Reverse Engineering on Windows

Cyber Skill Level Up UTM

CYBER SKILLS LEVEL-UP! 2025

RE:HACK

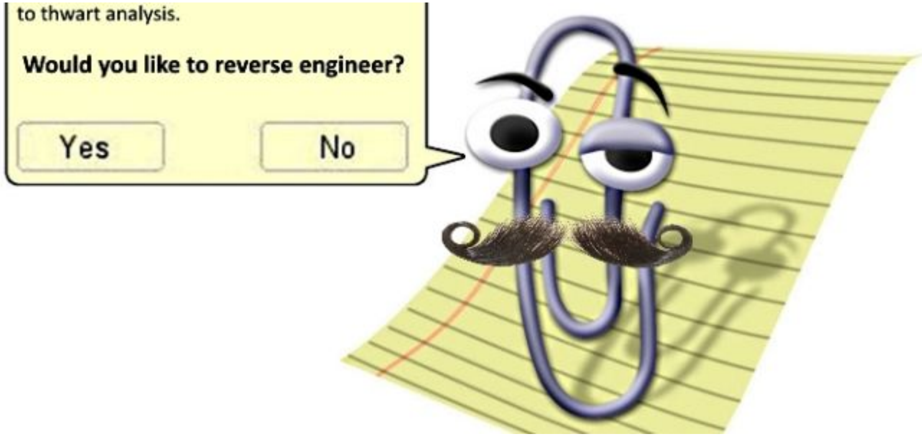
re:search + re:scrutiny + re:educate

./whoami

- Shreethaar (0x251e)
- UUM CS Student (Final Year)
- RE:UN10N
- MCC 2024 Alumni
- Interest: DFIR, RE, OSINT

./toc

1. Intro to RE
2. PE file format
3. x86 arch
4. Basic C and ASM
5. Cracking crackmes

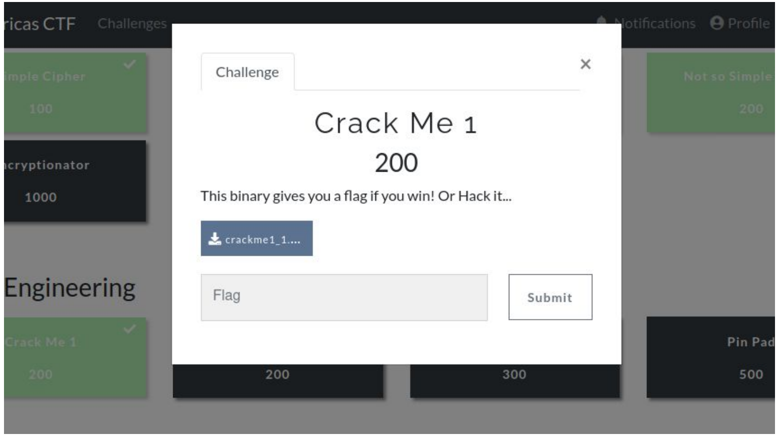


./intro_to_RE

RE is like taking apart a complex puzzle to figure out how it works

In CTF, often you are given a binary to get the flag

We need to decompile or disassemble it, identify what is the binary suppose to do.



Crack Me 1

200

This binary gives you a flag if you win! Or Hack it...

./intro_to_RE

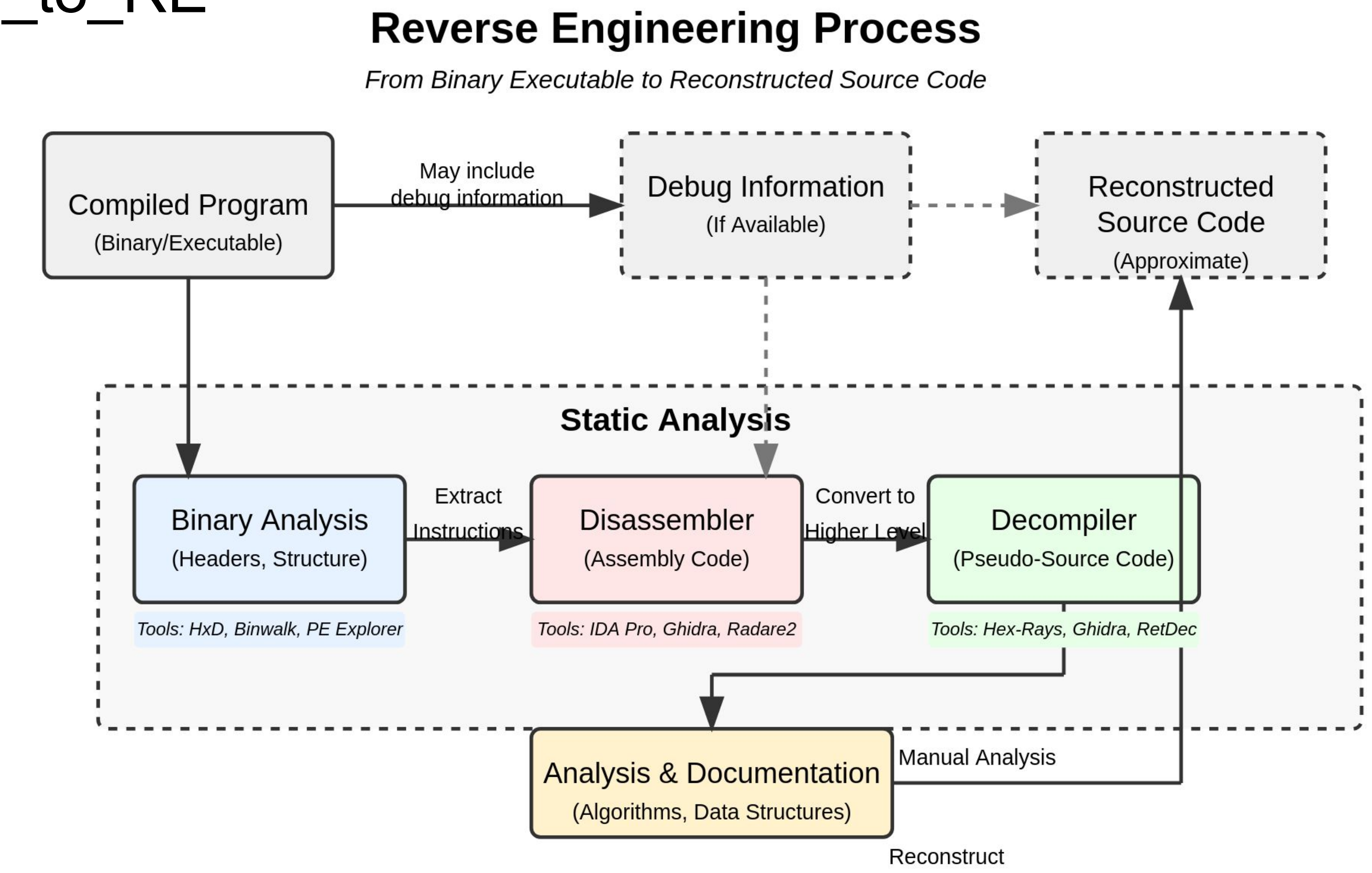
Why Do CTF include RE ?

1. Vulnerability research
2. Malware analysis
3. Binary exploitation
4. Forensic

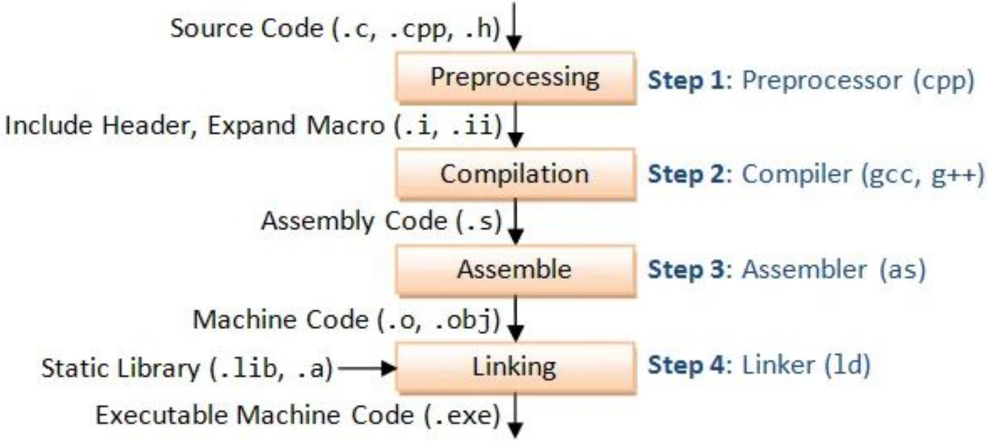
Benefits of learning RE ?

1. Gain deep understanding of how machines works
2. Relate both low level with OS
3. Learning ASM and system internals

./intro_to_RE

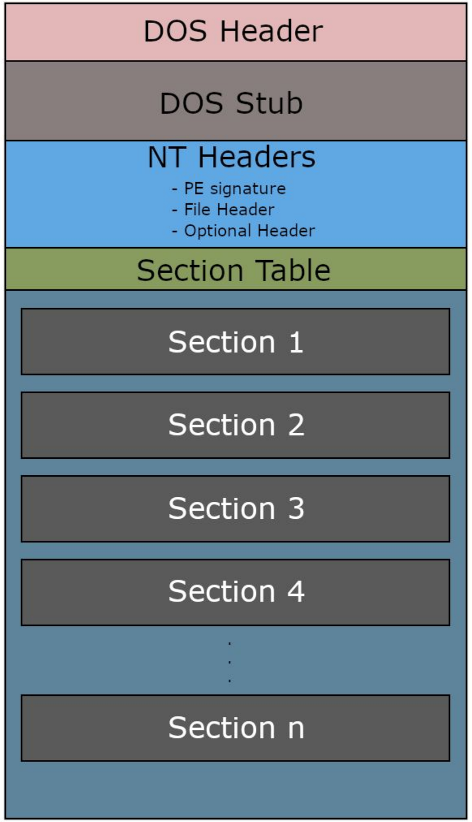

Reverse Engineering Process
From Binary Executable to Reconstructed Source Code
Compiled Program
(Binary/Executable)
May include
debug information
Debug Information
(If Available)
Reconstructed
Source Code
(Approximate)
Static Analysis
Binary Analysis
(Headers, Structure)
Extract
Instructions
Disassembler
(Assembly Code)
Convert to
Higher Level
Decompiler
(Pseudo-Source Code)
Tools: HxD, Binwalk, PE Explorer
Tools: IDA Pro, Ghidra, Radare2
Tools: Hex-Rays, Ghidra, RetDec
Analysis & Documentation
(Algorithms, Data Structures)
Manual Analysis
Reconstruct

./intro_to_RE


Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

- Preprocessing: “Getting the code ready” by handling **#include**, **#define** and remove comments
- Compilation: “Translate to assembly”, converting code to low-level instructions
- Assembly: “Convert to machine code”, ASM to machine-readable binary, output cant be run such as **.o** file (object file)
- Linking: “Build the final program”, combines **.o** with the libraries (**stdio.h, math.h**) to produce final executables

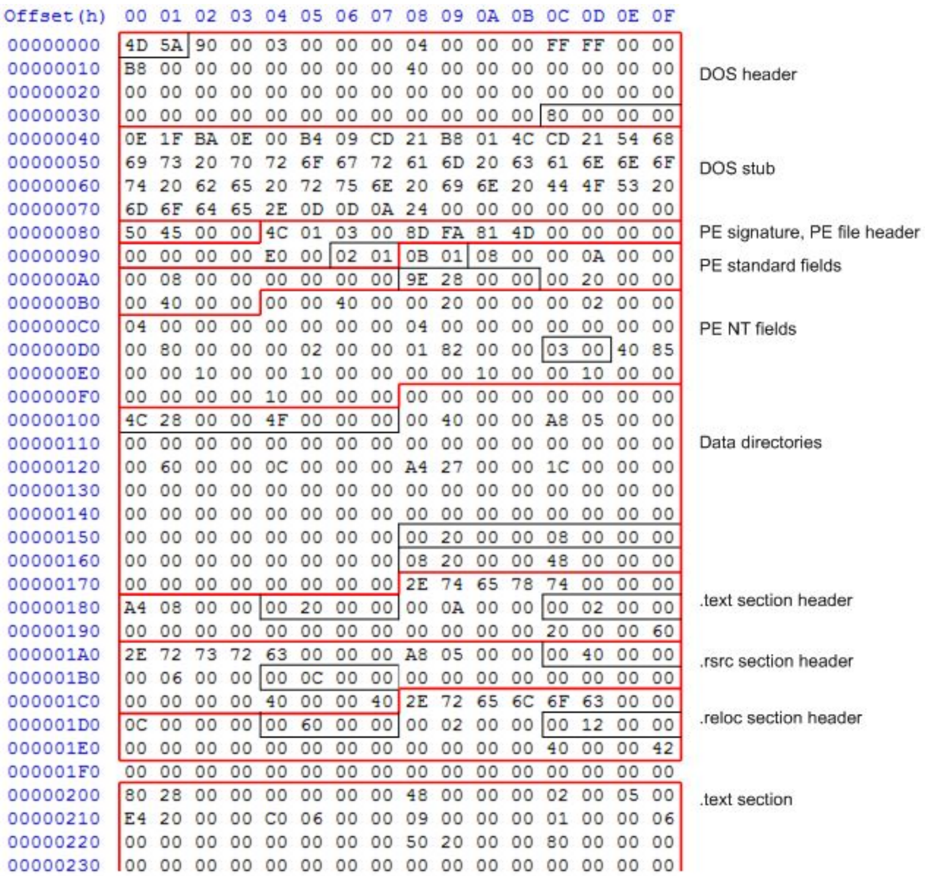
./pe_file_format



PE (Portable Executable):
- File format use by Windows for .exe, .dll and drivers
- Based on COFF (Common Object File Format)

