0x44.cc

Reversing for dummies · 0x44.cc

24-31 minutes

Context

Before I got into reverse engineering, executables always seemed like black magic to me. I always wondered how stuff worked under the hood, and how binary code is represented inside .exe files, and how hard it is to modify this 'compiled code' without access to the original source code.

But one of the main intimidating hurdles always seemed to be the assembly language, it's the thing that scares most people away from trying to learn about this field.

That's the main reason why I thought of writing this straight-to-the-point article that only contains the essential stuff that you encounter the most when reversing, albeit missing crucial details for the sake of brevity, and assumes the reader has a reflex of finding answers online, looking up definitions, and more importantly, coming up with examples/ideas/projects to practice on.

The goal is to hopefully guide an aspiring reverse engineer and arouse motivation towards learning more about this seemingly elusive passion.

Note: This article assumes the reader has elementary knowledge regarding the hexadecimal numeral system, as well as the C programming language, and is based on a 32-bit Windows executable case study - results might differ across different OSes/architectures.

Introduction

Compilation

After writing code using a compiled language, a compilation takes place ~~(duh)~~, in order to generate the output binary file (an example of such is an .exe file).

code.c
Compilation
program.exe

Compilers are sophisticated programs which do this task. They make sure the syntax of your ~~ugly~~ code is correct, before compiling and optimizing the resulting machine code by minimizing its size and improving its performance, whenever applicable.

Binary code

As we were saying, the resulting output file contains binary code, which can only be 'understood' by a CPU, it's essentially a succession of varying-length instructions to be executed in order - here's what some of them look like:

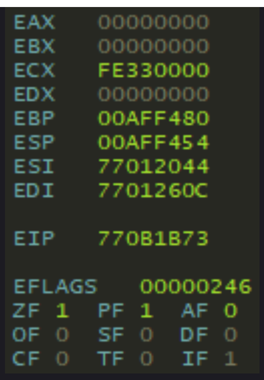
CPU-readable instruction data (in hex)	Human-readable interpretation
55	push ebp
8B EC	mov ebp, esp
83 EC 08	sub esp, 8
33 C5	xor eax, ebp
83 7D 0C 01	cmp dword ptr [ebp+0Ch], 1

These instructions are predominantly arithmetical, and they manipulate CPU registers/flags as well as volatile memory, as they're executed.

CPU registers

A CPU register is almost like a temporary integer variable - there's a small fixed number of them, and they exist because they're quick to access, unlike memory-based variables, and they help the CPU keep track of its data (results, operands, counts, etc.) during execution.

It's important to note the presence of a special register called the `FLAGS` register (`EFLAGS` on 32-bit), which houses a bunch of flags (boolean indicators), which hold information about the state of the CPU, which include details about the last arithmetic operation (zero: `ZF`, overflow: `OF`, parity: `PF`, sign: `SF`, etc.).


```
EAX    00000000
EBX    00000000
ECX    FE330000
EDX    00000000
EBP    00AFF480
ESP    00AFF454
ESI    77012044
EDI    7701260C

EIP    770B1B73

EFLAGS    00000246
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1
```
CPU registers visualized while debugging a 32-bit process on x64dbg, a debugging tool.

Some of these registers can also be spotted on the assembly excerpt mentioned previously, namely: `EAX`, `ESP` (stack pointer) and `EBP` (base pointer).

Memory access

As the CPU executes stuff, it needs to access and interact with memory, that's when the role of the *stack* and the *heap* comes.

These are (without getting into too much detail) the 2 main ways of 'keeping track of variable data' during the execution of a program:

🥞 ***Stack***

The simpler and faster of the two - it's a linear contiguous LIFO (last in = first out) data structure with a push/pop mechanism, it serves to remember function-scoped variables, arguments, and keeps track of calls (ever heard of a stack trace?)

