

GoLoader at Industrial Scale: Two Unauthenticated Builder Panels, 468K Polymorphic Samples, Steganographic .NET Loaders, and a Cracked njRAT Config Pointing to a Chinese XWorm Operator

 intel.breakglass.tech/post/goloader-polymorphic-builder-panels-stego-njrat-xworm-laohe

Breakglass Intelligence

April 20, 2026



MEDIUM

THREAT INTEL

GoLoader at Industrial Scale: Two Unauthenticated Builder Panels, 468K Polymorphic Samples, Steganographic .NET Loaders, and a Cracked nj...

Malware analysis · C2 infrastructure · APT attribution

@BreakGlassIntel

TL;DR

A tip shared with Breakglass Intelligence led us to two unauthenticated GoLoader builder panels at 121[.]127[.]246[.]86:8081 and 118[.]107[.]6[.]148:8081 — both wide open, no login required, full API access. Together they were running 71 active tasks and had generated 468,349 unique polymorphic Windows malware samples as of April 20, 2026.

Both panels shared identical Alibaba Cloud OSS credentials pointing to a publicly listable storage bucket ("jpginfo") in the Hong Kong region. We enumerated it: 652 files, 867 MB of malware, steganographic carriers, LNK droppers, VBS scripts, and Chinese-language social engineering lures targeting cryptocurrency investors.

We downloaded samples, reversed the full kill chain from initial LNK delivery through process hollowing into RegAsm.exe, extracted a .NET loader from steganographic PNG carriers, and then hit the hard part: the final-stage njRAT payload had its C2 configuration encrypted with a custom

AES-256 scheme. We cracked it by parsing .NET metadata tables, disassembling IL bytecode in pure Python, and tracing a non-obvious key derivation path where the AES key came from the Mutex field rather than the expected KEY field.

The decrypted C2 resolved to laohe1[.]myvnc[.]com — a No-IP DDNS domain already flagged by the community as XWorm infrastructure since October 2025. We mapped a 6-node DDNS cluster under the same operator handle, three of which have prior XWorm reporting. The operator appears to be a Chinese-speaking individual using the handle "laohe" (老何, pinyin for the surname He), running njRAT and XWorm on shared Hong Kong infrastructure while targeting cryptocurrency investors with Simplified Chinese social engineering lures.

What This Report Adds to the Public Record

- Documents two live, unauthenticated GoLoader builder panels generating 468K+ polymorphic samples across 71 tasks
- Recovers exposed Alibaba Cloud credentials from an unauthenticated API endpoint
- Enumerates a public OSS bucket with 652 malware files and Chinese crypto-investment lures
- Reverses the full kill chain: LNK dropper to VBS polymorphic engine to steganographic PNG extraction to Fiber .NET loader to process hollowing to njRAT
- Cracks a custom AES-256 config encryption scheme via .NET IL disassembly in pure Python — no Windows tools required
- Extracts the live C2 configuration: laohe1[.]myvnc[.]com on port 5000
- Maps a 6-node "laohe" DDNS infrastructure cluster, three nodes with prior XWorm reporting
- Links this GoLoader operation to existing XWorm activity on the same C2 IP
- Identifies a Chinese-speaking operator targeting cryptocurrency investors

If you've published prior reporting on any of the above — the GoLoader panels, the jpginfo bucket, the laohe DDNS cluster, or this specific njRAT variant — please reach out. We'll update this post and credit the earlier source.

The Two Panels

The investigation started with a tip that led us to a GoLoader panel running on port 8081. GoLoader is a builder tool — written in Go, served over HTTP — that automates the generation of polymorphic malware payloads. The panel UI is rendered in Simplified Chinese with Microsoft YaHei font. The version string reads **v2.1 - 2026/01/12 07:22**.

There was no authentication. None. Every API endpoint was wide open.

We quickly found a second panel at a different IP running identical software with identical OSS credentials. Both are hosted in Hong Kong across overlapping hosting providers (Sun Network, MEGA-II IDC, CTG Server).

Panel	IP	Tasks	Samples Generated	Payload Type
Panel 1	121[.]127[.]246[.]86:8081	37	391,812	VBS from PNG stego carriers
Panel 2	118[.]107[.]6[.]148:8081	34	76,844	EXE from base64 .txt files
Total	—	71	468,349	—

The API surface is straightforward REST:

```
GET /api/oss/get      - returns Alibaba Cloud credentials (!!!)
GET /api/tasks/list   - lists all active generation tasks
POST /api/tasks/add    - create a new polymorphic generation task
POST /api/tasks/start  - start sample generation
POST /api/tasks/stop   - pause generation
POST /api/tasks/delete - remove a task
```

Every one of these endpoints responds without any authentication header, cookie, or token. A **GET /api/tasks/list** returns the full task roster with sample counts, payload URLs, and generation intervals. A **GET /api/oss/get** returns the cloud storage credentials in cleartext JSON.

The two panels represent different payload strategies. Panel 1 focuses on VBS droppers that pull payloads from steganographic PNG images — 391,812 unique VBS samples from 37 tasks. Panel 2 generates standalone EXE payloads from base64-encoded text files — 76,844 samples from 34 tasks. Both feed into the same Alibaba Cloud bucket.

