

💡 FIRST PRINCIPLES: WHAT IS ADDRESSING, REALLY?

Think of the CPU like a blind beast that only knows how to poke memory by **number** — it doesn't "see" variable names, doesn't care about objects or data types — it just sees **addresses**.

So:

- 💡 **Addressing** = Telling the CPU *where* to look (or write) in its giant memory book.
- 🧠 **Address Space** = The full size of that book — how many unique "pages" (addresses) it can have. The range of memory addresses available to a process or system.

⌚ Byte-Addressability — Why It Matters

x86 is byte-addressable.

This means that the smallest unit of memory that can be uniquely addressed is a single byte (8 bits). While you might load or store larger units (words, doublewords, quadwords), the addresses always refer to the starting byte of that unit. So this part means I can only have 1 address per byte? Like 0xffffe say its pointing to some data in memory and an address can only be attached to a single byte or multiple bytes not less than a byte eg you can't give an address to 7 bits?

👉 **Each unique memory address points to exactly one byte.**

👉 **Not a bit. Not 5, 6, 7 or 9 bits. Just 1 full byte = 8 bits.**

You "can only have 1 address per byte? like 0xffffe pointing to some data..."

Yes! 0xffffe is the address of **a single byte** in memory.

This means that the smallest unit of memory that can be uniquely addressed is a single byte (8 bits).

While you might load or store larger units (words, doublewords, quadwords), the addresses always refer to the starting byte of that unit.

Even if that byte is *part* of a bigger thing (like a 4-byte integer), the CPU still treats addresses as pointing to **bytes only** — and you just use consecutive addresses to grab the rest:

- **32-bit value?** → uses 4 bytes → starts at 0xffffe, continues to 0xfffff, 0xffff0, 0xffff1
- **64-bit value?** → uses 8 bytes → starts at one byte address, spans 8 bytes. But each of those 8 bytes still has **its own unique address** like the 32-bit version shown above.

You can't address individual bits (like "give me bit #3 in byte at address 0xffffe").

If you need to access a bit, you do:

```
byte = memory[0xffffe];
bit3 = (byte >> 3) & 1;
```

See? Still fetched the whole byte — then *you* manually isolate the bit.

💡 **Analogy Time (Gen Z style):**

📦 Imagine memory is like a line of lockers.

- Every locker = 1 byte
- Locker number = memory address
- You can open locker #101 (0x65) and get what's inside — one byte
- Want a 64-bit doubleword? You just open 8 lockers in a row starting from some number.

BUT...

✗ You **can't** label "**the left sock inside locker #101**" as its own address

✓ You just open locker #101 and **dig into its contents** if you need something smaller.

⚠ Fun fact:

Some really old or embedded systems are **bit-addressable**, but **x86, ARM, RISC-V, etc — all byte-addressable**.

So, you're absolutely right to say:

An address can only refer to a byte — nothing smaller.

No "bit-addressing" unless you're working with custom hardware or ancient microcontrollers.

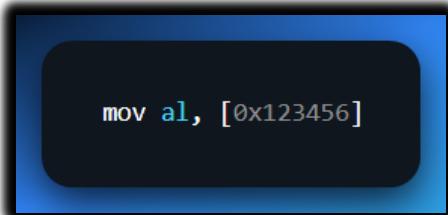
Byte-addressable sounds like a boring fact until you realize...

- You can modify **one single character in a password string**.
- You can NOP out a **single instruction** in shellcode.
- You can corrupt just the **last byte** of a return address, and change program flow.

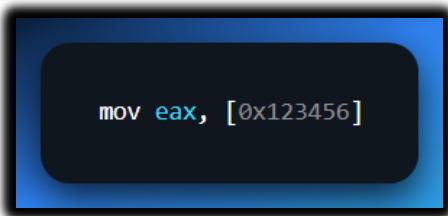
Byte-level control = surgical power.

In older architectures (word-addressable), you couldn't just poke one byte. But x86? You've got scalpel-level access. So:

Grab 1 byte from memory.



Grab 4 bytes starting from that address.