Based on PE file format:
- Able to find **entry point**, where program execution begins
- Understand what **imported APIs** the program uses
- Locate **code**, **data**, and **resources**
- Identify if binary is **compressed, obfuscated**
- Understand how program loads into memory

./pe_file_format



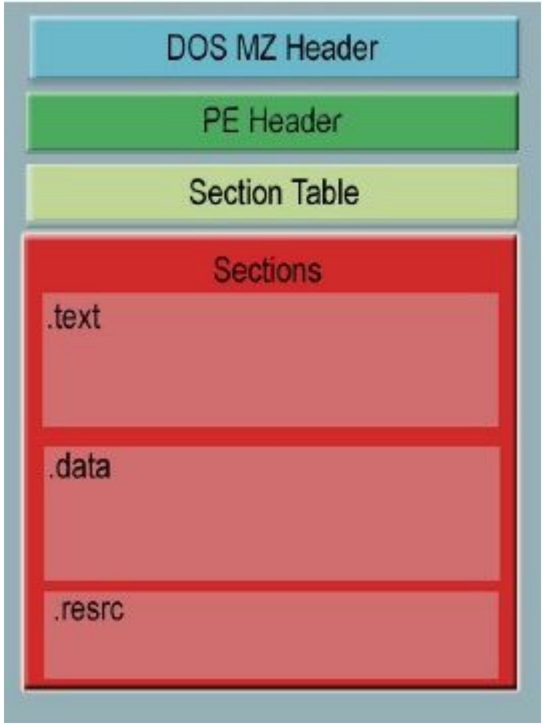
```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
00000010   B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030   00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00
00000040   0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
00000050   69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
00000060   74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20
00000070   6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00
00000080   50 45 00 00 4C 01 03 00 8D FA 81 4D 00 00 00 00
00000090   00 00 00 00 E0 00 02 01 0B 01 08 00 00 0A 00 00
000000A0   00 08 00 00 00 00 00 00 9E 28 00 00 00 20 00 00
000000B0   00 40 00 00 00 00 40 00 00 20 00 00 00 02 00 00
000000C0   04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
000000D0   00 80 00 00 00 02 00 00 01 82 00 00 03 00 40 85
000000E0   00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00
000000F0   00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00
00000100   4C 28 00 00 4F 00 00 00 00 40 00 00 A8 05 00 00
00000110   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000120   00 60 00 00 0C 00 00 00 A4 27 00 00 1C 00 00 00
00000130   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000140   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000150   00 00 00 00 00 00 00 00 00 20 00 00 08 00 00 00
00000160   00 00 00 00 00 00 00 00 08 20 00 00 48 00 00 00
00000170   00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00
00000180   A4 08 00 00 00 20 00 00 00 0A 00 00 00 02 00 00
00000190   00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60
000001A0   2E 72 73 72 63 00 00 00 A8 05 00 00 00 40 00 00
000001B0   00 06 00 00 00 0C 00 00 00 00 00 00 00 00 00 00
000001C0   00 00 00 00 40 00 00 40 2E 72 65 6C 6F 63 00 00
000001D0   0C 00 00 00 00 60 00 00 00 02 00 00 00 12 00 00
000001E0   00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 42
000001F0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000200   80 28 00 00 00 00 00 00 48 00 00 00 02 00 05 00
00000210   E4 20 00 00 C0 06 00 00 09 00 00 00 01 00 00 06
00000220   00 00 00 00 00 00 00 00 50 20 00 00 80 00 00 00
00000230   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

1. DOS Header
   - “MZ” magic hex signature (4D 5A)
   - Points the PE header locations
   - Includes DOS stub program

2. PE Header
   - PE Signature (50 45 00 00)
   - COFF Header, machine type, number of section, timestamp
   - Option header include **entry point**, **image base**, **section alignment**

3. Section Table:
   - Contains section names, size, permission, offsets
   - Array describing each section in the PE

./pe_file_format



PE Sections (common sections):

.text:

- contains executable machine code
- read and executable permission
- primary target to reverse

.data:

- global and static variable with initial values
- writeable during program execution

.rdata:

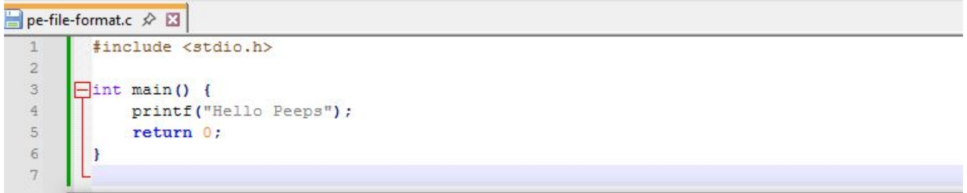
- string and constants which are read-only
- Import Address Table (IAT)

.bss:

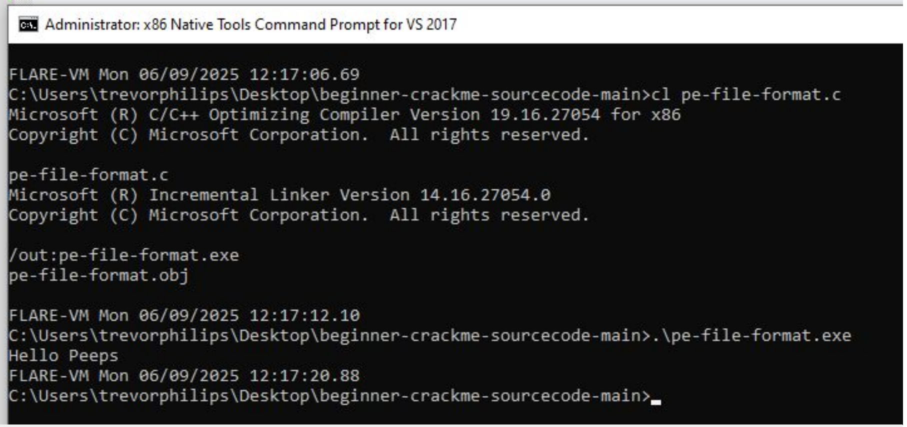
- uninitialized data, takes up space in memory but not in disk

./pe_file_format

```
#include <stdio.h>

int main() {
    printf(“Hello Peeps”);
    return 0;
}
```

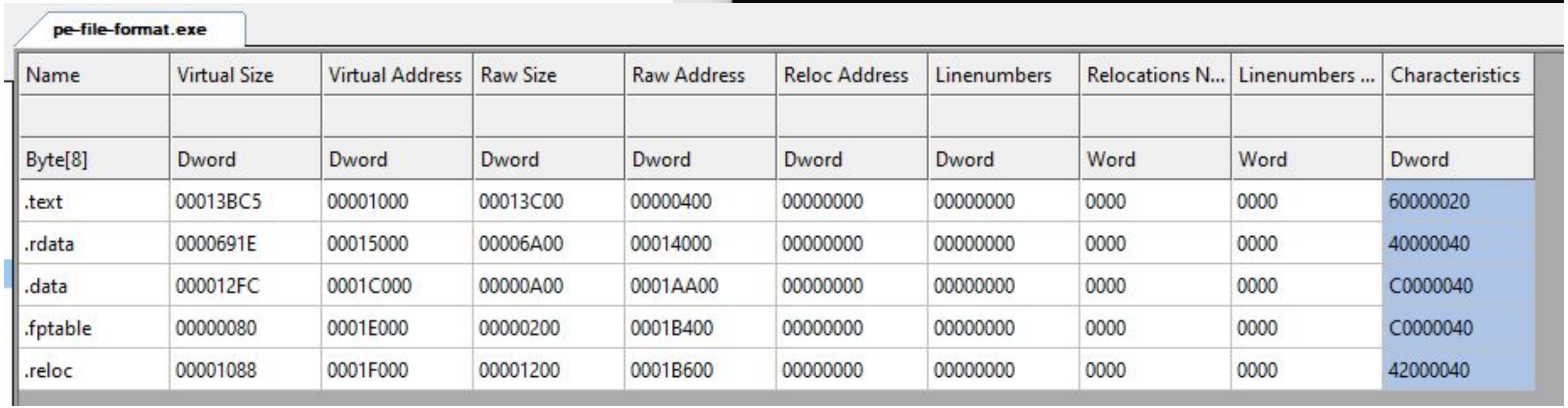


```c
#include <stdio.h>

int main() {
    printf("Hello Peeps");
    return 0;
}
```

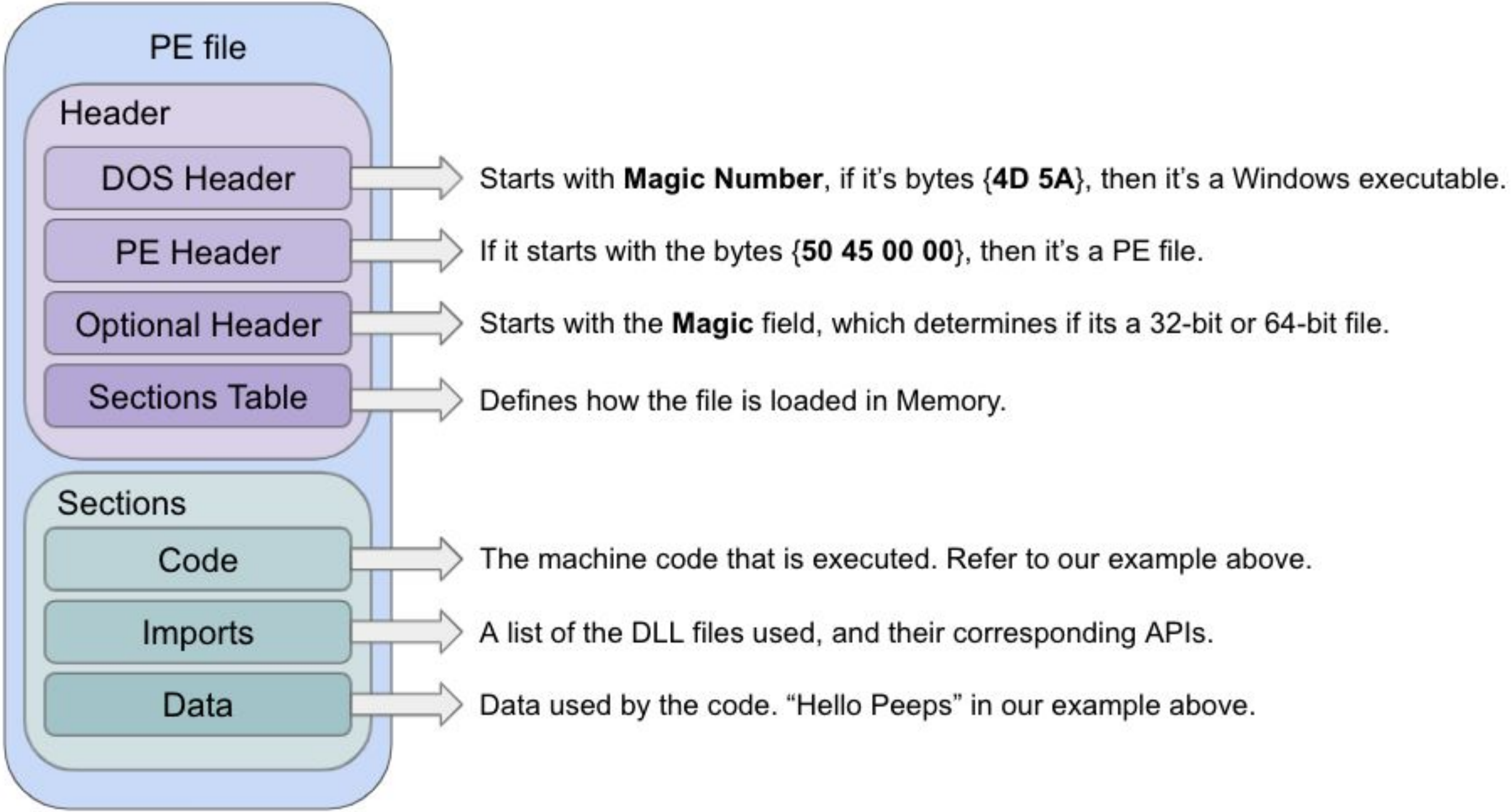


Administrator: x86 Native Tools Command Prompt for VS 2017

```
FLARE-VM Mon 06/09/2025 12:17:06.69
C:\Users\trevorphilips\Desktop\beginner-crackme-sourcecode-main>cl pe-file-format.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.16.27054 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

pe-file-format.c
Microsoft (R) Incremental Linker Version 14.16.27054.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:pe-file-format.exe
pe-file-format.obj

