Getting Started with Reverse Engineering :)

Flag Hunters: CTF Workshop | Saturday, 23-08-2025

prepared by Ariff/Rydzze

rydzze@rydzze

```
rydzze@rydzze:~$ whoami
```

“A little bit introduction about myself because why not? :)”

> Muhammad Ariff Ridzlan bin Mohd Faudzi

> Final Year Student – Bachelor of Computer Science (Honours) (**Artificial Intelligence**) @ MMU Melaka

> Malaysia Cybersecurity Camp 2024 (**MCC2024**) Alumni

> Focusing on **Reverse Engineering** and **Binary Exploitation**

> Enterprise SOC Intern @ **TM One**

> Some achievements in local CTFs (N3WBEES)

1. 5th Place in UM Cybersecurity Summit CTF 2025
2. 7th Place in CODE COMBAT [X] I-HACK 2024 CTF
3. 10th Place in APU IBoH CTF 2024 (National)

arffrdzln

rydzze

Rydzze#4966

rydzze@rydzze

```
rydzze@rydzze:~$ cat table_of_contents
```

🤔 🧮

./part_01

> Intro to RE

> ELF File Format

> x86 Assembly

🕵️ 🧑‍💻

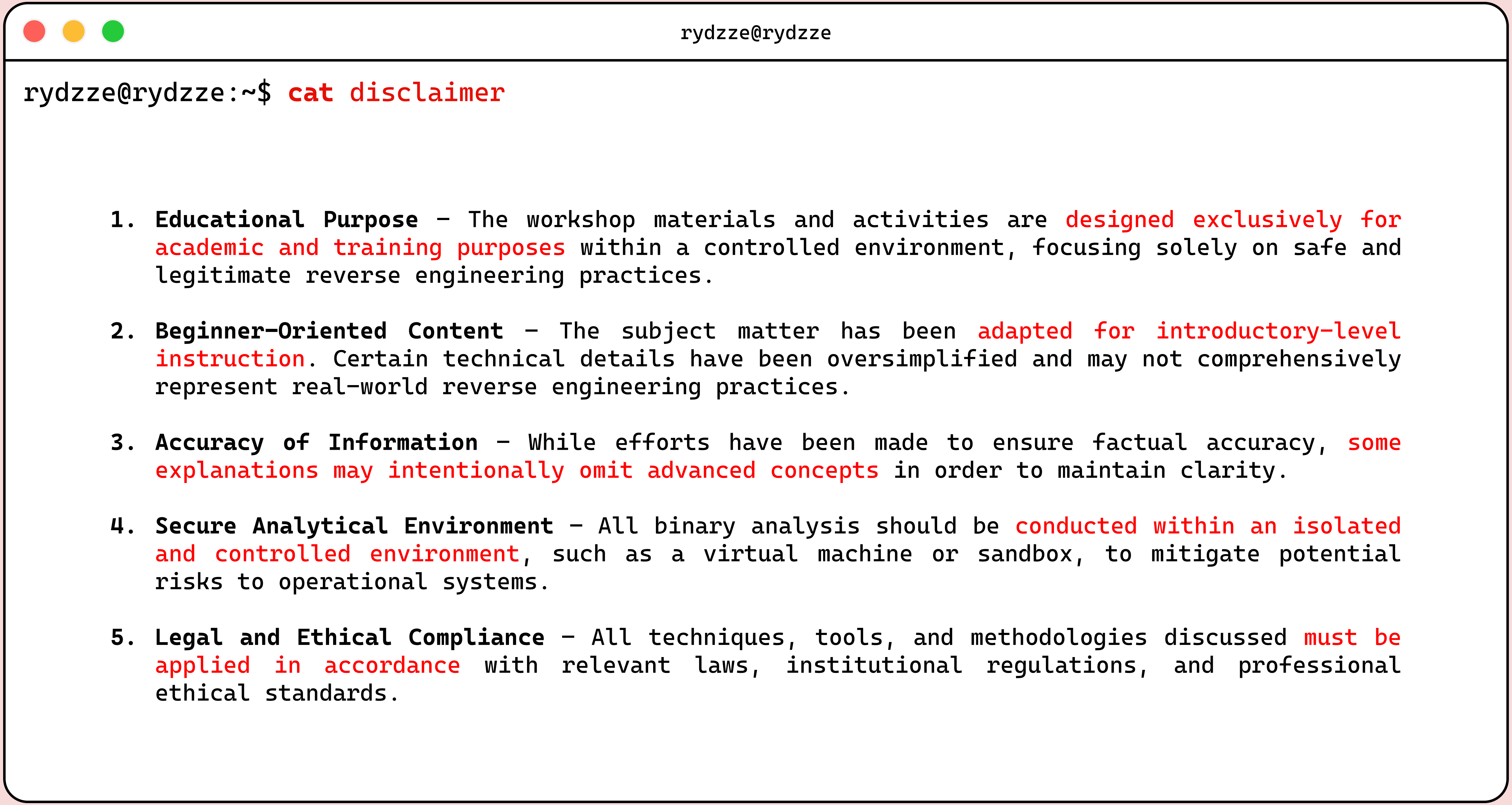
./part_02

> Static Analysis

> Examine Binaries

> Simple Crackme

P.S.: We will be focusing on C language, x86_64 Assembly, and ELF



```
rydzze@rydzze:~$ cat disclaimer
```

1. **Educational Purpose** – The workshop materials and activities are designed exclusively for academic and training purposes within a controlled environment, focusing solely on safe and legitimate reverse engineering practices.

2. **Beginner-Oriented Content** – The subject matter has been adapted for introductory-level instruction. Certain technical details have been oversimplified and may not comprehensively represent real-world reverse engineering practices.

3. **Accuracy of Information** – While efforts have been made to ensure factual accuracy, some explanations may intentionally omit advanced concepts in order to maintain clarity.

4. **Secure Analytical Environment** – All binary analysis should be conducted within an isolated and controlled environment, such as a virtual machine or sandbox, to mitigate potential risks to operational systems.

5. **Legal and Ethical Compliance** – All techniques, tools, and methodologies discussed must be applied in accordance with relevant laws, institutional regulations, and professional ethical standards.

rydzze@rydzze

🤔 🧮

./part_01

> Intro to RE

> ELF File Format

> x86 Assembly

```
rydzze@rydzze:~$ ./part_01
```

> What is **Reverse Engineering?**

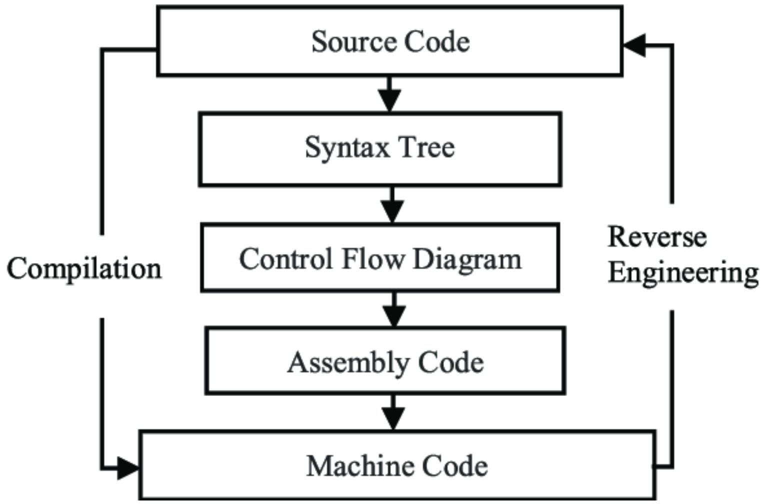
“*Reverse engineering is the* ***process of analysing and understanding the design, structure, and functionality*** *of a product or system by working backward from its final form. It involves taking apart an object or software to uncover its inner workings and understand how it was created*”

> **Forward Engineering**

Idea → Design → Code → Compile → Binary

> **Reverse Engineering**

Binary → Disassemble/Decompile → Analyse Logic → Reconstruct Design → Understand Idea



