令和7年度
「専門職業人材の最新技能アップデートのための専修学校リカレント教育（リ・スキリング）推進事業」

コンテナサーバーセキュリティ教材資料
問題編

本コンテナサーバーセキュリティ教材資料問題編は、文部科学省の教育政策推進事業委託費による委託事業として、一般社団法人全国専門学校情報教育協会が実施した令和７年度「専門職業人材の最新技能アップデートのための専修学校リカレント教育（リ・スキリング）推進事業」の成果物です。

情報技術者の技能アップデートのためのリカレント教育推進事業

目次

コンテナ・クラウドサービスのセキュリティ　概要編…1

演習1 Kubernetes環境の診断と防御　問題編…6

1-0:脆弱なK8s環境構築　問題編…11

1-1: ServiceAccount権限の最小化　問題編…14

1-2:Pod間通信の最小許可　問題編…16

1-3:PSS適用によるrootユーザ拒否　問題編…19

演習1 Kubernetes環境の診断と防御　合格判定基準…22

演習2 安全なコンテナ構築とランタイム防御　問題編…23

2-1:Trivyによる脆弱性診断　問題編…29

2-2:Cosignによる署名と検証　問題編…33

2-3:Cosign+Kyvernoによる検証　問題編…36

2-4:ランタイム保護による操作制御　問題編…40

演習2 安全なコンテナ構築とランタイム防御　合格判定基準…42

演習3 パイプラインへのセキュリティゲート導入　問題・解答編…43

3-1:事前準備　問題・解答編…50

3-2:Ruff/Black/CodeQLによるコードの静的解析 問題・解答編…52

3-3:事前準備② 問題・解答編…59

3-4:Build/Scan/PushのCI統合問題・解答編…61

3-5:Dependabotによるパッケージ脆弱性検出 問題・解答編…65

演習3 パイプラインへのセキュリティゲート導入 まとめ…68

演習3 パイプラインへのセキュリティゲート導入 合格判定基準…70

演習4 サービス間通信の暗号化とシークレット管理 問題編…71

4-1:シークレット情報のSecret管理 問題編…77

4-2:Istioインストール 問題編…79

4-3:mTLS通信による認証 問題編…83

4-4:AuthorizationPolicyによる認可 問題編…85

演習4 サービス間通信の暗号化とシークレット管理 合格判定基準…88

演習5 Kialiによる可視化と通信分析 問題編…89

5-1:Prometheus / Grafana / Kialiのインストール 問題編…95

5-2:Kialiトラフィックグラフによる可視化 問題編…98

5-3:Kialiを用いた通信分析 問題編…100

5-4:Grafanaダッシュボード作成 問題編…103

演習5 Kialiによる可視化と通信分析 合格判定基準…105

確認テスト…107

コンテナ・クラウドサービスのセキュリティ

概要編

コンテナ・クラウドサービスのセキュリティ　概要編

クラウドネイティブセキュリティの4C

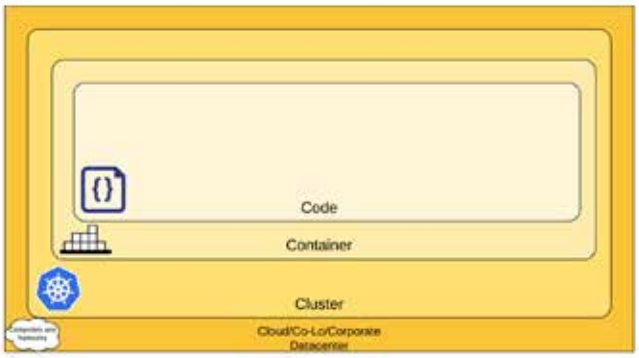
4C（Four Cs of Cloud Native Security）はクラウドネイティブ環境の防御構造を4層で捉える多層防御モデルです。Cloud → Cluster → Container → Codeの4層で構成され、外側（Cloud）から内側（Code）まで**全ての層でセキュリティを確保する必要がある**と定義しています。

Cloud（クラウド）
インフラ基盤（ネットワーク・IAM・暗号化）を安全に構成し、外部からの侵入を防ぎます。

Cluster（クラスター）
Kubernetes制御面を保護し、アクセス権限や通信経路を最小化します。

Container（コンテナ）
安全なコンテナイメージと実行環境を維持し、特権利用や脆弱な設定を防ぎます。

Code（コード）
安全なアプリ開発と依存管理を行い、脆弱性や秘密情報の混入を防ぎます。



https://kubernetes.io/ja/docs/concepts/security/overview/

クラウドネイティブセキュリティの基本原則

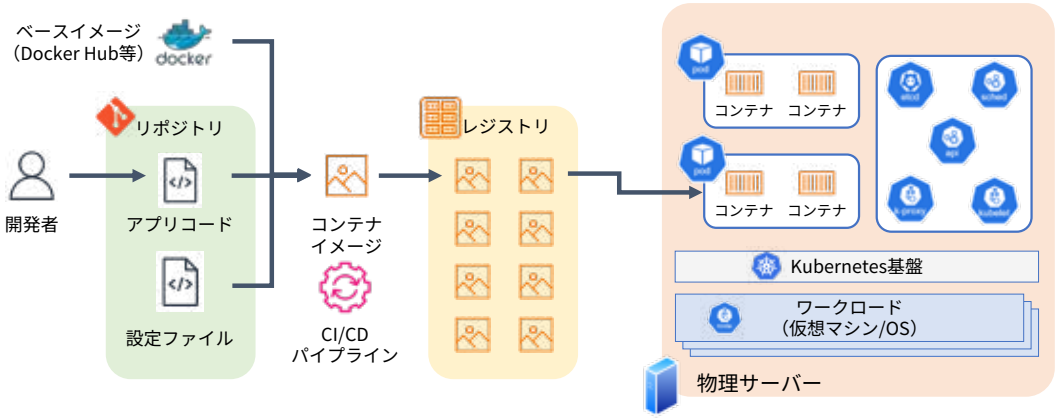
クラウドネイティブにおけるセキュリティの基本原則を抑えておきましょう。

最小権限の原則	ユーザー、ServiceAccount、コンテナプロセスなどに必要最小限の権限を付与し、侵害時の被害範囲を最小化します。
コンテナイメージの信頼性確保	ビルド時にスキャン・署名を行い、信頼されたレジストリのみを使用する。
多層防御	クラウド、オーケストラ、コンテナ、アプリの各層で防御を組み合わせることで、1つの防御層を突破しても全体に影響が出にくい設計にする。
ランタイム防御と分離	seccomp/AppArmorなどの機能で実行環境を制限し、コンテナ間の影響を遮断する。
継続的なセキュリティテスト	CI/CDに脆弱性スキャンや静的解析を組み込み、早期検知・改善のサイクルを回す。
監査とログ管理	全てのログを収集・分析し、不正や異常を即座に検知する。
自動化と標準化	IaCでセキュリティ設定をコード化し、一貫した環境を保つ。

コンテナ・クラウドサービスのセキュリティ　概要編

コンテナのライフサイクル

コンテナセキュリティを捉える前に、コンテナのライフサイクルを確認しましょう。
これらを構成する要素から、コンテナセキュリティの全体像を把握することができます。



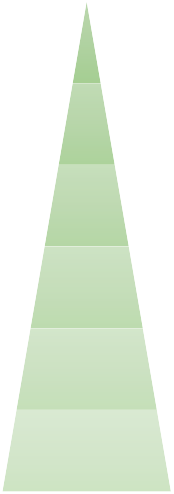
構成要素

①コード　②イメージ　③レジストリ　④オーケストラ　⑤コンテナ　⑥ホスト

コンテナ・クラウドサービスのセキュリティ　概要編

コンテナセキュリティの全体像

コンテナを構成する各要素に分け、セキュリティ対策を実施する必要があります。
各要素を連携させることで、コンテナ環境の多層防御が実現します。



①コード
安全なアプリケーションを開発し、脆弱性を含めない

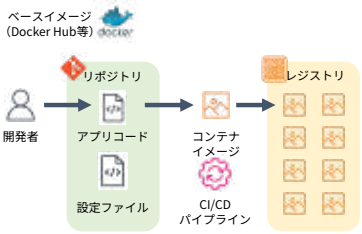
②イメージ
セキュアなコンテナイメージをビルド・検証する

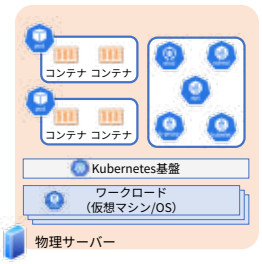
③レジストリ
信頼されたコンテナイメージのみを配布する

④オーケストラ（K8s基盤）
コンテナ実行環境の制御面を守る

⑤コンテナ（Pod/コンテナ）
実行中のコンテナを安全に維持する

⑥ホスト（物理/仮想サーバー）
OS・カーネル・ランタイムを防御し、基盤侵害を防ぐ





コンテナ・クラウドサービスのセキュリティ　概要編

各要素のコンテナセキュリティ対策

各要素の主なセキュリティ対策を抑えておくことで、運用するシステムの強化ポイントが見えてきます。

構成要素	目的・役割	主なセキュリティ対策
コード	安全なアプリケーションを開発し、脆弱性を持ち込まない	SAST/DAST、依存ライブラリの脆弱性スキャン、Secretsの埋め込み防止、入力値検証
イメージ	セキュアなコンテナイメージをビルド・検証する	最小ベースイメージ採用、不要パッケージ削除、イメージ署名（Cosign等）、脆弱性スキャン（Trivy等）
レジストリ	信頼されたイメージのみを配布する	アクセス制御（IAM/OIDC）、署名検証、スキャン自動化、レジストリ運用（ECR/GAR）
オーケストラ	コンテナ実行環境の制御面を守る	RBAC、NetworkPolicy、PodSecurityStandards（PSS）、Admission Controller（Kyverno）、Service Mesh
コンテナ	実行中のコンテナを安全に維持する	非root実行、readOnlyRootFS、seccomp/AppArmor、Runtime検知（Falco）、Observability
ホスト	OS・カーネル・ランタイムを防御し、基盤侵害を防ぐ	OSパッチ適用、最小権限ユーザー、auditd/syslog監査、ホスト分離（VM, sandbox）

コンテナ・クラウドサービスのセキュリティ　概要編

演習の進め方

kubernetes環境を利用してコンテナセキュリティについて理解を深めましょう。
コンテナの構成要素別に、5つの演習を用意しています。

演習１ **Kubernetes環境の診断と防御**	RBAC、NetworkPolicy、PSSの設定を通じて、Kubernetes環境における基本的なアクセス制御を体得します。 **関連要素：オーケストラ**
演習２ **安全なコンテナ構築とランタイム防御**	脆弱なDockerfileを改善し、イメージ署名やseccompなどの適用により、イメージ・実行時の安全性を高めます。 **関連要素：コンテナ、イメージ、ホスト**
演習３ **パイプラインへのセキュリティゲート導入**	GitHub Actionsを使ってコード解析・脆弱性スキャンを自動化することで、Shift-Leftを実現し、DevSecOpsの基本を学習します。 **関連要素：コード、イメージ、リポジトリ**
演習４ **サービス間通信の暗号化とシークレット管理**	Istioを導入し、相互TLS（mTLS）でサービス間通信を暗号化し、AuthorizationPolicyによるアクセス制御を行います。 **関連要素：オーケストラ、コンテナ**
演習５ **Kialiによる可視化と通信分析**	Kiali / Prometheus / Grafanaを導入し、サービス間通信を可視化することで、エラー率などの分析が出来るようにします。 **関連要素：オーケストラ、ホスト**

コンテナ・クラウドサービスのセキュリティ　概要編

演習環境

EKS等のマネージドなKubernetesサービス、もしくは、Docker Desktop / kindなどのローカルKubernetes環境を利用してください。

注意事項

- 個人/検証用クラスタで実施してください。本番・共有クラスタでの適用は控えてください。
- 脆弱な設定は演習完了後に即時削除してください。
- リソースは演習で用意するNamespace内に限定して作成しましょう。
- 演習によってはホストの負荷が高くなります。Kubernetes環境のリソースに余裕を持たせてください。
- 各演習の解説はEKS前提で進めます。ローカルPCもしくは別環境の場合は読み替えてください。

推奨

マネージドKubernetes (EKS/AKS/GKE)

本番に近い環境、マルチノード構成
NetworkPolicy対応CNI必須（Calico等）

Docker Desktop + Kubernetes

ローカル開発用、マルチノード構成
Kubernetes有効化は Settings > Kubernetes > Enable Kubernetes

演習環境　推奨構成

演習環境の推奨構成は以下の通りです。
演習を進めるうえで参考にしてください。

項目	AWS構成（推奨）	ローカル構成
実行環境	AWS CloudShell CloudShell＋EKS	ローカルPC Docker Desktop + kubectl + kind or minikube
クラスタ	Amazon EKSノード t3.medium × 3（推奨）	kind/minikubeクラスタローカルVM上で構築
スペック	t3.medium ×3 （vCPU 2／4GB×3台） EBS（gp3, 各ノード20〜30GB）	CPU：4コア以上、MEM：8GB以上 ストレージ：50G以上
補助ツール	aws cli, eksctl, kubectl, helm, trivy, cosign	kubectl, helm, docker, trivy, cosign
デプロイ方式	YAML適用／helm	YAML適用／helm
成果物保存	GitHubリポジトリ	ローカルディレクトリ or GitHubリポジトリ
コンテナイメージ管理	ECR（Elastic Container Registry）	ローカルDocker／Docker Hub
可視化・監視・分析	Prometheus + Kiali + Grafana（EKS上）	Prometheus + Kiali + Grafana（EKS上）

コンテナ・クラウドサービスのセキュリティ　概要編

参考）EKSのクラスター構築および削除

AWS EKSのクラスターの構築および削除はCLIもしくはAWS コンソールから行うことが可能です。
EKSクラスターの構築および削除コマンド（yamlファイルベース）を以下に記載します。

EKSクラスターの構築および削除手順

事前にeksctlをインストールしてから実施してください。

1. **EKSクラスターの構築**
   $ eksctl create cluster -f eks-cluster.yaml

2. **EKSクラスターの削除**
   $ eksctl delete cluster -f eks-cluster.yaml

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: cs-cluster
  region: ap-northeast-1
  version: "1.29"

managedNodeGroups:
  - name: lab-nodes
    instanceType: t3.medium
    desiredCapacity: 3
    volumeSize: 20
```

eks-cluster.yaml

参考）eksctlインストール

eksctl　のインストール手順を記載します。
必要に応じてインストールしてください。

eksctlインストール

1. 最新版の eksctl をダウンロード
```
curl --silent --location¥
  "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" ¥
  | tar xz -C /tmp
```

2. 実行パスへ移動（CloudShellユーザー領域）
```
sudo mv /tmp/eksctl /usr/local/bin
```

3. 実行権限を付与
```
sudo chmod +x /usr/local/bin/eksctl
```

4. バージョン確認
```
eksctl version
```

演習１　Kubernetes環境の診断と防御

問題編

演習１　Kubernetes環境の診断と防御

演習概要

脆弱なKubernetes環境に対して、Kubernetesの基本的な3つのセキュリティ対策を実装します。
対策前後での挙動の違いを通して、コンテナ環境のセキュリティ強化について理解を深めます。

Before：脆弱なK8s環境

過剰な権限
ServiceAccountに不要な権限が付与されている場合、攻撃者に悪用される可能性あり

無制限の通信
全Pod間通信が許可されている場合、横展開で他Podが攻撃されるリスクあり

特権Pod
ホストシステムへのアクセスが可能で、エスカレーションの危険性あり

After：安全な構成

最小権限
RBACで必要最小限の権限のみを付与し、攻撃範囲を限定

通信制御
NetworkPolicyで必要な通信のみを許可し、攻撃の拡散を防止

権限制限
PSSで特権実行を禁止し、権限昇格攻撃からシステムを保護

演習１　Kubernetes環境の診断と防御

演習内容

脆弱なKubernetes環境に対して、Kubernetesの基本的な3つのセキュリティ対策を実装します。
対策前後での挙動の違いを通して、コンテナ環境のセキュリティ強化について理解を深めます。

演習内容

1-1. RBAC： RBACで必要最小限の権限のみを付与し、攻撃範囲を限定

1-2. NetworkPolicy：NetworkPolicyで必要な通信のみを許可し、攻撃の拡散を防止

1-3. Pod Security Standards（PSS）：PSSで特権実行を禁止し、権限昇格攻撃からシステムを保護

取り扱うセキュリティ対策

RBAC
アクセス制御

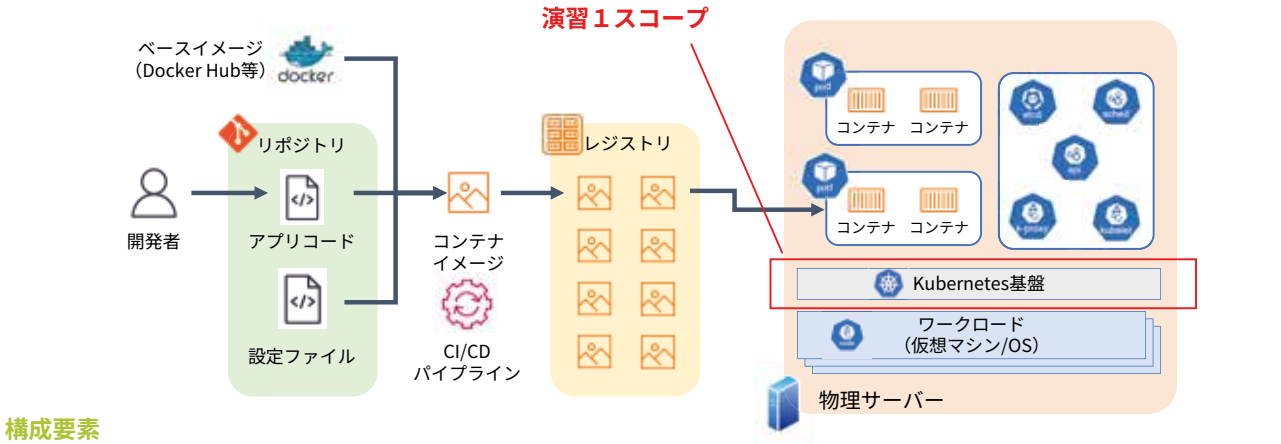
NetworkPolicy
通信制御

PSS
特権Pod制限

演習１　Kubernetes環境の診断と防御

演習１の位置づけ

演習１ではコンテナオーケストレーションをターゲットにセキュリティ対策を行います。



構成要素

①コード　②イメージ　③レジストリ　④オーケストラ　⑤コンテナ　⑥ホスト

演習１　Kubernetes環境の診断と防御

コンテナオーケストレーションにおけるセキュリティの問題

コンテナオーケストレーション環境では、1つの脆弱なPodが侵害されると、
クラスタ全体に被害が拡大するリスクがあります。

⚠ 想定されるセキュリティリスク

SAトークンでのAPIアクセス

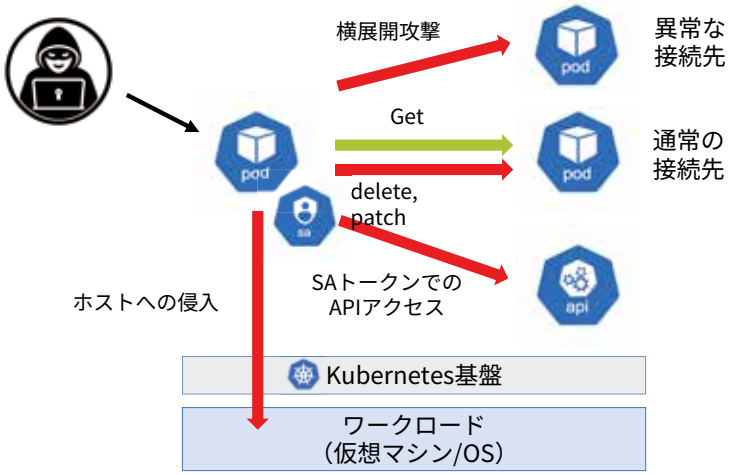
PodのServiceAccountトークンを利用して、
Kubernetes API経由で他Podを操作される

横展開攻撃

他Podへの攻撃、想定外な操作が可能となり、
環境が汚染され、データが盗まれるリスクがある

ホストへの侵入

rootユーザでホストにアクセスされ、
環境汚染やサーバの破壊などが起こりうる



演習１　Kubernetes環境の診断と防御

コンテナオーケストレーションにおけるセキュリティ対策

コンテナオーケストレーションのセキュリティリスクに対して、
RBAC / NetworkPolicy / PSSなどを実施することで防御することが可能です。

想定されるセキュリティリスク		実施するセキュリティ対策
SAトークンでのAPIアクセス PodのServiceAccountトークンを利用して、Kubernetes API経由で他Podを操作される	→	**RBAC**によるアクセス制御
横展開攻撃 他Podへの攻撃、想定外な操作が可能となり、環境が汚染され、データが盗まれるリスクがある	→	**NetworkPolicy**による通信制御
ホストへの侵入 rootユーザでホストにアクセスされ、環境汚染やサーバの破壊などが起こりうる	→	**PSS**による特権Pod制限

演習１　Kubernetes環境の診断と防御

前提知識：RBAC（Role-Based Access Control）

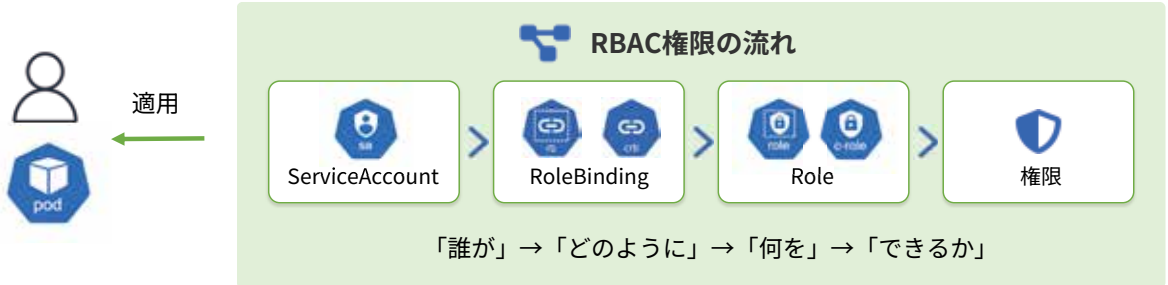
ユーザーやリソースにロール（役割）を割り当て、そのロールに紐づく権限を付与することで、システムやデータへのアクセスを管理する手法です。RBACを活用してアクセス制御を実現します。

RBACの基本概念

- ServiceAccount:　Podやユーザーに割り当てる実行アカウント
- RoleBinding/ClusterRoleBinding:　アカウントとロールの紐付け
- Role/ClusterRole:　権限の定義（namespace内/クラスタ全体）
- 最小権限の原則: 必要最小限の権限のみを付与する

RBACの適用例

- 参照Pod
  他Podに対してGETは許可
  それ以外は拒否としReadOnlyにする



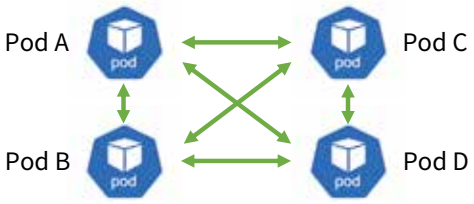
演習１ Kubernetes環境の診断と防御

前提知識：NetworkPolicy

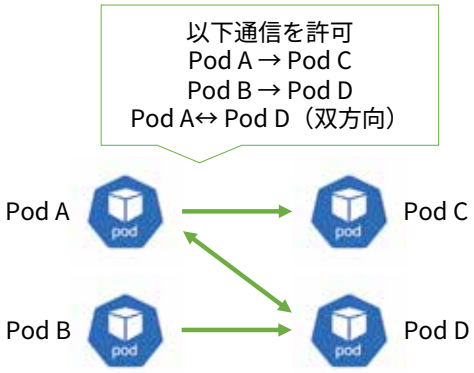
Pod間の通信を制御する Kubernetes の仕組みです。
ファイアウォールのような役割を持ち、明示的に許可された通信以外はブロックします。

NetworkPolicyの基本概念

- デフォルト： 全Pod間通信が許可（セキュリティリスク）
- NetworkPolicy： Pod間通信を制限するルール
- Ingress/Egress制御： 入出力の通信を個別に設定可能
- 柔軟な選択肢： Pod/Namespace/IPブロックで指定



適用前：すべてのPod間通信が許可（デフォルト）
セキュリティリスクが高い



適用後：必要な通信のみ許可

演習１ Kubernetes環境の診断と防御

前提知識：PSS（Pod Security Standards）

Podのセキュリティ設定を一定の基準で制御する仕組み。「危険なPod設定をクラスタに入れない」ためのチェック機構です。Namespace単位で適用可能です。

Privileged ⚠️	Baseline ✅	Restricted 🛡️
制限なし	一般的なワークロード向け	強固なセキュリティ
全機能にアクセス可能	既知の特権昇格を防止	非特権ユーザーのみ実行
セキュリティ上のリスクあり	利便性とセキュリティのバランス	ハードニングベストプラクティス
主な制約 制約なし - 特権昇格、ホストアクセスなど全て許可	**主な制約** 特権コンテナ禁止、ホストネームスペース不可、ホストパス不可、特定の機能のみ許可	**主な制約** Baselineの全制約＋特権昇格禁止、非rootユーザー必須、seccompプロファイル必須、権限厳格化

演習１　Kubernetes環境の診断と防御

演習１の進め方

脆弱なKubernetes環境に対して、Kubernetesの基本的な3つのセキュリティ対策を実装します。
対策前後でセキュリティが強化されることを確認しましょう。

1-0	**脆弱なK8s環境の構築**	広い権限、オープンな通信が可能なK8s環境を構築 K8s環境におけるセキュリティ上の問題点を確認
1-1	対策1 **ServiceAccount権限の最小化**	RBACを適用し、Podからの操作権限を最小化 他Podに対する操作を制限し、想定外な動作を排除
1-2	対策2 **Pod間通信の最小許可**	NetworkPolicyを適用し、Pod間通信を制限 許可されたPod通信のみ正常に機能
1-3	対策3 **PSS適用によるrootユーザ拒否**	PSSを適用し、Podの特権昇格を防止 ホストへのアクセスなどを制限

1-0：脆弱なK8s環境構築

問題編

演習１　Kubernetes環境の診断と防御

1-0：脆弱なK8s環境構築

脆弱なKubernetes環境を構築して、脆弱な環境であることを確認しましょう。
マニフェストを作成して、kubectlコマンドで各リソースを起動してください。

環境構築手順

※k8sクラスターは構築済の前提

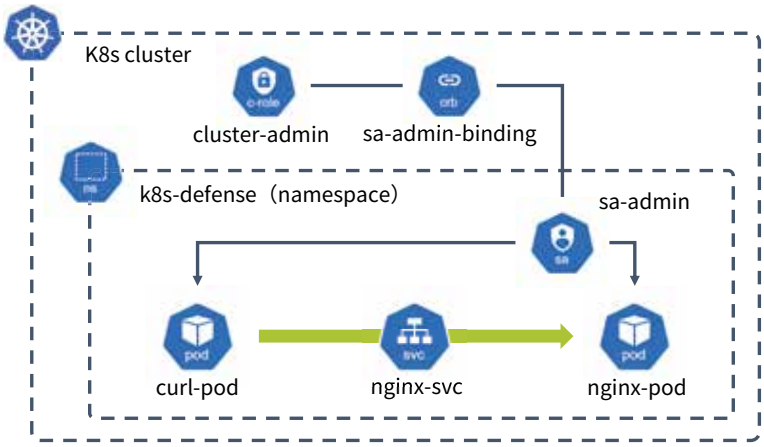
1. Namespace作成
   namespace : k8s-defense

1. ClusterRoleBinding/ClusterRole作成
   crb: sa-admin-binding
   c-role: cluster-admin

2. ServiceAccountの作成
   sa: sa-admin

1. curl Podの作成
   pod: curl-pod

2. nginx Pod/serviceの作成
   pod: nginx-pod
   service: nginx-svc



構築するk8s環境

演習１　Kubernetes環境の診断と防御

1-0：Namespace / ServiceAccount / ClusterRoleBinding のコード

Namespace、ServiceAccount、ClusterRoleBindingのマニフェストは以下の通りです。
sa-adminはクラスタ管理者権限を持っていて、適用したリソースに脆弱な権限を付与します。