⛰ ***Heap***

The heap, however, is pretty unordered, and is for more complicated data structures, it's typically used for dynamic allocations, where the size of the buffer isn't initially known, and/or if it's too big, and/or needs to be modified later.

Assembly instructions

As I've mentioned earlier, assembly instructions have a varying 'byte-size', and a varying number of arguments.

Arguments can also be either immediate ('hardcoded'), or they can be registers, depending on the instruction:

```
55         push    ebp     ; size: 1 byte,  argument: register
6A 01      push    1       ; size: 2 bytes, argument: immediate
```

Let's quickly run through a very small set of some of the common ones we'll get to see - feel free to do your own research for more detail:

Stack operations

- **push `value`** *; pushes a value into the stack (decrements `ESP` by 4, the size of one stack 'unit').*
- **pop `register`** *; pops a value to a register (increments `ESP` by 4).*

Data transfer

- **mov `destination`, `source`** ; *~~moves~~ copies a value from/to a register.*
- **mov `destination`, `[expression]`** ; *copies a value from a memory address resolved from a 'register expression' (single register or arithmetic expression involving one or more registers) into a register.*

Flow control

- **jmp `destination`** ; *jumps into a code location (sets `EIP` (instruction pointer)).*
- **jz/je `destination`** ; *jumps into a code location if `ZF` (the zero flag) is set.*
- **jnz/jne `destination`** ; *jumps into a code location if `ZF` is not set.*

Operations

- **cmp `operand1`, `operand2`** ; *compares the 2 operands and sets `ZF` if they're equal.*
- **add `operand1`, `operand2`** ; *operand1 += operand2;*
- **sub `operand1`, `operand2`** ; *operand1 -= operand2;*

Function transitions

- **call `function`** ; *calls a function (pushes current `EIP`, then jumps to the function).*
- **retn** ; *returns to caller function (pops back the previous `EIP`).*

Note: You might notice the words 'equal' and 'zero' being used interchangeably in x86 terminology - that's because comparison instructions internally perform a subtraction, which means if the 2 operands are equal, `ZF` is set.

Assembly patterns

Now that we have a rough idea of the main elements used during the execution of a program, let's get familiarized with the patterns of instructions that you can encounter reverse engineering your average everyday 32-bit PE binary.

Function prologue

A function prologue is some initial code embedded in the beginning of most functions, it serves to set up a new stack frame for said function.

It typically looks like this (X being a number):

```
55          push    ebp       ; preserve caller function's base pointer in stack
8B EC       mov     ebp, esp   ; caller function's stack pointer becomes base pointer (new stack
frame)
83 EC XX    sub     esp, X     ; adjust the stack pointer by X bytes to reserve space for local
variables
```

Function epilogue

The epilogue is simply the opposite of the prologue - it undoes its steps to restore the stack frame of the caller function, before it returns to it:

```
8B E5    mov    esp, ebp    ; restore caller function's stack pointer (current base pointer)
5D       pop    ebp         ; restore base pointer from the stack
C3       retn               ; return to caller function
```

Now at this point, you might be wondering - how do functions talk to each other? How exactly do you send/access arguments when calling a function, and how do you receive the return value? That's precisely why we have calling conventions.

Calling conventions: __cdecl

A calling convention is basically a protocol used to communicate with functions, there's a few variations of them, but they share the same principle.

We will be looking at the __cdecl (C declaration) convention, which is the standard one when compiling C code.

In __cdecl (32-bit), function arguments are passed on the stack (pushed in reverse order), while the return value is returned in the `EAX` register (assuming it's not a float).

