x86 PROCESSOR MODES OF OPERATION: THE CPU'S DIFFERENT HATS

Let’s begin from the frontlines of CPUs that are in the market.

⚙️ What is x86?

x86 is the **instruction set architecture (ISA)** — aka the low-level “language” — that your CPU understands.

It started with Intel’s **8086** processor in 1978, and all its descendants (8088, 80286, 80386, etc.) stuck to this architecture — hence the name "x86".

Think of x86 like **the blueprint of how to speak to a CPU** — how to run instructions like MOV, ADD, INT, and all those.

So, when you say:

"This app runs on x86"

You're saying:

"This app is made to run on a CPU that understands the 8086-style instruction set."

🔥 What is AMD?

AMD (Advanced Micro Devices) is a **company**, like Intel.
They make CPUs. But... here's the plot twist:

Back in the day, **AMD cloned Intel's x86 chips** (legally, through a licensing agreement), and later *developed their own* x86-compatible processors.

So, basically:

- *AMD builds CPUs that speak the x86 language.*
- *So does Intel.*
- *They are different brands, but they run the same kind of machine code.*

🧠 So, Why the Confusion?

Because of this:

- *When people say "x86", they often mean Intel-style CPUs in general (Intel or AMD).*
- *And when AMD released the first 64-bit x86 CPUs, they called the extension x86-64.*
- *Sometimes called AMD64 — which further confuses people.*

🤯 Real Talk: So...

Term	Means...
x86	The original 16/32-bit Intel instruction set architecture (8086, 80386, etc.)
x86-64 / AMD64	The 64-bit extension of x86, designed by AMD and later adopted by Intel
AMD	A company that makes x86-compatible CPUs (and GPUs too)
Intel	Also a company that makes x86-compatible CPUs (the originator)

⚔️ Who Wins?

Both AMD and Intel make x86 CPUs.

But AMD *designed* the 64-bit version first (x86-64), and Intel just adopted it (after failing with their own 64-bit attempt called IA-64).

So yeah:

🔥 *AMD gave birth to the modern 64-bit x86 you use today. Intel had to catch up.*

🧠 x86 Processor Modes — The CPU's Wardrobe Change

Your CPU isn't stuck wearing one outfit. It's got **multiple modes**, and each one changes *how it behaves*, *what it can access*, and *what it's allowed to do*.

These "modes" control things like:

- 🔐 **Who has privilege** (OS or app?)
- 🧠 **How memory is addressed** (flat vs segmented, 20-bit vs 32-bit vs 64-bit)
- 📜 **Which instructions are valid** (some modes unlock advanced features)

Basically, **the mode defines the rules of the world the CPU lives in**.

🏛️ Why So Many Modes?

Over time, as systems evolved from:

- 🪦 Simple MS-DOS single-tasking
- → 🧩 Multitasking OSes (Windows, Linux)
- → 🛡️ Secure kernel-user separation
- → 🤹 Virtualization and emulators

...Intel had to keep adding new hats (modes) to let the CPU play nice with all these roles. Instead of wiping out old tech, they just **layered the new stuff on top**.

So yeah — we got **Real Mode, Protected Mode, Long Mode, SMM, VMX**, and more.

👮 Who's in Charge?

The **Operating System (OS)** is the one flipping the mode switches behind the scenes. You, as the programmer or reverse engineer, are seeing the effects — but it's the OS that goes:

"Alright, flip to Protected Mode. I need privilege levels and memory protection now."

Or:

"Booting up? Cool, let's start in Real Mode, do some BIOS work, then shift into the real stuff."

🪱 Malware Angle

Why do you care?

