SXVM Documentation

Overview

SXVM is a red team utility designed for Windows x64 systems to support stealthy execution and prevent runtime environment detection or tampering. The tool focuses on evading traditional .NET runtime inspection and response behaviors, especially in security-monitored environments.

EntryPoint

The `Main` method in SXVM serves as the primary entry point of the application. It is responsible for preparing the runtime environment, unhooking EDR surveillance, detecting injected AV modules, and selecting the appropriate bypass strategy.

🔑 Function: `Main(string[] args)`

🧠 Behavior Summary

- **Custom PE Header Writing**
- **Unhooking AV/EDR API Hooks**
- **AV Detection via Module Inspection**
- **Adaptive Bypass Dispatching**

🔍 1. Custom PE Header Marking

```
ushort oldHeader = 0;
WriteCustomHeader(ExcludedHeader, out oldHeader);
```

Purpose:
Writes a specific value (`0x4F8F`) into the executable's PE header to tag the SXVM process. This allows external tools (e.g., rootkits or loaders) to **recognize and avoid interfering with SXVM**.

Use Case:

- Prevent DLL injection from unintended allies (e.g., external malware framework).
- Identify the process as "friendly" in multi-agent red team deployments.

🩹 2. Unhooking Suspicious DLLs

```
Unhooker.Unhook(@"ntdll.dll");
Unhooker.Unhook(@"kernel32.dll");
```

Purpose:
Restores the original syscall table and API entries from critical Windows DLLs to remove AV/EDR inline hooks.

Why These DLLs:

- `ntdll.dll`: Direct gateway to syscalls (most AVs hook here).
- `kernel32.dll`: Frequently hooked for process/thread operations.

Operational Benefit:
Removes memory instrumentation, restores syscall integrity, and prepares the environment for undetected behavior.

🔎 3. AV Module Scanning

```
foreach (ProcessModule module in Process.GetCurrentProcess().Modules)
{
        ...
}
```

Purpose:
Enumerates all loaded modules within the current process and searches for known antivirus modules. If matched, stores the vendor name for later bypass targeting.

🧬 AV Module Signatures

AV Vendor	DLLs Checked
Avast	`aswAMSI.dll`, `aswhook.dll`
BitDefender	`atcuf64.dll`, `bdhkm64.dll`
F-Secure	`fsamsi64.dll`, `fshook64.dll`, `fs_ccf_ipc_64.dll`
Sophos	`hmpalert.dll`, `SophosAmsiProvider.dll`
ESET	`eamsi.dll`
Kaspersky	`com_antivirus.dll`
Norton	`symamsi.dll`
Malwarebytes	`mbae64.dll`

🧰 4. AV-Aware Bypass Execution

```
switch (AntiVirus)
{
        ...
}
```

Purpose:
Once the AV is identified, the tool dynamically selects a tailored bypass routine. If no vendor is matched, it falls back to a default bypass.

Logic Table

AV Detected	Bypass Used
Avast	`Avast.Bypass(args)`
BitDefender	`BitDefender.Bypass(args)`
F-Secure	`Default.Bypass(args)`
Sophos	`Default.Bypass(args)`
ESET	`ESET.Bypass(args)`
Kaspersky	`Kaspersky.Bypass(args)`
Norton	`Default.Bypass(args)`
Malwarebytes	`Default.Bypass(args)`
Windows Defender/Unknown/None	`Default.Bypass(args)`

🧪 Notable Omission

```
throw new AccessViolationException();
```

This line is a **dead-end fallback**. It should never be reached unless something fails unexpectedly. It's not meant to be documented in production/usage context.

🧾 Additional Metadata

- **Entry Attribute:** `[STAThread]`
- **Header Signature:** `0x4F8F`
- **Platform:** `Windows x64`
- **Framework:** `.NET 4.5.1`
- **Execution Context:** Unprivileged (unless combined with an external rootkit or loader)

🔧 Function: `AttachHooks(string[] args)`

This internal static method installs runtime hooks into key .NET Framework behaviors. The goal is to disrupt introspection and sandbox evasion detection mechanisms commonly used by defenders or security software.

✅ Purpose

- Intercepts and neutralizes `Assembly.GetRawBytes()` (or its equivalents).
- Overrides `Environment.Exit()` to force an immediate and clean process termination.

🧩 Detailed Behavior

1. `GetRawBytes.Hook(args);`

Intent:
Hooks the functionality responsible for returning raw assembly bytes. This is crucial because

some .NET malware and packers use `Assembly.GetRawBytes()` (or internal methods that perform similar operations) to re-obtain the binary for dumping, analysis, or reflection.

Hook Effect:
Prevents .NET assemblies (including SXVM itself) from dumping their own memory/IL to disk or being observed by hooking into these low-level byte-fetching operations.

Obfuscation Use Case:
If a sandbox or analyst tries to monitor or dump the currently loaded module bytes from memory for reverse engineering, this hook ensures they receive empty or misleading data.

2. `EnvironmentExit.Hook(args);`

Intent:
Prevents suspicious exit calls by replacing `Environment.Exit()` with a hard kill `Process.GetCurrentProcess().Kill()`.

Hook Effect:
When a .NET app calls `Environment.Exit()`, it normally begins a graceful shutdown sequence which may trigger forensic tools, logs, or allow time for tamper detection mechanisms. This hook circumvents all of that.

Security Use Case:

- **Faster shutdown:** Ensures instant exit with no additional runtime cleanup.
- **Anti-debug/tamper:** Prevents an analyst from seeing managed `AppDomain` or `Finalizer` activity.
- **Cleaner exfiltration cutoff:** Useful for red team scenarios where an instant abort is safer.

Internal API

The `API.cs` module provides low-level internal methods that directly interact with process memory, implement EDR/AV patching logic, and enable secure resource extraction. These functions form the backbone of SXVM's stealth capabilities and runtime manipulation logic.

All methods in this file are internal, used only by trusted components inside SXVM's execution flow.

📦 WriteMemoryBlock

```
WriteMemoryBlock(IntPtr Address, byte[] src, uint size);
```

Description

Performs direct memory manipulation using unsafe pointers. Temporarily changes memory protection to `PAGE_EXECUTE_READWRITE (0x40)` to allow overwriting memory regions.

Used For:

- Code patching
- Signature modification
- In-memory data replacement

Safety:

- Surrounded by exception handling and memory protection restoration logic to ensure stability.

📤 ReadMemoryBlock

```
ReadMemoryBlock(IntPtr Address, uint size);
```

Description

Reads raw memory at a specified address and returns it as a byte array.

