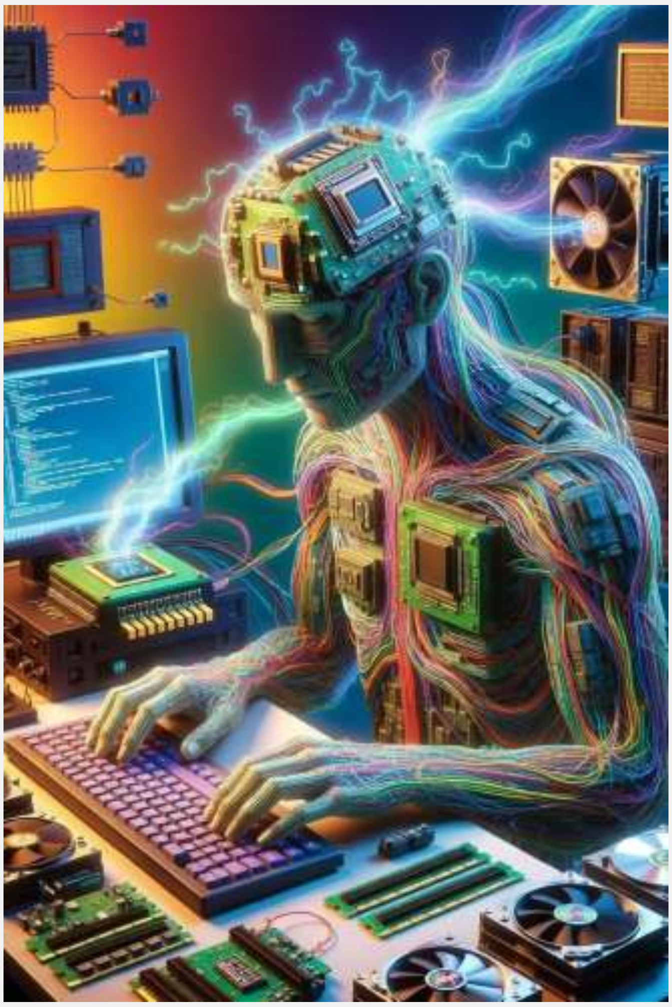
How Computers Actually Work

From Power Button to Programs — Explained Simply



Author: Rahul Kumar

YouTube: CyberWings Security

Domain: Cybersecurity

Table of Contents

1. Introduction: What Is a Computer Really?

Welcome to CyberWings Security! I'm Rahul Kumar, and today we're stripping away the layers to reveal what a computer truly is.

Most people think a computer is:

- A screen
- A keyboard
- Some apps

But in reality, a computer is:
A machine that accepts data, processes it using instructions, stores it, and produces output.

At its core, a computer does only one thing:
It processes electrical signals (ON and OFF)

Everything else—Windows, Linux, hacking tools, browsers, videos—is built on top of this simple idea. This foundational knowledge is why we at CyberWings Security believe understanding hardware is the first step to mastering cybersecurity.

2. The Language of Computers: Binary (0 and 1)

Computers do not understand:

- English
- Images
- Numbers like 10, 25, 100

They understand only two states:

State	*Meaning*	*Electrical Reality*
1	*ON / TRUE*	*Voltage present*
0	*OFF / FALSE*	*No voltage*

This system is called Binary.

Example:
Decimal number 5 → Binary representation 101

Every:

- Letter
- Image
- Video
- Program
- Even malware

...is converted into billions of 0s and 1s. When we analyze malicious code at CyberWings Security, we're often looking at these binary patterns.

3. From Power Button to Life: The Boot Process

When you press the power button, this chain reaction begins:

Step-by-step:

1. Power Supply Unit (PSU)

- Converts AC power → DC power
- Supplies correct voltage to components

2. BIOS / UEFI starts

- Stored in ROM (Read-Only Memory)
- Performs POST (Power On Self Test)
- Checks hardware functionality

3. Hardware Initialization

- RAM checked
- CPU detected
- Keyboard, disk, display initialized

4. Bootloader Loads

- Finds the Operating System
- Loads it into RAM

5. Operating System Starts

- Windows / Linux / macOS takes control
- Kernel loads, services start

Until the OS loads, you don't even have a usable computer. This boot sequence is a critical attack vector that we study in CyberWings Security for securing systems at the firmware level.