Because malware loves playing mode games. It might:

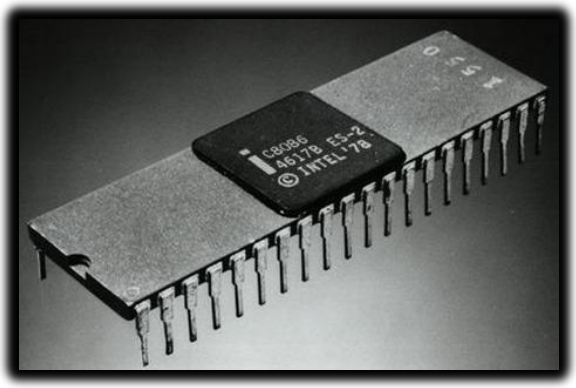
- *Switch to Real Mode or SMM to bypass protections.*
- *Use Ring 0 tricks in Protected Mode.*
- *Or abuse VMX (virtualization extensions) to hide inside fake hypervisors.*

Knowing the mode tells you **what kind of mischief is even *possible***.

🧠 Real Mode — "The BIOS Brain"

Real Mode is the foundational operating mode of all x86 processors.

This goes back to the earliest Intel 8086/8088 processors used in the original IBM PC.



🕹️ **Where it all begins:** Every x86 CPU *wakes up* in Real Mode when it powers on. Doesn't matter if you're on a Core i9 or some dusty 486 — the boot process *starts* here. This is for backward compatibility.

🔍 1. Memory Model (aka the CS:IP Magic Trick)

🧱 No Flatland Here

Unlike modern 64-bit systems that treat memory as one long continuous block (flat memory model), **Real Mode is segmented**. Think of it like a broken city: divided into blocks, with each block needing an address and an offset to find the right house.

🧮 Segmented Addressing (20-bit)

Despite using **16-bit registers**, the CPU in Real Mode still manages to address **1MB of memory**. How? It cheats a little:

🧠 Formula:

```
Physical Address = Segment * 16 + Offset
```

Translation? It takes the segment, multiplies it by 16 (aka shifts it left by 4 bits), and then adds the offset. This creates a **20-bit address** (despite the CPU only having 16-bit registers).

```
16-bit segments + 16-bit offsets = 💀 cursed 20-bit addresses
```

✅ Max Addressable Memory: 2^{20} = **1,048,576 bytes (1MB)**

❓ Why the Weird ×16?

Because **Intel was cooking in the 80s**. They had 16-bit registers (so max value = 65535 = 64KB). But 64KB wasn't enough.

They wanted more memory *without redesigning the whole architecture*, so they introduced this offset-based system where the segment tells you which 64KB "window" you're in, and the offset says where in that window.

It's like saying:

"Go to street 0x3000 → Now walk 0x0042 meters in."

🧠 Real Mode = Direct Physical Access

There's **zero memory protection**. When a program accesses memory, it's **talking directly to physical RAM**. No virtualization, no page tables, no mapping layers.

This is:

- ✅ Great for simplicity and speed.
- ❌ Terrible for safety.

Any program can:

- Overwrite the OS.
- Smash other program's memory.
- Trash the BIOS.
- Crash the whole system with a single bad pointer.

It's like giving every user root access with a shotgun and no armor.

🔑 Privilege Level: There is None

In Real Mode, **everyone is Ring 0** — the CPU doesn't enforce any privilege separation. BIOS, MS-DOS, your 1988 snake game — **they all run at the same level**.

Imagine giving everyone in a city the **master key to every door**, even the government ones.

Fine in the 80s, nightmare now.

📜 Instruction Set: 16-bit Only

Real Mode only supports the original **8086/8088 16-bit instruction set**. That means:

- No 32-bit instructions.
- No SIMD.
- No paging.
- No privilege levels.

It's vintage. And vintage doesn't come with modern safety gear.

🧠 Practical Impact (Why REs Care)

This segment-offset style still pops up:

- In **bootloaders** (MBR/VBR code is Real Mode!)
- In **BIOS analysis**
- In **low-level DOS programs**
- In early-stage **malware**, like bootkits and firmware rootkits

Also, when you first enter Ghidra or IDA for old binaries or shellcode, you'll often see this model used — and if you misunderstand segment logic, **you'll miscalculate addresses and get lost**.

