RDI register is nonvolatile and must be preserved by callee.

The RDI register (64 bits) can be divided into smaller parts.

- EDI (32 bits, low doubleword)
- DI (16 bits, low word)
- DIL (8 bits, low byte)

RSP (Stack Pointer)

Common use: Memory address of last stack entry (top of stack).

In Windows x64, RSP register is nonvolatile (stack pointer).

The RSP register (64 bits) can be divided into smaller parts.

- ESP (32 bits, low doubleword)
- SP (16 bits, low word)
- SPL (8 bits, low byte)

RBP (Frame Pointer)

Common use: Memory address of frame pointer.

In Windows x64, RBP register is nonvolatile and must be preserved by callee.

The RBP register (64 bits) can be divided into smaller parts.

- EBP (32 bits, low doubleword)
- BP (16 bits, low word)
- BPL (8 bits, low byte)

R8 .. R9 (Extra)

Common use: No implicit uses.

In Windows x64, R8 .. R9 registers are volatile, or potentially changed by a callee on return.

The R8 .. R9 registers (64 bits) can be divided into smaller parts.

- R8D .. R9D (32 bits, low doubleword)
- R8W .. R9W (16 bits, low word)
- R8B .. R9B (8 bits, low byte)

R10 .. R11 (Extra)

Common use: Used in SYSCALL and SYSRET instructions.

In Windows x64, R10 .. R11 registers (64 bits) must be preserved as needed by caller.

The R10 .. R11 registers (64 bits) can be divided into smaller parts.

- R10D .. R11D (32 bits, low doubleword)
- R10W .. R11W (16 bits, low word)
- R10B .. R11B (8 bits, low byte)

R12 .. R15 (Extra)

Default use: No implicit uses.

In Windows x64, R12 .. R15 registers (64 bits) are nonvolatile and must be preserved by callee.

The R12 .. R15 registers (64 bits) can be divided into smaller parts.

- R12D .. R15D (32 bits, low doubleword)
- R12W .. R15W (16 bits, low word)
- R12B .. R15B (8 bits, low byte)

Flat Memory Model

Virtual memory is a large virtual address space that is mapped to a smaller physical address space. Physical memory resides in RAM, and portions of memory may be swapped to disk as needed. The flat memory model is also known as unsegmented. Memory is visible to a program as a continuous, linear address space. It is byte-addressable. An address is called a linear address and is equal to the effective address. [1] [6]

MASM x64 Directives

Directives do not generate machine code directly, but they affect the build process and organization of the program.

Let's analyze the sample program.

See Scriptum 1.

The `extern` (or `extrn`) directive defines an external procedure. Procedures defined here are: `MessageBoxA`, which displays the dialog box, and `ExitProcess`, which terminates the application and returns an exit code to Windows.

The `.data` directive starts the initialized data section, while the `.code` directive starts the code section.

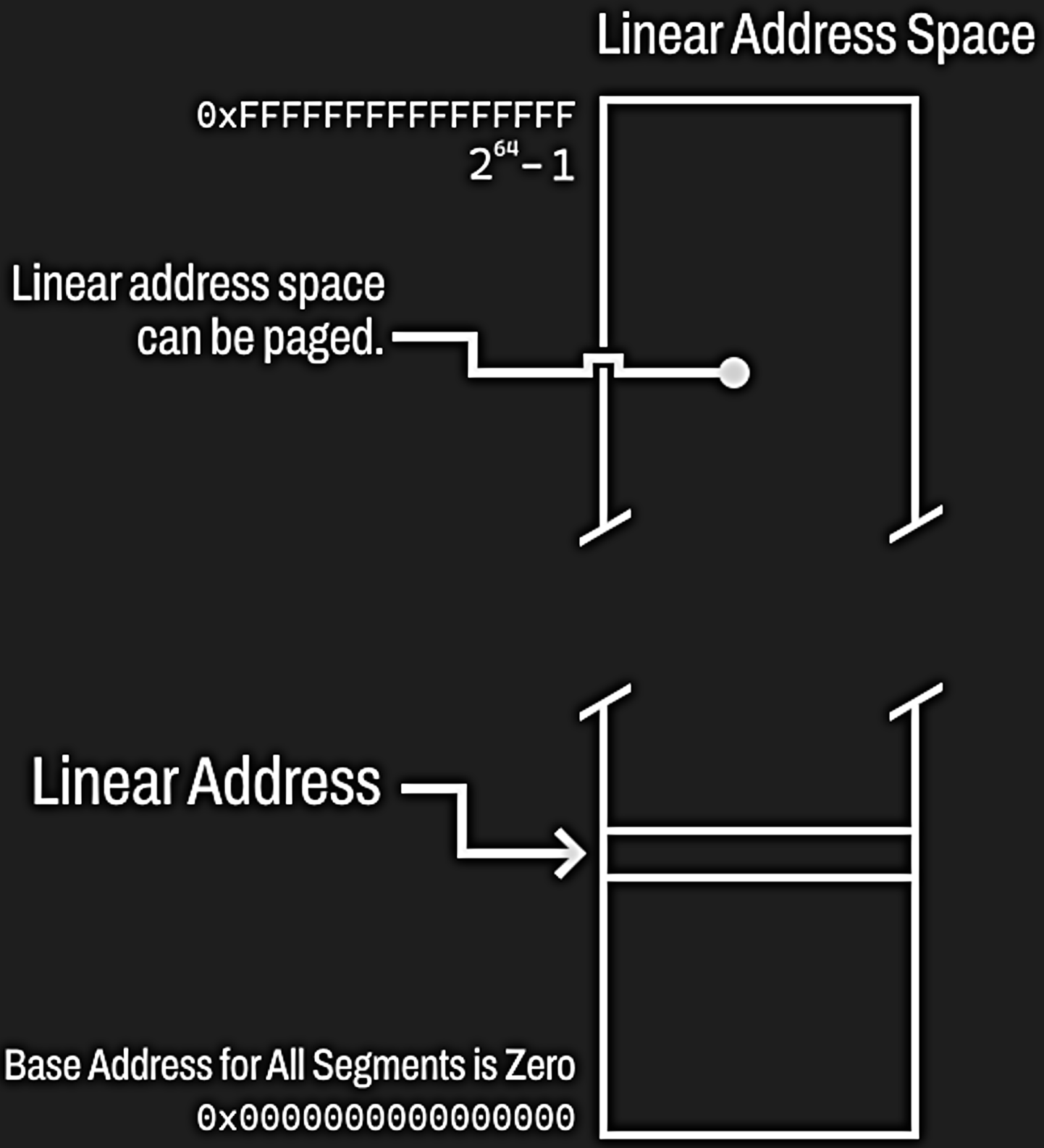
```
extrn MessageBoxA : proc
extrn ExitProcess : proc
.data
  e db "ethical.blue Magazine", 0
.code
  Main proc
    sub rsp, 28h
    xor r9, r9
    lea r8, e
    lea rdx, e
    xor rcx, rcx
    call MessageBoxA
    xor rcx, rcx
    call ExitProcess
  Main endp
end
```

Scriptum 1. Sample Program (MASM for x64)

The `db` (or `byte`) directive means define byte and allocates bytes with a specified initializer. In the sample program there is a zero-terminated ASCII string defined.