```yaml
# Namespace
apiVersion: v1
kind: Namespace
metadata:
  name: k8s-defense
```

namespace-k8s-defense.yaml

```yaml
# ServiceAccount（過剰権限を持つ）
apiVersion: v1
kind: ServiceAccount

metadata:
  name: sa-admin
  namespace: k8s-defense
```

service-account-admin.yaml

```yaml
# ClusterRoleBinding（クラスタ管理者権限を付与）
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding

metadata:
  name: sa-admin-binding

subjects:
- kind: ServiceAccount
  name: sa-admin
  namespace: k8s-defense

roleRef:
  kind: ClusterRole
  # kubernetes標準で用意されているClusterRoleを指定
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

cluster-role-bindng-admin.yaml

演習１　Kubernetes環境の診断と防御

1-0：curl Podとnginx Pod/Serviceのコード

curl Pod、nginx Podおよびserviceのマニフェストは以下の通りです。
両Podには、脆弱なsa-adminを適用してください。

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: curl
 namespace: k8s-defense
 labels:
  app: curl

spec:
 # 脆弱なSAの適用
 serviceAccountName: sa-admin
 containers:
 - name: curl
  image: curlimages/curl
  command: ["sleep", "3600"]
```

curl-pod.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: nginx
 namespace: k8s-defense
 labels:
  app: nginx

spec:
 # 脆弱なSAの適用
 serviceAccountName: sa-admin
 containers:
 - name: nginx
  image: nginx:latest
  ports:
  - containerPort: 80
```

nginx-pod.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
 name: nginx
 namespace: k8s-defense
spec:
 selector:
  app: nginx
 ports:
  - protocol: TCP
   port: 80
   targetPort: 80
```

nginx-service.yaml

演習１　Kubernetes環境の診断と防御

1-0：動作確認

K8s環境の構築が完了したら、各リソースを起動して、動作確認しましょう。

動作確認手順

1. **curl Podからnginx Podへのアクセス確認**
   curl Podからcurlコマンドを実行し、nginx のHTMLが表示されること
   ```
   $ kubectl exec -n k8s-defense -it curl -- curl -m 3 http://nginx
   ```

2. **sa-adminの操作権限確認**
   ServiceAccountの操作権限を確認し、podからget / delete できるか、権限を確認
   ```
   $ kubectl auth can-i get pods --as=system:serviceaccount:k8s-defense:sa-admin -n k8s-defense
   $ kubectl auth can-i delete pods --as=system:serviceaccount:k8s-defense:sa-admin -n k8s-defense
   ```

3. **sa-adminの操作権限一覧確認**
   ServiceAccountの操作権限を一覧で確認
   ```
   $ kubectl auth can-i --list --as=system:serviceaccount:k8s-defense:sa-admin
   ```

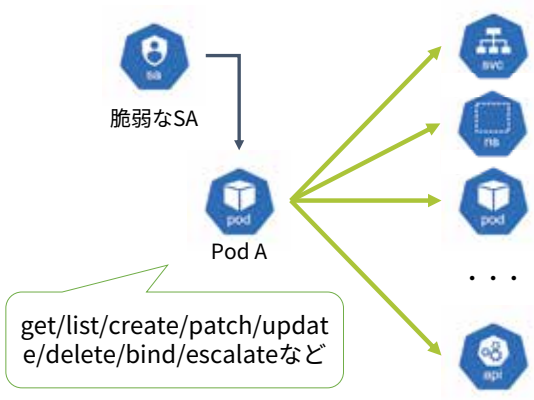
1-1： ServiceAccount権限の最小化

問題編

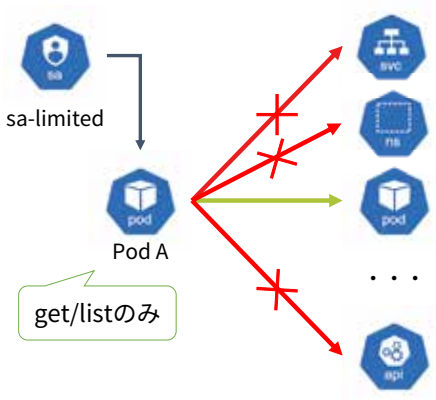
演習１ Kubernetes環境の診断と防御

1-1：期待される挙動

sa-adminは様々なリソースに様々な操作ができ、脆弱な状態でした。
sa-limitedに付け替え、Podへのget / listのみに絞ることがゴールです。



権限最大時（脆弱）



権限最小化時

演習１　Kubernetes環境の診断と防御

1-1：ServiceAccount権限の最小化

ServiceAccountの権限を最小化して、Pod操作が制限されていることを確認しましょう。
マニフェストを作成して、kubectlコマンドで各リソースを起動してください。

検証手順

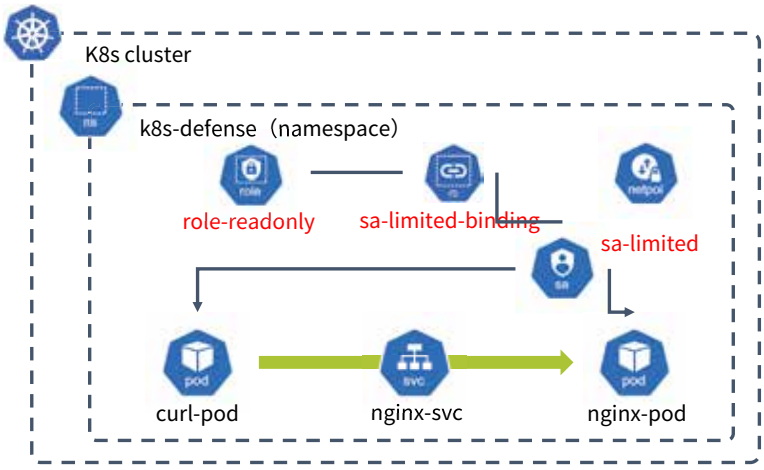
※k8sクラスターは構築済の前提

1. RoleBinding/Role作成
   rb: sa-limited-binding
   role: role-readonly

2. ServiceAccountの作成
   sa: sa-limited

1. curl Pod / nginx Podの修正
   SAをsa-adminからsa-limitedに変更
   Podの再起動が必要

2. ServiceAccountの権限確認
   sa-limitedの操作権限が
   制限されていることを確認



構築するk8s環境

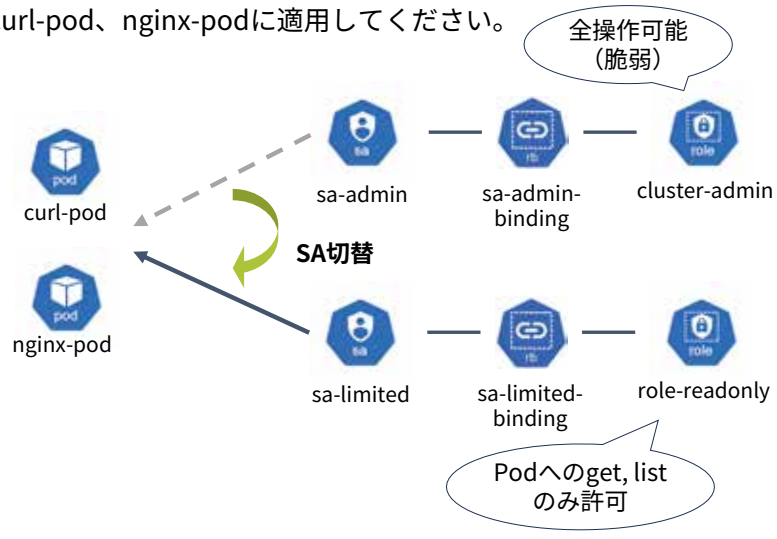
演習１　Kubernetes環境の診断と防御

1-1：Roleのコード

roleのマニフェストは以下の通りです。
role-readonlyは「pod」に対して「get」「list」のみ許可するロールです。
合わせて新規SA、RoleBindingも作成し、curl-pod、nginx-podに適用してください。

```
# Role（namespace内のPod閲覧のみ許可）
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role-readonly
  namespace: k8s-defense
rules:
- apiGroups: [""]
  resources: [“pods”]  # Podに対して
  verbs: [“get”, “list”]  # GET, LISTのみ許可
```

role-readonly.yaml



演習１　Kubernetes環境の診断と防御

1-1：動作確認

K8s環境の構築が完了したら、各リソースを起動して、動作確認しましょう。
sa-adminと比較して、sa-limitedの権限が制限されていることを確認してください。

動作確認手順

1. **curl PodからNginx Podへのアクセス確認**
   ※curl / nginx Podは事前に再作成し、両PodのSAにsa-limitedを適用してください。
   curl Podからcurlコマンドを実行し、nginx のHTMLが表示されること（変化なし）
   ```
   $ kubectl exec -n k8s-defense -it curl -- curl -m 3 http://nginx
   ```
2. **sa-limitedの操作権限確認**
   ServiceAccountの操作権限を確認し、podからget/listのみ出来るか、権限の確認
   ```
   $ kubectl auth can-i get pods --as=system:serviceaccount:k8s-defense:sa-limited -n k8s-defense
   $ kubectl auth can-i delete pods --as=system:serviceaccount:k8s-defense:sa-limited -n k8s-defense
   ```
3. **sa-limitedの操作権限一覧確認**
   ServiceAccountの操作権限を一覧で確認し、権限が制限されていることを確認
   ```
   $ kubectl auth can-i --list --as=system:serviceaccount:k8s-defense:sa-limited
   ```

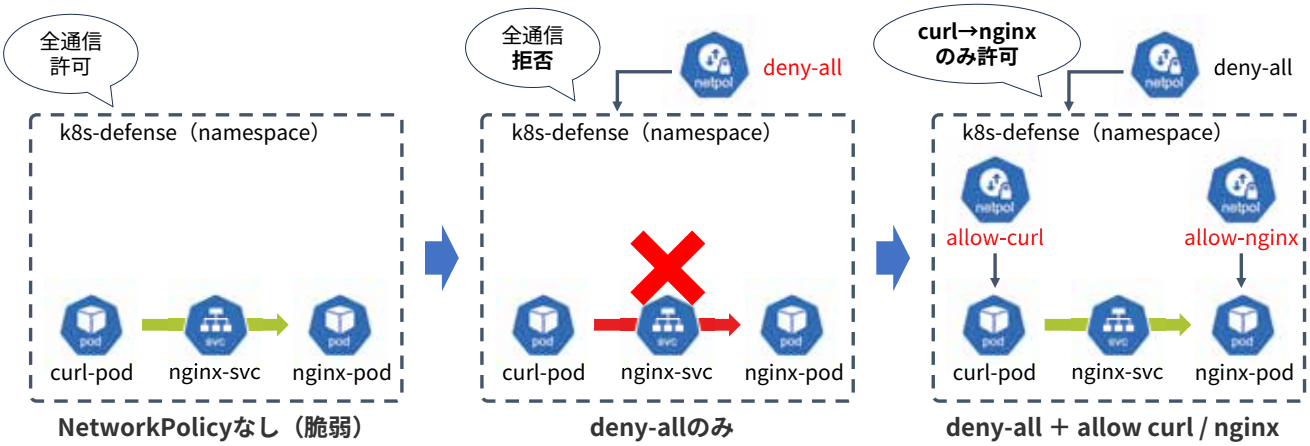
1-2：Pod間通信の最小許可

問題編

演習１　Kubernetes環境の診断と防御

1-2：期待される挙動

NetworkPolicyを2段階で適用し、Namespaceの通信を制御できることを確認しましょう。
全ての通信を閉塞した後、特定の通信経路だけ許可するのがベストプラクティスです。



演習１　Kubernetes環境の診断と防御

1-2： Pod間通信の最小許可

NetworkPolicyを設定し、Pod間の通信が制限されていることを確認しましょう。
マニフェストを作成して、kubectlコマンドで各リソースを起動してください。

検証手順

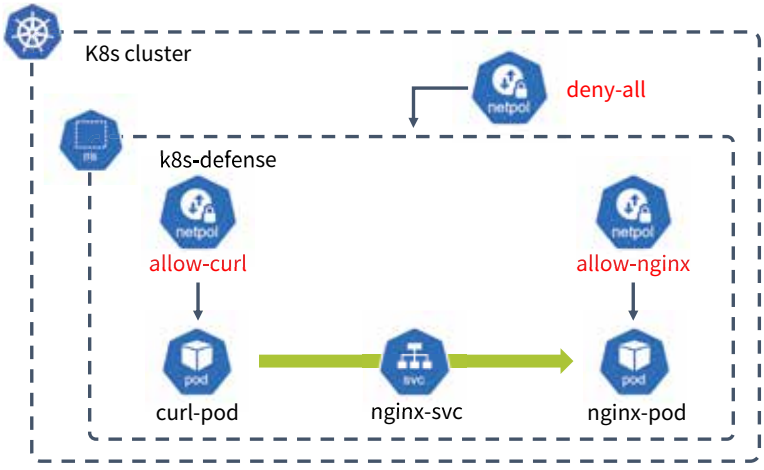
※SAはsa-limitedを適用したままでOKです

1. NetworkPolicy作成（Deny）
   Namespace内の全通信を拒否
   netpol: deny-all

2. NetworkPolicy適用後の通信可否

1. NetworkPolicy作成（Allow）
   netpol: allow-curl
   netpol: allow-nginx

2. NetworkPolicy適用後の通信可否



構築するk8s環境

演習１　Kubernetes環境の診断と防御

1-2：NetworkPolicyのコード

NetworkPolicy（deny-all）のマニフェストは以下の通りです。
DenyAllのNetworkPolicy適用後は、Namespace内の通信が全て拒否されます。

```
# deny-all policy
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: k8s-defense  #適用対象ns

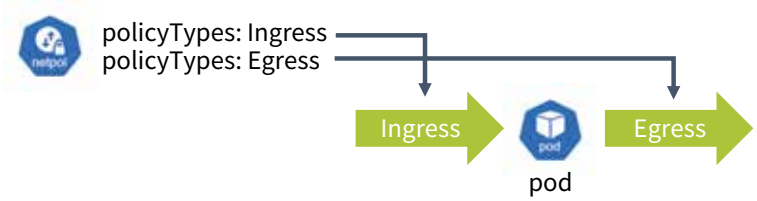
spec:
  podSelector: {}
  policyTypes:
  - Ingress  # 入力
  - Egress   # 出力
```

network-policy-denyall.yaml

※Allow用のマニフェストはご自身で作成してください。

PolicyType（Ingress/Egress）

NetworkPolicyのIngressは入力、Egressは出力。
適用リソースの入出力をそれぞれ設定することで通信が可能に。



適用対象



演習１　Kubernetes環境の診断と防御

1-2：動作確認

NetworkPolicy（deny-all）を適用したら、動作確認しましょう。
これでNamespace内の全通信が閉塞されます。

動作確認手順

1. **curl PodからNginx Podへのアクセス確認**
   curl Podからcurlコマンドを実行し、nginx に到達せずタイムアウトすること
   $ kubectl exec -n k8s-defense -it curl -- curl -m 3 http://nginx

演習１　Kubernetes環境の診断と防御

1-2：動作確認②

※NetworkPolicy（deny-all）は設定したままです。
続いて、NetworkPolicy（allow-curl、allow-nginx）を作成して、動作確認しましょう。
curl Pod、nginx Podの入出力を設定することで、curl-pod→nginx-podのみ通信が可能になります。

動作確認手順

1. **curl Podからnginx Podへのアクセス確認**
   curl Podからcurlコマンドを実行し、nginx のHTMLが表示されること
   ```
   $ kubectl exec -n k8s-defense -it curl -- curl -m 3 http://nginx
   ```

Allow用のマニフェストで、通信を可能にするヒント

allow-curl には「2つのEgressルール」を
設定してください。
・Egress（to nginx）
・Egress（to kube-system）　# K8s内DNS解決のため

allow-nginx には「1つのIngressルール」を
設定してください。
・Ingress（from curl）

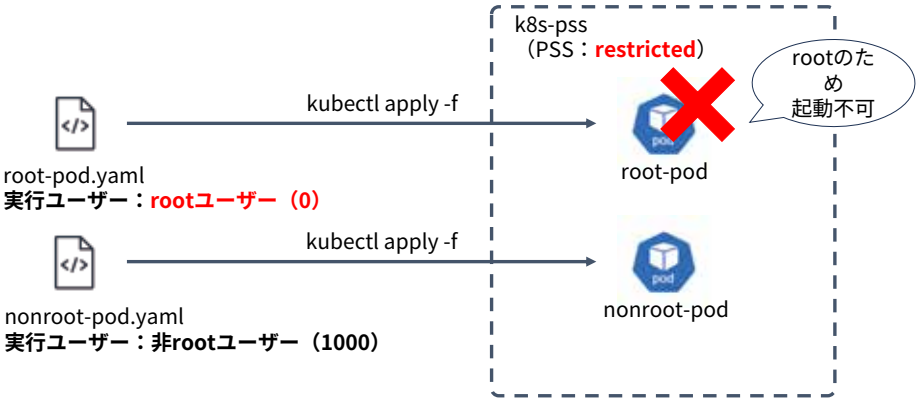
1-3：PSS適用によるrootユーザ拒否

問題編

演習１　Kubernetes環境の診断と防御

1-3：期待される挙動

k8s-pss namespaceにPSS：restrictedを適用し、rootユーザーを実行不可にします。
root-podは起動できませんが、nonroot-podであれば起動できるのを確認しましょう。



演習１　Kubernetes環境の診断と防御

1-3： PSS適用によるrootユーザ拒否

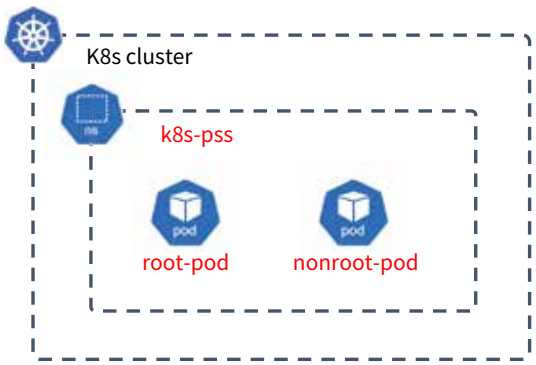
PSSを適用したNamespaceを用意し、rootユーザーのPodが起動できないことを確認します。
マニフェストを作成して、kubectlコマンドで各リソースを起動してください。

検証手順

1. 新規Namespace作成（PSS設定有）
   ns: k8s-pss

2. podの作成
   pod: root-pod
   pod: nonroot-pod

3. Podの起動可否を確認



構築するk8s環境

演習１　Kubernetes環境の診断と防御

1-3：Namespaceとroot-podのコード

Namespaceとroot-podのマニフェストは以下の通りです。
Namespaceにrestrictedが適用されると、rootユーザーでの起動はできなくなります。

