💡 FIRST PRINCIPLES: WHAT IS ADDRESSING, REALLY?

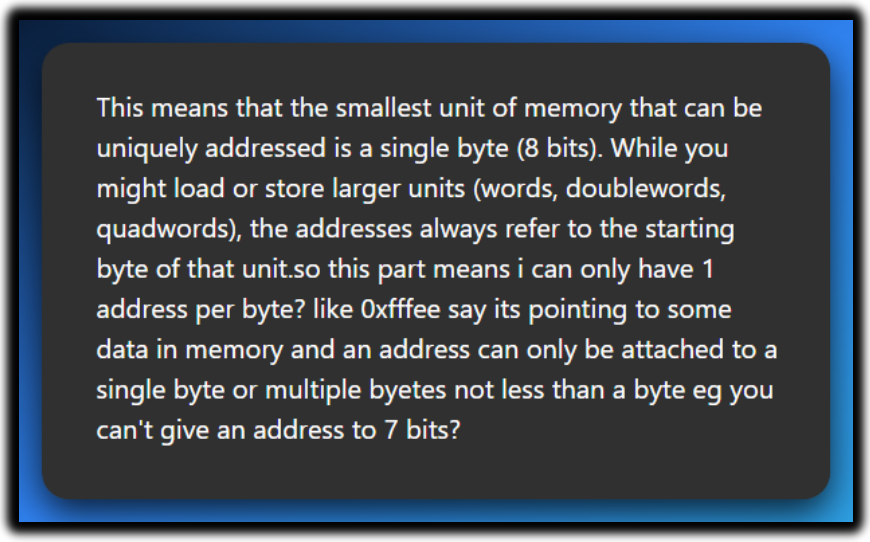
Think of the CPU like a blind beast that only knows how to poke memory by **number** — it doesn't "see" variable names, doesn't care about objects or data types — it just sees **addresses**.

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.



👉 **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 0xfffee pointing to some data..."*

Yes! 0xfffee 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 0xfffee, continues to 0xfffef, 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 0xfffee").

If you need to access a bit, you do:

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.

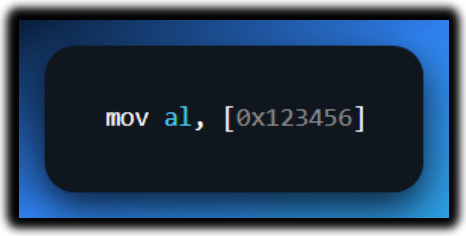
Byte-addressable sounds like a boring fact until you realize…

- 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.

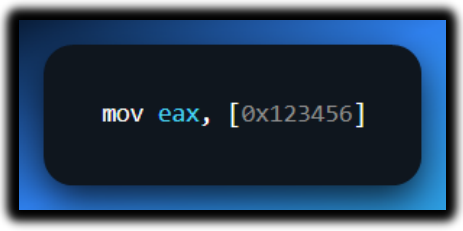
Byte-level control = surgical power.

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.



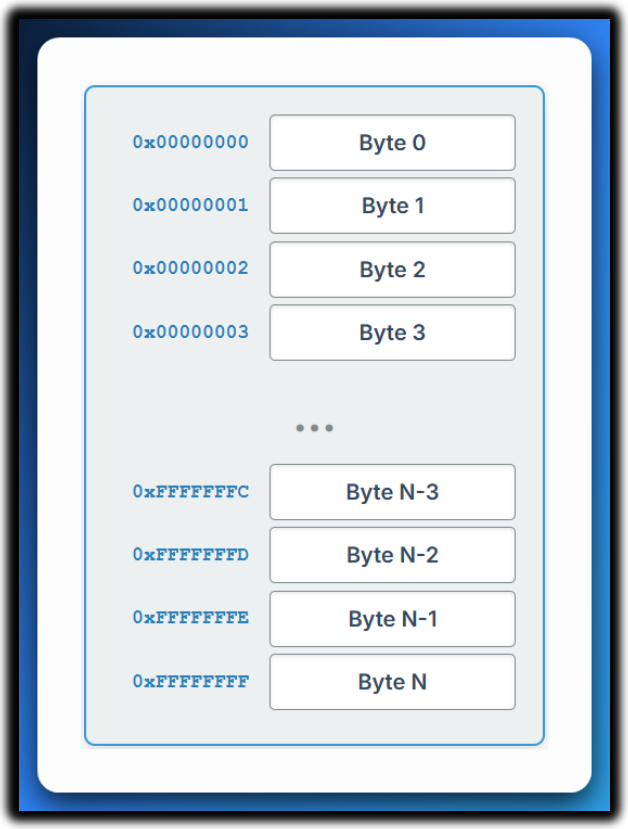
But the address always refers to the first byte.

