How a Trusted Dependency Turned Into a Silent Backdoor

Reproducing the Axios Supply Chain Attack — A Bytes Encrypt Lab

---

Introduction

Everything worked — the API responded, the UI appeared normal, and no alerts were triggered. And that was precisely the problem.

On March 31, 2026, malicious versions of the widely used axios package briefly appeared on npm. These versions behaved exactly like the original library while silently executing hidden logic in the background.

At **Bytes Encrypt**, we didn't just review the advisory. We reproduced the attack in a controlled lab to observe how it behaves in a real application.

Incident Overview

The compromised versions:

- axios@1.14.1
- axios@0.30.4

What made this dangerous:

- No corresponding GitHub release
- A hidden dependency (plain-crypto-js)
- Execution triggered during installation

The injected package used a postinstall script to execute obfuscated code, fetch a payload, and remove traces — leaving almost no visible evidence.

Lab Architecture (Bytes Encrypt)

This lab simulates a real-world supply chain attack environment:

- Frontend (User Interface) → :3001
- Backend (Node.js — vulnerable) → :5001

Bytes Encrypt Pvt . Ltd

- Attacker Server (Collector) → :8001

The key idea:

The application behaves normally, while a hidden execution path sends data to the attacker.

Attack Flow

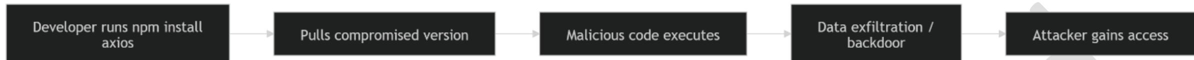
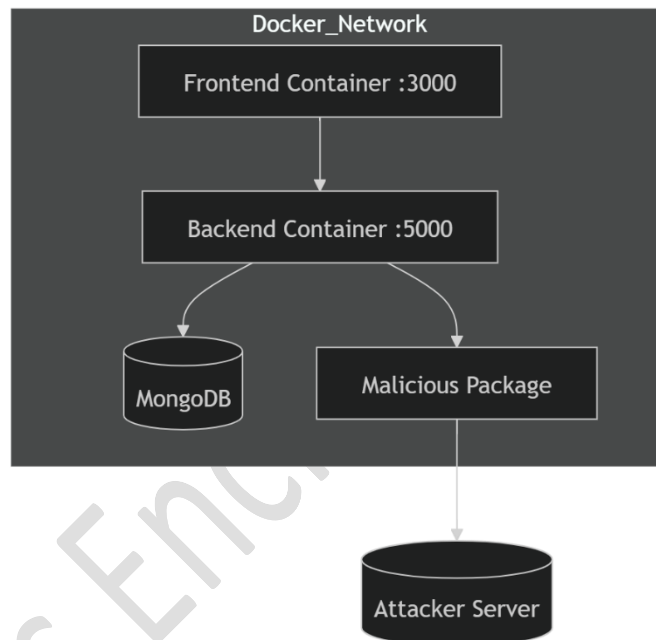


Figure 1:End-to-end attack flow illustrating how a compromised dependency is introduced during installation, executes automatically, and silently exfiltrates data while the application continues to



function normally.

Investigation Walkthrough

Bytes Encrypt Pvt. Ltd

Step 1 — Initial State (Everything Looks Normal)

