GITHUB ACTIONS WORM

Securing the Codebase: Unraveling the GitHub Actions Worm and Safeguarding Repositories via Actions Dependency Tree Analysis in DevSecOps

WWW.DEVSECOPSGUIDES.COM

GitHub Actions Worm

GitHub's CI/CD platform, GitHub Actions, has recently become a target for a sophisticated attack vector, posing threats to both open-source projects and internal repositories. In this article, we will explore the technical intricacies of this threat and provide step-by-step mitigation strategies with commands and code snippets.

Initial Attack Vectors

```
              +-------------------+
              | Initial Attack  |
              |    Vectors       |
              +-------------------+
                      |
                      |
                      |
       +---------------+---------------+
       |                            |
       v                            v
+-----------------+      +------------------------+
|   Repojacking  |      | NPM Package Maintainer |
|                 |      |    Email Hijacking     |
+-----------------+      +------------------------+
       |
       |
       v
+-------------------------+
| GitHub Actions Command|
|        Injection       |
+-------------------------+
       |
       |
       v
+-------------------------+
|   Other Attack Methods |
|   (e.g., Dependency    |
|   Confusion, Public-PPE,|
|   Access Token          |
|   Compromise)           |
+-------------------------+
       |
       |
       v
+-------------------------+
|      Create Worm       |
|      (Malware)          |
+-------------------------+
       |
       |
       v
+-------------------------+
|    Worm Propagation    |
|     and Infection      |
+-------------------------+
```

1. Repojacking

Attack Description: Attackers exploit GitHub's automatic redirection by registering a repository with a previously used name, redirecting consumers to malicious code. Repositories hosting actions are particularly vulnerable due to the bypassing of clone counts.

Imagine a scenario where an attacker attempts to repojack an open-source project named `example-repo` that hosts critical GitHub Actions workflows. We'll demonstrate mitigation steps using a sample GitHub repository and associated commands.

Step 1: Monitor Repository Name Changes

Monitor repository name changes by regularly checking for modifications using GitHub API or automation scripts.

Example Command:

```
# GitHub API request to get repository details
curl -s -H "Authorization: Bearer YOUR_GITHUB_TOKEN" \
  https://api.github.com/repos/owner/example-repo
```

Response (JSON):

```
{
  "id": 123456789,
  "name": "example-repo",
  "full_name": "owner/example-repo",
  "owner": {
    "login": "owner",
    "id": 987654321,
    "type": "Organization"
  },
  "private": false,
  "clone_url": "https://github.com/owner/example-repo.git",
  "created_at": "2023-01-01T12:00:00Z",
  "updated_at": "2023-01-10T15:30:00Z",
  "pushed_at": "2023-01-12T10:45:00Z"
  # Additional repository details...
}
```

By monitoring the `updated_at` and `pushed_at` fields, you can detect recent changes, indicating a possible repository name change.

Step 2: Enforce Additional Security Measures

To enforce additional security measures, implement GitHub Actions security checks and block certain actions if suspicious activity is detected.

Example GitHub Actions Workflow: **`.github/workflows/security_checks.yml`**

```
name: Security Checks

on:
  push:
    branches:
      - main

jobs:
  security_checks:
    runs-on: ubuntu-latest

    steps:
    - name: Check Repository Name Changes
      run: |
        if [ "$(git diff --name-only ${{ github.event.before }} ${{ github.sha }})" != "" ]; then
          echo "Repository name changed. Triggering security checks..."
          # Add security checks and notifications here
        else
          echo "No repository name change detected."
        fi
```

In this example, the workflow checks for changes between commits to identify repository name modifications and triggers security checks accordingly.

These measures help detect and respond to potential repojacking attempts, enhancing the security of repositories hosting critical GitHub Actions workflows.

Mitigation Steps:

1. Monitor repository name changes.
2. Enforce additional security measures for repositories hosting actions.

2. NPM Package Maintainer Email Hijacking

Attack Description: JavaScript-based actions are susceptible to attacks on NPM package maintainers' email accounts. Lack of 2FA facilitates password resets, allowing the creation of malicious package versions.

Consider a scenario where an attacker attempts to hijack the NPM package maintainer's email associated with a JavaScript-based GitHub Actions workflow. We'll demonstrate mitigation steps using a sample NPM package and associated commands.

Step 1: Enable 2FA for NPM Accounts

Enable two-factor authentication (2FA) for NPM accounts to add an extra layer of security.