4. The Brain: CPU (Central Processing Unit)

The CPU is where computation actually happens. Think of it as the worker in our factory analogy.

The Instruction Cycle:

1. Fetch instructions from memory
2. Decode what they mean
3. Execute the operation
4. Store the result

This happens billions of times per second (GHz = billions of cycles/second).

5. Inside the CPU: Core Components

5.1 Control Unit (CU)

- The traffic police for data
- Decides what happens next
- Directs data flow between components

5.2 Arithmetic Logic Unit (ALU)

- The calculator inside CPU
- Performs:
    - Addition, subtraction
    - Comparisons (greater than, equal to)
    - Logical operations (AND, OR, NOT)

5.3 Registers

- Ultra-fast memory inside CPU
- Stores:
    - Current instructions
    - Intermediate results
    - Memory addresses

Registers are faster than RAM but much smaller. In cybersecurity, register values can reveal what a program was doing when it crashed—critical for debugging exploits.

6. How a Program Actually Runs (Real Example)

Let's say you open Calculator and press: 5 + 3

What really happens:

1. Keyboard sends binary signals for '5', '+', '3'
2. OS interprets input, sends to Calculator app
3. CPU receives instruction: ADD 5, 3
4. ALU performs binary addition
5. Result stored in register
6. Output sent to screen driver
7. Pixels light up to show '8'

Key insight: The CPU doesn't know "5 + 3" as a math problem—it only knows binary addition patterns. This abstraction layer is what allows both legitimate programs and malware to run.

7. Memory Hierarchy: The Storage Ladder

7.1 RAM (Random Access Memory)

Feature	*RAM*
Speed	*Very fast*
Volatile	*Yes (data lost on power off)*
Purpose	*Running programs*

Example: Open browser → loaded into RAM | Close browser → removed from RAM

7.2 Storage (HDD / SSD)

Feature	*Storage*
Speed	*Slower than RAM*
Volatile	*No*
Purpose	*Permanent data*

Example: Files, OS, Applications

7.3 The Complete Hierarchy (Fastest to Slowest):

1. Registers (inside CPU) - Like your hand while writing
2. Cache (near CPU) - Like sticky notes on desk
3. RAM - Like your entire desk space
4. SSD/HDD - Like a filing cabinet

OS moves programs from storage → RAM → CPU registers. This movement creates attack opportunities that CyberWings Security teaches about in memory forensics.

8. Motherboard: The Nervous System

The motherboard:

- Connects all components
- Allows communication via buses

Types of buses:

- Data bus - Carries actual data
- Address bus - Carries memory addresses
- Control bus - Carries commands

Without the motherboard, components are useless individually. This interconnectivity is why physical security matters in cybersecurity.

9. Input/Output: The Interface with Reality

Input Devices (How data enters):

- Keyboard, mouse, webcam, microphone

- Each converts physical action → electrical signal → binary data

Output Devices (How results are shown):

- Monitor, printer, speakers
- Binary data → converted back into images, sound, text

Example: Typing 'H'

1. Press 'H' key → electrical signal
2. CPU interrupt → reads signal
3. OS maps to ASCII (01001000)
4. Display driver sends pattern to monitor
5. Pixels light up → you see 'H'

10. Operating System: The Real Boss

The OS is the manager in our factory analogy. It:

- Controls hardware resources
- Manages memory allocation
- Schedules CPU tasks
- Enforces security policies
- Provides user interface

Without OS:

- No multitasking
- No file system
- No applications

OS as a Middleman:
User → Application → OS → Hardware

This layered architecture creates security boundaries that both protect and can be exploited.

11. Multitasking: The Illusion of Simultaneity

Even with one CPU core:

- Programs don't run simultaneously
- CPU switches very fast between tasks
- This is called Context Switching

This is why malware can hide—it gets CPU time slices like any legitimate program. Understanding this helps in malware detection at CyberWings Security.