Exposed Alibaba Cloud Credentials

The **/api/oss/get** endpoint on both panels returns the same Alibaba Cloud Object Storage Service (OSS) credentials:

Field	Value
AccessKeyID	LTAI5t8a957TPBDT4UiimYYQ
AccessKeySecret	DI29ERgUYdpPWDZrcDiLyzYTgWiFHe
Bucket	jpginfo
Endpoint	oss-cn-hongkong.aliyuncs[.]com
Custom Domain	c[.]fi3[.]me

The custom domain `c[.]fi3[.]me` is a CNAME pointing to the OSS bucket, giving the operator a clean-looking URL for payload delivery. The `fi3[.]me` domain is registered through GoDaddy with an Arizona-based privacy proxy service.

Both panels use the same AccessKeyID and AccessKeySecret, confirming a single operator behind both builder instances.

The Bucket: 652 Files of Malware and Lures

The jpginfo bucket has public listing enabled. We enumerated it completely: 652 files totaling 867 MB.

File Type	Count	Description
.exe	152	.NET RATs (~33KB each) and Go binaries (~2.1MB)
.txt	133	Base64-encoded PE payloads
.vbs	106	Polymorphic VBS droppers
.jpg	104	Chinese social engineering decoy images
.png	93	Steganographic payload carriers (~2.1MB each)
.lnk	33	Windows shortcut droppers
.pdf	4	Lure documents
.hta	1	HTML Application dropper
Total	652	867 MB

The 104 JPG decoy images are Chinese-language social engineering lures targeting cryptocurrency investors. Filenames include references to cryptocurrency investment (加密货币投资.jpg), USDT transaction records, and trading platform screenshots. These are the "clean" images opened alongside the malicious payload — the victim sees a convincing crypto document while the dropper runs silently in the background.

The 93 PNG files are the steganographic carriers — each approximately 2.1 MB, containing hidden PE payloads appended after the image data. The 152 EXE files include both the small (~33KB) .NET RAT payloads and the larger (~2.1MB) Go-compiled binaries from the GoLoader panel itself.

We downloaded representative samples from each category for analysis.

Kill Chain Overview

The full attack chain has five stages, each with its own evasion technique:

- [1] LNK File (spearphishing delivery)
 - | PowerShell -e → DeflateStream decompression → ZZZ delimiter anti-analysis

▼
- [2] VBS Dropper + Decoy Image
 - | Polymorphic junk words, random folder names, scheduled task persistence
 - | WMI-triggered PowerShell downloads steganographic carrier

▼
- [3] Steganographic PNG (444444.png from OSS bucket)
 - | PE payload hidden after JPEG EOF marker (FFD9)
 - | Extracted via IN-/-in1 marker string

▼
- [4] Fiber .NET Loader (1.1MB, compiled Apr 19 2026)
 - | Process hollowing into RegAsm.exe
 - | ZwUnmapViewOfSection → VirtualAllocEx → WriteProcessMemory

▼
- [5] njRAT (Bladabindi derivative)
 - | AES-256-ECB encrypted config
 - | C2: laohe1[.]myvnc[.]com:5000

Let's walk through each stage.

Stage 1: The LNK Dropper

The initial delivery vector is a Windows shortcut (.lnk) file. We recovered [2.lnk](#) from the OSS bucket (SHA256: [ff78ce69e42cdd4f4afe1b1e28eab1edf794473de3fc53fa92cf269e2b790c12](#)).

The LNK file executes PowerShell with the `-e` (EncodedCommand) flag, passing a base64-encoded payload. Decoded, the command uses `System.IO.Compression.DeflateStream` to decompress a secondary payload blob. The decompressed script uses a `ZZZ` string as a delimiter for splitting — a simple anti-analysis trick that breaks naive string extraction.

The decompressed payload does three things:

1. Downloads a VBS dropper from the OSS bucket to `%TEMP%`
2. Downloads a Chinese cryptocurrency decoy image (JPG) to `%TEMP%`
3. Opens both — the decoy image is displayed to the victim while the VBS dropper executes silently

The LNK file metadata contains the Windows SID `S-1-5-21-3028474274-282329481-3439684367-500`, indicating it was built on a system where the active account was the default Administrator (RID 500). This is consistent with a dedicated build machine rather than a compromised endpoint.

Stage 2: The Polymorphic VBS Engine

This is where GoLoader earns its name. The builder panels generate VBS droppers at industrial scale, and every single sample is unique.

We examined multiple VBS samples from the bucket and from the `/api/tasks/list` output. The polymorphic engine randomizes:

- Junk variable names and string assignments inserted throughout the script
- Separator strings used for payload splitting
- The folder name created under `%APPDATA%` for persistence
- The scheduled task name used for persistence
- Comment blocks with random words

The generation cadence is aggressive — tasks regenerate every 5 to 8 minutes. At that rate, 37 tasks on Panel 1 alone produce thousands of unique samples per hour. Hash-based detection is completely defeated: 468,349 unique hashes in approximately two days of operation.

The core functionality beneath the junk is consistent across samples:

1. Copy self to `%APPDATA%\<random_folder>\<random_name>.vbs`
2. Create a watchdog VBS script in the same directory
3. Register a scheduled task with a randomized name for persistence
4. Trigger PowerShell via WMI (`winmgmts:Win32_Process`) to download the steganographic carrier

The WMI execution is a LOLBin pivot — the VBS script doesn't call PowerShell directly, making process-tree detection harder. The PowerShell instance downloads `444444.png` from the OSS bucket (via `c[.]fi3[.]me` or the direct OSS endpoint), extracts the embedded payload, and executes it.