FLARE-VM Mon 06/09/2025 12:17:12.10
C:\Users\trevorphilips\Desktop\beginner-crackme-sourcecode-main>.\pe-file-format.exe
Hello Peeps
FLARE-VM Mon 06/09/2025 12:17:20.88
C:\Users\trevorphilips\Desktop\beginner-crackme-sourcecode-main>_
```



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00013BC5	00001000	00013C00	00000400	00000000	00000000	0000	0000	60000020
.rdata	0000691E	00015000	00006A00	00014000	00000000	00000000	0000	0000	40000040
.data	000012FC	0001C000	00000A00	0001AA00	00000000	00000000	0000	0000	C0000040
.fptable	00000080	0001E000	00000200	0001B400	00000000	00000000	0000	0000	C0000040
.reloc	00001088	0001F000	00001200	0001B600	00000000	00000000	0000	0000	42000040



PE file

Header

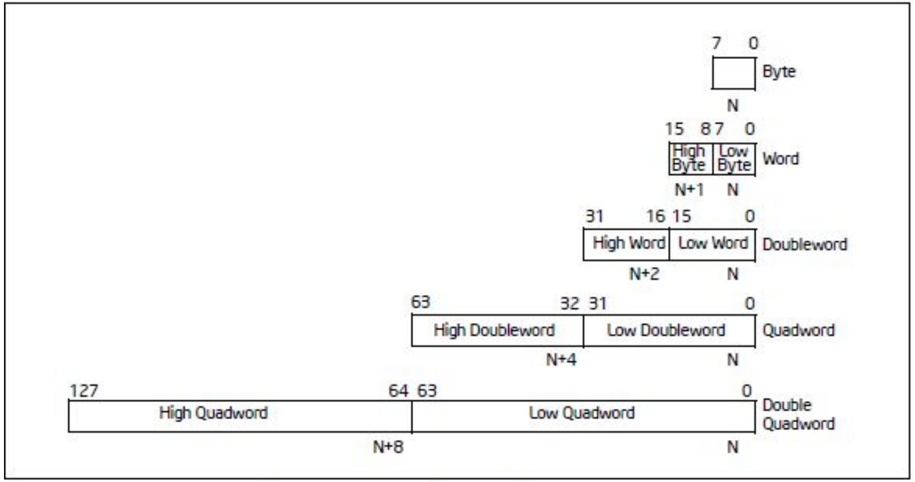
- DOS Header → Starts with **Magic Number**, if it's bytes {**4D 5A**}, then it's a Windows executable.
- PE Header → If it starts with the bytes {**50 45 00 00**}, then it's a PE file.
- Optional Header → Starts with the **Magic** field, which determines if its a 32-bit or 64-bit file.
- Sections Table → Defines how the file is loaded in Memory.

Sections

- Code → The machine code that is executed. Refer to our example above.
- Imports → A list of the DLL files used, and their corresponding APIs.
- Data → Data used by the code. “Hello Peeps” in our example above.

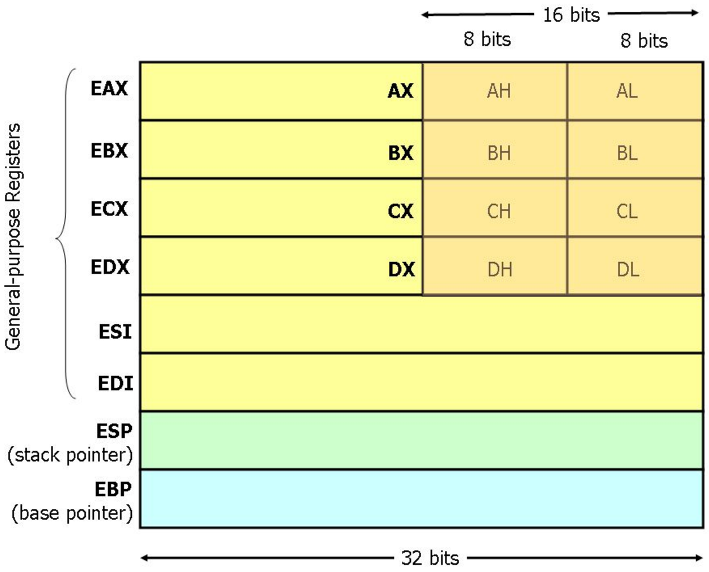
./x86_arch

- x86 architecture is a family of backward compatible instruction set based on Intel’s 8086 CPU
- The term “x86” was coined after several successors to the 8086 ended in “86” such as 80186, 80286 and etc.
- x86 refer as 32-bit instruction set, x86-64 refer as 64-bit instruction set



Byte = 8 bits (1 byte)
Word = 16 bits (2 bytes)
Doubleword = 32 bits (4 bytes)
Quadword = 64 bits (8 bytes)
Double Quadword = 128 bits (16 bytes)

./x86_arch



Data Registers

Register	Description	Usage
AL / AH / AX/ EAX	Accumulator Register	Arithmetic operations
BL / BH / BX / EBX	Base register	General data storage, index
CL / CH / CX / ECX	Counter register	Loop constructs
DL / DH / DX / EDX	Data register	Arithmetic

Address Registers

Register	Description	Usage
IP / EIP	Instruction Pointer	Program execution counter
SP / ESP	Stack Pointer	ESP will hold an offset to top of stacks memory location
BP / EBP	Base Pointer	Stack frame
SI / ESI	Source Index	String operation
DI / EDI	Destination Index	String operation

./x86_arch

Basic Assembly Instructions:

- MOV - Move data between registers; data between memory and registers; immediate value into registers
- PUSH - Push onto the stack
- POP - Pop off the stack
- ADD - Integer add
- SUB - Subtract
- MUL - Multiply
- DIV - Divide
- INC - Increment
- DEC - Decrement
- CMP - Compare
- AND
- OR
- XOR
- NOT

There is a lot more, here is a simple cheatsheet to refer:
https://github.com/7etsuo/x86

./x86_arch

mnemonic argument1, argument2, argument3	
MOV EAX, 1	Move 1 to EAX
ADD EDX, 5	Add 5 to EDX
SUB EBX, 2	Subtract 2 from EBX
AND ECX, 0	Bit-wise AND 0 to ECX
XOR EDX, 4	Bit-wise eXclusive OR 4 to EDX
SHL ECX, 6	Shift ECX left by six
ROR EBX, 3	Bit-wise rotate EBX right by 3
INC ECX	Increment ECX

./x86_arch

Memory Addressing:

1. Immediate Addressing:

```
mov eax, 0x1234
```

Moves value **0x1234** directly into **EAX** register

./x86_arch

Memory Addressing:

2. Register Addressing:

```
mov eax, ebx
```

Copy value from **EBX** into **EAX**

./x86_arch

Memory Addressing:

3. Direct (Absolute) Addressing

    ```
    mov eax, [0x401000]
    ```

    Moves value at memory address **0x401000** into **EAX**

./x86_arch

Memory Addressing:

4. Indirect Addressing

```
mov eax, [ebx]
```

Moves value from the **memory address** pointed by **EBX** into **EAX**

./x86_arch

Memory Addressing:

5. Base + Offset Addressing

```
mov eax, [ebx + 4]
```

Move the value at **EBX + 4** into EAX, used to access structure field or array elements

./x86_arch

Memory Addressing:

6. Base + Index Addressing

```
mov eax, [ebx + esi]
```

Moves the value from the address **EBX + ESI** into **EAX**

./x86_arch

Memory Addressing:

7. Base + Index Addressing

```
mov eax, [ebx + esi*4 + 8]
```

Moves the value from the address **EBX + (ESI x 4) + 8** into **EAX**

- EBX: base
- ESI: index
- 4: scale (can be 1, 2, 4 or 8)
- 8: offset

./x86_arch

Stack operation:

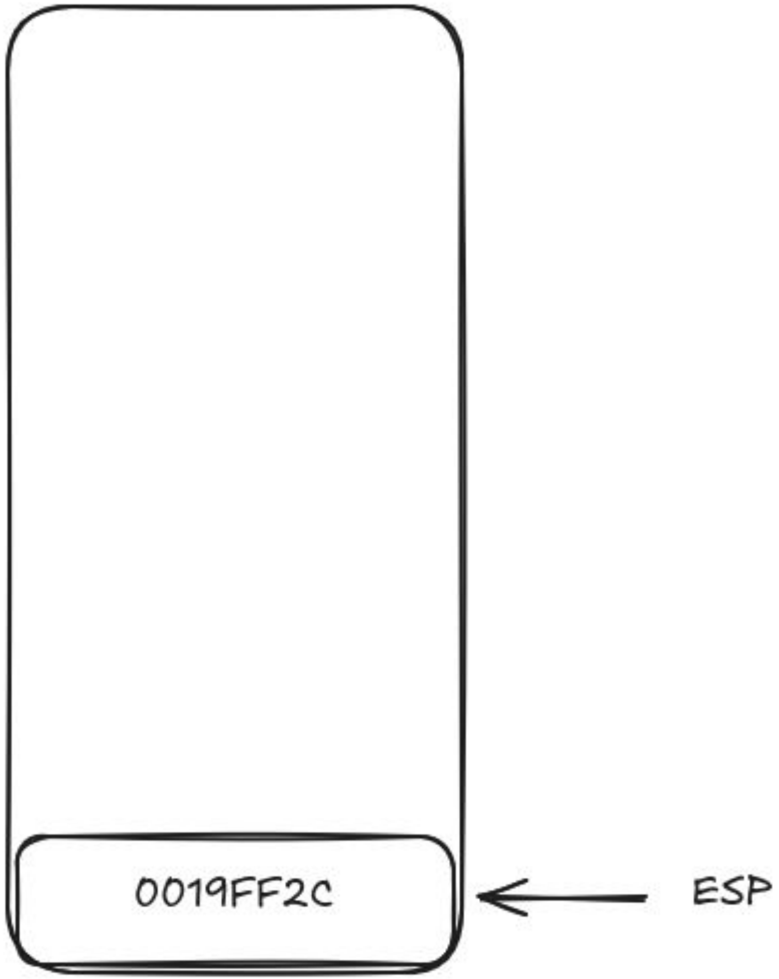
```
int __cdecl main(int argc, const char **argv, const char **envp)
_main           proc near

push    ebp
mov     ebp, esp
push    offset aHelloPeeps ; "Hello Peeps"
call    printf
add     esp, 4
xor     eax, eax
pop     ebp
retn