rydzze@rydzze

```
rydzze@rydzze:~$ ./part_01
```

Let’s take a look at the process of **C Compilation**

> **Preprocessor**

Handles #include and macros to prepare pure C code

> **Compiler**

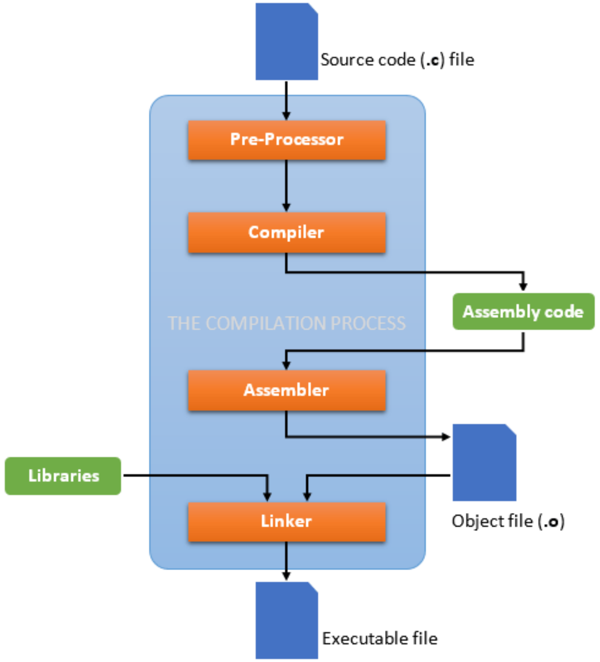
Translates C into optimised assembly

> **Assembler**

Converts assembly into machine-code object files

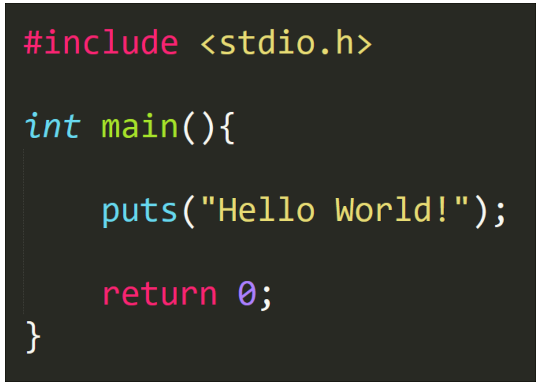
> **Linker**

Joins objects and libraries into an executable



rydzze@rydzze

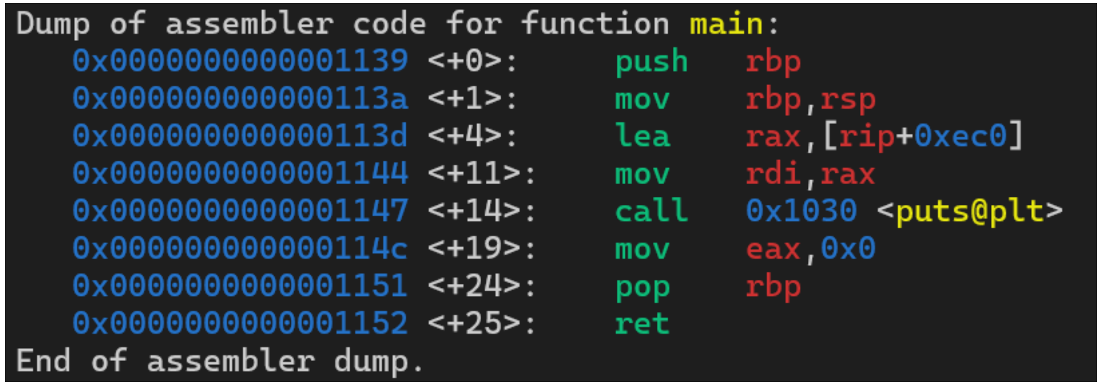
```
rydzze@rydzze:~$ ./part_01
```



```c
#include <stdio.h>

int main(){

    puts("Hello World!");

    return 0;
}
```

compile

>>>>>



```
Dump of assembler code for function main:
   0x0000000000001139 <+0>:     push   rbp
   0x000000000000113a <+1>:     mov    rbp,rsp
   0x000000000000113d <+4>:     lea    rax,[rip+0xec0]
   0x0000000000001144 <+11>:    mov    rdi,rax
   0x0000000000001147 <+14>:    call   0x1030 <puts@plt>
   0x000000000000114c <+19>:    mov    eax,0x0
   0x0000000000001151 <+24>:    pop    rbp
   0x0000000000001152 <+25>:    ret
End of assembler dump.
```

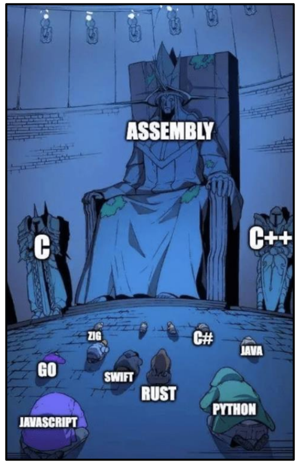
Original Source Code

Disassemble main() using gdb

When you use gdb to see the assembly code of a program:

```
rydzze@rydzze:~$ ./part_01
```

> What does it take to learn **Reverse Engineering?**



Solid Foundations in Programming & Computer Architecture

Hacking in the past
"I reverse engineered binaries stolen from an intelligence agency's server. Here's a zine with shouts, 0day exploit code, and the stdout of me r00ting the server for lulz."
Hacking now
"Metasploit module go brrrr."

Familiarity with Tools & Debugging

Analytical & Problem-Solving Mindset

```
rydzze@rydzze:~$ ./part_01
```

> **Importance** of Reverse Engineering

“Reverse engineering helps to ***understand how software or systems work****, enabling* ***vulnerability discovery****,* ***malware analysis****, interoperability, and security improvements”*

> In **CTF**, it is a challenge where we solve *puzzles* to obtain a flag 🧩🏁 *(usually)*

~~*TW! Claimed to be the hardest category :P*~~

> In **real life**, it is a serious work such as dissecting malware and conducting deep security research 🔍👾 ~~*or crack something*~~

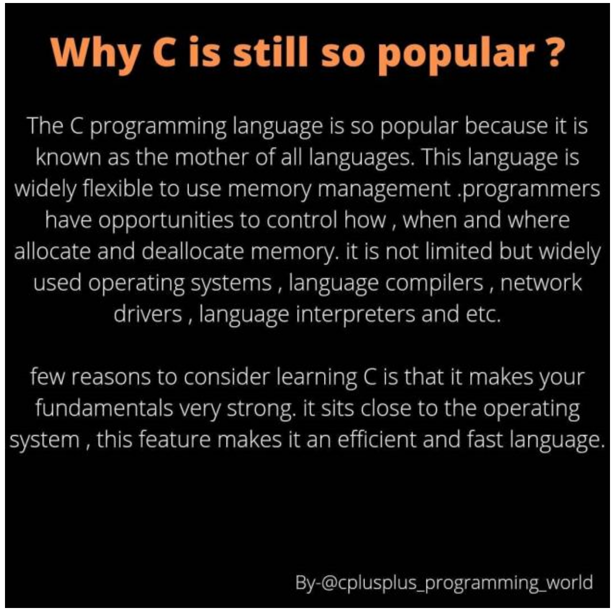
```
rydzze@rydzze:~$ ./part_01
```

Linus Torvalds

(creator of Linux)

If you want ultimate control over hardware, [Assembly] is the way to go – but at the cost of time and complexity.

WHEN YOU REALIZE,
ALL PROGRAMMING LANGUAGES AND OPERATING SYSTEMS ARE SOMEHOW MADE OF C
ASSEMBLY
imgflip.com



Why C is still so popular ?

The C programming language is so popular because it is known as the mother of all languages. This language is widely flexible to use memory management .programmers have opportunities to control how , when and where allocate and deallocate memory. it is not limited but widely used operating systems , language compilers , network drivers , language interpreters and etc.

few reasons to consider learning C is that it makes your fundamentals very strong. it sits close to the operating system , this feature makes it an efficient and fast language.

By-@cplusplus_programming_world

```
rydzze@rydzze:~$ ./part_01
```

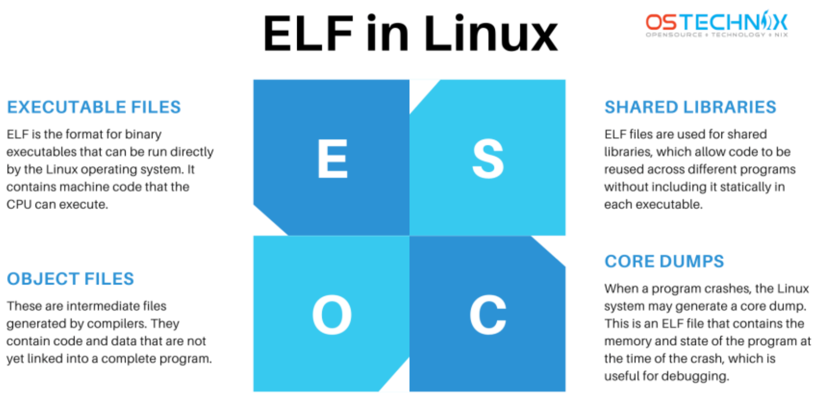
> What is ELF?

“*Executable and Linkable Format (ELF) is a common standard file format used in **Unix-like operating systems**, including Linux, for executable files, object code, shared libraries, and core dumps*”

Used by : Linux and Unix-like systems

Based on: Unix System V ABI

Purpose : Contains executables, shared libraries, and object code



ELF in Linux

EXECUTABLE FILES

ELF is the format for binary executables that can be run directly by the Linux operating system. It contains machine code that the CPU can execute.

OBJECT FILES

These are intermediate files generated by compilers. They contain code and data that are not yet linked into a complete program.

SHARED LIBRARIES

ELF files are used for shared libraries, which allow code to be reused across different programs without including it statically in each executable.

CORE DUMPS

When a program crashes, the Linux system may generate a core dump. This is an ELF file that contains the memory and state of the program at the time of the crash, which is useful for debugging.