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].

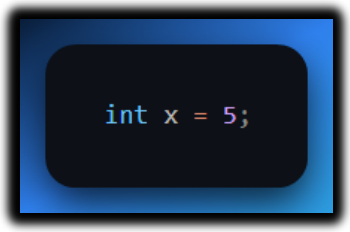
But reality? 👇

🚨 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:

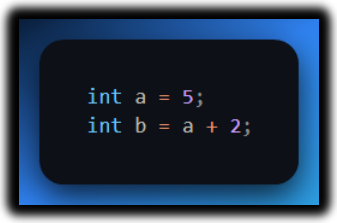
- Is virtual.
- Mapped to a page.
- Backed by physical RAM, or not.
- Maybe even swapped to disk.
- All depending on your mode and OS.

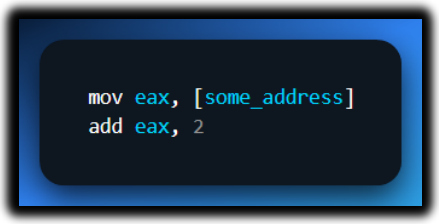
So yeah — it **starts** as a linear view. But when reversing or writing exploits, you'll often have to:

- 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:





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:*

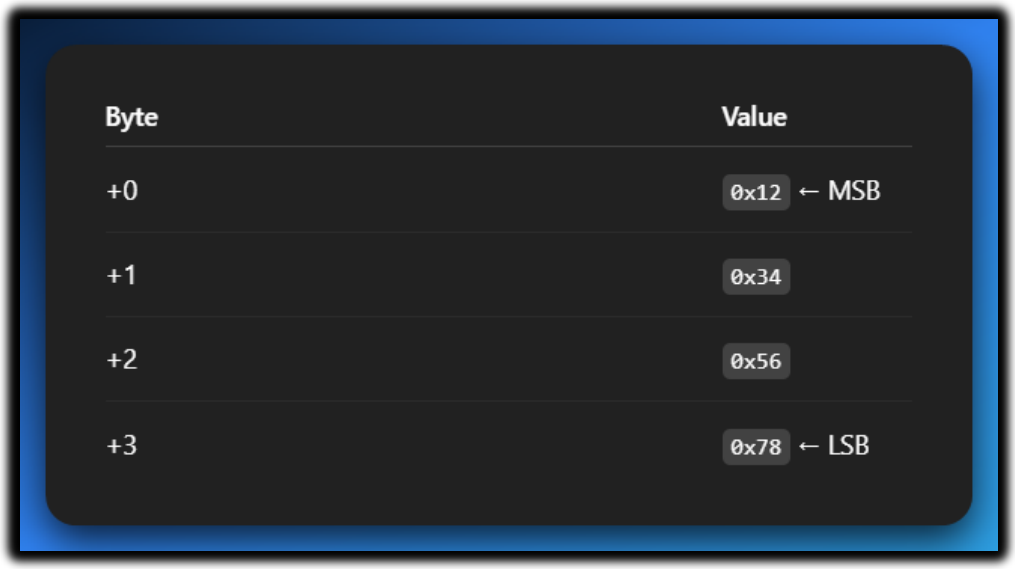
*"Little-endian puts the **little stuff first**." (just like how babies say "me first" 😆)*

🤯 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.”