This means that a `func(1, 2, 3);` call will generate the following:

```
6A 03            push    3
6A 02            push    2
6A 01            push    1
E8 XX XX XX XX    call    func
```

Putting everything together

Assuming `func()` simply does an addition on the arguments and returns the result, it would probably look like this:

```
int __cdecl func(int, int, int):

           prologue:
55           push    ebp              ; save base pointer
8B EC        mov     ebp, esp         ; new stack frame

           body:
8B 45 08     mov     eax, [ebp+8]      ; load first argument to EAX (return value)
03 45 0C     add     eax, [ebp+0Ch]    ; add 2nd argument
03 45 10     add     eax, [ebp+10h]    ; add 3rd argument

           epilogue:
5D           pop     ebp              ; restore base pointer
C3           retn                     ; return to caller
```

Now if you've been paying attention and you're still confused, you might be asking yourself one of these 2 questions:

1) Why do we have to adjust `EBP` by 8 to get to the first argument?

- If you check the definition of the `call` instruction we mentioned earlier, you'll realize that, internally, it actually pushes `EIP` to the stack. And if you also check the definition for `push`, you'll realize that it decrements `ESP` (which is copied to `EBP` after the prologue) by 4 bytes. In addition, the prologue's first instruction is also a `push`, so we end up with 2 decrements of 4, hence the need to add 8.

2) What happened to the prologue and epilogue, why are they seemingly 'truncated'?

- It's simply because we haven't had a use for the stack during the execution of our function - if you've noticed, we haven't modified `ESP` at all, which means we also don't need to restore it.

If conditions

To demo the flow control assembly instructions, I'd like to add one more example to show how an if condition was compiled to assembly.

Assume we have the following function:

```
void print_equal(int a, int b) {
    if (a == b) {
        printf("equal");
    }
    else {
        printf("nah");
    }
}
```

After compiling it, here's the disassembly that I got with the help of IDA:

```
void __cdecl print_equal(int, int):

     10000000   55                push   ebp
     10000001   8B EC             mov    ebp, esp
     10000003   8B 45 08          mov    eax, [ebp+8]       ; load 1st argument
     10000006   3B 45 0C          cmp    eax, [ebp+0Ch]     ; compare it with 2nd
  ┌┈ 10000009   75 0F             jnz    short loc_1000001A ; jump if not equal
  ┊  1000000B   68 94 67 00 10    push   offset aEqual  ; "equal"
  ┊  10000010   E8 DB F8 FF FF    call   _printf
  ┊  10000015   83 C4 04          add    esp, 4
┌─┊─ 10000018   EB 0D             jmp    short loc_10000027
│ ┊
│ └ loc_1000001A:
│    1000001A   68 9C 67 00 10    push   offset aNah    ; "nah"
│    1000001F   E8 CC F8 FF FF    call   _printf
│    10000024   83 C4 04          add    esp, 4
│
└── loc_10000027:
     10000027   5D                pop    ebp
     10000028   C3                retn
```

Give yourself a minute and try to make sense of this disassembly output (for simplicity's sake, I've changed the real addresses and made the function start from `10000000` instead).

In case you're wondering about the `add esp, 4` part, it's simply there to adjust `ESP` back to its initial value (same effect as a `pop`, except without modifying any register), since we had to `push` the printf string argument.

Basic data structures

Now let's move on and talk about how data is stored (integers and strings especially).