12. Where Cybersecurity Fits In (Critical Section)

Understanding how computers work is the foundation of cybersecurity:

Attackers exploit:

- Memory vulnerabilities (buffer overflows target RAM)
- CPU instruction manipulation (return-oriented programming)
- OS behavior (privilege escalation)
- Boot process (rootkits in BIOS/UEFI)

Defenders analyze:

- Memory forensics (what was in RAM during attack?)
- Malware analysis (what binary patterns indicate malicious intent?)
- Exploit development (to build better defenses)
- Reverse engineering (understanding malicious binaries)

Example: A Keylogger Attack

1. Malware installs via phishing email
2. Hooks into keyboard input layer
3. Logs keystrokes to hidden file
4. Sends file to attacker via network
5. Defense: Antivirus scans memory/processes, detects abnormal keyboard hooking behavior

13. Simple Analogy: The Computer Factory

Computer Part	*Factory Role*	*Cybersecurity Relevance*
CPU	*Worker*	*Executes both legitimate and malicious instructions*
RAM	*Work table*	*Where active programs (including malware) live*
Storage	*Warehouse*	*Where malware persists between reboots*
OS	*Manager*	*Security policy enforcement point*
Input	*Raw materials*	*Attack vectors (malicious files, network packets)*
Output	*Finished product*	*Data exfiltration, display corruption*

14. Key Takeaways for Aspiring Security Professionals

1. Computers run on binary - Everything reduces to 0s and 1s
2. CPU executes instructions step by step - Understand this to reverse engineer malware
3. RAM is temporary, storage is permanent - Critical distinction for forensics
4. OS controls everything - Most attacks target OS vulnerabilities
5. Security starts with fundamentals - You can't protect what you don't understand

15. Why This Matters for Cybersecurity

At CyberWings Security, we believe:

- You cannot defend a system without understanding how it works
- Every layer of abstraction is both a protection and a potential vulnerability
- The best security professionals think like the computer itself

Whether you're analyzing malware, conducting forensics, or building secure systems—you're working with these fundamental components.

A Cybersecurity-First, Internals-Level Foundation

1. Why Cybersecurity Starts *Below* the Operating System

Most cybersecurity learners start at:

- Tools
- Commands
- Frameworks
- CVEs

That is **backwards**.

Attacks do not target “software” — they target **assumptions in how computers work**.

Malware does not attack *apps* first.
It attacks:

- Memory behavior
- CPU execution flow
- Privilege boundaries
- Instruction handling
- OS trust models

If you don’t understand **how the machine actually executes instructions**, you are:

- Blind in malware analysis
- Weak in exploit understanding
- Dependent on tools
- Easily fooled by obfuscation

2. Binary Is Not “0 and 1” — It Is *Control*

In cybersecurity, binary is not a number system.
Binary is **control over hardware state**.

Binary means:

- Voltage present → instruction executes
- Voltage absent → instruction does not exist

Every security concept reduces to:

Who controls which bits, at what time, with what privilege

Security implication:

- Buffer overflow = attacker controls bits beyond intended boundary
- ROP = attacker controls instruction pointer bits
- Privilege escalation = attacker flips permission-related bits

3. CPU Execution Model (This Is Where Exploits Live)

The CPU does **not understand programs**.
It understands only:

Instructions fetched from memory

The CPU Trust Assumption:

"Whatever instruction is at this memory address is legitimate."

