resillion

Assure. Secure. Innovate.

When Internet Information Services (IIS) becomes an execution platform

Author : Lawrence Amer,
Red Team specialist, Resillion

Introduction

Enterprise security controls have evolved significantly over the past decade. Traditional malware that writes executables to disk, spawns suspicious child processes or abuses well-known scripting engines is increasingly detected by modern Endpoint Detection and Response (EDR) and behavioural monitoring systems.

Yet one highly trusted platform remains widely deployed and is often overlooked in post-compromise scenarios: **Internet Information Services (IIS).**

IIS hosts internal portals, APIs, SharePoint deployments, authentication gateways and legacy business applications across many enterprise environments. At the process level, it runs inside a Microsoft-signed trusted process (w3wp.exe), commonly operates with elevated privileges and often sits inside trusted network zones. In many organisations, web root permissions are more permissive than intended, creating a powerful opportunity for post-exploitation abuse.

This paper examines the research we conducted to understand what happens when an attacker gains write access to an IIS web root.

Rather than dropping a traditional payload to disk, we demonstrate how a single Active Server Pages Extended (ASPX) page can act as a fully reflective native Dynamic Link Loader (DLL). The payload never exists as a standalone file. Instead, it is parsed, mapped, relocated and executed entirely in memory within the trusted IIS worker process. By reimplementing key parts of the Windows Portable Executable (PE) loader in C#, we transform IIS into a stealthy, in-memory execution platform suitable for lateral movement.

This is not simply a web shell demonstration. It explores how modern detection strategies can (and can't) surface this activity. It examines how to:

- reproduce Windows loader behaviour in managed code
- understand PE internals and x64 exception handling
- minimise static detection surface
- blend malicious execution into trusted application behaviour.

For red team operators or offensive security engineers, it shows how a commonly deployed enterprise web server can be transformed into a stealthy, in-memory execution platform to perform lateral movement.

For SOC analysts or cyber defenders, it means a shift in detection strategy. When payloads are fileless and run inside trusted processes, visibility must move beyond the filesystem.

IIS is not just a web server. In the wrong hands, it becomes a compelling target for post-exploitation techniques.

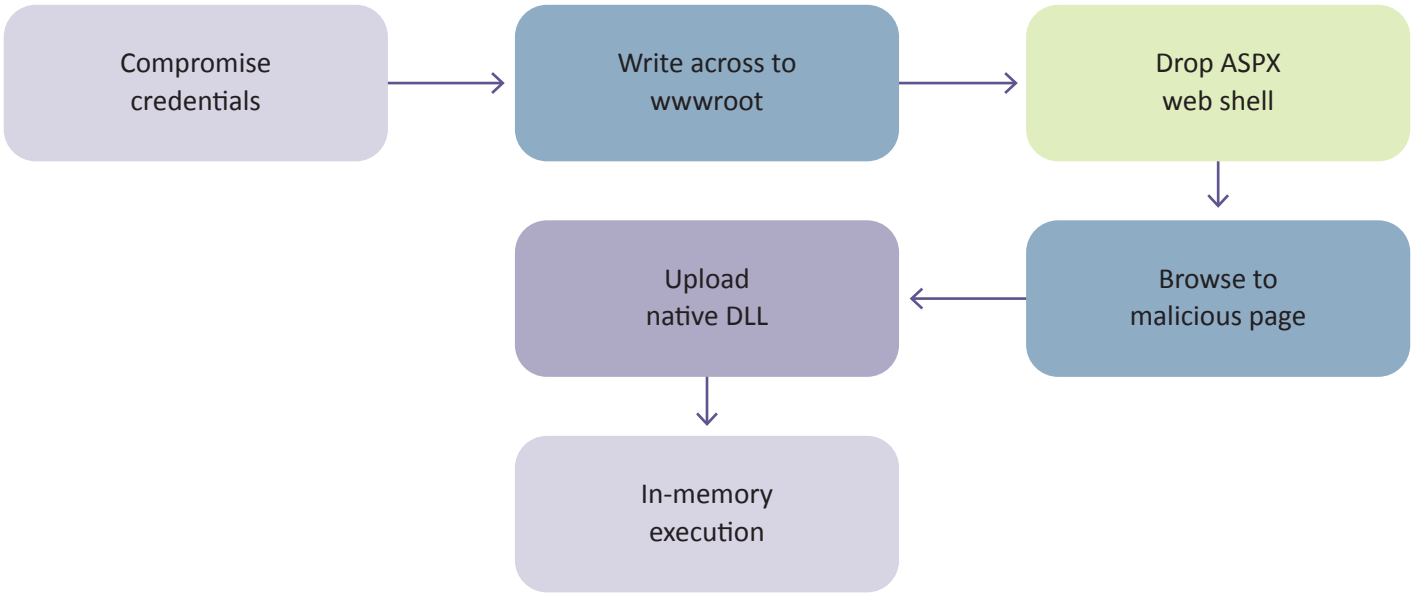
Why IIS is a prime lateral movement target