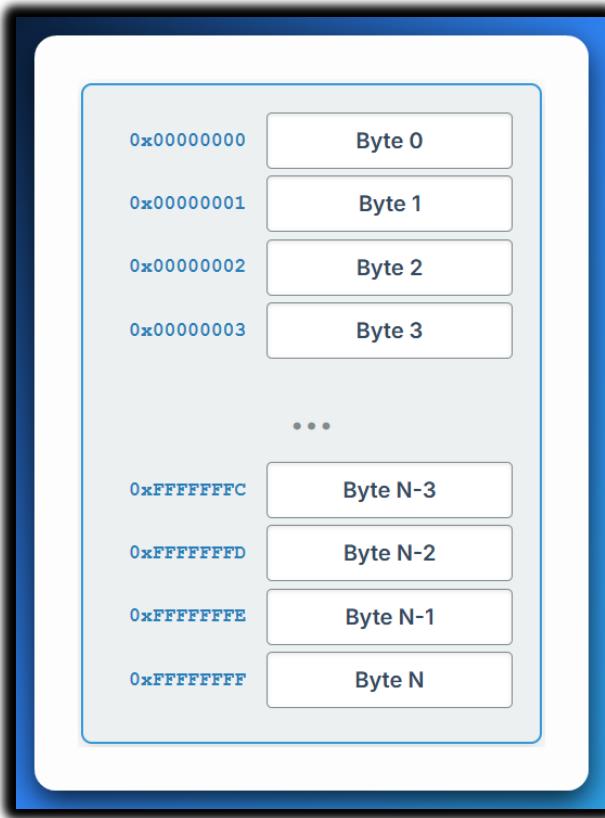
But the address always refers to the first byte.

That's like saying:

"Go to house 123456. If I say byte, steal one envelope. If I say word, take four in a row, starting from that house."

💡 Linear Memory Model: The Illusion of Simplicity

This is the *illusion* every beginner sees:



See the [Addressing.html](#)

Looks clean, right? Like RAM is just a big array of bytes you can index into like `RAM[0x123456]`.

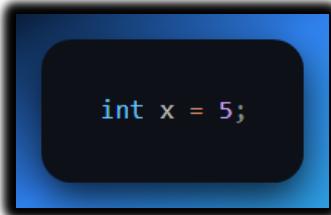
But reality? 🤔

⚠️ The Plot Twist: Linear View Is a Lie (Sort of)

Underneath this smooth address space is a **warzone** of mappings, protections, virtual memory tricks, segmentation (in real/protected mode), and paging tables.

- In **real mode**, segment:offset math gives fake “linear” addresses.
- In **protected mode**, the OS maps your `0x00400000` to some other **physical** location via page tables.
- In **long mode (x86-64)**, even *more* indirection.

So, when you think:



Your code might think x lives at 0x00401234, but under the hood, that:

- Is virtual.
- Mapped to a page.
- Backed by physical RAM, or not.
- Maybe even swapped to disk.
- All depending on your mode and OS.

So yeah — it **starts** as a linear view. But when reversing or writing exploits, you'll often have to:

- Find the **real** physical or kernel address.
- Traverse **page tables**.
- Bypass **address randomization (ASLR)**.
- Reconstruct **segment mappings** in 16-bit code.

🔥 Assembly vs High-Level: Why You NEED to Know This

In C:

```
int a = 5;  
int b = a + 2;
```

```
mov eax, [some_address]  
add eax, 2
```

You have to **know** where that address points.

- Did the OS give you that address?
- Is it in the data section?
- Heap? Stack?
- Is that memory even mapped and readable?

One wrong assumption = segmentation fault, or worse, silently reading junk data.

⌚ Final Thought (For This Chunk)

Addressing isn't just "how memory works."

It's **how everything works** once you're outside the compiler's sandbox.

- Want to write a shellcode? You'll hand-write addresses.
- Want to trace a malware unpacker? You'll follow memory derefs.
- Want to write a bootloader? You'll directly manage segment:offset pairs.
- Want to debug a crash? You'll match an instruction with a corrupt address.

So, here's a rule to tattoo mentally:

- ! If you don't control the address, you don't control the code.
- ! If you don't understand the address, you're blind.

⌚ Endianness — The CPU's Accent When Speaking Memory

Endianness is like the **dialect** your CPU uses to write down numbers in memory.

You and your friend both know what the number “0x12345678” means — but if one writes it **backwards**, and the other reads it **forwards**, y'all are gonna get very confused real quick.

Let's break it down to **core concepts, visuals, real-world cases**, and why it matters in RE/malware/file parsing.

📦 What is Endianness?