```yaml
# Namespace（PSS適用）
apiVersion: v1
kind: Namespace
metadata:
  name: k8s-pss
  labels:
    # enforce:restrictedでrootは起動不可
    pod-security.kubernetes.io/enforce: restricted
    # audit:baselineは一般的な制限
    pod-security.kubernetes.io/audit: baseline
```

namespace-k8s-pss.yaml

```yaml
# root実行Pod
apiVersion: v1
kind: Pod
metadata:
  name: root-pod
  namespace: k8s-pss
spec:
  containers:
  - name: root
    image: busybox
    command: ["sh", "-c", "sleep 3600"]
    securityContext:
      runAsUser: 0  # rootユーザー
```

root-pod.yaml

※nonroot-pod.yamlは別途用意してください

演習１　Kubernetes環境の診断と防御

1-3：動作確認

root-pod、nonroot-podのマニフェストが準備出来たら、動作確認しましょう。
rootユーザーの拒否により、セキュリティが強化されました。

動作確認手順

1. **root Podの起動**
   root Podをkubectl applyしても、起動できないことを確認
   $ kubectl apply –f root-pod.yaml

2. **nonroot Podの起動**
   nonroot Podはkubectl applyで、起動できることを確認
   $ kubectl apply –f nonroot-pod.yaml

演習１ Kubernetes環境の診断と防御

合格判定基準

演習１ Kubernetes環境の診断と防御

合格判定基準

演習1の合格判定基準は以下の通りです。

1-0：脆弱なK8s環境構築

- Namespace（例：k8s-defense）、sa-admin、clusterRoleBindingが作成できている
- curl Podとnginx Podをデプロイ済みで、両方にsa-adminを適用している
- curl Podからnginx PodへHTTPアクセスできる

1-1： ServiceAccount権限の最小化

- sa-limited、role、roleBindingを作成し、curl / nginx Podにsa-limitedを適用している
- sa-limitedの権限確認で、Podへのget/listのみ許可され、それ以外の操作は拒否される

1-2：Pod間通信の最小許可

- deny-all ポリシーを適用後、Pod間通信が遮断されている（curl Podからnginx Podへの通信がタイムアウト）
- allow-curl および allow-nginx を作成し、必要なIngress/Egressのみ許可
- 再テストでcurl Podからnginx Podへの通信が成功

1-3：PSS適用によるrootユーザ拒否

- Namespace（例：k8s-pss）にenforce:restrictedラベルが設定されている
- root-pod（runAsUser: 0）をapplyした際、Forbiddenエラーになり、root実行できないことを確認
- nonroot-pod（runAsUser: 1000）をapplyし、正常にRunningになることを確認

演習 2 安全なコンテナ構築とランタイム防御

問題編

演習 2　安全なコンテナ構築とランタイム防御

演習概要

コンテナイメージの堅牢化とランタイムを保護するセキュリティ対策を実装し、現実的かつ効果的なコンテナ防御について理解を深めます。

Before：脆弱なコンテナ

脆弱なイメージ
不要なパッケージが多く、脆弱性を多く含むイメージは、Pod侵害される可能性が高まります。

署名なし
サプライチェーンやイメージの検証が実施されない場合、改ざんなどの検知が遅れます。

ランタイム制御なし
ホストシステムへのアクセスやプロセスの乗っ取りなどにより、システム全体に影響を及ぼします。

After：安全な構成

軽量化＋非root化
脆弱性が減り、コンテナの軽量化にも繋がり、セキュリティリスクが軽減されます。

署名と検証
イメージ署名とデプロイ前確認により、イメージの正しさを確認し、改ざんを検知できます。

権限制限
seccomp/AppArmorプロファイル適用により、不要な攻撃から防御できます。

演習２　安全なコンテナ構築とランタイム防御

演習内容

コンテナイメージ・レジストリの堅牢化とランタイムを保護するセキュリティ対策を実装し、現実的かつ効果的なコンテナ防御について理解を深めます。

演習内容

2-1. Trivy：Trivyでイメージをスキャンし、脆弱性を可視化。合わせてコンテナサイズを軽量化

2-2. Cosign：Cosignでイメージ署名し、イメージの検証を実現

2-3. Kyverno：Kyvernoを利用し、Cosignで署名したイメージのみクラスターにデプロイ許可

2-4. ランタイム保護：seccomp、capabilities dropによるランタイムの保護

取り扱うセキュリティ対策

Trivy
脆弱性スキャン

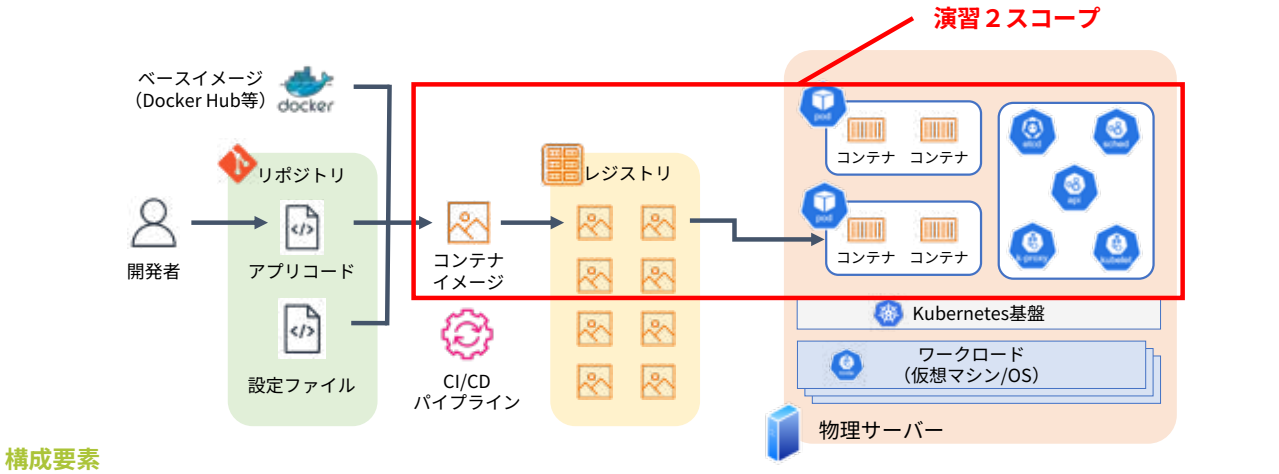
Cosign/Kyverno
イメージ署名・検証

ランタイム保護
操作制限

演習２　安全なコンテナ構築とランタイム防御

演習２の位置づけ

演習2ではイメージ、レジストリ、コンテナをターゲットにセキュリティ対策を行います。



構成要素

①コード　②イメージ　③レジストリ　④オーケストラ　⑤コンテナ　⑥ホスト

演習２　安全なコンテナ構築とランタイム防御

イメージ・レジストリにおけるセキュリティの問題

コンテナイメージに含まれる脆弱性や不要なパッケージ、レジストリでのイメージ改ざんにより、認証情報の流出やKubernetes基盤への攻撃が容易になり、基盤全体へ影響が波及する可能性があります。

⚠ 想定されるセキュリティリスク

脆弱なパッケージの悪用

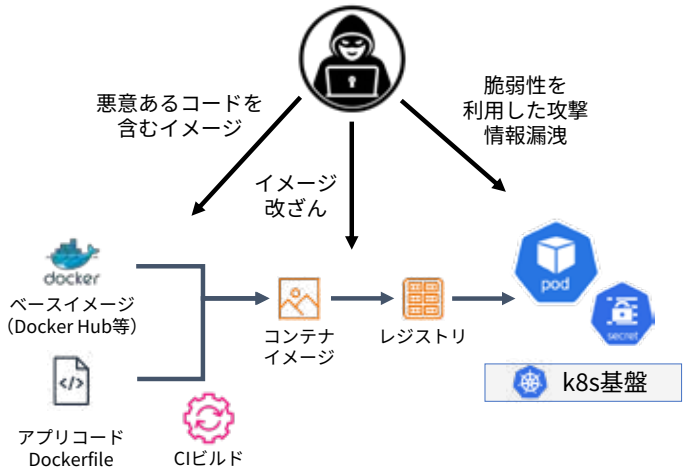
イメージの脆弱性・ツールを利用して、
悪意あるユーザーによる攻撃やシステム侵入が容易に

イメージの改ざん（署名なし）

改ざんされたイメージかどうか判別がつかず、
悪意あるコードを含んだイメージがデプロイされる

サイズ肥大化による管理漏れ

レイヤーが多く更新管理が困難になり、
セキュリティパッチ漏れなどが起きうる
シークレット情報の漏洩の可能性もあり



演習２　安全なコンテナ構築とランタイム防御

イメージ・レジストリに対するセキュリティ対策

イメージ・レジストリが抱えるセキュリティリスクに対して、
脆弱性スキャン / イメージ署名・検証 / イメージ軽量化などを実施することで防御できます。

⚠ 想定されるセキュリティリスク		✔ 実施するセキュリティ対策
脆弱なパッケージの悪用 イメージの脆弱性・ツールを利用して、悪意あるユーザーによる攻撃やシステム侵入が容易に	→	**Trivy**による脆弱性スキャン
イメージの改ざん（署名なし） 改ざんされたイメージかどうか判別がつかず、悪意あるコードを含んだイメージがデプロイされる	→	**Cosign/Kyverno**による イメージ署名・検証
サイズ肥大化による管理漏れ レイヤーが多く更新管理が困難になり、セキュリティパッチ漏れなどが起きうる シークレット情報の漏洩の可能性もあり	→	**Dockerfile**の修正による軽量化

演習２　安全なコンテナ構築とランタイム防御

コンテナランタイムおけるセキュリティの問題

コンテナランタイムに不具合がある場合、ホストへの侵入や横展開攻撃により、クラスター全体に被害が拡大するリスクがあります。

⚠ 想定されるセキュリティリスク

APIアクセス・横展開攻撃

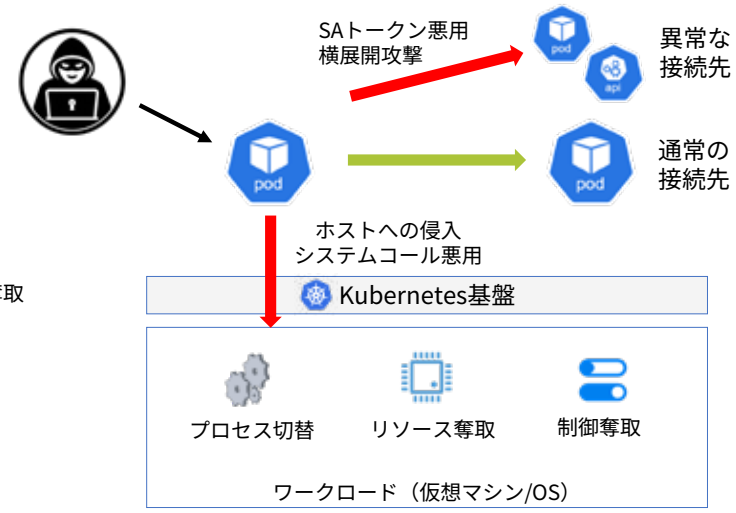
Kubernetes API経由で他Podを操作するなど、
クラスター全体に被害が出る可能性あり

ホストへの侵入

root権限や過剰なCapabilityを悪用され、
ホストOSへ侵入され、他アプリやノードの制御が奪取

システムコール悪用

ホストファイル操作や権限昇格だけでなく、
外部通信を通じた被害の拡大などが起こりうる



演習２　安全なコンテナ構築とランタイム防御

コンテナランタイムに対するセキュリティ対策

コンテナランタイムのセキュリティリスクに対して、
seccompやCapability制限などのランタイム保護を実施することで防御することが可能です。

⚠ 想定されるセキュリティリスク

APIアクセス・横展開攻撃

Kubernetes API経由で他Podを操作するなど、
クラスター全体に被害が出る可能性あり

ホストへの侵入

root権限や過剰なCapabilityを悪用され、
ホストOSへ侵入され、他アプリやノードの制御が奪取

システムコール悪用

ホストファイル操作や権限昇格だけでなく、
外部通信を通じた被害の拡大などが起こりうる

✔ 実施するセキュリティ対策

→ **非rootユーザー**実行・権限/通信制御

課題1のRBAC / Network Policy / PSSも
ランタイムのセキュリティ対策として効果的

→ **ランタイム保護**による操作制御

演習２　安全なコンテナ構築とランタイム防御

前提知識：脆弱性診断（Trivy）

イメージやファイルシステムなどの脆弱性を診断し、OSパッケージやアプリケーションの依存関係などを包括的に診断するツールです。デプロイ前にセキュリティリスクを検出できれば、事前に対処できます。

脆弱性診断の必要性

- 本番環境にデプロイする前にセキュリティ上の問題を検出
- 既知の脆弱性を含むイメージがデプロイされるリスクを低減
- 攻撃者に悪用される前に修正を適用する機会を確保

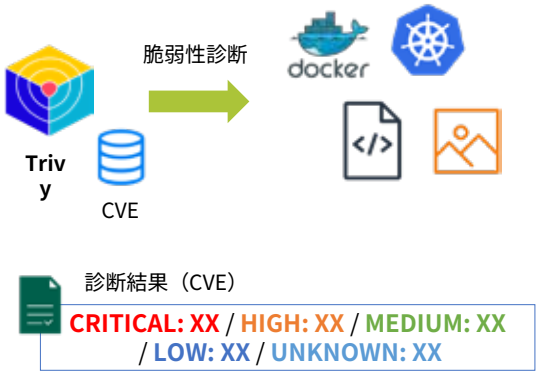
Trivyについて

演習2は、Trivyを使って脆弱性診断を行います。

- オープンソースの脆弱性診断ツール
- イメージ、ファイルシステムなどマルチスキャンが可能
- CVEベースで重大度を確認可能（Critical / Highなど）
- CI/CDパイプラインと連携可能
- 多彩な形式でレポート出力が可能

その他ツール

- Grype / Clair / Docker Scoutなど



演習２　安全なコンテナ構築とランタイム防御

前提知識：イメージ署名と検証（Cosign / Kyverno）

コンテナイメージにデジタル署名することで、イメージの改ざん防止と信頼性を保証できます。
「誰がビルドしたか」「改変されていないか」を署名と検証で確認することができます。

イメージ署名と検証の必要性

- コンテナイメージの改ざんリスクに備える
- 署名：ビルド済みイメージが正規のものか署名を付与
- 検知：正規のイメージ化判断して、環境にデプロイ

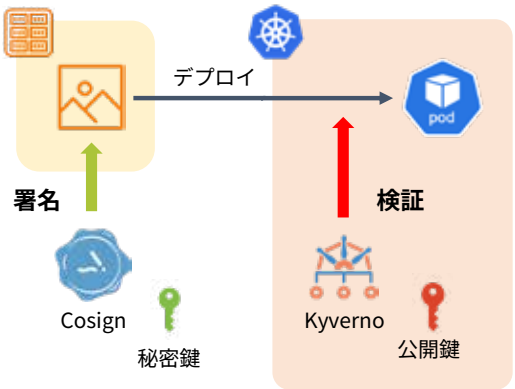
演習2では、Cosign（署名）＋Kyverno（検証）を組み合わせ、
信頼できるイメージのみ動作するクラスターを実現します。

Cosign

- イメージにデジタル**署名を付与**するツール
- 署名データはレジストリ（ECRなど）に保存
- SBOMも署名可能。

Kyverno

- KubernetesのAdmission Controllerとして動作
- デプロイ時に署名付きイメージのみ許可
- Cosign署名をポリシーで自動検証

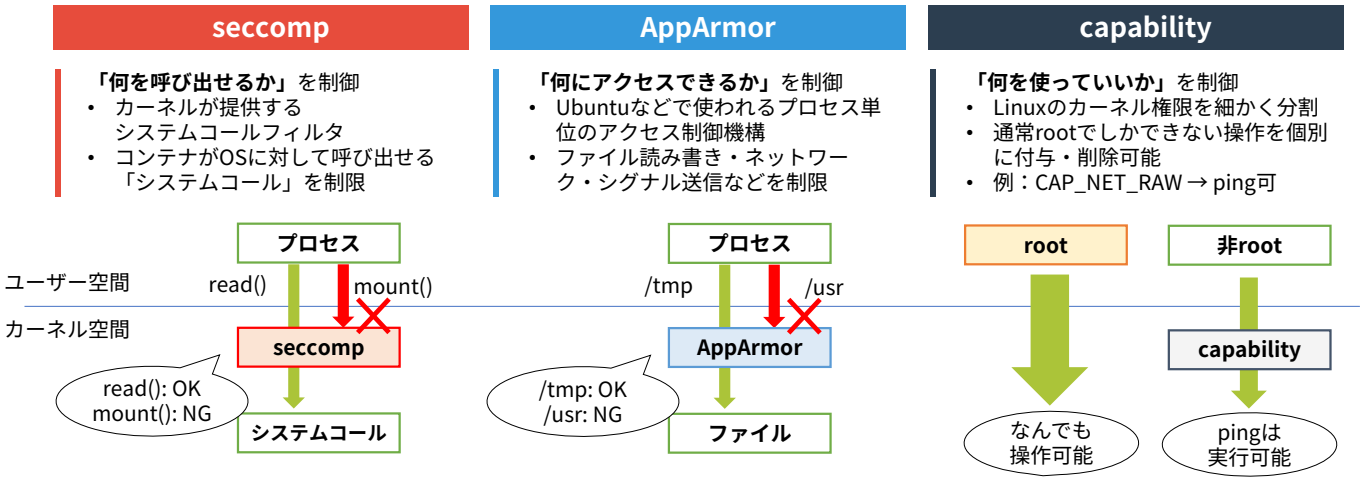


Cosign＋Kyverno のアーキテクチャ

演習２　安全なコンテナ構築とランタイム防御

前提知識：ランタイムセキュリティ制御

ランタイム実行時の攻撃面を最小化し、侵害時の影響を局所化するために導入します。
コンテナ実行時のカーネル、ファイルシステム、ネットワークへのアクセスを適切に制限します。



演習２　安全なコンテナ構築とランタイム防御

演習２の進め方

コンテナイメージとレジストリの堅牢化とランタイムを保護するセキュリティ対策を実装し、現実的かつ効果的なコンテナ防御について理解を深めます。

2-1	対策1 **Trivyによる脆弱性診断**	Trivyでイメージをスキャンして、脆弱性を可視化 Dockerfileを修正し、脆弱性とサイズ肥大化に対処
2-2	対策2 **Cosignによる署名と検証**	Cosignによるイメージ署名および検証を導入
2-3	対策2-2 **Cosign＋Kyvernoによる検証**	Kyvernoで署名済イメージのみクラスターにデプロイ可能
2-4	対策3 **ランタイム保護による操作制御**	seccomp/capabilities削減/AppArmorの適用により、 ランタイムを保護し、システムコールの利用を制限

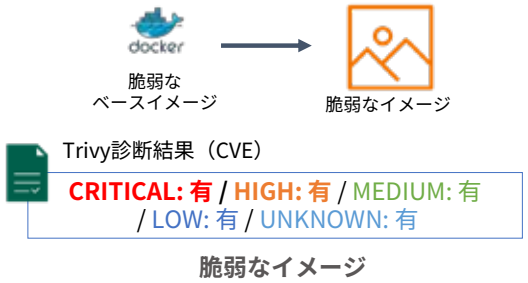
2-1：Trivyによる脆弱性診断

問題編

演習２　安全なコンテナ構築とランタイム防御

2-1：期待される挙動

Trivyを用いてコンテナイメージの脆弱性を診断し、セキュリティリスクを可視化します。
その後、Dockerfileで堅牢なイメージを作成し、脆弱性の低減と軽量化を実現します。



脆弱なイメージ



堅牢なイメージ

イメージの修正ポイント

- CRITICAL/HIGH件数: 重大な脆弱性の対処
- 脆弱なパッケージ: 不要なライブラリの削減
- サイズ: イメージサイズの軽量化（数百MB→100MB程度以下）
- ベースイメージ: 脆弱性の少ないベースの選定（alpine、distroless）
- 非root起動、権限制御

演習２　安全なコンテナ構築とランタイム防御

2-1：Trivyによる脆弱性診断

Trivyをインストールし、イメージの脆弱性診断を実行し、セキュリティリスクを可視化します。
続いてDockerfileで堅牢なイメージを作り、脆弱性の低減を確認しましょう。

演習2-1 手順

1	**Trivyのインストール**	Trivyを作業インスタンス（ローカルPC、CloudShellなど）にインストール
2	**脆弱なイメージのスキャン**	NginxのイメージをTrivyでスキャンし、セキュリティリスクを可視化
3	**堅牢なイメージの作成・診断**	Dockerfileで、ベースイメージ変更、非root化により、堅牢なイメージを作成し、脆弱性の低減、サイズ縮小を確認

演習２　安全なコンテナ構築とランタイム防御

2-1：Trivyのインストール

TrivyをローカルPCもしくはCloudShellにインストールしてください。

インストール手順

1. **Trivyのインストール**

   Trivyをインストールしてください。

   ```
   $ curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sudo sh -s -- -b /usr/local/bin
   $ trivy version
   ```

2. **Trivyの実行準備（CloudShellの場合）**

   Trivyの実行時、CVEデータベースを構築するのに1GB弱必要です。
   CloudShellの場合、home領域は容量不足となるため、暫定的に/tmpに保存する手順を載せます。

   ```
   $ mkdir -p /tmp/trivy-db
   $ export TRIVY_CACHE_DIR=/tmp/trivy-db
   ```

演習２　安全なコンテナ構築とランタイム防御

2-1：脆弱なイメージのスキャン

Trivyをインストール出来たら、nginx:latestをスキャンしてください。
nginx:latestが脆弱なイメージであることを確認しましょう。

動作確認手順

1　Nginxのイメージスキャン

Trivyでnginx:latestのイメージスキャンを行ってください。