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_01
```

> The Structure of **ELF file format**

ELF Header: The first structure in an ELF file with metadata (file type, arch…, entry point)

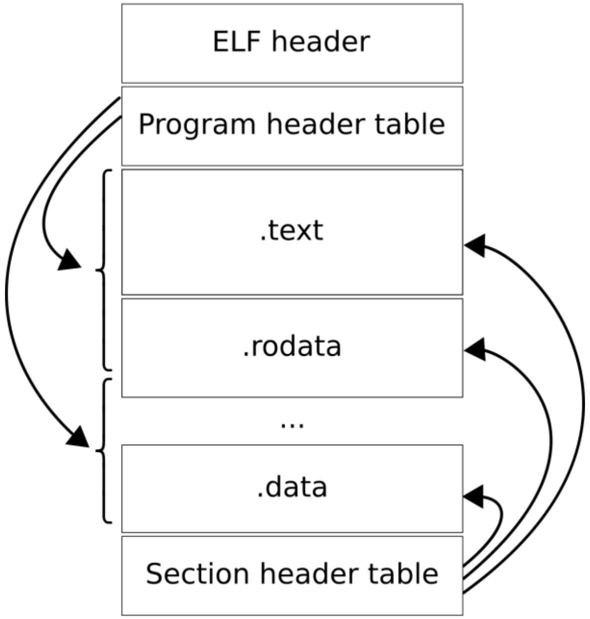
Program Header Table: Describes segments for loading into memory at runtime

.text : Section containing executable code

.rodata: Section storing read-only data like constants and strings

.data : Section holding initialized global and static variables

Section Header Table: Contains info about all sections, (mainly for linking, debugging)



rydzze@rydzze

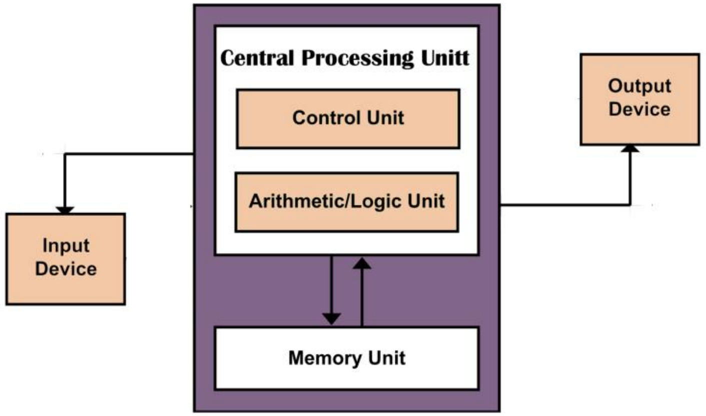
```
rydzze@rydzze:~$ ./part_01
```

Basic of x86 Assembly

```
rydzze@rydzze:~$ ./part_01
```

> Before that, what is **x86 architecture**?

> *“x86 is a family of CISC instruction set architectures used in most PCs, laptops, and servers, originally developed by Intel. The name comes from early Intel CPUs like the 8086, 80186, 80286, 80386, and 80486, all ending in "86"”*



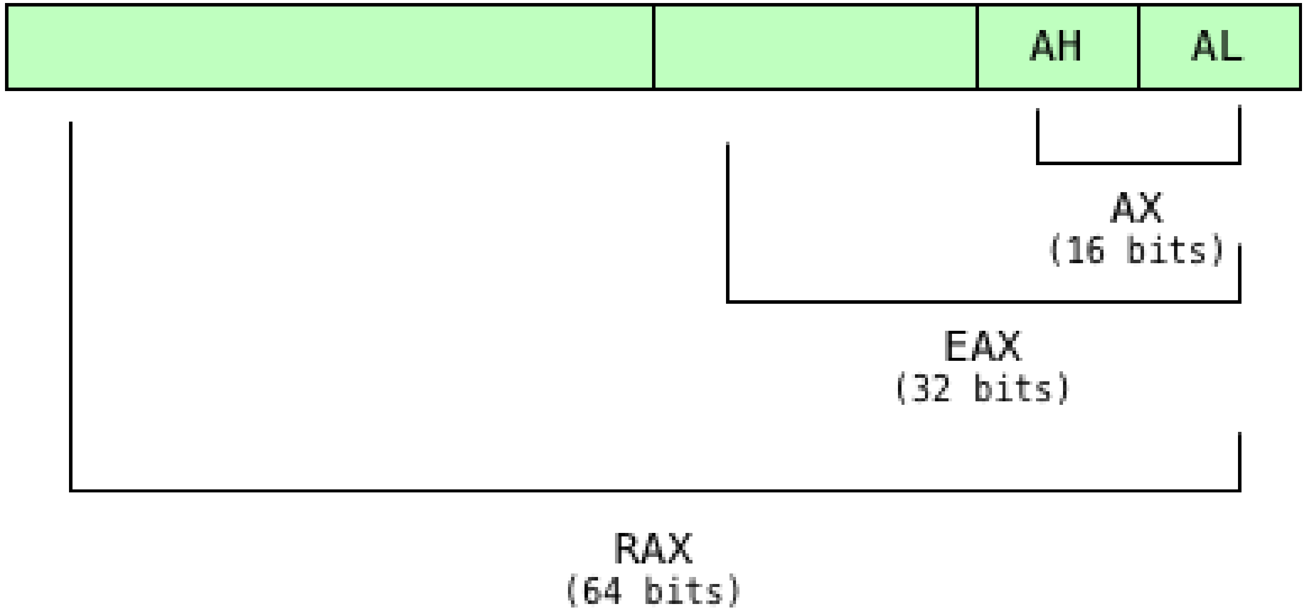
> **Key Facts**

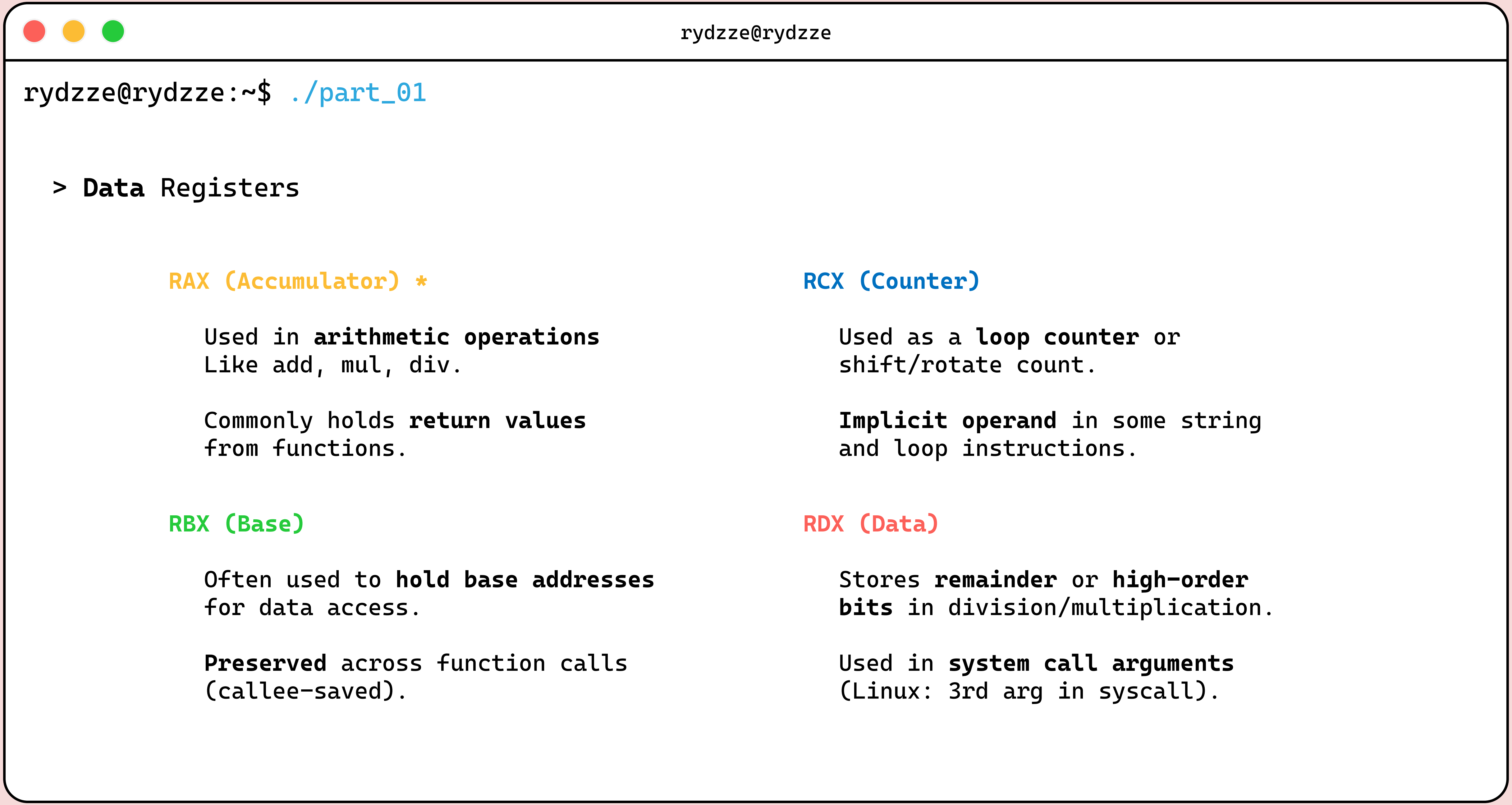
- Introduced in 1978 with the Intel 8086.
- Evolved from 16-bit to 32-bit (IA-32) and 64-bit (x86-64 or AMD64).
- Supports strong backward compatibility.

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_01
```

> **Bit Sizes** of Registers





```
rydzze@rydzze:~$ ./part_01
```

> **Data** Registers

RAX (Accumulator) *

Used in **arithmetic operations**
Like add, mul, div.

Commonly holds **return values**
from functions.

RBX (Base)

Often used to **hold base addresses**
for data access.

Preserved across function calls
(callee-saved).

RCX (Counter)

Used as a **loop counter** or
shift/rotate count.

Implicit operand in some string
and loop instructions.

RDX (Data)

Stores **remainder** or **high-order**
bits in division/multiplication.

Used in **system call arguments**
(Linux: 3rd arg in syscall).