When a value spans **multiple bytes** (e.g., a 32-bit int = 4 bytes), the CPU has to decide:

“Which byte goes to the *lowest* memory address? The most significant one (biggest part) or the least significant one (smallest part)?”

That decision = **endianness**.

👉 Little-Endian (Used by Intel x86/x64, AMD64)

“Put the Least Significant Byte (LSB) at the Lowest Address.”

It's **backwards** to how humans normally write numbers — but not to the CPU.

Example:

Let's store 0x12345678 into memory.

Byte	Value
+0	0x78 ← LSB (least significant byte)
+1	0x56
+2	0x34
+3	0x12 ← MSB (most significant byte)

So, in **hex dumps or memory editors**, you'll see: **78 56 34 12**

And you gotta mentally flip it to read the real number. This **byte reversal** is why malware analysts constantly “swap bytes” in IDA or Ghidra.

🧠 *Mnemonic:*

"Little-endian puts the **little stuff first**." (just like how babies say "me first" 😊)

💡 **Big-Endian (Used in old-school CPUs, network protocols)**

"Put the **Most Significant Byte (MSB)** at the Lowest Address."

It feels more “natural” to humans. Example:

Same number: **0x12345678**

Byte	Value
+0	0x12 ← MSB
+1	0x34
+2	0x56
+3	0x78 ← LSB

So, in memory dump: **12 34 56 78**

Looks cleaner, right? But almost *none* of your PCs today use it natively — unless you're reading:

- Network packets (TCP/IP headers use big-endian)
- Older PowerPC binaries
- Foreign firmware formats (e.g., routers, smart devices)

🧠 *Mnemonic:*

"**Big-endian puts the big stuff first.**"

Real-World Analogy

Imagine you have a box labeled “4-digit PIN: 1234.”

- Big-endian guy: opens the box and reads left to right: 1, 2, 3, 4 
- Little-endian guy: opens the box and reads **right to left**: 4, 3, 2, 1  (unless he knows to flip it)

So, if someone drops a little-endian number on a big-endian parser = chaos. 

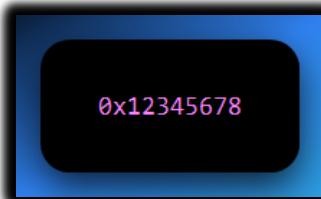
Why This Matters in Reversing / Malware Analysis

1. Hex Dumps Look “Reversed”

When analyzing a binary or memory:



You must **reverse** it to understand what it *really* means:



If you forget? You'll misinterpret pointers, constants, even jump addresses.

2. Parsing File Formats

Think about parsing a BMP, PE header, ELF, ZIP, or MP3:

- The structure says: “next 4 bytes = offset”
- You read 40 00 00 00
- That's 0x40 in little-endian... not 0x40000000!

3. Network Forensics

Network protocols use **big-endian** ("network byte order").
So if malware sends a packet like:



That might mean **port 80** (HTTP), not 0x5000. In C:

```
htons(80); // convert host byte order to network byte order
```

💡 **htonl, ntohs, htons, ntohs** = functions to flip bytes correctly.

4. Cross-Platform Hell

You're dealing with:

- An iOS app (uses ARM).
- A router firmware (MIPS).
- A PC binary (x86).

All of these may interpret the **same bytes** differently unless you respect their endianness.
Binary parsing tools (like Binwalk) usually try to auto-detect, but you must know what's up.

✖ Bonus Fun: Obfuscation via Endianness

Some malware writers or packers intentionally **flip bytes** or **misalign fields** to throw off analysts e.g. a pointer to code might be stored as:

```
xor eax, eax  
mov al, 0x78  
shl eax, 8  
mov al, 0x56  
...
```

So, you get: 0x12345678 → but not in one go.

Or they do something like:

```
db 0x78, 0x56, 0x34, 0x12 ; raw bytes  
call [esp]                  ; treat them as a pointer
```

Knowing endianness lets you spot this and go:

"Aha! This junk is a valid pointer in disguise."

TLDR Summary

Type	Stored As (0x12345678)	Used By
Little-endian	78 56 34 12	x86, x86-64, most modern systems
Big-endian	12 34 56 78	Networking, older CPUs, PowerPC

- **Always know** the target architecture.
- **Always flip** bytes if the dump looks weird.
- **Never assume** raw bytes = the final number.

This knowledge stacks FAST. You're not just learning addressing — you're building the instincts of a *binary surgeon*.