```
$ docker pull nginx:latest
$ trivy image nginx:latest
```

実行結果

Trivyのスキャンが完了し、脆弱性診断の結果を確認しましょう。

CRITICAL / HIGHの脆弱性を含むことを確認できればOK



演習２　安全なコンテナ構築とランタイム防御

2-1：堅牢なイメージの作成・診断

Dockerfileを作成し、堅牢なイメージを作成してください。
作成したイメージをスキャンし、脆弱性およびイメージサイズの削減を確認してください。

作業手順

1　Dockerfileの作成

Dockerfileを作成してください。ただし以下要件を満たすこと。

- 簡易なアプリケーションを実装（print関数、1行程度でOK）
- 脆弱性の少ないイメージを選択（python：3.12-alpineなど）
- 非rootユーザーで起動

2　ビルドしたイメージのスキャン

docker build / tagで作成したイメージをスキャンし、脆弱性の低減を確認してください。

```
$ docker build -t xxx:xxx .
$ trivy image <イメージ名>
```

3　イメージサイズの確認

ビルドしたイメージサイズを確認し、nginx:latest に比べて小さくなっていれば良いです。

```
$ docker image ls
```

演習２　安全なコンテナ構築とランタイム防御

参照）2-1：Dockerfileのコード

Dockerfileのサンプルコードは以下の通りです。
独自のアプリケーションや別イメージを準備頂いても構いません。

```
FROM python:3.12-alpine

# ユーザー追加
RUN adduser -D appuser

# ワークディレクトリ
WORKDIR /app

# 簡易アプリ
COPY <<'APP' app.py
print("Hello Secure World!")
APP
USER appuser

# 起動時にメッセージを表示してから待機
CMD ["sh", "-c", "python3 app.py && sleep infinity"]
```

Dockerfile

演習２　安全なコンテナ構築とランタイム防御

2-1：ECRの自動イメージスキャン

ECRにはTrivyをベースとしたイメージスキャン機能が組み込まれています。
リポジトリに対してプッシュ時にスキャンを有効化することで、イメージpush時に自動で脆弱性を診断してくれます。



リポジトリ設定画面



ECRのイメージ自動スキャン結果

演習２　安全なコンテナ構築とランタイム防御

2-1：ECRを利用する場合

ECRを利用する場合は、リポジトリを用意してからイメージをpushしてください。
ECRは、2-2、2-3のイメージ署名と検証でも利用しますので、ここで準備しておきましょう。

ECR利用準備

1. **AWS ECRへのログイン**

```
$ aws ecr get-login-password --region $REGION ¥
  | docker login --username AWS --password-stdin ¥
    $(aws sts get-caller-identity --query Account --output text).dkr.ecr.$REGION.amazonaws.com
```

2. **リポジトリの作成**

ECRのGUIから作成することも可能です。イメージの自動スキャンは有効化しましょう。

```
$ aws ecr create-repository ¥
 --repository-name cs-app --region "$REGION"
 --image-scanning-configuration scanOnPush=true
```

3. **イメージのプッシュ**

```
$ docker tag <イメージ名>:<タグ名> <リポジトリURI>:latest
$ docker push <リポジトリURI>:latest
```

2-2：Cosignによる署名と検証

問題編

演習２　安全なコンテナ構築とランタイム防御

2-2：Cosignによる署名と検証

Cosignをインストールして、イメージに署名します。
署名されたイメージを検証し、署名されていることを確認しましょう。
　ローカルPCの場合はローカル、CloudShellの場合はECRに保存されたイメージに署名します。

演習2-2 手順

1. **Cosignのインストール**
   Cosignを作業インスタンスにインストール
   暗号鍵・公開鍵を作成
2. **イメージ署名**
   Cosignでローカル or ECRのイメージに署名
3. **イメージ検証**
   Cosignで署名したイメージを検証し、
   署名済みイメージであることを確認



演習２　安全なコンテナ構築とランタイム防御

2-2：Cosignのインストール

CosignをローカルPCもしくはCloudShellにインストールしてください。

インストール手順

1. **Cosignのインストール**

   Cosignをインストールしてください。
   ⚠ Cosignは**v2.4.1**を利用します。

```
$ curl -LO "https://github.com/sigstore/cosign/releases/download/v2.4.1/cosign-darwin-arm64"
$ chmod +x cosign-darwin-arm64
$ sudo mv cosign-darwin-arm64 /usr/local/bin/cosign
$ cosign version
```

2. **秘密鍵・公開鍵の作成**

   Cosignで秘密鍵・公開鍵を作成します。
   　鍵作成時にパスワードを設定します。
   　後程イメージに署名する際に利用します。

```
$ cosign generate-key-pair
$ ls -la cosign.*
```

```
prac2 $ cosign generate-key-pair
Enter password for private key:
Enter password for private key again:
Private key written to cosign.key
Public key written to cosign.pub
prac2 $ ll
total 40
-rw-------. 1 cloudshell-user cloudshell-user  653 Nov  7 04:24 cosign.key
-rw-r--r--. 1 cloudshell-user cloudshell-user  178 Nov  7 04:24 cosign.pub
-rw-r--r--. 1 cloudshell-user cloudshell-user  317 Nov  6 02:27 Dockerfile
```

演習２　安全なコンテナ構築とランタイム防御

2-2：イメージの署名

Cosignでイメージに署名しましょう。
CloudShellの場合は、事前にECRにイメージをpushし、ECR上のイメージに署名します。

イメージ署名の手順

1 **イメージ準備**

署名するイメージを準備してください。
演習２-1で作成したイメージでも大丈夫です。
　CloudShellの場合は、ECRに格納してください。

2 **イメージの署名**

Cosignでイメージに署名してください。鍵作成時のパスワード入力が必要です。

ローカルイメージに署名する場合（例：myapp:latest）
$ cosign sign --key cosign.key myapp:latest

ECR上のイメージに署名する場合
$ cosign sign --key cosign.key <ECRリポジトリURI>/myapp:latest

演習２　安全なコンテナ構築とランタイム防御

2-2：イメージの検証

それでは、Cosignでイメージを検証し、正しく署名されているかを確認しましょう。

イメージ検証の手順

1 **イメージの検証**

Cosignでイメージを検証しましょう。

ローカルイメージに署名する場合（例　myapp:latest）
$ cosign verify --key cosign.pub myapp:latest

ECR上のイメージに署名する場合
$ cosign verify --key cosign.pub <ECRリポジトリURI>/myapp:latest

実行結果

以下、メッセージが出力されていれば、正しく署名されています。

```
- The signatures were verified against the specified public key
```

2-3：Cosign＋Kyvernoによる検証

問題編

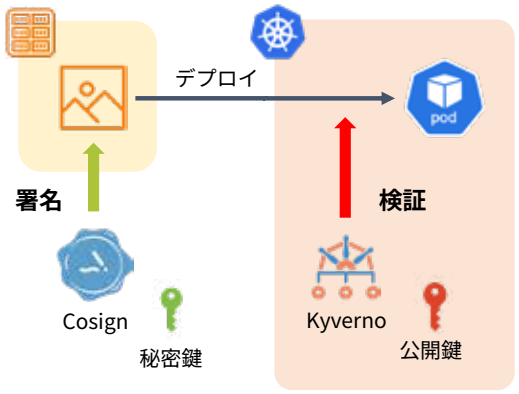
演習２　安全なコンテナ構築とランタイム防御

2-3：Cosign＋Kyvernoによる検証

KubernetesクラスターにKyvernoをインストールして、署名されたイメージのみデプロイできることを検証しましょう。

演習2-3 手順

1. **Kyvernoのインストール**
   KyvernoをKubernetesクラスターにインストール
2. **Kyvernoの署名検証ポリシー設定**
   Kyvernoの署名検証ポリシーを設定し、
   署名のないイメージはデプロイ拒否の状態にする
3. **デプロイ検証**
   Kubernetesクラスターへのデプロイを通じて、
   署名の有無による動作の違いを確認



Cosign＋Kyverno のアーキテクチャ

演習２　安全なコンテナ構築とランタイム防御

2-3：Kyvernoのインストール

KyvernoをKubernetesクラスターにインストールしてください。
Helmでのインストールを想定していますが、curlなどを利用しても構いません。

インストール手順

1 **Helmの設定**

Helm経由でKyvernoをインストールするため、Helmを設定します。

```
$ curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
$ helm version
$ helm repo add kyverno https://kyverno.github.io/kyverno/
$ helm repo update
```

1 **Kyvernoのインストール**

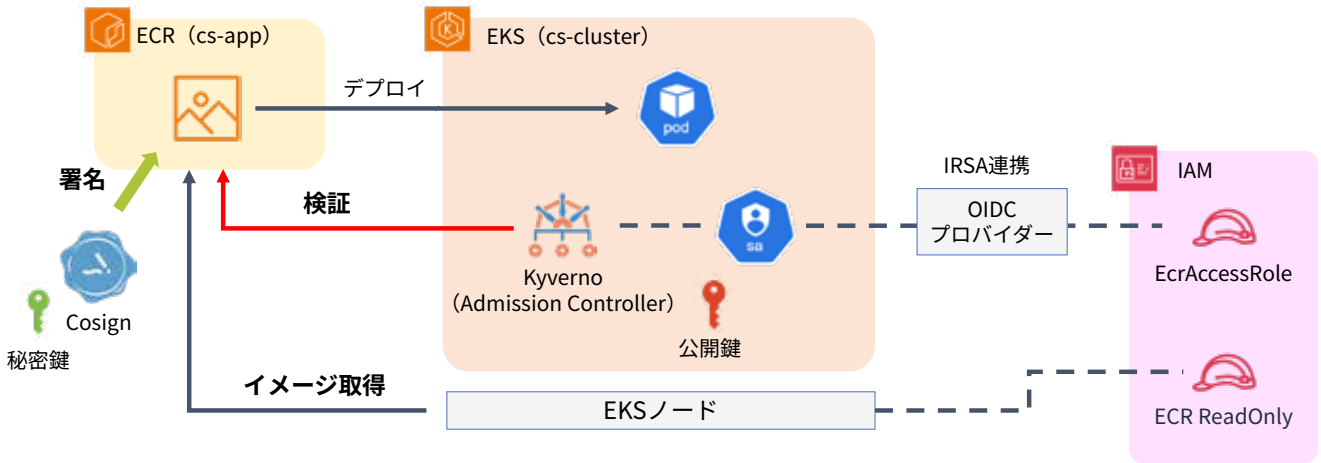
Namespace kyvernoを作成してから、Kyvernoをインストールしてください。
　kyvernoはversion:v3.4.4の方が動作が安定します。

```
$ kubectl create namespace kyverno
$ helm install kyverno kyverno/kyverno  --namespace kyverno  --version v3.4.4 ¥
  --set installCRDs=true  --set admissionController.generateSelfSignedCert=true
$ kubectl get pod -n kyverno
```

演習２　安全なコンテナ構築とランタイム防御

2-3：EKS環境設定

ECRに登録したイメージをEKSでデプロイする場合、EKSのリソースにECRを操作するためのIAMロールを付与する必要があります。ECR＋EKSの構成の場合は、SA / IAM / OIDC / IRSA連携も設定してください。
（難易度：高）



演習２　安全なコンテナ構築とランタイム防御

2-3：EKS環境設定

EKSノードおよびKyvernoからECRにアクセスし読み取り可能とするためにはAWS側の設定が必要です。
環境構築の流れを説明します。ECR＋EKSを利用している場合は、以下を実施してください。

① EKSノードへのIAMポリシー追加

- EKSノードにECR読み取り権限を付与

② Kyverno用 IAMポリシーの作成

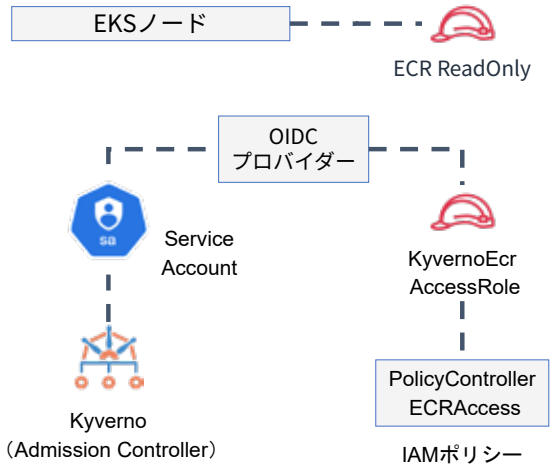
- IAMポリシーを作成し、ECRの読み取り権限を設定
- OIDCプロバイダーを作成、K8sクラスターに紐付け

③ Kyverno用 IAMロールの作成

- ServiceAccountの信頼ポリシーを作成
- IAMロールを作成し、IAMポリシーをアタッチ

④ Kyverno へのIAMロールの適用

- ServiceAccountを作成し、OIDC連携
- Kyvernoを再起動して、SAを適用



演習２　安全なコンテナ構築とランタイム防御

2-3：Kyvernoの署名検証ポリシー設定

Kyvernoの署名検証ポリシーを設定し、署名されたイメージのみKubernetesクラスターにデプロイできるように設定しましょう。

署名検証ポリシー設定

1　署名ポリシー設定

Kyvernoの署名ポリシー用マニフェストを用意して、適用してください。

cosignの公開鍵を確認して、マニフェストに貼り付けてください。
インデントに注意してください。

```
$ cat cosign.pub
$ kubectl apply -f verify-signed-images.yaml
```

cosign.pubの出力例

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQ
EAwK3p1s9E2F6rjN9HZzcv2xg3VjQ8mU7PfRzG1B6K8
-----END PUBLIC KEY-----
```

演習２　安全なコンテナ構築とランタイム防御

参照）2-3： Kyvernoの署名検証ポリシーのコード

署名検証ポリシーのサンプルコードは以下の通りです。
　keyセクションに、cosign.pubの公開鍵の値を貼り付けてください。
　署名検証対象以外のリポジトリは検証しない設定のため、必要に応じてブロックする必要があります。

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
 name: verify-signed-images
spec:
 validationFailureAction: enforce  # ポリシーの強制
 background: false
 rules:
 - name: verify-signature
   match:
    any:
    - resources:
       kinds:
       - Pod
# 右に続く
```

```
verifyImages:
  - image: “<リポジトリURI>*“
    repository: "<リポジトリURI>"
key: |
     -----BEGIN PUBLIC KEY-----
      （cosign.pubのPUBLIC KEYを貼り付けてください）
     -----END PUBLIC KEY-----
    required: true
    verifyDigest: true
    mutateDigest: true
    useCache: false
```

verify-signed-images.yaml

演習２　安全なコンテナ構築とランタイム防御

2-3：デプロイ検証

Kyvernoの設定が完了したら、設定された署名されているかを確認しましょう。
　署名検証ポリシーで定義したリポジトリが検証対象となります。

デプロイ検証の手順

1　イメージの登録

同一リポジトリに2つのイメージを登録し、
片方だけcosignで署名してください。
　ダイジェスト（sha256）の値も異なる必要があります
- 署名済イメージ（tag: latest）
- 署名がないイメージ（tag: unsigned）



ECRリポジトリのイメージ登録例

2　イメージのデプロイ確認

2つのイメージをデプロイできるか検証してください。

署名済イメージ（デプロイ成功）
$ kubectl run test-ok --image=<リポジトリURI>:latest -restart=Never

未署名イメージ（デプロイ失敗）
$ kubectl run test-ng --image=<リポジトリURI>:unsigned --restart=Never

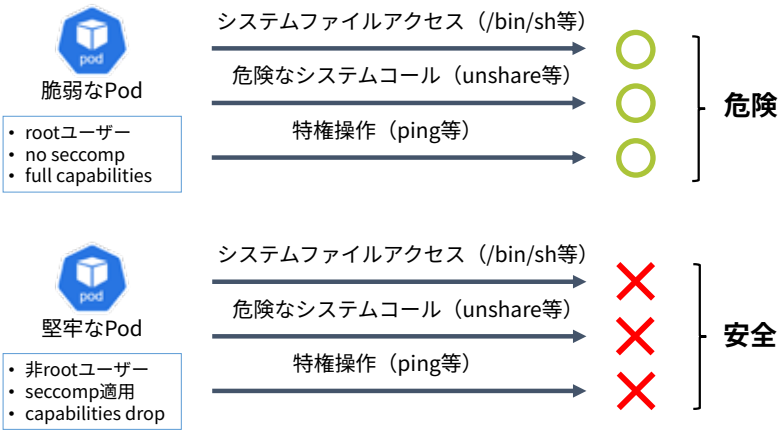
2-4：ランタイム保護による操作制御

問題編

演習２　安全なコンテナ構築とランタイム防御

2-4：期待される挙動

コンテナランタイムの保護の有無による挙動の違いを理解して、ランタイム保護の必要性を学習します。
本演習では、非root化、seccomp、capabilityを活用してコンテナランタイムを保護します。



ランタイムセキュリティ

seccomp
コンテナがOSに対して呼び出せる「システムコール」を制限します。

capability
Linuxのカーネル権限を分割し、rootでしかできない操作を個別に付与・削除します。

AppArmor
ファイル読み書き・ネットワーク・シグナル送信などをアクセス先を制限します。

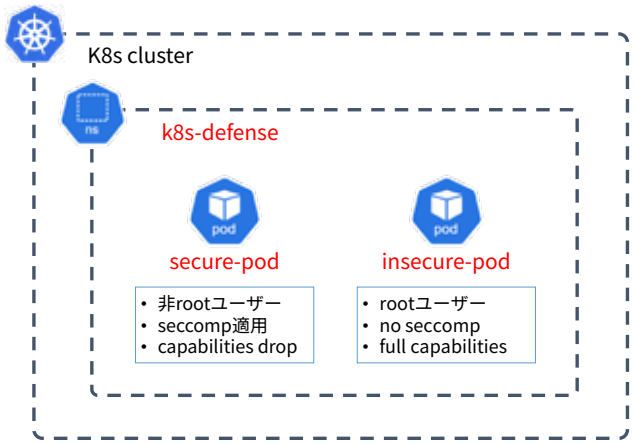
演習２　安全なコンテナ構築とランタイム防御

2-4：ランタイム保護による操作制御

Pod セキュリティ構成（ユーザー権限・seccomp・capabilities）の違いによる挙動の差分を確認し、ランタイム保護の有効性を認識しましょう。

検証手順

1. 新規Namespace作成（未作成の場合）
   ns: k8s-defense
2. podの作成
   pod: secure-pod
   pod: insecure-pod
3. Podの動作確認
   ランタイム保護の影響確認



構築するk8s環境

演習２　安全なコンテナ構築とランタイム防御

2-4：secure-podとinsecure-podのコード

ランタイムを保護したPod（secure-pod）と脆弱なPod（insecure-pod）のマニフェストは以下の通りです。非rootユーザー起動、seccomp適用、capabilitiesのdropが差分です。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
  namespace: k8s-defense
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000   # 非rootユーザー起動
    seccompProfile:   # seccompプロファイル適用
      type: RuntimeDefault
  containers:
  - name: demo
    image: alpine
    command: ["sleep", "infinity"]
    securityContext:
      allowPrivilegeEscalation: false # 特権昇格拒否
      capabilities:
        drop: ["ALL"]   # capabilityのドロップ
```

secure-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: insecure-pod
  namespace: k8s-defense
spec:
  containers:
  - name: demo
    image: alpine
    command: ["sleep", "infinity"]
    securityContext:
      allowPrivilegeEscalation: true   # 特権昇格許可
```

insecure-pod.yaml

演習２　安全なコンテナ構築とランタイム防御

2-4：動作確認

両方のPodに対するコマンドの実行結果の違いから、ランタイム保護の有効性を確認しましょう。

動作確認手順

1. **堅牢なPodからのコマンド実行**

```
$ kubectl exec -n k8s-defense secure-pod -- id
$ kubectl exec -n k8s-defense secure-pod -- unshare --user --mount
$ kubectl exec -n k8s-defense secure-pod -- ping -c 2 8.8.8.8
```

2. **脆弱なPodからのコマンド実行**

