Void Dokkaebi Uses Fake Job Interview Lure to Spread Malware via Code Repositories

By By: Lucas Silva Apr 21, 2026 Read time: 11 min (3007 words)

Published: 2026-04-21 · Archived: 2026-05-06 02:01:41 UTC

Key takeaways

- Void Dokkaebi (aka Famous Chollima) has evolved beyond single-target social engineering into a self-propagating supply chain threat. A compromised developer's repository becomes an infection vector for the next wave of victims, creating a worm-like propagation chain through the developer ecosystem.
- The campaign spreads through trusted development workflows, using malicious VS Code tasks and injected code that can execute during normal development activity. When compromised code reaches organizational or popular open-source repositories, contributors, forks, and downstream projects can also be exposed.
- Analysis in March 2026 identified more than 750 infected repositories, over 500 malicious VS Code task configurations, and 101 instances of the commit tampering tool. Repositories belonging to organizations such as DataStax and Neutralinojs were also identified carrying infection markers.
- The campaign uses blockchain infrastructure for payload staging, including Tron, Aptos, and Binance Smart Chain, which puts parts of its delivery infrastructure beyond traditional takedowns.

Introduction

Void Dokkaebi, also tracked as Famous Chollima, is a North Korea-aligned intrusion set that systematically targets software developers who hold cryptocurrency wallet credentials, signing keys, and access to continuous integration/continuous delivery (CI/CD) pipelines and production infrastructure. As previously documented by TrendAI™ Research, the group poses as recruiters from cryptocurrency and AI firms, luring developers into cloning and executing code repositories as part of fabricated job interviews. This is a pattern independently tracked across the industryopen on a new tab since 2024, but less attention has been paid to what happens after the initial compromise.

Our analysis reveals that Void Dokkaebi's operations do not end with a single infected developer. The compromised machine becomes a launchpad, with the threat actor weaponizing the victim's own repositories and turning their code contributions into infection vectors for downstream developers. The result is a self-sustaining propagation chain resembling a worm's behavior rather than a traditional targeted attack. This report details the propagation model, the malware it delivers, the scale of contamination we observed, and what organizations can do about it.

The infection paths and how the worm spreads

The propagation relies on two distinct mechanisms that work in tandem. The first spreads passively through Visual Studio Code (VS Code) workspace configurations that travel with committed code. The second is an active technique where the threat actor, having gained remote access to a developer's machine, injects obfuscated JavaScript into the victim's repositories and rewrites git history to conceal the tampering.