🚪 4. What It's Still Used For:

- **BIOS/UEFI Boot:** The firmware runs in Real Mode during POST (power-on self-test) and initial boot sequence.
- **Bootloaders:** Stage 1 bootloaders like GRUB or bootmgr (before handing off to protected mode).
- **Legacy Systems:** Emulating DOS games, boot disks, or old BIOS software.
- **Malware or Bootkits:** Some rootkits live here briefly to hook interrupt vectors before the OS takes over.

💡 In RE, if you're looking at something hooking INT 13h, INT 10h, INT 21h, etc., you're likely still in Real Mode land.

🔄 Switching from Real Mode

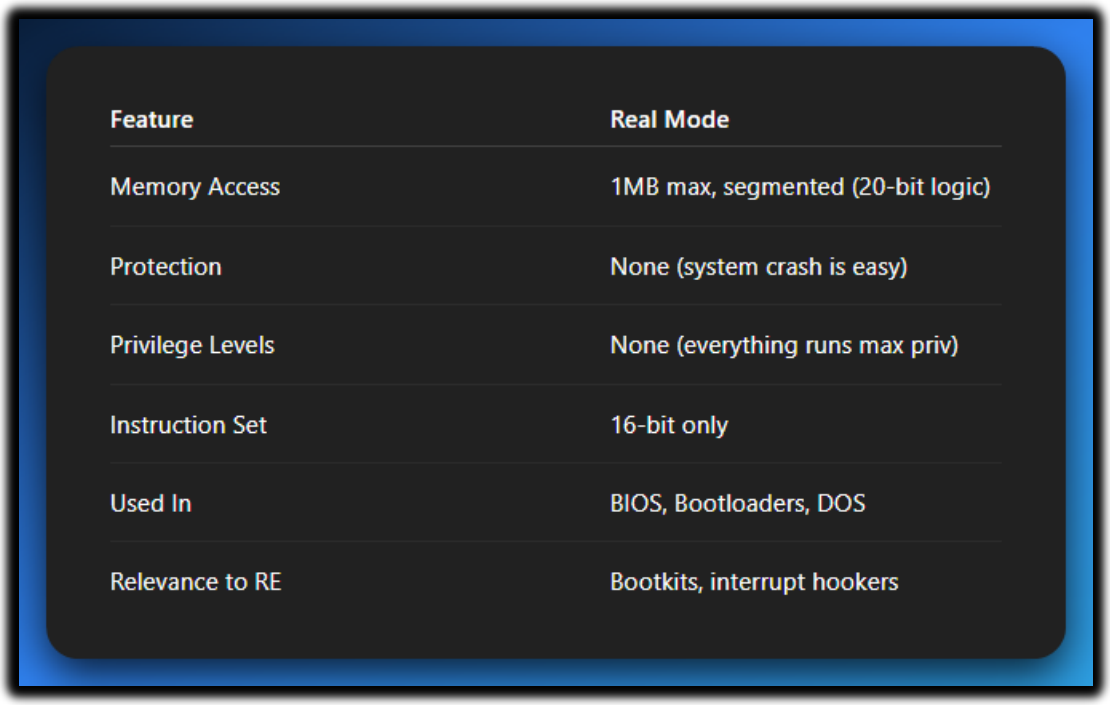
You can't just *press a button* and become modern.

The switch to **Protected Mode** involves:

- *Setting the PE (Protection Enable) bit in CR0.*
- *Loading a valid GDT (Global Descriptor Table).*
- *Far jumping to the new code segment to flush the pipeline.*

Once in Protected Mode — boom — you get memory protection, paging, multitasking, etc.