```
$ kubectl exec -n k8s-defense insecure-pod -- id
$ kubectl exec -n k8s-defense insecure-pod -- unshare --user --mount
$ kubectl exec -n k8s-defense insecure-pod -- ping -c 2 8.8.8.8
```

想定される挙動

Pod	id	unshare	ping
secure-pod	非root（1000）	失敗	失敗
insecure-pod	root（0）	成功	成功
備考	非root化でPSSに対応	危険なシステムコール →seccomp適用で抑止	root権限の操作 →capabilities dropで root操作を抑止

演習２　安全なコンテナ構築とランタイム防御

合格判定基準

演習２　安全なコンテナ構築とランタイム防御

合格判定基準

演習2の合格判定基準は以下の通りです。

2-1：Trivyによる脆弱性スキャン

- Trivyのインストールが完了していること。
- nginx:latest のイメージスキャンを実行でき、脆弱性診断の結果が表示されること。
- Dockerfileを利用してイメージをビルドし、脆弱性の低減（CRITICAL/HIGHが0件）、サイズ軽量化ができていること。

2-2：Cosignによる署名と検証

- Cosignのインストールが完了し、鍵ペアを作成できていること。
- cosign sign によりローカルもしくはECRのイメージに署名できていること。
- cosign verify により署名したイメージを正しく検証できること。

2-3：Cosign＋Kyvernoによる検証

- Kyvernoのインストールが完了し、Podが正常に稼働していること。
- イメージレジストリに、署名済と未署名のイメージの2つを登録できていること。
- Kyvernoの署名検証ポリシーが適用され、署名済イメージのみデプロイできること（検証対象のリポジトリに限る）
- （EKSの場合）SA / IAM / OIDC / IRSA連携などを用意し、EKS→ECRに対してイメージ検証・取得ができる状態なこと。

2-4：ランタイム保護による操作制限

- 堅牢なPodと脆弱なPodを起動できていること。
- 両Podで、id、unshare、pingを実行し、挙動の違いを確認できていること。

演習３ パイプラインへのセキュリティゲート導入

問題・解答編

演習３ パイプラインへのセキュリティゲート導入

演習概要

Shift-Leftの概念に基づき、コードの静的解析・依存ライブラリ検査・イメージスキャンを自動化し、安全なCIパイプラインを構築します。アプリは簡易なPythonを用意して、CIパイプラインを回します。

Before：危険な開発プロセス

手動レビュー依存
コードレビューは属人的な場合、チェックに限界があり、レビュー品質がばらつきやすい。

本番環境への脆弱なコードの混入
セキュリティチェックが不十分な場合、脆弱なコードのままそのまま本番環境にデプロイされる。

依存ライブラリの脆弱性を検知できない
外部パッケージの脆弱性に気づかず、知らない間に攻撃範囲が増大している可能性がある。

After：安全なパイプライン

PR作成時の自動スキャン
コードレビューの自動化により機械的・網羅的な検査が可能。自動化によりレビューの抜け漏れを防止

重大度の高い脆弱性を含むコードをブロック
安全ではないコードをチェックし、重大な脆弱性がある場合は、ビルドを停止することで品質保証を強化

依存ライブラリの脆弱性を自動検知・通知
Dependabotで自動通知を利用し、知らないうちに脆弱なライブラリを使っている状態を防止

演習３ パイプラインへのセキュリティゲート導入

演習内容

Shift-Leftの概念に基づき、コードの静的解析・依存ライブラリ検査・イメージスキャンを自動化し、安全なCIパイプラインを構築します。アプリは簡易なPythonを用意して、CIパイプラインを回します。

CIパイプライン構成



利用するツール

Ruff+Black＋CodeQL
コードの静的解析

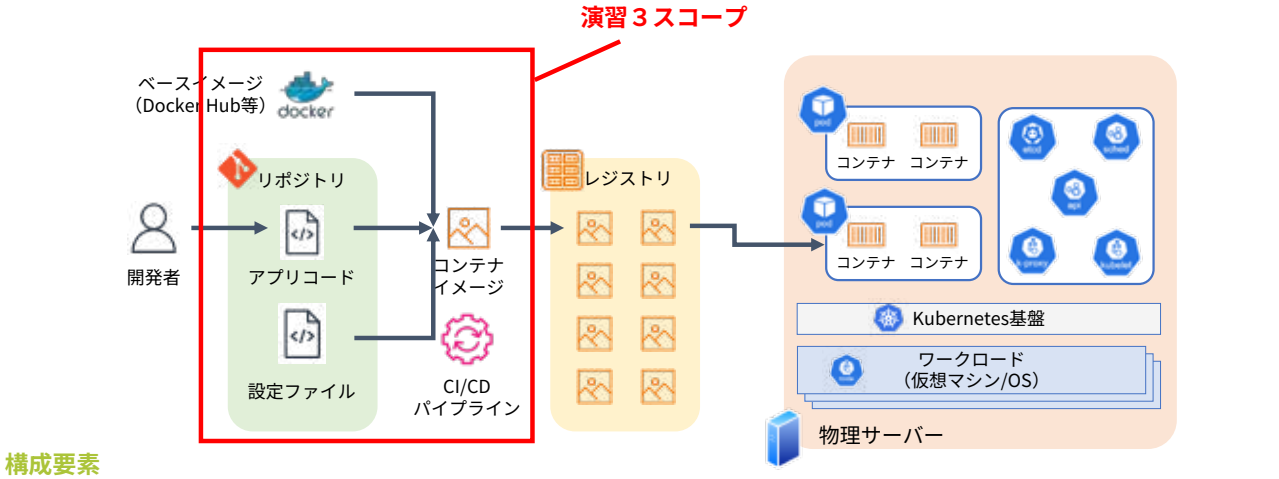
Trivy
脆弱性診断

Dependabot
依存ライブラリ監視
※CIパイプライン外に実装

演習３ パイプラインへのセキュリティゲート導入

演習３の位置づけ

演習3ではコード、イメージをターゲットにセキュリティ対策を行います。



構成要素

①コード　②イメージ　③レジストリ　④オーケストラ　⑤コンテナ　⑥ホスト

演習３ パイプラインへのセキュリティゲート導入

コードにおけるセキュリティの問題

低品質なコードや脆弱なコーディング、依存ライブラリといった問題が放置されると、意図しない脆弱性が本番環境まで持ち込まれ、攻撃の踏み台や情報漏洩につながる危険があります。

⚠ 想定されるセキュリティリスク

低品質なコード

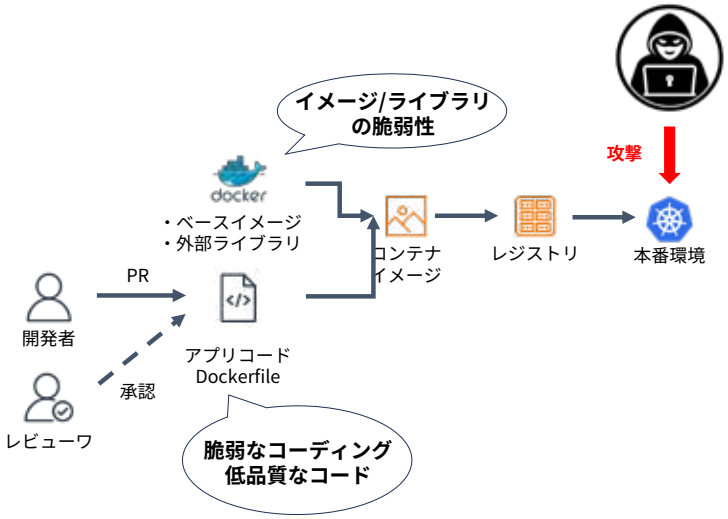
低品質なコードは潜在的な脆弱性を内包し、
長期的なセキュリティリスクとなります。

脆弱なコーディングパターン

意図せず挿入されたセキュリティホールにより、
システム侵害や情報漏洩の可能性が高まります。

依存ライブラリの脆弱性

ライブラリに含まれる既知の脆弱性が、
アプリ全体のセキュリティリスクとなります。



演習３ パイプラインへのセキュリティゲート導入

コードに対するセキュリティ対策

コード品質・脆弱なコーディング・依存ライブラリの脆弱性に対して、Shift-Leftの考えに基づき、自動化されたセキュリティゲートで早期に検知することで、パイプライン全体の安全性を高めることができます。

想定されるセキュリティリスク

低品質なコード

低品質なコードは潜在的な脆弱性を内包し、
長期的なセキュリティリスクとなります。

脆弱なコーディングパターン

意図せず挿入されたセキュリティホールにより、
システム侵害や情報漏洩の可能性が高まります。

依存ライブラリの脆弱性

ライブラリに含まれる既知の脆弱性が、
アプリ全体のセキュリティリスクとなります。

実施するセキュリティ対策

パイプラインへのセキュリティゲート構築

- パイプライン（GitHub Actions）
- コードスキャン（Ruff+Black）
- コードロジック検証（CodeQL）
- イメージ脆弱性診断（Trivy）
- 依存ライブラリ検出（Dependabot）

演習３ パイプラインへのセキュリティゲート導入

前提知識：Shift-Left

開発ライフサイクルの早い段階（左側）でセキュリティ検査を実行する考え方です。従来の「開発・テスト後にセキュリティ検査」という流れから「開発中からセキュリティ検査を随時行う」にシフトします。



シフトレフトの必要性

早期検査によるリスク低減
PR作成やコミット時にセキュリティチェックすることで早期に脆弱性を発見・是正可能

リリース速度の維持
継続的なセキュリティチェックにより、後工程での手戻りを減少、速いリリースサイクルを維持

修正コストの最小化
開発初期に脆弱性を検出することで、修正にかかるコストを最小限に抑えられる

CIによる自動化の重要性
CIパイプラインでの自動化により一貫した品質担保し、人的ミスを排除

演習３ パイプラインへのセキュリティゲート導入

前提知識：GitHub Actions

簡潔な構文でパイプライン（ワークフロー）を実行するCI/CDプラットフォームです。
ワークフローをYAMLファイルで定義し、リポジトリに配置するだけで自動実行されます。

GitHub Actionsの基本

- YAMLファイルによる宣言的ワークフロー定義
- イベント駆動型（push、PRなど）でワークフロー実行
- マーケットプレイスから既存のアクションを流用可能
- OIDCを利用してAWS等のクラウドに安全接続

演習3で実装するCIパイプライン（ワークフロー）



ワークフロー定義例

(.github/workflows/ci.yaml)

```
name: CI
on:
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
      - run: npm ci
      - run: npm test
```

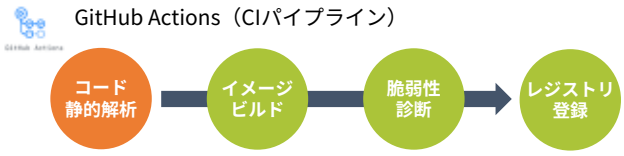
演習３ パイプラインへのセキュリティゲート導入

前提知識：コードの静的解析（Ruff＋Black＋CodeQL）

コードの静的解析は、人間の見落としを防ぎ、コードを動かす前にバグ・不整合・脆弱性を自動で検出することができます。結果として、品質向上・レビュー効率化・開発コスト削減に直結します。

コード解析のツールは、言語、用途、プロジェクト規模などに合わせて、最適なものを選ぶ必要があります。

演習3では**Python**のアプリケーションを扱うため、
Ruff＋Black＋CodeQLを組み合わせたCIパイプラインを用意し、コード解析を自動化します。



Ruff　高速Python**リンター**。バグの温床となる"怪しいコード"を機械的に検出。Flake8/pyflakes/pylintの主要ルールを網羅しつつ高速・省リソース。500以上のルールをカスタマイズ可能。

Black　自動コード**フォーマッター**。コードを一貫したスタイルに整えて、可読性とレビュー効率を最大化。チーム内のコーディングスタイルを統一し、差分を最小化。

CodeQL　**セキュリティ解析**エンジン。コードの流れや動作を解析し、セキュリティの欠陥を検出。SQLインジェクションなどのアプリの脆弱性を検知。

演習３ パイプラインへのセキュリティゲート導入

前提知識：パッケージ脆弱性検出（Dependabot）

イメージやファイルシステムなどの脆弱性を診断し、OSパッケージやアプリケーションの依存関係などを包括的に診断するツールです。デプロイ前にセキュリティリスクを検出できれば、事前に対処できます。

パッケージ脆弱性検出の必要性

パッケージの脆弱性は早期に発見・対処することが望まれます。
- アプリケーションで依存パッケージ（OSS）を使うのが現在の開発では当たり前
- ライブラリは頻繁に更新され、ライブラリ間の依存関係も複雑で、手動で追うのは困難
- 実装後に新しい脆弱性が見つかった場合、そのまま放置され、攻撃されうる

Dependabot

GitHubに組み込まれたパッケージ脆弱性検出ツールです。
- プロジェクト内のパッケージの脆弱性を自動的に検出。既知のCVEに基づいてリスクを評価
- 脆弱性が発見された依存関係や更新可能なパッケージに対して 、自動的にプルリクエストを生成
- パッケージエコシステム（npm、pip、docker等）ごとにチェック頻度を指定可能。

演習３ パイプラインへのセキュリティゲート導入

演習３の進め方

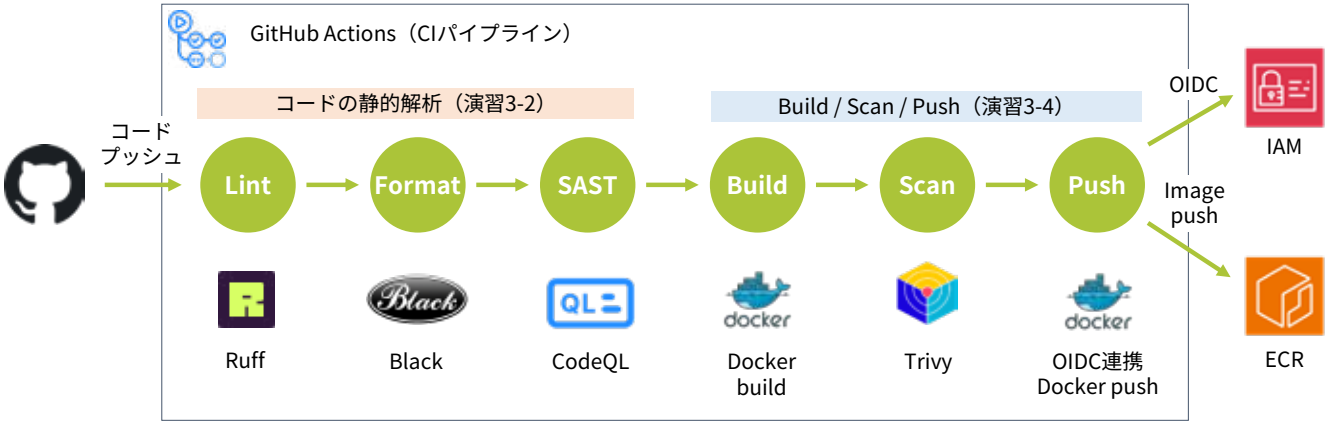
Shift-Leftの概念に基づき、コードの静的解析・依存ライブラリ検査・イメージスキャンを自動化し、安全なCIパイプラインを構築します。アプリは簡易なPythonを用意して、CIパイプラインを回します。

3-1	事前準備	GitHubプロジェクト、アプリケーションコードなどを準備します
3-2	対策1 Ruff / Black / CodeQL によるコードの静的解析	GitHub ActionsのワークフローにRuff / Black / CodeQLを定義し、Pythonコードの静的解析を自動化
3-3	事前準備②	IAM、OIDC、ECRリポジトリを準備し、GitHubからECRにイメージをプッシュ可能にします
3-4	対策2 Build / Scan / Push のCI統合	ワークフローにイメージビルド、脆弱性チェック、ECRへのイメージプッシュを統合し、CIパイプラインを拡充
3-5	対策3 Dependabotによるパッケージ脆弱性検出	Dependabotを設定し、プロジェクト内のパッケージの脆弱性検出を自動化

演習３ パイプラインへのセキュリティゲート導入

CIパイプライン構成

演習3-1～3-4でCIパイプラインを構成します。CIパイプラインで、Pythonコード解析、イメージビルド、脆弱性スキャンを実施し、最終的にはECRにイメージ登録するところまで実装します。



※本演習はCIパイプライン構築を取り扱います。お時間のある方は試験やCDのパイプライン構築に挑戦してみてください。

演習３ パイプラインへのセキュリティゲート導入

演習環境

演習3は、GitHub、AWSを利用して進めます。
GitHubおよびAWSのアカウント（IAM操作権限含む）を準備してから取り組んでください。

3-1：事前準備

問題・解答編

演習３ パイプラインへのセキュリティゲート導入

3-1：事前準備

CIパイプラインを構築するため、GitHubリポジトリおよびアプリケーションを準備します。
GitHubアカウントを準備してから取り組んでください。

演習3-1手順

1 **GitHubリポジトリ作成**

以下要件で、新規リポジトリ**sample-app**を作成してください。

- Repository name：sample-app
- Public / Private：どちらでもOK
- .gitignore：有効にして、Pythonを選択
- License: 任意

2 **GitHubリポジトリ構成**

簡易なPythonアプリケーションおよびDockerfileを準備し、
sample-appリポジトリに登録してください。

- requirements.txtは必ず用意してください
- サンプルコードは用意しますが、独自に準備頂くことも可能です

```
sample-app/
├── app.py
├── requirements.txt
├── Dockerfile
└── .gitignore
```

sample-app
リポジトリ構成例

演習３ パイプラインへのセキュリティゲート導入

3-1：事前準備

簡易なアプリケーション関連のサンプルコードは以下の通りです。
独自のアプリケーションを準備頂いても構いません。

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/")
def index():
    return jsonify({"message": "Hello from CI/CD Pipeline!"})

@app.route("/health")
def health():
    return jsonify({"status": "ok"})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

app.py

```
flask==2.3.2
```

requirements.txt

```
FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

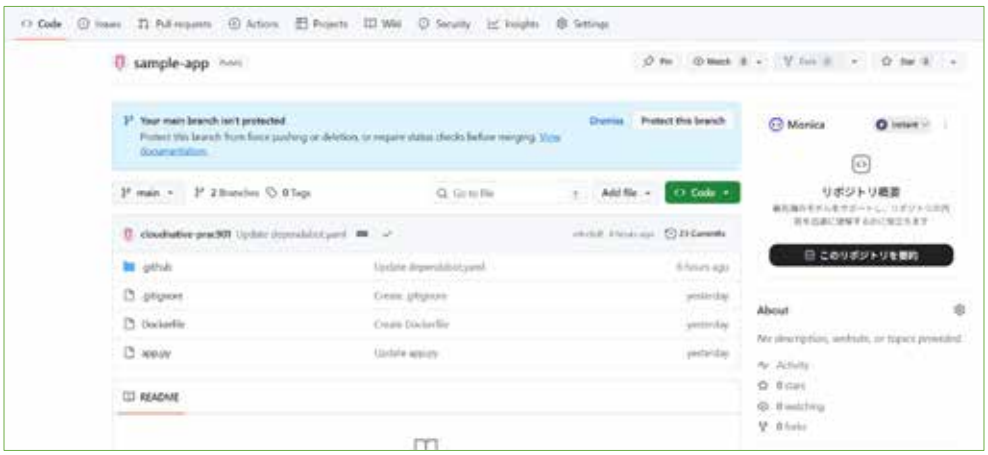
EXPOSE 5000
CMD ["python", "app.py"]
```

Dockerfile

演習３ パイプラインへのセキュリティゲート導入

3-1：事前準備

sample-appリポジトリ構成が以下の様になっていれば、演習3-1は完了です。



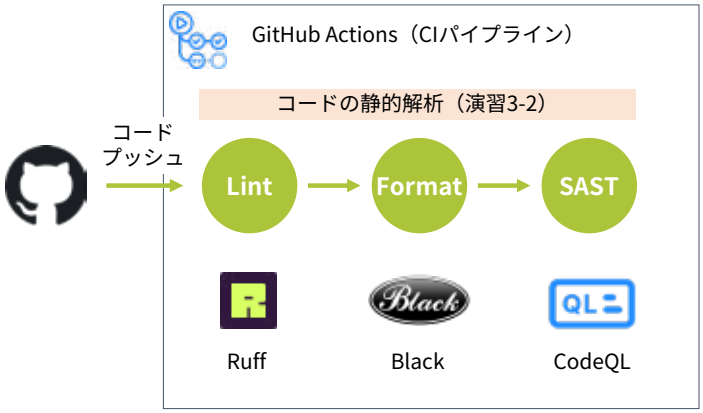
3-2：Ruff / Black / CodeQL によるコードの静的解析

問題・解答編

演習３ パイプラインへのセキュリティゲート導入

3-2：期待される挙動

GitHub Actionsのワークフローを準備し、Pythonコードの静的解析を定義してください。
1つのワークフローに Lint / Format / SAST のジョブを順番に定義して、静的解析を自動化します。



Lintジョブ（Ruff）
Python**リンター、Ruff**を使用し、
コードの誤りを指摘

Formatジョブ（Black）
Python自動**フォーマッター、Blackを使用し、**
コードを一貫したスタイルに整形

SASTジョブ（CodeQL）
セキュリティ解析エンジン、CodeQLを利用し、
コードの流れや動作から脆弱性を検出

演習３ パイプラインへのセキュリティゲート導入

3-2： Ruff / Black / CodeQL によるコードの静的解析

GitHub Actionsのワークフローを準備し、Pythonコードの静的解析を定義してください。
1つのワークフローに Lint / Format / SAST のジョブを順番に定義して、静的解析を自動化します。

演習3-2手順

1. **GitHub Actions ワークフロー作成**
   .github/workflow/secure-pipeline.yaml
   を作成してください。
2. **Lintジョブの追加**
   Ruffを実行するLintジョブを追加してください。
3. **Formatジョブの追加**
   Blackを実行するFormatジョブを追加してください。
4. **SASTジョブの追加**
   CodeQLを実行するSASTジョブを追加してください。

```
sample-app/
├── .github/
│   └── workflows/
│       └── secure-pipeline.yaml
├── app.py
├── requirements.txt
├── Dockerfile
└── .gitignore
```

sample-app
リポジトリ構成例
※赤字ファイルを追加

演習３ パイプラインへのセキュリティゲート導入

3-2：GitHub Actions ワークフローテンプレート

GitHub Actions ワークフロー（.github/workflow/secure-pipeline.yaml）のテンプレートを提示します。
以降、Jobをいくつか追加してCIパイプラインを拡充します。