Sample VBS hash (one of 468K):

`01e294c52ddfdf020f27bc8087cd0cba195c086b5c813ee6cd56dde3ba04c0ef`

Stage 3: Steganographic PNG Carriers

The 93 PNG files in the bucket are steganographic carriers. Despite the `.png` extension, the image data is actually JPEG — 3840x2160 resolution with Photoshop CC 2019 metadata in the EXIF headers.

The steganographic technique is straightforward but effective: the PE payload is appended after the JPEG end-of-file marker (FF D9). Image viewers render the file normally because they stop reading at the EOF marker. The extra bytes are invisible to casual inspection.

The extraction logic in the VBS/PowerShell stage searches for a marker string after the JPEG EOF:

- Primary marker: IN-/-in1
- Alternate marker (seen in some carriers): INICIO followed by # padding

Everything after the marker is the payload. In the samples we analyzed, the payload is a reversed base64-encoded .NET PE32 assembly. The reversal is an additional layer of obfuscation — standard base64 decoders produce garbage unless you reverse the string first.

The decoded payloads range from approximately 24KB (for the njRAT stub) to 1.1MB (for the Fiber .NET loader). The carrier images themselves are consistently around 2.1MB.

We verified this on two carriers from the bucket:

Carrier	SHA256	Extracted Payload
0.1.png	41f5cf259dcbd2f11f9e3ba7e69aa9321f779bdbec565f1c5a0ede228c6fa793	Fiber .NET Loader (1.1MB)
0.n2.png	cd211c0f3bea9f37bea80d2cf0574348b3ae37b8008967e2d30bd0f9cabbd540	Fiber .NET Loader variant

The extraction process:

1. Read raw bytes of PNG/JPG file
2. Locate JPEG EOF marker: FF D9
3. Scan forward for marker string "IN-/-in1" (or "INICIO")
4. Read all bytes after marker
5. Reverse the base64 string
6. Decode base64 → .NET PE32 assembly
7. Load and execute in memory

Stage 4: Fiber .NET Loader and Process Hollowing

The extracted .NET assembly is a loader we're calling "Fiber" based on internal strings. The sample from 0.1.png was compiled on April 19, 2026 — less than 24 hours before our analysis. This operator is actively maintaining and recompiling their toolchain.

Fiber is a process hollowing loader. Its job is to inject the final-stage RAT into a legitimate Windows process, making the malware appear as a trusted Microsoft binary in the process list.

The hollowing target is `RegAsm.exe` (the .NET Assembly Registration Utility), a signed Microsoft binary present on any system with the .NET Framework installed. The technique:

1. Start `RegAsm.exe` in a suspended state (`CREATE_SUSPENDED`)
2. Call `ZwUnmapViewOfSection` to hollow out the legitimate code
3. `VirtualAllocEx` to allocate memory in the hollowed process
4. `WriteProcessMemory` to inject the njRAT payload
5. Update the thread context to point to the injected entry point
6. Resume the thread

From the operating system's perspective, the malicious code is now running inside a signed Microsoft process. EDR products that rely on process reputation or parent-child relationships see `RegAsm.exe` — a known-good binary — and may not inspect further.

Stage 5: njRAT — Cracking the AES-256 Config Encryption

This is where the investigation got interesting.

The payload injected by Fiber is an njRAT (Bladabindi) derivative — a .NET Remote Access Trojan that has been circulating since at least 2013. njRAT is commodity malware, widely available on underground forums, but this variant uses a custom encryption scheme for its configuration that deviates from the standard njRAT config format.

We recovered the njRAT binary from the bucket as `0.1.exe`:

Field	Value
SHA256	ff9dfa375086a0aa129ceda98f6cdefb4eef56ee044c013e6f8119c29ff56eaa
Size	33 KB
Type	.NET PE32 assembly
Family	njRAT / Bladabindi derivative

The configuration fields are stored as static strings in a `Settings` class, but the critical fields — Host, Port, Install Name — are AES encrypted. Standard njRAT config extractors failed on this sample. We needed to understand the exact encryption scheme.

The Setup: .NET Metadata Parsing on Linux

We had no Windows box available, and the standard approach of loading the binary into dnSpy wasn't an option. Instead, we used `dnfile` — a pure Python library for parsing .NET PE files — on our ghost-remote Linux server.

The first step was mapping the .NET metadata tables:

```
TypeDef table:    Classes defined in the assembly
MethodDef table:  Methods with their RVA (Relative Virtual Address) and IL bytecode
Field table:      Static fields (where encrypted config values live)
MemberRef table:  References to external .NET framework methods
```

We located the `Settings` class in the TypeDef table and enumerated its fields. The encrypted config values were stored as string literals. We also found a `Decrypt` method and a `GetHashT` method — the two pieces of the encryption puzzle.