_main           endp
```

./x86_arch

Stack operation:

Before _main starts, **CRTStartup** or previous entry point

CRT = C RunTime

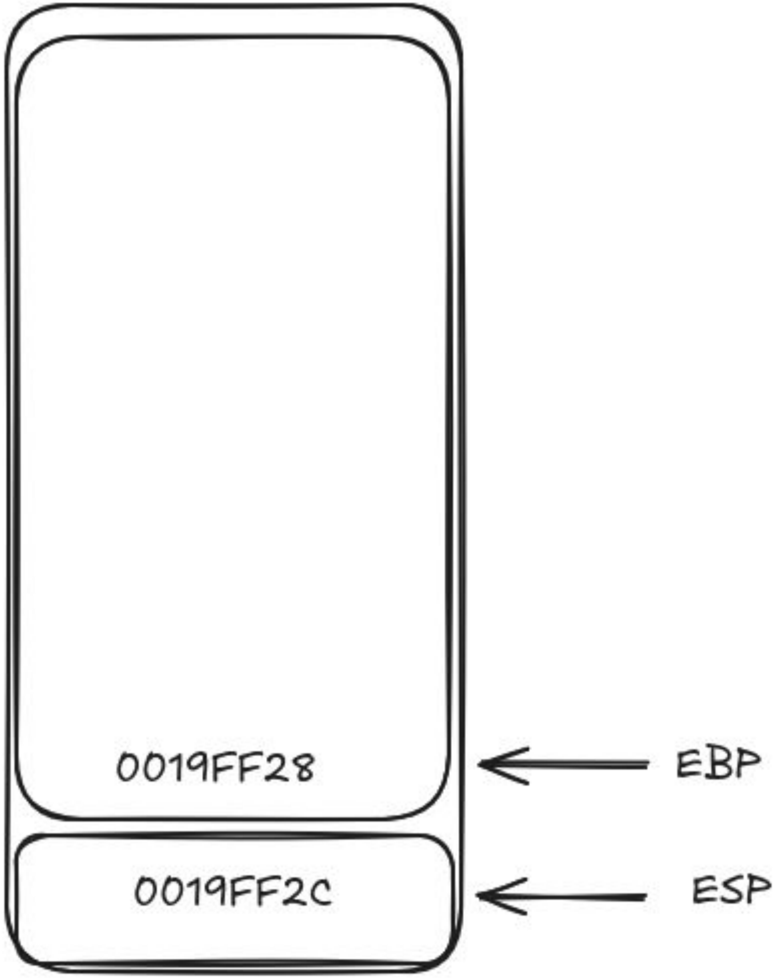


./x86_arch

Stack operation:

`push ebp`

Save old EBP on the stack



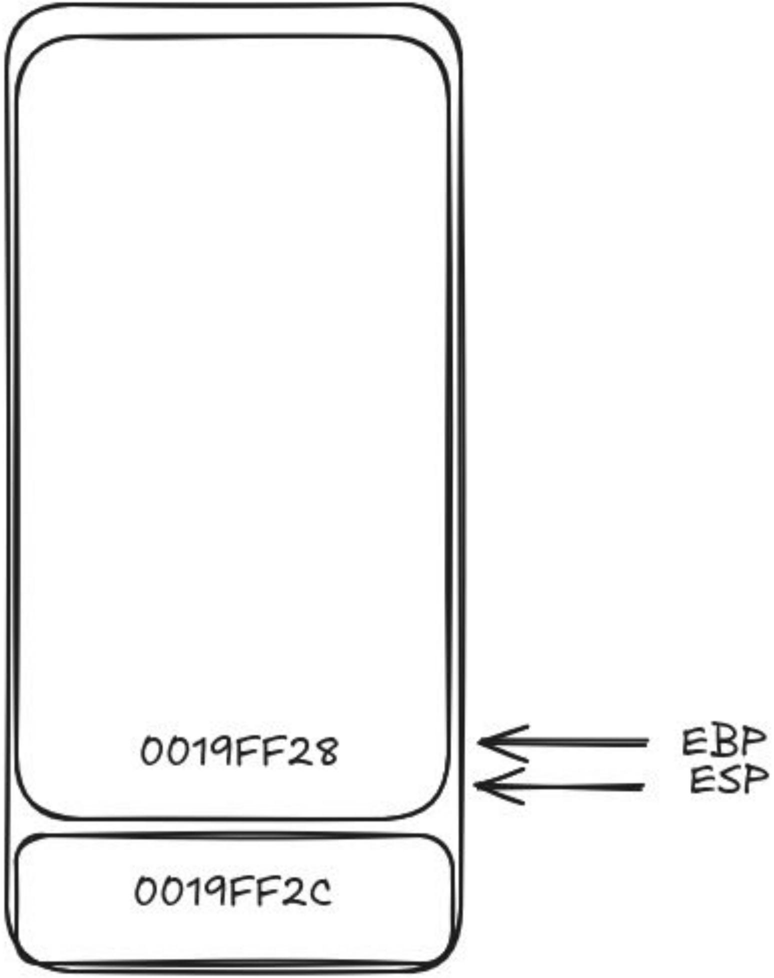
./x86_arch

`mov ebp, esp`

Copy ESP to EBP

With first two instruction, we called it **prologue**

If there is any local variable, you will notice a **sub esp, X**

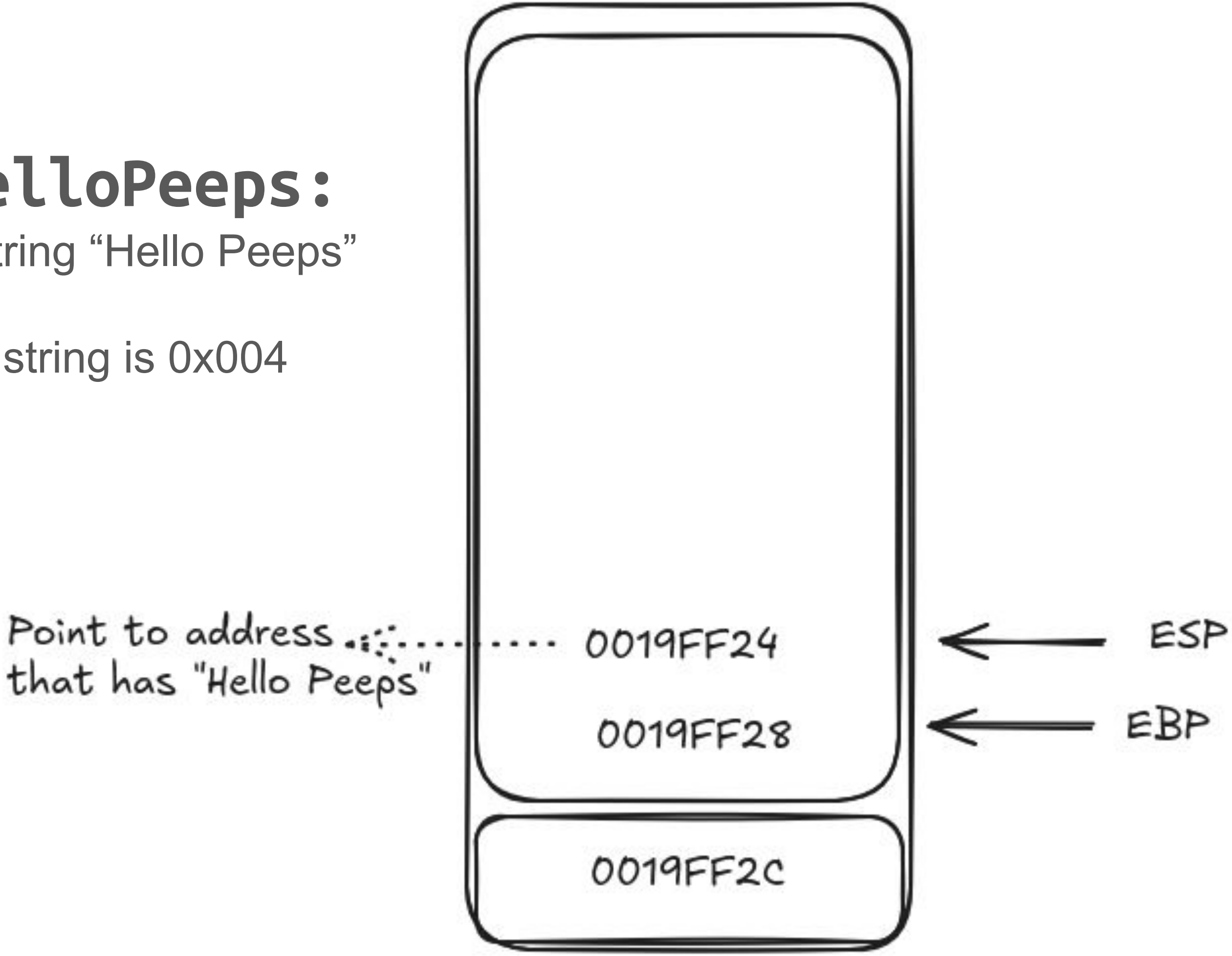


./x86_arch

`push offset aHelloPeeps:`

Pushes the pointer to the string “Hello Peeps”

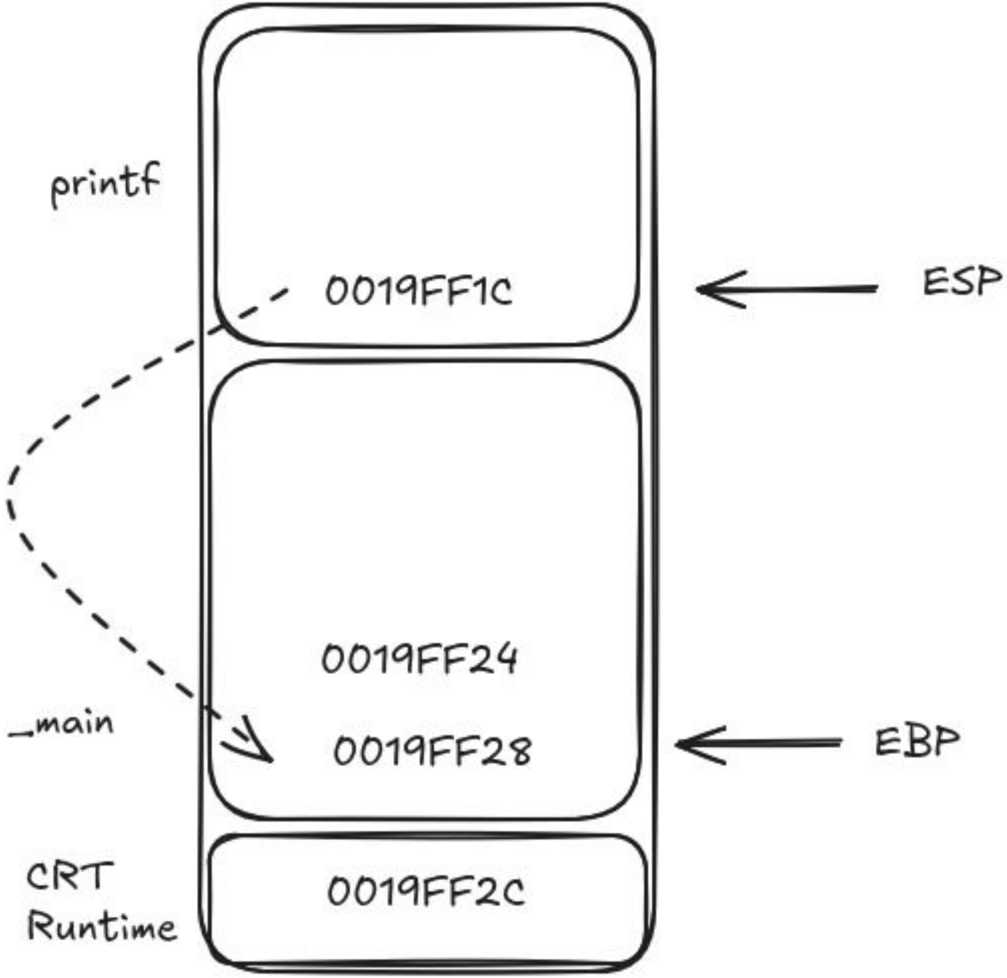
Assume the address of the string is 0x004



./x86_arch

`call printf:`

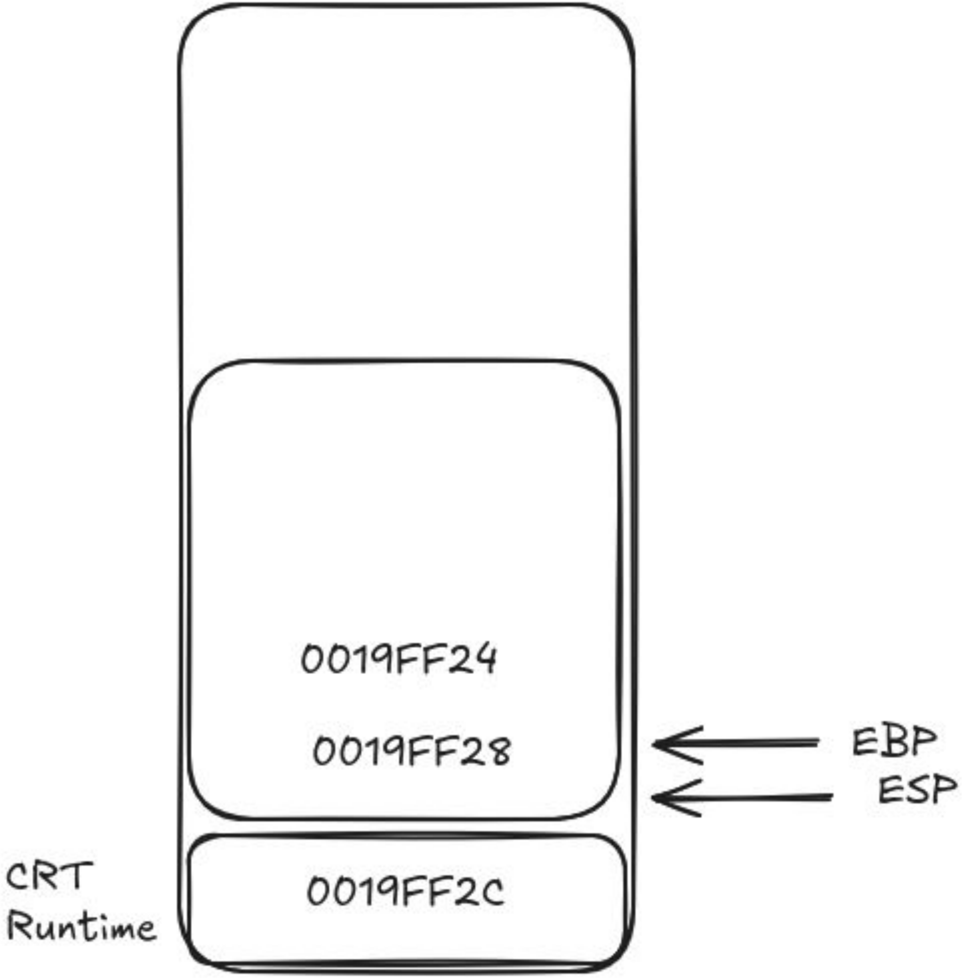
Pushes return address from _main onto the stack



./x86_arch

`add esp, 4:`

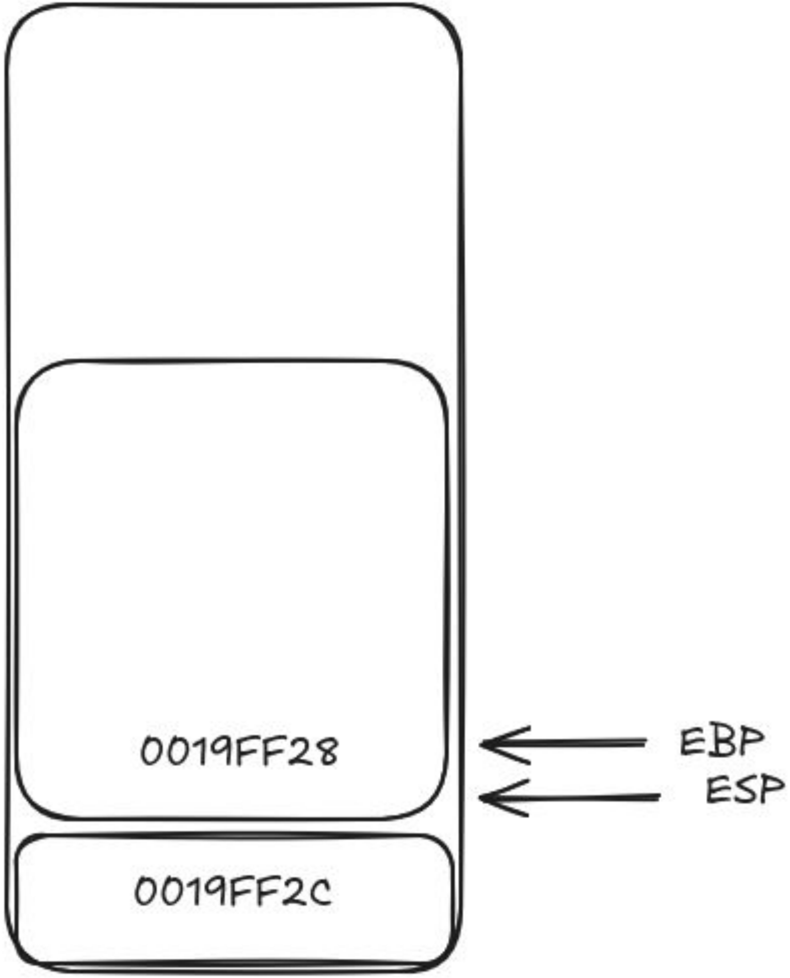
After return from printf, clean up the stack by adjusting ESP



./x86_arch

`xor eax, eax:`

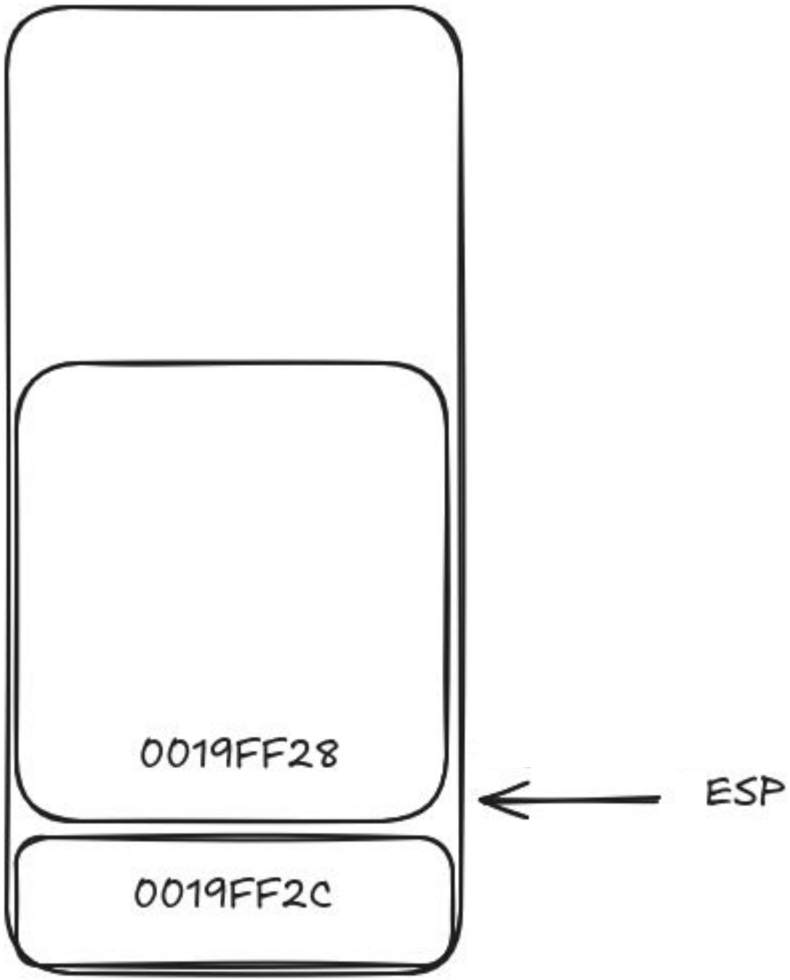
Clears return value at register, no changes in stack



./x86_arch

pop ebp:

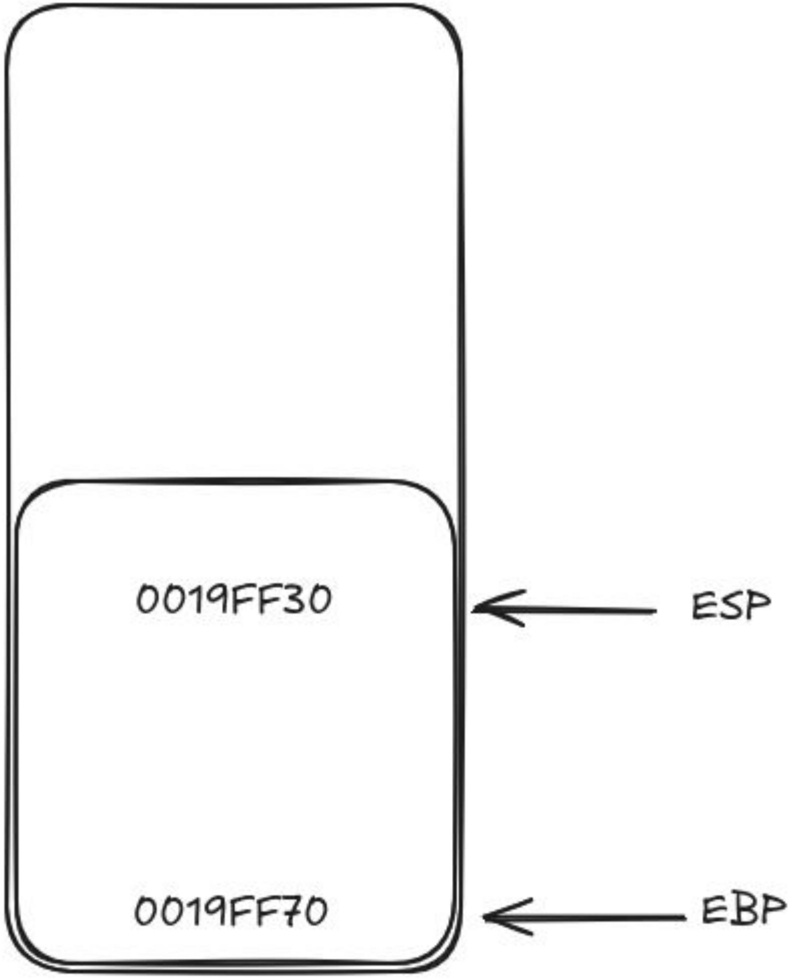
Restore caller’s base pointer
Basically undoing push ebp



./x86_arch

`ret:`

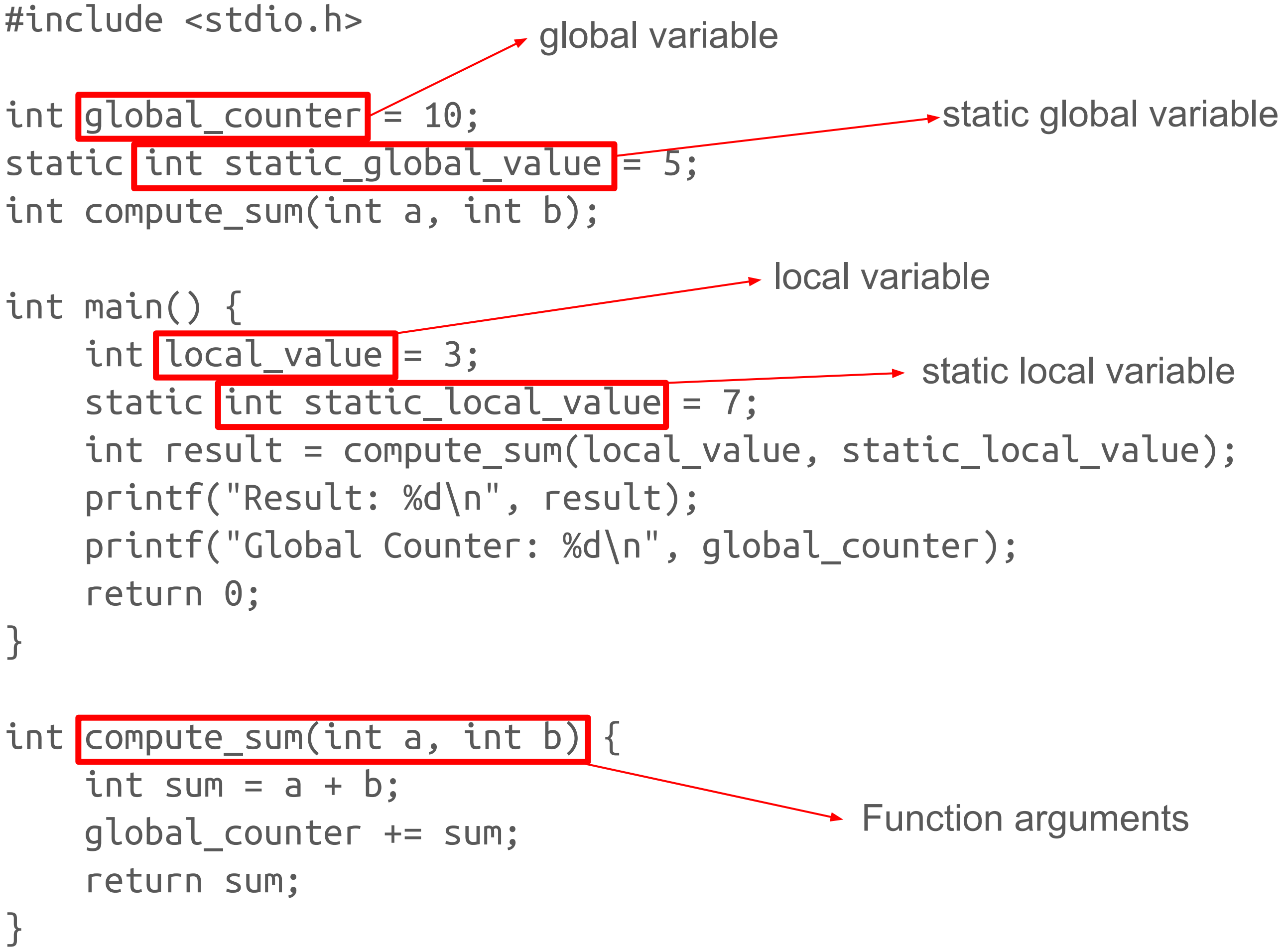
Pops return address and jumps to it



./basic_c_asm