This single assumption enables:

- Shellcode execution
- Code injection
- ROP chains
- JOP attacks

4. The Instruction Cycle (Attack Surface Explained)

Normal Instruction Cycle:

1. **Fetch** instruction from memory address
2. **Decode** instruction
3. **Execute** instruction
4. **Update instruction pointer**
5. Repeat

Security Reality:

If an attacker controls:

- Memory contents → arbitrary code execution

- Instruction pointer → control flow hijack
- Stack contents → function redirection

This is why:

- Stack is protected
- ASLR exists
- DEP/NX exists

5. Registers: The Hidden Crown Jewels

Registers are:

- Faster than RAM
- Directly control execution

Security-critical registers:

- **Instruction Pointer (IP / RIP / EIP)**
  → controls *what runs next*
- **Stack Pointer (SP)**
  → controls *function calls*
- **Flags Register**
  → controls conditional execution

Exploit perspective:

Exploitation is mostly about **register manipulation**

ROP = chaining gadgets by controlling RIP
Privilege escalation = altering control registers

6. Memory Is Not "Storage" — It Is a Battlefield

Virtual Memory (Critical for Security)

Processes do NOT access real RAM directly.

They see:

- **Virtual addresses**
  Mapped to:
- **Physical memory**

Why this matters:

- Memory isolation
- Process separation
- Kernel protection

When this fails:

- Kernel exploits
- Meltdown / Spectre
- DMA attacks

7. Stack vs Heap (This Is Mandatory Knowledge)

Stack:

- Function calls
- Local variables
- LIFO structure

Attacks:

- Stack buffer overflow
- Return address overwrite

Heap:

- Dynamic memory
- Objects, buffers

Attacks:

- Heap overflow
- Use-after-free
- Double free

Many modern exploits move from stack → heap because stack protections improved.

8. Operating System: The Security Enforcer

The OS is:

- Not just a convenience layer
- A **security policy engine**

OS responsibilities:

- Memory isolation

- Privilege separation
- Process scheduling
- Access control
- Hardware abstraction

Security principle:

The OS decides **who is allowed to do what with hardware**

9. User Mode vs Kernel Mode (The Biggest Boundary)

User Mode:

- Applications
- Browsers
- Tools
- Malware (initially)

Kernel Mode:

- Drivers
- OS core
- Hardware access

Crossing from user → kernel is **game over**

That's why:

- Kernel exploits are rare but deadly
- Drivers are high-risk attack surfaces

10. System Calls: The Only Legal Door

Applications cannot:

- Access disk
- Access memory
- Access network

  directly.

They must use **system calls**.

Security angle:

- Hooking syscalls = stealth malware

- Abusing syscalls = privilege escalation
- Monitoring syscalls = detection

11. Processes, Threads, and Malicious Behavior

Process:

- Isolated memory
- Own resources

Thread:

- Shares memory
- Independent execution

Malware abuse:

- Inject threads into trusted processes
- Hollow legitimate processes
- Abuse shared memory

12. Boot Process: Rootkits Start Here

Boot sequence is **implicit trust**.

If attacker compromises:

- Bootloader
- Firmware
- UEFI

Then:

- OS security is meaningless
- Antivirus never sees malware

This is where:

- Bootkits
- Firmware malware
- Persistent threats live

13. Why Antivirus Is Not Enough

Antivirus works at:

- File level
- Known signatures

But attacks operate at:

- Memory
- Execution flow
- Kernel level

Modern defense needs:

- Behavior analysis
- Memory inspection
- CPU telemetry

14. Why Cybersecurity Tools Fail Without Fundamentals

Without internals knowledge:

- Logs are confusing
- Memory dumps look random
- Exploit code looks magical

With internals knowledge:

- You *see* execution paths
- You *predict* attack behavior
- You *understand* failures

15. Mental Model for Security Engineers

Think in layers:

Electricity

↓

Binary

↓

Instructions

↓

CPU Execution

↓

Memory

↓

Operating System

↓

Applications

↓

User

Attacks move **upwards**
Defenses must work **downwards**

16. Core Cybersecurity Truth

You cannot secure what you do not understand.

And you **cannot understand security** without understanding:

- CPU behavior
- Memory model
- OS trust boundaries

CPU & Exploitation Deep Dive

How the Processor Becomes the Primary Attack Surface

1. The Most Dangerous Security Assumption Ever Made

Every modern CPU is built on one fatal assumption:

“If an instruction exists at a memory address, it must be legitimate.”