Used For:

- Creating backups of patched memory for later restoration
- Inspecting runtime memory states (e.g., to capture original AMSI bytes)

🧬 PatchEDR

```
PatchEDR(bool Patch_AMSI, bool Patch_ETW, bool UseWriteProcessMemory);
```

Description

Patches EDR/AV-related functions (AMSI and ETW) by overwriting their prologue with a `RET` instruction (`0xC3`). This effectively disables logging or scanning at runtime.

Functions Targeted:

Name	Location	Effect
`AmsiScanBuffer`	`amsi.dll`	Disables AMSI content scanning
`EtwEventWrite`	`ntdll.dll`	Suppresses ETW logging
`NtTraceEvent`	`ntdll.dll`	Disables tracing

Fallback Logic:
Supports both:

- `WriteMemoryBlock` (direct patching)
- `WriteProcessMemory` (for more flexible injection scenarios)

♻️ RestorePatchIntegrity

```
RestorePatchIntegrity(bool UseWriteProcessMemory, bool Exit = true);
```

Description

Restores the original bytes of patched AV/EDR functions using previously captured memory snapshots.

Originally Used In:
Environment Exit hooks to avoid AVs (e.g., BitDefender) flagging processes during termination if known AV functions remain patched.

Status:
Deprecated in SXVM. Modern patching on specific AVs (e.g., BitDefender) is now handled via

hardware breakpoints to avoid memory tampering altogether.

🧪 ExtractResource

```
ExtractResource(String filename);
```

Description

Extracts an embedded resource from the executing assembly.

Primary Use:
Loads `payload.bin` (Compressed + AES-encrypted .NET payload), which contains the actual post-execution implant or loader logic.

Resource Setup:

- Added to project in Visual Studio
- Set to: **"Embedded Resource"**
- Can be dynamically added using `dnlib` if needed

🔐 Global State Variables

```
private static byte[] Original_AmsiScanBuffer;
private static byte[] Original_EtwEventWrite;
private static byte[] Original_NtTraceEvent;
```

Purpose:
Temporarily stores original memory for AMSI, ETW, and NT functions before patching. Used by `RestorePatchIntegrity` if rollback is needed.

⚙️ Implementation Notes

- All unsafe operations are decorated with `[HandleProcessCorruptedStateExceptions]` to catch low-level memory faults.

- Patch size is hardcoded to **30 bytes**, which is typically sufficient to overwrite common prologues without affecting function size stability.
- `VirtualProtect` is used for altering memory protection during read/write operations.
- All memory interaction logic is designed to work within the **current process** (no remote injection).

🔗 GetManagedFunctionPointer

```
GetManagedFunctionPointer(void* managedPointer);
```

Description

Retrieves the native address of a managed (.NET) function pointer. Managed methods in .NET are first compiled to IL, but eventually JITed into native code. This method extracts that JITed memory address.

Use Case:

- Required when modifying or analyzing JIT-compiled .NET methods in memory.
- Often used for patching or redirecting execution at runtime.

What is JIT?

JIT stands for **Just-In-Time compilation**. It is a core part of the .NET runtime's execution model.

When you compile a C# program, it's not immediately turned into native machine code. Instead, it's compiled into an intermediate language (IL), also known as **CIL (Common Intermediate Language)** or **MSIL**.

When a method is **invoked for the first time at runtime**, the .NET runtime uses the **JIT compiler** to translate the IL into **native machine code**, which is then cached in memory and executed directly.

Key Points:

- **IL Code** is platform-independent bytecode (resides in `.dll`/`.exe` files).
- **JIT Compilation** occurs *at runtime*, not ahead-of-time (AOT).
- **JITed Memory** is where the native code is stored, this is what `GetManagedFunctionPointer` accesses.
- JITing provides **optimization** and **portability**, but creates a window where runtime introspection and patching can occur.

Red Team Relevance:

By accessing the JITed memory:

- You can **hook, modify, or monitor** managed methods *after* they've been compiled to native code.

🔎 FindBytes

```
FindBytes(IntPtr startAddress, uint size, byte[] pattern);
```

Description

Scans a memory region for a specific byte pattern.

Primary Role:
Used as a helper function in `FindECallFunction` and `FindECallFunctionViaModule` to locate function names or pointers embedded in modules. Often part of an ECall resolver or binary pattern matcher.

Resilient:
Returns `IntPtr.Zero` if pattern not found or if access errors occur (safe for stealthy scans).

☁️ FindECallFunction

```
FindECallFunction(string moduleName, string functionName);
```

Description

Resolves the address of a **CLR internal function** (ECall) from a known module name and function string.

What's an ECall?

ECalls are native implementations of managed .NET methods. They reside in the CLR's internal libraries (like `clr.dll`, `mscorwks.dll`, etc.).

How It Works:

- Locates the ASCII string of the function name.
- Scans for a pointer to that string.
- Reads the function address just before that pointer (pointer math trick).

Use Case:
Bypasses limitations of reflection or symbol access when resolving internal runtime routines.

☁️ FindECallFunctionViaModule

```
FindECallFunctionViaModule(IntPtr hModule, string functionName);
```

Description

Same as `FindECallFunction` but uses a preloaded module handle instead of resolving one by name.

Use Case:

- More efficient when scanning multiple functions in a module.
- Avoids repeated `GetModuleHandle` calls.

🧾 WriteCustomHeader

```
WriteCustomHeader(ushort NewHeader, out ushort OldHeader);
```

Description

Overwrites the DOS signature of the current executable module in memory with a custom header value.

Purpose:
Tags the SXVM process with a known identifier so that friendly rootkits or loaders can detect and **avoid injecting** into it.

Why the DOS Header?
It's a rarely-checked memory field that remains accessible and stable in memory post-launch.

📦 GetModuleByAddress

```
GetModuleByAddress(IntPtr address);
```

Description

Returns the module object (from `Process.Modules`) that contains the given memory address.

Use Case:
Used when verifying whether a found pointer belongs to a specific module, which can be important for validating AOB scans.

Safe & Silent:
Returns `null` if the address is invalid or outside known module ranges.

🔬 SearchAoB

```
SearchAoB(string pattern, string ModuleName);
```

Description

Scans a named module for a pattern (Array of Bytes / AoB), supporting wildcards (`??`).

Pattern Format:

- Bytes: `"E8 ?? ?? ?? ?? 48 8B"`
- `??` represents **wildcards** (SearchAoB function will ignore this byte during a scan)

Internal Flow:

1. Get module by name from current process.
2. Read all bytes into a buffer.
3. Match pattern using sliding window scan.
4. Validate that result is inside the intended module.