💥 TLDR



Feature	Real Mode
Memory Access	1MB max, segmented (20-bit logic)
Protection	None (system crash is easy)
Privilege Levels	None (everything runs max priv)
Instruction Set	16-bit only
Used In	BIOS, Bootloaders, DOS
Relevance to RE	Bootkits, interrupt hookers

Usage:

System Boot-up (BIOS/UEFI): All x86 systems begin in Real Mode. The BIOS (Basic Input/Output System) or UEFI firmware starts execution here to perform initial hardware

checks and load the boot loader. The boot loader's first task is often to switch the processor out of Real Mode into a more advanced mode if a modern OS is being loaded.

Legacy Applications (MS-DOS): This mode is primarily used for running very old 16-bit applications designed for MS-DOS, which were written with the assumption of direct hardware access.

Switching: While simple, Real Mode includes mechanisms (like setting specific bits in control registers and jumping to new code) to allow the processor to transition into other, more advanced modes.

2. Protected Mode: The Dawn of Modern Computing

Protected Mode, introduced with the Intel 80286 processor, is a monumental leap forward and the native state for all modern 32-bit (and implicitly 64-bit) operating systems like Windows, Linux, and macOS. It provides the essential features for robust multitasking and secure computing environments.

- **Memory Model:**
    - **Memory Protection:** This is the "protection" in Protected Mode. Programs are given their own isolated memory areas called **segments** (and later, pages with paging enabled). The processor hardware enforces these boundaries, preventing one program from reading from or writing to memory belonging to another program or the operating system. If a program attempts an unauthorized memory access, the CPU generates a **protection fault** (e.g., a "segmentation fault" or "general protection fault"), which the OS can then handle (usually by terminating the offending program).
        - **Analogy:** Each program now has its own designated office space in a skyscraper. The CPU (building security) ensures that no program (employee) can enter or tamper with another program's office without explicit permission.
    - **Virtual Memory and Paging:** Protected Mode fully supports **virtual memory**, which allows the OS to use disk space as an extension of RAM. It also introduces **paging**, a more granular memory management scheme than segmentation. With paging, memory is divided into fixed-size "pages" (typically 4 KB). This enables the OS to present a consistent, large virtual address space to each program, even if the physical memory is fragmented or smaller.

    - **Larger Address Space:** With the 80386 and later processors, Protected Mode supports a 32-bit address space, allowing access to **4 Gigabytes (GB)** of memory.

- **Privilege Levels (Rings): The Security Tiers**
    - Protected Mode implements **hardware-supported privilege levels**, often visualized as "rings." Intel x86 processors have four rings: Ring 0, Ring 1, Ring 2, and Ring 3.
        - **Ring 0 (Kernel Mode):** This is the most privileged level, with direct access to all hardware and memory. Operating system kernels (like the core of Windows or Linux) run in Ring 0.
        - **Ring 1, 2:** These are typically unused by modern OSs, though they can be configured for drivers or other privileged components.
        - **Ring 3 (User Mode):** This is the least privileged level. All user applications (web browsers, games, word processors) run in Ring 3. They have limited access to hardware and memory and must make system calls to the OS kernel (Ring 0) to request privileged operations.
    - **Impact:** This ring protection model is crucial for system stability and security. Malware in Ring 3, for instance, cannot directly modify the kernel in Ring 0 or other programs' memory without triggering a protection fault, assuming the OS is properly designed.
- **Multitasking:** Protected Mode enables efficient **multitasking**. The OS can manage multiple programs simultaneously by giving each program a "time slice" of CPU execution, rapidly switching between them while preserving their individual states (context switching). Memory protection ensures that these concurrent programs don't interfere with each other.
- **Full Instruction Set:** All x86 instructions, including 32-bit instructions and floating-point operations, are available in Protected Mode.
- **Usage:** This is the mode in which virtually all modern 32-bit (and many 64-bit compatibility modes) operating systems and their applications run.

3. Virtual-8086 Mode (VM86 Mode): Bridging the Past and Present

Virtual-8086 Mode is a special sub-mode of **Protected Mode**, introduced with the 80386 processor. Its purpose is to allow legacy 16-bit Real Mode applications (like old MS-DOS programs) to run within a modern multitasking Protected Mode environment without requiring any modifications to the old code.

