National Institute Of Science And Technology Of Tunisia

x86 and x64 Architectures

Authors :

Name and Forename :

Med Ali Ouachani

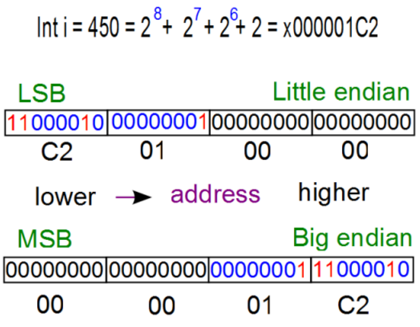
Iyed Mejri

Year :
2022 - 2023

I - Introduction:

To start we got to say that the things that we are going to explain here are very important than you would ever think, it's the core of the reverse engineering process. We will start with the 0x86 architecture which is a little-endian architecture based on the Intel 8086 processor. The little-endian convention is a type of addressing that refers to the order of data stored in memory. In this convention, the least significant bit (or "littlest" end) is first stored at address 0, and subsequent bits are stored incrementally.

Here is a picture that explains bot the little endian and the big endian:



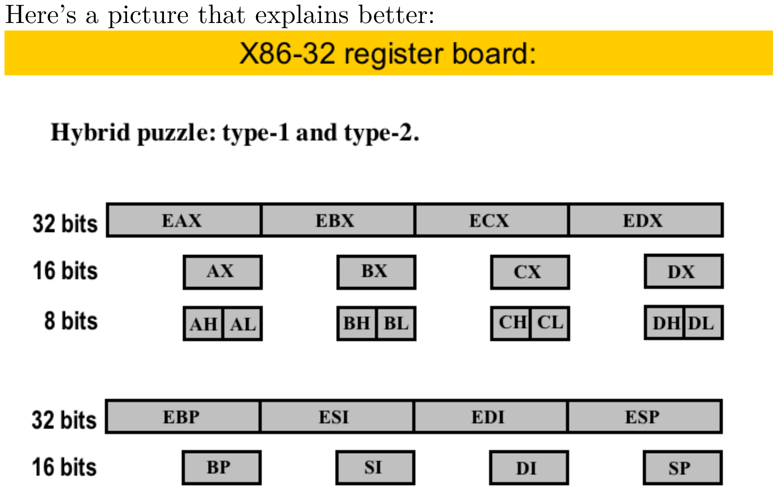
Hmm simply put, the x86 is the 32-bit implementation of the Intel architecture (IA-32) as defined in the Intel Software Development Manual. Generally speaking, It can operate in two modes: real and protected. Real mode is the processor state When it is first powered on and only supports a 16-bit instruction set. Protected The mode is the processor state supporting virtual memory, paging, and other Features. It is the state in which modern operating systems execute. The 64-bit extension of the architecture is called x64 or x86-64. We will dive into the pioneers of each of the architectures and explain them step by step. The main difference between the x86 and the x64 is the size of the registers. In the x64 architecture, the registers are 64-bit in size, which allows the CPU to store more data and access it faster. The register width also determines the amount of memory a computer can utilize. Don't worry if you don't know what registers are, we will explain them further.

II - Register Set and Data Types:

1 - Register Set:

When operating in protected mode, the x86 architecture has eight 32-bit general- purpose registers (GPRs): *EAX*, *EBX*, *ECX*, *EDX*, *EDI*, *ESI*, *EBP*, and *ESP*. Some of them can be further divided into 8- and 16-bit registers. The instruction pointer is stored in the *EIP* register.

Here's a picture that explains better:



Some general-purpose registers and their usage:

Register	Purpose
ECX	Counter in loops
ESI	Source in string/memory operations
EDI	Destination in string/memory operations
EBP	Base frame pointer
ESP	Stack pointer

2 - Data Types:

The common data types are as follows:
Byte: 8-bits. Examples: *AL*, *BH*, *CL*.
Word: 16-bits. Examples: *AX*, *BX*, *CX*.
Double word: 32-bits. Examples: *EAX*, *EBX*, *ECX*.
Quad word: 64-bits. While x86 does not have 64-bit GPRs, it can combine two registers, usually *EDX:EAX*, and treat them as 64-bit values in some sce-narios. For example, the RDTSC instruction writes a 64-bit value to *EDX:EAX*.