Use Case:

- Signature-based memory scanning
- Patching or locating functions with no stable export
- Avoids needing symbols or reflection

Unhooking

The `Unhooker` class is a core utility in **SXVM** that removes **inline hooks** placed by EDRs and antivirus software within loaded system DLLs such as `ntdll.dll` and `kernel32.dll`. It does this by restoring the original `.text` section directly from disk, effectively bypassing most common user-mode API hooks.

🔁 `Unhook(string a)`

```
Unhook("ntdll.dll");
```

Description

Restores the original `.text` section of a DLL (e.g., `ntdll.dll`, `kernel32.dll`) in memory using a mapped, unmodified file copy from disk.

Technique Summary

- Detects system architecture (`System32` vs `SysWOW64`)
- Reads the original DLL from disk
- Maps the clean image into memory
- Copies the original `.text` section over the loaded module
- Restores memory protection

Effect:
Eliminates most user-mode inline hooks (e.g., from EDRs like Defender, ESET, etc.)

Use Case

Used early in execution (e.g., in `Main`) to prepare a clean execution environment and prevent interference with system calls.

🧭 `GetLoadedModuleAddress(string DLLName)`

Description

Retrieves the base address of a loaded module (DLL) by name using `Process.GetCurrentProcess().Modules`.

Use Case:
Needed to identify memory regions associated with target modules for patching or restoration, without using `GetModuleHandle`.

📤 `GetLibraryAddress(string DLLName, string FunctionName)`

Description

Custom implementation of `GetProcAddress` without using the Windows API directly.

Internally Uses:

- `GetLoadedModuleAddress`
- `GetExportAddress`

Use Case:
Used to retrieve function pointers safely even in environments where API calls are hooked or monitored.

📦 `GetExportAddress(IntPtr ModuleBase, string ExportName)`

Description

Manually resolves the address of a function exported by a DLL by walking its **Export Address Table (EAT)**.

What Is the Export Address Table?

The **EAT** is a PE (Portable Executable) data directory used by Windows to list functions exported by a DLL (e.g., `LoadLibrary`, `VirtualAlloc`). By navigating the EAT manually, SXVM avoids relying on `GetProcAddress`.

Key Steps:

1. Locate PE header
2. Read export directory RVA
3. Match function name in `Names` array
4. Resolve ordinal and corresponding RVA
5. Convert to an absolute address

Use Case:
Essential when bypassing user-mode API monitoring or attempting stealthy function resolution.

🧠 Operational Summary

Method	Purpose
`Unhook`	Restores DLL to its unhooked `.text` section
`GetLoadedModuleAddress`	Finds base address of a module in current process
`GetLibraryAddress`	Safe, API-less function resolver
`GetExportAddress`	Manual traversal of PE headers to resolve exports

🔐 Evasion Value

- **API Hook Removal:**
  EDRs often inline-hook `ntdll!NtXxx` APIs. This completely removes those hooks.
- **Anti-Monitoring:**
  Avoids using suspicious API calls (`GetProcAddress` , `LoadLibrary` , etc.)

Hardware Breakpoints

This component disables **AMSI** (Antimalware Scan Interface) using **hardware breakpoints** instead of traditional inline patching or memory writes. This technique is significantly stealthier, avoiding memory scanners, EDR heuristics, and direct tampering flags.

🔧 Function: `Bypass()`

```
HardwareBreakpointAmsiPatch.Bypass();
```

Description

Sets a **hardware breakpoint** on `AmsiScanBuffer` (a Windows AV scan entry point), then hijacks execution at runtime via a vectored exception handler. The exception is intercepted, patched dynamically, and then resumed cleanly.

🧠 How It Works (High Level)

1. **Resolve `AmsiScanBuffer` Address**
    - Uses `GetProcAddress` on `amsi.dll`.
2. **Locate Exception Handler**
    - Uses reflection to find a method with a custom `[HardwareBreakpointAmsiPatchHandlerMethod]` attribute.
3. **Add Vectored Exception Handler**
    - Calls `AddVectoredExceptionHandler` to intercept `EXCEPTION_SINGLE_STEP`.
4. **Configure Hardware Breakpoint**
    - Sets a hardware breakpoint on `AmsiScanBuffer` using `Dr0`–`Dr3` + `Dr7`.
5. **Intercept & Modify Execution**
    - When AMSI is hit, it modifies the return value to `AMSI_RESULT_CLEAN (0)` and skips execution entirely.

🧰 Internal Registers and Structures

💾 `CONTEXT64` Registers Used

Register	Purpose
Dr0–Dr3	Holds the addresses for hardware breakpoints
Dr6	Status register that reports which breakpoint was triggered
Dr7	Control register to enable breakpoints
Rip	Instruction pointer – where execution is redirected to
Rsp	Stack pointer – used to read the return address and arguments
Rax	Return register – used here to return `0` (AMSI clean)

> 💡 Hardware breakpoints are handled *entirely* in CPU registers, meaning **no in-memory patching**, which is a big stealth win.

🧱 `Handler()` Exception Callback

```
[HardwareBreakpointAmsiPatchHandlerMethod]
private static long Handler(IntPtr exceptions);
```

Description

This is the vectored exception handler that executes when the CPU hits the hardware breakpoint.

Behavior:

- Verifies that the exception is a `EXCEPTION_SINGLE_STEP`
- Ensures the faulting instruction is `AmsiScanBuffer`
- Emulates a successful AMSI scan by:
  - Writing `AMSI_RESULT_CLEAN (0)` to the result pointer
  - Returning 0 in `Rax`
  - Skipping over the `AmsiScanBuffer` call (via `Rip`)
  - Adjusting `Rsp` to simulate a clean return

🧪 Enabling the Breakpoint

```
EnableBreakpoint(ctx, pABuF, 0);
```

Description

Configures the `CONTEXT64` structure to install the hardware breakpoint.

Bit-Level Control via `Dr7`

- Each debug register is enabled via bits in `Dr7`
- Bits 0–1, 2–3, etc. for `Dr0`, `Dr1`, etc.
- `SetBits()` handles precise bit manipulation

🔍 Constants

Constant	Value	Meaning
`EXCEPTION_SINGLE_STEP`	`0x80000004`	Triggered on hardware breakpoint hit

Constant	Value	Meaning
`EXCEPTION_CONTINUE_EXECUTION`	`-1`	Resume execution from handler
`EXCEPTION_CONTINUE_SEARCH`	`0`	Pass exception to next handler
`AMSI_RESULT_CLEAN`	`0`	Clean result for AMSI

🛡️ Red Team Value

- **Bypasses AMSI without touching memory**
- **No changes to `amsi.dll` bytes**
- **Avoids EDR inline hook detection**
- **Works with RX (Read-Execute) mapped DLLs, without having to change it's page protections**
- **Evades tools watching for `WriteProcessMemory`, `VirtualProtect`, etc.**

📦 Summary of Function Calls

Function	Purpose
`GetProcAddress`	Resolve `AmsiScanBuffer` address
`AddVectoredExceptionHandler`	Add handler for single-step breakpoints
`SetThreadContext` / `GetThreadContext`	Read/write CPU register state
`Marshal.AllocHGlobal`	Allocate unmanaged memory for `CONTEXT64`
`Marshal.StructureToPtr`	Push C# structure into native pointer memory
`Marshal.ReadInt64` / `ReadIntPtr` / `WriteInt32`	Read/write process memory

HarmonyPatcher

`HarmonyPatcher` is a custom, dynamically-invoked wrapper around the Harmony .NET library. Unlike typical implementations that embed Harmony statically, SXVM **loads it dynamically**

from encrypted resources, decrypts and decompresses it at runtime, and executes patching logic **entirely in memory**.