```c
#include <stdio.h>

int global_counter = 10;
static int static_global_value = 5;
int compute_sum(int a, int b);

int main() {
    int local_value = 3;
    static int static_local_value = 7;
    int result = compute_sum(local_value, static_local_value);
    printf("Result: %d\n", result);
    printf("Global Counter: %d\n", global_counter);
    return 0;
}

int compute_sum(int a, int b) {
    int sum = a + b;
    global_counter += sum;
    return sum;
}
```

./basic_c_asm



```c
#include <stdio.h>

int global_counter = 10;
static int static_global_value = 5;
int compute_sum(int a, int b);

int main() {
    int local_value = 3;
    static int static_local_value = 7;
    int result = compute_sum(local_value, static_local_value);
    printf("Result: %d\n", result);
    printf("Global Counter: %d\n", global_counter);
    return 0;
}

int compute_sum(int a, int b) {
    int sum = a + b;
    global_counter += sum;
    return sum;
}
```

global variable

static global variable

local variable

static local variable

Function arguments

./basic_c_asm

1. Local variable:
   - Dynamically allocated on stack memory
   - Temporarily available

2. Static variable:
   - Usually located inside memory section
   - Initialization occurs once and then the variable retains its value
   - Only accessible from within the function

3. Global variable:
   - Usually located inside memory section
   - Static location, always accessible from everywhere

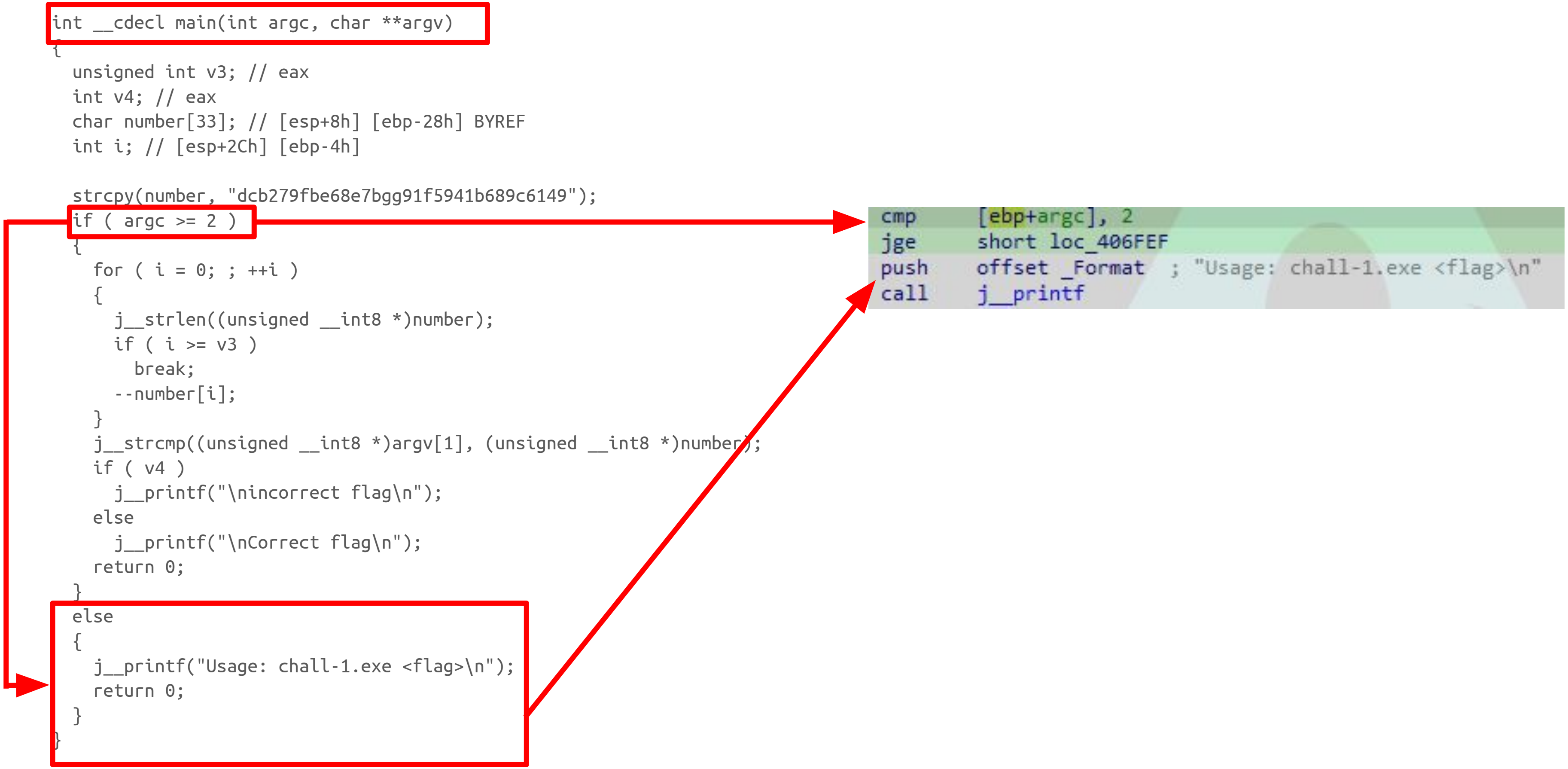
./basic_c_asm