Example Command (assuming NPM CLI is installed):

```
# Enable 2FA for the NPM account
npm profile enable-2fa
```

Follow the prompts to set up two-factor authentication for the NPM account.

Step 2: Regularly Audit and Rotate Credentials

Regularly audit and rotate credentials associated with the NPM account to minimize the risk of unauthorized access.

Example Command:

```
# List all NPM credentials
npm token list
```

Example Output:

```
┌──────────────────┬────────────┬─────────────┐
│ token            │ created    │ read-only   │
├──────────────────┼────────────┼─────────────┤
│ abcdefghijklmnop │ 2023-01-01 │ false       │
│ qrstuvwxyzabcdef │ 2023-02-01 │ true        │
└──────────────────┴────────────┴─────────────┘
```

Identify and review the existing tokens. If any are outdated or unnecessary, revoke them and create new ones.

Example Command to Revoke a Token:

```
# Revoke an NPM token
npm token revoke abcdefghijklmnop
```

By regularly auditing and rotating NPM credentials, you reduce the window of opportunity for attackers to exploit outdated or compromised credentials.

These measures help secure JavaScript-based GitHub Actions workflows by ensuring that NPM package maintainers' accounts are protected with 2FA and that credentials are regularly audited and rotated.

Mitigation Steps:

1. Enable 2FA for NPM accounts.
2. Regularly audit and rotate credentials.

3. GitHub Actions Command Injection

Attack Description: Workflows relying on untrusted input in bash commands are vulnerable to command injection. Fields like issue titles and commit messages should be treated as untrusted input.

In this scenario, we'll consider a GitHub Actions workflow that uses user-provided input in a bash command. We'll demonstrate mitigation steps to prevent command injection using input validation.

Step 1: Sanitize Inputs

Sanitize inputs to ensure they do not contain malicious commands. Use input validation before using them in bash commands.

Example GitHub Actions Workflow: `.github/workflows/sanitize_inputs.yml`

```
name: Sanitize Inputs

on:
  pull_request:
    types:
      - opened
      - synchronize
      - reopened

jobs:
  sanitize_inputs:
    runs-on: ubuntu-latest

    steps:
    - name: Check for Untrusted Input
      run: |
        # Extract the title of the pull request
        PR_TITLE=$(jq -r '.pull_request.title' $GITHUB_EVENT_PATH)

        # Validate the title to ensure it doesn't contain malicious commands
        if [[ ! "$PR_TITLE" =~ ^[a-zA-Z0-9_]+$ ]]; then
          echo "Error: Pull request title contains invalid characters."
          exit 1
        fi

        # Use the sanitized input in the workflow
        echo "Sanitized PR Title: $PR_TITLE"
    # Additional workflow steps...
```

In this example, the workflow checks the title of a pull request using `jq` and validates it against a regular expression to ensure it contains only alphanumeric characters and underscores. If the title contains invalid characters, the workflow exits with an error.

Step 2: Validate Data Sources

Validate data sources and treat them as untrusted input, especially when they are used in bash commands.

Example GitHub Actions Workflow: `.github/workflows/validate_data_sources.yml`

```
name: Validate Data Sources

on:
  push:
    branches:
      - main

jobs:
  validate_data_sources:
    runs-on: ubuntu-latest

    steps:
    - name: Check for Untrusted Commit Message
      run: |
        # Extract the commit message of the latest commit
        COMMIT_MESSAGE=$(git log -1 --pretty=format:"%s")

        # Validate the commit message to ensure it doesn't contain malicious commands
        if [[ ! "$COMMIT_MESSAGE" =~ ^[a-zA-Z0-9_]+$ ]]; then
          echo "Error: Commit message contains invalid characters."
          exit 1
        fi

        # Use the sanitized input in the workflow
        echo "Sanitized Commit Message: $COMMIT_MESSAGE"
    # Additional workflow steps...
```

This workflow checks the commit message of the latest commit and validates it against a regular expression to ensure it contains only alphanumeric characters and underscores.

By implementing these steps, you can significantly reduce the risk of command injection in GitHub Actions workflows that rely on user-provided input.

Mitigation Steps:

1. Sanitize inputs.
2. Validate data sources to prevent command injection.

Other Attack Methods

- **Dependency Confusion**
- **Public-PPE**
- **Compromising Maintainer’s Access Token**
- **Hidden Malicious Code in Pull Requests**

Actions Dependency Tree

Understanding the GitHub Actions dependency tree is crucial for comprehending the attack's propagation.