- **The Problem it Solved:** Before VM86 mode, running a DOS program on a Protected Mode OS was problematic because DOS programs expected direct hardware access and had no concept of memory protection.
- **How it Works (Virtualization):**
    - The operating system, running in Protected Mode (Ring 0), creates a **virtualized 8086 environment** for each DOS program.
    - Each virtual-8086 "monitor" (part of the OS) intercepts privileged instructions and hardware access attempts made by the 16-bit DOS application.
    - Instead of letting the DOS program directly access hardware or memory (which would violate Protected Mode's rules and crash the system), the OS's virtual-8086 monitor **emulates** or **filters** these accesses. For example, if a DOS program tries to directly write to video memory, the OS intercepts this, redraws the virtual screen, and then updates the actual physical screen, giving each DOS program its own "virtual" display.
    - This provides a safe, isolated environment. If a DOS program crashes within its virtual-8086 session, it typically does not affect other programs or the stability of the main operating system.
- **Multitasking Legacy Apps:** A modern OS can run multiple, completely separate virtual-8086 sessions concurrently, allowing you to multitask several old DOS applications alongside your modern Windows or Linux applications.
- **Analogy:** Imagine a historical reenactment. The modern city (Protected Mode OS) sets up a special, fenced-off area (Virtual-8086 mode) where actors (DOS programs) can live and behave exactly as they did in the 1980s, completely unaware of the modern city around them. Any requests they make (e.g., for supplies) are handled by designated "handlers" (the OS's VM86 monitor) who translate their old-fashioned requests into modern actions, ensuring they don't break character or disrupt the present-day city.

4. Long Mode: The 64-bit Frontier

Long Mode is the operating mode utilized by all **64-bit x86 processors (x86-64 or AMD64 architectures)**. It's the latest major evolution of the x86 architecture, designed to break the 4GB memory barrier and enable significantly larger address spaces and register sets.

- **Memory Model:**
    - **64-bit Addressing:** Long Mode fundamentally changes the addressing scheme to 64-bits (though current implementations typically use 48-bit

virtual and 52-bit physical addresses). This allows access to vastly larger amounts of memory – theoretically up to 16 Exabytes (EB) of virtual address space and 4 Petabytes (PB) of physical address space – far exceeding anything previously possible.

    - **Mandatory Paging:** In Long Mode, **paging is mandatory**. Segmentation (as it was used for memory protection in 32-bit Protected Mode) is largely "flat" and effectively disabled for memory protection, with paging taking over that role entirely.

- **Increased Processing Power & Registers:**
    - **64-bit Registers:** General-purpose registers (like RAX, RBX, RCX, RDX, etc.) are extended to 64 bits, allowing the CPU to process larger chunks of data in a single operation.
    - **More Registers:** Long Mode also introduces 8 additional general-purpose registers (R8-R15) and 8 additional SSE/AVX registers (XMM8-XMM15/YMM8-YMM15), significantly improving performance by reducing the need to constantly load/store data from memory.
- **Sub-modes:** Long Mode itself has two primary sub-modes:
    - **64-bit Mode:** This is where true 64-bit applications and operating system kernels run, leveraging all the 64-bit features.
    - **Compatibility Mode:** This allows 32-bit and 16-bit Protected Mode applications to run on a 64-bit OS. The processor operates in a 32-bit or 16-bit environment within this mode, but the underlying OS is still in 64-bit Long Mode. **Crucially, Real Mode or Virtual-8086 Mode programs cannot be run natively in Long Mode; they typically require virtualization software (like a full VM) to run.**
- **Usage:** All modern 64-bit operating systems (Windows x64, Linux x64, macOS) and their applications run in Long Mode.