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_01
```

> **Pointer** Registers

RSP (Stack Pointer)

Points to the **top of the stack**.
Adjusted by `push`, `pop`, `call`, and `ret`.

RBP (Base Pointer)

Points to the **current stack frame**.
Used to **access local variables** and **function args**.

> **Index** Registers

RSI (Source Index)

Source address in memory ops.
Holds **2nd argument** in 64-bit calls.

RDI (Destination Index)

Destination address in memory ops.
Holds **1st argument** in 64-bit calls.

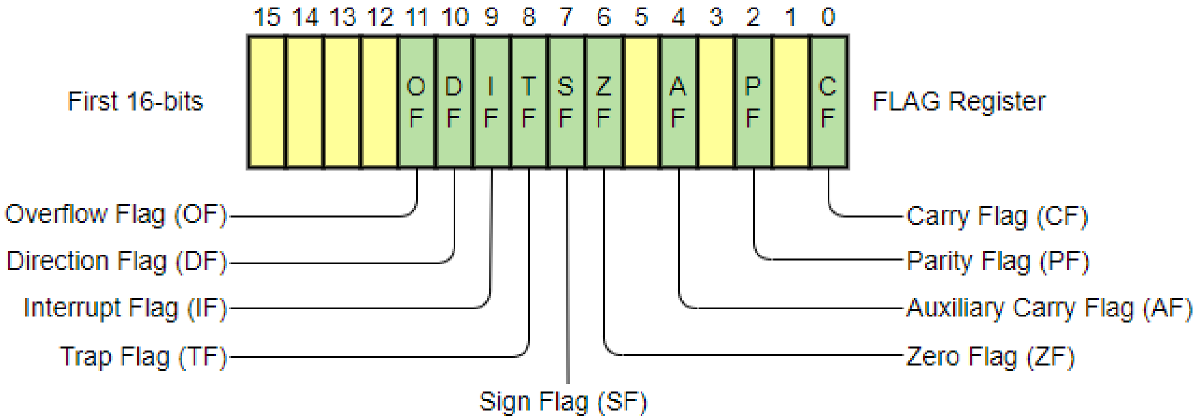
RIP (Instruction Pointer) *

Points to **next instruction**.
Changes with jumps, calls, returns.

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_01
```

> **Flag** Registers



Usage of Flag Registers

- Set by arithmetic/logic instructions.
- Checked by conditional jumps (e.g., JE, JNE).

```
rydzze@rydzze:~$ ./part_01
```

> Instruction Format

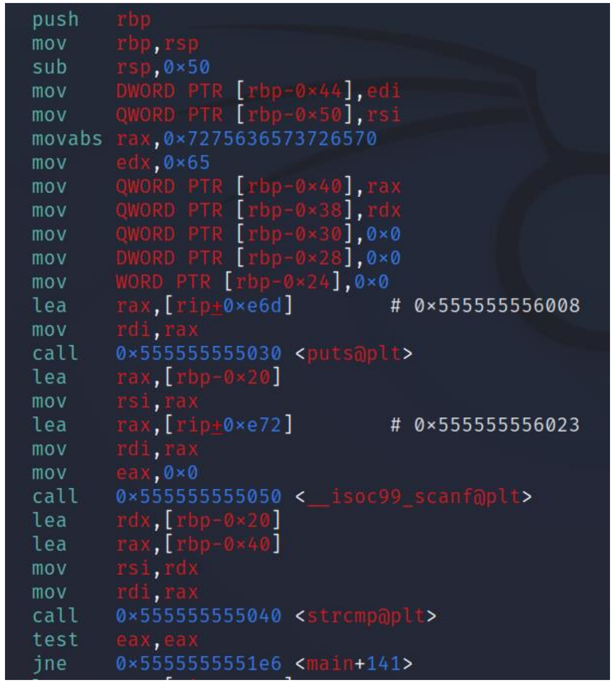
```
<mnemonic> <destination>, <source>
```

> Operands

Register, memory, or immediate values.

> Examples

```
mov rax, 5    ;Move immediate value to register.

add rax, rbx  ;Add two registers.

cmp rax, rbx  ;Compare values.

jmp label     ;Jump to label.
```



```
push   rbp
mov    rbp,rsp
sub    rsp,0x50
mov    DWORD PTR [rbp-0x44],edi
mov    QWORD PTR [rbp-0x50],rsi
movabs rax,0x7275636573726570
mov    edx,0x65
mov    QWORD PTR [rbp-0x40],rax
mov    QWORD PTR [rbp-0x38],rdx
mov    QWORD PTR [rbp-0x30],0x0
mov    DWORD PTR [rbp-0x28],0x0
mov    WORD PTR [rbp-0x24],0x0
lea    rax,[rip+0xe6d]        # 0x555555556008
mov    rdi,rax
call   0x555555555030 <puts@plt>
lea    rax,[rbp-0x20]
mov    rsi,rax
lea    rax,[rip+0xe72]        # 0x555555556023
mov    rdi,rax
mov    eax,0x0
call   0x555555555050 <__isoc99_scanf@plt>
lea    rdx,[rbp-0x20]
lea    rax,[rbp-0x40]
mov    rsi,rdx
mov    rdi,rax
call   0x555555555040 <strcmp@plt>
test   eax,eax
jne    0x5555555551e6 <main+141>
```

🕵️ 🧑‍💻

./part_02

> Static Analysis

> Examine Binaries

> Simple Crackme

rydzze@rydzze

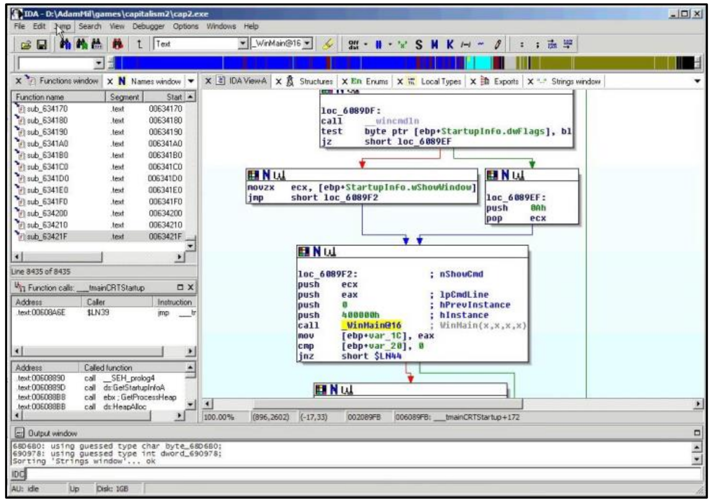
```
rydzze@rydzze:~$ ./part_02
```

> What is **Static Analysis**?

> “*Static analysis is the process of **examining a program’s binary or source code without running it**, in order to uncover its structure, logic flow, and data layouts*”

> **Key Objectives**

- Identify functions, loops, and branching logic
- Extract and interpret embedded resources
- Construct control-flow and data-flow representations to map how data moves through the code



IDA - D:\AdamMil\games\capitalism2\cap2.exe

Function name	Segment	Start
sub_634170	.text	00634170
sub_634180	.text	00634180
sub_634190	.text	00634190
sub_6341A0	.text	006341A0
sub_6341B0	.text	006341B0
sub_6341C0	.text	006341C0
sub_6341D0	.text	006341D0
sub_6341E0	.text	006341E0
sub_6341F0	.text	006341F0
sub_634200	.text	00634200
sub_634210	.text	00634210
sub_63421F	.text	0063421F

Line 8435 of 8435

Function calls: ___tmainCRTStartup

Address	Caller	Instruction
.text:00608A6E	$LN39	jmp ___tr

Address	Called function
.text:00608890	call __SEH_prolog4
.text:0060889D	call ds:GetStartupInfoA
.text:006088B8	call ebx ; GetProcessHeap
.text:006088BB	call ds:HeapAlloc

```
loc_6089DF:
call    __wincmdln
test    byte ptr [ebp+StartupInfo.dwFlags], bl
jz      short loc_6089EF
```

```
movzx   ecx, [ebp+StartupInfo.wShowWindow]
jmp     short loc_6089F2
```

```
loc_6089EF:
push    0Ah
pop     ecx
```

```
loc_6089F2:                     ; nShowCmd
push    ecx
push    eax                     ; lpCmdLine
push    0                       ; hPrevInstance
push    400000h                 ; hInstance
call    _WinMain@16             ; WinMain(x,x,x,x)
mov     [ebp+var_1C], eax
cmp     [ebp+var_20], 0
jnz     short $LN44
```

100.00% (896,2602) (-17,33) 002089FB 006089FB: ___tmainCRTStartup+172

Output window

```
68D680: using guessed type char byte_68D680;
690978: using guessed type int dword_690978;
Sorting 'Strings window'... ok
```

IDC

AU: idle Up Disk: 1GB

```
rydzze@rydzze:~$ ./part_02
```

> What is **Disassembler?**

"A disassembler is a tool that converts raw machine-code bytes into human-readable assembly instructions. By mapping opcodes to mnemonics and showing registers, calls, and jumps, it helps you trace exactly what the processor will execute"

> What is **Decompiler?**

"A decompiler is a more advanced tool that attempts to reconstruct higher-level source-like code (such as C or C++) from a compiled binary. It abstracts away low-level assembly into functions, loops, and data structures, making complex logic way easier to understand"