The `proc` and `endp` directives mark the start and end of a procedure block with a specified label. The `end` directive marks the end of the module.

Flat Memory Model



Author: Dawid Farbaniec

Microsoft x64 Calling Convention

Convention means an accepted way of doing something. It refers to established norms. In the sample program (Scriptum 2), Main is the caller, while CreateFileA is the callee. The first four integer arguments are passed to a function in the RCX, RDX, R8, and R9 registers. The fifth and more arguments are passed on the stack. The stack must be 16-byte aligned. The caller allocates shadow space on the program stack so the callee can save the first four registers and passed arguments.

The RSP register points to the top of the stack. To allocate space on the stack, decrease RSP. For example, `sub rsp, 28h` subtracts 40 (decimal) from current RSP value. The stack must be 16-byte aligned. Notice that there is a return address placed on the stack by default, which gives 40 + 8 = 48 bytes. Forty-eight is divisible by sixteen without remainder, ensuring 16-byte alignment.

It was previously mentioned that the first four integer arguments are passed to a function in the RCX, RDX, R8, and R9 registers. The fifth and more arguments are passed on the stack. Let's examine a function call with seven arguments (Scriptum 2).

```
extrn CreateFileA : proc
extrn ExitProcess : proc
.data
  ;[...]
.code
  Main proc
    ;[...]
    sub rsp, 38h
    mov qword ptr [rsp+30h], 0
    mov qword ptr [rsp+28h], \
      FILE_ATTRIBUTE_NORMAL
    mov qword ptr [rsp+20h], \
      CREATE_NEW
    xor r9, r9
    xor r8, r8
    mov rdx, GENERIC_WRITE
    lea rcx, szFileName
    call CreateFileA
    ;[...]
  Main endp
end
```

Scriptum 2. CreateFileA Function Call (x64)

The RSP register points to the top of the stack, so instruction `mov qword ptr [rsp+20h], CREATE_NEW` copies the `CREATE_NEW` constant to a 20h offset from the top of the stack.

The mechanism is similar for other arguments (only the offset changes).

x64 Instruction Set Reference

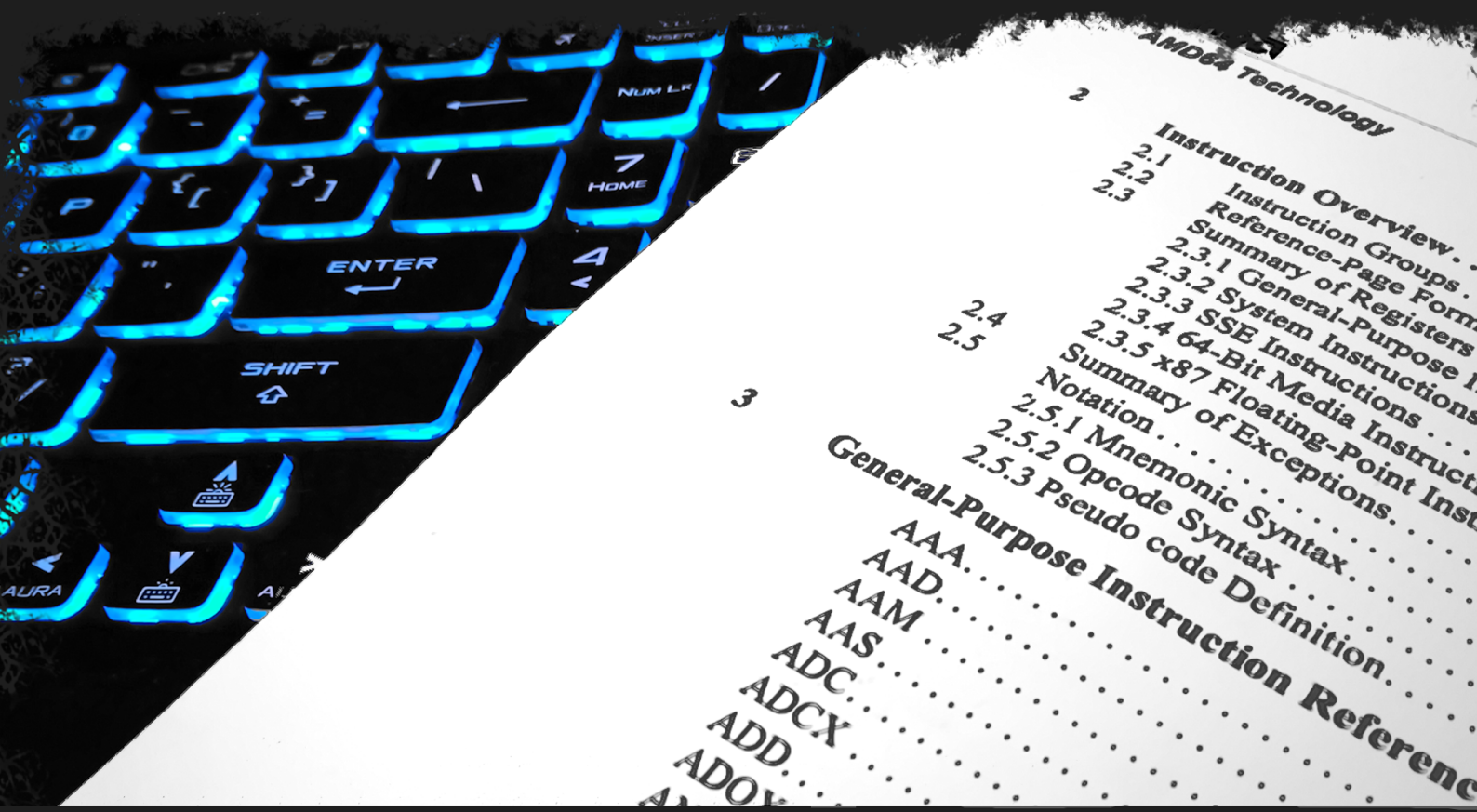
Every time you encounter an unfamiliar instruction, you should search for a description in the AMD64 Architecture Programmer's Manual or The Intel 64 and IA-32 Architectures Software Developer's Manual. [1] [6]

These documents can be tough to learn from without prior knowledge, but when it comes to descriptions of available instructions, there's no better or more authoritative source of knowledge.

Start by reading about the MOV instruction.
Next, read about ADD, SUB, MUL, DIV, AND, OR, XOR, NOT, ANDN, PUSH, POP, CALL, RET, LOOP, and Jcc (JMP, JNE, JE etc.).

More instructions will become clear over time.

Even in this text, you may encounter an unfamiliar instruction.
Don't panic — open the manual.



Sample.TimeElemental.Win64.A

Somewhere in 3OHA, an anomaly zone. Morph finds a small (3072 bytes) executable file on an old machine in a forgotten part of the laboratory.

When executed in an isolated environment, the sample shows no graphical user interface, prints no messages to the screen, and generates no network traffic.

I think this could be an educational artifact from exercises performed a long time ago. Some reverse engineering is needed to confirm the program's behavior. — said Morph.

Download the sample program.

`https://ethical.blue/en/time.zip`

It is a good idea to generate a hash of a sample before analysis. Hashes, such as SHA512, can be compared to fingerprints. A small change in data integrity completely changes the hash.