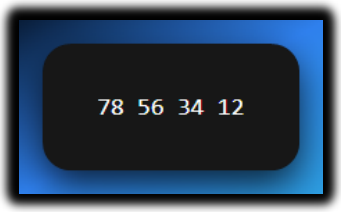
- Big-endian guy: opens the box and reads left to right: 1, 2, 3, 4 ✅
- Little-endian guy: opens the box and reads **right to left**: 4, 3, 2, 1 ❌ (unless he knows to flip it)

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

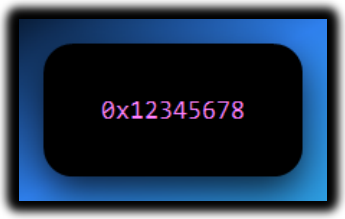
🕵️ Why This Matters in Reversing / Malware Analysis

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

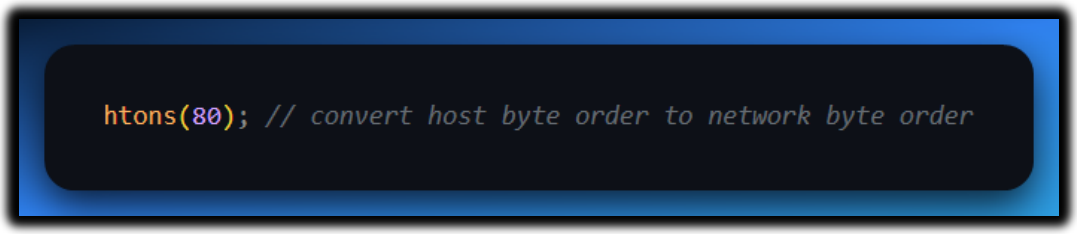
Think about parsing a BMP, PE header, ELF, ZIP, or MP3:

- 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:



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

4. Cross-Platform Hell

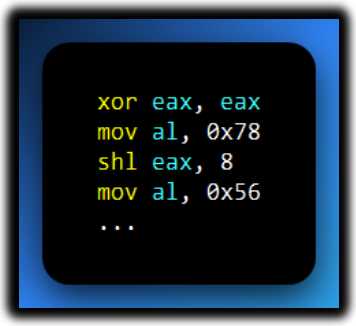
You're dealing with:

- 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

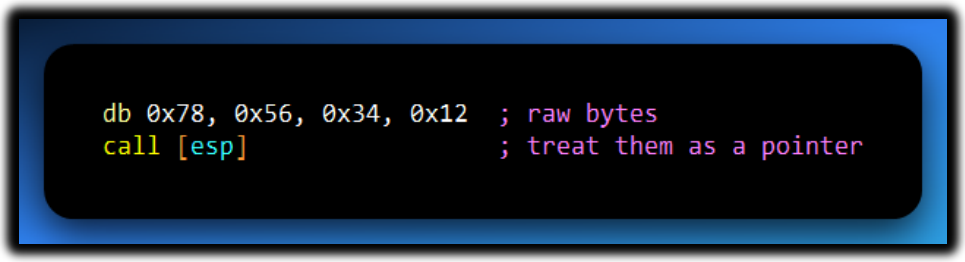
Some malware writers or packers intentionally **flip bytes** or **misalign fields** to throw off analysts e.g. a pointer to code might be stored as:



```
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:

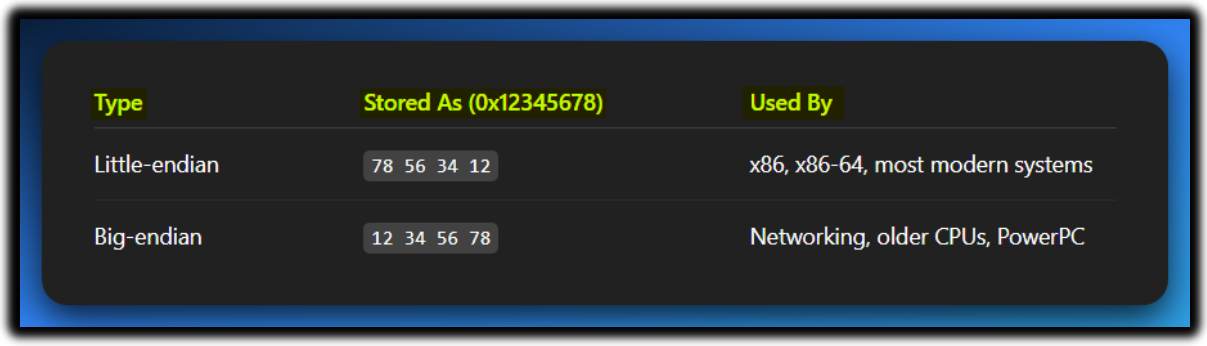