In many enterprise environments, **Internet Information Server (IIS)** is everywhere. It hosts internal portals, API gateways, SharePoint instances and legacy line-of-business applications.

From a red team perspective, IIS is a compelling target for lateral movement for several reasons:

- **Write access to wwwroot is often achievable**
  Service accounts running web applications frequently have write permissions to their own web root. Compromising a database with xp_cmdshell, exploiting a file upload vulnerability or reusing credentials from a compromised admin workstation can all provide write access.
- **ASPX pages are compiled and executed on first request**
  There is no separate deployment step. Drop the file, browse to it and the code runs.
- **Execution occurs inside the w3wp.exe process context**
  This is a trusted, Microsoft-signed process. It legitimately loads DLLs, makes network connections and performs complex operations - so malicious activity blends in with normal IIS behaviour.
- **Full trust is the default**
  Unless the application pool has been explicitly configured for partial trust (which is rare in modern deployments), ASP.NET code has full access to P/Invoke and unmanaged code.
- **Limited AV/EDR visibility at delivery time**
  The HTTP request that delivers payload bytes into an ASPX page appears as normal web traffic. The payload bytes do not touch the filesystem as a standalone file. However, the IIS server's AV/EDR still has visibility into process behaviour, so it is important to carefully consider what is executed and how it runs.

The attack chain is straightforward:



Phase 1: The basic approach

During our research, we created several loaders and performed tests, collecting results across different aspects and environments.

The first version was intentionally simple. The goal was to prove the concept: **can we load and execute an arbitrary native DLL from an ASPX page?**

How v1 worked

The v1 loader followed the standard Windows API approach:

- Accept a DLL via file upload or Base64 input.
- Write it to a temporary file on disk (%TEMP%\{guid}.dll).
- Call LoadLibrary() on the temporary path.
- Call GetProcAddress() to locate the exported function.
- Marshal a delegate and invoke it.
- Clean up: FreeLibrary(), then delete the temporary file.

The P/Invoke declarations were minimal - only LoadLibrary, GetProcAddress, FreeLibrary and GetLastError.

Function signatures were handled through a small set of delegate types covering common calling conventions (cdecl, stdcall) and common return types (void/int/string), both with and without arguments.

Why v1 was insufficient

While version one worked well functionally, it introduced several operational problems that increased the likelihood of detection:

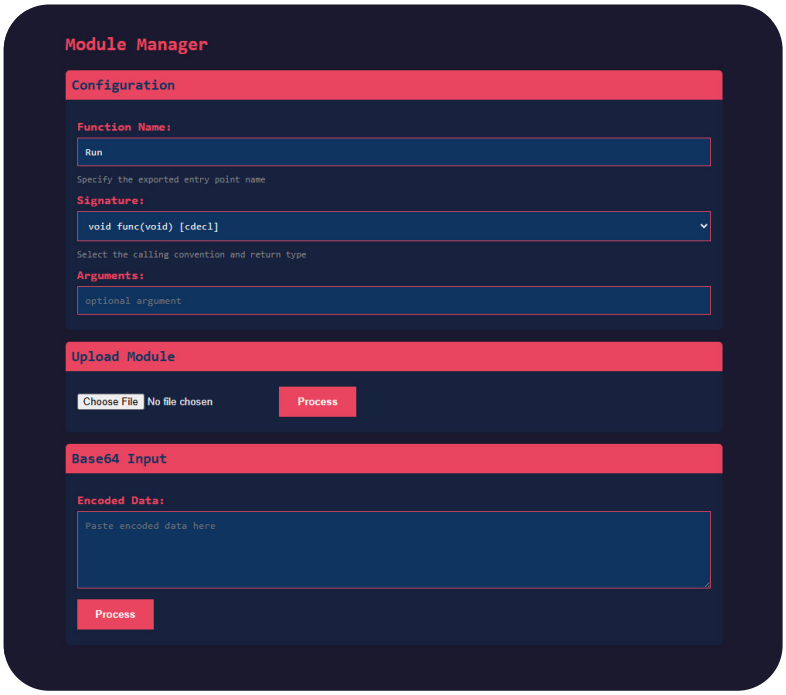
Issue	Impact
DLL written to %TEMP%	AV/EDR performs real-time file scanning on write events
LoadLibrary call on untrusted path	Logged by Sysmon (Event ID 7), ETW and EDR
Temp file exists on disk (however briefly)	Forensic artifact, recoverable even after deletion
Predictable file naming pattern	.dll in temp is suspicious
Standard DllImport signatures	Static analysis of the ASPX reveals intent