III - x86 Instruction Set:

The x86 instruction set allows a high level of flexibility in terms of data movement between registers and memory. The movement can be classified into five general methods:
A - **Immediate to register**
B - **Register to register**
C - **Immediate to memory**
D - **Register to memory and vice versa**
E - **Memory to memory**

1 - Syntax:

Depending on the assembler/disassembler, there are two syntax notations for x86 assembly code, Intel and AT&T:

Intel:

```
mov ecx, 0x11223344
mov ecx, [eax]
mov ecx, eax
```

AT&T:

```
movl $0xAABBCCDD, %ecx
movl (%eax), %ecx
movl %eax, %ecx
```

It is important to note that these are the same instructions but written differ- ently. There are several differences between Intel and AT&T notation, but the most notable ones are as follows:
A - AT&T prefixes the register with **%**, and immediates with **$**. Intel does not do this.
B - AT&T adds a prefix to the instruction to indicate operation width. For example, MOVL (long), MOVB (byte), etc. Intel does not do this.
C - AT&T puts the source operand before the destination. Intel reverses the order.
Personally, I prefer to use the Intel syntax.

2 - Data Movement:

Instructions operate on values that come from registers or main memory. The most common instruction for moving data is *MOV*. The simplest usage is to move a register or immediate to register. For example:

```
mov esi, 0xABCD1200 ; set ESI = 0xABCD1200
mov esi, ecx ; set ESI = ECX
```

The next common usage is to move data to/from memory. Similar to other assembly language conventions, x86 uses square brackets ([]) to indicate memory access. (The only exception to this is the LEA instruction, which uses [] but does not actually reference memory.) Memory access can be specified in several different ways, so we will begin with the simplest case:

```
mov dword ptr [eax], 1 ; set the memory at address EAX to 1
mov ecx, [eax] ; set ECX to the value at address EAX
mov [eax], ebx ; set the memory at address EAX to EBX
mov [esi+34h], eax ; set the memory address at (ESI+34) to EAX
mov eax, [esi+34h] ; set EAX to the value at address (EAX+34)
mov edx, [ecx+eax] ; set EDX to the value at address (ECX+EAX)
```

3 - Arithmetic Operations:

The Fundamental arithmetic operations such as addition, subtraction, multiplication, and division are natively supported by the instruction set. Bit-level operations such as AND, OR, XOR, NOT, and left and right shift also have native corresponding instructions. With the exception of multiplication and division, the remaining instructions are straightforward in terms of usage. These operations are explained with the following examples:

```
add eax, 0x1 ; EAX = EAX + 0x1
sub eax, 0x2 ; EAX = EAX - 0x2
sub ecx, eax ; ECX = ECX - EAX
sub esp, 0Ah ; ESP = ESP - 0xA
inc ecx ; ECX = ECX + 1
dec edi ; EDI = EDI - 1
or eax, 0FFFFFFFFh ; EAX = EAX | 0xFFFFFFFF
and ecx, 8 ; ECX = ECX & 7
xor eax, eax ; EAX = EAX ^ EAX = 0
not edi ; EDI = ~EDI
shl cl, 4 ; CL = CL << 4
shr ecx, 1 ; ECX = ECX >> 1
rol al, 1 ; rotate AL left 3 postions
ror al, 1 ; rotate AL right 1 postions
```

Unsigned and signed multiplication is done through the MUL and IMUL instruc-tions, respectively. The MUL instruction has the following general form: MUL reg/ memory. That is, it can only operate on register or memory values. The register is multiplied with AL, AX, or EAX and the result is stored in *AX*, *DX:AX*, or *EDX:EAX*, depending on the operand width. For example:

```
mul ecx ; EDX:EAX = EAX * ECX
mul dword ptr [esi + 4] ; EDX:EAX = EAX * dword_at(esi + 4)
mul cl ; AX = AX * CL
mul dx ; DX:AX = AX * DX
```

The reason why the result is stored in *EDX:EAX* for 32-bit multiplication is because the result potentially may not fit in one 32-bit register.
Unsigned and signed division is done through the DIV and IDIV instructions, respectively. They take only one parameter (divisor) and have the following form: DIV/IDIV reg/mem. Depending on the divisor's size, DIV will use either *AX*, *DX:AX*, or *EDX:EAX* as the dividend, and the resulting quotient/remainder pair are stored in *AL/AH*, *AX/DX*, or *EAX/EDX*. For example:

```
div ecx ; EDX:EAX / ECX, quotient in EAX and reminder in EDX
div cl ; AX / CL, quotient in AL and reminder in AH
div dword ptr [esi + 24h] ; EDX:EAX / dword_at(esi + 0x24), quotient in EAX
; and the reminder in EDX
```