The CPU:

- Does NOT check intent
- Does NOT know “good” or “bad” code
- Does NOT understand programs or users

It only does:

Fetch → Decode → Execute

Exploitation = abusing this blind trust

2. CPU Is Not “Running Programs” — It Is Executing Addresses

This distinction is critical.

The CPU does **not execute files**
The CPU does **not execute functions**

The CPU executes:

Instructions at the address stored in the Instruction Pointer (IP/RIP)

Security Translation:

If an attacker controls:

- Instruction Pointer → they control execution
- Memory at that address → they control behavior

This is why:

- RIP overwrite = code execution

- ROP exists
- JOP exists
- Return address is sacred

3. Instruction Pointer (RIP): The Crown Jewel

What RIP Actually Is

- A CPU register
- Holds the **next instruction address**
- Auto-increments after each instruction

Normal Flow:

RIP → instruction_1

RIP → instruction_2

RIP → instruction_3

Exploit Flow:

RIP → attacker_controlled_address

Once RIP is hijacked:

- The program is no longer in control
- The OS is still *trusting* the CPU
- Security controls are bypassed *by design*

4. Stack: The Most Abused Data Structure in History

The stack exists to support:

- Function calls
- Local variables
- Return addresses

Typical Stack Frame (Simplified):

|----------------|

| Return Address |

|----------------|

| Saved RBP |

|----------------|

| Local Vars |

|----------------|

What the CPU Expects:

- Return address stays untouched
- Stack grows & shrinks predictably

What Attackers Do:

- Overwrite local variables
- Overwrite return address
- Redirect execution

Buffer overflow is not a “bug” — it’s a trust violation

5. Stack Buffer Overflow (Mechanics, Not Myth)

Vulnerable Code (Conceptual):

```
char buf[64];

gets(buf);
```

What Happens:

1. CPU allocates stack space
2. buf lives *below* return address

3. Input exceeds 64 bytes
4. Data overwrites return address
5. Function returns
6. CPU jumps to attacker-controlled RIP

CPU does **exactly what it was told**

6. Why Modern Exploits Rarely Inject Code Directly

Early exploits:

- Injected shellcode
- Jumped directly to it

Defenses introduced:

- NX / DEP (No Execute)
- Stack non-executable
- Heap protections

Attacker Adaptation:

“Fine. I’ll reuse your code.”

This gave birth to **ROP**.

7. Return-Oriented Programming (ROP): CPU Abuse at Its Finest

Key Idea:

- Use existing executable instructions
- Chain them using ret

ROP Gadget:

pop rdi

ret

Why This Works:

- ret pops address from stack
- CPU blindly jumps
- Attacker controls stack

- Attacker controls execution

ROP does not inject code — it hijacks control flow

8. Why CPU Can’t Detect ROP

From CPU’s perspective:

- Instructions are valid
- Memory is executable
- Control flow is “legal”

CPU has **no semantic understanding** of:

- Intent
- Attack patterns
- Logic

That’s why:

- ROP is powerful
- Detection is OS-level, not CPU-level

9. Registers Attackers Care About (Very Important)

General-Purpose:

- RAX, RBX, RCX, RDX

Control Registers:

- RIP → execution
- RSP → stack control
- RBP → frame control
- EFLAGS → conditional logic

Exploit Goal:

Gain influence over **control registers**

Even partial control is enough.

10. Calling Conventions: Exploitation Depends on This

Example (x86-64 Linux):

- Arguments passed via registers:
  - RDI, RSI, RDX, RCX

ROP chains must respect this or crash.

Exploit Reality:

You are not just chaining gadgets
You are **emulating a function call using CPU rules**

This is why exploit dev is *engineering*, not hacking.

11. Context Switching: How Malware Hides

CPU executes **one thread at a time**.

OS rapidly switches:

- Registers
- Stack
- Instruction pointer

Malware Trick:

- Hide in legitimate process
- Execute briefly
- Yield CPU
- Avoid detection windows

This is CPU scheduling abuse.

12. User Mode vs Kernel Mode (CPU-Enforced Boundary)

CPU enforces **privilege levels**.