```
int __cdecl main(int argc, char **argv)
{
  unsigned int v3; // eax
  int v4; // eax
  char number[33]; // [esp+8h] [ebp-28h] BYREF
  int i; // [esp+2Ch] [ebp-4h]

  strcpy(number, "dcb279fbe68e7bgg91f5941b689c6149");
  if ( argc >= 2 )
  {
    for ( i = 0; ; ++i )
    {
      j__strlen((unsigned __int8 *)number);
      if ( i >= v3 )
        break;
      --number[i];
    }
    j__strcmp((unsigned __int8 *)argv[1], (unsigned __int8 *)number);
    if ( v4 )
      j__printf("\nincorrect flag\n");
    else
      j__printf("\nCorrect flag\n");
    return 0;
  }
  else
  {
    j__printf("Usage: chall-1.exe <flag>\n");
    return 0;
  }
}
```

Steps:

1. Observe main function and understand how arguments are used
2. Readable strings are useful
3. Go function by function
4. Trace the logic flow

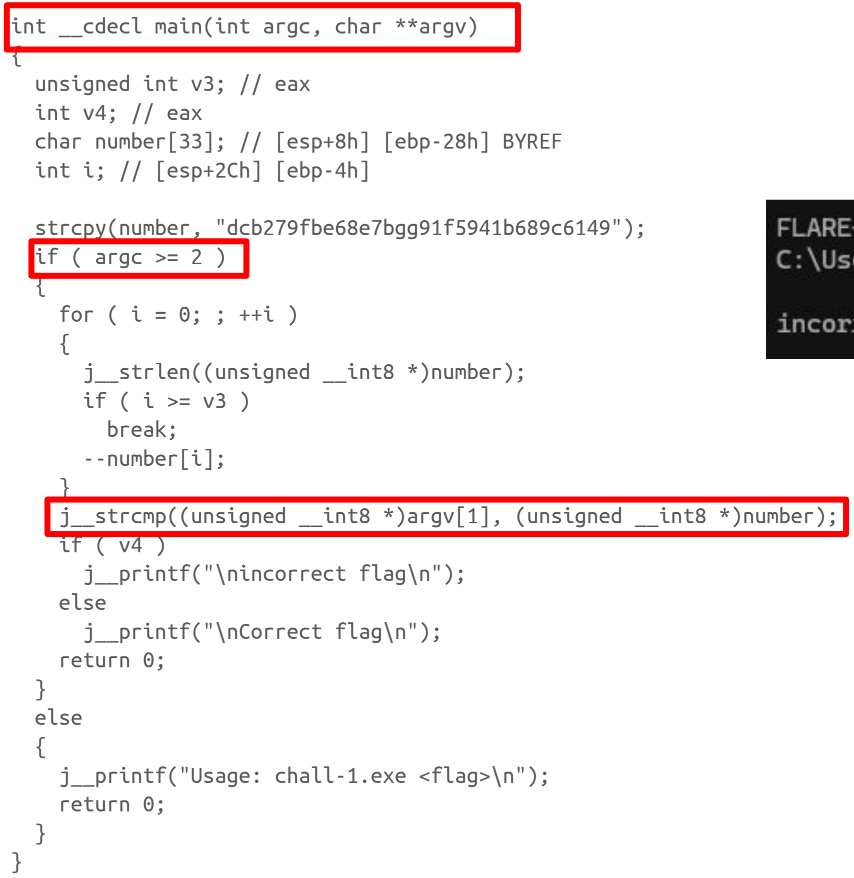
./basic_c_asm



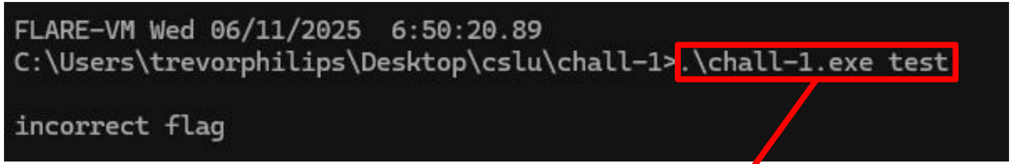
```
int __cdecl main(int argc, char **argv)
{
  unsigned int v3; // eax
  int v4; // eax
  char number[33]; // [esp+8h] [ebp-28h] BYREF
  int i; // [esp+2Ch] [ebp-4h]

  strcpy(number, "dcb279fbe68e7bgg91f5941b689c6149");
  if ( argc >= 2 )
  {
    for ( i = 0; ; ++i )
    {
      j__strlen((unsigned __int8 *)number);
      if ( i >= v3 )
        break;
      --number[i];
    }
    j__strcmp((unsigned __int8 *)argv[1], (unsigned __int8 *)number);
    if ( v4 )
      j__printf("\nincorrect flag\n");
    else
      j__printf("\nCorrect flag\n");
    return 0;
  }
  else
  {
    j__printf("Usage: chall-1.exe <flag>\n");
    return 0;
  }
}
```

```
cmp     [ebp+argc], 2
jge     short loc_406FEF
push    offset _Format  ; "Usage: chall-1.exe <flag>\n"
call    j__printf
```

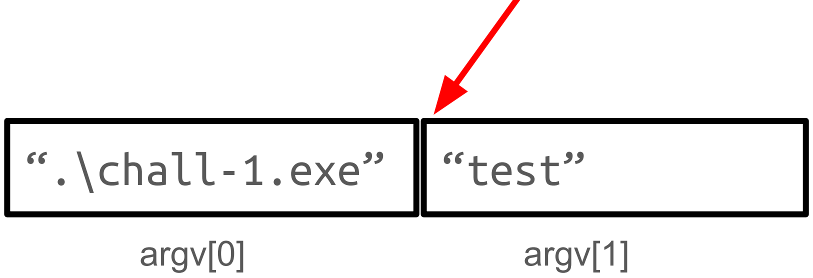
./basic_c_asm



```
int __cdecl main(int argc, char **argv)
{
  unsigned int v3; // eax
  int v4; // eax
  char number[33]; // [esp+8h] [ebp-28h] BYREF
  int i; // [esp+2Ch] [ebp-4h]

  strcpy(number, "dcb279fbe68e7bgg91f5941b689c6149");
  if ( argc >= 2 )
  {
    for ( i = 0; ; ++i )
    {
      j__strlen((unsigned __int8 *)number);
      if ( i >= v3 )
        break;
      --number[i];
    }
    j__strcmp((unsigned __int8 *)argv[1], (unsigned __int8 *)number);
    if ( v4 )
      j__printf("\nincorrect flag\n");
    else
      j__printf("\nCorrect flag\n");
    return 0;
  }
  else
  {
    j__printf("Usage: chall-1.exe <flag>\n");
    return 0;
  }
}
```



```
FLARE-VM Wed 06/11/2025  6:50:20.89
C:\Users\trevorphilips\Desktop\cslu\chall-1>.\chall-1.exe test

incorrect flag
```

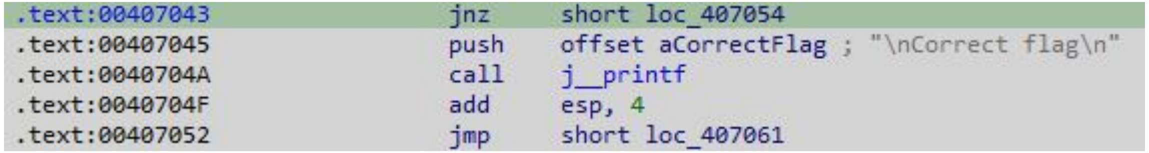


./basic_c_asm

```
int __cdecl main(int argc, char **argv)
{
  unsigned int v3; // eax
  int v4; // eax
  char number[33]; // [esp+8h] [ebp-28h] BYREF
  int i; // [esp+2Ch] [ebp-4h]

  strcpy(number, "dcb279fbe68e7bgg91f5941b689c6149");
  if ( argc >= 2 )
  {
    for ( i = 0; ; ++i )
    {
      j__strlen((unsigned __int8 *)number);
      if ( i >= v3 )
        break;
      --number[i];
    }
    j__strcmp((unsigned __int8 *)argv[1], (unsigned __int8 *)number);
    if ( v4 )
      j__printf("\nincorrect flag\n");
    else
      j__printf("\nCorrect flag\n");
    return 0;
  }
  else
  {
    j__printf("Usage: chall-1.exe <flag>\n");
    return 0;
  }
}
```

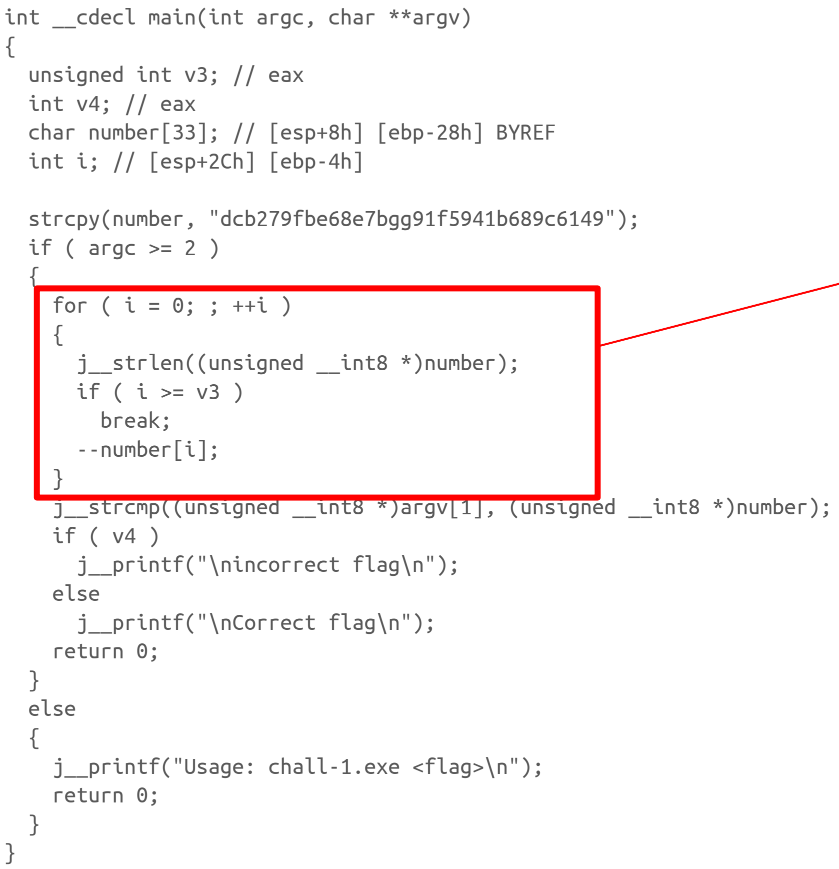
number is a variable that store “dcb279fbe68e7bgg91f5941b689c6149”

v4 stores return value of strcmp: -1 (less than 0), 0 (equal) or 1 (greather than 0)

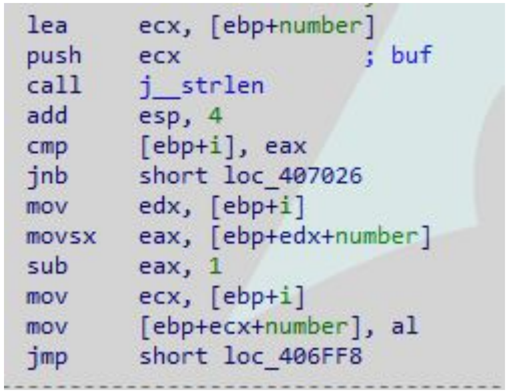


```
.text:00407043                 jnz     short loc_407054
.text:00407045                 push    offset aCorrectFlag ; "\nCorrect flag\n"
.text:0040704A                 call    j__printf
.text:0040704F                 add     esp, 4
.text:00407052                 jmp     short loc_407061
```

./basic_c_asm