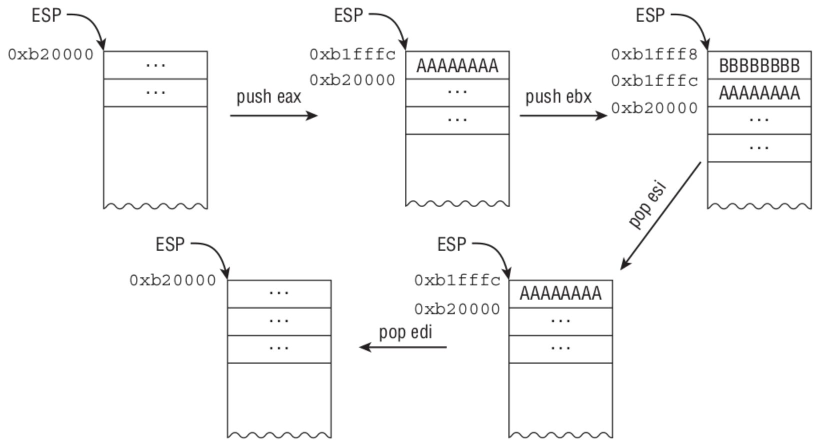
IV - Stack Operations and Function Invocation:

This is an important part of truly understanding the behavior of a binary through the compilation process. To start with, the stack is actually a fundamental data structure in programming languages and operating systems. For example, local variables in C are stored on the functions' stack space. When the operating system transitions from ring 3 to ring 0, it saves state information on the stack. Conceptually, a stack is a last-in-first-out data structure supporting two operations: push and pop. Push means to put something on top of the stack and pop means to remove an item from the top. Concretely speaking, on x86, a stack is a contiguous memory region pointed to by ESP and it grows downwards. Push/pop operations are done through the *PUSH/POP* instructions and they implicitly modify *ESP*. The *PUSH* instruction decrements *ESP* and then writes data at the location pointed to by *ESP*; *POP* reads the data and increments *ESP*. The default auto-increment/decrement value is 4, but it can be changed to 1 or 2 with a prefix override. In practice, the value is almost always 4 because the OS requires the stack to be double-word aligned.

Let's suppose that the *ESP* initially points to 0xb20000 and you have the following code:

```
mov eax,0AAAAAAAAh
mov ebx,0BBBBBBBBh
mov ecx,0CCCCCCCCh
mov edx,0DDDDDDDDh
push eax
; address 0xb1fffc will contain the value 0xAAAAAAAA and ESP
; will be 0xb1fffc (=0xb20000-4)
push ebx
; address 0xb1fff8 will contain the value 0xBBBBBBBB and ESP
; will be 0xb1fff8 (=0xb1fffc-4)
pop esi
; ESI will contain the value 0xBBBBBBBB and ESP will be 0xb1fffc
; (=0xb1fff8+4)
pop edi
; EDI will contain the value 0xAAAAAAAA and ESP will be 0xb20000
; (=0xb1fffc+4)
```

A figure that illustrates the stack layout:



ESP can also be directly modified by other instructions, such as ADD and SUB. While high-level programming languages have the concept of functions that can be called and returned from, the processor does not provide such abstraction. At the lowest level, the processor operates only on concrete objects, such as registers or data coming from memory. How are functions translated at the machine level? They are implemented through the stack data structure! Consider the following function:

C :

```c
int subme(short a, short b) {
    return a-b;
}
```

Assembly :

```
push ebp
mov epb, esp
movsx eax, word ptr [ebp + 8]
movsx ecx, word ptr [ebp + 0ch]
sub eax, ecx
mov esp, ebp
pop ebp
retn
```

The function is invoked with the following code:

C :

```c
sum = subme(x, y);
```

Assembly :

```
push eax
..
push ecx
```

```
4 call subme
5 add esp, 8
```

Before going into the details, first consider the *CALL/RET* instructions and calling conventions. The CALL instruction performs two operations:

1. It pushes the return address (address immediately after the CALL instruction) on the stack.

2. It changes *EIP* to the call destination. This effectively transfers control to the call target and begins execution there.