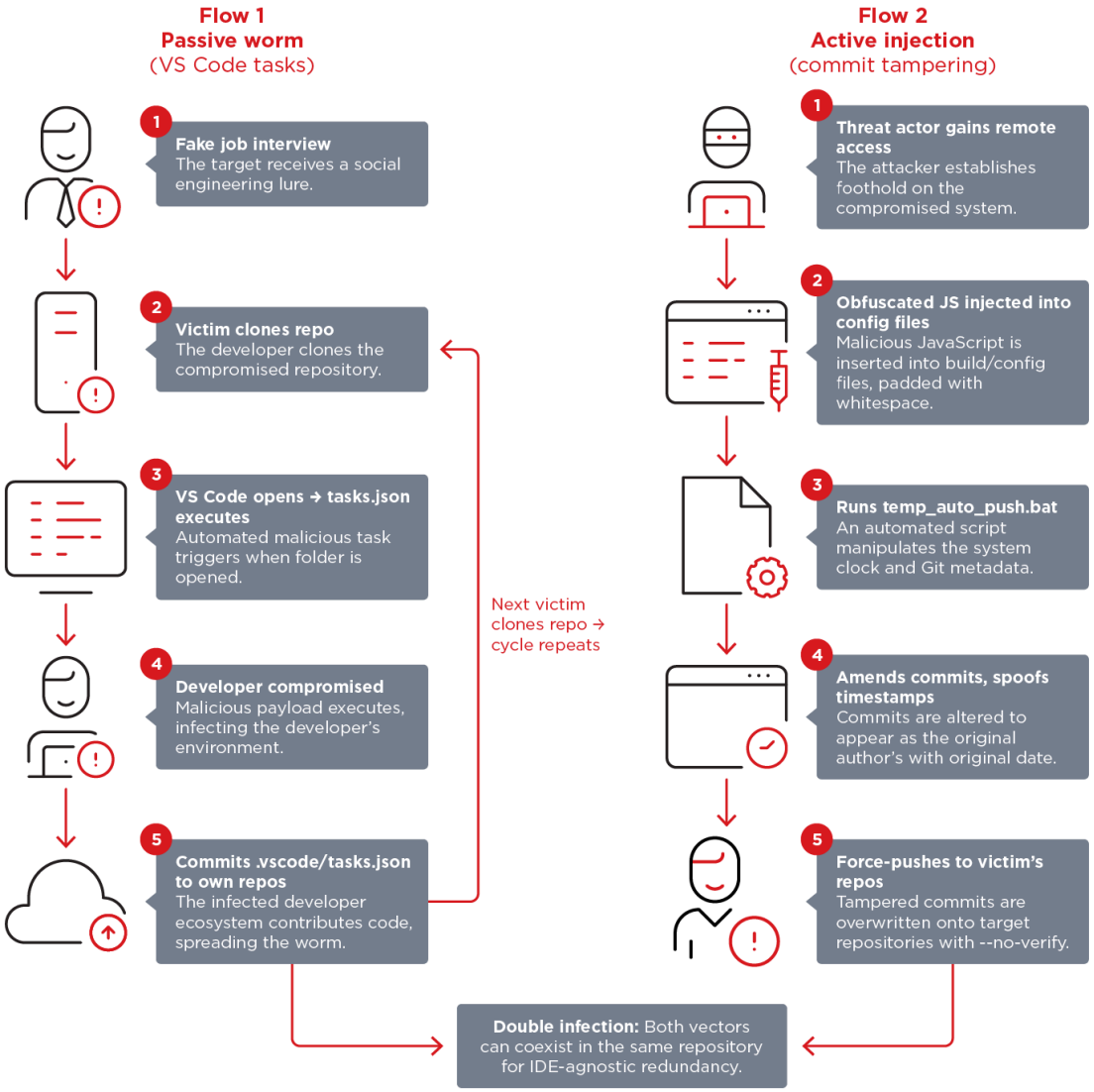

Figure 1. Infection paths used by Void Dokkaebi, with the first flow done via VSCode and the second via active injection

The initial infection begins with a fabricated job interview where the victim is asked to clone a code repository and review or run it as part of a technical assessment. The repositories are hosted on GitHub, GitLab, or Bitbucket, and appear to be legitimate coding projects. The delivery mechanism abuses VS Code's workspace task system, a technique that has been independently documented by Microsoftopen on a new tab, OpenSourceMalwareopen on a new tab, and Abstract Securityopen on a new tab.

The attack works as follows:

1. The repository contains a .vscode/tasks.json file with a task configured to run automatically when the workspace is opened (runOn: folderOpen).

2. When the victim opens the project in VS Code and accepts the workspace's trust prompt, the task executes without further interaction.
3. In some cases, the task fetches the backdoor directly from a remote URL. In others, it launches a font or image file bundled in the repository that contains the malicious payload, a different execution variant that achieves the same result.

The developer's ecosystem is compromised at this point, but the worm-like behavior begins when the victim commits that code to GitHub. Whether pushing the project itself or reusing components in other work, the malicious .vscode/tasks.json is committed along with it. The .vscode folder is hidden by default in file explorers and is commonly absent from .gitignore files, making it an effective trojan horse. Any developer who subsequently clones that repository and opens it in VS Code receives the same trust prompt. If accepted, the cycle repeats.

This creates a self-propagating chain. Each compromised developer seeds new repositories with the infection vector, and each new victim becomes a potential distributor. Unlike traditional social engineering where the attack ends with the initial target, here the range of infection expands with every commit.

Flow 2: Active injection and commit tampering

In parallel, we observed a second propagation mechanism. Users who were already compromised by Void Dokkaebi had multistage obfuscated JavaScript code added to the source code files in their repositories.