The temporary file was the biggest issue. Even though it was deleted in the finally block, the window between File.WriteAllBytes and File.Delete was enough for real-time scanning to inspect and flag the payload.

From a forensic point of view, the file could also be recovered from the MFT or volume shadow copies long after deletion.

To improve the approach, we needed to eliminate disk I/O entirely.

Phase 2: Going reflective – Phantom loader



The core idea behind reflective loading is to replicate everything LoadLibrary does - but manually, using only memory operations.

In this approach, the Dynamic Link Library (DLL) bytes are delivered from an HTTP request directly into executable memory (specifically inside w3wp.exe) without ever writing the payload to disk.

To achieve this, a custom Portable Executable (PE) loader was implemented from scratch in C#. The loader performs the steps below, in order:

1. Parse the DOS and NT headers to validate the PE file.
2. Allocate a memory region large enough to hold the entire image.
3. Copy each PE section into its correct memory address.
4. Process base relocations (since the image will not load at its preferred base address).
5. Resolve all imports by loading required dependencies and patching the Import Address Table (IAT).
6. Handle any delay-loaded imports.
7. Register exception handlers (critical for x64 architectures).
8. Execute TLS callbacks.
9. Apply the correct memory protections for each section.
10. Call the entry point (DllMain) to initialise the C Runtime (CRT).
11. Flush the instruction cache.

Each of these steps must be implemented correctly. Any mistake can crash the w3wp.exe process.
For this reason, it's strongly recommended to test the payload thoroughly in a controlled test environment before deploying it, to avoid disrupting the IIS worker pool process.

Building the Portable Executable (PE) loader in C#

1. PE header parsing

The first challenge was defining all the PE structures in C#. The PE format is deeply nested:

`DOS Header -> NT Signature -> File Header -> Optional Header -> Section Headers -> Data Directories -> Sections`

Each structure required a `[StructLayout(LayoutKind.Sequential)]` definition with precise field sizes and alignment.

Even a small mistake in a single field offset would shift everything that follows, causing every subsequent parse to fail.

We defined structures for:

- IMAGE_DOS_HEADER (64 bytes, the e_lfanew field points to the NT headers)
- IMAGE_FILE_HEADER (machine type tells us x86 vs x64)
- IMAGE_OPTIONAL_HEADER32 / IMAGE_OPTIONAL_HEADER64 (different sizes!)
- IMAGE_SECTION_HEADER (name, virtual address, raw data pointer, characteristics)
- IMAGE_IMPORT_DESCRIPTOR, IMAGE_EXPORT_DIRECTORY, IMAGE_BASE_RELOCATION
- IMAGE_TLS_DIRECTORY32/64, IMAGE_DELAYLOAD_DESCRIPTOR, RUNTIME_FUNCTION

2. Memory allocation

The image requires a contiguous block of memory at least `SizeOfImage` bytes in size. Initially, this memory is allocated with read-write-execute (RWX) permissions; permissions are tightened later once setup is complete.

```
module.Base = VirtualAlloc(IntPtr.Zero, (UIntPtr)sizeOfImage, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
```

We then copy the PE headers and each section to their correct offsets within this allocation.

3. Base relocations

Because we cannot control where `VirtualAlloc` places the allocated memory, the image will almost certainly not be loaded at its preferred base address. As a result, every absolute address embedded in the binary must be adjusted by the delta between the preferred base and the actual base.

This adjustment is handled through the relocation table.