```yaml
name: Secure DevSecOps Pipeline

on:   # 起動契機
  push:
    branches: [main]
  pull_request:
    branches: [main]

permissions:   # パーミッション
  id-token: write       # OIDC 用
  contents: read
  security-events: write  # CodeQL 用

env:   # 環境変数
  AWS_REGION: ap-northeast-1
  ECR_REPOSITORY: sample-app
  IMAGE_TAG: latest

jobs:   # 以降、Job定義を追加
```

.github/workflow/secure-pipeline.yaml

ワークフローの構成イメージ

3-2、3-4でワークフローを拡充します。
演習3実施後の構成例は以下の様になります。

- name
- on
- permissions
- env
- jobs
  - lint（Ruff）
  - format（Black）
  - sast（CodeQL）
  - build_scan_push
    （Docker Build + Trivy + DockerPush）

演習３ パイプラインへのセキュリティゲート導入

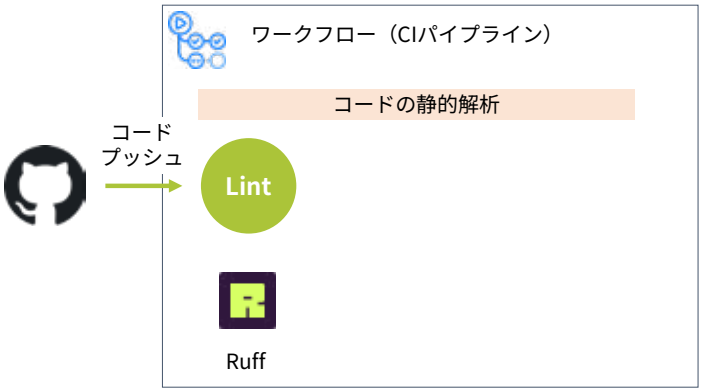
3-2：Lintジョブ（Ruff）の追加

ワークフローにLintジョブを追加して、コード解析が出来るようにしてください。
今回はRuffでエラーを検知しても、そのまま後続に進むように実装してください。

Lintジョブ要件

- Pythonのコード解析をするジョブです
- Ruffをインストールして実行する
- Ruffの実行結果がfailedであっても、
  後続ジョブに進んでください

ワークフロー構築状況



演習３ パイプラインへのセキュリティゲート導入

3-2：Lintジョブ（Ruff）サンプルコード

Lintジョブのサンプルコードを提示します。

```yaml
jobs:
  lint:
    name: Ruff Lint
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.11"

      - name: Install Ruff
        run: pip install ruff

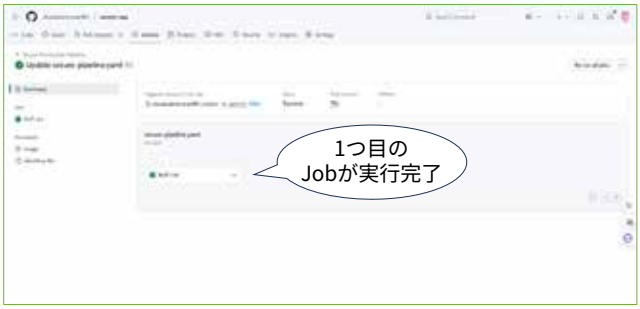
      - name: Run Ruff Lint
        run: ruff check . || true   # ruff failedでも継続
```

Lintジョブ　サンプルコード

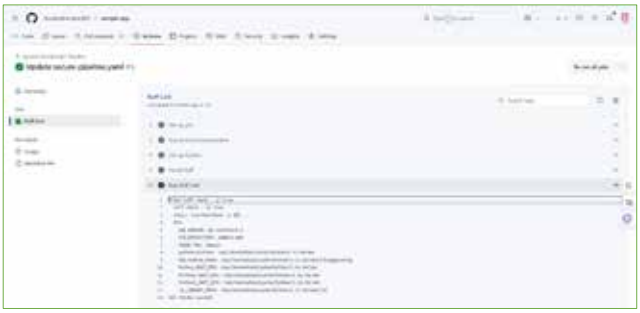
演習３ パイプラインへのセキュリティゲート導入

3-2：動作確認（Lintジョブ）

ワークフローにLintジョブを追加後に、GitHubで動作確認してください。
ワークフローの実行結果にエラーがなく、Ruff Lintも完了していればOKです。



GitHubリポジトリ > Actionsタブ >
ワークフローを選択



Ruff Lintの詳細
Run Ruff Lintから、All Check passedを確認可能

演習３ パイプラインへのセキュリティゲート導入

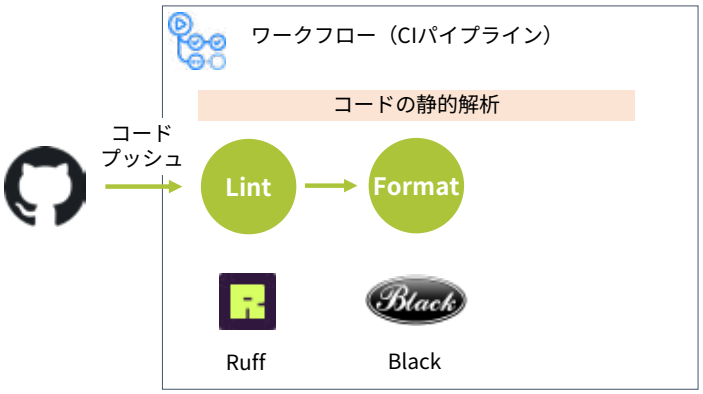
3-2：Formatジョブ（Black）の追加

ワークフローにFormatジョブを追加して、コード解析が出来るようにしてください。
今回はBlackでエラーを検知しても、そのまま後続に進むように実装してください。

Formatジョブ要件

- Pythonのコードを整形するジョブです
- Blackをインストールして実行する
- Blackの実行結果がexit 1を返却しても、
  後続ジョブに進んでください

ワークフロー構築状況



演習３ パイプラインへのセキュリティゲート導入

3-2：Formatジョブ（Black）サンプルコード

Formatジョブのサンプルコードを提示します。

```yaml
jobs:
  format:
    name: Black Format Check
    runs-on: ubuntu-latest
    needs: lint  # lint成功後に実行

    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.11"

      - name: Install Black
        run: pip install black

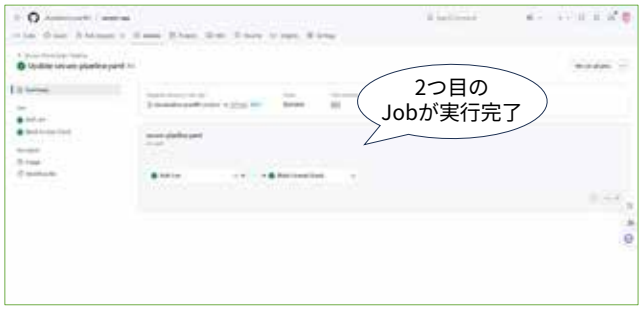
      - name: Run Black (check mode)
        run: black --check . || true  # Black が exit 1 を返しても後続に進む
```

Formatジョブ　サンプルコード

演習３ パイプラインへのセキュリティゲート導入

3-2：動作確認（Formatジョブ）

ワークフローにFormatジョブを追加後に、GitHubで動作確認してください。
ワークフローの実行結果にエラーがなく、Black Format Checkが完了しエラーがなければOKです。
　Blackを実行した結果、フォーマットエラーがある場合は必要に応じて修正してください。



GitHubリポジトリ > Actionsタブ >
ワークフローを選択



Black Format Checkの詳細
Run Black（check mode）から
フォーマットエラーの有無を確認可能

演習３ パイプラインへのセキュリティゲート導入

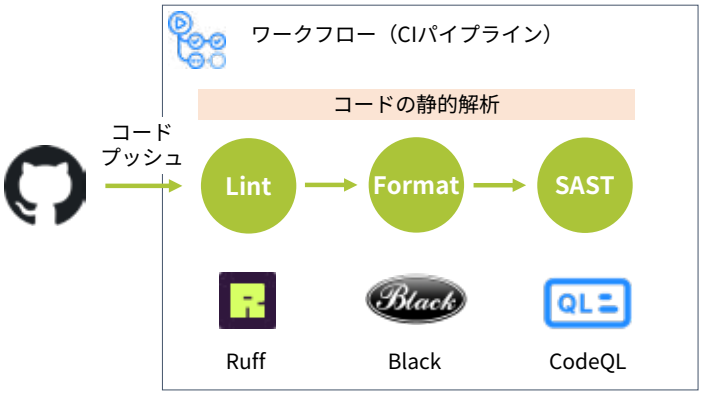
3-2：SASTジョブ（CodeQL）の追加

ワークフローにSASTジョブを追加して、コード解析が出来るようにしてください。

SASTジョブ要件

- Pythonのコードの脆弱性を確認するジョブです
- CodeQLをインストールして実行する
- GitHubのCodeScanningが有効になること

ワークフロー構築状況



演習３ パイプラインへのセキュリティゲート導入

3-2：SASTジョブ（CodeQL）サンプルコード

SASTジョブのサンプルコードを提示します。

```
jobs:
 sast:
  name: CodeQL Analysis
  runs-on: ubuntu-latest
  needs: format  # format成功後に実行

  steps:
   - name: Checkout code
     uses: actions/checkout@v4

   - name: Initialize CodeQL
     uses: github/codeql-action/init@v4
     with:
       languages: python

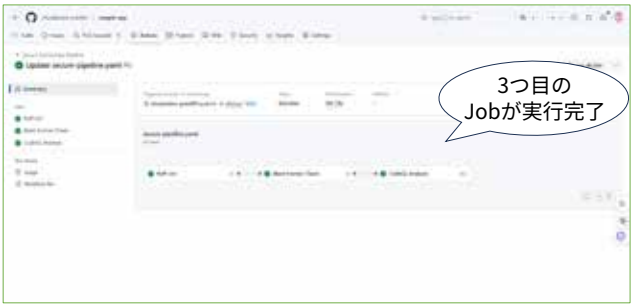
   - name: Perform CodeQL Analysis
     uses: github/codeql-action/analyze@v4
```

SASTジョブ　サンプルコード

演習３ パイプラインへのセキュリティゲート導入

3-2：動作確認（SASTジョブ）

ワークフローにSASTジョブを追加後に、GitHubで動作確認してください。
ワークフローの実行結果にエラーがなく、CodeQL Analysisが完了してることを確認してください。



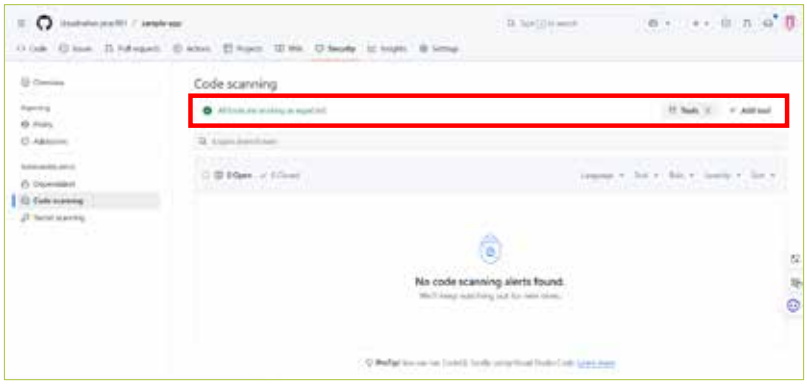


CodeQL Analysisの詳細

演習３ パイプラインへのセキュリティゲート導入

3-2：動作確認（SASTジョブ）

CodeQLのスキャン結果は、GitHubのセキュリティタブから確認できます。
Securityタブ > CodeScanning からスキャン結果に問題がないことを合わせて確認してください。



Code Scanningの結果

3-3：事前準備②

問題・解答編

演習３ パイプラインへのセキュリティゲート導入

3-3：事前準備②

CIパイプラインでビルドしたイメージをECRにプッシュするため、ECRにリポジトリを作成します。またGitHubからAWSにアクセスするためOIDC連携を設定します。AWSコンソールを開いて作業しましょう。

演習3-3手順

1 AWS ECRリポジトリ作成

ECRに新規リポジトリ**sample-app**を作成してください。

Amazon ECR > Private Repository > Repositories > 「リポジトリを作成」

- リポジトリ名：sample-app
- イメージタグ：mutable
- 暗号化設定：AES-256
- プッシュ時にスキャン：任意

2 OIDCプロバイダ作成

IAMからIDプロバイダを作成してください。

IAM > IDプロバイダ > 「プロバイダを追加」

- プロバイダタイプ：OpenID Connect
- プロバイダのURL：https://token.actions.githubusercontent.com
- 対象者：sts.amazonaws.com

演習３ パイプラインへのセキュリティゲート導入

3-3：事前準備②

CIパイプラインでビルドしたイメージをECRにプッシュするため、AWS ECRにリポジトリを作成します。
またGitHubからAWSにアクセスするためのOIDC連携を設定します。

演習3-3手順

3 IAMロール作成

IAMで新規IAMロールを作成し、適切なエンティティとIAMポリシーを設定ください。
IAM > ロール > 「ロールを作成」

- 信頼されたエンティティタイプ：ウェブアイデンティティ
- アイデンティティプロバイダ：token.actions.githubusercontent.com
- Audience：sts.amazonaws.com
- GitHubOrganization：<GitHubアカウント名>
- GitHub repository：sample-app
- GitHub branch：*

- 許可ポリシー：AmazonEC2ContainerRegistryFullAccess　のアタッチ
- IAMロール名：github-actions-devsecops-role

演習３ パイプラインへのセキュリティゲート導入

3-3：事前準備②

AWS ECRにsample-appリポジトリが作成されていれば、ECRの準備は完了です。



AWS ECR / sample-appリポジトリ

演習３ パイプラインへのセキュリティゲート導入

3-3：事前準備②

AWS IAMに新規IDプロバイダおよび新規IAMロールが作成されていれば、IAMの準備は完了です。
IAMロールにはIAMポリシー（AmazonEC2ContainerRegistryFullAccess）の付与を確認してください。



IDプロバイダ /
token.actions.githubusercontent.com



IAMロール /
github-actions-devsecops-role
IAMポリシー：AmazonEC2ContainerRegistryFullAccess

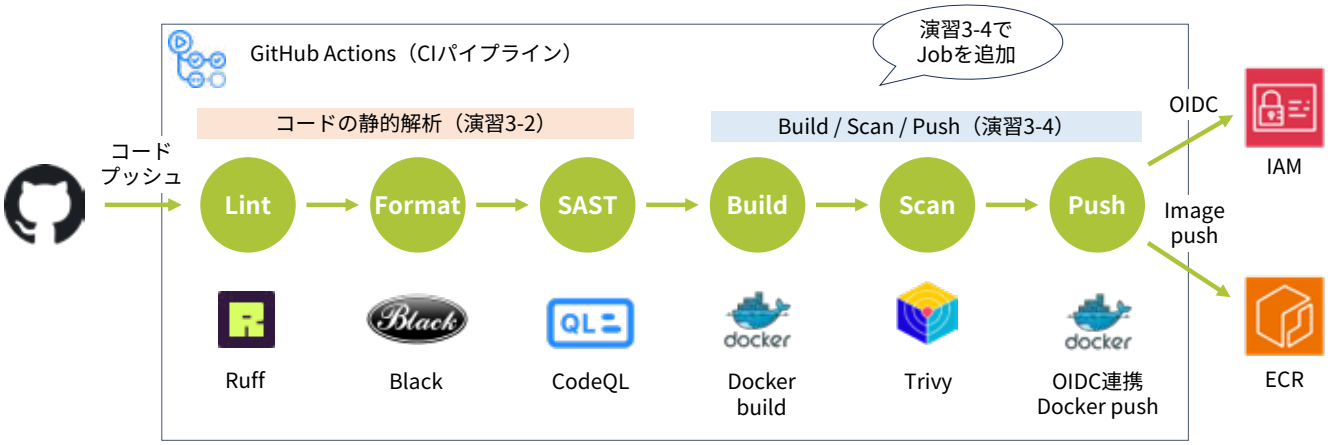
3-4：Build / Scan / Push のCI統合

問題・解答編

演習３ パイプラインへのセキュリティゲート導入

3-4：期待される挙動

GitHub Actionsのワークフローにビルド、スキャン、プッシュするジョブを追加してください。
コード解析からECRへのイメージプッシュまでを統合し、1本のワークフローで実現します。



※本演習はCIパイプライン構築を取り扱います。お時間のある方は試験やCDのパイプライン構築に挑戦してみてください。

演習３ パイプラインへのセキュリティゲート導入

3-4： Build / Scan / Push のCI統合

GitHub Actionsのワークフローにビルド、スキャン、プッシュするジョブを追加してください。
コード解析からECRへのイメージプッシュまでを統合し、1本のワークフローで実現します。

演習3-4手順

1. **Build_Scan_Pushジョブの追加**
   1つジョブを追加し、ビルド、スキャン、プッシュまで実行するようにしてください。
   ⚠ Build / Scan / Push は1つのジョブにまとめます

2. **動作確認**
   ワークフローの実行結果を確認し、ECRにイメージが登録されていることを確認してください。

```
sample-app/
├── .github/
│   └── workflows/
│       └── secure-pipeline.yaml
├── app.py
├── requirements.txt
├── Dockerfile
└── .gitignore
```

sample-app
リポジトリ構成例
※赤字ファイルを修正

演習３ パイプラインへのセキュリティゲート導入

3-4：Build_Scan_Pushジョブの追加

ワークフローにBuild_Scan_Push ジョブを追加してください。
3つの処理を1ジョブにまとめて処理させる形とします。

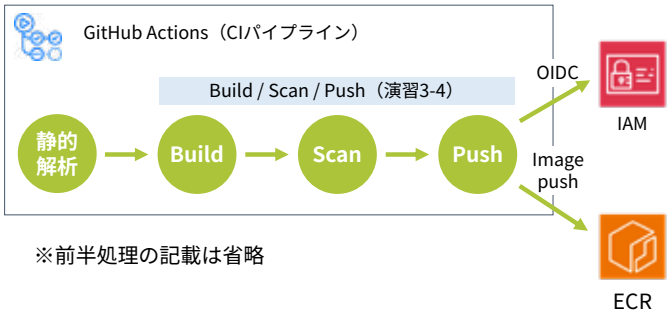
Build_Scan_Push ジョブ要件

Build処理
- sample-appのDockerfileを利用し、docker buildでコンテナイメージを作成
- タグはsample-app:latestを付与

Scan処理
- 作成したイメージをTrivyでスキャン
- 重大な脆弱性（Critial、High）がある場合、ワークフローを停止

Push処理
- AWSにOIDC連携し、GitHubからAWSへの接続を確立
- ECRにログイン
- イメージにタグをつけ、ECR/sample-appリポジトリにイメージを登録

ワークフロー構築



※前半処理の記載は省略

演習３ パイプラインへのセキュリティゲート導入

3-4：Build / Scan処理　サンプルコード

Build / Scan処理のサンプルコードを提示します。
Docker buildしたイメージをTrivyでスキャンします。

```
jobs:
  build_scan_push:
    name: Build, Scan and Push
    runs-on: ubuntu-latest
    needs: sast  # sast成功後に実行
    if: github.ref == 'refs/heads/main'

    steps:
      - uses: actions/checkout@v4

      # Build Docker Image
      - name: Build Docker image
        run: |
          docker build -t
${{ env.ECR_REPOSITORY }}:${{ env.IMAGE_TAG }} .
```

```
      # Trivy Image Scan
      - name: Trivy image scan
        uses: aquasecurity/trivy-action@master
        with:
          image-ref:
${{ env.ECR_REPOSITORY }}:${{ env.IMAGE_TAG }}
          severity: CRITICAL,HIGH
          format: table
          exit-code: 1   # 重大な脆弱性がある場合は実行中止
```

Build / Scan処理　サンプルコード

演習３ パイプラインへのセキュリティゲート導入

3-4：Push処理　サンプルコード

Push処理のサンプルコードを提示します。
AWSとOIDC連携を確立し、ECRにログインしてからECRにイメージをプッシュします。

```
    # AWS OIDC
    - name: Configure AWS Credentials
      uses: aws-actions/configure-aws-credentials@v4
      with:
        role-to-assume:
arn:aws:iam::<ACCOUNT_ID>:role/github-actions-devsecops-
role   # 作成したIAMロールを指定
        aws-region: ap-northeast-1

    # ECR login
    - name: Login to ECR
      id: ecr-login
      uses: aws-actions/amazon-ecr-login@v2
      with:
        region: ${{ env.AWS_REGION }}
```

```
    # Push image to ECR
    - name: Push Docker Image to ECR
      run: |
        docker tag
${{ env.ECR_REPOSITORY }}:${{ env.IMAGE_TAG }} ¥
          ${{ steps.ecr-
login.outputs.registry }}/${{ env.ECR_REPOSITORY }}:${{ env.IM
AGE_TAG }}

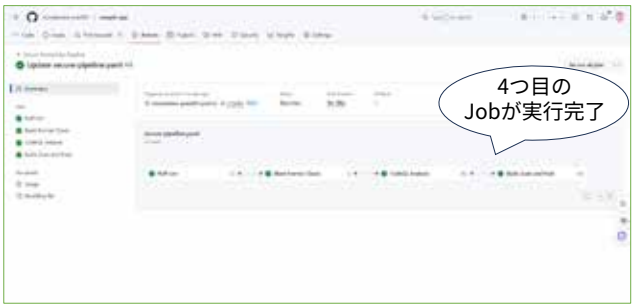
        docker push ${{ steps.ecr-
login.outputs.registry }}/${{ env.ECR_REPOSITORY }}:${{ env.IM
AGE_TAG }}
```

Push処理　サンプルコード

演習３ パイプラインへのセキュリティゲート導入

3-4：動作確認（Build_Scan_Pushジョブ）

ワークフローにBuild_Scan_Pushジョブを追加後に、GitHubで動作確認してください。
全て正常に実行完了し、Trivyのイメージスキャンで重大な脆弱性が出てないことを確認してください。





Build, Scan and Pushの詳細
（Trivy image scanの実行結果）

演習３ パイプラインへのセキュリティゲート導入

3-4：動作確認（ECR リポジトリ）

AWS ECR / sample-appリポジトリに、latestタグのイメージが登録されていれば、確認は完了です。
これでコード解析からECRプッシュまで自動化するCIパイプラインを構築できました。



AWS ECR / sample-appリポジトリ

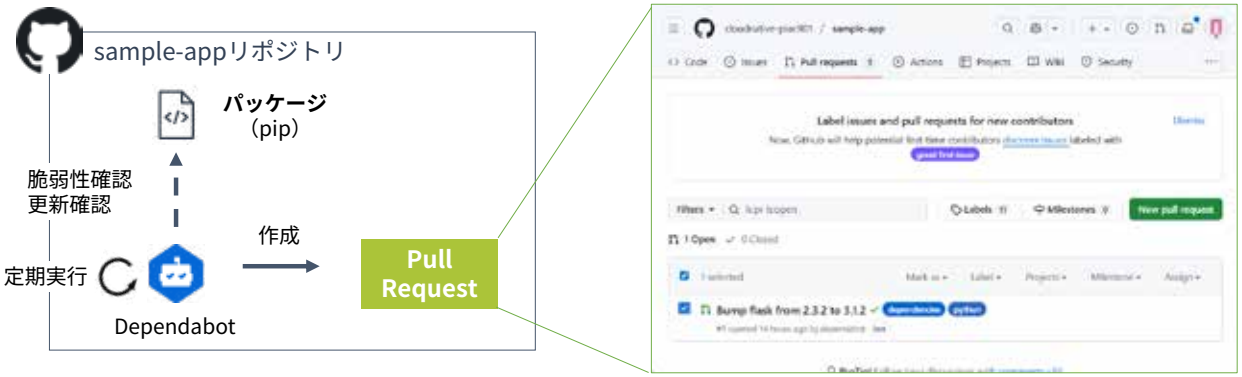
3-5：Dependabotによるパッケージ脆弱性検出

問題・解答編

演習３ パイプラインへのセキュリティゲート導入

3-5：期待される挙動

GitHubにDependabotを設定し、定期的にパッケージの脆弱性と更新を検知するようにしてください。
また検知した際は、Pull Requestを自動作成するようにしてください。



自動作成されたPull Request

演習３ パイプラインへのセキュリティゲート導入

3-5： Dependabotによるパッケージ脆弱性検出

GitHubの Actionsのワークフローを準備し、Pythonコードの静的解析を定義してください。
1つのワークフローに Lint / Format / SAST のジョブを順番に定義して、静的解析を自動化します。

演習3-5手順

1. **GitHub Advanced Security設定**
   GitHub側でDependabotをアクティベート

2. **DependabotのYAML準備**
   .github/dependabot.yaml を作成
   ⚠.github直下が配置先です（workflows直下ではありません）

3. **動作確認**
   Dependabotの実行結果を確認