The threat actor targets various configuration files and common entry points, choosing files that developers are less likely to scrutinize closely. The obfuscated JavaScript, which functions as the multistage loader described in the next section, is added to the end of the file.

Whitespace is often added to push this additional code to the right edge of the screen and make it invisible during casual code review, or when inspecting code differences. Because these configuration files are evaluated as JavaScript by Node.js tooling (e.g., build tools, linters, bundlers), any code appended to them executes automatically whenever the corresponding tool runs.

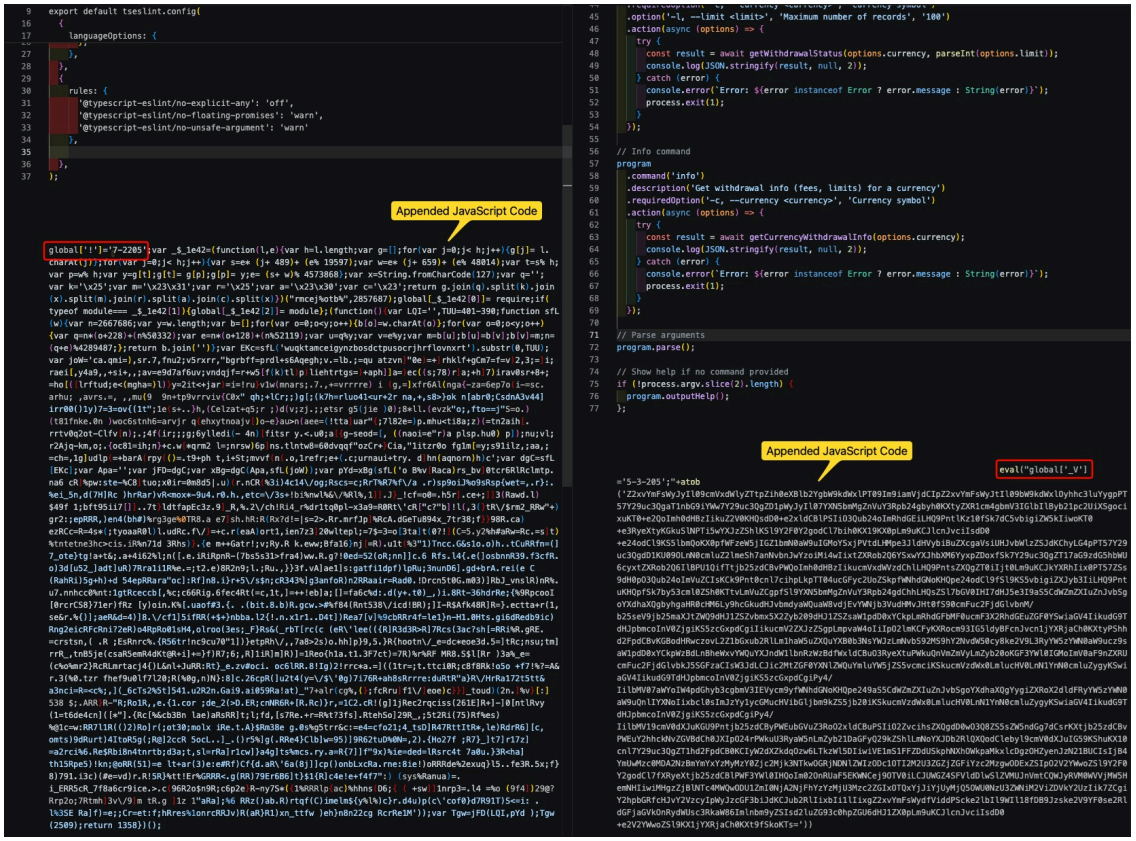

Figure 2. Code snippets of a GitHub repository containing the injected code

Both the code injection and its concealment are performed remotely through malware installed during the initial compromise (flow 1). To conceal the tampering, the threat actor employs a Windows batch script (temp_auto_push.bat) that extracts metadata from the most recent Git commit, its date, time, author name, email, and commit message.