The relocation table is organised into blocks, each containing multiple relocation entries. Every entry specifies:

- A page offset.
- A relocation type.

The relocation type determines how the value should be patched:

- On **x86**, the type is IMAGE_REL_BASED_HIGHLOW, which patches a 32-bit value.
- On **x64**, the type is IMAGE_REL_BASED_DIR64, which patches a 64-bit value.

During processing, each referenced address is updated by adding the base delta, ensuring the image functions correctly at its new memory location.

```
long delta = (long)module.Base - (long)preferredBase;

// For each relocation entry:

if (type == IMAGE_REL_BASED_DIR64)

{

    IntPtr patchAddr = IntPtr.Add(module.Base, blockVA + offset);

    long value = Marshal.ReadInt64(patchAddr);

    Marshal.WriteInt64(patchAddr, value + delta);

}
```

4. Import resolution

The import directory contains a list of required DLLs along with the functions that must be resolved from each one.

For every imported DLL, we call the real `LoadLibrary` to get a handle. Then, for each imported function, we call `GetProcAddress` and write the resolved function pointer into the Import Address Table (IAT).

```
IntPtr hDll = LoadLibrary(dllName);

string funcName = Marshal.PtrToStringAnsi(IntPtr.Add(hintNameAddr, 2));

IntPtr funcAddr = GetProcAddress(hDll, funcName);

Marshal.WriteInt64(IntPtr.Add(thunkAddr, offset), (long)funcAddr);
```

Imports can be resolved in two ways:

- **By name** – using the function's string name.
- **By ordinal** – using the function's numeric export ordinal instead of a name.

Each entry must be resolved and patched correctly; otherwise, the module will fail when it attempts to call an unresolved function.

The import directory contains a list of DLLs and the functions needed from each. For each imported DLL, we call `LoadLibrary` (the real one) to get a handle, then `GetProcAddress` for each function and write the resolved address into the Import Address Table (IAT): Imports can be by name or by ordinal.

5. x64 exception handling

On x64 Windows, structured exception handling (SEH) relies on **table-based unwinding**. This means the operating system must know the layout of every function's stack frame to correctly unwind the stack when an exception happens.

If a DLL's function table isn't registered, any exception inside that DLL can **crash the entire process**.

The registration is done with:

```
RtlAddFunctionTable(exceptionTable, entryCount, (ulong)module.Base);
```

Skipping this step leads to **mysterious crashes in ntdll.dll** during exception dispatch - one of the most difficult bugs we've had to deal with.

6. Section protections

Once all patching is complete, we **reinforce memory protections by section:**

- **Code sections** → PAGE_EXECUTE_READ
- **Data sections** → PAGE_READWRITE
- **Read-only sections** → PAGE_READONLY

This approach is important for two reasons: some DLLs **verify their own section protections**, and it also **reduces the risk of detection** by avoiding large regions with RWX permissions.

CRT initialisation with DllMain entrypoint

This was a subtle but critical issue. When Windows loads a DLL via `LoadLibrary`, the OS automatically calls `DllMain` (DLL_PROCESS_ATTACH). This isn't just a notification. For DLLs compiled with MSVC, the actual entry point is **_DllMainCRTStartup**, which:

- Initialises the C runtime (heap, stdio, locale)
- Runs C++ global constructors
- Sets up thread-local storage (TLS) for the CRT
- Then calls the user-defined DllMain (if present)

Skipping this step means that **any call to malloc, printf, sprintf, new, or other CRT functions** in the loaded DLL can crash or behave unpredictably. Even the exported default function you want to call (e.g., Run) **depends on the CRT being initialised**.

We noticed some payloads still crash despite this, and we recommend **opening an issue on the GitHub repository** if you find the same behaviour during testing.

Solution: Shellcode-based entry point invocation

Initially, we tried calling `DllMain` via a C# delegate `(Marshal.GetDelegateForFunctionPointer)`, but this triggered **access violations** on modern Windows. The CLR's marshaling layer adds instrumentation around delegate calls that interferes with the entry point's expectations.

The reliable solution was to **build a minimal x64 calling stub at runtime**:

```
mov rcx, <hModule>       ;
First arg: module base

mov edx, <reason>        ;
Second arg: DLL_PROCESS_ATTACH (1)

xor r8, r8               ;
Third arg: NULL

mov rax, <entrypoint>    ;
Target address

jmp rax                  ;
Transfer control
```

This stub is allocated in a **separate executable memory region** and executed via `(CreateThread + WaitForSingleObject)`. Using a thread avoids CLR interference entirely, letting the stub run on a **clean native thread** with no managed frames on the stack.