First, in the PowerShell console window, change the current directory to a folder with a sample.

For example:

```
cd "C:\ethicalblue\"
```

A hash (SHA512) can be generated using the following PowerShell command-let.

```
(Get-FileHash -Path time.exe -
   Algorithm SHA512).Hash.ToLower();
```

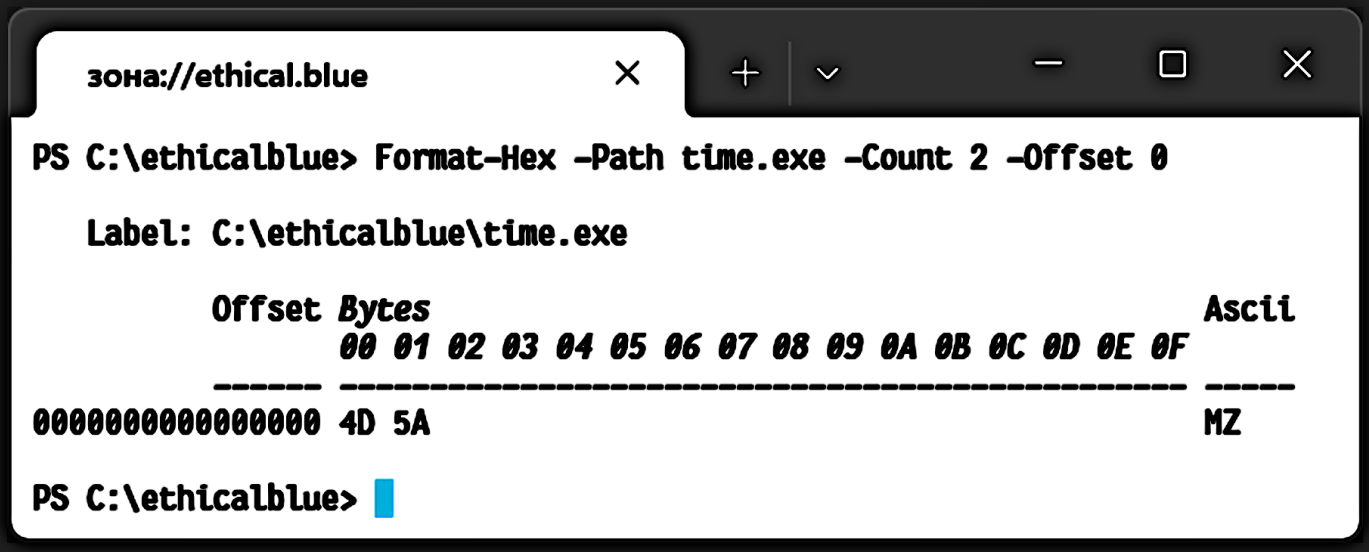
Hash of a found sample is `dbdee6dba5d655bf3bc70b90b4dbb57e7ccb5eb42c6ad01f94fb71ff46bdb43d666677957c56cec8fbaeffb1d8e9e67ab0fbdd5e77b8017700cbd3232817662b`.

It's worth noting that there are two main types of sample analysis.

- Static analysis examines file characteristics and disassembled code without executing the code.
- Dynamic analysis examines program's behavior by placing breakpoints, stepping through the code, intercepting network traffic, and using automated sandboxes to run the sample in a virtual machine and collect execution logs.

Found sample is a Portable Executable (PE64) file. Portable Executable file format is described in Microsoft Portable Executable and Common Object File Format Specification. [8]

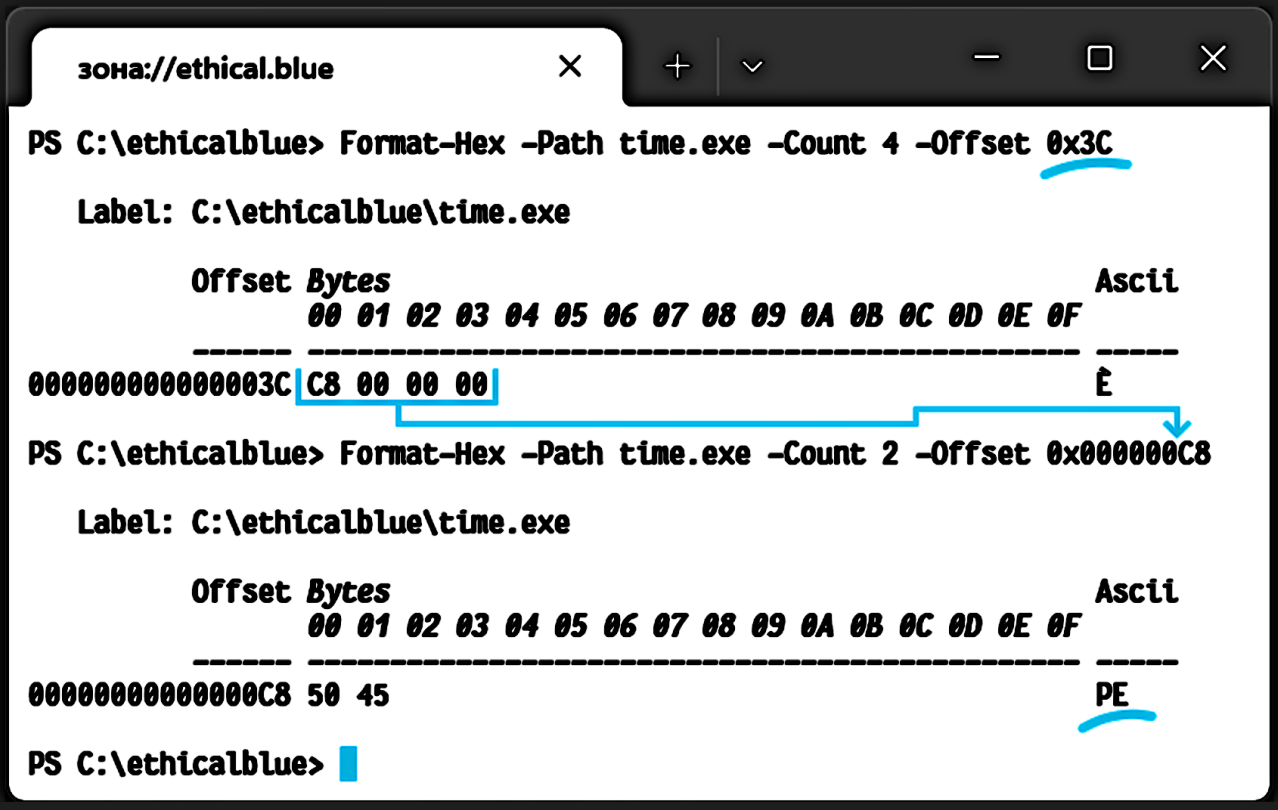
The first thing that stands out in a memory dump of the executable file is the MZ signature at the beginning.



Documentation [8] states that

> At location 0x3c, the stub has the file offset to the PE signature. This information enables Windows to properly execute the image file, even though it has an MS-DOS stub. This file offset is placed at location 0x3c during linking.

As proof, let's read four bytes from offset 0x3C. These bytes represent the offset to the PE signature.