They then temporarily alter the system clock to match the original commit's timestamp. The modified commit is amended with the injected code while preserving all original authorship metadata. The --no-verify flag is used to bypass pre-commit hooks and CI/CD security checks. After the amendment, the system clock is restored, and the rewritten commit is force-pushed to the remote branch. In the git history, the attacker appears indistinguishable from the original commit.

```
@echo off
for /f "delims=" %%A in ('cmd /c "git log -1 --date=format-local:%%Y-%%m-%%d --format=%%cd"') do set LAST_COMMIT_DATE=%%A
for /f "delims=" %%A in ('cmd /c "git log -1 --date=format-local:%%H:%%M:%%S --format=%%cd"') do set LAST_COMMIT_TIME=%%A
for /f "delims=" %%A in ('cmd /c "git log -1 --format=%%s"') do set LAST_COMMIT_TEXT=%%A
for /f "delims=" %%A in ('cmd /c "git log -1 --format=%%an"') do set USER_NAME=%%A
for /f "delims=" %%A in ('cmd /c "git log -1 --format=%%ae"') do set USER_EMAIL=%%A
for /f "delims=" %%A in ('git rev-parse --abbrev-ref HEAD') do set CURRENT_BRANCH=%%A
echo %LAST_COMMIT_DATE% %LAST_COMMIT_TIME%
echo %LAST_COMMIT_TEXT%
echo %USER_NAME% (%USER_EMAIL%)
echo Branch: %CURRENT_BRANCH%
set CURRENT_DATE=%date%
set CURRENT_TIME=%time%
date %LAST_COMMIT_DATE%
time %LAST_COMMIT_TIME%
echo Date temporarily changed to %LAST_COMMIT_DATE% %LAST_COMMIT_TIME%
git config --local user.name %USER_NAME%
git config --local user.email %USER_EMAIL%
git add .
git commit --amend -m "%LAST_COMMIT_TEXT%" --no-verify
date %CURRENT_DATE%
time %CURRENT_TIME%
echo Date restored to %CURRENT_DATE% %CURRENT_TIME% and complete amend last commit
git push -uf origin %CURRENT_BRANCH% --no-verify
@echo on
```

Figure 3. Code snippets of the temp_auto_push.bat script

In some compromised repositories, we observed both techniques being present simultaneously (i.e., the malicious .vscode/tasks.json alongside the appended obfuscated JavaScript). We believe that there were cases where developers fell victim to *both* propagation methods separately, but also cases where the attackers used both techniques on one victim.

This “double infection” mechanism provides redundancy. The tasks.json catches developers using VS Code (triggering on folder open), while the injected JavaScript executes for anyone who builds or runs the project regardless of their IDE. Together, they guarantee malware execution.

The organizational amplifier

The worm-like propagation poses higher risk when it reaches developers with commit access to organizational or popular open-source repositories. We identified compromised repositories belonging to the following organizations:

- DataStaxopen on a new tab: At least five repositories found compromised between January 31 and February 3, 2026, which have since been cleaned.
- Neutralinojsopen on a new tab: They had 8,400 stars and 495 forks, where all four repositories were force-pushed with malicious commits in a single automated burst on March 2, 2026. The commits were backdated between 5 and 35 days to blend with legitimate history, and the attack went undetected for 3 days until identified and remediated by the OpenSourceMalware teamopen on a new tab.

These organizations were found carrying malicious code snippets consistent with these techniques. While we cannot confirm the exact chain of events within these organizations, the indicators are consistent with a scenario where a contributor with commit access was first compromised through the social engineering lure (flow 1), which subsequently enabled the infection of the organizational repositories (flow 2). Once a repository of this scale is compromised, every contributor, every fork, and every downstream project that depends on it becomes a potential victim. This amplifies the scope of the campaign from a single developer to an entire ecosystem.

This propagation model is fundamentally different from traditional supply chain attacks, such as the SolarWinds incident that required the compromise of the build infrastructure. Here, no build system is breached. The attack exploits something far simpler:

- Developer workflow habits
- The tendency to not include .vscode folders in gitignore
- Not reviewing configuration files line by line
- Trusting the contents of their own repositories.