For context, CLR (Common Language Runtime) marshaling is the process of converting data representations between managed code (.NET) and unmanaged code (native C/C++ and Win32 APIs).

Phase 3: Hardening the loader against detection

At this stage, we had a **working reflective loader** - the DLL payload never touches the disk. However, the loader itself (loader.aspx) resides on the filesystem and is **packed with signatures** that would make any antivirus analyst sit up and take notice.

To make things clearer:

Surface	Threat	Our mitigation
Network	Scanners, unauthorised users finding the page	Access gating (token, host pinning, IP whitelist, User Agent (UA) blocking, fake 404 page)
Filesystem	Windows Defender static scan, EDR file analysis	Dynamic API resolution, string obfuscation, identifier renaming
Runtime	Antimalware Scan Interface (AMSI), behavioural monitoring, ETW	Inherent to reflective loading; further hardening per-payload

1. Reducing the static signature footprint

This was the most important change for AV evasion.

In v1 and v2 of the loader, we had more than a dozen `[DllImport("kernel32.dll")]` declarations referencing well-known Windows APIs - many of which are heavily signatured by antivirus engines.

To reduce detection risk, we needed a more subtle approach.

2. Dynamic API resolution

Starting with dynamic API resolution, the new version of the Phantom loader keeps only two static `DllImport` declarations - the absolute minimum required for bootstrapping:

```
[DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Unicode)]
static extern IntPtr LoadLibrary(string n);
[DllImport("kernel32.dll", SetLastError = true, CharSet = CharSet.Ansi)]
static extern IntPtr GetProcAddress(IntPtr h, string n);
```

These two APIs are used by countless legitimate .NET applications and are not suspicious on their own.

All other Windows APIs such as `VirtualAlloc`, `VirtualProtect`, `VirtualFree`, `CreateThread`, `FlushInstructionCache`, `WaitForSingleObject`, `CloseHandle`, `RtlAddFunctionTable`, and others are resolved dynamically at runtime using `GetProcAddress`.

By avoiding static imports for these commonly signatured functions, the loader reduces its static footprint.

This makes it significantly harder for AV scanners to identify potentially malicious behaviour through straightforward signature matching, compared to the earlier approach that exposed every API directly via DllImport.

3. String obfuscation via reversal

One common detection indicator used by anti-malware engines is the presence of recognisable function names, string patterns and other static artifacts. Even lightly obfuscated strings can still be easy targets for static scanners.

To reduce this exposure, API names are stored **reversed** in the source code and decoded back to their original form at runtime.

```
static string _d(string s)
{
    char[] c = s.ToCharArray();
    Array.Reverse(c);
    return new string(c);
}
// “VirtualAlloc” stored as “collAlautriV”
static D_VA pVA() {
    if (_pVA == null) _pVA = _r<D_VA>(HK(), “collAlautriV”);  return _pVA; }
```

Even the DLL names are reversed:

- kernel32.dll is stored as “lld.23lenrek”
- ntdll.dll is stored as “lld.lldtn”

A static scan of the file reveals no occurrences of `VirtualAlloc`, `CreateThread`, `VirtualProtect`, or any other commonly flagged suspiscious API names.

4. Runtime-computed constants

Suspicious hexadecimal constants such as `0x40 (PAGE_EXECUTE_READWRITE)` and `0x1000 (MEM_COMMIT)` are well-known detection indicators.

To avoid exposing these recognisable values directly, we replaced them with computed expressions that resolve to the same values at runtime.

```
static uint _MC  { get { return (uint)(1 << 12); } }   // 0x1000 MEM_COMMIT
static uint _MR  { get { return (uint)(1 << 13); } }   // 0x2000 MEM_RESERVE
static uint _PERW { get { return (uint)(1 << 6); } }   // 0x40   PAGE_EXECUTE_READWRITE
```

The values are computed at runtime using bit-shift operations. As a result, no suspicious constants appear directly in the source code.