```
int __cdecl main(int argc, char **argv)
{
  unsigned int v3; // eax
  int v4; // eax
  char number[33]; // [esp+8h] [ebp-28h] BYREF
  int i; // [esp+2Ch] [ebp-4h]

  strcpy(number, "dcb279fbe68e7bgg91f5941b689c6149");
  if ( argc >= 2 )
  {
    for ( i = 0; ; ++i )
    {
      j__strlen((unsigned __int8 *)number);
      if ( i >= v3 )
        break;
      --number[i];
    }
    j__strcmp((unsigned __int8 *)argv[1], (unsigned __int8 *)number);
    if ( v4 )
      j__printf("\nincorrect flag\n");
    else
      j__printf("\nCorrect flag\n");
    return 0;
  }
  else
  {
    j__printf("Usage: chall-1.exe <flag>\n");
    return 0;
  }
}
```



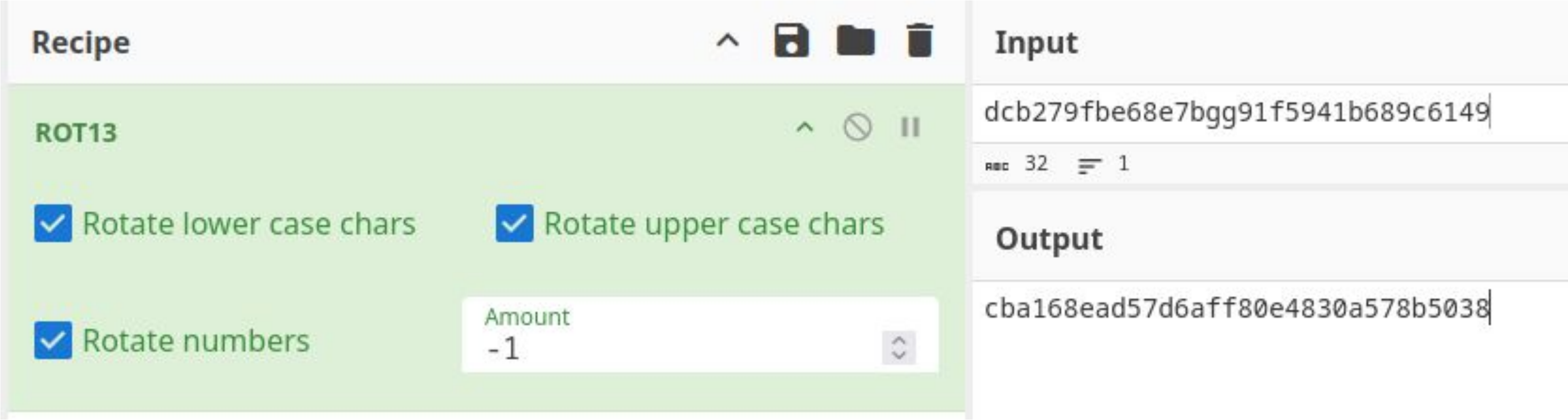
```
lea     ecx, [ebp+number]
push    ecx             ; buf
call    j__strlen
add     esp, 4
cmp     [ebp+i], eax
jnb     short loc_407026
mov     edx, [ebp+i]
movsx   eax, [ebp+edx+number]
sub     eax, 1
mov     ecx, [ebp+i]
mov     [ebp+ecx+number], al
jmp     short loc_406FF8
```

1. Address of “number” loads into ECX and used by strlen to calculate the length of string
2. The value will be stored in EAX (v3) after strlen function is executed
3. cmp with jnb is the compare loop counter
4. - - number[i] is ASCII decrement

So in short, number will a new value that is decrement by ASCII value of 1

A -> 41 - 1 = 40 -> @
B -> 42 - 1 = 41 -> A
C -> 43 - 1 = 42 -> B

./basic_c_asm



Recipe

ROT13

☑ Rotate lower case chars ☑ Rotate upper case chars

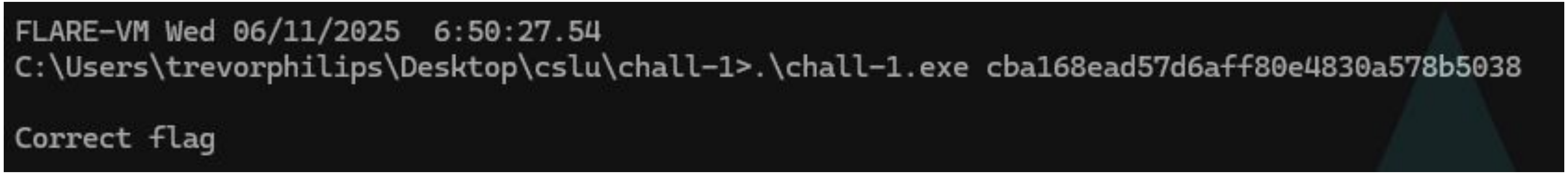
☑ Rotate numbers Amount -1

Input

```
dcb279fbe68e7bgg91f5941b689c6149
```

Output

```
cba168ead57d6aff80e4830a578b5038
```

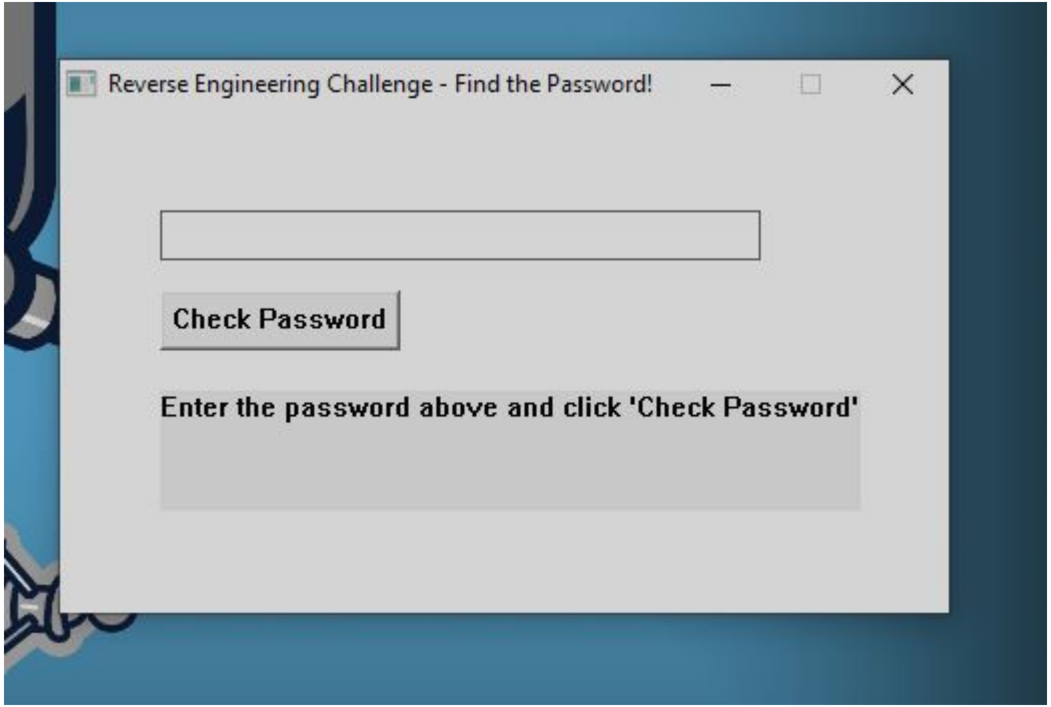


```
FLARE-VM Wed 06/11/2025  6:50:27.54
C:\Users\trevorphilips\Desktop\cslu\chall-1>.\chall-1.exe cba168ead57d6aff80e4830a578b5038

Correct flag
```

./basic_c_asm

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "dcb279fbe68e7bgg91f5941b689c6149\n";
    int len = strlen(str);
    for(int i=0;i<=len;i++) {
        --str[i];
    }
    printf("%s\n",str);
    return 0;
}
```

./cracking_crackmes



Reverse Engineering Challenge - Find the Password!

Enter the password above and click 'Check Password'

We have look at console application, how about GUI ?

./cracking_crackmes

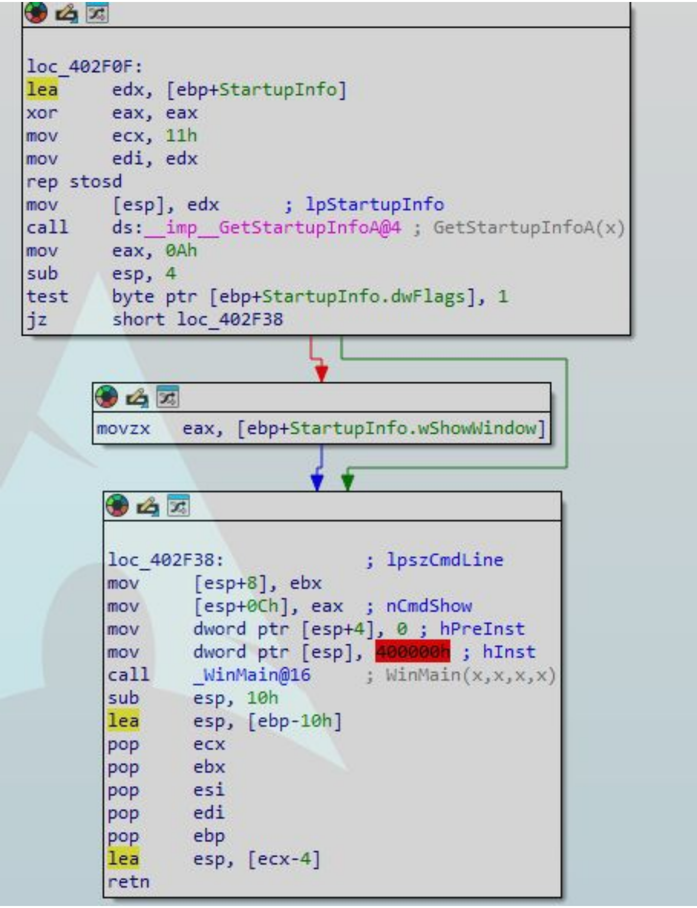
Every Windows program includes an entry-point function named either WinMain or wWinMain. The following code shows the signature for wWinMain:

```
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance, PWSTR pCmdLine, int nCmdShow);
```

How does the compiler know to invoke **WinMain** instead of the standard **main** function? What actually happens is that the Microsoft C runtime library (CRT) provides an implementation of **main** that calls **WinMain**.

The CRT does some more work inside **main**. For example, it calls any static initializers before **WinMain**.

https://learn.microsoft.com/en-us/windows/win32/learnwin32/winmain--the-application-entry-point



```
loc_402F0F:
lea     edx, [ebp+StartupInfo]
xor     eax, eax
mov     ecx, 11h
mov     edi, edx
rep stosd
mov     [esp], edx      ; lpStartupInfo
call    ds:__imp__GetStartupInfoA@4 ; GetStartupInfoA(x)
mov     eax, 0Ah
sub     esp, 4
test    byte ptr [ebp+StartupInfo.dwFlags], 1
jz      short loc_402F38
```

```
movzx   eax, [ebp+StartupInfo.wShowWindow]
```

```
loc_402F38:                     ; lpszCmdLine
mov     [esp+8], ebx
mov     [esp+0Ch], eax  ; nCmdShow
mov     dword ptr [esp+4], 0 ; hPreInst
mov     dword ptr [esp], 400000h ; hInst
call    _WinMain@16     ; WinMain(x,x,x,x)
sub     esp, 10h
lea     esp, [ebp-10h]
pop     ecx
pop     ebx
pop     esi
pop     edi
pop     ebp
lea     esp, [ecx-4]
retn
```

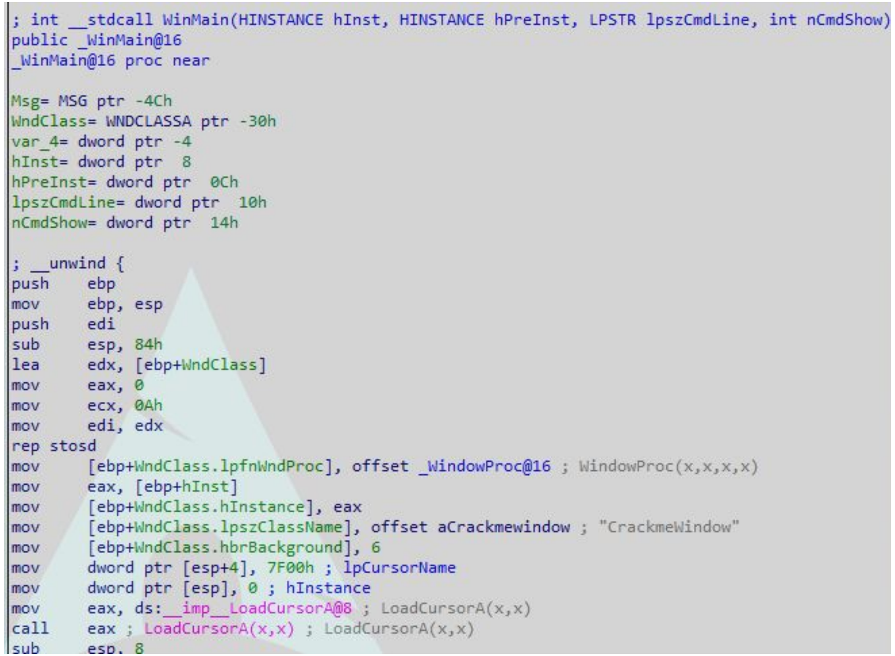
./cracking_crackmes

_main -> _WinMain -> _WindowProc

WindowProc is a callback function that handles input by the GUI interface.