RET simply pops the address stored on the top of the stack into *EIP* and transfers control to it (literally like a "POP EIP" but such an instruction sequence does not exist on x86). For example, if you want to begin execution at 0x01512321, you can just do the following:

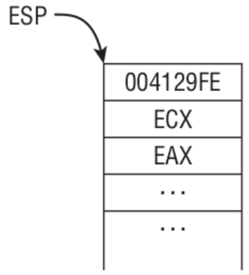
```
push 0x01512321
ret
```

A calling convention is a set of rules dictating how function calls work at the machine level. It is defined by the Application Binary Interface (ABI) for a par- ticular system. For example, should the parameters be passed through the stack, in registers, or both? Should the parameters be passed in from left-to-right or right-to-left? Should the return value be stored on the stack, in registers, or both? There are many calling conventions, but the popular ones are CDECL, STDCALL, THISCALL, and FASTCALL.

A table for calling Conventions:

	CDECL	STDCALL	FASTCALL
Parameters	Pushed on the stack from right-to-left. Caller must clean up the stack after the call.	Same as CDECL except that the callee must clean the stack.	First two parameters are passed in ECX and EDX. The rest are on the stack.
Return value	Stored in EAX.	Stored in EAX.	Stored in EAX.
Non-volatile registers	EBP, ESP, EBX, ESI, EDI.	EBP, ESP, EBX, ESI, EDI.	EBP, ESP, EBX, ESI, EDI.

We now return to the code snippet to discuss how the function subme is invoked. In line 1 and 3, the two parameters are pushed on the stack; ECX and EAX are the first and second parameter, respectively. Line 4 invokes the subme function with the CALL instruction. This immediately pushes the return address, 0x4129FE, on the stack and begins execution at 0x4113A0. This is a figure that illustrates the stack layout:



let me now explain the rest of the program, after line 4 executes, we are now in the subme function body. Line 1 pushes EBP on the stack. Line 2 sets EBP to the current stack pointer. This two-instruction sequence is typically known as the function prologue because it establishes a new function frame. Line 4 reads the value at address EBP+8, which is the first parameter on the stack; line 5 reads the second parameter. Note that the parameters are accessed using EBP as the base register. When used in this context, EBP is known as the base frame pointer (see line 2) because it points to the stack frame for the current function, and parameters/locals can be accessed relative to it. The compiler can also be instructed to generate code that does not use EBP as the base frame pointer through an optimization called frame pointer omission. With such optimization, access to local variables and parameters is done relative to ESP, and EBP can be used as a general register like EAX, EBX, ECX, and so on. Line 6 adds the numbers and saves the result in EAX. Line 8 sets the stack pointer to the base frame pointer. Line 9 pops the saved EBP from line 1 into EBP. This two-instruction sequence is commonly referred to as the function epilogue because it is at the end of the function and restores the previous function frame. At this point, the top of the stack contains the return address saved by the CALL instruction at 0x4129F9. Line 10 performs a RET, which pops the stack and resumes execution at 0x4129FE. Line 5 in the snippet shrinks the stack by 8 because the caller must clean up the stack per CDECL's calling convention. If the function subme had local variables, the code would need to grow the stack by subtracting ESP after line 2. All local variables would then be accessible through a negative offset from EBP.

V - Control Flow:

In this section, we will be describing how the system implements conditional execution for higher-level constructs like if/else, switch/case, and while/for. All of these are implemented through the CMP, TEST, JMP, and Jcc instructions and EFLAGS register. The following list summarizes the common flags in EFLAGS:

1 - **ZF/Zero flag**: Set if the result of the previous arithmetic operation in zero.
2 - **SF/sign flag**: Set to the most significant bit of the result.
3 - **CF/Carry flag**: Set when the result requires a carry. It applies to unsigned numbers.
4 - **OF/Overflow flag**: Set if the result overflows the max size. It applies to signed numbers.

Arithmetic instructions update these flags based on the result. For example, the instruction *SUB EAX, EAX* would cause ZF to be set. The *Jcc* instructions, where "cc" is a conditional code, changes control flow depending on these 17 flags. There can be up to 16 conditional codes, but the most common ones are described in this table:

CONDITIONAL CODE	ENGLISH DESCRIPTION	MACHINE DESCRIPTION
B/NAE	Below/Neither Above nor Equal. Used for unsigned operations.	CF=1
NB/AE	Not Below/Above or Equal. Used for unsigned operations.	CF=0
E/Z	Equal/Zero	ZF=1
NE/NZ	Not Equal/Not Zero	ZF=0
L	Less than/Neither Greater nor Equal. Used for signed operations.	(SF ^ OF) = 1
GE/NL	Greater or Equal/Not Less than. Used for signed operations.	(SF ^ OF) = 0
G/NLE	Greater/Not Less nor Equal. Used for signed operations.	((SF ^ OF) | ZF) = 0

Because assembly language does not have a defined type system, one of the few ways to recognize signed/unsigned types is through these conditional codes. The CMP instruction compares two operands and sets the appropriate condi- tional code in EFLAGS; it compares two numbers by subtracting one from another without updating the result. The TEST instruction does the same thing except it performs a logical AND between the two operands.