5. Entry point stub

At some stages of development, the entry point stub code had hardcoded byte arrays for x86 and x64 shellcode stubs:

```
shellcode = new byte[] {
    0x48, 0xB9, 0, 0, 0, 0, 0, 0, 0, 0,  // mov rcx, hModule
    0xBA, 0, 0, 0, 0,                     // mov edx, reason
    0x4D, 0x31, 0xC0,                     // xor r8, r8
    ... };
```

To address this issue, we changed how the loader builds stub programmatically:

```
stub = new byte[30];
stub[0] = 0x48; stub[1] = 0xB9;
Array.Copy(BitConverter.GetBytes((long)m.Base), 0, stub, 2, 8);
stub[10] = 0xBA;
Array.Copy(BitConverter.GetBytes(reason), 0, stub, 11, 4)
```

Instruction bytes are assigned individually, with operands computed from runtime values. Unlike previous development releases, the file contains no recognisable opcode sequences that could be detected as static patterns.

6. Identifier and comment sanitisation

All structure names, method names, variable names and comments were sanitised:

Before	After
IMAGE_DOS_HEADER	S_DOS
IMAGE_SECTION_HEADER	S_SH
MemoryModule	ModInfo
LoadFromMemory	MapModule
CallDllMain	InvokeEP
ProcessRelocations	DoRelocs
VirtualAlloc	pVA()

All comments containing words such as “`shellcode`”, “`bypass`”, “`inject`”, “`reflective`”, “`memory DLL loader`”, or “`w3wp`” were either removed or rewritten in neutral language. Additionally, any references to loaders, DLLs, execution or similar terms were carefully stripped from the code.

Execution methods

The Phantom loader supports multiple execution methods, each with its own trade-offs:

Direct Delegate Call

This is the simplest approach. `Marshal.GetDelegateForFunctionPointer` converts the exported function address into a callable C# delegate. It works for most functions but introduces **CLR marshaling overhead** and can interfere with functions that are sensitive to **stack state**.

CreateThread (recommended)

The exported function address is passed directly as `lpStartAddress` to `CreateThread`. The function runs on a clean native thread with no CLR frames on the stack.

This is the most reliable method for functions that:

- Perform complex operations.
- Call deep into the CRT.
- Use structured exception handling.
- Create threads internally.

Example usage:

```
IntPtr ht = pCT()(IntPtr.Zero, UIntPtr.Zero, funcAddr, IntPtr.Zero, 0, out tid);
pWFSO()(ht, 30000); // Wait up to 30 seconds
```

QueueUserAPC

This method queues the function as an **APC** on the current thread and triggers it with an alertable wait.

It's useful when **thread creation is monitored**, but it's limited to functions with the PAPCFUNC signature (a single ULONG_PTR parameter).

Automating post-exploitation in a live red team engagement

During a recent red team engagement, we obtained access to a compromised service account with write permissions to the `IIS wwwroot` directory. This level of access allowed us to upload files to the web root, so we deployed an ASPX-based loader named **"Phantom."**

After successfully deploying the Phantom loader, we were faced with two operational paths:

- Establish a SOCKS proxy tunnel and browse to the IIS web server instance, and manually provide the DLL implant to Phantom loader
- Develop a custom .NET utility that could be executed directly in memory via an execute-assembly capability from our existing implant, automating the entire DLL delivery and injection process.

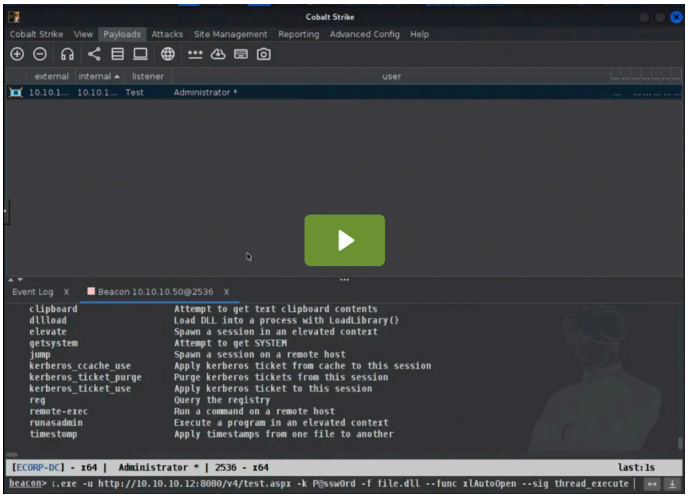
While a SOCKS proxy can provide interactive access, it introduces additional operational overhead and potential detection telemetry. Instead, we built a lightweight .NET tool named **PhantomLink**, aligned with our loader's naming convention, to automate the process and minimise operator interaction.