It is also distinct from traditional network worms, which exploit software vulnerabilities to propagate. This campaign propagates through trust in development tools, in colleagues' commits, and in open-source projects.

With the propagation model established, we now turn to the malware that these infection vectors deliver.

The malware in brief: DEV#POPPER RAT variant

The tasks.json vector (flow 1) acts as a straightforward downloader, fetching and executing a payload from a remote URL or bundled file. However, the obfuscated JavaScript injected into source code files (flow 2) is part of a more complex approach. It functions as a multistage loader, which is designed to retrieve and execute payloads from blockchain infrastructure. It progresses through four stages, each employing layers of string shuffling, hexadecimal obfuscation, and character swap algorithms to hinder analysis.

The loader queries the Tron blockchain API to fetch a transaction from a hardcoded wallet address. The data extracted from this transaction is used as a reference key to retrieve an encrypted payload from a Binance Smart Chain (BSC) transaction's input data field. If the Tron query fails, the loader falls back to the Aptos blockchain as an alternative data source.

The retrieved payload is XOR-decrypted using a hardcoded key and executed via eval() or by spawning a persistent hidden background process. Across stages, the loader rotates wallet addresses and transaction hashes, allowing each stage to independently update its pointers by simply posting a new transaction to the corresponding blockchain without modifying the malware's code.

This blockchain-based staging mechanism is particularly significant because it functions as a general-purpose delivery platform. Since the payload is retrieved dynamically from immutable blockchain transactions, the threat actor can deliver any malware from their toolset by simply updating the blockchain reference, including other malware that have been linked to North Korea, such as InvisibleFerret, OtterCookie, OmniStealer, DEV#POPPER, and BeaverTail, all of which have been observed in Void Dokkaebi's operations. A single infected repository can serve as a delivery vector for different payloads at different times, depending on the threat actor's operational objectives.

DEV#POPPER RAT

One of the payloads delivered through this infrastructure is a variant of the DEV#POPPER RAT (version marker 260311), a cross-platform Node.js remote access trojan (RAT) previously documented by eSentireopen on a new tab.

The variant we analyzed introduces a multi-operator session management system, where several operators can work on a compromised machine simultaneously through independent command queues. This indicates team-based operations rather than a single attacker.

The backdoor communicates with its command-and-control (C&C) server via WebSocket (using socket.io-client). It uses HTTP for file uploads, directory exfiltration, and logging, specifically through the '/verify-human/[VERSION]' endpoint for heartbeat and notification, and '/u/f' for data exfiltration.

These distinctive network patterns provide researchers and analysts with reliable signatures for identifying infected devices. WebSocket connections to unexpected endpoints combined with HTTP traffic matching these URL patterns on developer workstations are strong indicators of compromise.

Two aspects of this variant are directly relevant to the propagation model:

- The RAT specifically detects and avoids CI/CD environments (e.g., GitLab CI, BuildBot) and cloud sandboxes, executing only on real developer workstations. This means automated pipeline scanning will miss it entirely.
- For persistence, it injects versioned code (markers: C250617A through C250620A) into developer applications (e.g., Antigravity, VS Code, Cursor, Discord, GitHub Desktop) and creates a hidden .node_modules folder for Node.js module search order hijacking. This persistence into developer tooling creates additional opportunities for the worm-like propagation described earlier.

The scale of contamination

To quantify the campaign's reach, we scanned public code hosting platforms in late March 2026. The following statistics provide a snapshot of the contamination across public repositories.