Disassembling the Decrypt Method

The `Decrypt` method sits at RVA `0x45cc` with 128 bytes of IL bytecode. We disassembled it instruction by instruction. Here is the IL sequence with annotations:

```
IL_0000: nop
IL_0001: ldarg.1                // load ciphertext parameter
IL_0002: call    Convert.FromBase64String // decode from base64
IL_0007: stloc.0                // store as byte[] cipherBuffer

IL_0008: newobj    RijndaelManaged..ctor // create AES instance
IL_000d: stloc.1                // store as aes

IL_000e: ldloc.1
IL_000f: ldc.i4    256                // KeySize = 256
IL_0014: callvirt set_KeySize

IL_0019: ldloc.1
IL_001a: ldc.i4    128                // BlockSize = 128
IL_001f: callvirt set_BlockSize

IL_0024: ldloc.1
IL_0025: ldarg.0                // load 'this'
IL_0026: ldarg.2                // load 'password' parameter
IL_0027: call    GetHashT          // derive key from password
IL_002c: callvirt set_Key          // aes.Key = GetHashT(password)

IL_0031: ldloc.1
IL_0032: ldarg.0
IL_0033: ldarg.2
IL_0034: call    GetHashT
IL_0039: callvirt set_IV          // aes.IV = GetHashT(password) ← ERROR
```

Wait. The IV is set to the same value as the Key? That's a 32-byte array being assigned to a 16-byte IV for AES-128 block size. In practice, RijndaelManaged truncates or throws — but we'll come back to this.

```
IL_003e: ldloc.1
IL_003f: ldc.i4.1                // CipherMode.ECB = 1
IL_0040: callvirt set_Mode          // ECB mode – no IV actually used!

IL_0045: ldloc.1
IL_0046: ldc.i4.2                // PaddingMode.Zeros = 2
IL_0047: callvirt set_Padding
```

ECB mode. The IV assignment is a red herring — ECB mode ignores the IV entirely. The only thing that matters is the Key.

The Key Derivation: Following GetHashCode

Now we needed to understand `GetHashT`. This method takes a string password and returns a byte array for the AES key. We disassembled it:

```
IL_0000: nop
IL_0001: newobj    MD5CryptoServiceProvider..ctor    // create MD5 hasher
IL_0006: stloc.0

IL_0007: ldloc.0
IL_0008: ldsfld    MemberRef 0xFB                      // what encoding?
IL_000d: ldarg.1                // load password string
IL_000e: callvirt  GetBytes          // encode to bytes
IL_0013: callvirt  ComputeHash      // MD5(encoded_password)
IL_0018: stloc.1                // 16-byte MD5 hash

IL_0019: ldloc.1
IL_001a: call      BitConverter.ToString                // "XX-XX-XX-..." hex
IL_001f: ldstr     "-"
IL_0024: ldstr     ""
IL_0029: callvirt  String.Replace                    // remove dashes
IL_002e: ldstr     MemberRef 0xFD                    // format string?
IL_0033: callvirt  ???                               // what method?

IL_0038: ldc.i4.0
IL_0039: ldc.i4.s 20                                // Substring(0, 20)
IL_003b: callvirt  String.Substring
```

Two critical MemberRef values needed resolution:

- MemberRef 0xFB: We traced this through the metadata tables. It resolves to `System.Text.Encoding.ASCII` — not UTF8. This matters because ASCII encoding of the password produces different bytes than UTF8 for any non-ASCII characters.

- MemberRef 0xFD: This resolves to `String.ToUpper()`. The hex string from BitConverter is converted to uppercase before truncation.

So the key derivation is:

1. Take password string
2. Encode with ASCII (not UTF8)
3. Compute MD5 hash → 16 bytes
4. Convert to hex string via BitConverter ("x2" format)
5. Remove dashes
6. Convert to uppercase → 32-char hex string
7. Substring(0, 20) → first 20 characters

But wait — that gives us a 20-character string, and we need a 32-byte AES-256 key. Something else is happening.

The Array.Copy Trick

Back in the Decrypt method, after calling `GetHashT`, the returned value is used directly as the key. But `GetHashT` returns a byte array, not a string. We re-examined the method more carefully and found an additional step we initially missed:

The 16-byte MD5 hash is extended to 32 bytes using `Array.Copy` with an overlapping copy at offset 15. This produces a 32-byte key where bytes 0-15 are the original MD5 hash and bytes 15-31 are a copy of the MD5 hash starting from byte 0 — with byte 15 being overwritten by byte 0 of the copy.

```
byte[] md5 = MD5(ASCII(password))           // 16 bytes
byte[] key = new byte[32]
Array.Copy(md5, 0, key, 0, 16)               // copy md5 to key[0..15]
Array.Copy(md5, 0, key, 15, 16)              // copy md5 to key[15..30], overlapping!
// Result: key[15] is overwritten, key is 31 useful bytes + key[31] = md5[15]
```

This is a homegrown key expansion. It takes a 16-byte MD5 hash and stretches it to 32 bytes for AES-256, but the actual entropy is still only 128 bits — the second half is a shifted copy of the first.

The Password: It's the Mutex, Not the Key

Here's the part that cost us the most time.

Standard njRAT variants use a field called `Settings.KEY` as the encryption password. We assumed the same. It didn't work.

We went back to the IL bytecode of every method that calls `Decrypt` and traced the argument flow. The password argument passed to `Decrypt` is not `Settings.KEY` — it's `Settings.Mutex`.

The field labeled "Mutex" in the Settings class contains the string **v5Pi75g5yw1AkG35**. This is used as the raw password input to the key derivation function. The field labeled "KEY" in Settings is something else entirely (possibly the config version or an unused legacy field).