This offers both stealth and modular control over runtime hooking of .NET methods.

🧩 Key Features

- **No Static References:** Harmony is **not linked at compile time**, instead it's stored encrypted as an embedded resource.
- **Dynamic Memory Loading:** Assembly is decrypted and loaded via `Assembly.Load(...)`, meaning it never touches the disk.
- **Runtime Hooking:** Patches methods using `prefix` and/or `postfix` hooks dynamically.
- **Garbage-Free Memory:** Decryption key is wiped after use to prevent string recovery from memory.

🔑 Internal Workflow

1. 🔒 Decryption & Memory Loading

```
_0Harmony = Assembly.Load(
        API.Decompress(
                API.AESDecrypt(
                        API.ExtractResource("0Harmony.bin"),
_0HarmonyDecryptionKey
                )
        )
);
```

Explanation:

- `0Harmony.bin` is a slightly modified, Compressed and AES-encrypted version of the Harmony DLL.
- This is decrypted using a runtime key, decompressed, then loaded into memory.

Stealth Note:

`_0HarmonyDecryptionKey` is wiped using:

```
_0HarmonyDecryptionKey = null;
GC.Collect();
```

> 🔐 This avoids forensics tools capturing key material from memory dumps.

🧪 Method: `Initialize()`

```
HarmonyPatcher.Initialize();
```

Description

- Loads Harmony if not already loaded
- Generates a random `PatchID`
- Wipes the encryption key securely

⚙️ Method: `Patch(TypeInfo originalMethod, Type prefixMethod, Type postfixMethod)`

```
HarmonyPatcher.Patch(original, typeof(Prefix), typeof(Postfix));
```

Description

Patches a method in memory using Harmony by applying a **prefix** or **postfix** (or both).

Parameters

Parameter	Description
`originalMethod`	Info about the method to patch (type, name, type signature)
`prefixMethod`	Optional class with a static `Prefix()` method
`postfixMethod`	Optional class with a static `Postfix()` method

Example Flow

1. Resolves original method via `AccessTools.Method(...)`
2. Resolves patch methods (if provided)
3. Creates `HarmonyMethod` instances for prefix/postfix
4. Invokes `Patch(...)` via reflection

📦 Struct: `TypeInfo`

```
new TypeInfo(typeof(MyClass), "TargetMethod", new Type[] { typeof(string) });
```

Description

Used to wrap method metadata cleanly for dynamic lookup via `AccessTools`.

Fields:

Field	Purpose
`InternalMethodType`	Type that owns the method
`InternalMethodName`	Name of the method to patch
`InternalTypeParameters`	Array of argument types (used for method resolution)

🧠 Red Team Relevance

Capability	Benefit
In-memory-only Harmony	No disk artifacts; anti-forensics friendly
Dynamic patching of .NET methods	Hook or alter runtime logic of .NET functions or .NET assemblies
Encrypted DLLs & wiped keys	Evades string scanning, memory forensics
Selective patching via reflection	No compile-time references = no static signatures

---

🛠️ Supported Use Cases

- **Hooking analysis methods** (e.g., `Assembly.Load`)
- **Suppressing sandbox checks**
- **Hijacking telemetry/reporting methods**
- **Altering third-party framework logic**
- **Intercepting Windows Forms or WPF behavior**

🔚 Notes

- Requires `0Harmony.bin` to be embedded in the project as a resource.
- Designed to be self-contained, meaning no static dependencies or external libraries needed at runtime.
- Prefix/postfix logic must reside in classes with **static** methods named `Prefix()` or `Postfix()`.

SXVM Hooking Library

The `SXVM` class is a **custom hooking library** that allows native memory locations (like API entry points) to be overwritten with **JMP stubs to managed C# delegates**.

Unlike detouring frameworks like EasyHook or Detours, this tool:

- Avoids external dependencies
- Is self-contained and stealthy
- Hooks native -> managed
- Bypasses static AV detection
- Operates entirely in-process

🔧 Key Features

- 🧬 **Memory introspection** with `VirtualQuery`, `VirtualProtect`
- 🧱 **In-memory delegate builder** for dynamically typed function signatures
- 🚪 **Hotpatch support**: Restore, remove, or permanently apply hooks

- 🪝 **Native-to-managed stubs** using hardcoded JMP opcodes

🔍 Primary Hooking Flow

`Hook(IntPtr Address, Delegate OP_HOOK, bool Permanent, bool Exit)`

```
Hook(address, delegate, true, false);
```

Behavior:

1. 🔍 Queries memory at `Address` to fetch original page protections (via `VirtualQuery`)
2. 🧠 Stores the original 22-byte memory region (`Original_Hook`)
3. 📦 Creates a new **delegate** stub from the user's `Delegate` instance using IL emit
4. 🔗 Injects a **JMP stub** at the target:

```
48 B8 [64-bit address]  ; MOV RAX, EntryPtr
FF E0                   ; JMP RAX
```

5. ✅ Optionally marks as permanent or enables auto-kill on hook exit

---

📦 Memory Handling

`ReadMemoryBlock / WriteMemoryBlock`

Used to safely access raw memory by temporarily elevating memory protections.

Protection Flags:
Uses `PAGE_EXECUTE_READWRITE (0x40)` for editing, and restores original permissions using a custom `ConvertProtectToFlags()` mapper.