Following the documentation [8] the next field is the Machine Type.

```
зона://ethical.blue

PS C:\ethicalblue> Format-Hex -Path time.exe -Count 2 -Offset (0x000000C8+0x04)

   Label: C:\ethicalblue\time.exe

          Offset Bytes                                                   Ascii
                 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
          ------ ----------------------------------------------- -----
00000000000000CC 64 86                                           d

PS C:\ethicalblue>
```

The 0x8664 bytes are a Windows constant IMAGE_FILE_MACHINE_AMD64 from Visual C++ header winnt.h

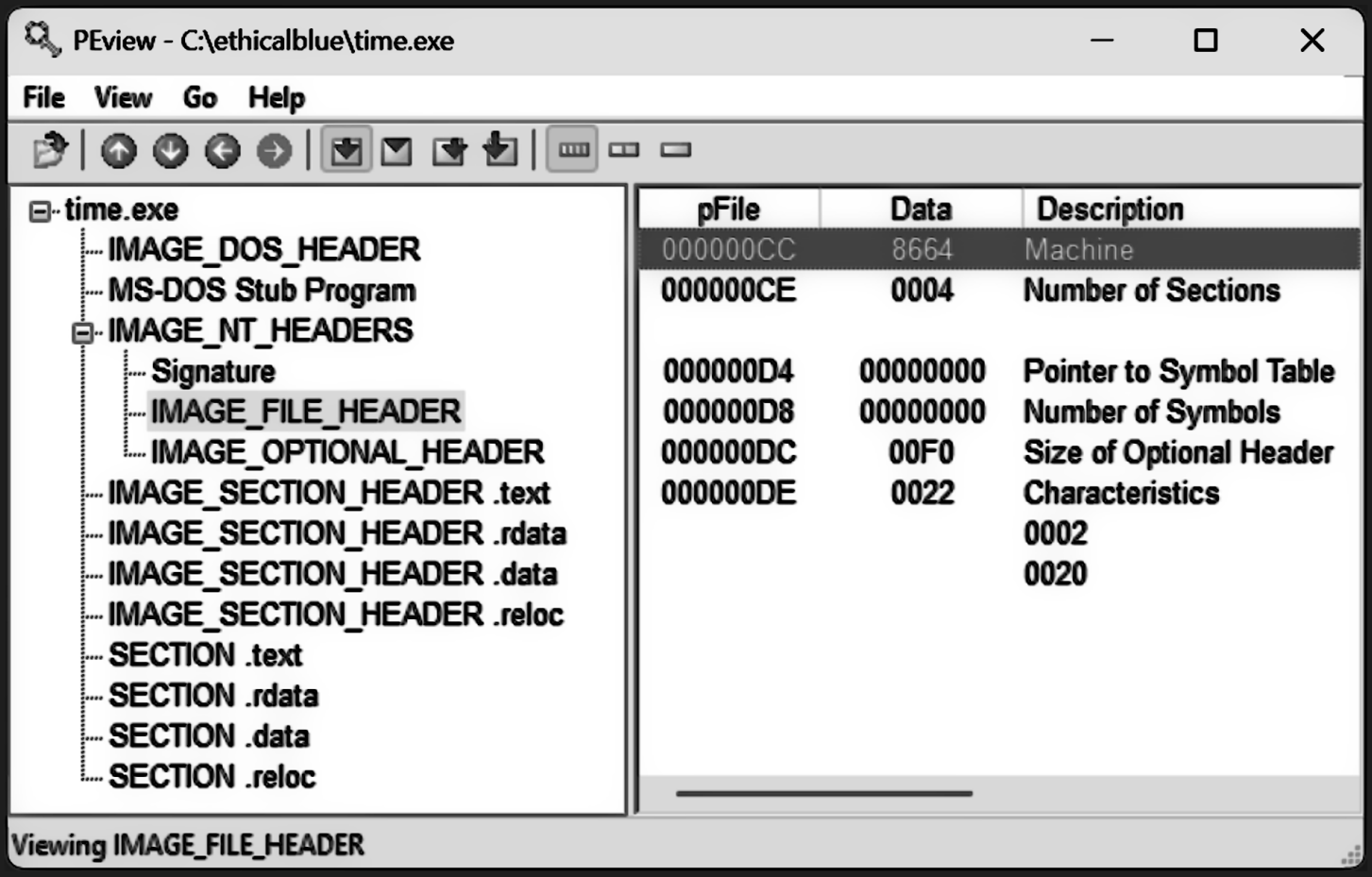
```
winnt.h

Plik  Edytuj  Wyświetl

#define IMAGE_FILE_MACHINE_MIPS16         0x0266  // MIPS
#define IMAGE_FILE_MACHINE_ALPHA64        0x0284  // ALPHA64
#define IMAGE_FILE_MACHINE_MIPSFPU        0x0366  // MIPS
#define IMAGE_FILE_MACHINE_MIPSFPU16      0x0466  // MIPS
#define IMAGE_FILE_MACHINE_AXP64          IMAGE_FILE_MACHINE_ALPHA64
#define IMAGE_FILE_MACHINE_TRICORE        0x0520  // Infineon
#define IMAGE_FILE_MACHINE_CEF            0x0CEF
#define IMAGE_FILE_MACHINE_EBC            0x0EBC  // EFI Byte Code
#define IMAGE_FILE_MACHINE_AMD64          0x8664  // AMD64 (K8)
#define IMAGE_FILE_MACHINE_M32R           0x9041  // M32R little-endian
#define IMAGE_FILE_MACHINE_ARM64          0xAA64  // ARM64 Little-Endian
#define IMAGE_FILE_MACHINE_CEE            0xC0EE

Wiersz 19485, kolumna 67 | 66 z 841 491 | Zwykły teks | 100% | Windows (C | UTF-8
```

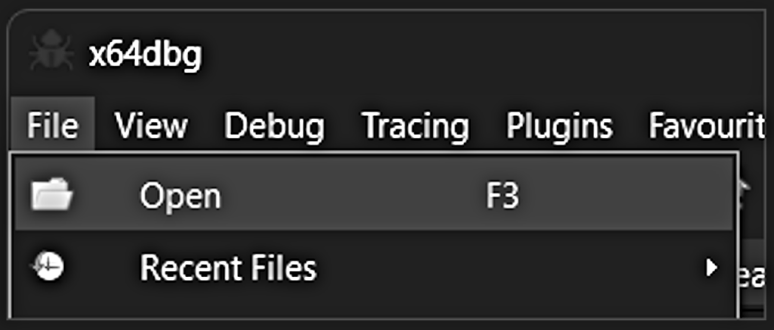
Manually parsing bytes provides a good educational introduction, but for smooth work with PE/COFF files, tools like PEview are recommended. It's worth noting that the sample has four sections. The .text section contains machine code, and the .data section contains the program's data.