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_02; ./bin1
```

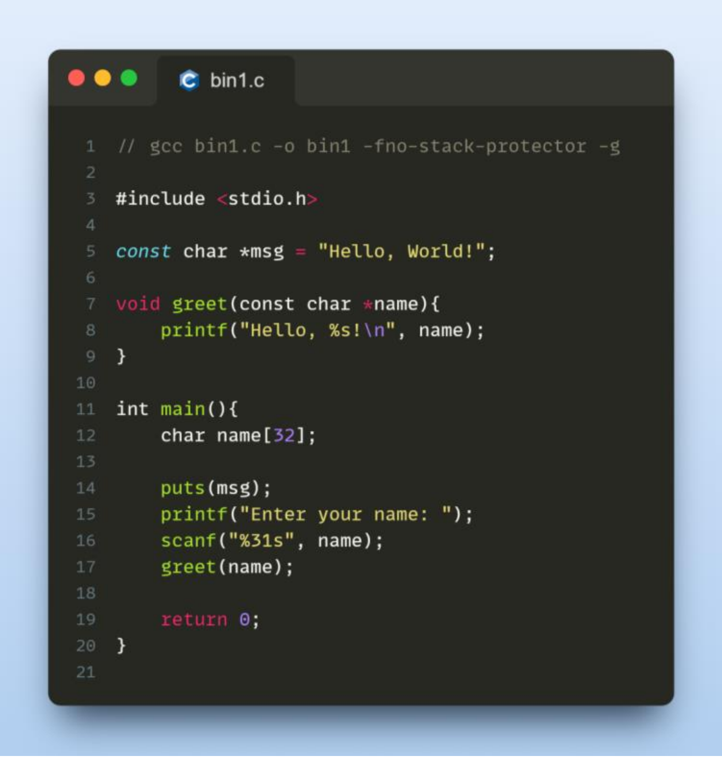
> Let's start our **hands-on** with **bin1**

"This C program prints "Hello, World!", asks the user for their name, and then greets them using that name"

> Load the ELF binary inside IDA :D

> Before that, try out these command-line utils !!!

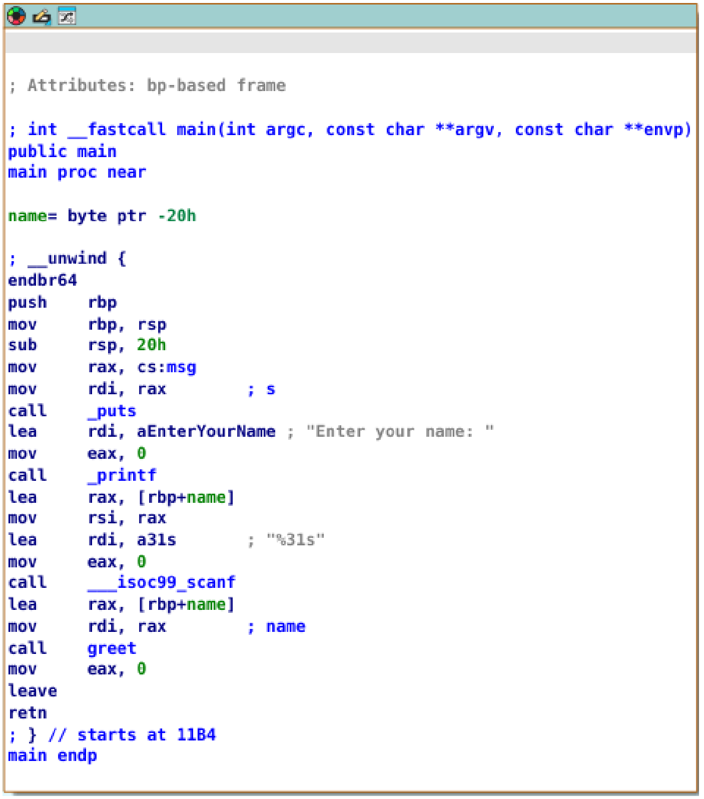
- file
- ldd
- strings
- readelf
- objdump



```c
// gcc bin1.c -o bin1 -fno-stack-protector -g

#include <stdio.h>

const char *msg = "Hello, World!";

void greet(const char *name){
    printf("Hello, %s!\n", name);
}

int main(){
    char name[32];

    puts(msg);
    printf("Enter your name: ");
    scanf("%31s", name);
    greet(name);

    return 0;
}

```

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_02; ./bin1
```



Function prologue

puts(msg);

printf("Enter your name: ");

scanf("%31s", name);

greet(name);

Function epilogue

```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

name= byte ptr -20h

; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     rax, cs:msg
mov     rdi, rax        ; s
call    _puts
lea     rdi, aEnterYourName ; "Enter your name: "
mov     eax, 0
call    _printf
lea     rax, [rbp+name]
mov     rsi, rax
lea     rdi, a31s       ; "%31s"
mov     eax, 0
call    ___isoc99_scanf
lea     rax, [rbp+name]
mov     rdi, rax        ; name
call    greet
mov     eax, 0
leave
retn
; } // starts at 11B4
main endp
```

But what is function prologue and function epilogue?

rydzze@rydzze

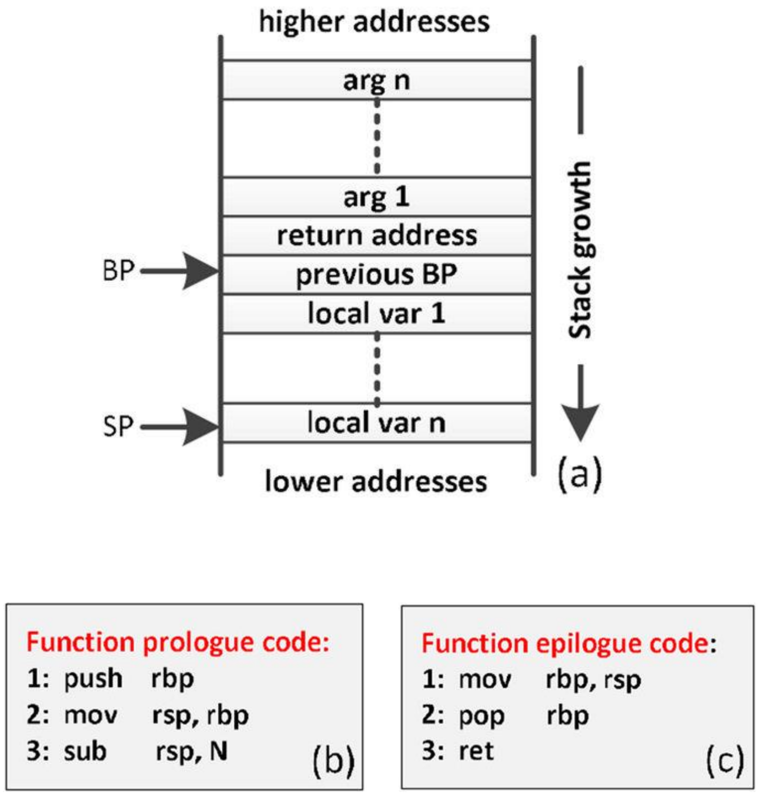
```
rydzze@rydzze:~$ ./part_02; ./bin2
```

> Function Prologue

Function prologue is the setup code at the start of a function. It prepares the stack frame by saving the old base pointer (RBP) and setting up a new base pointer for the function’s local variables and parameters.

> Function Epilogue

Function epilogue is the cleanup code at the end of a function. It restores the saved base pointer and stack pointer to their previous state before returning control to the caller.



Function prologue code:

```
1: push   rbp
2: mov    rsp, rbp
3: sub    rsp, N
```

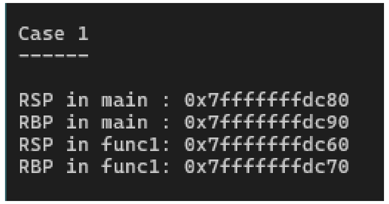
Function epilogue code:

```
1: mov    rbp, rsp
2: pop    rbp
3: ret
```

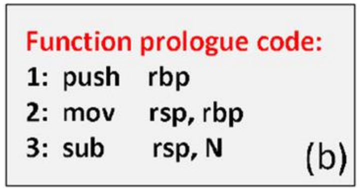
rydzze@rydzze

```
rydzze@rydzze:~$ ./part_02; ./bin2
```

> Case 1



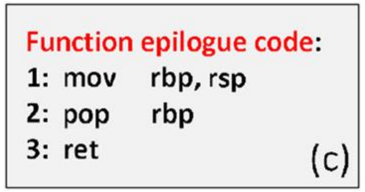
```
Case 1
------

RSP in main : 0x7fffffffdc80
RBP in main : 0x7fffffffdc90
RSP in func1: 0x7fffffffdc60
RBP in func1: 0x7fffffffdc70
```



Function prologue code:

```
1: push   rbp
2: mov    rsp, rbp
3: sub    rsp, N
```