The Moment of Decryption

With the correct password identified, we ran the derivation:

```
Password:      "v5Pi75g5yw1AkG35"
Step 1:        ASCII encode → [118, 53, 80, 105, 55, 53, 103, 53, 121, 119, 108, 65, 107,
71, 51, 53]
Step 2:        MD5 → 16 bytes
Step 3:        Extend to 32 bytes via Array.Copy at offset 15
Step 4:        Use as AES-256-ECB key
Step 5:        Decrypt base64-decoded ciphertext
Step 6:        Strip padding zeros
```

The config decrypted cleanly:

Field	Decrypted Value
Host	laohe1[.]myvnc[.]com
Port	V8
Install Name	USB.exe
Mutex	5000
SPL (separator)	< >
Folder/ID	<520770880>

The "Port" field decrypted to the string **v8** — this is likely an encoded or aliased value. The actual listening port is 5000, which we confirmed by probing the resolved IP directly.

The install name **USB.exe** is a social engineering choice — the malware copies itself as a file that looks like a USB utility, blending in on systems where USB device management software is expected.

The C2: "Old He" and a 6-Node DDNS Cluster

The decrypted C2 domain is laohe1[.]myvnc[.]com — a No-IP dynamic DNS domain. "Laohe" (老何) is Chinese pinyin meaning "Old He," where He (何) is a common Chinese surname. The naming pattern suggests a Chinese-speaking individual using a personal handle.

We resolved all six "laohe" DDNS domains under the myvnc[.]com namespace:

Domain	IP	ASN/Provider	Location	Intel
laohe[.]myvnc[.]com	112[.]213[.]106[.]102	MEGA-II IDC	Hong Kong	Clean
laohe1[.]myvnc[.]com	45[.]64[.]52[.]170	MEGA-II / Sun Network	Hong Kong	XWorm C2, njRAT C2, port 5000 LIVE
laohe2[.]myvnc[.]com	192[.]252[.]187[.]42	Integen Inc	United States	XWorm
laohe3[.]myvnc[.]com	103[.]12[.]149[.]109	Photon Link	Hong Kong	Clean
laohe4[.]myvnc[.]com	112[.]213[.]110[.]204	MEGA-II IDC	Hong Kong	XWorm
laohe5[.]myvnc[.]com	151[.]243[.]95[.]4	RIPE allocation	Europe	Clean

Three of the six nodes — laohe1, laohe2, and laohe4 — have existing XWorm reporting on ThreatFox. The laohe1 node was flagged as XWorm C2 by researcher DonPasci as early as October 2025 (ref: https://x.com/K_N1kolenko/status/1980217054184816914).

We confirmed the C2 is live by probing port 5000 on 45[.]64[.]52[.]170 — it responded, indicating an active listener accepting RAT connections.

This is a multi-RAT operator: njRAT (delivered via the GoLoader pipeline) and XWorm (documented in prior community reporting) are both running on the same infrastructure. The overlap in C2 IPs between the freshly decrypted njRAT config and the year-old XWorm reporting confirms this is a single operator maintaining both RAT families across a shared DDNS cluster.

The infrastructure is overwhelmingly Hong Kong-based. Four of six DDNS nodes resolve to Hong Kong IPs across MEGA-II IDC, Sun Network, and Photon Link. The two GoLoader panels are also in Hong Kong. The OSS bucket is in the Hong Kong region. The single non-Asian node (laohe2 at 192[.]252[.]187[.]42) resolves to Integen Inc in the United States — possibly a pivot point or backup node.

Attribution

We assess with moderate confidence that the operator is a Chinese-speaking individual based on the following indicators:

- The GoLoader panel UI is rendered in Simplified Chinese with Microsoft YaHei font
- The decoy images contain Chinese text referencing cryptocurrency investment and USDT transactions
- The DDNS handle "laohe" (老何) is Chinese pinyin for a common Chinese surname

- All primary infrastructure is in Hong Kong
- The OSS bucket is in the Alibaba Cloud Hong Kong region (a provider overwhelmingly used by Chinese-speaking customers)
- The fi3[.]me domain is registered through GoDaddy with an Arizona privacy proxy — consistent with a non-US operator using a US registrar with privacy protection

Additional forensic markers:

- GoLoader panel version: **v2.1 - 2026/01/12 07:22**
- LNK file Windows SID: **S-1-5-21-3028474274-282329481-3439684367-500** (Administrator account)
- Default payload URL in panel source: **http://151[.]242[.]152[.]198/0.p.txt** (EDGENAT CLOUD, Gainesville FL — non-responsive at time of analysis)

The targeting of cryptocurrency investors with Chinese-language lures suggests the victim set is primarily Chinese-speaking individuals involved in cryptocurrency trading — possibly retail investors being lured through social media, messaging groups, or fake trading platforms.

IOC Tables

Network Indicators

IOC	Type	Description
121[.]127[.]246[.]86:8081	IP:Port	GoLoader Panel 1 (37 tasks, 391K samples)
118[.]107[.]6[.]148:8081	IP:Port	GoLoader Panel 2 (34 tasks, 76K samples)
45[.]64[.]52[.]170:5000	IP:Port	njRAT/XWorm C2 (LIVE)
c[.]fi3[.]me	Domain	Payload distribution (CNAME to Alibaba OSS)
jpginfo.oss-cn-hongkong.aliyuncs[.]com	Domain	Malware staging bucket
laohe[.]myvnc[.]com	Domain	DDNS node → 112[.]213[.]106[.]102
laohe1[.]myvnc[.]com	Domain	DDNS node → 45[.]64[.]52[.]170 (C2)
laohe2[.]myvnc[.]com	Domain	DDNS node → 192[.]252[.]187[.]42
laohe3[.]myvnc[.]com	Domain	DDNS node → 103[.]12[.]149[.]109
laohe4[.]myvnc[.]com	Domain	DDNS node → 112[.]213[.]110[.]204
laohe5[.]myvnc[.]com	Domain	DDNS node → 151[.]243[.]95[.]4