Endianness

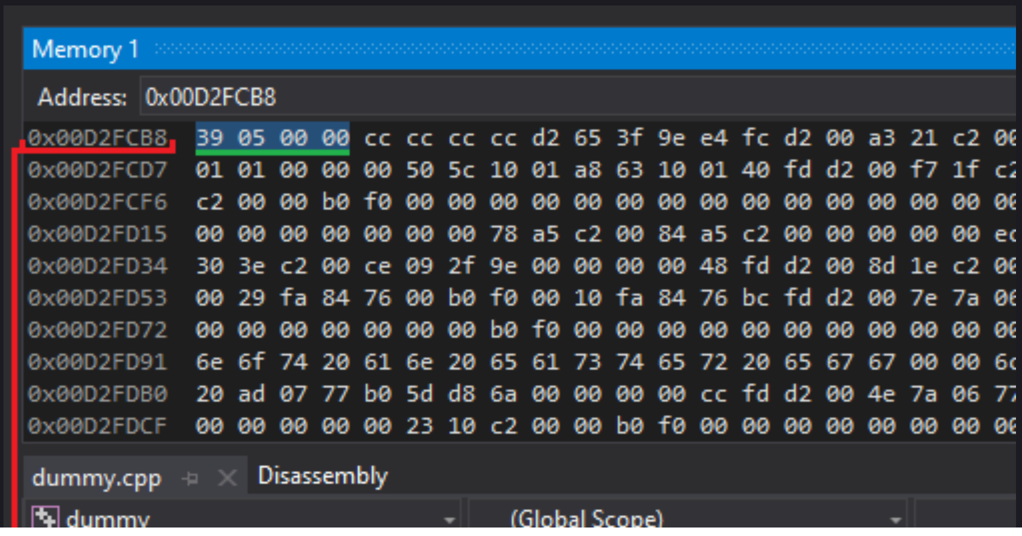
Endianness is the order of the sequence of bytes representing a value in computer memory.

There's 2 types - big-endian and little-endian:

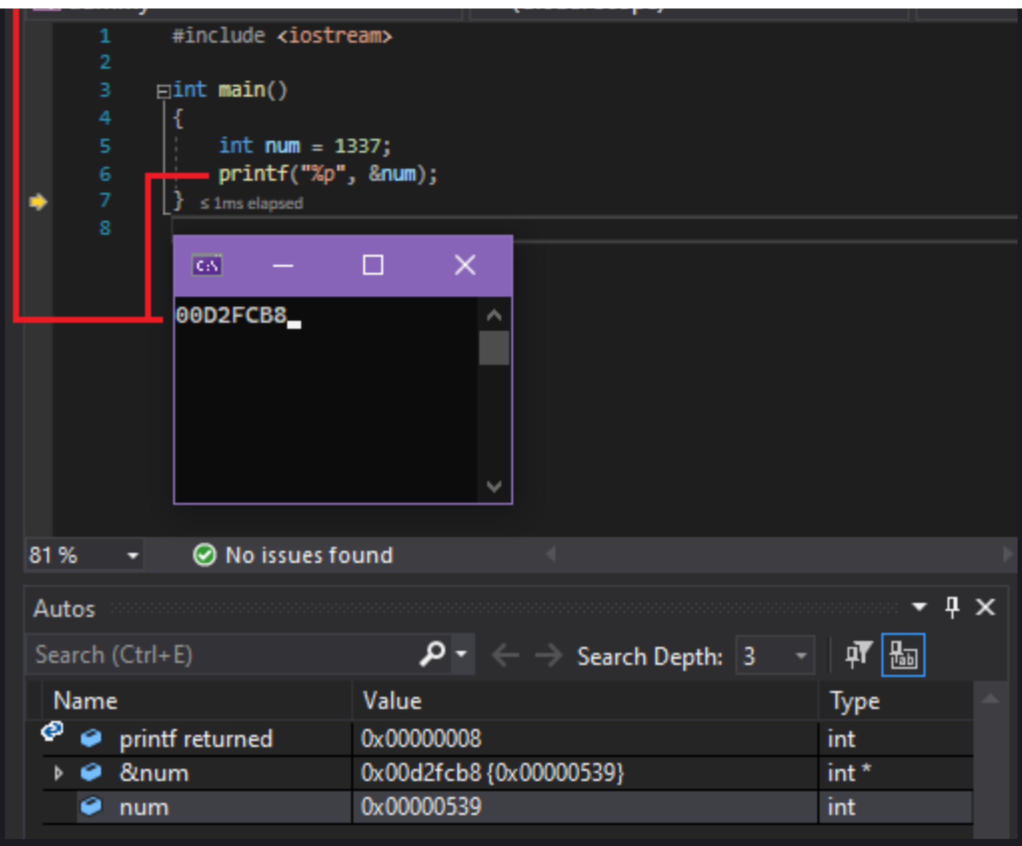
For reference, x86 family processors (the ones on pretty much any computer you can find) always use little-endian.