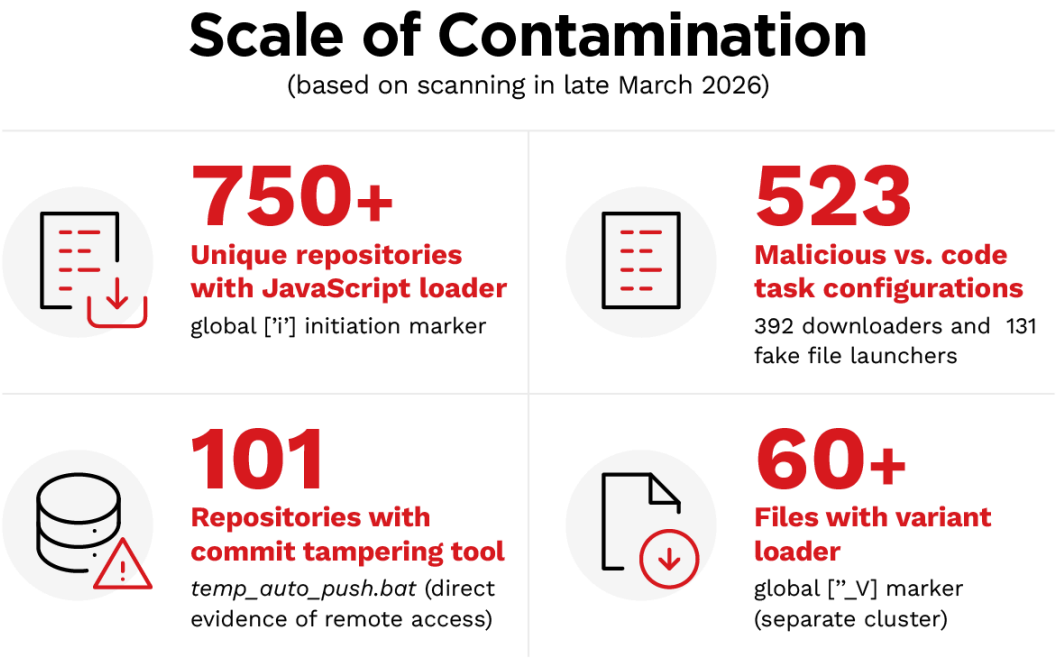

Figure 4. The scale of infection as per TrendAI™ Research analysis in late March

Our scan identified over 750 unique repositories containing the obfuscated JavaScript loader, identified by the global['!'] initialization marker. However, in several instances, multiple repositories were infected, resulting in a higher number of source code files being compromised.

An additional 60 source code files carried a variant loader using a different marker, (global['_V']). This suggests either an evolution of the tooling or a separate Void Dokkaebi cluster operating with modified tooling, a pattern of operational autonomy we also observed in the RAT variants discussed in earlier sections.

On the VS Code infection vector, we found 392 .vscode/tasks.json files configured as downloaders and 131 additional tasks.json files launching fake font or image files. They are a different execution variant but have the same underlying infection mechanism. Any developer who clones the containing repository and opens it in VS Code will be prompted to trust the workspace. If accepted, the malicious task executes automatically.

We found the commit tampering tool (temp_auto_push.bat) in at least 101 repositories. This is direct evidence that the threat actor had active remote access to the developer's machine and deliberately chose to weaponize their repositories, regardless of whether the injected JavaScript was still present at the time of our scan.

The infected repositories (numbering more than 750) include threat actor-operated repositories and legitimately compromised users' repositories, and possibly a mix of both. We cannot determine in every case whether the injection was performed via the commit tampering tool or through direct file manipulation.

These numbers represent what was visible on public code repositories at the time of our analysis. The actual scale of contamination is likely larger. Figure 4 does not include repositories that were identified and cleaned before our scan, private repositories not indexed by public search, and forks or clones that propagated the infection to environments we cannot observe.

The numbers also represent the downstream impact of cascading propagation and not the number of individually compromised developers. A single compromised developer can infect multiple files across several repositories. Each compromised developer's repositories became the infection source for the next wave of victims. When viewed alongside the organizational cases discussed earlier, this points to a self-sustaining campaign where a relatively small, initial investment in social engineering can produce a large infection surface.

Actionable guidance

The following recommendations directly address the mechanisms Void Dokkaebi relies on for propagation and persistence. These are prioritized by impact:

- Use isolated environments for interview coding assignments. Never execute code from job interviews on production or personal machines. Use disposable virtual environments destroyed after the assessment. This is the single most effective way to prevent the initial compromise.
- Add .vscode/ to .gitignore and enforce this across organizational repositories. This breaks the passive worm propagation vector entirely.
- Enforce branch protection and signed commits. Block force pushes, require pull requests, and require GPG- or SSH-signed commits. The commit-tampering tool relies on git push --force and cannot forge

cryptographic signatures. These controls directly neutralize it.