User Mode:

- Limited instructions
- No hardware access

Kernel Mode:

- Full control

- Direct memory & hardware access

Exploit Goal:

Escalate execution **into kernel mode**

Once achieved:

- AV dies
- Logs lie
- System belongs to attacker

13. Why Kernel Exploits Are Hard (and Valuable)

Kernel exploitation requires:

- Deep CPU understanding
- Paging knowledge
- Interrupt handling
- Memory isolation bypass

But once done:

Persistence + stealth = achieved

14. Why Meltdown & Spectre Were Revolutionary

They proved:

- CPU *optimizations* can break security
- Speculative execution leaks data
- CPU is not security-neutral

The processor itself became an attack surface.

15. Defensive Controls vs CPU Reality

Defense	What It Tries	Why CPU Still Obeys
ASLR	Randomize	CPU doesn’t care
NX	Block exec	ROP reuses exec
Stack Canary	Detect	CPU executes anyway
DEP	Prevent shellcode	CPU executes gadgets

Defenses **detect or restrict**, but CPU **executes**.

16. The Exploiter’s Mental Model

“Where does the CPU get its next instruction?”

If you can answer that at every moment,
you can:

- Build exploits
- Break malware
- Understand crashes
- Read core dumps
- Reverse binaries

17. Core Cybersecurity Truth (CPU Edition)

The CPU is not broken.
Security fails because we *trust it blindly*.

Memory Internals for Hackers & Defenders

Why Memory Is the Real Battleground

1. Why Memory Matters More Than Files

Most beginners think attacks live in:

- Files
- Executables
- Scripts

Reality:

Modern attacks live in memory

Why?

- Memory is fast
- Memory is volatile
- Memory is trusted
- Memory is hard to inspect

Malware prefers memory because:

- AV scans files
- EDR struggles with transient execution
- Memory disappears on reboot

2. Virtual Memory: The First Security Illusion

What Processes *Think* They See

Each process believes:

- It owns memory starting at 0x00000000
- It has a full address space

What Actually Happens

- OS maps **virtual addresses → physical memory**
- Same virtual address in two processes ≠ same physical memory

Security Purpose

- Process isolation
- Crash containment
- Privilege separation

Security Failure

If isolation breaks:

- One process reads another
- Kernel memory leaks
- Sandbox escapes

3. Virtual Address Space Layout (Critical)

A typical user process memory layout:

High Addresses

Stack

Shared Libraries|

Heap

BSS / Data

Code (Text)

Low Addresses

Attacker Question:

“Which region can I influence, and which region does the CPU trust?”

4. Stack: Control Flow Territory

Stack Properties

- LIFO
- Grows downward
- Stores:
  - Return addresses
  - Local variables
  - Saved registers

Why Stack Is Dangerous

- Implicit trust
- CPU assumes return addresses are valid
- Small mistakes = full control

Defender View

- Stack canaries
- Shadow stacks
- ASLR

Attacker Adaptation

- Partial overwrite
- Info leaks
- ROP chains

5. Heap: Logic Corruption Territory

Heap Properties

- Dynamic memory
- Long-lived
- Shared across program logic

Heap Attacks Enable:

- Object corruption
- Pointer redirection
- Logic manipulation

- Silent exploitation

Famous Heap Bugs

- Use-after-free
- Double free
- Heap overflow

Heap exploits don’t always crash—**they lie quietly**.

6. Memory Permissions (CPU + OS Enforcement)

Each memory page has permissions:

- Read
- Write
- Execute

NX / DEP:

- Data ≠ Code

Why It Failed:

- Code reuse (ROP)
- JIT spraying
- RWX legacy regions

7. Memory from Defender Perspective

Defenders care about:

- Unexpected executable memory
- Abnormal page permissions
- Suspicious memory regions
- In-memory only payloads

Memory forensics > disk forensics in modern attacks.

OS Internals for Security

The Operating System Is a Security Policy Engine

8. OS Is Not Software — It Is a Referee

The OS decides:

- Who gets CPU
- Who gets memory
- Who touches hardware
- Who can talk to whom

Security is not enforced by apps.
It is enforced by **OS trust boundaries**.