To give you a live example of this concept, I've compiled a Visual Studio C++ console app, where I declared an `int` variable with the value `1337` assigned to it, then I printed the variable's address using `printf()`, on the main function.

Then I ran the program attached to the debugger in order to check the printed variable's address on the memory hex view, and here's the result I obtained:



```
0x00D2FCB8  39 05 00 00 cc cc cc cc d2 65 3f 9e e4 fc d2 00 a3 21 c2 00
0x00D2FCD7  01 01 00 00 00 50 5c 10 01 a8 63 10 01 40 fd d2 00 f7 1f c2
0x00D2FCF6  c2 00 00 b0 f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00D2FD15  00 00 00 00 00 00 00 78 a5 c2 00 84 a5 c2 00 00 00 00 00 ec
0x00D2FD34  30 3e c2 00 ce 09 2f 9e 00 00 00 00 48 fd d2 00 8d 1e c2 00
0x00D2FD53  00 29 fa 84 76 00 b0 f0 00 10 fa 84 76 bc fd d2 00 7e 7a 06
0x00D2FD72  00 00 00 00 00 00 00 b0 f0 00 00 00 00 00 00 00 00 00 00 00
0x00D2FD91  6e 6f 74 20 61 6e 20 65 61 73 74 65 72 20 65 67 67 00 00 6c
0x00D2FDB0  20 ad 07 77 b0 5d d8 6a 00 00 00 00 cc fd d2 00 4e 7a 06 77
0x00D2FDCF  00 00 00 00 00 23 10 c2 00 00 b0 f0 00 00 00 00 00 00 00 00
```



```
#include <iostream>

int main()
{
    int num = 1337;
    printf("%p", &num);
}
```

```
00D2FCB8
```

Name	Value	Type
printf returned	0x00000008	int
&num	0x00d2fcb8 {0x00000539}	int *
num	0x00000539	int

To elaborate more on this - `int` variables are 4 bytes long (32 bits) (in case you didn't know), so this means that if the variable starts from the address `D2FCB8` it would end right before `D2FCBC` (+4).

To go from human readable value to memory bytes, follow these steps:

decimal: `1337` -> hex: `539` -> bytes: `00 00 05 39` -> little-endian: `39 05 00 00`

Signed integers

This part is interesting yet relatively simple. What you should know here is that integer signing (positive/negative) is typically done on computers with the help of a concept called two's complement.

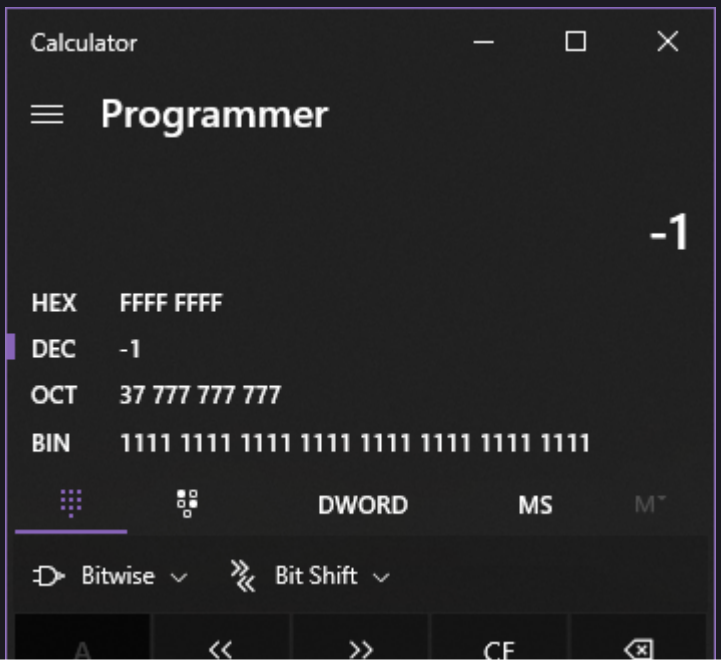
The gist of it is that the lowest/first half of an integer is reserved for positive numbers, while the highest/last half is for negative numbers, here's what this looks like in hex, for a 32-bit signed int (highlighted = hex, in parenthesis = decimal):