🏗️ DelegateTypeBuilder

```
Type delegateType = BuildDelegateType(MethodInfo methodInfo);
```

Purpose:

- Dynamically emits a custom delegate type matching any method signature at runtime
- Uses `System.Reflection.Emit` to define and compile a `MulticastDelegate` with the right signature
- Required for hooking **arbitrary native functions** to managed code

---

🔁 Hook Management

Method	Purpose
`Hook()`	Applies previously set hook
`Unhook()`	Restores original code at hook address
`RestoreVMHook()`	Writes back original bytes
`AddVMHook()`	Writes the custom JMP bytes
`VM_EXIT()`	Emergency failsafe to kill the process

💡 Internals: Inline Hook Details

Hook Format (11 bytes):

Offset	Bytes	Purpose
0–1	`0x48 0xB8`	`MOV RAX, imm64`
2–9	`[8-byte address]`	Pointer to managed delegate
10–11	`0xFF 0xE0`	`JMP RAX`

> This ensures a **64-bit safe jump** from any location to any managed entry point.

⚙️ Supporting Structures

`MEMORY_BASIC_INFORMATION`

Used with `VirtualQuery` to determine the memory protection level of any region before overwriting.

`AllocationProtectEnum`

Enum mirror of Windows page protection constants.

🔐 Red Team Use Cases

- Hook any Windows API and redirect to managed logic
- Intercept DLL exports for sandbox detection, telemetry suppression, or behavior modification

---

⚠️ Caveats and Considerations

- Only works within the current process (no remote process support)
- Hook size assumes 64-bit jump (11 bytes minimum)
- Do **not** hook functions smaller than 11 bytes
- No trampoline support: original function execution must be recreated manually

---

DLLFromMemory

The `DLLFromMemory` class is a custom **manual PE loader** for .NET. It loads a **native DLL** (or EXE) **directly from memory** without loading directly from disk or calling `LoadLibrary`.

This technique is commonly used in:

- **In-memory execution**

🧬 Key Features

- 🧠 Parses and maps PE headers manually (DOS, NT, sections, etc.)
- 🔧 Applies relocations (`.reloc`) if base address doesn't match
- 📥 Manually resolves imports (`GetProcAddress` , `LoadLibrary`)
- 🔐 Executes TLS callbacks & entry point
- 🧼 Can unload from memory cleanly via `Dispose()`

📦 Constructor: `DLLFromMemory(byte[] data)`

```
DLLFromMemory dll = new DLLFromMemory(peBytes);
```

Description

- Accepts a raw DLL/EXE byte array
- Parses and loads the module into virtual memory
- Calls its entry point (if applicable)

🎯 `GetDelegateFromFuncName<T>()`

```
var fn = dll.GetDelegateFromFuncName<MyDelegate>("ExportedFunction");
```

Purpose

- Retrieves a native function exported by the in-memory DLL and wraps it as a **managed delegate**

🧼 `Dispose()` / `Close()`

- Cleans up memory regions and imports
- Calls `DLL_PROCESS_DETACH` if DLL entry point was invoked
- Frees all allocated memory via `VirtualFree`

⚙️ Internals (Summary of Important Steps)

✅ Header Parsing

- Validates DOS and NT headers
- Determines if PE is 32-bit or 64-bit

✅ Memory Allocation

- Allocates memory using `VirtualAlloc`
- Maps headers and section data

✅ Base Relocations

- Adjusts addresses if the module isn't loaded at its preferred base
- Walks `.reloc` table to patch absolute addresses

✅ Import Resolution

- Reads import table (`.idata`)
- Resolves symbols using `GetProcAddress`
- Patches IAT manually

✅ TLS Callbacks

- Executes Thread-Local Storage (TLS) callbacks if defined

✅ Section Protection Finalization

- Sets memory protections for each section based on characteristics
- Discards sections marked as discardable

🔐 Red Team & Evasion Value

Technique	Benefit
Manual PE loading	**Bypasses LoadLibrary / API monitoring**
No file on disk	**Memory-only execution**
Clean unload	Leaves no permanent artifact in memory (if cleaned via `Dispose`)
TLS callback support	Compatible with packers, protectors, and certain implants

---

🚨 Limitations / Considerations

- Does not support packed DLLs without relocation and IAT fixups
- Thread safety is not guaranteed
- Must use correct calling conventions and delegate signatures
- No support for .NET mixed-mode assemblies

🔍 Kaspersky Bypass

This bypass targets **Kaspersky's System Watcher (HIPS)**, which performs behavioral analysis by monitoring the **call stack**, API usage, and module origins of suspicious behavior.

By hijacking a trusted function in a **Microsoft-signed assembly** (in this case, `System.Windows.Forms.dll`), we can execute malicious logic **within a call stack that appears fully legitimate**, effectively **spoofing trust**.

🧠 Why This Works

Kaspersky's HIPS (Host Intrusion Prevention System):

- Analyzes runtime behavior for API abuse (e.g., `AmsiScanBuffer`, `VirtualAlloc`, etc.)
- Tracks the **origin of the call**, especially **return addresses**
- Applies **allowlisting rules** to known Microsoft modules like `System.Windows.Forms.dll`

By executing the payload *as if it's being called from* `MessageBox.Show`, Kaspersky falsely classifies the behavior as benign because:

> “It looks like trusted Microsoft code is doing it.”

🔬 Technical Breakdown

🔨 1. Patch ETW only

```
PatchEDR(false, true, false);
```

- Disables **ETW (Event Tracing for Windows)**
- Leaves AMSI enabled initially to avoid immediate detection during early stages

---

🔍 2. Bypass AMSI with Hardware Breakpoint

```
HardwareBreakpointAmsiPatch.Bypass();
```

- Sets a **hardware breakpoint** on `AmsiScanBuffer`
- Catches execution via vectored exception handler
- Avoids patching AMSI memory directly, which is stealthier

📦 3. Inline Hook MessageBox.Show

```
IntPtr KasperskyAddress =
typeof(System.Windows.Forms.MessageBox).GetMethod("Show", ...);
...
sxvm_kaspersky.Hook(KasperskyAddress, kasperskyBypass, true, false);
```

- Hooks `MessageBox.Show(string)`
- Replaces first few bytes with a JMP to `_KasperskyBypass`
- When `MessageBox.Show("A")` is called, it **immediately redirects** to custom code

---

🧬 4. Run Payload With Spoofed Call Context

```
private static void _KasperskyBypass();
```

- Runs under the **return address of** `System.Windows.Forms.MessageBox.Show`
- Kaspersky sees the stack as:

```
[System.Windows.Forms.dll → AmsiScanBuffer → Some suspicious call]
```

```
Instead of:
```

```
[UserApp.exe → AmsiScanBuffer → Suspicious call]
```

- This call context spoofing is what bypasses Kaspersky's HIPS detection

---

🩹 5. Unhook + Run Payload

```
sxvm_kaspersky.Unhook();
Program.AttachHooks(null);
MethodInfo mi = Assembly.Load(...).EntryPoint;
mi.Invoke(...);
```