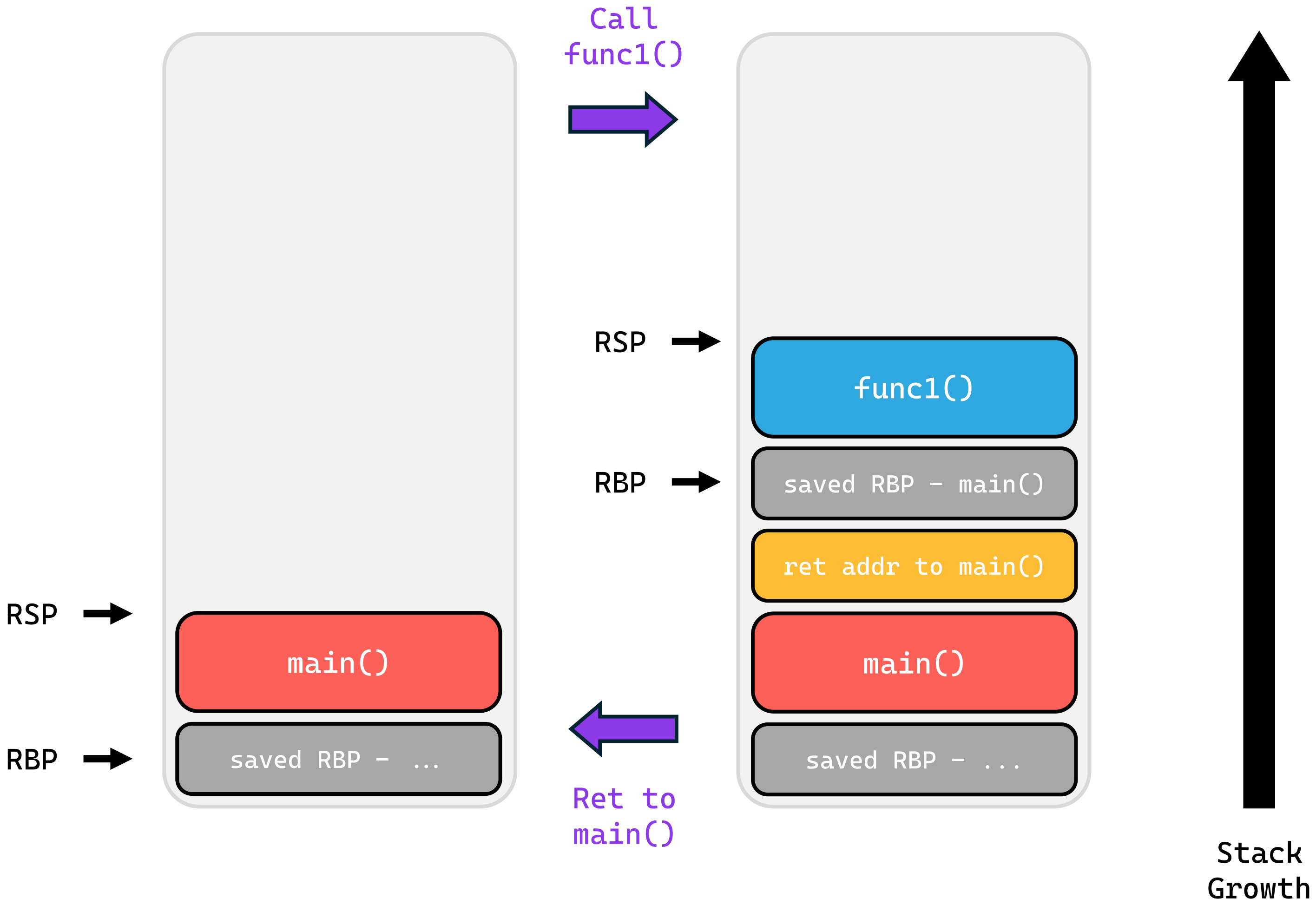
(b)



Function epilogue code:

```
1: mov    rbp, rsp
2: pop    rbp
3: ret
```

(c)

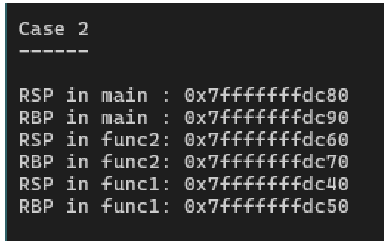


sorry if it is not accurate :)

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_02; ./bin2
```

> Case 2

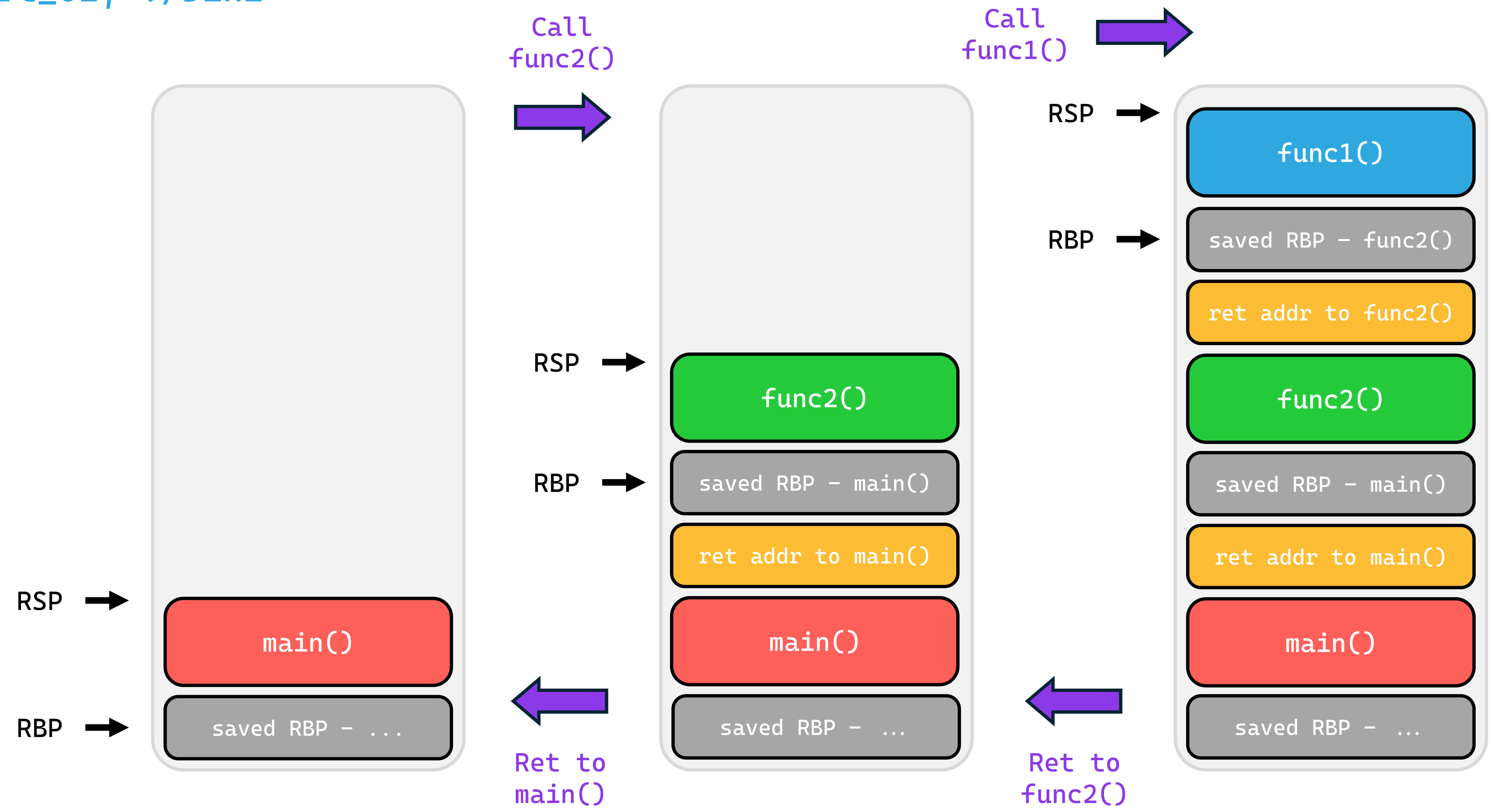


```
Case 2
------

RSP in main : 0x7fffffffdc80
RBP in main : 0x7fffffffdc90
RSP in func2: 0x7fffffffdc60
RBP in func2: 0x7fffffffdc70
RSP in func1: 0x7fffffffdc40
RBP in func1: 0x7fffffffdc50
```

Function prologue code:
1: push rbp
2: mov rsp, rbp
3: sub rsp, N
(b)

Function epilogue code:
1: mov rbp, rsp
2: pop rbp
3: ret
(c)

Direction of Stack Growth goes up



sorry if it is not accurate :)

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_02; ./bin3
```

> **Warm-up** – Password Checker

This login program claims to be “*ultra secure*” with a very long password and strict checks.

Your mission is simple, figure out the correct username and password to bypass the authentication and gain access.

Objective?

Analyse the provided binary, find the hidden credentials, and make the program print “Access Granted!”

Password: firstpassword

```
rydzze@rydzze:~$ ./part_02; ./bin3
```

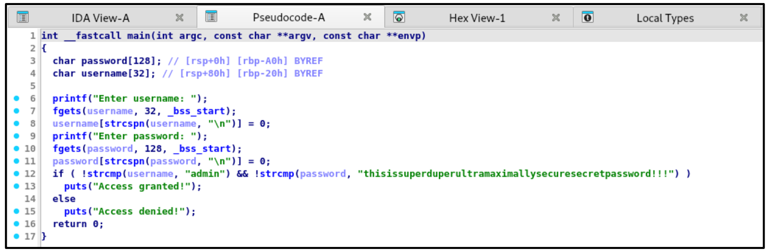
> What is **strcmp()**?

strcmp() is a C standard library function that compares two strings and returns an integer based on their lexicographical difference.

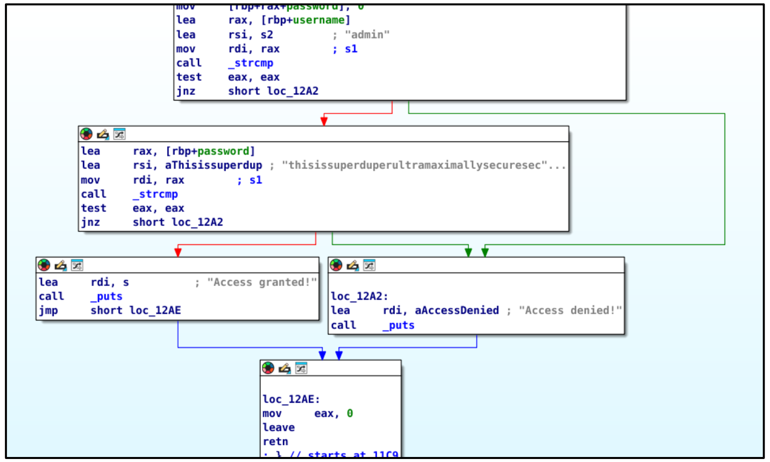
```
strcmp( first_str , second_str );
        $rdi        $rsi
```

In short, returns 0 if the strings are equal. Otherwise, returns non-zero if different.