9. User Mode vs Kernel Mode (Revisited, Deeper)

User Mode

- Applications
- Browsers
- Malware (initial stage)

Kernel Mode

- Drivers
- Memory manager
- Scheduler
- Security enforcement

Kernel compromise = absolute trust collapse

10. System Calls: The Only Legal Gateway

Applications cannot:

- Open files
- Allocate memory
- Send packets

They must ask the kernel via **syscalls**.

Malware Abuse:

- Direct syscalls (bypass hooks)
- Syscall number spoofing
- Hooking syscall tables

Defender Focus:

- Syscall behavior anomalies
- Argument inspection
- Sequence patterns

11. Process & Thread Internals

Process

- Own virtual memory
- Own handles
- Own security token

Thread

- Own stack
- Shares memory
- Own instruction pointer

Malware Tricks

- Thread injection
- Process hollowing
- APC injection
- Early-bird injection

All rely on **OS internals knowledge**.

12. OS Scheduling & Stealth

CPU scheduling:

- Time-sliced
- Priority-based

Malware:

- Executes briefly

- Yields CPU
- Avoids long execution windows

This defeats naive behavioral detection.

13. OS Memory Manager as Attack Surface

The memory manager:

- Allocates pages
- Sets permissions
- Maps memory

Bugs here lead to:

- Kernel memory disclosure
- Privilege escalation
- Sandbox escape

How Malware Actually Executes

From Entry to Persistence

14. Malware Does Not “Run” — It Transitions States

Malware lifecycle:

Entry

→ Execution

→ Persistence

→ Privilege Escalation

→ Defense Evasion

→ Payload Action

Each stage abuses:

- CPU behavior
- Memory trust
- OS mechanisms

15. Initial Execution Techniques

- Exploit → shellcode
- Script → interpreter abuse
- Living-off-the-land binaries
- Fileless memory loaders

Goal:

Get instructions into CPU execution path

16. In-Memory Execution (Critical)

Modern malware:

- Loads payload into memory
- Marks pages executable
- Never touches disk

Techniques:

- Reflective loading
- Manual PE mapping
- Stolen handles

This defeats:

- Signature AV
- File scanners

17. Persistence Mechanisms (OS Abuse)

Persistence lives in:

- Registry
- Scheduled tasks
- Services
- Boot process
- Kernel drivers

Advanced malware:

- Firmware persistence
- Bootkits
- Hypervisor abuse

18. Defense Evasion (Why Detection Is Hard)

Malware evades by:

- API unhooking
- Direct syscalls
- Sleep obfuscation
- Environmental checks

CPU still executes.
OS still schedules.
Defenders must *observe*, not assume.

19. Memory-Only Malware (The Nightmare)

Characteristics:

- No files

- No registry
- No obvious indicators
- Lives in memory
- Re-infects on boot via logic

Detection requires:

- Memory forensics
- Behavioral correlation
- OS internals awareness

20. One Unified Mental Model

Think like this:

CPU → Executes blindly

Memory → Trusted container

OS → Enforces rules

Malware → Abuses assumptions

Defense → Observes violations

21. Final Cybersecurity Truth

Attackers don’t break systems.
They use them exactly as designed.

Security is about:

- Understanding design
- Detecting abuse
- Reducing trust

About Rahul Kumar & CyberWings Security

Hey, I'm **Rahul Kumar**, founder of **CyberWings Security**. Our mission is to bridge the gap between theoretical computer science and practical cybersecurity. We believe that understanding how computers actually work at the hardware level is the foundation of all security knowledge.

Subscribe to CyberWings Security for more deep-dives into:

- Ethical Hacking
- Penetration Testing
- Digital Forensics
- Incident Response
- Hardware-level security
- Malware analysis fundamentals
- Memory forensics
- Secure system design

Stay Secure. Stay Curious. Understand the Foundations.

https://www.linkedin.com/in/rahul-kumar2698/

https://www.youtube.com/@cyberwingssecurity