If/Else :

If-else constructs are quite simple to recognize because they involve a compare/ test followed by a Jcc. For example:

```
push ebp
push ecx
pop eax
cmp eax, 0x9
jne exit

push 0x2
push 0x3
call subme
mov eax, 0

exit:
    pop ebp
    retn
```

As you can see this piece of code is pushing EBP and ECX into the stack, then it pops the value of ECX into the EAX then it's testing if the EAX = 0, if the result is true than it pushes 2 values into the stack and calls the subme function from the previous section, else it just jumps to the Exit function.

VI - Loops :

At the machine level, loops are implemented using a combination of Jcc and JMP instructions. In other words, they are implemented using if/else and goto constructs. The best way to understand this is to rewrite a loop using only if else and goto. Consider the following example:

```
for (int i=0; i<10; i++) {
    printf("%d\n", i);
}
printf("done!\n");
```

When compiled, both versions are identical at the machine-code level:

```
mov edi, ds:__imp_printf
xor esi, esi
lea ebx, [ebx + 0]
location1:
    push esi
    push offset Format ; "%d\n"
    call edi ; __imp__printf
```

```
8     inc esi
9     add esp, 8
10    cmp esi, 0xA
11    jl location1
12 push offset aDone; "done!\n"
13 call edi; __imp__printf
14 add esp, 4
15
```

Hm, let me explain a little what's happening in there, in line 1 sets *EDI* to the printf function. Line 2 sets *ESI* to 0. Line 4 begins the loop; however, note that it does not begin with a comparison. There is no comparison here because the compiler knows that the counter was initialized to 0 (see line 2) and is obviously going to be less than 10 so it skips the check. Lines 5–7 call the printf function with the right parameters (format specifier and our number). Line 8 increments the number. Line 9 cleans up the stack because printf uses the CDECL calling convention. Line 10 checks to see if the counter is less than 0xA. If it is, it jumps back to location1. If the counter is not less than 0xA, it continues execution at line 12 and finishes with a printf.

VII - Challenges :

1 - First Challenge :

Can you retrieve the flag out of that x64 ASM code?

```
SECTION .data
array db 0x55 ,0x5a ,0x5b ,0x1a8 ,0x1aa ,0x52 ,0x54 ,0x5c ,0x1ac ,0x1ac
,0x19b ,0x1a2 ,0x53 ,0x1ab ,0x51 ,0x1b2 ,0x55 , 0x18f ,0x1b5 ,0x18b ,0x18f
,0x189 ,0x18b ,0x1a6 ,0x183 ,0x182 , 0x19e ,0x18e ,0x185 ,0x19c ,0x199
,0x193 ,0x199 ,0x46 ,0x1e1

SECTION .text

global main

main:
    xor rax, rax
    xor rdi, rdi
    mov rdx, 0x32
    sub rsp, 0x32
    mov rsp, rsi
    syscall

    mov r10, 0
l:
    movzx r11, byte [rsp + r10]
    movzx r12, byte [array + r10]
    add r11, r10
    add r11, 0xAB
    xor r11, 0xAB
    and r11, 0xff
    cmp r11, r12
    jne b
```

```

    add r10, 1
    cmp r10, 0x23
    jne l

    mov rax, 0x3c
    mov rdi, 0
    syscall

b:
    mov rax, 0x3c
    mov rdi, 1
    syscall
```

Solver :

```python
#! /usr/bin/python3

# Author => IronByte
# Flag = SECURINETS{SAHA_Chabeb_Keep_going!}

T = [0x55 ,0x5a ,0x5b ,0x1a8 ,0x1aa ,0x52 ,0x54 ,0x5c ,
0x1ac ,0x1ac ,0x19b ,0x1a2 ,0x53 ,0x1ab ,0x51 ,0x1b2 ,
0x55 ,0x18f ,0x1b5 ,0x18b ,0x18f ,0x189 ,0x18b ,0x1a6 ,
0x183 ,0x182 ,0x19e ,0x18e ,0x185 ,0x19c ,0x199 ,0x193 ,
0x199 ,0x46 ,0x1e1]

def solver():
        ans = []
        i = 0
        for byt in T:
                elt = byt ^ 0xAB
                elt = elt - 0xAB
                elt = elt - i
                i = i + 1
                ans.append(elt)
        return ans

print("".join(chr(elt) for elt in solver()))
```

2 - Second Challenge :

Download the binary from this link Click Me !