```
db 0x78, 0x56, 0x34, 0x12  ; raw bytes
call [esp]                 ; treat them as a pointer
```

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*.

🧠🪄 The Holy Trinity of Addressing: Logical vs Linear vs Physical

Welcome to the **Matrix of Memory**. What you see as [my_variable] in assembly?

That ain't real. It's a *lie the OS and CPU tell you* to keep your code sane.

Here's the three layers of that lie:

1. Logical Address (aka Virtual Address): Your Personal Map

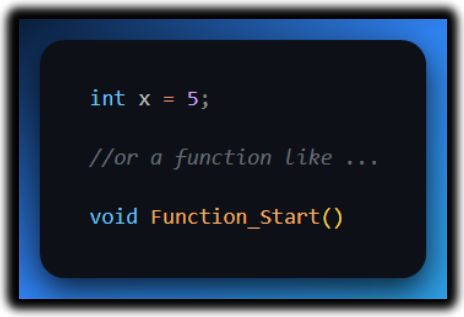
A **logical address** is indeed the address generated by the CPU when your program is running.

When you write something like **int x = 5;** in your C code, you're telling the compiler, "Hey, I need a spot in memory to store this integer x."

Think of the **logical address** as your **program's personal map or a wish list for memory**. 🗺️

This address is **virtual** because it's not the actual physical spot in RAM. It's more like a promise or a placeholder.

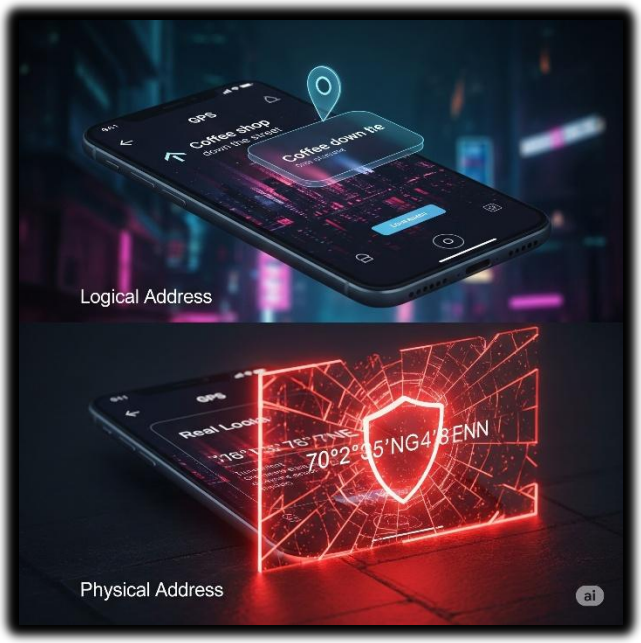
When you're writing code in C, Assembly, or Python, and you declare a variable like…



… you're essentially giving these pieces of your program **logical names** and assuming they'll reside at a specific, predictable "address" within your program's memory space.

Real-world example: When you tell your friend, "Meet me at 'the coffee shop down the street'," you're giving them a **logical address**.

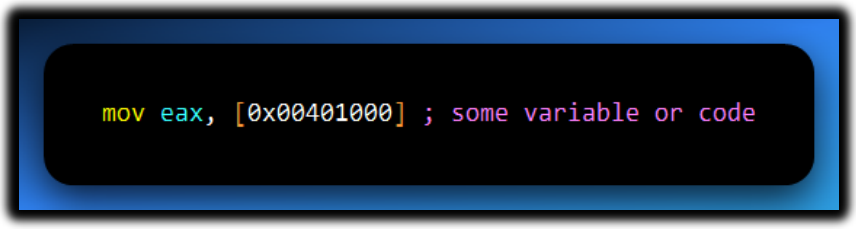
You don't know the exact street number or building ID yet, but you know the *concept* of where you want to go.



Your program thinks similarly: **"I need a variable here,"** or **"My function starts there."**

It's an address *relative to your program's perspective*.

ASM Code you write:



That 0x00401000? That's a **logical address**.

But like a street address on a letter, you don't actually deliver it.

You hand it to the system, and say **"make sure this reaches the correct house."**

🧠 Used in:

- Assembly code.
- Decompiled functions in Ghidra.
- High-level program instructions.
- ptrace, read()/write() syscalls.
- Malware writing to its own heap/stack.

🧠 Pro Tip:

Every process sees a **clean, fake world** of memory starting at 0. That's the illusion virtual memory provides.

🧩 2. Linear Address (aka Post-Segmentation Address)

If you're in Protected Mode (which all modern OSes are), then the CPU first checks segmentation (CS, DS, SS, etc.), applies any segment base, and gives you a linear address.

The MMU doesn't always directly translate logical to physical. Often, there's an intermediate step, especially in modern operating systems using protected mode and paging.

But here's the modern twist:

99.9% of OSes configure segments with base = 0, limit = 4GB+, and basically ignore segmentation.

So, in most systems: **Logical Address ≈ Linear Address**

But if you're in:

- **Real mode** (bootloader/BIOS)
- **VM or sandbox**
- **Old malware or self-modifying code**
- **Or messing with LDT/GDT**

...then this difference **matters**. Like, *"you will misread a pointer and crash"* kind of matters.

📦 Analogy:

This is the **fully qualified address** the post office uses internally — like *"Banana Street, Zone 17, Nairobi, Kenya, 00505."*