Types of Actions

- JavaScript
- Docker
- Composite (using action.yml file)

Dependencies and CI/CD Pipelines

Actions can depend on others through action.yml files or CI/CD pipelines, forming a tree of interconnected dependencies.

Mitigation Steps:

1. Regularly audit and monitor dependencies, especially actions used in workflows.

Compromised Actions Infecting Dependency Actions

Attackers exploit dependencies between actions to spread malware. GitHub Actions dependency tree analysis reveals interconnected relationships.

Dumping Secrets from a Runner's Memory

Attack Description: GitHub Actions runners expose secrets to jobs when they start, allowing attackers to dump secrets from the runner's memory even before they are used.

Imagine a scenario where an attacker attempts to exploit GitHub Actions runners to dump secrets from the runner's memory. We'll demonstrate mitigation steps using a GitHub Actions workflow that handles secrets with care.

Step 1: Regularly Rotate Secrets

Regularly rotating secrets helps minimize the risk of exposure even if an attacker gains access to the runner's memory.

Example GitHub Actions Workflow: `.github/workflows/rotate_secrets.yml`

```
name: Rotate Secrets

on:
  schedule:
    - cron: '0 0 * * *' # Run daily

jobs:
  rotate_secrets:
    runs-on: ubuntu-latest

    steps:
    - name: Rotate Secrets
      run: |
        # Rotate sensitive secrets
        rotate_secrets_script.sh
    # Additional workflow steps...
```

In this example, the workflow is scheduled to run daily and executes a script (`rotate_secrets_script.sh`) responsible for rotating sensitive secrets.

Step 2: Limit Access to Secrets

Limiting access to secrets ensures that only authorized users and workflows can access sensitive information.

Example GitHub Actions Workflow: `.github/workflows/limit_secret_access.yml`

```
name: Limit Secret Access

on:
  pull_request:
    types:
      - opened
      - synchronize
      - reopened

jobs:
  limit_secret_access:
    runs-on: ubuntu-latest

    steps:
```

```
    - name: Checkout Repository
      uses: actions/checkout@v2

    - name: Use Secret with Limited Access
      run: |
        # Use the secret in a secure manner
        use_secret_script.sh
    # Additional workflow steps...
```

This workflow restricts secret access to pull requests, ensuring that only specific events trigger the use of sensitive information.

Step 3: Monitor for Unauthorized Access

Implement monitoring mechanisms to detect and respond to unauthorized access to secrets.

Example GitHub Actions Workflow: `.github/workflows/monitor_secret_access.yml`

```
name: Monitor Secret Access

on:
  workflow_run:
    workflows:
      - 'Limit Secret Access'
    types:
      - completed

jobs:
  monitor_secret_access:
    runs-on: ubuntu-latest

    steps:
    - name: Monitor Secret Access
      run: |
        # Implement monitoring logic to detect unauthorized access
        monitor_secret_access_script.sh
    # Additional workflow steps...
```

This workflow runs after the "Limit Secret Access" workflow is completed and includes a script (`monitor_secret_access_script.sh`) to monitor and log secret access.

Mitigation Steps:

1. Regularly rotate secrets.
2. Limit access to secrets.
3. Monitor for unauthorized access.

Overriding Action’s Code

Attack Description: Attackers infect an action's repository by leveraging compromised secrets to push code changes, infecting dependent repositories through branches or tags.

In this scenario, we'll address the risk of attackers infecting an action's repository and spreading malware. We'll implement mitigation steps to secure GitHub Actions workflows against this type of attack.

Step 1: Restrict Permissions of GITHUB_TOKEN

By restricting the permissions of the GITHUB_TOKEN, you limit the actions it can perform and reduce the risk of unauthorized code changes.

Example GitHub Actions Workflow: `.github/workflows/restrict_token_permissions.yml`

```
name: Restrict Token Permissions

on:
  push:
    branches:
      - main

jobs:
  restrict_token_permissions:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Repository
      uses: actions/checkout@v2

    - name: Use GITHUB_TOKEN with Restricted Permissions
      run: |
        # Use the GITHUB_TOKEN with restricted permissions
        use_token_script.sh
    # Additional workflow steps...
```

This workflow ensures that the GITHUB_TOKEN is used with restricted permissions, limiting its impact on the repository.

Step 2: Configure Branch and Tag Protections

Configuring branch and tag protections adds an additional layer of security to prevent unauthorized changes.

Example GitHub Repository Settings:

1. Go to your GitHub repository.
2. Navigate to "Settings" > "Branches."
3. Configure branch protection rules for critical branches.