Let's first start by running the binary:

```
ironbyte@Med-Ali-Ouachani:~/challs$ ./bin
Username :mike
Password : 123
```

```
It's either bad username or bad password
```

As you can in this task we have to find out the username and the password. Let's start debugging the binary using **GDB**.

Let's fire GDB and debug that binary :

```
ironbyte@Med-Ali-Ouachani:~/challs$  gdb bin
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bin...
(No debugging symbols found in bin)

```

Now step by step i will go and search for the main function and try to disassemble it.

```
gdb) i functions
All defined functions:

Non-debugging symbols:
0x00001000  _init
0x00001040  __libc_start_main@plt
0x00001050  printf@plt
0x00001060  __stack_chk_fail@plt
0x00001070  puts@plt
0x00001080  __isoc99_scanf@plt
0x00001090  __cxa_finalize@plt
0x000010a0  _start
0x000010d0  __x86.get_pc_thunk.bx
0x000010e0  deregister_tm_clones
0x00001120  register_tm_clones
0x00001170  __do_global_dtors_aux
0x000011c0  frame_dummy
0x000011c9  __x86.get_pc_thunk.dx
0x000011cd  main
0x00001280  __stack_chk_fail_local
0x00001298  _fini
```

```
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
   0x000011dd <+0>:     lea    ecx,[esp+0x4]
   0x000011e1 <+4>:     and    esp,0xfffffff0
   0x000011e4 <+7>:     push   DWORD PTR [ecx-0x4]
   0x000011e7 <+10>:    push   ebp
   0x000011e8 <+11>:    mov    ebp,esp
   0x000011ea <+13>:    push   ebx
   0x000011eb <+14>:    push   ecx
   0x000011ec <+15>:    sub    esp,0x30
   0x000011ef <+18>:    call   0x10e0 <__x86.get_pc_thunk.bx>
   0x000011f4 <+23>:    add    ebx,0x2dd4
   0x000011fa <+29>:    mov    eax,ecx
   0x000011fc <+31>:    mov    eax,DWORD PTR [eax+0x4]
   0x000011ff <+34>:    mov    DWORD PTR [ebp-0x2c],eax
   0x00001202 <+37>:    mov    eax,gs:0x14
   0x00001208 <+43>:    mov    DWORD PTR [ebp-0xc],eax
   0x0000120b <+46>:    xor    eax,eax
   0x0000120d <+48>:    sub    esp,0xc
   0x00001210 <+51>:    lea    eax,[ebx-0x1fc0]
   0x00001216 <+57>:    push   eax
   0x00001217 <+58>:    call   0x1060 <printf@plt>
   0x0000121c <+63>:    add    esp,0x10
   0x0000121f <+66>:    sub    esp,0x8
   0x00001222 <+69>:    lea    eax,[ebp-0x20]
   0x00001225 <+72>:    push   eax
   0x00001226 <+73>:    lea    eax,[ebx-0x1fb5]
   0x0000122c <+79>:    push   eax
   0x0000122d <+80>:    call   0x1090 <__isoc99_scanf@plt>
   0x00001232 <+85>:    add    esp,0x10
   0x00001235 <+88>:    sub    esp,0xc
   0x00001238 <+91>:    lea    eax,[ebx-0x1fb2]
   0x0000123e <+97>:    push   eax
   0x0000123f <+98>:    call   0x1060 <printf@plt>
   0x00001244 <+103>:   add    esp,0x10
   0x00001247 <+106>:   sub    esp,0x8
   0x0000124a <+109>:   lea    eax,[ebp-0x24]
   0x0000124d <+112>:   push   eax
   0x0000124e <+113>:   lea    eax,[ebx-0x1fa6]
   0x00001254 <+119>:   push   eax
   0x00001255 <+120>:   call   0x1090 <__isoc99_scanf@plt>
   0x0000125a <+125>:   add    esp,0x10
   0x0000125d <+128>:   mov    eax,DWORD PTR [ebp-0x24]
   0x00001260 <+131>:   cmp    eax,0x7e7
   0x00001265 <+136>:   jne    0x1295 <main+184>
   0x00001267 <+138>:   sub    esp,0x8
   0x0000126a <+141>:   lea    eax,[ebx-0x1fa3]
   0x00001270 <+147>:   push   eax
```

```
    0x00001271 <+148>:         lea    eax,[ebp-0x20]
    0x00001274 <+151>:         push   eax
    0x00001275 <+152>:         call   0x1040 <strcmp@plt>
    0x0000127a <+157>:         add    esp,0x10
    0x0000127d <+160>:         test   eax,eax
    0x0000127f <+162>:         jne    0x1295 <main+184>
    0x00001281 <+164>:         sub    esp,0xc
    0x00001284 <+167>:         lea    eax,[ebx-0x1f9d]
    0x0000128a <+173>:         push   eax
    0x0000128b <+174>:         call   0x1080 <puts@plt>
    0x00001290 <+179>:         add    esp,0x10
    0x00001293 <+182>:         jmp    0x12a7 <main+202>
    0x00001295 <+184>:         sub    esp,0xc
    0x00001298 <+187>:         lea    eax,[ebx-0x1f8c]
    0x0000129e <+193>:         push   eax
    0x0000129f <+194>:         call   0x1080 <puts@plt>
    0x000012a4 <+199>:         add    esp,0x10
    0x000012a7 <+202>:         mov    eax,0x0
    0x000012ac <+207>:         mov    edx,DWORD PTR [ebp-0xc]
    0x000012af <+210>:         sub    edx,DWORD PTR gs:0x14
    0x000012b6 <+217>:         je     0x12bd <main+224>
    0x000012b8 <+219>:         call   0x12d0 <__stack_chk_fail_local>
    0x000012bd <+224>:         lea    esp,[ebp-0x8]
    0x000012c0 <+227>:         pop    ecx
    0x000012c1 <+228>:         pop    ebx
    0x000012c2 <+229>:         pop    ebp
    0x000012c3 <+230>:         lea    esp,[ecx-0x4]
    0x000012c6 <+233>:         ret
End of assembler dump.
```