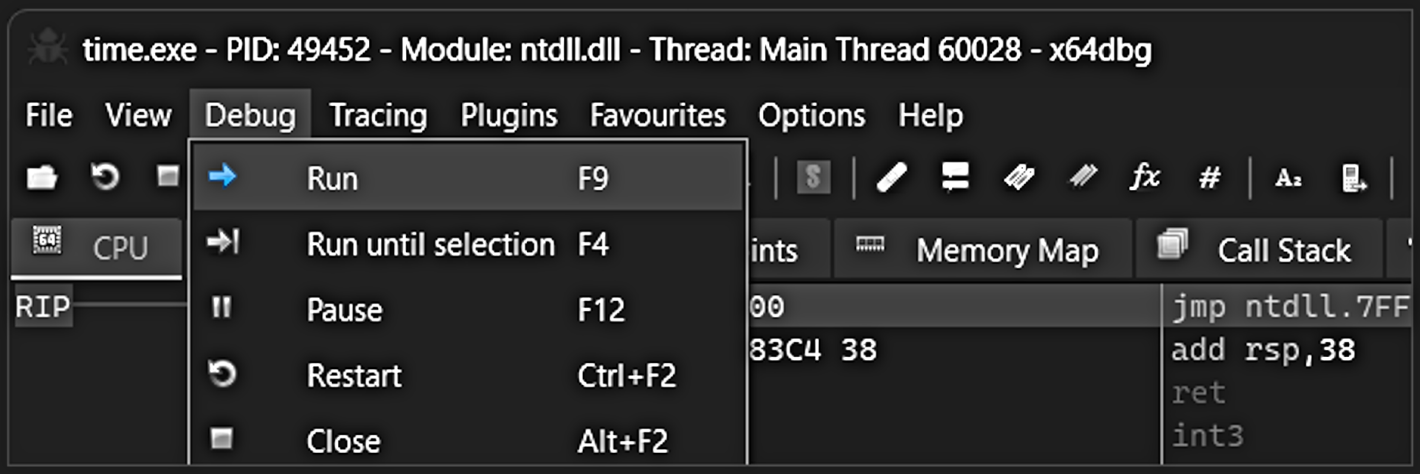
It's time to switch to a more advanced tool, like x64dbg.

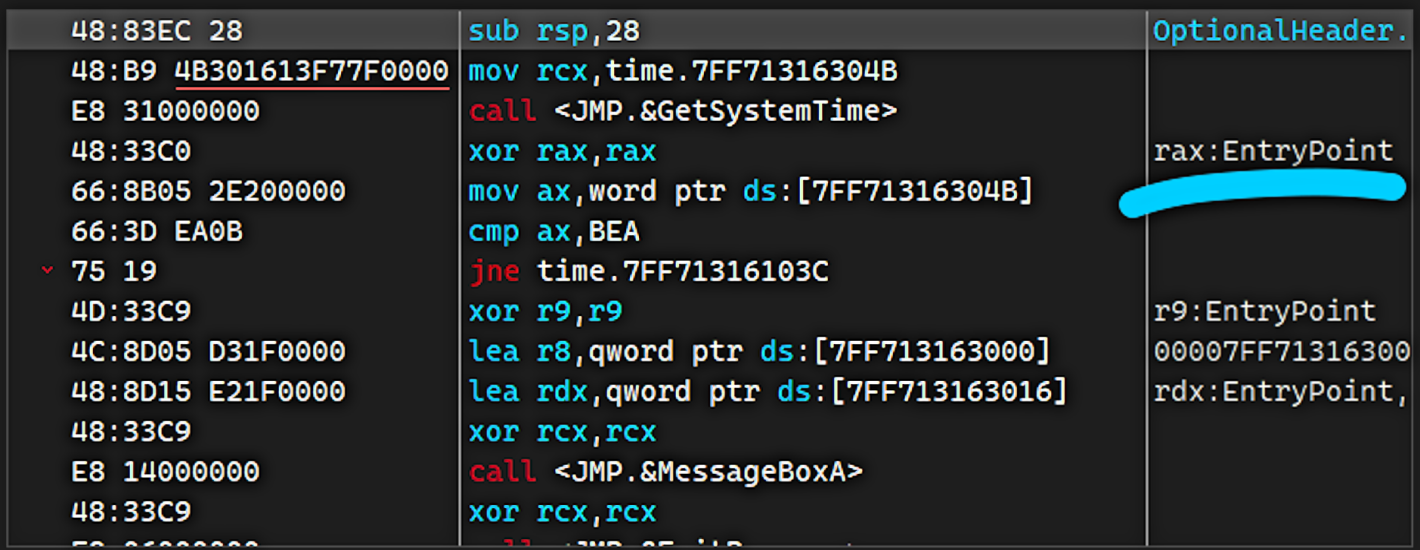
If the tool is not installed, open x64dbg.com and download the program.

Open the sample in x64dbg.

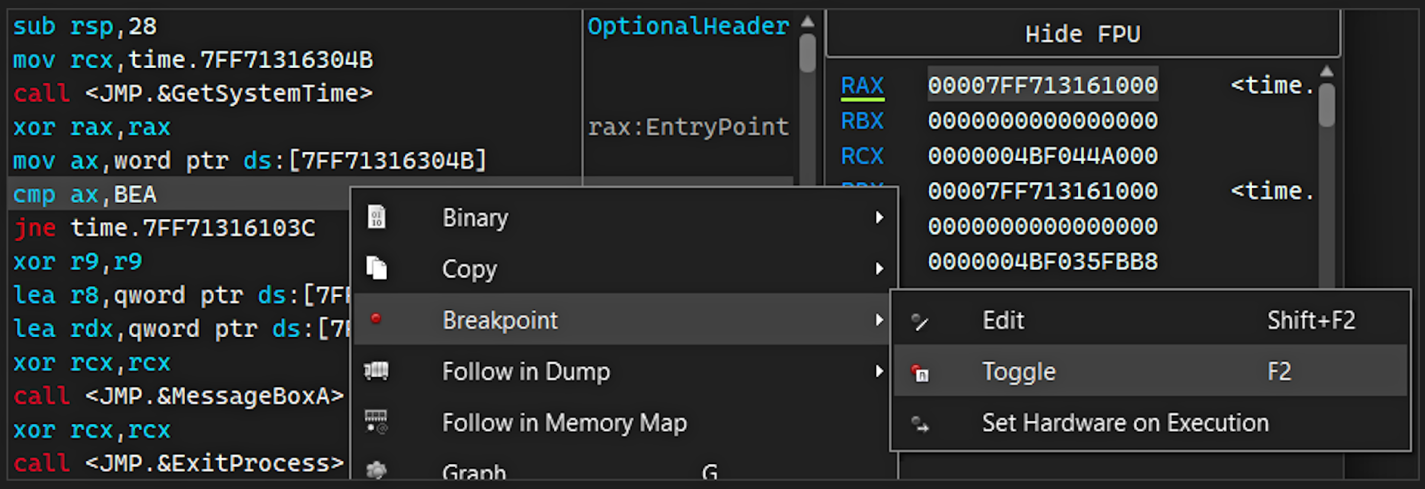


From the top menu, click Debug » Run until execution reaches the Entry Point. If the Entry Point has not been reached after loading the executable, click Debug » Run once more.