- Removes the hook (to restore MessageBox)
- Attaches any internal SXVM hooks (e.g., `GetRawBytes`, `EnvironmentExit`)
- Loads and executes Compressed+AES Encrypted `.NET payload` from memory (`payload.bin`)
- Arguments are forwarded from original `args[]`

📚 Summary

Step	Purpose
Patch ETW	Disable telemetry/logging
Hardware-patch AMSI	Bypass pattern-based AV scanning
Hook `MessageBox.Show()`	Jump into SXVM code with trusted return address
Spoof trusted module context	Bypass Kaspersky HIPS behavior scoring
Load + execute in-memory payload	Fully covert .NET implant execution

---

📦 Obfuscated Trust Chain

```
[System.Windows.Forms.dll] → [SXVM logic] → [Payload Assembly]
```

This call flow *spoofs trust* and creates a “clean” stack trace in tools like:

- Kaspersky System Watcher
- Specific API Monitors
- Specific behavior-based AV engines

🔍 ESET Bypass

This bypass targets **ESET’s EDR & AMSI layer**, specifically the `eamsi.dll` module and its behavioral monitoring hooks. The strategy involves hijacking a trusted **internal ESET function**, rerouting execution into SXVM, and patching any injected hooks (e.g., on network-related CLR calls) with our own logic.

This creates the illusion that **ESET is calling itself**, bypassing internal trust scoring mechanisms.

🎯 Why This Works

ESET applies aggressive runtime monitoring and injects hooks into:

- AMSI scanning (`eamsi.dll`)
- .NET method calls like `System.Net.Sockets.Socket.Connect`
- Thread activity and memory allocations

Their detection engines use:

- **Call origin verification**
- **Memory integrity checks**
- **Whitelisted return address paths**

By **replacing a known internal ESET function (** `atexit` **) in** `eamsi.dll` **)** with our own code and invoking it, we make the **call chain appear internal**, tricking ESET into trusting the behavior.

🧠 Technique Breakdown

1. 🔩 Patch EDR + AMSI

```
PatchEDR(true, true, false);
```

- **AMSI + ETW** are disabled
- This removes both memory scanning and telemetry hooks

---

2. 🔍 Locate `atexit()` in `eamsi.dll`

```
IntPtr ESETAddress = SearchAoB("48 83 EC 28 E8 BB FF FF FF 48 F7 D8 1B C0 F7
D8", "eamsi.dll");
```

- Uses an **Array of Bytes (AoB)** scan to locate the `atexit` function
- Found by analyzing `eamsi.dll` in **IDA or Ghidra**
- This function is **trusted** because it originates from an official ESET DLL

---

3. 🪝 Hook ESET's `atexit`

```
sxvm_eset.Hook(ESETAddress, eSETBypass, true, false);
```

- Places a JMP to `_ESETBypass()` at the start of `atexit()`
- This lets SXVM **run its logic under the trusted return address** of `eamsi.dll`

---

4. 📞 Invoke Hijacked Function

```
ESETHijack hijack = Marshal.GetDelegateForFunctionPointer<ESETHijack>
(ESETAddress);
hijack();
```

- Calls the now-hooked function

- From ESET’s perspective, the call path is **entirely internal**

---

5. 🧩 `_ESETBypass()` - Main Execution Logic

```
Program.AttachHooks(null);
HarmonyPatcher.Patch(... Socket.Connect ...)
```

Actions:

- **Unhooks** our redirection from `atexit`
- **Attaches SXVM hooks** (`GetRawBytes`, `Environment.Exit`, etc.)
- **Patches** `Socket.Connect()` using in-memory Harmony to **overwrite** ESET’s CLR hook
  - This nullifies ESET’s .NET network monitoring
  - Our patch simply returns `true`, skipping any ESET logic
- Loads and executes `payload.bin` (decrypted & decompressed C# binary)

Summary of Behavior

Step	Description
Patch AMSI/ETW	Removes scanning & logging
Locate `eamsi.dll` function	Finds trusted function ESET itself uses
Hook `atexit()`	Redirects ESET function to SXVM logic
Spoof internal origin	Calls appear to come from ESET → avoids scoring or blocking
Patch CLR `Socket.Connect`	Bypasses .NET network inspection logic injected by ESET
Load and run `payload.bin`	Executes in-memory C# implant

🧠 Red Team Benefit

- **Trusted call origin spoofing**: Completely bypasses behavior scoring
- **Bypass CLR-level AV hooks**: Hooks like `Socket.Connect` can be fully suppressed

🧪 Stack Trace Spoofing Example

```
[eamsi.dll!atexit()] → [SXVM logic] → [payload.bin]
```

From ESET's point of view, this is:

> “Internal, trusted code executing internal logic.”

Which **fails to trigger alerts** due to **allowlisted origins**.

🔍 BitDefender Bypass

This bypass is designed to defeat **Bitdefender’s Advanced Threat Protection (ATP)**, which goes **far beyond AMSI and ETW**. In the paid editions, BitDefender injects **native-level hooks** into the .NET CLR’s internal functions, most notably:

> `nLoadImage` : The native backend function behind `Assembly.Load(...)`

BitDefender monitors and scans **byte arrays passed into the CLR**, blocking payloads **before** .NET ever begins execution, even when **AMSI and/or ETW is patched or disabled**.

🧠 Why This Works

BitDefender’s hook is placed in `clr.dll` directly, not through .NET reflection or user-mode APIs. It targets `nLoadImage`, which handles all `Assembly.Load(...)` or `Assembly.Load(byte[])` routines in .NET.

To evade this:

1. We **find the real memory address of** `nLoadImage` in `clr.dll`
2. We **manually map a clean copy of** `clr.dll` from disk (unhooked)
3. We **copy the clean bytes of** `nLoadImage` from the mapped image
4. We **overwrite BitDefender’s hooked version** in memory

> This effectively removes Bitdefender's native hook *without* breaking the CLR or triggering anti-debugging responses.

🔬 Technical Breakdown

1. 🔧 Patch ETW (Only)

```
PatchEDR(false, true, false);
```

- On BitDefender, AMSI is defeated via a hardware breakpoint
- Only ETW patching is needed at this stage

---

2. ⚙️ Bypass AMSI

```
HardwareBreakpointAmsiPatch.Bypass();
```

- Ensures pattern-based scans don't get caught
- AMSI alone isn't enough, but still useful to patch out

---

3. 📥 Locate `clr.dll`

```
if (module.ModuleName == "clr.dll")
{
        ...
}
```

- The **Common Language Runtime (CLR)** is the core of the .NET runtime
- `clr.dll` hosts many internal methods, including:
    - `nLoadImage` - native image loader
    - `Garbage Collection`, etc.