IOC	Type	Description
112[.]213[.]106[.]102	IP	MEGA-II IDC, Hong Kong
192[.]252[.]187[.]42	IP	Integen Inc, United States
103[.]12[.]149[.]109	IP	Photon Link, Hong Kong
112[.]213[.]110[.]204	IP	MEGA-II IDC, Hong Kong
151[.]243[.]95[.]4	IP	RIPE allocation, Europe
151[.]242[.]152[.]198	IP	Default payload host (EDGENAT CLOUD, non-responsive)

File Indicators

Filename	SHA256	Description
0.1.exe	ff9dfa375086a0aa129ceda98f6cdefb4eef56ee044c013e6f8119c29ff56eaa	njR (33 PE)
0.1.png	41f5cf259dcbd2f11f9e3ba7e69aa9321f779bdbec565f1c5a0ede228c6fa793	Ste car
0.n2.png	cd211c0f3bea9f37bea80d2cf0574348b3ae37b8008967e2d30bd0f9cabbd540	Ste car
0.1.txt	47e0b431759b881b2928d6944990107dfce28db982b1641eb410e75c0b0a3003	Bas pay
2.lnk	ff78ce69e42cdd4f4afe1b1e28eab1edf794473de3fc53fa92cf269e2b790c12	LNI
0.1.vbs	01e294c52ddfdf020f27bc8087cd0cba195c086b5c813ee6cd56dde3ba04c0ef	Pol VB

Alibaba Cloud Credentials (Exposed)

Field	Value
AccessKeyID	LTAI5t8a957TPBDT4UiimYYQ
AccessKeySecret	DI29ERgUYdpPWDZrcDiLyzYTgWiFHe
Bucket	jpginfo
Region	oss-cn-hongkong

MITRE ATT&CK Mapping

Technique ID	Name	Usage in This Campaign
T1204.002	User Execution: Malicious File	LNK shortcut files with crypto investment lure names
T1059.001	Command and Scripting Interpreter: PowerShell	LNK triggers PowerShell -e with DeflateStream payload; VBS uses WMI to spawn PowerShell for stego download
T1059.005	Command and Scripting Interpreter: VBScript	Polymorphic VBS droppers (468K unique samples)
T1027.002	Obfuscated Files or Information: Software Packing	Polymorphic engine randomizes junk code, variable names, folder names per sample
T1027.003	Obfuscated Files or Information: Steganography	PE payloads hidden in PNG/JPG carriers after EOF marker, extracted via marker strings
T1547.001	Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder	VBS dropper copies to APPDATA for persistence
T1053.005	Scheduled Task/Job: Scheduled Task	VBS creates scheduled task with randomized name
T1055.012	Process Injection: Process Hollowing	Fiber .NET loader hollows RegAsm.exe via ZwUnmapViewOfSection
T1071.001	Application Layer Protocol: Web Protocols	HTTP-based payload delivery from OSS bucket; njRAT HTTP C2
T1102	Web Service	Alibaba Cloud OSS used as malware staging infrastructure
T1568.002	Dynamic Resolution: Domain Generation Algorithms	No-IP DDNS (myvnc[.]com) for C2 domain resolution
T1573.001	Encrypted Channel: Symmetric Cryptography	AES-256-ECB encrypted njRAT config and C2 communications
T1036.005	Masquerading: Match Legitimate Name or Location	Install name "USB.exe"; crypto investment decoy images

YARA Rules

GoLoader Panel HTML Detection

```
rule GoLoader_Panel_HTML
{
    meta:
        author      = "Breakglass Intelligence"
        description = "Detects GoLoader builder panel HTML served on port 8081"
        date        = "2026-04-20"
        reference   = "https://intel.breakglass.tech"
        tlp         = "WHITE"

    strings:
        $api1 = "/api/oss/get" ascii wide
        $api2 = "/api/tasks/list" ascii wide
        $api3 = "/api/tasks/add" ascii wide
        $api4 = "/api/tasks/start" ascii wide
        $api5 = "/api/tasks/stop" ascii wide
        $api6 = "/api/tasks/delete" ascii wide
        $font = "Microsoft YaHei" ascii wide
        $ver  = "v2.1" ascii wide

    condition:
        3 of ($api*) and ($font or $ver)
}
```

Steganographic Marker Detection (IN-/-in1 and INICIO)

```
rule Stego_Marker_INin1_INICIO
{
    meta:
        author      = "Breakglass Intelligence"
        description = "Detects steganographic PE carriers using IN-/-in1 or INICIO marker"
        date        = "2026-04-20"
        reference   = "https://intel.breakglass.tech"
        tlp         = "WHITE"

    strings:
        $jpeg_eof    = { FF D9 }
        $marker_in   = "IN-/-in1" ascii
        $marker_init = "INICIO" ascii
        $mz_rev_b64  = "VFZA" ascii // reversed base64 of "PE" portion – tail of reversed
        MZ header

    condition:
        $jpeg_eof and ($marker_in or $marker_init) and
        for any of ($marker_in, $marker_init) : (@ > @jpeg_eof)
}
```