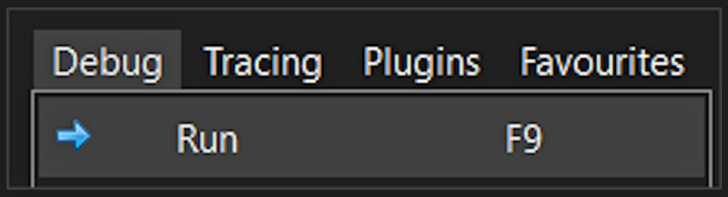




There is a call to the GetSystemTime function, and the returned value in the AX register is compared to 0xBEA using the CMP instruction. Right-click and toggle a breakpoint on the CMP instruction line.



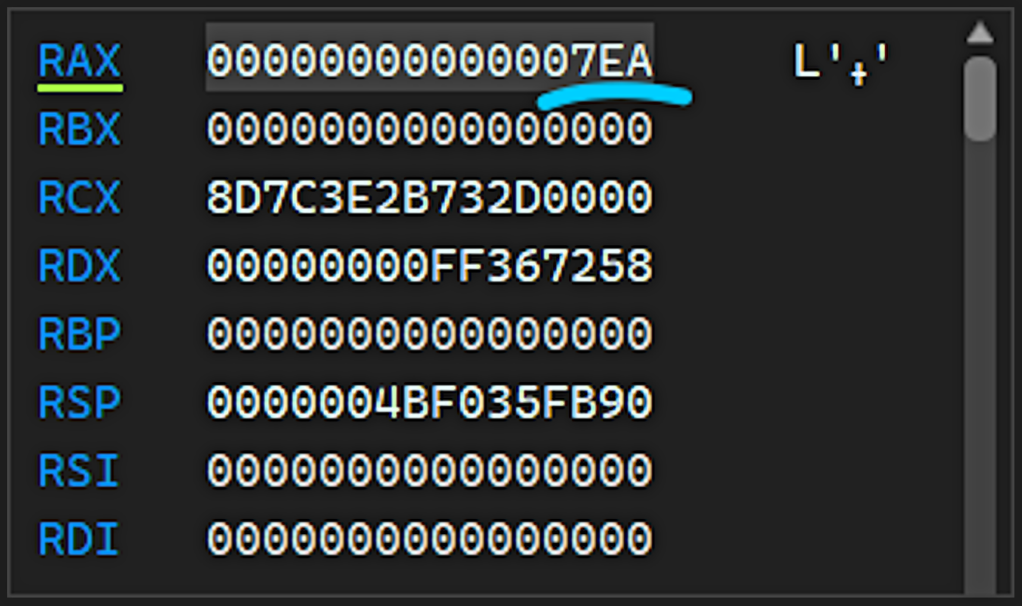
Select Debug » Run to execute code until the breakpoint is hit.



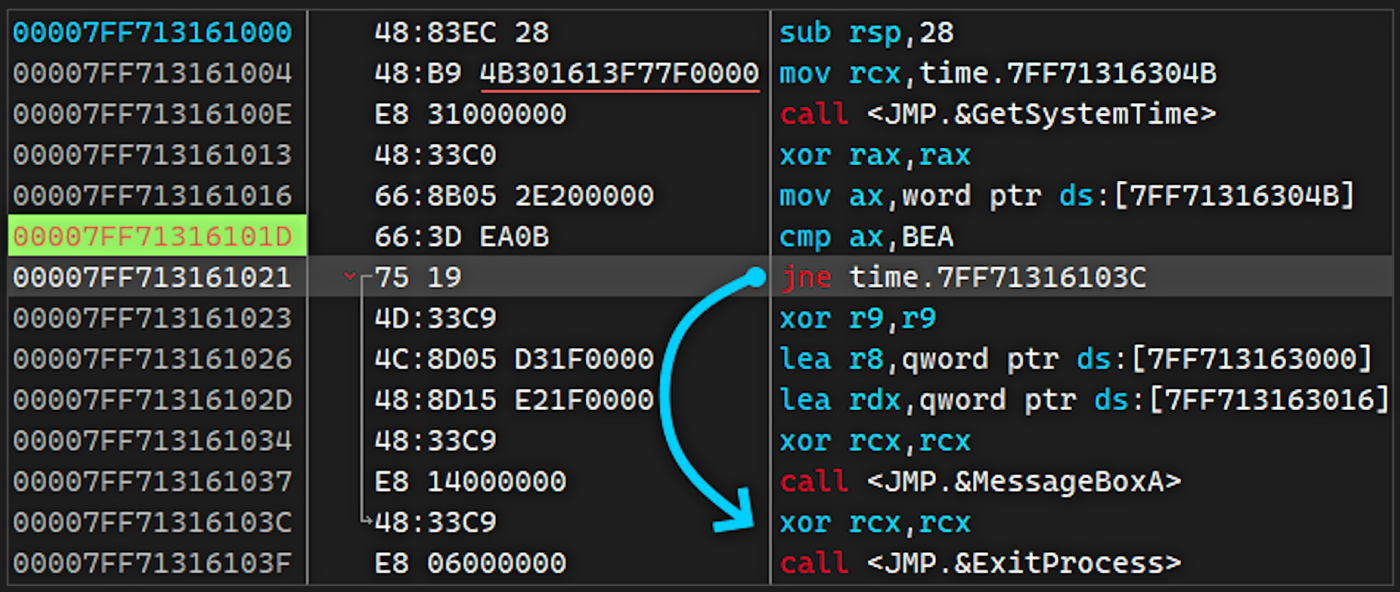
Documentation [1] states that

> The CMP instruction performs subtraction of the second operand (source) from the first operand (destination), like the SUB instruction, but it does not store the resulting value in the destination operand. It leaves both operands intact. The only effect of the CMP instruction is to set or clear the arithmetic flags (OF, SF, ZF, AF, CF, PF) according to the result of subtraction.

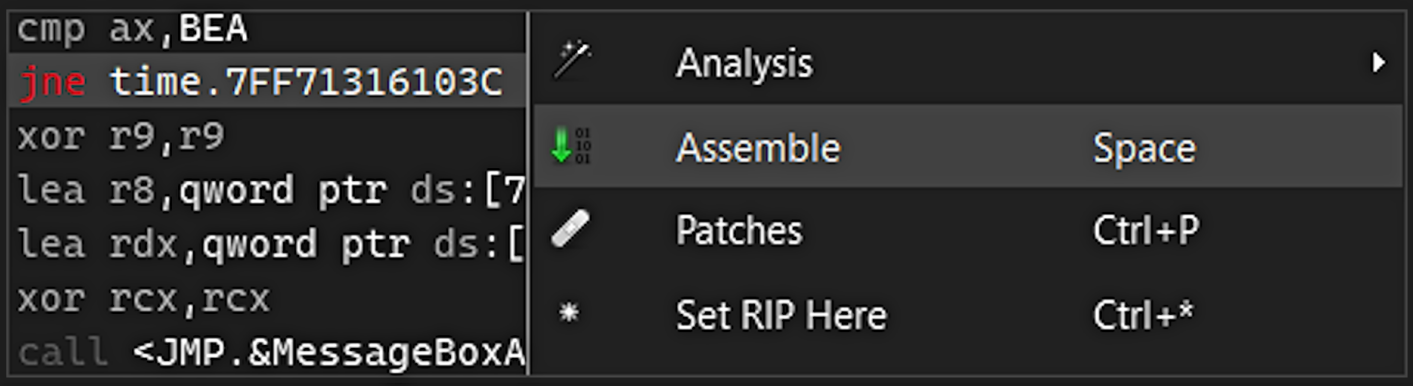
The CMP instruction operands are 0x7EA and 0xBEA. In decimal, these values are 2026 and 3050, respectively. Therefore, the instruction compares the year returned by GetSystemTime to 3050.