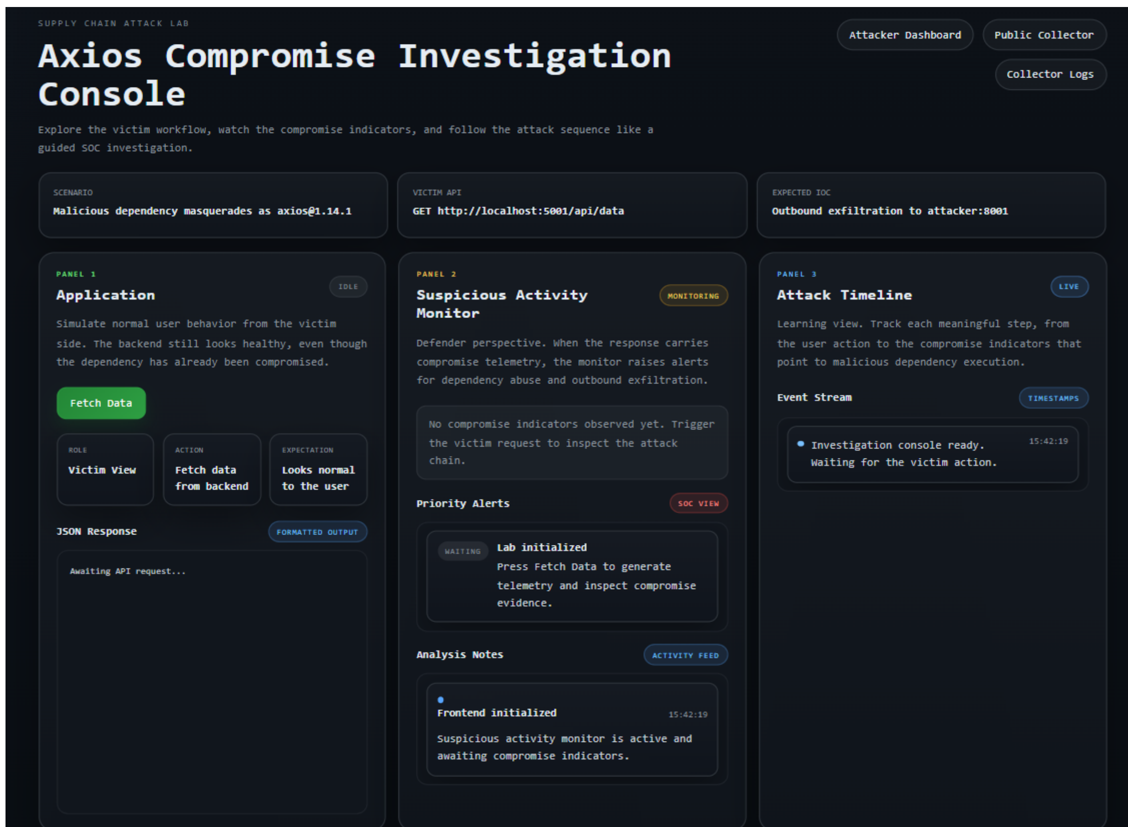


Figure 2: The application is in a clean state before any user interaction. No alerts are present, and no suspicious activity has been recorded. User opens the application. No alerts. No suspicious activity.

What this shows:

- Application is running normally.
- No compromise indicators yet
- User is about to trigger the request.

Key Insight:

At this stage, the system appears completely safe — which is exactly what makes this attack effective.

Step 2 — User Action (Fetch Data)

User clicks:

Fetch Data

The backend processes the request and returns a valid response.