---

4. 🔎 Find Hooked `nLoadImage`

```
IntPtr RealCLRAddress = FindECallFunction("clr.dll", "nLoadImage");
```

- This address points to the **BitDefender-hooked** version

- It’s patched by ATP on load

---

5. 🧬 Load Clean Copy of CLR

```
DLLFromMemory MemCLR = new DLLFromMemory(File.ReadAllBytes(CLRFilePath));
```

- Maps `clr.dll` manually from disk into memory
- **No Windows loader calls**, bypasses ATP’s load monitoring, otherwise BitDefender would detect the dll being loaded and place hooks on it
- Used only to extract clean function bytes (we don’t call into it)

---

6. 🧠 Copy Clean Function Bytes

```
IntPtr MemCLRAddress = FindECallFunctionViaModule(MemCLR.pCode, "nLoadImage");
byte[] CLRPatch = ReadMemoryBlock(MemCLRAddress, 30);
```

- Reads **30 unmodified bytes** from clean `nLoadImage`
- This range is sufficient to overwrite BitDefender’s hook safely

---

7. 🛠 Overwrite the Hook

```
WriteMemoryBlock(RealCLRAddress, CLRPatch, (uint)CLRPatch.Length);
```

- **Directly patches memory** at `nLoadImage` in the real `clr.dll`
- Removes BitDefender’s native trampoline or detour
- Execution of `Assembly.Load()` now resumes without inspection

---

8. 🔚 Clean Up

```
MemCLR.Close();
GC.Collect();
```

- Releases manually mapped resources
- Clears decryption keys and resources to minimize footprint and memory usage

---

9. 🚀 Load Payload Safely

```
MethodInfo mi = Assembly.Load(...).EntryPoint;
mi.Invoke(...);
```

- BitDefender has no hook left to intercept the load
- Payload is executed safely and undetected

🧠 Summary Table

Component	Purpose
`nLoadImage`	Native CLR function behind `Assembly.Load`
`FindECallFunction`	Locates native method by name in memory
`DLLFromMemory`	Manual PE loader to map clean `clr.dll` from disk
`ReadMemoryBlock`	Reads unmodified bytes of clean `nLoadImage`
`WriteMemoryBlock`	Patches over hooked version in real `clr.dll`
`HardwareBreakpointAmsiPatch`	Stealth AMSI bypass to prevent detection of memory patches on `AmsiScanBuffer`

🚩 Red Team Value

Feature	Impact
Manual mapping of CLR	Avoids loader visibility to AV/EDR
Native hook overwrite	Removes even invisible hooks placed deep in system modules
AMSI + ETW patch combo	Comprehensive passive evasion
Memory-only payload	No disk writes, no file IOCs

🧪 Visualized Flow

```
[Real clr.dll] — (hooked by ATP) → nLoadImage
|
├→ Replace nLoadImage body with bytes from:
|     [Manual clone of clr.dll]
|       └→ Execute Assembly.Load(...) - undetected
```

❌ Why You Can't Safely Copy `.text` Section of `clr.dll` to Unhook It

🔧 The Temptation

You might think:

> "Why just patch 30 bytes of `nLoadImage` when I can copy the entire `.text` section of `clr.dll` from a clean version and overwrite the in-memory version to wipe **all** BitDefender (or other AV) hooks?"

While logical, this **breaks the CLR** and crashes the process. Here's why.

🧠 How the CLR Internals Work

`.text` is the **main code section** of `clr.dll`. It includes:

- Critical function implementations (e.g., `nLoadImage`)
- JIT interface logic
- Internal state tracking
- Thread-local and domain-local offsets
- Code stubs dynamically modified at runtime

☠️ Why Overwriting the `.text` Section Crashes the CLR

1. 🧬 Self-Modifying Behavior

The CLR **modifies its own** `.text` **section** at runtime, particularly:

- JIT-related stubs
- NGEN dispatch tables
- Internal hooks used for garbage collection, exceptions, appdomain isolation

Overwriting these with a static version **invalidates** instruction pointers and corrupts return addresses across multiple threads.

---

2. 🧠 Live Pointers and Executing Threads

`.text` contains **active function frames** that are:

- Currently being executed
- Pinned in the call stack across multiple threads

If you overwrite these in-place:

- Threads return to addresses that now point to **mismatched or unmapped instructions**
- This results in:
    - `AccessViolationException`
    - Invalid opcode exceptions
    - Silent crashes or abrupt process termination

✅ Safer Alternative (What SXVM Does)

Instead of bulk-patching `.text`, SXVM:

- Finds the **specific native function hook** (e.g., `nLoadImage`)
- Manually maps a clean copy of `clr.dll`
- Copies **only the minimal instruction window** (e.g., first 30 bytes)
- Overwrites **only** the function's prologue, preserving surrounding state

> This is **surgical** unhooking, enough to defeat ATP, but not enough to destabilize the CLR.

🔐 Summary

Reason	Impact
Live instruction frames	Threads crash on returning to now-invalid code
CLR self-modifies `.text`	Code you overwrite may have been dynamically patched internally

---

💡 TL;DR

Overwriting the full `.text` of `clr.dll` is like deleting the memory of a live process mid-execution. It breaks code, context, and execution across the entire runtime.

🔍 Avast Bypass

This bypass targets **Avast's runtime memory protection**, which actively guards key inspection points such as:

- **AMSI** (`AmsiScanBuffer`)
- **ETW** (`EtwEventWrite`, `NtTraceEvent`)

Unlike most AVs, **Avast crashes or terminates the process** if these functions are patched using normal memory writing (e.g., pointer casts or `Marshal.Copy`). To bypass this, we must patch them **exclusively via** `WriteProcessMemory`, which appears to be **"trusted"** by Avast's internal hooks.

🧠 Why This Works

There are two strong theories:

Theory 1: 🧵 Kernel-Level Trap / Write Surveillance

- Avast may place **kernel-mode callbacks** or hooks on key memory pages (e.g., `.text` section of `amsi.dll`)
- If a memory write occurs without going through a **recognized system call (like `WriteProcessMemory`)**, Avast **flags it as tampering** and crashes the process

Theory 2: 🧲 Exception Handler Watchdogs

- Avast might set up a **guard page** or use **vectored exception handlers (VEH)** to catch direct memory writes
- By using `WriteProcessMemory`, the write goes through the **Windows kernel** and bypasses VEH traps

> In either case, `WriteProcessMemory` is "seen" and trusted by Avast, while unsafe manual writes are treated as hostile.