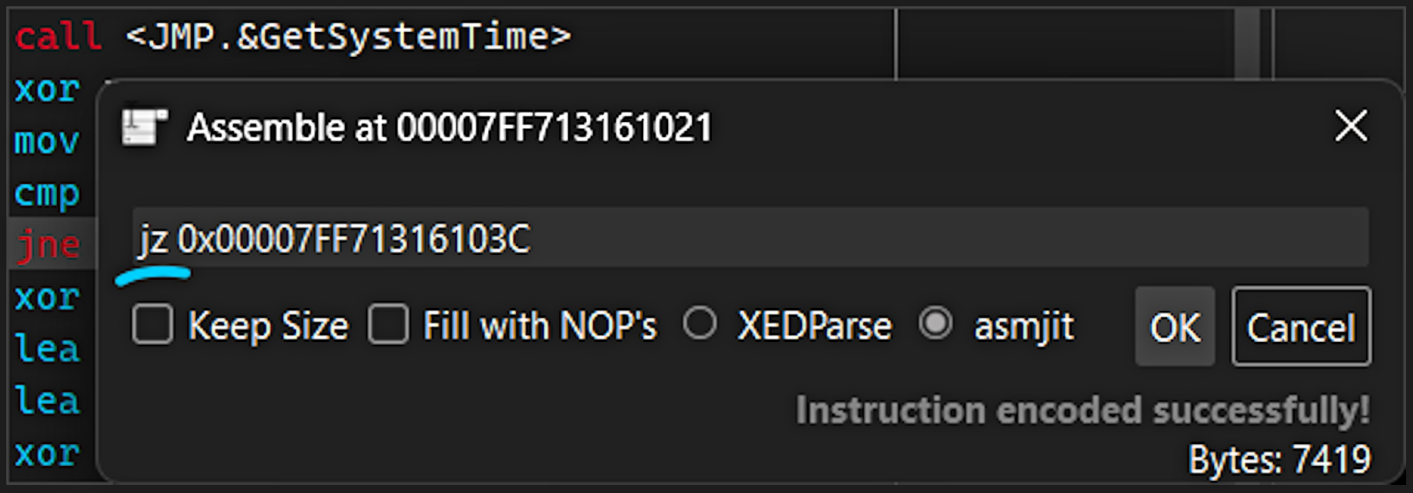
If the year is not 3050, the ExitProcess function is called.
The program terminates, and no MessageBox dialog is shown.



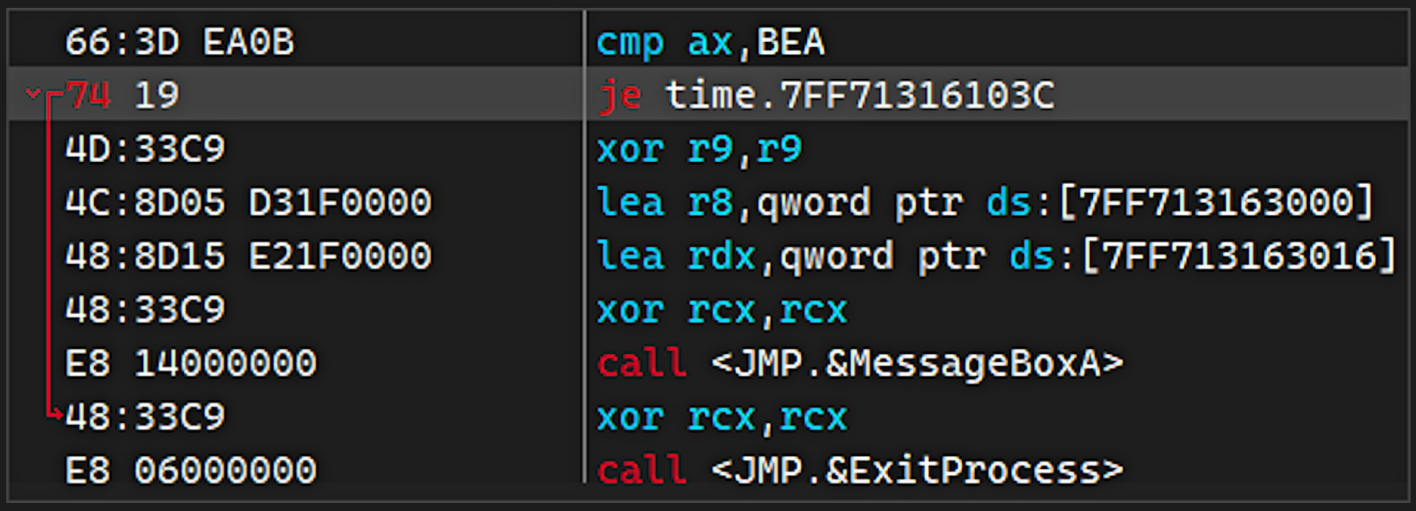
A simple way to see the MessageBox dialog is to change the JNE (Jump If Not Equal) instruction to JE (Jump If Equal). Right-click the JNE instruction and select Assemble.



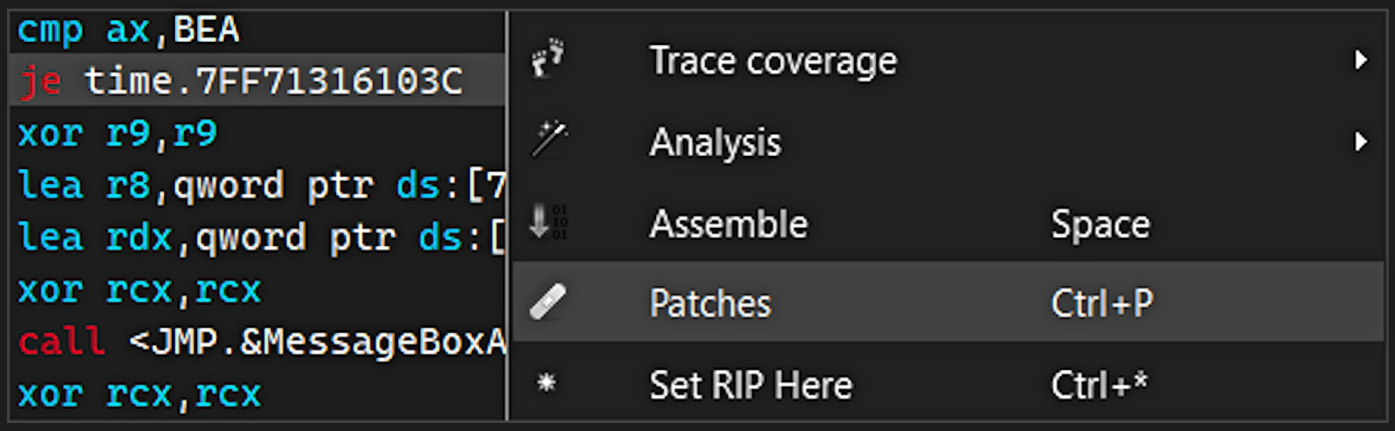
The JNE (Jump If Not Equal) instruction is sometimes shown as JNZ (Jump If Not Zero), and JE (Jump If Equal) as JZ (Jump If Zero). Enter JZ in the Assemble text box, click OK, and close the window.