```
sample-app/
├── .github/
│   ├── dependabot.yaml
│   └── workflows/
│       └── secure-pipeline.yaml
├── app.py
├── requirements.txt
├── Dockerfile
└── .gitignore
```

sample-app
リポジトリ構成例
※赤字ファイルを追加

演習３ パイプラインへのセキュリティゲート導入

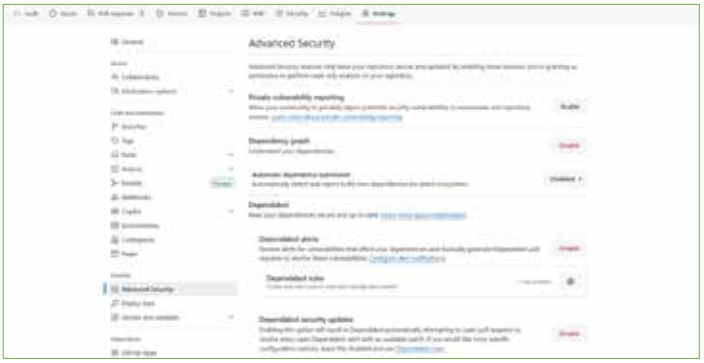
3-5：GitHub Advanced Security設定

GitHubのAdvanced SecurityからDependabot関連のセキュリティを有効にしてください。

有効にする処理

Settingタブ > Advanced Security
以下を有効にします
（**Disable表示**になればOK）

- Dependency graph
- Dependabot alerts
- Dependabot security updates



Disable表示になること

演習３ パイプラインへのセキュリティゲート導入

3-5： Dependabot　サンプルコード

Dependabot（.github/dependabot.yaml）のサンプルコードを提示します。

Dependabot要件

Pythonパッケージを定期チェックさせてください。
その他エコシステムを定義しても良いです。

- 確認対象エコシステム：pip
- チェック頻度：日次
- PR上限：5

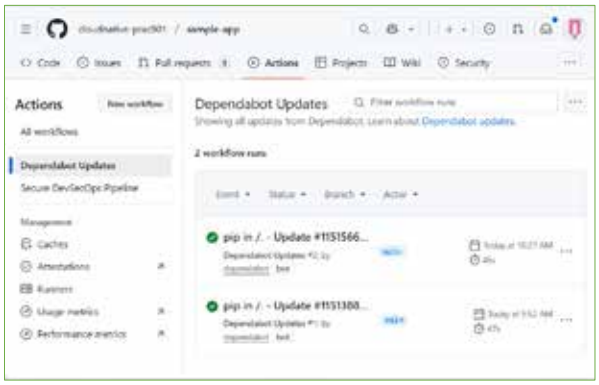
```
version: 2
updates:
  - package-ecosystem: "pip"
    directory: "/"
    schedule:
      interval: "daily"
      timezone: "Asia/Tokyo"
      time: "10:00"
    open-pull-requests-limit: 5
```

.github/dependabot.yaml

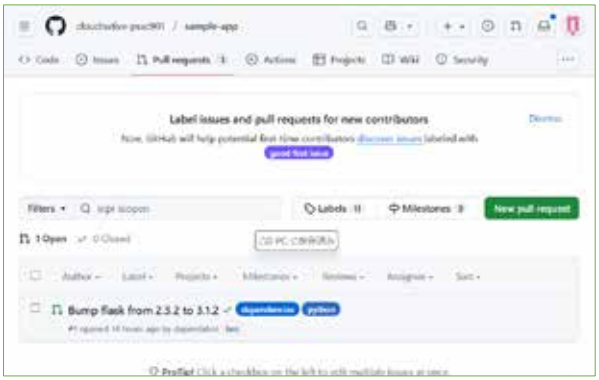
演習３ パイプラインへのセキュリティゲート導入

3-5：Dependabotの動作確認

.githubにdependabot.yamlを配置したら、実行ログを確認しましょう。パッケージの脆弱性や更新があった場合は、PRの発行も確認できます。これでパッケージの定期チェックの仕組みを用意できました。



Dependabodの実行ログ
Actionタブ > Dependabot Updates を選択



Dependabod起因のPull request

演習３ パイプラインへのセキュリティゲート導入

まとめ

演習３ パイプラインへのセキュリティゲート導入

まとめ：対策効果とベストプラクティス

GitHub Actions（CI/CDパイプライン）

CI/CDパイプラインで、品質と安全性を自動で担保し、変更を安全に本番へ届ける仕組みづくりが重要

ベストプラクティス

- テスト・Lint・セキュリティチェックを並列で実行し、Shift-Leftで早期に不具合を検知
- 環境変数・シークレットはGitHub SecretsやOIDCで安全に管理し、リポジトリに埋め込まない
- Artifact（ビルド結果・イメージ）は常にダイジェストで扱い、再現性を確保
- PRベースのワークフローでレビューと自動チェックを強制し、品質基準を満たした場合のみマージ・デプロイ

Ruff + Black（コード品質の標準化）

スタイルの標準化と潜在的なバグパターンや未使用変数を早期発見し、脆弱性混入リスクを大幅に低減

ベストプラクティス

- 適切なルールセットを定義し、自動フォーマット適用により、レビュー効率化と可読性を向上
- CI/CDパイプラインに組み込み、開発フェーズ内で早期に修正
- Ruff + BlackはPythonのコード解析ツールのため、言語や規模等に合わせて最適なツールを選択してください

演習３ パイプラインへのセキュリティゲート導入

まとめ：対策効果とベストプラクティス

CodeQL（セキュリティ脆弱性の検出）

コードレベルでのセキュリティ問題をShift-Leftで検出し、修正コストを最小化しながら堅牢なアプリケーションを構築

ベストプラクティス

- PR時に検証を組み込み、SQLインジェクションなど重大な脆弱性を自動検知してセキュリティ問題の混入を防止
- security-events: writeの権限を付与し、検出結果をGitHubのSecurityタブに蓄積
- 言語固有のクエリスイートを適切に選択・カスタマイズ

Dependabot（依存関係の脆弱性管理）

依存パッケージ管理は重要なセキュリティ対策。自動監視・更新による継続的保護が必須

ベストプラクティス

- 既知の脆弱性を含む依存ライブラリを自動検出し、アラートや脆弱性修正パッチPRを自動生成
- パッケージのエコシステムに複数の対象（npm、pipなど）を指定
- 影響の大きな更新はバージョンアップ計画を立てて段階的に実施

演習３ パイプラインへのセキュリティゲート導入

合格判定基準

演習３ パイプラインへのセキュリティゲート導入

合格判定基準

演習3の合格判定基準は以下の通りです。

3-1：事前準備（GitHub）

- GitHubにsample-appリポジトリが作成され、Python アプリケーションコードが push されていること
- CIに必要なディレクトリ構造（app.py、Dockerfile、requirements.txt）が正しく配置されていること

3-2：Ruff / Black / CodeQL によるコードの静的解析

- GitHub Actions のワークフローに Ruff / Black / CodeQL が正しく定義され、ジョブが成功していること
- Ruff と Black の実行結果を確認できていること
- CodeQLの静的解析が実行され、Code scanning alerts に結果が反映されていること

3-3：事前準備②（ECR、IAM。OIDC）

- ECRにsample-appリポジトリが作成されていること
- GitHub Actions 用のIAMロールが作成され、適切なIAMポリシーが適用されていること

演習３ パイプラインへのセキュリティゲート導入

合格判定基準

演習3の合格判定基準は以下の通りです。

3-4：Build / Scan / Push のCI統合

- GitHub Actions で Docker イメージの Build が成功し、Trivy による脆弱性スキャンが実行されていること
- （任意）Trivy のセキュリティゲート（CRITICAL/HIGH で exit 1）が動作し、不適切なイメージをブロックされること
- GitHub ActionsからAWSへのOIDC認証が完了していること
- GitHub Actionsで ECR へのログインが成功していること
- ワークフローが正常終了し、 Docker イメージが ECR に Push され、ECR 上に最新イメージが登録されていること

3-5：Dependabotによるパッケージ脆弱性検出

- dependabot.yml を作成・配置し、スケジュールに基づいて Dependabot が実行されていること
- Dependabot が PR を自動生成すること（脆弱性や更新がない場合は発行されません）
- （任意）Dependabot が作成した PR の内容（CVE 情報、更新理由）を理解し、マージ or 必要な判断ができること

演習４　サービス間通信の暗号化とシークレット管理

問題編

演習４　サービス間通信の暗号化とシークレット管理

演習概要

「すべての通信・アクセスを検証する」というゼロトラストの基本原則に則ったセキュリティ対策を実施し、Kubernetes環境における暗号化・認証・認可を実践的に体得します。

Before：脆弱な通信環境

シークレット情報が平文で保存される
環境変数やConfigMapに直接格納され、取得・漏洩しやすい

サービス間が平文HTTP
通信が暗号化されておらず、盗聴・改ざんリスクがある

サービス識別なし（なりすまし可）
接続元の検証ができず、なりすましやサービス間の横展開攻撃の可能性

After：ゼロトラストな環境

Kubernetes Secret管理
シークレット情報を暗号化して保存し、必要なPodのみアクセス可能に制限

サービス間はIstioによりmTLS化
サイドカー（istio-proxy）が自動的に通信を暗号化し、盗聴・改ざんを防止

AuthorizationPolicyで通信元を厳格化
明示的に許可された通信元からのみ接続を受付

演習４　サービス間通信の暗号化とシークレット管理

演習内容

「すべての通信・アクセスを検証する」というゼロトラストの基本原則に則ったセキュリティ対策を実施し、Kubernetes環境における暗号化・認証・認可を実践的に体得します。

演習内容

4-1. Kubernetes Secret：アプリケーションのシークレット情報をKubernetes Secretで安全に管理・提供

4-2. Istioインストール：Istioをインストールし、サンプルアプリケーションをデプロイ

4-3. PeerAuthentication：istioのPeerAuthenticationを設定し、サービス間通信のmTLS認証を有効化

4-4. AuthorizationPolicy：ServiceMesh間の通信をAuthorizationPolicyで制御

取り扱うセキュリティ対策

Secret
シークレット情報管理

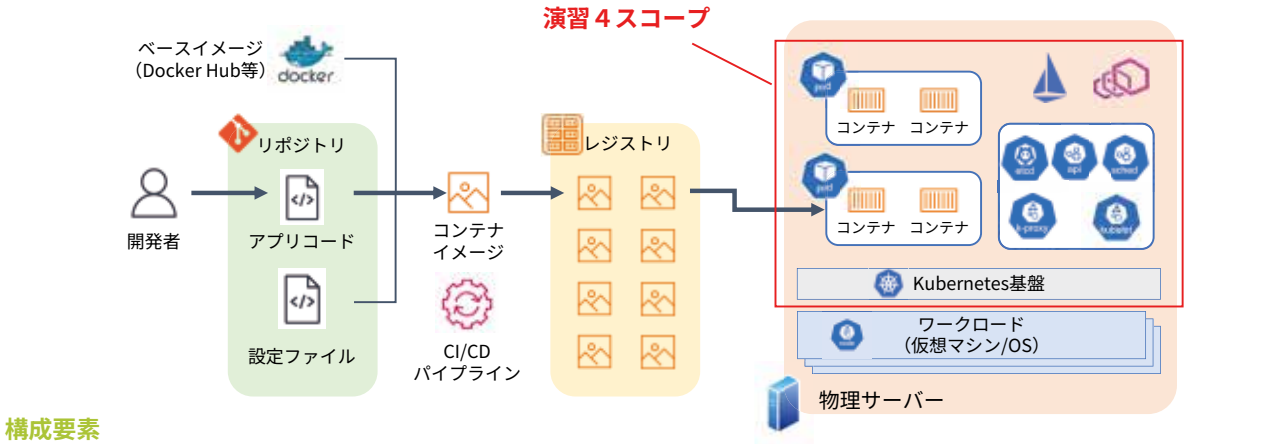
Istio / mTLS
サービス間のmTLS通信

AuthorizationPolicy
ServiceMesh内の通信制御

演習４　サービス間通信の暗号化とシークレット管理

演習４の位置づけ

演習４ではオーケストレーション、コンテナをターゲットにセキュリティ対策を行います。



演習４　サービス間通信の暗号化とシークレット管理

コンテナオーケストレーションにおける更なるセキュリティの問題

RBAC / NetworkPolicy等の対策（課題１）のみではサービス間の通信の安全性までは担保されません。
信頼できるネットワークは存在しないという前提で、環境を見直す必要があります。

⚠ 想定されるセキュリティリスク

シークレット情報の平文管理

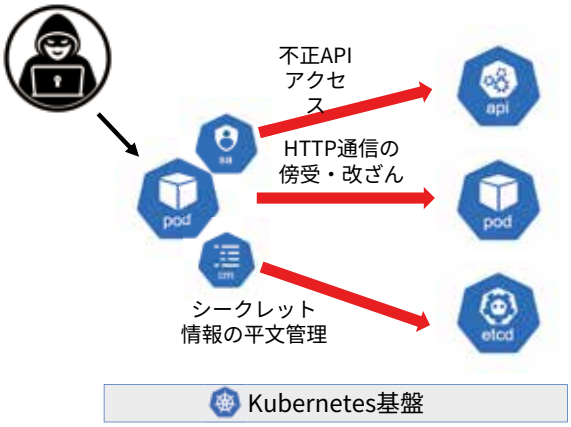
Kubernetesのシークレット情報を平文で管理していると
シークレット情報へのアクセスが容易で情報漏洩に繋がる

不正APIアクセス

Podから正規のServiceAccountトークンが可能なため、
トークン奪取によるなりすましが可能

傍受・改ざん

Pod間通信がHTTP（平文）の場合、通信経路上で
傍受・改ざんが可能



演習４　サービス間通信の暗号化とシークレット管理

コンテナオーケストレーションにおける更なるセキュリティ対策

従来の「ネットワーク内部は安全」という境界型防御の考え方を捨て、常に認証・認可・暗号化を行い、最小権限で運用するゼロトラストなセキュリティ対策を導入していきます。

想定されるセキュリティリスク		実施するセキュリティ対策
シークレット情報の平文管理 Kubernetesのシークレット情報を平文で管理していると シークレット情報へのアクセスが容易で情報漏洩に繋がる	→	**Secret**によるシークレット情報管理
不正APIアクセス Podから正規のServiceAccountトークンが可能なため、 トークン奪取によるなりすましが可能	→	**Istio**によるサービス間の**mTLS通信**
傍受・改ざん Pod間通信がHTTP（平文）の場合、通信経路上で 傍受・改ざんが可能	→	**AuthorizationPolicy**による ServiceMesh内の通信制御

演習４　サービス間通信の暗号化とシークレット管理

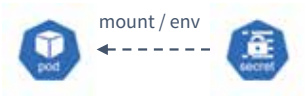
前提知識：Secret管理

アプリケーションで扱うAPIキー、パスワードなどのシークレット情報を含みます。シークレット情報は分離して管理・制御することで、システム全体のセキュリティを強化することができます。

シークレット情報の課題

- **環境変数へのシークレット情報格納**
  環境変数はプロセス情報から簡単に参照可能なため、
  Pod内の全てのコンテナから閲覧できてしまう
- **ConfigMapへのシークレット保存**
  ConfigMapは暗号化されずに保存され、
  クラスター内で簡単に参照できるため機密情報の保管に不向き
- **マニフェストへのハードコーディング**
  Gitリポジトリに直接APIキーなどが記述され、
  コード管理ツールで履歴も含めて流出するリスクがあります
- **基本的なSecretの管理課題**
  etcdに平文保存される基本的なKubernetes Secret自体も
  適切なRBACやSecret暗号化なしでは安全とは言えない

Secretの適切な利用



- **Kubernetes Secretの作成**
  シークレット情報を管理。環境変数やファイルとして
  Podにマウント可能。etcd内では暗号化。
- **RBACによるアクセス制御**
  Secret参照権限を必要なServiceAccountのみに制限し、
  不正アクセスを防止。監査ログも活用

外部Secret管理サービスの活用（推奨）

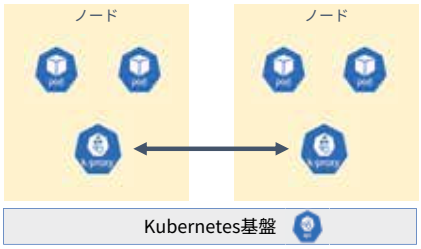
External Secrets Operatorなどを用いて、外部ツールでSecretを管理することも考えましょう。

Sealed secrets　HashiCorp Vault　AWS Secrets Manager

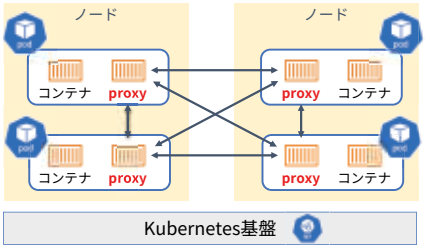
演習４　サービス間通信の暗号化とシークレット管理

前提知識：サービスメッシュ

マイクロサービス間の通信を制御・可視化します。各サービスに通信を制御するサイドカーコンテナを置くことで、アプリケーションを変更せずにトラフィック制御やセキュリティを実現します。



通常のKubernetes



サービスメッシュ

サービスメッシュの背景

- マイクロサービス化によりサービス間通信が複雑化
- 再試行・暗号化・監視などを各サービスで個別に実装する運用が困難
- サービスメッシュの導入により、それらの機能を共通化し、通信を一元的に管理することが可能に

主な特徴

- **可観測性（Observability）**：メトリクス収集、トレース、ログ分析
- **トラフィック管理**：ルーティング制御、リトライ、F/O
- **セキュリティ**：mTLS通信、認証・認可制御
- **ポリシー管理**：通信ポリシーを一元的に設定・適用

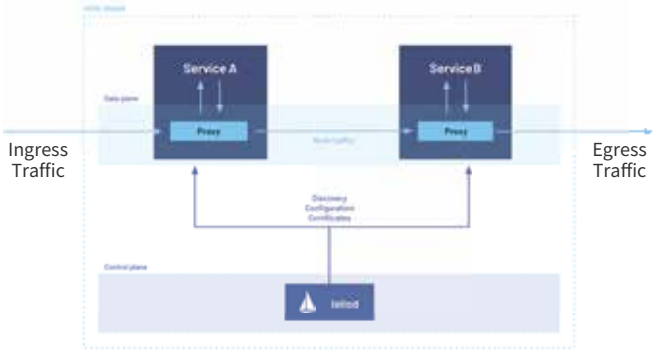
演習４　サービス間通信の暗号化とシークレット管理

前提知識：Istio

Istioは、制御プレーン（Istiod）とデータプレーン（Envoy）で構成されるサービスメッシュの代表的なツールです。トラフィック制御からセキュリティまで様々な機能を有しています。

主な特徴

- **トラフィック管理**
  （VirtualService / DestinationRule）
  カナリアリリース、Blue/Greenデプロイ、FailOver制御、外部通信制御（IngressGW / EgressGW）
- **ポリシー管理**　課題4-2
  （EnvoyFilter / Sidecar）
  通信ルール・外部連携の拡張
  ラベル付与のみでistio-proxy（Envoy）をPodに注入
- **セキュリティ**　課題4-3、4-4
  （PeerAuthentication / AuthorizationPolicy）
  mTLSによる暗号化通信、認可制御（Zero Trust）
- **可観測性**　課題5
  （Prometheus / Grafana / Kiali）
  メトリクス・トレース・可視化ダッシュボード



Istioのアーキテクチャ
https://istio.io/latest/docs/ops/deployment/architecture/

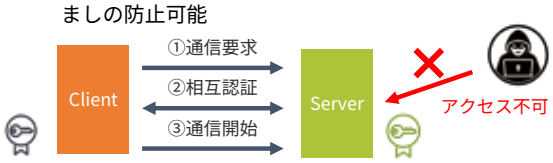
演習４　サービス間通信の暗号化とシークレット管理

前提知識：mTLSとAuthorizationPolicy（認証と認可）

Istioを利用してサービスメッシュ内の認証・認可を実装することで、ゼロトラストな環境を実現します。
mTLS：通信経路を暗号化し、通信相手を相互認証（AuthN：認証）
AuthorizationPolicy：通信相手・パス・メソッドなどに基づくアクセス制御（AuthO：認可）

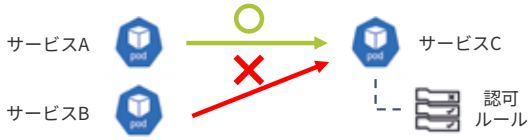
mTLS

- 通信時、クライアントとサーバーの双方が証明書を提示し、相互に認証する仕組み
- 暗号化に加え、サービス間の認証も行うため、なりすましの防止可能



AuthorizationPolicy

- 「どのサービス」が「どのサービス」にアクセス出来るかを制御する仕組み
- Ingress/Egressにも適用でき、サービスメッシュを出入りするトラフィックのアクセス制御も可能



Istioにおける設定方法

- **PeerAuthentication**のCRDで制御可能

  STRICT ：　mTLSのみ許可（推奨）
  PERMISSIVE：　mTLS / 平文も許可
  DISABLE：　mTLS無効

Istioにおける設定方法

- **AuthorizationPolicy**のCRDで制御可能

  Action ：　DENY / ALLOW
  Rules：　from / to

演習４　サービス間通信の暗号化とシークレット管理

演習４の進め方

「すべての通信・アクセスを検証する」というゼロトラストの基本原則に則ったセキュリティ対策を実施し、Kubernetes環境における暗号化・認証・認可を実践的に体得します。

4-1	対策1 **シークレット情報のSecret管理**	シークレット情報をSecretで管理し、configと分離
4-2	**Istioのインストール**	Istioおよびサンプルアプリケーションをインストールし、サービスメッシュを構築
4-3	対策2 **mTLS通信による認証**	mTLS通信の設定によりPod間通信を暗号化
4-4	対策3 **AuthorizationPolicyによる認可**	AuthorizationPolicyの設定によるアクセス制御

4-1：シークレット情報のSecret管理

問題編

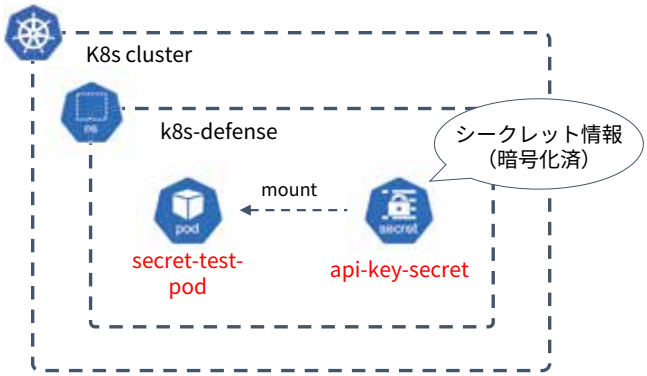
演習４　サービス間通信の暗号化とシークレット管理

4-1：シークレット情報のSecret管理

Secretリソースを作成し、Podにマウントして注入していきます。
この演習を通じて、基本的なシークレット情報の渡し方や分離管理の重要性を学習しましょう。

検証手順

1. 新規Namespace作成（未作成の場合）
   ns: k8s-defense
2. Secretの作成
   secret: api-key-secret
3. Podの作成
   pod: secure-test -pod
4. Pod / Secretの確認



構築するk8s環境

演習４　サービス間通信の暗号化とシークレット管理

4-1：Secret / Podのデプロイ

SecretとPodをデプロイしてください。
Secretリソースをマウントできることを確認してください。

作業手順

1. **Secretの作成**
   今回はコマンドラインでSecretを作成してください。
   ```
   $ kubectl create secret generic api-key-secret  -n k8s-defense ¥
     --from-literal=API_KEY=1234-SECRET-5678
   ```

2. **Podのデプロイ**
   Secretを読み込むPodを作成してください。
   Pod内の環境変数およびボリュームに作成したシークレットをマウントしてください。
   ```
   $ kubectl apply -f secret-test-pod.yaml
   ```

演習４　サービス間通信の暗号化とシークレット管理

4-1：Podのサンプルマニフェスト

Secretを取り込むPodのサンプルマニフェストを記載します。
環境変数とボリュームマウントの2つの方法で、Secretからシークレット情報をPodに取り込んでいます。

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
  namespace: k8s-defense
spec:
  containers:
  - name: app
    image: busybox
    command: ["sh", "-c", "env | grep API_KEY && cat
/mnt/secrets/API_KEY && sleep 3600"]

# 右に続く
```

```
    env:
    - name: API_KEY
      valueFrom:
        secretKeyRef:
          name: api-key-secret
          key: API_KEY
    volumeMounts:
    - name: secret-volume
      mountPath: "/mnt/secrets" # Secretをvolumeにmount
  volumes:
  - name: secret-volume
    secret:
      secretName: api-key-secret
```