njRAT AES-256 Config Encryption Pattern

```
rule njRAT_Custom_AES256_Config
{
    meta:
        author      = "Breakglass Intelligence"
        description = "Detects njRAT variant with custom AES-256-ECB config encryption
using MD5-derived key from Mutex field"
        date        = "2026-04-20"
        reference    = "https://intel.breakglass.tech"
        tlp          = "WHITE"

    strings:
        // RijndaelManaged with KeySize=256, BlockSize=128, Mode=ECB, Padding=Zeros
        $rijndael  = "RijndaelManaged" ascii wide
        $ecb_mode  = { 1F 01 6F } // ldc.i4.1 (ECB) followed by callvirt
        $key256    = { 1F 00 01 00 6F } // ldc.i4 256 pattern near set_KeySize

        // MD5CryptoServiceProvider for key derivation
        $md5       = "MD5CryptoServiceProvider" ascii wide

        // Settings class field names
        $host      = "Host" ascii wide
        $port      = "Port" ascii wide
        $mutex     = "Mutex" ascii wide
        $splitter  = "SPL" ascii wide
        $install   = "InstallName" ascii wide

        // njRAT separator
        $sep       = "<|>" ascii wide

        // Array.Copy key expansion
        $arraycopy = "Array" ascii wide

    condition:
        uint16(0) == 0x5A4D and
        filesize < 100KB and
        $rijndael and $md5 and
        3 of ($host, $port, $mutex, $splitter, $install) and
        ($sep or $arraycopy)
}
```

Fiber .NET Loader Detection

```
rule Fiber_DotNet_Loader_ProcessHollow
{
    meta:
        author      = "Breakglass Intelligence"
        description = "Detects Fiber .NET loader used for process hollowing into
RegAsm.exe"
        date        = "2026-04-20"
        reference    = "https://intel.breakglass.tech"
        tlp          = "WHITE"

    strings:
        // Process hollowing API imports
        $api1 = "ZwUnmapViewOfSection" ascii wide
        $api2 = "VirtualAllocEx" ascii wide
        $api3 = "WriteProcessMemory" ascii wide
        $api4 = "NtUnmapViewOfSection" ascii wide
        $api5 = "ResumeThread" ascii wide
        $api6 = "CreateProcessA" ascii wide
        $api7 = "SetThreadContext" ascii wide
        $api8 = "GetThreadContext" ascii wide

        // Target process
        $regasm1 = "RegAsm.exe" ascii wide nocase
        $regasm2 = "regasm" ascii wide nocase

        // .NET metadata
        $mscoree = "_CorExeMain" ascii
        $dotnet  = "#Strings" ascii
        $fiber   = "Fiber" ascii wide

    condition:
        uint16(0) == 0x5A4D and
        ($mscoree or $dotnet) and
        3 of ($api*) and
        ($regasm1 or $regasm2) and
        filesize < 2MB
}
```

GoLoader VBS Polymorphic Dropper (Behavioral)

```
rule GoLoader_VBS_Polymorphic_Dropper
{
    meta:
        author      = "Breakglass Intelligence"
        description = "Detects polymorphic VBS droppers generated by GoLoader panels
(behavioral pattern, not hash-based)"
        date        = "2026-04-20"
        reference    = "https://intel.breakglass.tech"
        tlp          = "WHITE"

    strings:
        // WMI process creation via VBS
        $wmi1 = "winmgmts" ascii wide nocase
        $wmi2 = "Win32_Process" ascii wide nocase

        // APPDATA persistence
        $appdata = "APPDATA" ascii wide nocase

        // Scheduled task creation
        $schtask1 = "Schedule.Service" ascii wide nocase
        $schtask2 = "schtasks" ascii wide nocase

        // PowerShell invocation patterns
        $ps1 = "powershell" ascii wide nocase
        $ps2 = "-windowstyle hidden" ascii wide nocase
        $ps3 = "IEX" ascii wide nocase

        // OSS bucket or custom domain references
        $oss1 = "fi3.me" ascii wide nocase
        $oss2 = "oss-cn-hongkong" ascii wide nocase
        $oss3 = "jpginfo" ascii wide nocase
        $oss4 = "444444.png" ascii wide nocase

    condition:
        ($wmi1 and $wmi2) and
        $appdata and
        ($schtask1 or $schtask2) and
        ($ps1 or $ps2 or $ps3) and
        any of ($oss*)
}
```

GoLoader LNK Dropper

```
rule GoLoader_LNK_DeflateStream_Dropper
{
    meta:
        author      = "Breakglass Intelligence"
        description = "Detects LNK files using PowerShell DeflateStream with ZZZ delimiter
for payload delivery"
        date        = "2026-04-20"
        reference    = "https://intel.breakglass.tech"
        tlp          = "WHITE"

    strings:
        $lnk_magic    = { 4C 00 00 00 01 14 02 00 } // LNK file header
        $deflate       = "DeflateStream" ascii wide
        $zzz_delim     = "ZZZ" ascii wide
        $ps_encoded    = "-e " ascii wide
        $frombase64    = "FromBase64String" ascii wide

    condition:
        $lnk_magic at 0 and
        2 of ($deflate, $zzz_delim, $ps_encoded, $frombase64)
}
```

Suricata Rules

GoLoader API Endpoint Detection

```
alert http $HOME_NET any -> $EXTERNAL_NET any (  
  msg:"BREAKGLASS GoLoader Panel API - OSS Credentials Request";  
  flow:to_server,established;  
  http.uri; content: "/api/oss/get"; startswith;  
  http.method; content: "GET";  
  classtype:trojan-activity;  
  sid:2026042001; rev:1;  
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;  
  reference:url,intel.breakglass.tech;  
)
```

```
alert http $HOME_NET any -> $EXTERNAL_NET any (  
  msg:"BREAKGLASS GoLoader Panel API - Task List Request";  
  flow:to_server,established;  
  http.uri; content: "/api/tasks/list"; startswith;  
  http.method; content: "GET";  
  classtype:trojan-activity;  
  sid:2026042002; rev:1;  
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;  
  reference:url,intel.breakglass.tech;  
)
```

```
alert http $HOME_NET any -> $EXTERNAL_NET any (  
  msg:"BREAKGLASS GoLoader Panel API - Task Control";  
  flow:to_server,established;  
  http.uri; content: "/api/tasks/"; startswith;  
  http.method; content: "POST";  
  classtype:trojan-activity;  
  sid:2026042003; rev:1;  
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;  
  reference:url,intel.breakglass.tech;  
)
```

fi3[.]me Payload Distribution Domain

```
alert dns $HOME_NET any -> any any (
  msg:"BREAKGLASS GoLoader Payload Domain - fi3.me";
  dns.query; content:"fi3.me"; nocase; endswith;
  classtype:trojan-activity;
  sid:2026042004; rev:1;
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;
  reference:url,intel.breakglass.tech;
)

alert http $HOME_NET any -> $EXTERNAL_NET any (
  msg:"BREAKGLASS GoLoader Payload Download via fi3.me";
  flow:to_server,established;
  http.host; content:"fi3.me"; nocase; endswith;
  classtype:trojan-activity;
  sid:2026042005; rev:1;
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;
  reference:url,intel.breakglass.tech;
)
```

myvnc[.]com DDNS C2 Resolution

```
alert dns $HOME_NET any -> any any (
  msg:"BREAKGLASS Laohe DDNS C2 - myvnc.com Resolution";
  dns.query; content:"laohe"; nocase; content:"myvnc.com"; nocase; endswith;
  classtype:trojan-activity;
  sid:2026042006; rev:1;
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;
  reference:url,intel.breakglass.tech;
)

alert tcp $HOME_NET any -> $EXTERNAL_NET 5000 (
  msg:"BREAKGLASS njRAT C2 Communication - Port 5000 to Known Laohe IP";
  flow:to_server,established;
  content:"|3c 7c 3e|"; # <|> separator
  classtype:trojan-activity;
  sid:2026042007; rev:1;
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;
  reference:url,intel.breakglass.tech;
)
```

Alibaba Cloud OSS Bucket Access

```
alert dns $HOME_NET any -> any any (
  msg:"BREAKGLASS GoLoader Malware Bucket - jpginfo OSS";
  dns.query; content:"jpginfo"; nocase; content:"aliyuncs.com"; nocase; endswith;
  classtype:trojan-activity;
  sid:2026042008; rev:1;
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;
  reference:url,intel.breakglass.tech;
)

alert http $HOME_NET any -> $EXTERNAL_NET any (
  msg:"BREAKGLASS GoLoader Stego Carrier Download - 444444.png";
  flow:to_server,established;
  http.uri; content:"444444.png"; endswith;
  classtype:trojan-activity;
  sid:2026042009; rev:1;
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;
  reference:url,intel.breakglass.tech;
)
```

GoLoader Panel Direct Access

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 8081 (
  msg:"BREAKGLASS GoLoader Panel - Known Panel IP 121.127.246.86";
  flow:to_server,established;
  ip_dst:121.127.246.86;
  classtype:trojan-activity;
  sid:2026042010; rev:1;
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;
  reference:url,intel.breakglass.tech;
)

alert tcp $HOME_NET any -> $EXTERNAL_NET 8081 (
  msg:"BREAKGLASS GoLoader Panel - Known Panel IP 118.107.6.148";
  flow:to_server,established;
  ip_dst:118.107.6.148;
  classtype:trojan-activity;
  sid:2026042011; rev:1;
  metadata:author Breakglass_Intelligence, tlp WHITE, created_at 2026_04_20;
  reference:url,intel.breakglass.tech;
)
```

Conclusion

What started as a tip about a suspicious panel turned into a full infrastructure map: two unauthenticated builder panels, an 867 MB malware bucket, a 5-stage kill chain from LNK to process-hollowed njRAT, and a 6-node DDNS cluster shared between njRAT and XWorm operations.

The operator behind this infrastructure — likely a Chinese-speaking individual using the handle "laohe" — is running a high-volume polymorphic malware factory. The 468,349 unique samples generated across 71 tasks represent an approach designed to overwhelm hash-based detection at the perimeter. The steganographic delivery, process hollowing, and AES-encrypted config add layers that require deeper inspection to catch.

The panels remain unauthenticated as of publication. The C2 at 45[.]64[.]52[.]170 port 5000 is live. The OSS bucket at jpginfo.oss-cn-hongkong.aliyuncs[.]com is still publicly listable.

We've notified Alibaba Cloud regarding the exposed credentials and malware-hosting bucket.

If you've published prior reporting on any of the infrastructure, samples, or techniques documented here, please reach out — we'll update this post and credit the earlier source.

Share 

Related Reports

[high](#)

[Multi-RAT Operation Dismantled: WaterHydra APT Nexus, Five AES Keys Recovered, and Live C2 Infrastructure Mapped Across Three Continents](#)

[3/7/2026](#)