The conditional jump is changed.

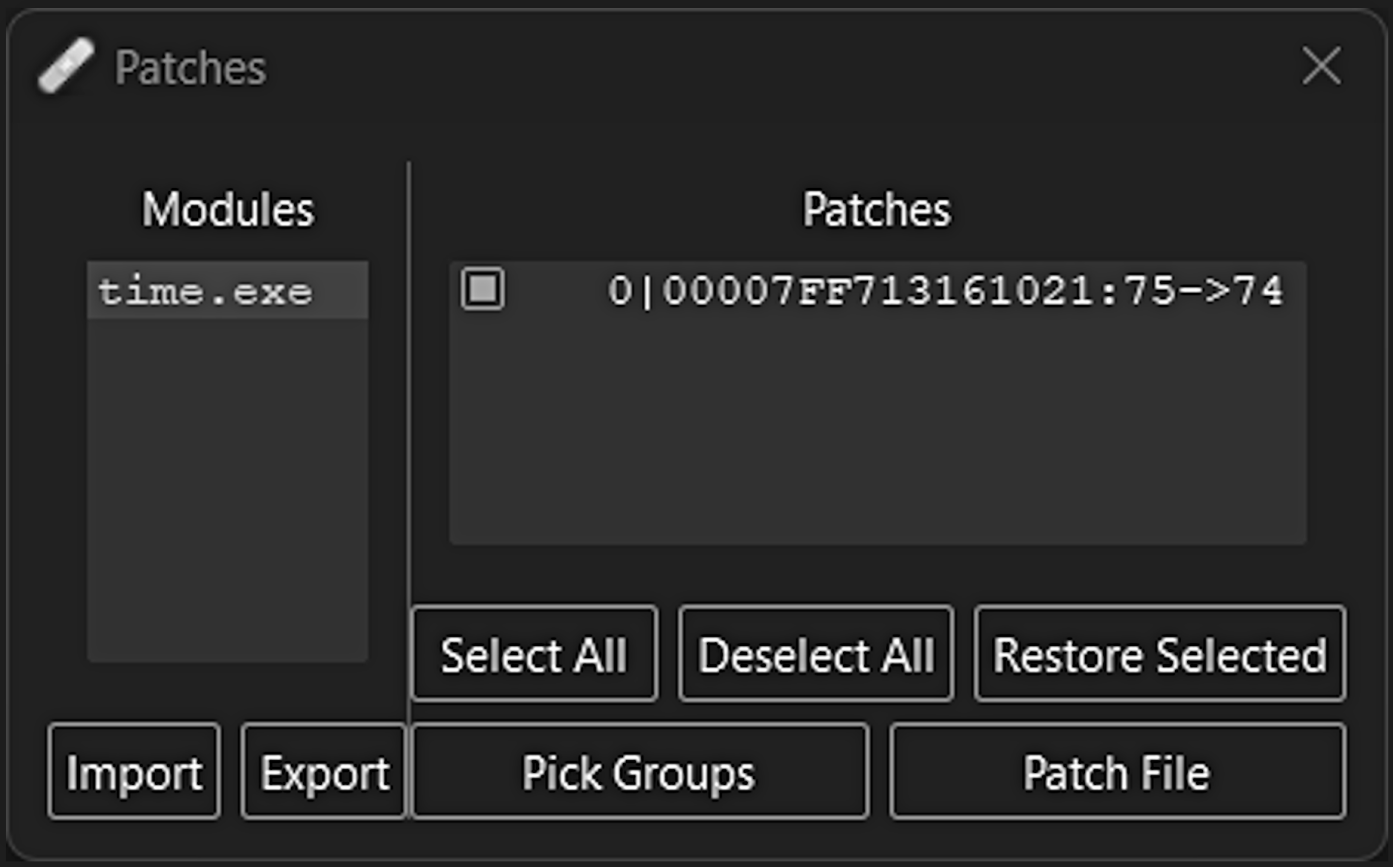


Right-click on disassembly listing and select Patches.

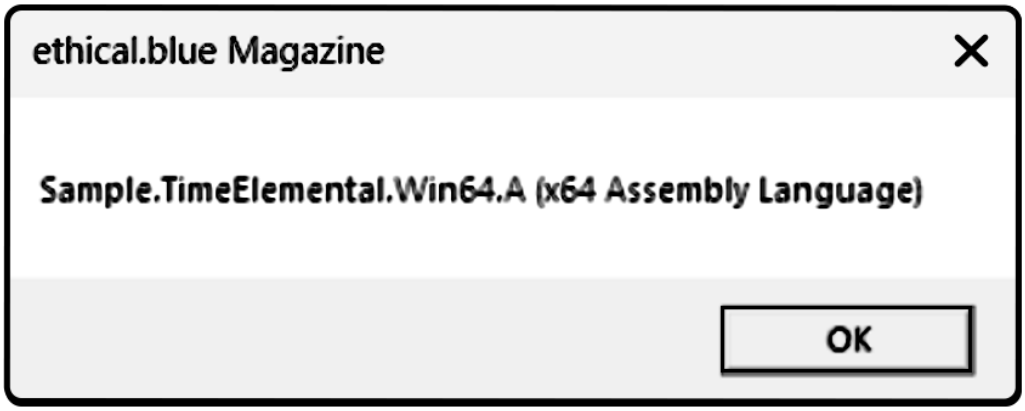


The Patches window displays the modified bytes in the analyzed file. Changing the mnemonic from JNE to JE changes the opcode from 0x75 to 0x74. Only one byte is modified.

Click Patch File to save the changes to a file.



Let's execute the modified sample file. The MessageBox dialog will be displayed.



The next educational sample joined my collection of weird programs. — said Morph.

Machine:

`3oha://fe80::ae54:3ccc:1c43:7a33`

Connection closed.

Time Elemental

ethical.blue / Art

Dawid Farbaniec

Bibliography

[1] Advanced Micro Devices, Inc. (AMD), *AMD64 Architecture Programmer's Manual*, 2024.

[2] Advanced Micro Devices, Inc. (AMD), *AMD Secure Random Number Generator Library*, 2025.

[3] Advanced Micro Devices, Inc. (AMD), *System V Application Binary Interface*, 2025.

[4] Intel Corporation, *Intel Advanced Vector Extensions 10.2 Architecture Specification*, 2025.

[5] Intel Corporation, *Intel Architecture Instruction Set Extensions and Future Features*, 2025.

[6] Intel Corporation, *The Intel 64 and IA-32 Architectures Software Developer's Manual*, 2025.

[7] S. P. Kulkarni, D. E. Huang, and E. W. Bethel, *From Bits to Qubits: Challenges in Classical-Quantum Integration*, 2025.

[8] Microsoft Corporation, *Microsoft Portable Executable and Common Object File Format Specification*, 2025.