👮 CPU uses this as an **intermediate step**:

- If paging is OFF → this = physical
- If paging is ON → this goes to MMU → becomes physical

🧠 You only really *see* linear addresses when you're working in:

- Kernel exploit dev
- BIOS/bootloaders
- IDT/GDT/LDT manipulation

🧱 3. Physical Address

This is where the bits **actually** sit on your RAM sticks. The real, raw deal.

It's what the **Memory Controller** sees after the MMU's done playing translator.

📦 Analogy:

It's the **GPS coordinates** of the exact house. No ambiguity. If you gave this to a drone, it'd drop a missile precisely.

👾 Malware sometimes **fakes page table entries** to point logical addresses to custom physical regions (e.g. rootkits hiding in unused RAM).

💾 Address Space — The Sandbox You Live In

The "address space" is the **full range** you can operate in.

🧠 Two Types:

a) Logical Address Space:

- What the CPU lets your program see.
- In 32-bit → 0x00000000 to 0xFFFFFFFF (4 GB max)
- In 64-bit → Theoretically 2^{64} = 18 exabytes.
- Real CPUs? Usually 48–52 bits of that is actually usable due to page table limits.

🧪 Example:
You might only have 16GB RAM, but your process can **pretend** it has 4GB all to itself. That's the magic of…

🌌 Virtual Memory: The Great Illusion

A beautiful lie told by the OS and hardware so that every process thinks it owns the whole machine.

🔍 Why It Exists:

- So, two programs don't step on each other's toes
- So, your compiler, calculator, and malware can all run in isolated peace
- So, the system can **swap** memory to disk when RAM is tight
- So, your kernel can pull off magic like:
    - ➢ Mapping .exe files directly into memory.
    - ➢ Running hundreds of apps with only 16GB RAM.
    - ➢ Isolating apps from each other with memory protection.

🚀 Key Features of Virtual Memory

✅ Overcommitting:

A program can malloc(100GB) even if you have 8GB RAM.
Why? Because memory is just an illusion until actually used (lazy allocation, demand paging).

✅ Memory Protection:

If one program touches another's memory → boom → segfault.

This is thanks to page tables + the MMU checking access flags.

✅ Memory Mapping Files:

Use mmap() to treat part of a file like it's memory.

Used in:

- Malware loading PE files manually
- Lazy-loading resources
- Fast IO for DBs, browsers, games

✅ Shared Libraries (DLLs/.so):

One physical copy of libc.so can be **mapped** into the memory of **every** program running.

Virtual memory maps it **in different spots**, but it's the *same underlying data*.

💣 Attack Surface for Malware & Reverse Engineers

- 🔓 **Manipulating Page Tables:** Rootkits can hide themselves by mapping their memory as "unmapped" to tools like ps, top, or memdump.
- 🛑 **Anti-Debugging via Memory Layout:** Malware may check if certain addresses are mapped to detect debuggers or sandboxes.
- 💉 **Android Reverse Engineering:**
  - SELinux policies can stop access to kernel pages.
  - mmap() tricks can obfuscate payload loading.
  - Native libraries (via NDK) use virtual memory like crazy.
- 🧠 **You (the analyst) must:**
  - Know which layer you're looking at
  - Translate logical ↔ physical ↔ file offset when needed
  - Understand how malware might *trick* tools like IDA by manipulating memory maps

🧠 Summary Table: