# Malicious Node Package Deploys OtterCookie

Archived: 2026-04-05 15:27:31 UTC

The Blackpoint SOC recently contained an incident involving **OtterCookie**, a North Korean linked malware family delivered through a trojanized  open-source project hosted on Bitbucket. The campaign specifically targets developers and the financial sector, leveraging the trust placed in open-source dependencies to achieve initial access and stage downstream payloads. The loader code, disguised as part of a 3D chess application, intentionally triggers a failure during initialization so its catch block fetches an “error” message from a remote API and executes that returned string in-process, ultimately staging and deploying the OtterCookie malware.

Once executed, the malware unpacked a fake application named **client-app** into a user writable **System32** directory under **%AppData%\Roaming.** A crafted **package.json** ensured required dependencies were installed before establishing Command and Control (C2) communications, where the malware exfiltrated system details and screenshots to prepare tailored payloads.

The attack chain relied on multiple supporting artifacts, including **.bat**, **.ps1**, **.dat**, and **.js** files. A launcher, **svchost.bat**, invoked the PowerShell loader **svchost.ps1**, which initialized C2 and persisted via a registry Run key **WindowsUpdate**. From there the loader abused **node.exe** to execute **.sysupdater.dat**, an obfuscated JavaScript payload responsible for clipboard collection, screenshot capture, and staging further commands.

To expand visibility, the operators later deployed **simple-keyboard-monitor.ps1**, a PowerShell based keylogger that captured keystrokes and sent them to the same C2 infrastructure. Combined with clipboard polling and screenshot

collection, **OtterCookie** gave the adversary continuous surveillance of user activity and a path to credential theft.

## Key Findings

- A trojanized Bitbucket repository posing as a 3D chess project served as the initial infection vector. Its code intentionally triggered an error during initialization, causing the catch block to fetch and execute attacker-supplied JavaScript from a remote API.
- Execution of the remote code deployed a trojanized Node module, which unpacked a malicious application (**client-app**) through its **package.json** script.
- The package established C2 communications immediately after installation, exfiltrating host details and screenshots.
- The malware relied on multiple staging artifacts (**.bat**, **.ps1**, **.dat**, and **.js**) to support execution and persistence.
- The batch file **svchost.bat** executed **svchost.ps1**, a loader that initiated C2 communications and persisted via a registry Run key **WindowsUpdate**.
- The loader abused **node.exe** to execute the hidden payload **.sysupdater.dat**, which spawned cmd loops, invoked PowerShell commands such as **Get-Clipboard**, and captured screenshots to the temp directory.
- A PowerShell based keylogger, **simple-keyboard-monitor.ps1**, was deployed to log keystrokes and exfiltrate user activity.
- The campaign targeted developers and financial organizations, aiming to capture sensitive data and enable financial theft.

## Observed Killchain

## From Pyongyang with Love

The infection originated from a Bitbucket repository that outwardly hosted a benign 3D chess project, with normal developer artifacts alongside an initialization routine wrapped in a try/catch. At runtime the app intentionally triggers the failure path so the catch block runs, which appears as a harmless recovery step to casual reviewers (Figure 1).

***Figure 1: Bitbucket 3D chess repository containing the trojanized initialization routine.***

When the catch block runs it issues an HTTP request to an operator-controlled endpoint at serve-cookie[.]vercel[.]app, calling /api/ipcheck-encrypted/<apiKey> with a custom header. The loader assigns the response to a local variable and then compiles and invokes the returned string as JavaScript with require passed into its scope, turning the remote response into executable code on the host. This try/fail design converts ordinary error handling into a stealthy staging and execution channel, because the visible repository remains largely benign while the remote response performs the malicious work (Figure 2).

***Figure 2: Try/catch loader that retrieves and executes remote code on failure.***

Execution of the server supplied code resulted in a trojanized Node module being written to disk and installed, which unpacked a malicious application named **client-app** into the local node tree. During setup the module wrote a crafted **package.json** and ensured the Node runtime would pull in its required dependencies. This gave the operators a controlled entry point, blending with the normal workflow of developers who frequently install new npm packages.

As part of the trojanized install the malware created a fake **System32** directory under the user's **%AppData%\Roaming** path and used it as the central workspace for execution and persistence. Into that location the actor placed **svchost.bat**, **svchost.ps1**, **package.json** / **package-lock.json,** a populated **node_modules** tree, and the obfuscated JavaScript payload **.sysupdater.dat**, which together form the loop that launches and maintains the stealer execution (Figure 3).

***Figure 3: Fake System32 Staging Directory Contents***

The choice of the name **svchost** for both the batch and PowerShell scripts is deliberate, mimicking a legitimate Windows service host executable to increase the chance the artifacts are overlooked during casual triage. By imitating the trusted **Windows System32** path, but placing it in a user writable location, the adversary ensures persistence while

hiding in plain sight. Persistence itself is maintained through a registry Run key, where a value named **WindowsUpdate** points directly to the **svchost.bat** script in this fake **System32** folder.

The on-disk loop begins with execution of **svchost.bat**, a small file that acts as the execution primer. This script repeatedly calls **PowerShell** with **-ExecutionPolicy Bypass**, redirects all output to null, and runs **svchost.ps1** from the staging folder (Figure 4). Its purpose is simply to ensure the PowerShell stage is executed reliably on every logon while producing almost no visible activity. Coupled with the **WindowsUpdate** Run key, **svchost.bat** guarantees the loop will restart automatically, making the overall activity resilient even if the Node process itself is killed.

***Figure 4: Contents of the svchost.bat script.***

The **svchost.ps1** script is the primary engine behind the execution loop. Each run sets the working directory to the fake **System32** staging folder, checks for a **node_modules** directory, and if dependencies are missing it writes **package.json** and performs an unattended **npm install** to restore the runtime environment before launching the stealer, making the loader self-healing and resilient to basic cleanup attempts (Figure 5). Because the script validates and rebuilds its Node environment on each invocation, **svchost.ps1** guarantees the loop can recover from file deletion or process termination and reliably hands control to the primary JavaScript payload in a hidden window.

***Figure 5: Contents of the svchost.ps1 script.***

The **package.json** manifest lists the exact libraries the actor needs, and those dependencies map directly to capability: **axios** and **form-data** for HTTP and file exfiltration, **screenshot-desktop** for screen capture, **socket.io-client** for realtime C2 channels, and **uuid** for host identification. When **npm install** runs, npm places the actual package code into a **node_modules** folder, which contains the JavaScript, metadata, and any native addons or **.bin** helpers the stealer will require at runtime.

After staging, **svchost.ps1** launches **node.exe** with **.sysupdater.dat** as the argument via a hidden **Start-Process**, handing runtime control to Node while keeping the process windowless. The active stealer logic lives entirely in **.sysupdater.dat**, a heavily obfuscated JavaScript payload interpreted by Node (Figure 6).

***Figure 6: Contents of the .sysupdater.dat file.***

Analysis of **.sysupdater.dat** shows the payload is heavily obfuscated and packed, consistent with output from **obfuscater[.]io**. The sample uses identifier renaming, large encoded string blocks, control flow flattening, dead code injection, and self-defense/debug protections that rely on layered **eval**/**Function** calls and runtime decoding, all intended to slow static analysis and disrupt static review. At runtime the actor sets the process title to **"Node.js Javascript Runtime"** to blend the malicious execution among legitimate processes, further complicating casual observation (Figure 7).

***Figure 7: Node process using a spoofed title.***

The stealer enforces single instance execution by creating a lock file named **cc.pid** in the system temp directory (Figure 8). On startup it checks for the presence of **cc.pid; i**f the file exists, the saved PID is read and probed with **process.kill(pid, 0)** to test whether the process is still alive. If an active process is found, the stealer logs the condition and exits to avoid running a duplicate instance. If the PID is stale, execution continues, and the file is updated with the new process ID. To manage cleanup, the malware registers handlers for normal **exit**, **SIGINT**, and **SIGTERM** that remove the lock file before terminating, ensuring future runs can proceed without interference.

***Figure 8: Lock file logic using cc.pid to enforce single instance execution.***

The stealer embeds a unique identifier, compromised of random characters, which it uses to tag telemetry and exfiltrated data, allowing the operators to associate this compromised host with the specific infection. It implements an asynchronous **makeLog(message)** routine that issues an **HTTP POST** to **hxxp[://]86.106.85[.]234/api/service/makelog,** packaging the supplied message along with the local hostname, the static UID, and a type flag **t: 5** as JSON. This lightweight logging endpoint is used throughout the payload to report status and deliver small pieces of collected data.

All network calls are plaintext HTTP, and the code is defensive against failures. Each Axios promise is followed by a **.catch(() => {})**, and calls are wrapped in an outer try/catch with an empty handler, so network failures or malformed responses do not crash the process. In practice the stealer repeatedly calls **makeLog** to beacon, surface errors, and exfiltrate short text items (for example clipboard contents), enabling the operator to correlate events across compromised hosts using the shared UID and hostname metadata.

The stealer includes an async **setHeader** routine that fingerprints the host and determines whether it is likely running in a virtualized environment. On Windows the code runs **wmic computersystem get model,manufacturer**, on macOS

it calls **system_profiler SPHardwareDataType**, and on Linux it reads **/proc/cpuinfo**; the returned text is lowercased and scanned for indicators such as **vmware**, **virtualbox**, **qemu**, **kvm**, **xen**, **parallels**, and **bochs**, and on Windows it also checks for **microsoft corporation**. When a match is found the payload sets an **isVM** flag to **true** (Figure 9).

***Figure 9: Virtualization check logic inside setHeader function.***

The stealer issues an initial registration **POST** request to **hxxp://86.106.85[.]234/api/service/process/<uid>** over plaintext HTTP, sending a machine profile that includes **os.type()**, **os.platform()**, an OS release annotated with "**(VM)**" or "**(Local)**", the **hostname**, and **os.userInfo() (username, uid/gid, home directory, shell where applicable)**, together with the **static uid** and a type flag **t: 5**.

Network failures during registration are forwarded to **makeLog(e.message)** so errors are reported without interrupting execution. The loader calls **setHeader()** immediately on startup, making this registration the stealer's first beacon and giving the operator a quick inventory of the host and its likely virtualization state (Figure 10).

***Figure 10: Initial registration POST sending host profile to C2.***

The **socketServer** routine turns the stealer into an interactive C2 agent by silently installing **socket.io-client** at runtime and opening a **WebSocket** session to **hxxp://86.106.85[.]234:4552**. The runtime npm install uses noise suppressing flags (**–no-progress, –loglevel silent**) and hides the console on Windows, reducing visible artifacts while ensuring the client library is present even on otherwise clean hosts.

Once loaded, the code establishes a resilient beacon with aggressive reconnection logic, up to 15 reconnect attempts, a 2 second delay between retries, and a 2 second timeout per attempt. This keeps the channel alive without long blocking waits and enabling real-time commands and exfiltration from the operator (Figure 11).

***Figure 11: Runtime installation of socket.io-client and WebSocket C2 connection.***

Once the socket connects the stealer exposes a full set of operator controls via event handlers. The command listener accepts arbitrary shell commands and executes them with **child_process.exec**, suppressing spawned consoles on

Windows and allowing very large outputs via a 300 MB buffer so command results can be returned as **stdout** over the same socket.

Responses, stderr, and error messages are posted back with correlation fields (**cid, sid, code**) and the per-victim uid to preserve session context across requests. A **whour** probe triggers a **whoIm** reply that fingerprints the host via **OS type**, **platform**, **kernel/release**, **hostname**, and **os.userInfo()**, giving the operator a quick inventory for targeting. In practice this yields a fully interactive, outbound only C2 loop that avoids inbound listeners, blends with normal web traffic, and fetches dependencies on demand to reduce static detection surface (Figure 12).

***Figure 12: Socket event handlers enabling command execution and fingerprinting.***

The stealer includes a clipboard watcher that passively taps copied data and sends changes to the operator (Figure 13). After a 3 second startup delay it enters a 500 ms polling loop, using **pbpaste** on macOS and **Get-Clipboard** on Windows to read the clipboard, trimming the result and comparing it to the last captured value. When a change is detected, it waits for a 500 ms debounce, then calls **handleClipboardChange**, which forwards the new clipboard text to the remote log via **makeLog**. On Windows the code hides spawned consoles and ignores **stdio** to minimize visible artifacts during polling, and a **lastClipboardContent** variable prevents duplicate submissions.

***Figure 13: Clipboard watcher loop polling for changes every 500 ms.***

The clipboard watcher doesn't just generate logs; it actively harvests sensitive data copied during normal activity. With polling and debounce both tuned to 500 ms, it captures passwords, MFA codes, API tokens, wallet seeds, and command snippets at near real time without generating noisy duplicates. Errors are caught and passed into **makeLog**, so collection continues even if a read fails, and the reliance on a global **currentClipboardContent** variable, while fragile, is enough to keep the loop running. In practice this design creates a low noise, cross platform clipboard tap that continuously feeds user sourced secrets into the stealer's exfiltration pipeline.

Beyond the clipboard watcher, Blackpoint SOC identified two additional JavaScript artifacts on the compromised device that appear tied to **OtterCookie** but diverge in functionality. These samples suggest the operators were layering or experimenting with multiple tools rather than relying on a single payload. The first of these files, **tmp.js**, acts as another stealer with overlapping behaviors but distinct implementation choices.

**tmp.js** mirrors elements of the main **.sysupdater.dat** payload while introducing its own capabilities (Figure 14). Similarly to the main payload, it sets the process title to "**Node.js Javascript Runtime**" to blend in with legitimate activity, then silently installs **node-global-key-listener**, **screenshot-desktop**, and **sharp** via npm with suppressed output. After a short delay it initializes a global keyboard hook and maintains a plaintext buffer called text that records keystrokes. The routine accounts for modifier keys like **SHIFT** and **CTRL**, translates input into characters, and tags special keys such as **<TAB>**, **<BS>**, arrow keys, and **<CTRL>**…**</CTRL>**, while ignoring mouse events.

***Figure 14: tmp.js code showing process title spoofing and keyboard hook setup.***

On key release **tmp.js** starts a 3 second debounce; when that timer fires it captures a desktop screenshot, resizes the image to a width of 1024 pixels, compresses it to JPEG at quality 60 using sharp, and writes the result to **<tmp>/windows cache/2.jpeg** (Figure 13). The code then uploads the JPEG and the current keystroke buffer to **hxxp://86.106.85[.]234:4558/upload** as multipart form-data, including headers such as **userkey: 507**, **hostname**, **path**, and **t: "5"**. If the upload fails the payload ensures the stash directory **<tmp>/windows cache** exists and appends the keystroke buffer to **1.tmp** for later exfiltration. All these actions are guarded by broad **try**/**catch** blocks, so errors are suppressed, and the process continues running even if dependencies fail to install or individual steps error out.

***Figure 15: Screenshot capture, resize, and upload post-debounce.***

A PowerShell keylogger was also observed, **simple-keyboard-monitor.ps1**, which embeds a small **C# type** via **Add-Type** to **P/Invoke user32.dll** functions (**GetAsyncKeyState** and **GetKeyState**) and then spins an infinite polling loop to capture keystrokes (Figure 16). The mixed PowerShell/C# approach gives the script low level access to keyboard state without relying on Node, and the loop based reader is designed for steady, persistent collection of typed input; outputs are typically buffered and written to disk or posted back to C2 via the same logging/exfil routines used by the JavaScript components.

***Figure 16: PowerShell keylogger using Add-Type to P/Invoke user32.dll.***

The PowerShell keylogger maintains a 256 element boolean array to track key-down state and reads modifier states (**Shift**, **Ctrl**, **Alt**, **CapsLock**), then polls every 10 ms across **virtual-key** codes 0-255 to detect fresh presses. On a new press it converts the **virtual-key** into a printable token using **GetKeyName**, applying **Shift XOR Caps** logic for letter case, mapping digits to their shifted symbols (for example 1 becomes ! when Shift is held), encoding function keys as **[F1]…[F12]**, and normalizing navigation/control keys to tokens like **[TAB]**, **[ENTER]**, **[BACKSPACE]**, and arrow names; numpad keys and punctuation are handled with appropriate shift variants. Each resolved token is emitted to standard output via **Console.WriteLine**, producing a readable stream of keystrokes for collection and exfiltration (Figure 17).

***Figure 17: Keylogger logic converting virtual-key codes to printable tokens.***

## Say Cheese, Comrade: OtterCookie's C2 Exposed

The primary C2 server used for exfiltration by the JavaScript stealer is **86.106.85[.]234**, which was actively reachable during analysis (Figure 18). A lookup on Censys attributes the host to **M247** infrastructure, provisioned out of **Manchester, England.** This indicates the adversary is leveraging a commercial VPS provider rather than bespoke infrastructure.

***Figure 18: Censys result showing 86.106.85[.]234 hosted by M247 in Manchester, UK.***

The Blackpoint SOC also observed an additional C2 endpoint at **78.46.94[.]230**, which resolved to a VPS hosted by **Hetzner Online GmbH** in **Germany** (Figure 19). This second host appears to serve as supplementary infrastructure, indicating the operators maintained more than one exfiltration node across different providers.

***Figure 19: Hetzner VPS at 78.46.94[.]230 identified as secondary C2.***

Notably, the primary C2 **86.106.85[.]234** also exposed SMB and reported its system hostname as **WIN-3O8L0TLB135**, providing a glimpse into the underlying host. Open-source research into that hostname led to the identification of four additional IP addresses registered through the same provider, each resembling clones of the primary C2 server:

- 146.70.87[.]202
- 193.27.14[.]208
- 193.187.148[.]116
- 86.106.85[.]90

This cluster suggests the operators provisioned multiple VPS nodes under the same naming convention to provide redundancy or serve different stages of their campaign.

## Methodology and Attribution

The incident originated from a high-fidelity detection that matched a behavioral signature associated with OtterCookie malware. This alert was notable, as no comparable activity or signature hits had been observed across any other customer environments within the last 90 days. The uniqueness and fidelity of the match prompted a targeted investigation into the payload's lineage and behavioral characteristics.

Analysis revealed that the sample's structure and functionality strongly align with previously documented OtterCookie variants. As outlined in NTT Security's OtterCookie Research the observed source code shares near identical segments with version 4 of OtterCookie, including overlapping function names, logic flow, and exfiltration routines. These consistencies provide high confidence that the sample represents an OtterCookie variant.

Further correlation was established through ANY.RUN's OtterCookie Malware Analysis, which documents the same core TTPs observed in this incident. Both analyses highlight clipboard data exfiltration as a primary function, along with screenshot collection and remote command staging. These behaviors were mirrored in our investigation.

The campaign also used developer-focused lures on Bitbucket and GitHub, centered on a trojanized "chess" project that executed the OtterCookie payload. The actors abused open-source project scaffolding to conceal loader logic, used JavaScript/Node-based staging, and exhibited post-exploitation behavior that closely aligns with Lazarus playbooks targeting engineers.

Attribution is difficult by design because tooling is shared, infrastructure rotates, and indicators can be staged. Even so, the TTPs and IOCs observed here closely match the OtterCookie malware family, which has been repeatedly linked to North Korean operators. Overlapping C2 patterns, Node based loader behavior, and the use of npm-style packaging and dependencies mirror previously documented OtterCookie tradecraft. This indicates strong alignment while stopping short of a definitive attribution.

Additionally, Lazarus group operators have been known to leverage legitimate online tools such as deobfuscate[.]io to obfuscate JavaScript payloads within fake NPM packages and broader malware frameworks like Beavertail, as noted by ANY.RUN. The use of similar obfuscation patterns within this sample aligns with TTPs historically associated with the Lazarus group.

In summary, Blackpoint confidently assesses this sample as an OtterCookie variant based on code and behavioral overlap. While the observed activity shares tradecraft with previously documented Lazarus operations, we do not make definitive attribution to that group with the evidence currently available.

## Final Thoughts

This case illustrates how OtterCookie leveraged a seemingly benign Bitbucket project to stage infection through an intentional try/catch execution path, converting a standard error handler into a remote code execution channel. From there, the malware deployed a trojanized Node module to establish a durable foothold, targeting developer and finance environments where package installs are routine. The malware unpacked a trojanized client, registered persistence via a Run key **WindowsUpdate**, and abused Node to chain into PowerShell for clipboard collection and screenshot capture, later extending coverage with a PowerShell based keylogger. These components worked together to provide continuous visibility into user activity while keeping noise low and communications in clear text.

Blackpoint SOC contained the intrusion, traced back the C2 infrastructure, and the lessons from this campaign now feed into stronger supply-chain defenses and behavior-driven detections tuned for developer endpoints. Treating open-source dependencies as part of the attack surface, and monitoring for behaviors like staged npm installs, fake system directories, and Node invoked with unusual arguments, significantly reduces exposure and accelerates response.

## Recommendations

- Require approval of third-party packages before they are installed or used in development.
- Audit package source code and manifests to ensure nothing malicious has been introduced.
- Provide an internal, vetted repository where developers can pull authorized packages.
- Segment and harden developer workstations to limit access to sensitive production data.
- Restrict and monitor outbound network traffic from developer endpoints to reduce C2 exposure.

- Verify download links and package sources before installation.
- Continuously audit installed software to validate legitimacy and catch unauthorized changes.
- Use the Blackpoint SOC to detect and respond to these types of incidents.

## Indicators of Compromise (IOCs)

| IP | Description |
|---|---|
| 86.106.85[.]234 | Primary Command and Control (C2) |
| 78.46.94[.]230 | Secondary Command and Control (C2) |
| 146.70.87[.]202 | Possible C2 Identified via OSINT |
| 193.27.14[.]208 | Possible C2 Identified via OSINT |
| 193.187.148[.]116 | Possible C2 Identified via OSINT |
| 86.106.85[.]90 | Possible C2 Identified via OSINT |

| Filename | Hash |
|---|---|
| svchost.bat | 57533E0BF2A9857BF1E603039B6B3E9EEB9CB5B53BB490D4ECDAA57EFAD0D27C |
| simple-keyboard-monitor.ps1 | ADEE6C5CC15432F0E4A5202B2653AEB057D988DC18ECDCDED7038B91BD8212B2 |
| svchost.ps1 | 8B3DEB9426B405EB8E08AC2A9868E55980A3A24B7F033E4040A08C029D721894 |
| cc.pid | C077F24E0C8FD9B10AF046F7811046BC97FE9723A354FAE129FD49720DA5C87E |
| tmp.js | C0CEC1CA432EB8AE0CB43325CA10C25B436EE88EDD6F08E4B74BC1EE27E83766 |
| package.json | 51322339B720D6E81BEF2B7D415C242E222939C0A3624C5EDC791DB9472F7EC8 |
| .sysupdater.dat | 8D3B4D38914029C2B9CF83F6DED99FA1E73F89FC390FFF369BDDEDAB4729F04D |

| IOC | Explanation |
|---|---|
| labs525 | Malicious User tied to BitBucket Link |
| terin6 | User tied to malicious App hosting in Github |
| hxxps[://]bitbucket[.]org/labs525/chess/src/main/ | Malicious BitBucket Repo |
| hxxps[://]github[.]com/terin6/CHESS | Malicious Github Repo |

---

Source: https://blackpointcyber.com/blog/malicious-node-package-deploys-ottercookie/