Positives (1/2): `00000000` (0) -> `7FFFFFFF` (2,147,483,647 or `INT_MAX`)

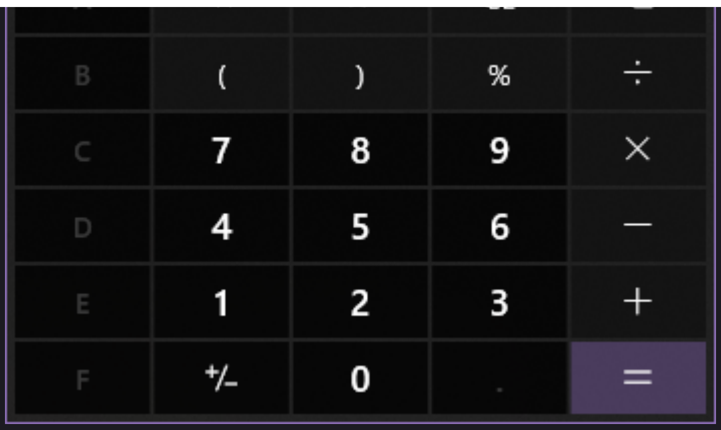
Negatives (2/2): `80000000` (-2,147,483,648 or `INT_MIN`) -> `FFFFFFFF` (-1)

If you've noticed, we're always *ascending* in value. Whether we go up in hex or decimal. And that's the crucial point of this concept - arithmetical operation do not have to do anything special to handle signing, they can simply treat all values as unsigned/positive, and the result would still be interpreted correctly (as long as we don't go beyond `INT_MAX` or `INT_MIN`), and that's because integers will also *'rollover'* on overflow/underflow by design, kinda like an analog odometer.

Protip: The Windows calculator is a very helpful tool - you can set it to programmer mode and set the size to DWORD (4 bytes), then enter negative decimal values and visualize them in hex and binary, and have fun performing operations on them.



```
-1
HEX  FFFF FFFF
DEC  -1
OCT  37 777 777 777
BIN  1111 1111 1111 1111 1111 1111 1111 1111
```

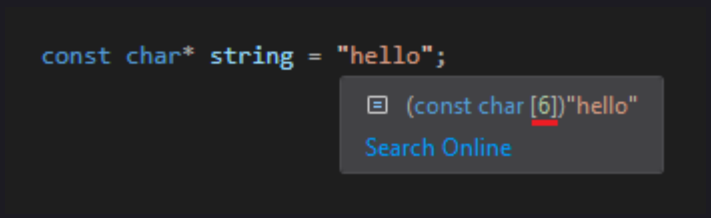


Strings

In C, strings are stored as `char` arrays, therefore, there's nothing special to note here, except for something called null termination.

If you ever wondered how `strlen()` is able to know the size of a string, it's very simple - strings have a character that indicates their end, and that's the null byte/character - `00` or `'\0'`.

If you declare a string constant in C code, and hover over it in Visual Studio, for instance, it will tell you the size of the generated array, and as you can see, for this reason, it's one element more than the 'visible' string size.



```
const char* string = "hello";
```

(const char [6])"hello"
Search Online

Note: The endianness concept is not applicable on arrays, only on single variables. Therefore, the order of characters in memory would be normal here - low to high.

Making sense of `call` and `jmp` instructions

Now that you know all of this, you're likely able to start making sense of some machine code, and emulate a CPU with your brain, to some extent, so to speak.