By configuring branch protection, you make it harder for attackers to override an action's code through unauthorized changes.

Step 3: Regularly Audit Workflow Files

Regularly auditing workflow files helps identify and address potential security vulnerabilities.

Example GitHub Actions Workflow: `.github/workflows/audit_workflow_files.yml`

```
name: Audit Workflow Files

on:
  pull_request:
    types:
      - opened
      - synchronize
      - reopened

jobs:
  audit_workflow_files:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Repository
      uses: actions/checkout@v2

    - name: Audit Workflow Files
      run: |
        # Implement a script to audit workflow files for security vulnerabilities
        audit_workflow_files_script.sh
    # Additional workflow steps...
```

This workflow triggers on pull requests and includes a script (`audit_workflow_files_script.sh`) to audit workflow files for security vulnerabilities.

Mitigation Steps:

1. Restrict permissions of GITHUB_TOKEN.
2. Configure branch and tag protections.
3. Regularly audit workflow files.

GitHub Actions Worm - A Real-world Demo

A detailed walkthrough of a GitHub Actions worm demonstrates how attackers exploit propagation mechanisms to compromise target repositories, emphasizing the need for robust security controls.

Impact Assessment

- **Attack Graphs at Scale**
- **Public Disclosure and Remediation**
- **Potential Impact on Private Repositories**

Mitigation Steps:

Implementing Strict Pipeline-Based Access Controls (PBAC)

Scenario:

Implementing strict Pipeline-Based Access Controls (PBAC) is crucial for ensuring that GitHub Actions workflows are granted the least privileges necessary. In this scenario, we'll set up a GitHub Actions workflow that enforces strict PBAC.

Example GitHub Actions Workflow: `.github/workflows/enforce_pbac.yml`

```
name: Enforce PBAC

on:
  push:
    branches:
      - main

jobs:
  enforce_pbac:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Repository
      uses: actions/checkout@v2

    - name: Enforce PBAC
      run: |
        # Implement strict PBAC logic
        enforce_pbac_script.sh
    # Additional workflow steps...
```

In this workflow, the `enforce_pbac_script.sh` script enforces strict PBAC logic, ensuring that the workflow is granted the least privileges and access it needs to fulfill its purpose.

Configuring Branch and Tag Protections

Scenario:

Configuring branch and tag protections adds an additional layer of security to GitHub repositories. Let's configure branch and tag protections for the `main` branch.

Example GitHub Repository Settings:

1. Go to your GitHub repository.
2. Navigate to "Settings" > "Branches."
3. Under "Branch protection rules," click on "Add rule."
4. Configure protection rules for the `main` branch:
    - Require pull request reviews before merging.
    - Include administrators.
    - Enforce status checks to pass before merging.
    - Disallow force pushes.

By configuring these rules, you ensure that changes to the `main` branch go through a review process, status checks, and are protected against force pushes.

Monitoring Network Connections

Scenario:

Monitoring and limiting outbound network connections from workflow runners help prevent the download of malicious code into pipelines and stop malware from reporting to command and control (C2) servers.

Example GitHub Actions Workflow: `.github/workflows/monitor_network_connections.yml`

```
name: Monitor Network Connections

on:
  pull_request:
    types:
      - opened
      - synchronize
      - reopened

jobs:
  monitor_network_connections:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Repository
      uses: actions/checkout@v2

    - name: Monitor Network Connections
      run: |
        # Implement network monitoring logic
        monitor_network_connections_script.sh
    # Additional workflow steps...
```

In this workflow, the `monitor_network_connections_script.sh` script monitors network connections to detect and prevent unauthorized outbound connections.

Pinning Actions Using Commit Hashes

Scenario:

Pinning actions using commit hashes helps reduce the risk of using a maliciously modified action. Let's modify a GitHub Actions workflow to use a specific commit hash for an action.

Example GitHub Actions Workflow: `.github/workflows/pin_action.yml`

```
name: Pin Action

on:
  push:
    branches:
      - main

jobs:
  pin_action:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Repository
      uses: actions/checkout@v2

    - name: Use Action with Commit Hash
      uses: owner/action-repo@3f4a1b2
    # Additional workflow steps...
```

In this workflow, the `uses` field specifies the action with a specific commit hash (`3f4a1b2`). This ensures that the workflow always uses the intended version of the action.

By implementing these mitigation steps, you enhance the security of GitHub Actions workflows, enforcing strict PBAC, configuring branch and tag protections, monitoring network connections, and pinning actions using commit hashes.