- Audit repositories for known infection markers. Search for global['!'] and global['_V'] in source code files, and check for temp_auto_push.bat. If found, assume that the developer workstation was compromised. Isolate the machine, revoke credentials, and notify collaborators and downstream consumers.
- Scrutinize configuration file changes. Files such as postcss.config.mjs, tailwind.config.js, eslint.config.mjs, and next.config.mjs are targeted because they are rarely reviewed closely. Inspect for content appended beyond the visible area of the screen. Apply the same review rigor as application source code.
- Monitor for blockchain API and C&C traffic. Outbound connections to api.trongrid.io, fullnode.mainnet.aptoslabs.com, and Binance Smart Chain RPC endpoints from developer workstations are high-confidence indicators. Additionally, monitor for connections to MongoDB port 27017 and HTTP patterns /u/f and /verify-human/[VERSION].
- Do not rely solely on CI/CD pipeline scanning. The RAT detects and avoids CI/CD environments. Endpoint-level detection on developer workstations is essential.
- Treat VS Code workspace trust prompts as a security decision. Inspect .vscode/tasks.json for runOn: folderOpen tasks before granting trust.
- Apply network-level blocks for known infrastructure. The section on indicators of compromise (IoCs) lists Vercel-hosted downloaders, URL shortener redirectors, and C&C addresses associated with this campaign. These IoCs can support detection and threat hunting for related activity across the environment.
- Include interview-based social engineering in security awareness training. The attack pattern, “clone and run this repo as part of your interview,” should be part of developer-focused security programs.

Conclusion

Void Dokkaebi’s recent activities represent a shift in how supply chain attacks can operate. Rather than compromising build systems or package registries, the threat actor exploits the trust developers place in their own tools, their colleagues’ commits, and the open-source projects they depend on. A single compromised developer becomes the seed for an infection that propagates across personal repositories, organizational codebases, and popular open-source projects without requiring any further social engineering.

The scale of infection also confirms that this is an active and expanding campaign. Organizations that treat developer workstations and repository workflows as part of their attack surface will be better positioned to detect and disrupt this threat before it propagates.

TrendAI™ Research will keep tracking Void Dokkaebi and related campaigns, delivering actionable intelligence so that your organization stays ahead of emerging threats. Our threat intelligence, paired with advanced detection capabilities, helps keep your organization protected against sophisticated attacks going after cryptocurrency assets and sensitive enterprise data.

TrendAI Vision One™ customers are protected from the IoCs listed in the table below.

TrendAI Vision One™ Threat Intelligence Hub

TrendAI Vision One™ Threat Intelligence Hubproducts provides the latest insights on emerging threats and threat actors, exclusive strategic reports from TrendAI™ Research, and TrendAI Vision One™ Threat Intelligence Feed

in the TrendAI Vision One™ platform.

Emerging Threats: Fake Interview Lures Used by Void Dokkaebi to Spread Malware Through Git Repositories

Threat Actor: Void Dokkaebi

TrendAI Vision One™ Intelligence Reports (IOC Sweeping)

Fake Interview Lures Used by Void Dokkaebi to Spread Malware Through Git Repositories

Hunting Queries

TrendAI Vision One™ Search App

TrendAI Vision One™ customers can use the Search App to match or hunt the malicious indicators mentioned in this blog post with data in their environment.

Pontential Code Commit File Created

eventSubId: 101 AND objectFilePath: temp_auto_push.bat

Outbound Connection to Potential Void Dokkaebi C2
eventSubId: (204 OR 301) AND dst: (136.0.9.8 OR 198.105.127.210 OR 23.27.202.27 OR 154.91.0.196 OR 23.27.20.143 OR 85.239.62.36 OR 83.168.68.219 OR 166.88.4.2 OR 23.27.120.142)

More hunting queries are available for TrendAI Vision One™ with Threat Intelligence Hub entitlement enabled.

Indicators of Compromise (IoCs)

The indicators of compromise for this entry can be found here.

Tags