Let's take the [`print_equal()` example], but let's only focus on the `printf()` `call` instructions this time.

```
void print_equal(int, int):
...
     10000010   E8 DB F8 FF FF    call   _printf
...
     1000001F   E8 CC F8 FF FF    call   _printf
```

You might be wondering to yourself - wait a second, if these are the same instructions, then why are their bytes different?

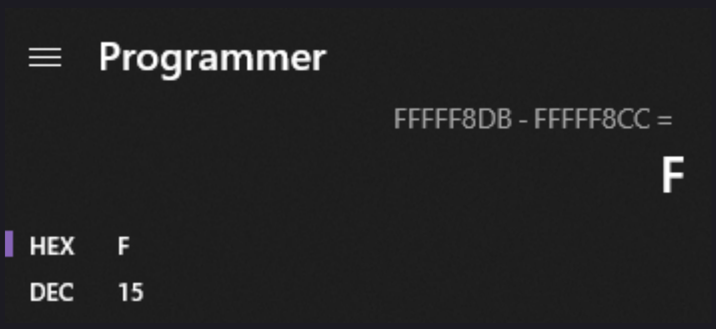
That's because, `call` (and `jmp`) instructions (usually) take an *offset* (relative address) as an argument, not an absolute address.

An offset is basically the difference between the current location, and the destination, which also means that it can be either negative or positive.

As you can see, the [opcode] of a `call` instruction that takes a 32-bit offset, is `E8`, and is followed by said offset - which makes the full instruction: `E8 XX XX XX XX`.

Pull out your calculator, ~~why'd you close it so early?!~~ and calculate the difference between the offset of both instructions (don't forget the endianness).

You'll notice that (the absolute value of) this difference is the same as the one between the instruction addresses (`1000001F` - `10000010` = `F`):



Programmer
FFFFF8DB - FFFFF8CC =
F
HEX F
DEC 15

Another small detail that we should add, is the fact that the CPU only executes an instruction after fully 'reading' it, which means that by the time the CPU starts 'executing', `EIP` (the instruction pointer) is already pointing at the *next* instruction to be executed.

That's why these offsets are actually accounting for this behaviour, which means that in order to get the *real* address of the target function, we have to also *add* the size of the `call` instruction: 5.

Now let's apply all these steps in order to resolve `printf()`'s address from the first instruction on the example:

```
10000010   E8 DB F8 FF FF    call   _printf
```

1) Extract the offset from the instruction: `E8 (DB F8 FF FF)` -> `FFFFF8DB` (-1829)

2) Add it to the instruction address: `10000010` + `FFFFF8DB` = `0FFFF8EB`

3) And finally, add the instruction size: `0FFFF8EB` + 5 = `0FFFF8F0` (`&printf`)

The exact same principle applies to the `jmp` instruction:

```
...
┌─── 10000018   EB 0D             jmp    short loc_10000027
...
└── loc_10000027:
     10000027   5D                pop    ebp
...
```

The only difference in this example is that `EB XX` is a short version `jmp` instruction - which means it only takes an 8-bit (1 byte) offset.

Therefore: `10000018` + `0D` + 2 = `10000027`

Conclusion

That's it! You should now have enough information (and hopefully, motivation) to start your journey reverse engineering executables.

Start by writing dummy C code, compiling it, and debugging it while single-stepping through the disassembly instructions (Visual Studio allows you to do this, by the way).

Compiler Explorer is also an extremely helpful website which compiles C code to assembly for you in real time using multiple compilers (select the `x86 msvc` compiler for Windows 32-bit).

After that, you can try your luck with closed-source native binaries, by the help of disassemblers such as Ghidra and IDA, and debuggers such as x64dbg.

Note: If you've noticed inaccurate information, or room for improvement regarding this article, and would like to improve it, feel free to submit a pull request on GitHub.

Thanks for reading!

(edited)