Know Your Pipeline Dependencies

The concept of dependencies applies not only to software applications but also to CI/CD pipelines. Managing pipeline dependencies is crucial for security.

Mitigation Steps:

1. Track and manage pipeline dependencies similarly to software components.
2. Implement strict access controls.
3. Regularly audit dependencies.

Protecting Your Workflows and Assets

Let's assume a scenario where a development team wants to enhance the security of their GitHub Actions workflows against potential worm attacks. We'll demonstrate mitigation steps using GitHub repository settings and workflow configurations.

Step 1: Set Minimal Permissions for GITHUB_TOKEN and PAT

Configure minimal permissions for the GITHUB_TOKEN and Personal Access Tokens (PAT) used in GitHub Actions workflows. Limiting permissions helps reduce the impact of potential worm attacks.

Example GitHub Repository Settings:

1. Go to your GitHub repository.
2. Navigate to "Settings" > "Secrets."
3. Edit the GITHUB_TOKEN secret and set minimal permissions.

Example Workflow Configuration: `.github/workflows/minimal_permissions.yml`

```
name: Minimal Permissions

on:
  push:
    branches:
      - main
```

```
jobs:
  minimal_permissions:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout Repository
      uses: actions/checkout@v2

    - name: Use GITHUB_TOKEN with Minimal Permissions
      run: |
        # Perform actions that require minimal permissions
    # Additional workflow steps...
```

By configuring minimal permissions for the GITHUB_TOKEN and PAT, you ensure that workflows only have the necessary access to fulfill their purpose.

Step 2: Configure Branch and Tag Protections

Configure branch and tag protections to prevent unauthorized changes and ensure that only authorized personnel can modify critical branches and tags.

Example GitHub Repository Settings:

1. Go to your GitHub repository.
2. Navigate to "Settings" > "Branches."
3. Configure branch protection rules for critical branches.

Example Workflow Configuration: `.github/workflows/branch_protection.yml`

```
name: Branch Protection

on:
  push:
    branches:
      - main

jobs:
  branch_protection:
    runs-on: ubuntu-latest

    steps:
    - name: Ensure Branch Protection
      run: |
        # Check if branch protection rules are configured for main branch
        # Add additional checks and notifications as needed
    # Additional workflow steps...
```

By configuring branch protection, you add an extra layer of security to prevent unauthorized changes to critical branches.

Step 3: Monitor and Limit Outbound Network Connections

Monitor and limit outbound network connections from GitHub Actions runners to prevent the download of malicious code and block malware from reporting to command and control (C2) servers.

Example GitHub Actions Workflow: `.github/workflows/monitor_network_connections.yml`

```
name: Monitor Network Connections

on:
  push:
    branches:
      - main
```

```
jobs:
  monitor_network_connections:
    runs-on: ubuntu-latest

    steps:
    - name: Limit Outbound Network Connections
      run: |
        # Add firewall rules or network restrictions to limit outbound connections
    # Additional workflow steps...
```

By limiting outbound network connections, you reduce the risk of GitHub Actions runners downloading malicious payloads or communicating with external servers.

Step 4: Pin Actions Using Commit Hashes

Pin actions using commit hashes to ensure that only verified and trusted versions of actions are used in workflows.

Example GitHub Actions Workflow: `.github/workflows/pin_actions.yml`

```
name: Pin Actions

on:
  push:
    branches:
      - main

jobs:
  pin_actions:
    runs-on: ubuntu-latest

    steps:
    - name: Use Pinned Version of an Action
      uses: owner/example-action@f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a
    # Additional workflow steps...
```

By pinning actions using commit hashes, you ensure that workflows consistently use a specific version of an action, reducing the risk of using maliciously modified versions.

These mitigation steps collectively enhance the security of GitHub Actions workflows and assets against potential worm attacks.

Mitigation Steps:

1. Set minimal permissions for GITHUB_TOKEN and PAT.
2. Configure branch and tag protections.
3. Monitor and limit outbound network connections.
4. Pin actions using commit hashes.

Implement a combination of these controls based on their effectiveness and the specific requirements of your workflows to enhance the overall security posture of GitHub Actions in your repositories.

In conclusion, understanding and securing the GitHub Actions workflow dependencies are paramount to mitigating the risks posed by the evolving threat landscape. DevSecOps practices, coupled with continuous monitoring and mitigation strategies, are essential for safeguarding repositories against the GitHub Actions Worm.

Reference

- https://www.paloaltonetworks.com/blog/prisma-cloud/github-actions-worm-dependencies/
- https://devsecopsguides.com/