NOTE!!! It changes CPU flags, especially the Zero Flag (ZF) to track equality or difference between characters.



```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  char password[128]; // [rsp+0h] [rbp-A0h] BYREF
  char username[32]; // [rsp+80h] [rbp-20h] BYREF

  printf("Enter username: ");
  fgets(username, 32, _bss_start);
  username[strcspn(username, "\n")] = 0;
  printf("Enter password: ");
  fgets(password, 128, _bss_start);
  password[strcspn(password, "\n")] = 0;
  if ( !strcmp(username, "admin") && !strcmp(password, "thisissuperduperultramaximallysecuresecretpassword!!!") )
    puts("Access granted!");
  else
    puts("Access denied!");
  return 0;
}
```



```
mov     [rbp+rax+password], 0
lea     rax, [rbp+username]
lea     rsi, s2         ; "admin"
mov     rdi, rax        ; s1
call    _strcmp
test    eax, eax
jnz     short loc_12A2
```

```
lea     rax, [rbp+password]
lea     rsi, aThisissuperdup ; "thisissuperduperultramaximallysecuresec"...
mov     rdi, rax        ; s1
call    _strcmp
test    eax, eax
jnz     short loc_12A2
```

```
lea     rdi, s          ; "Access granted!"
call    _puts
jmp     short loc_12AE
```

```
loc_12A2:
lea     rdi, aAccessDenied ; "Access denied!"
call    _puts
```

```
loc_12AE:
mov     eax, 0
leave
retn
; } // starts at 11C9
```

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_02; ./bin3
```

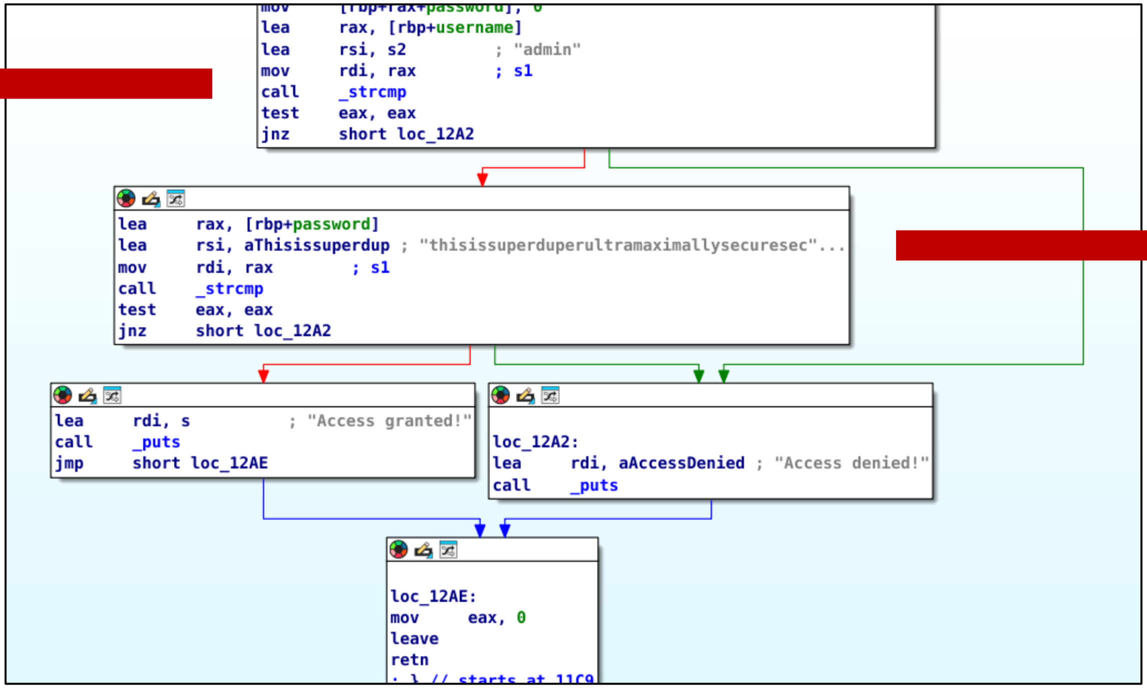
1. Load the username as second arg
2. Load our input as first arg
3. Compare arguments
4. Set/Clear ZF

Jump if Not Zero (JNZ)

ZF = 0? Jump

ZF = 1? Nuh uh

P.S.: another name of JNZ is Jump if Not Equal (JNE)



```
mov     [rbp+rax+password], 0
lea     rax, [rbp+username]
lea     rsi, s2         ; "admin"
mov     rdi, rax        ; s1
call    _strcmp
test    eax, eax
jnz     short loc_12A2
```

```
lea     rax, [rbp+password]
lea     rsi, aThisissuperdup ; "thisissuperduperultramaximallysecuresec"...
mov     rdi, rax        ; s1
call    _strcmp
test    eax, eax
jnz     short loc_12A2
```

```
lea     rdi, s          ; "Access granted!"
call    _puts
jmp     short loc_12AE
```

```
loc_12A2:
lea     rdi, aAccessDenied ; "Access denied!"
call    _puts
```

```
loc_12AE:
mov     eax, 0
leave
retn
; } // starts at 11C9
```

1. Load the password as second arg
2. Load our input as first arg
3. Compare arguments
4. Set/Clear ZF

Jump if Not Zero (JNZ)

ZF = 0? Jump

ZF = 1? Nuh uh

P.S.: another name of JNZ is Jump if Not Equal (JNE)

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_02; ./bin4
```

> **Warm-up** – Flag Checker

We’ve hidden the flag inside this binary, but it’s not stored in plain text.
The program will encrypt your input and compare it to a secret value.

Your job is to figure out the original flag that, once encrypted, matches the hidden one.

Objective?

Analyse the code, reverse the encryption, and uncover the correct flag.

Password: definitelynotsecondpassword

```
rydzze@rydzze:~$ ./part_02; ./bin4
```

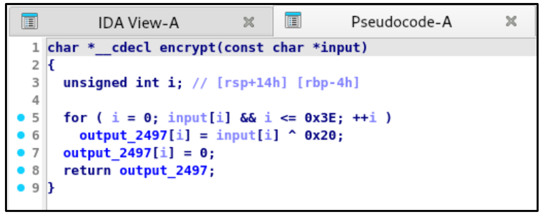
> What is XOR?

Exclusive OR, or XOR is a bitwise operation that returns 1 if exactly one of the two bits is 1, and 0 otherwise.

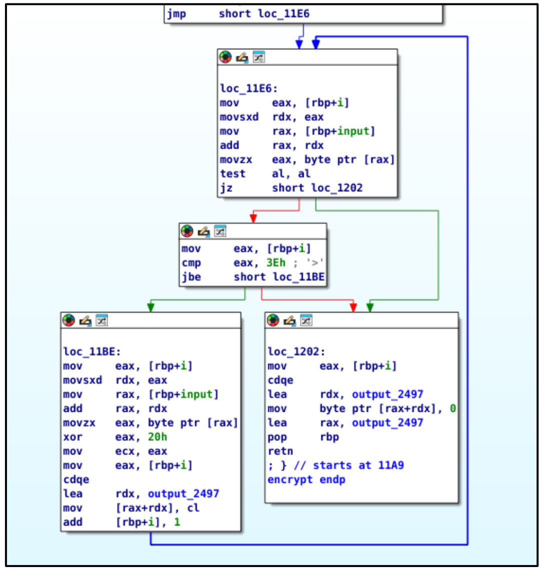
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Bit Position	7	6	5	4	3	2	1	0
Char 'A' (0x41)	0	1	0	0	0	0	0	1
Key (0x0F)	0	0	0	0	1	1	1	1
Result (0x4E)	0	1	0	0	1	1	1	0

NOTE!!! XOR is used in reverse engineering to hide or reveal data by easily reversing simple encryptions or obfuscations.



```
char *__cdecl encrypt(const char *input)
{
  unsigned int i; // [rsp+14h] [rbp-4h]

  for ( i = 0; input[i] && i <= 0x3E; ++i )
    output_2497[i] = input[i] ^ 0x20;
  output_2497[i] = 0;
  return output_2497;
}
```

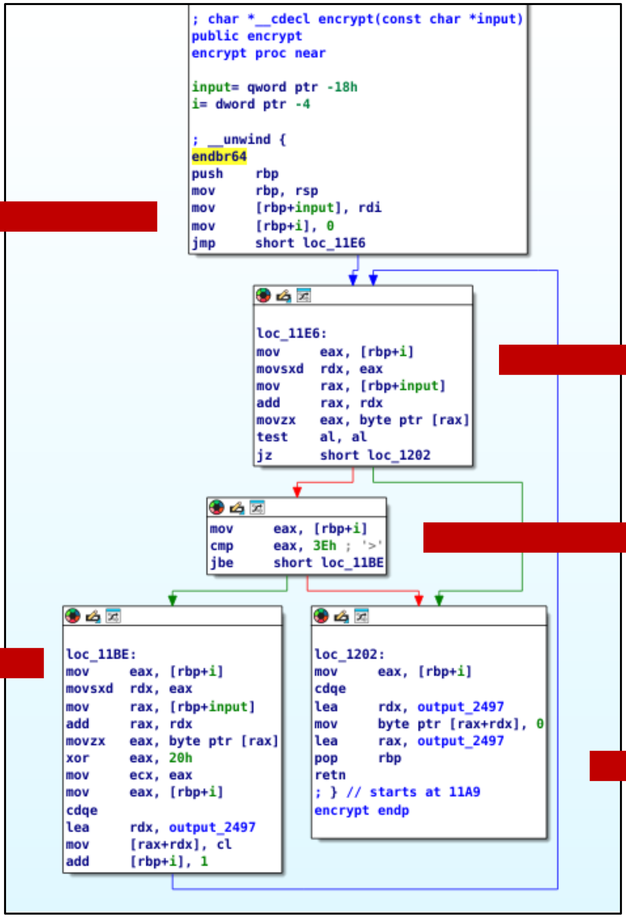


```
jmp     short loc_11E6

loc_11E6:
mov     eax, [rbp+i]
movsxd  rdx, eax
mov     rax, [rbp+input]
add     rax, rdx
movzx   eax, byte ptr [rax]
test    al, al
jz      short loc_1202

mov     eax, [rbp+i]
cmp     eax, 3Eh ; '>'
jbe     short loc_11BE

loc_11BE:
mov     eax, [rbp+i]
movsxd  rdx, eax
mov     rax, [rbp+input]
add     rax, rdx
movzx   eax, byte ptr [rax]
xor     eax, 20h
mov     ecx, eax
mov     eax, [rbp+i]
cdqe
lea     rdx, output_2497
mov     [rax+rdx], cl
add     [rbp+i], 1

loc_1202:
mov     eax, [rbp+i]
cdqe
lea     rdx, output_2497
mov     byte ptr [rax+rdx], 0
lea     rax, output_2497
pop     rbp
retn
; } // starts at 11A9
encrypt endp
```