BytesEncrypt

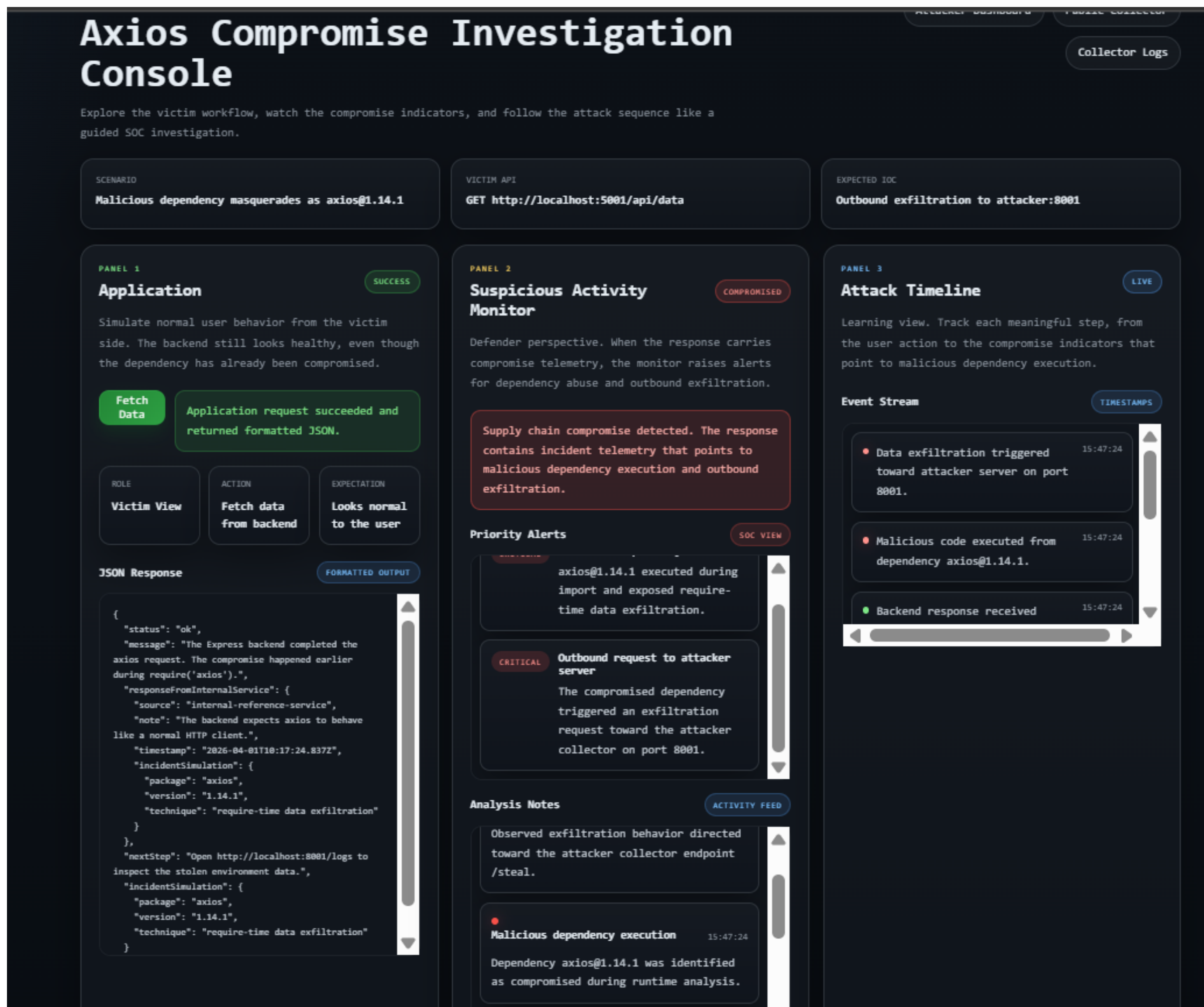


Figure 3:The request completes successfully, returning valid data while simultaneously flagging the system as compromised.

What this shows:

- Application reports **SUCCESS**
- JSON response is correct.
- But security panel shows **COMPROMISED.**

Key Insight:

This is the core of the attack:

The application works correctly while being compromised at the same time.

Step 3 — Hidden Execution (What You Don't See)

Behind the scenes, the dependency triggers:

Bytes Encrypt

- Execution during import/startup
- Execution during API request

This leads to silent data exfiltration.

Example behaviour:

{

  "event": "dependency_request_exfiltration",

  "package": "axios@1.14.1"

}

Key Insight:

This logic is not part of your application — it runs inside a trusted dependency.

Step 4 — Attacker Perspective

While the user sees a normal response, the attacker receives:

- Environment variables
- API keys
- Internal service URLs

Bytes Encrypt Pvt. Ltd

ATTACKER INFRASTRUCTURE

Attacker Dashboard

Collector Home | Dashboard | Public View | Log View | GET /steal

Review sanitized captured payloads, correlate exfiltration events, and inspect highlighted demo keys without exposing personal machine details.

TOTAL EVENTS
1

LATEST CAPTURE
2026-04-01T10:11:55.506Z

FLAGS OBSERVED
1

Exfiltration Events

EVENT FEED

Timeline-friendly summaries of the sanitized payloads received by the attacker collector.