the interesting part is:

```
    0x5655625d <+128>:         mov    eax,DWORD PTR [ebp-0x24]
    0x56556260 <+131>:         cmp    eax,0x7e7
    0x56556265 <+136>:         jne    0x56556295 <main+184>
    0x56556267 <+138>:         sub    esp,0x8
    0x5655626a <+141>:         lea    eax,[ebx-0x1fa3]
    0x56556270 <+147>:         push   eax
    0x56556271 <+148>:         lea    eax,[ebp-0x20]
    0x56556274 <+151>:         push   eax
    0x56556275 <+152>:         call   0x56556040 <strcmp@plt>
    0x5655627a <+157>:         add    esp,0x10
    0x5655627d <+160>:         test   eax,eax
```

The [ebp -0x24] refers to a variable in the code might be the password, since he checking if the EAX is compared with the 0x727 = 2023. The other variable is [ebp - 0x20] which is the username. As you can see the username is pushed to the stack with another variable at the offset = ebx - 0x1fa3 which might be the Username that he is comparing with.

Let's try to put a break point there and find out what is the value at the offset = ebx - 0x1fa3.

```
Breakpoint 2, 0x56556270 in main ()
```

```
(gdb) i r
eax            0x56557025          1448439845
ecx            0xffffcf34          -12492
edx            0x0                 0
ebx            0x56558fc8          1448447944
esp            0xffffcf38          0xffffcf38
ebp            0xffffcf78          0xffffcf78
esi            0xffffd044          -12220
edi            0xf7ffcb80          -134231168
eip            0x56556270          0x56556270 <main+147>
eflags         0x292               [ AF SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb) x/s eax
No symbol table is loaded.  Use the "file" command.
(gdb) x/s $eax
0x56557025:        "Samir"
```

So the username was "Samir" and the password was "2023". Let's try to loging using those creds.

```
ironbyte@Med-Ali-Ouachani:~/challs$ ./bin
Username :Samir
Password : 2023
Congratulation !
```

Congratulation you have cracked your first program!
Here's a a link to a **GDB** CheatSheet Click ME !.