The **PhantomLink** assembly is designed to be executed reflectively (e.g., through an execute-assembly feature commonly available in Command and Control (C2) frameworks). It supports:

- Supplying the target DLL as either:
  - A local file path (remote target), or
  - A Base64-encoded string
  - Stager to fetch payload from remote resource.
- Specifying the exported function to invoke
- Selecting the injection method
- Passing additional loader parameters as required

By using the PhantomLink .NET tool, we reduced manual steps, lowered detection risk, and improved repeatability during the operation. We plan to continue enhancing it and introduce additional techniques to handle larger payloads.

The following quick demo shows how to use it within Cobalt Strike. If you prefer not to provide a file path (e.g., a Beacon), you can supply it as a Base64-encoded value instead or use the stager option.



You can access the tooling and full source code in the GitHub repository linked here. The repository includes complete implementation, documentation, and detailed usage instructions.

Detection and blue team guidance

In the interest of responsible disclosure, this section outlines detection strategies and hardening recommendations based on the techniques demonstrated throughout this article.

The loader operates entirely within the IIS worker process (`w3wp.exe`), never writes files to disk and resolves sensitive API calls dynamically at runtime. As a result, traditional signature-based detection is largely ineffective.

However, the technique still produces behavioural artifacts that defenders can monitor and use for detection.

1. Memory-level detection

The reflective loading process requires allocating memory regions with executable permissions inside w3wp.exe. Under normal IIS operation, the worker process does not allocate private RWX (Read-Write-Execute) or RX (Read-Execute) memory regions outside of the CLR's own JIT activity.

Defenders should baseline their environment and alert on the following:

- **Private executable memory regions** within w3wp.exe that are not backed by a known module on disk. Tools such as Moneta, PE-sieve, or custom ETW consumers can help spot these anomalies.
- **VirtualAlloc calls with PAGE_EXECUTE_READWRITE (0x40) protection** originating from the managed CLR context inside an IIS worker process. This is a strong indicator of in-process code injection, as legitimate ASP.NET applications rarely require this protection flag.

2. Thread execution anomalies

The loader spawns threads from unbacked memory.

This behaviour stands out under scrutiny. Defenders should monitor for:

- **Thread creation where the start address does not map to any loaded module.** Both execution paths used by the loader (CreateThread and QueueUserAPC) exhibit this characteristic.

EDR telemetry sources such as Microsoft-Windows-Threat-Intelligence ETW or kernel callbacks (e.g., PsSetCreateThreadNotifyRoutine) can help surface this activity.

3. Web server hardening

Preventing the web shell from being deployed in the first place remains the most effective mitigation:

- **Restrict write permissions on web root directories.** The IIS application pool identity should have read-only access to the web root. Separate, tightly scoped service accounts should handle deployments. This alone eliminates the most common lateral movement path used for web shell placement.
- **Enable file integrity monitoring (FIM)** on all IIS content directories. Any new .aspx, .ashx, .asmx or .config file appearing outside of an approved deployment window should trigger an immediate alert.

Conclusion

This research demonstrated a complete attack chain - from write access to an IIS web root to arbitrary native code execution in memory. We explored multiple techniques to bypass detection and demonstrate what an attacker can realistically achieve within an IIS environment.

The key takeaway for red teamers is that IIS web roots are high-value lateral movement targets. A single ASPX file can provide a persistent, authenticated platform for in-memory native code execution inside a trusted process.

For blue teamers, the message is equally clear: monitor your web roots, investigate RWX memory allocations in w3wp.exe and don't assume that application whitelisting alone will prevent code execution.

For more information on the Red Teaming services provided by Resillion visit our website or get in touch to speak to an expert today.

Author biography:

Lawrence is a cyber security expert with over a decade of experience in red team operations, penetration testing and security research. He has been recognised by leading technology companies including Microsoft, Facebook, Yahoo, SteelSeries and Sony for responsible vulnerability disclosure. An active contributor to open-source security tools, Lawrence's research has been referenced by industry publications such as Threatpost and BleepingComputer, underscoring his influence in the cyber security community.

resillion

Assure. Secure. Innovate.