5. System Management Mode (SMM): The Covert Maintainer

System Management Mode (SMM) is a special-purpose, highly privileged operating mode intended for system-level functions, often transparent to the operating system and applications. It was introduced with the Intel 386SL.

- **Purpose:** SMM is primarily used by the **system firmware (BIOS/UEFI)** and hardware manufacturers for low-level system management tasks such as:
    - **Power Management (ACPI):** Adjusting CPU fan speeds, controlling power states (sleep, hibernate), battery management.

    - **System Security:** Handling specific security features (e.g., some forms of DRM, Trusted Platform Module interactions).
    - **Hardware Emulation/Compatibility:** Providing legacy support for certain hardware devices or handling specific chipset quirks. For instance, emulating a PS/2 mouse/keyboard for older OS components even if the physical devices are USB.
- **Entry Mechanism (SMI):** SMM is entered by a **System Management Interrupt (SMI#)** signal, which can be triggered by specific hardware events (e.g., pressing the power button, overheating) or by software writing to a special I/O port.
- **Privilege Level & Transparency:**
    - SMM code runs at an even *higher* privilege level than Ring 0 (kernel mode). It's sometimes informally called "Ring -2."
    - When an SMI occurs, the CPU saves its entire current state (registers, flags, etc.) into a special, protected memory area called **SMRAM (System Management RAM)**. The CPU then switches to SMM, executes the SMI handler code (located in SMRAM), and once the handler completes, it restores the CPU's original state from SMRAM and returns to the previous mode (e.g., Protected Mode).
    - This process is designed to be **completely transparent** to the operating system, which is unaware that the CPU temporarily left its control to perform SMM tasks.
- **Security Implications (for you!):** For reverse engineers and malware analysts, SMM is a particularly interesting (and dangerous) area. Because it's transparent to the OS and runs at such a high privilege, malware that manages to gain control within SMM can be incredibly difficult to detect, analyze, and remove. It can persist even across OS reboots and bypass many standard security measures. This makes SMM a prime target for sophisticated rootkits and bootkits.

Switching Between Modes: A Carefully Orchestrated Dance

Transitioning between these modes is not a simple flip of a switch. The processor must execute a specific instruction or a precise sequence of instructions to change modes. These transitions often require special privileges (usually only the operating system kernel can initiate them) and involve careful manipulation of control registers (like CR0, CR4) and descriptor tables that define memory segments and pages.

- For instance, switching from Real Mode to Protected Mode involves enabling the "Protection Enable" (PE) bit in the CR0 control register and setting up segment descriptors and potentially page tables.
- Similarly, entering Long Mode requires first setting up a minimal Protected Mode environment, then enabling PAE (Physical Address Extension) in CR4, loading page tables, and finally setting the Long Mode Enable (LME) bit in a Model Specific Register (MSR) before enabling paging.

The OS as the Mode Manager

Ultimately, the **Operating System (OS)** is the arbiter of which mode the x86 processor operates in. Modern OSs start in Real Mode, then quickly transition to Protected Mode (for 32-bit OSs) or Long Mode (for 64-bit OSs) and stay there for the vast majority of their operation. Applications run within the context established by the OS, conforming to the rules and limitations of that mode (e.g., user-mode applications running in Ring 3).

Each mode defines its own distinct set of rules for:

- **Registers:** While many registers are consistent, their effective sizes and interpretations can change (e.g., 16-bit vs. 32-bit vs. 64-bit registers).
- **Memory Management System:** How addresses are translated from logical to physical, and how memory protection is enforced.
- **Interrupt Handling Mechanism:** How the CPU responds to hardware and software interrupts.

Understanding these modes is foundational. As you delve deeper into reverse engineering and malware analysis, you'll find that much of the challenge lies in understanding *which mode* code is executing in, how it got there, and what privileges it has based on that mode. Malware often tries to elevate its privilege or hide its presence by exploiting vulnerabilities related to these mode transitions or by executing in highly privileged modes like SMM.