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_02; ./bin4
```

1. Copy our input from first arg
2. Initialise var, i = 0
3. Jump to block at below …



```
; char *__cdecl encrypt(const char *input)
public encrypt
encrypt proc near

input= qword ptr -18h
i= dword ptr -4

; __unwind {
endbr64
push    rbp
mov     rbp, rsp
mov     [rbp+input], rdi
mov     [rbp+i], 0
jmp     short loc_11E6
```

```
loc_11E6:
mov     eax, [rbp+i]
movsxd  rdx, eax
mov     rax, [rbp+input]
add     rax, rdx
movzx   eax, byte ptr [rax]
test    al, al
jz      short loc_1202
```

```
mov     eax, [rbp+i]
cmp     eax, 3Eh ; '>'
jbe     short loc_11BE
```

```
loc_11BE:
mov     eax, [rbp+i]
movsxd  rdx, eax
mov     rax, [rbp+input]
add     rax, rdx
movzx   eax, byte ptr [rax]
xor     eax, 20h
mov     ecx, eax
mov     eax, [rbp+i]
cdqe
lea     rdx, output_2497
mov     [rax+rdx], cl
add     [rbp+i], 1
```

```
loc_1202:
mov     eax, [rbp+i]
cdqe
lea     rdx, output_2497
mov     byte ptr [rax+rdx], 0
lea     rax, output_2497
pop     rbp
retn
; } // starts at 11A9
encrypt endp
```

Loop Termination Condition

1. Check if input[i] == 0
   (no more character?)

   If yes, jump to exit

2. Compare i with 0x3e / 62
   (exceed allocated size?)

   If i > 62, jump to exit

XOR Obfuscation Process

1. Copy a char from our input
2. ASCII char XOR with 0x20
3. Save it into an array, output

Return to main() with output as return value

rydzze@rydzze

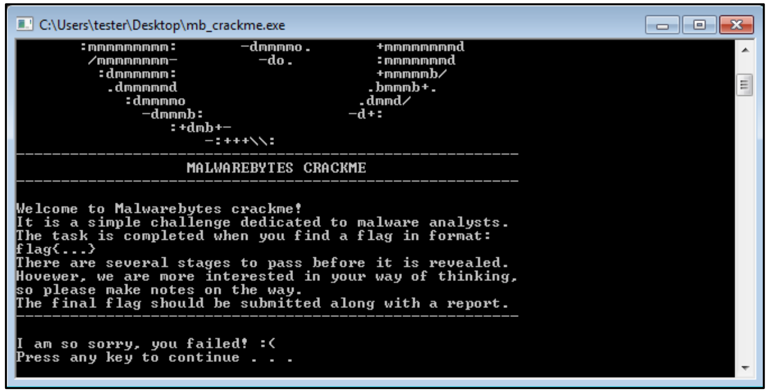
```
rydzze@rydzze:~$ ./part_02; ./crackme
```

> What is **crackme challenge**?

*“A crackme challenge is a specially created program meant to **test a person’s ability to reverse engineer software** by uncovering hidden information or bypassing built-in protections”*

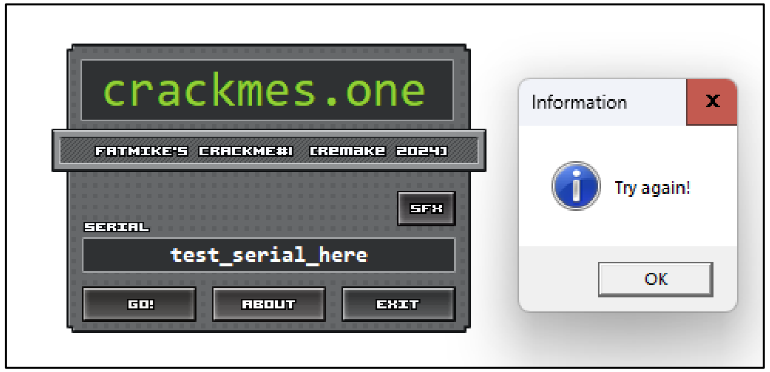
It involves analysing code using tools like debuggers or disassemblers to find keys, passwords, or bypass security checks

Why? To develop and improve their skills in reverse engineering and software security, as well as to learn how software protections operate



C:\Users\tester\Desktop\mb_crackme.exe

```
       :mmmmmmmmm:       -dmmmmo.       +mmmmmmmmd
        /mmmmmmmm-         -do.         :mmmmmmmd
         :dmmmmmm:                      +mmmmmb/
          .dmmmmmd                     .bmmmb+.
            :dmmmmo                   .dmmd/
              -dmmmb:               -d+:
                 :+dmb+-
                     -:+++\\:
------------------------------------------------------
                   MALWAREBYTES CRACKME
------------------------------------------------------

Welcome to Malwarebytes crackme!
It is a simple challenge dedicated to malware analysts.
The task is completed when you find a flag in format:
flag{...}
There are several stages to pass before it is revealed.
Hovever, we are more interested in your way of thinking,
so please make notes on the way.
The final flag should be submitted along with a report.
------------------------------------------------------

I am so sorry, you failed! :(
Press any key to continue . . .
```



SERIAL: test_serial_here

Try again!

rydzze@rydzze

```
rydzze@rydzze:~$ ./part_02; ./crackme
```

> **Short Test** – Crackme Challenge

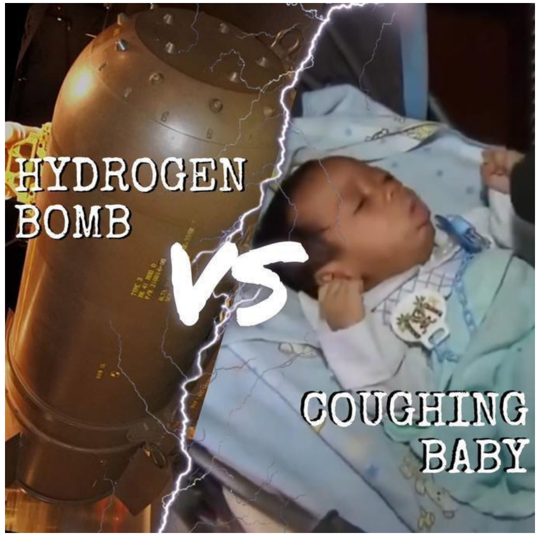
This challenge presents a simple crackme that validates a single correct flag. When executed, it prompts for your input and performs a series of internal checks to determine if the flag is valid.

Only the exact flag will pass these checks.

Objective?

Analyse the binary, determine the correct flag that it’s expecting, and make the program display the success message.

Password: myfirstcrackmeyippee



rydzze@rydzze

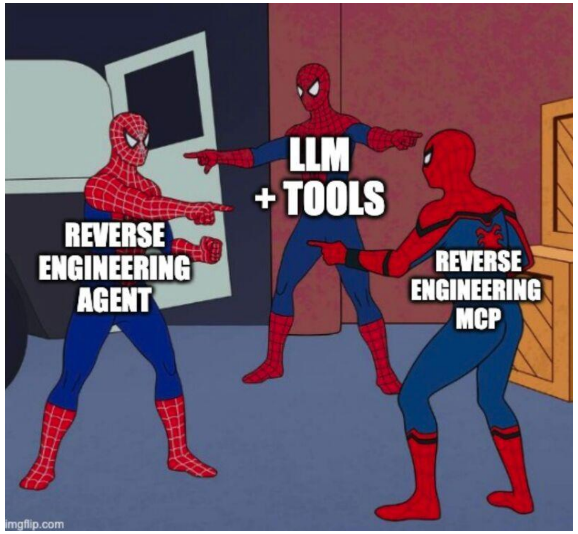
```
rydzze@rydzze:~$ cat epilogue
```

> How to get good at **Reverse Engineering**?

1. **Build a Solid Low-Level Foundation**
   Learn how assembly works, understand CPU registers, memory layout, and calling conventions

2. **Just like *math*, all you need is practices**
   The more binaries you take apart, the better you will recognise patterns and common tricks

3. **Mix Static and Dynamic Analysis**
   Combine tools like IDA/Ghidra for reading code and x64dbg/GDB for running and stepping through it



> ***Learning* Platforms**

MY : SKR CTF, EQCTF, ...
INT : picoCTF, HTB, THM

> **Reverse Engineering**

Malware Unicorn RE101, Intro to RE, crackmes.one, and more :)

> **Binary Exploitation**

pwn.college, pwnable.tw, and more lol :)

rydzze@rydzze

```
rydzze@rydzze:~$ sudo rm -rf /
```

Thanks for Attending!

Keep exploring and enjoying your reverse engineering journey 😇

Uhmmm ... Q&A?