Event 1

FLAG CAPTURED

2026-04-01T10:11:55.506Z

dependency_startup_exfiltration from axios@1.14.1

SOURCE IP
redacted-for-public-view

USER AGENT
axios/1.14.1

SECRET_KEY
FLAG{supply_chain_node_attack}

Captured Data

FORENSICS

Pretty formatted JSON with demo keys and the lab flag highlighted in red.

Captured Payload 1

JSON EVIDENCE

2026-04-01T10:11:55.506Z


```
{
  "receivedAt": "2026-04-01T10:11:55.506Z",
  "sourceIp": "redacted-for-public-view",
  "userAgent": "axios/1.14.1",
  "payload": {
    "event": "dependency_startup_exfiltration",
    "package": {
      "name": "axios",
      "version": "1.14.1"
    },
    "collectedAt": "2026-04-01T10:11:55.423Z",
    "environment": {
      "AXIOS_EXFIL_URL": "http://attacker:8001/steal",
      "BACKEND_INTERNAL_URL": "http://127.0.0.1:5001/internal/reference-data",
      "BACKEND_PORT": "5001",
      "SECRET_KEY": "FLAG{supply_chain_node_attack}",
      "OPENAI_API_KEY": "sk-demo-supplychain-collector-000000000000",
      "AWS_ACCESS_KEY_ID": "AKIADEMOATTACKLAB01",
      "AWS_SECRET_ACCESS_KEY": "demo/aws/secret/for/public/log-view",
```


Figure 4: The attacker dashboard displays multiple exfiltration events triggered by backend activity.

What this shows:

- Multiple exfiltration events
- Timestamped attack timeline
- Captured sensitive data.

Key Insight:

Every user action results in attacker-side data capture — without affecting the frontend.

Step 5 — Log Evidence (Ground Truth)

Now we verify what happened at runtime.

Bytes Encrypt

```
attacker-1  | [attacker] captured sanitized payload
attacker-1  | {
attacker-1  |   "receivedAt": "2026-04-01T10:27:02.295Z",
attacker-1  |   "sourceIp": "redacted-for-public-view",
attacker-1  |   "userAgent": "axios/1.14.1",
attacker-1  |   "payload": {
attacker-1  |     "event": "dependency_request_exfiltration",
attacker-1  |     "package": {
attacker-1  |       "name": "axios",
attacker-1  |       "version": "1.14.1"
attacker-1  |     },
attacker-1  |     "collectedAt": "2026-04-01T10:27:02.149Z",
attacker-1  |     "environment": {
attacker-1  |       "AXIOS_EXFIL_URL": "http://attacker:8001/steal",
attacker-1  |       "BACKEND_INTERNAL_URL": "http://127.0.0.1:5001/internal/reference-data",
attacker-1  |       "BACKEND_PORT": "5001",
attacker-1  |       "SECRET_KEY": "FLAG{supply_chain_node_attack}",
attacker-1  |       "OPENAI_API_KEY": "sk-demo-supplychain-collector-000000000000",
attacker-1  |       "AWS_ACCESS_KEY_ID": "AKIADEMOATTACKLAB01",
attacker-1  |       "AWS_SECRET_ACCESS_KEY": "demo/aws/secret/for/public/log-view",
attacker-1  |       "GITHUB_TOKEN": "ghp_demo_supply_chain_public_token_12345",
attacker-1  |       "NPM_TOKEN": "npm_demo_supply_chain_registry_token"
attacker-1  |     },
attacker-1  |     "system": {
attacker-1  |       "hostname": "victim-host-redacted",
attacker-1  |       "platform": "linux",
attacker-1  |       "release": "redacted",
attacker-1  |       "arch": "x64",
attacker-1  |       "uptime": 0,
attacker-1  |       "username": "lab-user"
attacker-1  |     },
attacker-1  |     "execution": {
attacker-1  |       "cwd": "/app/backend",
attacker-1  |       "pid": 1,
attacker-1  |       "ppid": 0,
attacker-1  |       "argv": [
attacker-1  |         "node",
attacker-1  |         "server.js"
attacker-1  |       ]
attacker-1  |     },
attacker-1  |     "redaction": {
attacker-1  |       "personalInfoHidden": true,
attacker-1  |       "originalEnvKeyCount": 12,
attacker-1  |       "displayedDemoKeys": [
attacker-1  |         "SECRET_KEY",
attacker-1  |         "OPENAI_API_KEY",
attacker-1  |         "AWS_ACCESS_KEY_ID",
attacker-1  |         "AWS_SECRET_ACCESS_KEY",
```