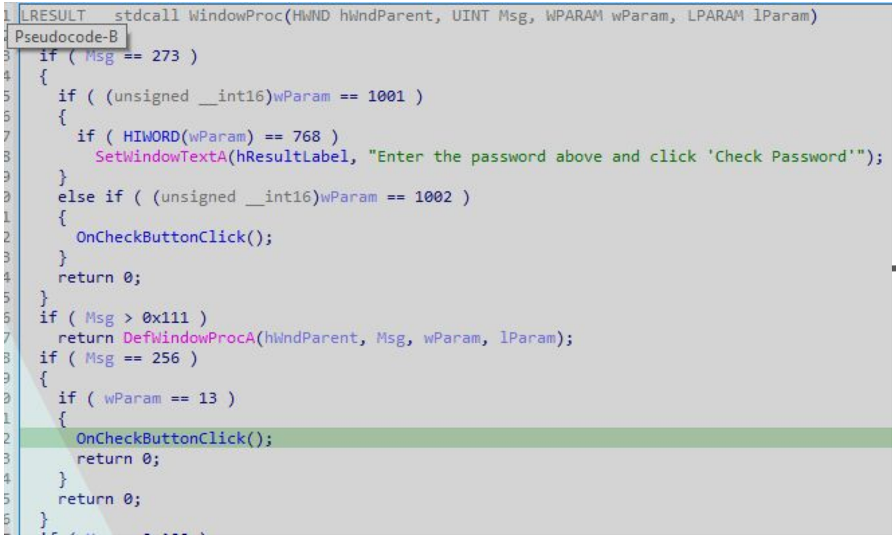
It is only called when Windows system whenever events occur like mouse clicks and keyboard input

https://learn.microsoft.com/en-us/windows/win32/api/winuser/nc-winuser-wndproc

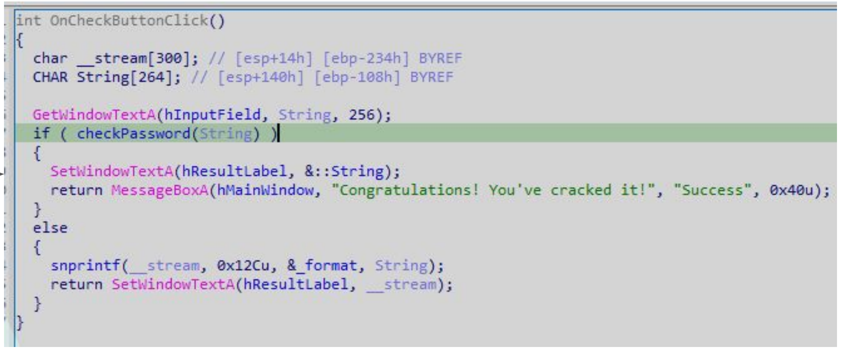


```
; int __stdcall WinMain(HINSTANCE hInst, HINSTANCE hPreInst, LPSTR lpszCmdLine, int nCmdShow)
public _WinMain@16
_WinMain@16 proc near

Msg= MSG ptr -4Ch
WndClass= WNDCLASSA ptr -30h
var_4= dword ptr -4
hInst= dword ptr  8
hPreInst= dword ptr  0Ch
lpszCmdLine= dword ptr  10h
nCmdShow= dword ptr  14h

; __unwind {
push    ebp
mov     ebp, esp
push    edi
sub     esp, 84h
lea     edx, [ebp+WndClass]
mov     eax, 0
mov     ecx, 0Ah
mov     edi, edx
rep stosd
mov     [ebp+WndClass.lpfnWndProc], offset _WindowProc@16 ; WindowProc(x,x,x,x)
mov     eax, [ebp+hInst]
mov     [ebp+WndClass.hInstance], eax
mov     [ebp+WndClass.lpszClassName], offset aCrackmewindow ; "CrackmeWindow"
mov     [ebp+WndClass.hbrBackground], 6
mov     dword ptr [esp+4], 7F00h ; lpCursorName
mov     dword ptr [esp], 0 ; hInstance
mov     eax, ds:__imp__LoadCursorA@8 ; LoadCursorA(x,x)
call    eax ; LoadCursorA(x,x) ; LoadCursorA(x,x)
sub     esp, 8
```

./cracking_crackmes



```
LRESULT __stdcall WindowProc(HWND hWndParent, UINT Msg, WPARAM wParam, LPARAM lParam)

  if ( Msg == 273 )
  {
    if ( (unsigned __int16)wParam == 1001 )
    {
      if ( HIWORD(wParam) == 768 )
        SetWindowTextA(hResultLabel, "Enter the password above and click 'Check Password'");
    }
    else if ( (unsigned __int16)wParam == 1002 )
    {
      OnCheckButtonClick();
    }
    return 0;
  }
  if ( Msg > 0x111 )
    return DefWindowProcA(hWndParent, Msg, wParam, lParam);
  if ( Msg == 256 )
  {
    if ( wParam == 13 )
    {
      OnCheckButtonClick();
      return 0;
    }
    return 0;
  }
```



```
int OnCheckButtonClick()
{
  char __stream[300]; // [esp+14h] [ebp-234h] BYREF
  CHAR String[264]; // [esp+140h] [ebp-108h] BYREF

  GetWindowTextA(hInputField, String, 256);
  if ( checkPassword(String) )
  {
    SetWindowTextA(hResultLabel, &::String);
    return MessageBoxA(hMainWindow, "Congratulations! You've cracked it!", "Success", 0x40u);
  }
  else
  {
    snprintf(__stream, 0x12Cu, &_format, String);
    return SetWindowTextA(hResultLabel, __stream);
  }
}
```

OnCheckButtonClick function will contain function that perform the password checking.

and we got checkPassword() now reverse the checkPassword function

./cracking_crackmes

```
BOOL __cdecl checkPassword(int a1)
{
  int i; // [esp+Ch] [ebp-4h]

  for ( i = 0; aPassword1[i] && *(_BYTE *)(i + a1); ++i )
  {
    if ( aPassword1[i] - 1 != *(char *)(i + a1) )
      return 0;
  }
  return !aPassword1[i] && !*(_BYTE *)(i + a1);
}
```

Can you figure out how the input is checked?

./cracking_crackmes

PE vs. ELF

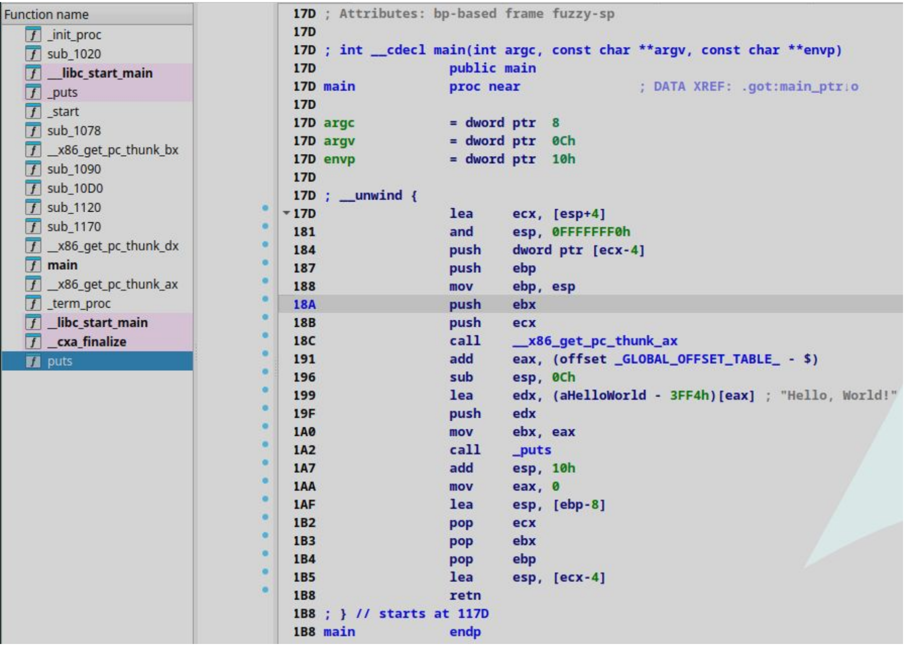
1. PE are more complex to parse compare to ELF which has more direct memory layout

2. ELF often keeps function names and metadata, even when stripped, PE files are usually stripped of symbols. Debug info (PDB files) is separate and rarely available.

3. PE has Import Address Tables (IAT) which requires a debugger to trace

4. Calling conventions and ABI differences

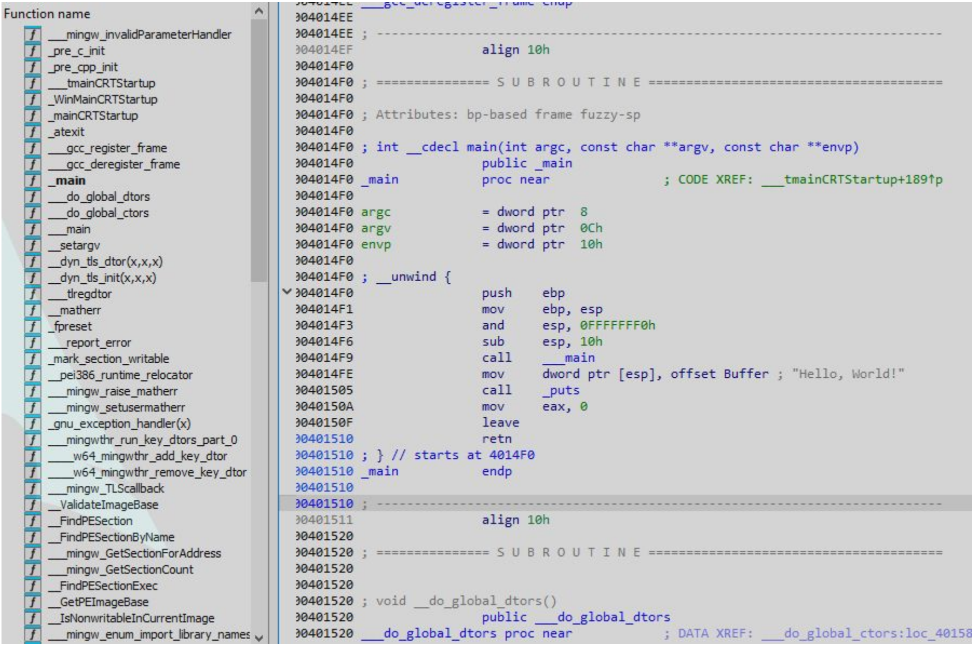
ELF



```
Function name
_init_proc
sub_1020
__libc_start_main
_puts
_start
sub_1078
__x86_get_pc_thunk_bx
sub_1090
sub_10D0
sub_1120
sub_1170
__x86_get_pc_thunk_dx
main
__x86_get_pc_thunk_ax
_term_proc
__libc_start_main
__cxa_finalize
puts
```

```
17D ; Attributes: bp-based frame fuzzy-sp
17D
17D ; int __cdecl main(int argc, const char **argv, const char **envp)
17D                 public main
17D main            proc near               ; DATA XREF: .got:main_ptr↓o
17D
17D argc            = dword ptr  8
17D argv            = dword ptr  0Ch
17D envp            = dword ptr  10h
17D
17D ; __unwind {
17D                 lea     ecx, [esp+4]
181                 and     esp, 0FFFFFFF0h
184                 push    dword ptr [ecx-4]
187                 push    ebp
188                 mov     ebp, esp
18A                 push    ebx
18B                 push    ecx
18C                 call    __x86_get_pc_thunk_ax
191                 add     eax, (offset _GLOBAL_OFFSET_TABLE_ - $)
196                 sub     esp, 0Ch
199                 lea     edx, (aHelloWorld - 3FF4h)[eax] ; "Hello, World!"
19F                 push    edx
1A0                 mov     ebx, eax
1A2                 call    _puts
1A7                 add     esp, 10h
1AA                 mov     eax, 0
1AF                 lea     esp, [ebp-8]
1B2                 pop     ecx
1B3                 pop     ebx
1B4                 pop     ebp
1B5                 lea     esp, [ecx-4]
1B8                 retn
1B8 ; } // starts at 117D
1B8 main            endp
```

PE



```
Function name
___mingw_invalidParameterHandler
_pre_c_init
_pre_cpp_init
___tmainCRTStartup
_WinMainCRTStartup
_mainCRTStartup
_atexit
___gcc_register_frame
___gcc_deregister_frame
_main
___do_global_dtors
___do_global_ctors
___main
__setargv
__dyn_tls_dtor(x,x,x)
__dyn_tls_init(x,x,x)
___tlregdtor
__matherr
_fpreset
___report_error
_mark_section_writable
__pei386_runtime_relocator
___mingw_raise_matherr
___mingw_setusermatherr
_gnu_exception_handler(x)
___mingwthr_run_key_dtors_part_0
_____w64_mingwthr_add_key_dtor
_____w64_mingwthr_remove_key_dtor
___mingw_TLScallback
__ValidateImageBase
__FindPESection
__FindPESectionByName
___mingw_GetSectionForAddress
___mingw_GetSectionCount
__FindPESectionExec
__GetPEImageBase
__IsNonwritableInCurrentImage
___mingw_enum_import_library_names
```

```
004014EE ___gcc_deregister_frame endp
004014EE
004014EE ; ---------------------------------------------------------------------------
004014EF                 align 10h
004014F0
004014F0 ; =============== S U B R O U T I N E =======================================
004014F0
004014F0 ; Attributes: bp-based frame fuzzy-sp
004014F0
004014F0 ; int __cdecl main(int argc, const char **argv, const char **envp)
004014F0                 public _main
004014F0 _main           proc near               ; CODE XREF: ___tmainCRTStartup+189↑p
004014F0
004014F0 argc            = dword ptr  8
004014F0 argv            = dword ptr  0Ch
004014F0 envp            = dword ptr  10h
004014F0
004014F0 ; __unwind {
004014F0                 push    ebp
004014F1                 mov     ebp, esp
004014F3                 and     esp, 0FFFFFFF0h
004014F6                 sub     esp, 10h
004014F9                 call    ___main
004014FE                 mov     dword ptr [esp], offset Buffer ; "Hello, World!"
00401505                 call    _puts
0040150A                 mov     eax, 0
0040150F                 leave
00401510                 retn
00401510 ; } // starts at 4014F0
00401510 _main           endp
00401510
00401510 ; ---------------------------------------------------------------------------
00401511                 align 10h
00401520
00401520 ; =============== S U B R O U T I N E =======================================
00401520
00401520
00401520 ; void __do_global_dtors()
00401520                 public ___do_global_dtors
00401520 ___do_global_dtors proc near            ; DATA XREF: ___do_global_ctors:loc_40158
```

./git_gud_at_it

Practise, practise and practise:

1. https://forum.tuts4you.com/files/category/30-challenge-of-reverse-engineering/
2. https://crackmes.one/

Code your own crackmes, break it, try with different concepts like anti-debugger, obfuscation, encryption and packing.

Also, trying with different programming languages like Python, Java, Golang and etc

Read this if you are keen to explore RE: https://fullstackreverser.com/posts/Become-a-Full-Stack-Reverser/