🧬 Technical Breakdown

1. 🔒 Patch AMSI and ETW (Safely)

```
PatchEDR(true, true, true);
```

- The third `true` flag ensures `UseWriteProcessMemory = true`
- Internally calls:

```
WriteProcessMemory(handle, addr, patch, ...);
```

- Avoids any crash or panic during patching

---

2. 🧠 Attach SXVM Hooks

```
Program.AttachHooks(null);
```

---

3. 📦 Load and Execute Payload

```
MethodInfo mi = Assembly.Load(...).EntryPoint;
mi.Invoke(null, new object[] { args });
```

🔐 Summary

Component	Role
`WriteProcessMemory`	Ensures Avast sees the patch as legitimate
`PatchEDR(true, true, true)`	Patches AMSI + ETW via trusted API
`Program.AttachHooks()`	Installs .NET stealth layer
`Assembly.Load(...)`	Loads the real payload undetected from memory

🚩 Red Team Insight

Technique	Reason
Avoids direct memory writes	Prevents Avast crash on AMSI/ETW patching
Uses system APIs	Evades watchdogs or trap exceptions
Memory-only execution	Keeps payload invisible to filesystem-based detection

---

💡 TL;DR

> Avast **trusts memory patching** when it goes through `WriteProcessMemory`, but **crashes** if done directly. This bypass leverages that quirk to neutralize AMSI + ETW _without triggering Avast's possible kernel or VEH-level tripwires.

🔍 Default Bypass

The `Default` bypass is the **standard execution flow** used when:

- The target AV is **Windows Defender (WD)**
- The system has **no AV**, or
- The detected AV is **unsupported or non-invasive**
- A known AV is detected, but it **does not require a special bypass path**

This routine assumes **no low-level hooks** on the CLR or AMSI, making it ideal for **typical Windows environments**, home-user targets, or generic sandbox escape attempts.

🧠 Why This Works

Windows Defender (and many weaker AVs):

- Rely heavily on **AMSI and ETW** for detection
- Do **not hook native CLR functions** like `nLoadImage`
- Rarely monitor low-level memory calls unless real-time scanning is enabled

By simply **patching AMSI and ETW**, we remove their inspection vectors. Once those are disabled:

> Defender is **effectively blind** to in-memory .NET payloads.

🧬 Technical Breakdown

1. 🔒 Patch AMSI + ETW

```
PatchEDR(true, true, false);
```

- First `true` = patch AMSI (e.g., `AmsiScanBuffer`)
- Second `true` = patch ETW (`EtwEventWrite`, `NtTraceEvent`)
- Third `false` = use direct memory writes (not `WriteProcessMemory`), which is fine for Defender

---

2. 🧠 Attach SXVM Runtime Hooks

```
Program.AttachHooks(null);
```

Hooks internal .NET functions for added stealth:

- Prevents module byte dumping (`GetRawBytes`)
- Overrides `Environment.Exit` with `Process.Kill` for cleaner exit

---

3. 🧬 Load Payload In-Memory

```
MethodInfo mi = Assembly.Load(...).EntryPoint;
mi.Invoke(null, new object[] { args });
```

- Payload is loaded **entirely from memory**
- Extracted from resource (`payload.bin`)
- AES-decrypted and decompressed before execution

Since **AMSI and ETW are patched**, Defender **cannot scan or log** this process.

✅ Summary Table

Component	Role
`PatchEDR(true, true, false)`	Disables AMSI + ETW via direct memory patching
`Program.AttachHooks()`	Installs internal anti-analysis / evasion patches
`Assembly.Load(...)`	Loads and executes encrypted in-memory payload

💡 Use Cases

- 💻 **Windows Defender** targets
- 🧪 **Unknown AV** with no observable behavior
- 🛠 **Offline sandbox** or **test VMs**
- 🚨 **Fallback path** if a named AV bypass fails

🧠 Red Team Insight

Behavior	Implication
Does not patch `nLoadImage`	Not needed for Defender, no native CLR hooks present
Uses direct memory write	Works fine, Defender doesn't trap on patch origin
Minimal complexity	Smaller footprint, lower chance of detection

---

🧪 Example Execution Flow

```
[Patched AMSI + ETW] → [SXVM Hooks Active] → [In-memory Payload Execution]
```

Defender sees no alerts, no bytes to scan, and no ETW telemetry, because it **can't**.

⚠️ Limitations of SXVM

Despite being a powerful in-memory EDR/AV bypass utility, **SXVM is not a full end-to-end solution**. It is specifically designed to neutralize runtime defenses, not file-based or pre-execution detection.

❌ Not Scantime FUD (Scantime Fully Undetectable)

> SXVM is **not Scantime FUD at compile time**, the binary **must be obfuscated** before execution.

🔍 Why:

- **Most AVs (Defender, BitDefender, Avast, etc.)** scan the payload's raw bytes **before** it's loaded into memory via `Assembly.Load(...)`.
- If those bytes contain recognizable IL patterns, strings, metadata, or opcodes, the AV will **quarantine or block** the binary **before SXVM has a chance to run**.

✅ What SXVM *does*:

Once the binary is successfully loaded via the `Assembly.Load(...)` call, **SXVM takes over** and:

- Patches AMSI / ETW
- Unhooks .NET and native API traps
- Bypasses behavioral scoring via call origin spoofing
- Ensures undetected runtime execution across multiple AVs

🔐 Obfuscation Requirement

To be effective, the binary (`SXVM`) **must be**:

- Obfuscated in a **way not flagged by the target AVs**
- Free of suspicious strings or method names (`ExecutePayload` , `DecryptionKey` , etc.)
- Ideally compressed + encrypted (AES, XOR, etc.), then decrypted in memory

> 📌 Obfuscation must bypass the **AVs you're trying to use SXVM on**, otherwise the loader dies before SXVM can run.

📋 Other Notable Limitations

Limitation	Details
No Kernel EDR Evasion	SXVM is a **user-mode bypass**; does not defeat kernel drivers or hypervisor-based telemetry (e.g., Kaspersky KLIF, Sophos Intercept X kernel mode)
.NET-only Payloads	SXVM is designed for **in-memory .NET execution** via `Assembly.Load(byte[])` . Native payloads must be manually handled.
Requires Valid .NET Runtime	Target system must have .NET CLR (4.5.1+) installed and loaded (pre-installed on all Windows 10 and 11 machines at the time of writing); does not auto-deploy CLR
Sandbox-Sensitive	In heavy instrumented sandboxes (e.g., Cuckoo with AMSI enabled), early-stage detections (file write, entropy checks) may still trigger
Doesn't Persist	SXVM does not implement persistence, it's a runtime execution tool, not a full implant/dropper

---

🧠 TL;DR

> **SXVM bypasses runtime defenses. You must handle scantime evasion of the SXVM binary yourself.**

Once `Assembly.Load(...)` can execute without detection, **SXVM will handle the rest**.