Figure 5: Raw logs showing outbound requests containing environment data and internal context.

What this shows:

- Outbound requests to attacker server
- Captured environment variables.
- Secrets like API keys and tokens

Key Insight:

This is the strongest evidence:

- The backend is actively sending sensitive data outside the system.

Step 6 — Endpoint Verification

Checking the attacker endpoint:

```
POST endpoint for exfiltration
```

Figure 6: The /steal endpoint responds with a minimal message, indicating that it is designed primarily for POST-based data collection.

What this shows:

- Endpoint exists but appears harmless.
- Only accepts POST requests for real exfiltration.

Key Insight:
The attacker keeps the endpoint minimal to avoid detection.

Why This Attack Is Dangerous

This attack does not break your system.

Instead, it:

- Preserves normal functionality.
- Executes malicious logic silently.
- Uses your backend as a data pipeline.

The system becomes both the victim and the attacker's tool.

Detection Insights

Based on the lab:

Network Level

- Unexpected outbound POST requests

Bytes Encrypt Pvt. L

- Communication with unknown services

Dependency Level

- Unexpected packages (plain-crypto-js)
- Version mismatches

Runtime Level

- Code execution on import.
- Behaviour outside business logic

Remediation

1. Downgrade immediately

- npm install axios@1.14.0

2. Check dependencies

- grep -r "plain-crypto-js" package-lock.json

3. Rotate secrets

- Assume all exposed credentials are compromised.

4. Rebuild clean environments

5. Enforce safe installs

Indicators of Compromise

- plain-crypto-js@4.2.1 present
- Outbound requests to attacker endpoints
- Execution during npm install
- Unexpected environment data exposure

Final Takeaway

This attack didn't break the application.

Bytes Encrypt Pvt . Ltd

It used it.

The UI stayed clean.

The API worked normally.

The compromise remained invisible.

Until you looked at the logs.

One Line That Matters

If your dependency executes code, your supply chain is part of your attack surface.

Conclusion

The Axios supply chain compromise demonstrates how modern attacks leverage trust rather than exploiting application logic. By embedding malicious behaviour within a widely used dependency, the attack was able to operate without disrupting normal functionality.

As observed in the Bytes Encrypt lab, the application continued to return valid responses while sensitive data was simultaneously transmitted to an external system. This disconnect between visible behaviour and underlying activity makes such attacks particularly difficult to detect using conventional testing or monitoring approaches.

The key takeaway is clear: application correctness does not guarantee security.
Effective defence requires visibility beyond the application layer — including dependency integrity, runtime behaviour, and outbound network activity.

Methodology

This analysis is based on:

- Public incident disclosures and vendor advisories
- Controlled lab reproduction conducted by Bytes Encrypt
- Runtime observation of dependency behaviour, network activity, and exfiltration patterns

---

References

Bytes Encrypt Pvt . Ltd

1. Vercel

   Axios Package Compromise and Remediation Steps

   Available at: https://vercel.com/changelog/axios-package-compromise-and-remediation-steps

2. Socket Research Team

   Axios npm Package Compromised — Detailed Analysis

   Available at: https://socket.dev/blog/axios-npm-package-compromised

3. npm Documentation

   package-lock.json and Deterministic Installs

   Available at: https://docs.npmjs.com/cli/v10/configuring-npm/package-lock-json

4. OpenSSF

   Open-Source Supply Chain Security Guidance

   Available at: https://openssf.org/

Bytes Encrypt Pvt . Ltd