secret-test-pod.yaml

演習 4　サービス間通信の暗号化とシークレット管理

4-1：Pod / Secretの設定確認

SecretとPodのデプロイが完了したら、シークレット情報の見え方を確認してください。
シークレット情報のマウント方法および分離管理について理解を深めましょう。

Pod / Secretの設定確認

1 **Podへのマウント確認**

Podにシークレット情報を環境変数およびマウントが設定できているか確認してください。

```
$ kubectl exec -n k8s-defense secret-test-pod -- env | grep API_KEY
$ kubectl exec -n k8s-defense secret-test-pod -- cat /mnt/secrets/API_KEY
```

2 **Secretの確認**

Secretのシークレット情報を確認し、暗号化されていることを確認してください。
API_KEYはbase64で暗号化されているため、復号化して値を確認してください。

```
$ kubectl get secret api-key-secret -n k8s-defense -o yaml
```

4-2：Istioインストール

問題編

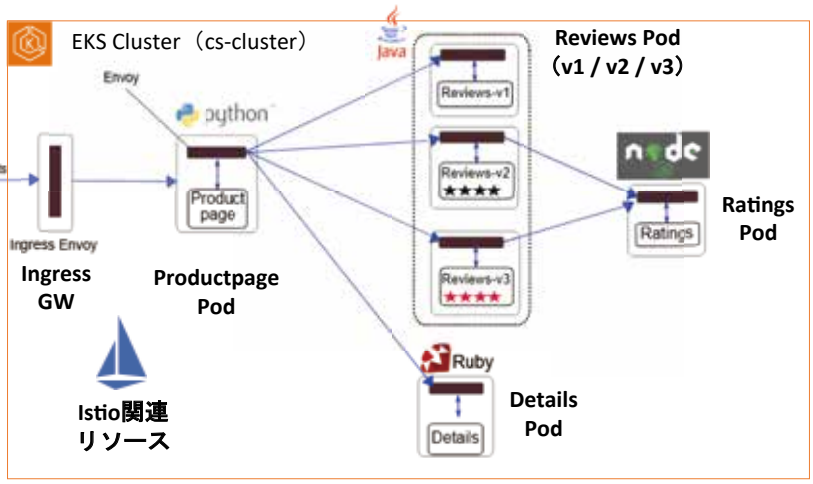
演習４　サービス間通信の暗号化とシークレット管理

4-2：期待される挙動

EKSクラスターにIstioをインストールし、Istioのサンプルアプリ（book-info）を立ち上げましょう。
その際、各Podにistio-proxy（Envoy）が注入された状態にします。



book-infoブラウザ画面



https://istio.io/latest/docs/examples/bookinfo/

演習４　サービス間通信の暗号化とシークレット管理

4-2：Istioインストール

EKSクラスターにIstioをインストールし、Istioのサンプルアプリケーション（book-info）を立ち上げましょう。その際、各Podにistio-proxy（Envoy）が注入された状態にします。

演習4-2 手順

1	**不要リソースのアンインストール**	Kyvernoなど課題4-1以前で利用したリソースを削除
2	**Istioのインストール**	EKSクラスターにIstioをインストール
3	**book-infoアプリのデプロイ**	Istioのサンプルアプリケーションをデプロイし、起動状態を確認

演習４　サービス間通信の暗号化とシークレット管理

4-2：不要リソースのアンインストール

余計なリソースが残っている場合、Istioのインストールに失敗する場合があります。
またリソース不足になる可能性があるため、Kyvernoや他リソースをアンインストールしてください。

1 Kyvernoのアンインストール

Kyvernoをアンインストールしてください。Namespaceも合わせて削除してください。

```
$ helm uninstall kyverno -n kyverno
$ kubectl delete namespace kyverno --ignore-not-found
```

2 Kyvernoの残リソース削除

KyvernoのWebhook設定（2種類）も合わせて削除してください。

```
$ kubectl get mutatingwebhookconfigurations
$ kubectl delete mutatingwebhookconfiguration <リソース名> --ignore-not-found

$ kubectl get validatingwebhookconfigurations
$ kubectl delete validatingwebhookconfiguration <リソース名>  --ignore-not-found
```

3 その他リソース削除

課題1〜4-1までのリソースがあれば、kubectl delete等で削除してください。
その他、不要リソースがあれば、合わせて削除してください。

演習４　サービス間通信の暗号化とシークレット管理

4-2：Istioのインストール

IstioをKubernetesクラスターにインストールしてください。

インストール手順

1 Istioのインストール

Istioの公式サイトから、Istioをインストールしてください。
⚠本演習ではIstio v1.27.3を利用します（ディレクトリ名もistio-1.27.3で表記します）。

```
$ curl -L https://istio.io/downloadIstio | sh -
$ cd istio-1.27.3

istio-1.27.3 $ export PATH=$PWD/bin:$PATH
istio-1.27.3 $ istioctl install --set profile=demo -y
```

2 Istioの状態確認

Istioのリソースが正常に起動していることを確認してください。

```
istio-1.27.3 $ kubectl get namespace istio-system
istio-1.27.3 $ kubectl get all -n istio-system
```

演習４　サービス間通信の暗号化とシークレット管理

4-2：book-infoアプリのデプロイ

Istioのサンプルアプリケーション（book-info）をデプロイしてください。
その際、デプロイした各Podにistio-proxyが注入されていることを確認してください。

インストール手順

1 **Namespaceの設定**

book-infoアプリケーション用のNamespace（book-info）を作成してください。
合わせて、book-info Namespaceにistio-envoy注入のラベルを付与します。

```
istio-1.27.3 $ kubectl create namespace book-info
istio-1.27.3 $ kubectl label namespace book-info istio-injection=enabled
```

2 **book-infoのアプライ**

ダウンロードしたistioのサンプルアプリケーション（book-info）をインストールしてください。

```
istio-1.27.3 $ kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml -n book-info
istio-1.27.3 $ kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml -n book-info
```

演習４　サービス間通信の暗号化とシークレット管理

4-2：book-infoの動作確認①

book-infoの動作確認をしてください。
まず各リソースの起動確認とPod間の内部通信を確認します。

動作確認手順

1 **book-info関連リソースの起動確認**

book-info関連のリソースの起動状態を確認してください。

```
istio-1.27. 3 $ kubectl get pods -n book-info -w   # 6つのPodが2/2で起動すること
istio-1.27. 3 $ kubectl get gateway -n book-info
istio-1.27. 3 $ kubectl get virtualservice -n book-info
```

2 **内部通信確認**

Ratings PodからProductpage Podにアクセスし、Pod間でHTTP通信できることを確認してください。
HTMLのタイトルが表示されればOKです。

```
istio-1.27.3 $ kubectl exec -n book-info "$(kubectl get pod -n book-info ¥
  -l app=ratings -o jsonpath='{.items[0].metadata.name}')" ¥
  -c ratings -- curl -s productpage:9080/productpage | grep -o "<title>.*</title>"
```

演習４　サービス間通信の暗号化とシークレット管理

4-2：book-infoの動作確認②

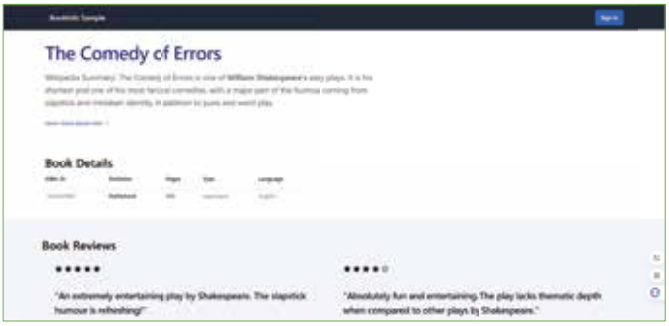
ブラウザからbook-infoにアクセスしてください。Web画面を表示できれば、4-2は完了です。

動作確認手順

3 **外部通信確認**

ブラウザーからbook-infoを確認できれば、準備完了です。

EKSの場合、自動生成されるロードバランサー、CLBの<DNS名>/productpage、からアクセスできます。



book-infoアプリ（<DNS名>/productpage）



CLB

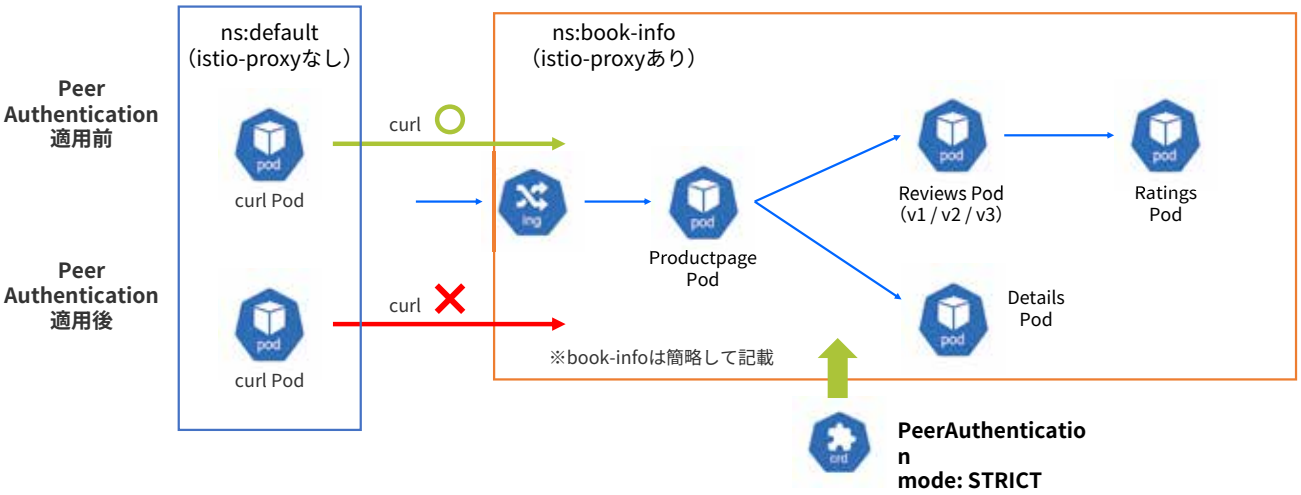
4-3：mTLS通信による認証

問題編

演習４　サービス間通信の暗号化とシークレット管理

4-3：期待される挙動

PeerAuthenticationでmTLSを有効化します。前後でサービスメッシュ外からの通信が可能か、挙動の変化を確認し、mTLSの有効性を確認しましょう。



演習４　サービス間通信の暗号化とシークレット管理

4-3：mTLS有効化の適用と動作検証

PeerAuthenticationでmTLSを有効化します。前後でサービスメッシュ外からの通信が可能か、挙動の変化を確認し、mTLSの有効性を確認しましょう。

動作確認手順

1　curlを実行可能なPodの起動・動作検証

book-info以外のNamespaceに新しくcurl Podを起動し、
book-infoのPodにアクセス可能なこと（200OK）を確認してください。

```
$ kubectl run tmp --rm -it --image=curlimages/curl:8.6.0 -- sh
$ curl -v http://details.book-info:9080/details/0
```

2　PeerAuthentication / mTLSの適用

マニフェストを作成し、PeerAuthenticationを設定します。

```
$ nano peer-auth.yaml
$ kubectl apply -f  peer-auth.yaml
```

```yaml
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: book-info
  namespace: book-info
spec:
  mtls:
    # mTLSを強制（平文通信を拒否）
    mode: STRICT
```

peer-auth.yaml

3　mTLS : STRICT化後　動作検証

curl Podからbook-infoのPodにアクセスできないこと（connection reset by peer）を確認してください。

```
$ kubectl run tmp --rm -it --image=curlimages/curl:8.6.0 -- sh
$ curl -v http://details.book-info:9080/details/0
```

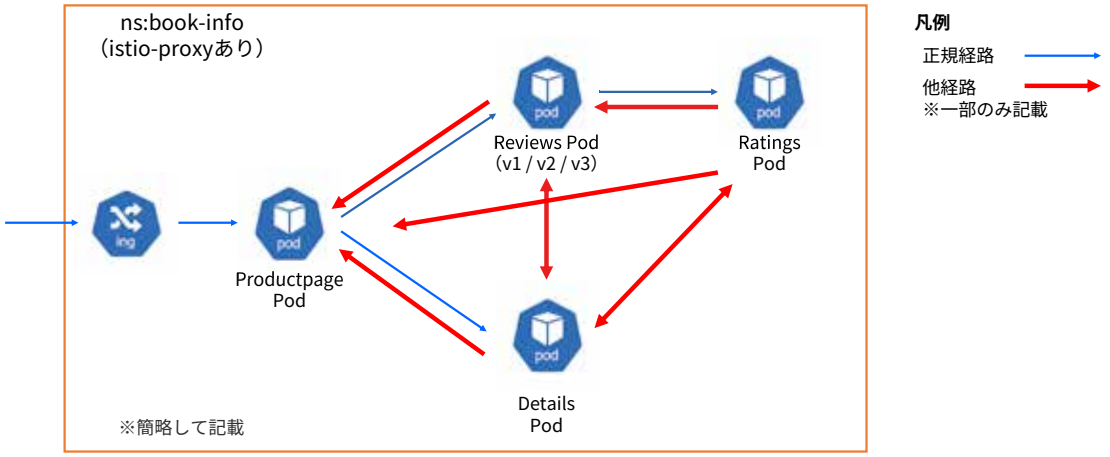
4-4：AuthorizationPolicyによる認可

問題編

演習４　サービス間通信の暗号化とシークレット管理

4-4：期待される挙動

AuthorizationPolicyに正規の通信経路を設定します。
設定後は、それ以外の経路からの通信が拒否されることを確認します。



演習４　サービス間通信の暗号化とシークレット管理

4-4：動作検証

AuthorizationPolicy適用前に、正規経路および他経路でのPod間通信を確認しておきましょう。
適用前のため、どのPod間通信も制限されていないはずです。

動作検証手順

1 **book-info Pod間の通信確認**

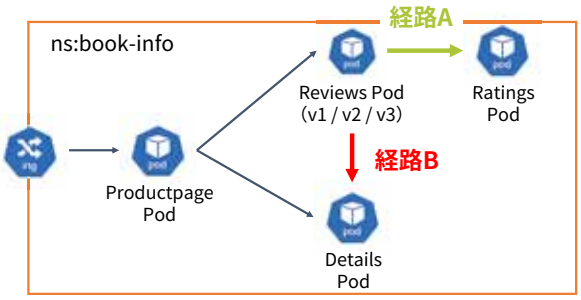
サンプルで、Reviews v2 Pod →Ratings / Details に向けてのPod間通信を確認します。
⚠ これ以外の経路でのPod間通信を確認しても構いません。

経路A（正規）：reviews v2 → ratings 通信（HTTP 200 OK）

```
$ kubectl exec -n book-info $(kubectl get pod -n book-info ¥
 -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}') ¥
 -c reviews -- curl -s -o /dev/null -w ¥
 "HTTP %{http_code}¥n" http://ratings:9080/ratings/0
```

経路B（他）：reviews v2 → details 通信（HTTP 200 OK）

```
$ kubectl exec -n book-info $(kubectl get pod -n book-info ¥
 -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}') ¥
 -c reviews -- curl -s -o /dev/null ¥
 -w "HTTP %{http_code}¥n" http://details:9080/details/0
```



演習４　サービス間通信の暗号化とシークレット管理

4-4：AuthorizationPolicyの設計

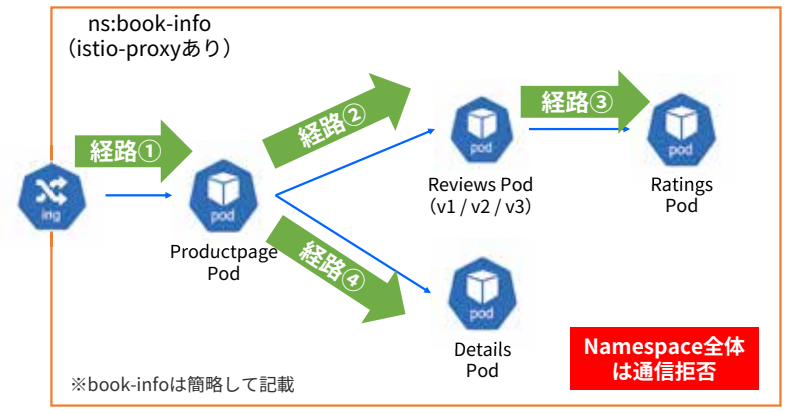
正規の通信経路を整理して、 AuthorizationPolicy（CRD）の設定に落としこみましょう。
Namespace内の通信を全て拒否し、必要な通信経路を許可していきます。

通信経路

以下をAuthorizationPolicyに設定

拒否
- Namespace全体を拒否（Deny All）

許可
- 経路①：Ingress→Productpage Pod
  ※Ingressからの経路も許可が必要
- 経路②：Productpage Pod → Reviews Pod
- 経路③：Reviews Pod → Ratings Pod
- 経路④：Productpage Pod → Details Pod



演習４　サービス間通信の暗号化とシークレット管理

4-4：AuthorizationPolicyのサンプルマニフェスト

AuthorizationPolicyのサンプルマニフェストを記載します。
サンプルをベースにマニフェストを用意してください。

```
# Deny all request
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-all
  namespace: book-info
spec: {}
```

Namespace全体：Deny All

```
# Allow-reviews-to-ratings
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-reviews-to-ratings
  namespace: book-info
spec:
  selector:
    matchLabels:
      app: ratings  # 適用対象：ratings
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/book-info/sa/bookinfo-reviews"]
```

経路③：Reviews Pod → Ratings Pod

演習４　サービス間通信の暗号化とシークレット管理

4-4：AuthorizationPolicyの設定と動作検証

AuthorizationPolicyのマニフェストを準備して適用してください。
適用後に再度、Pod間通信を確認して、挙動の違いを確認してください。

検証手順

1. **AuthorizationPolicyのマニフェスト準備**

   通信経路をマニフェストに落とし込んで適用してください。
   ⚠ マニフェストの分割単位は問いません。

2. **内部通信確認**

   Reviews v2 Pod →Ratings / Details に向けてのPod間通信を確認し、挙動の違いを確認してください。

   経路A（正規）：reviews v2 → ratings 通信（HTTP 200 OK）

```
$ kubectl exec -n book-info $(kubectl get pod -n book-info ¥
  -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}') ¥
  -c reviews -- curl -s -o /dev/null -w  "HTTP %{http_code}¥n" http://ratings:9080/ratings/0
```

   経路B（他）：reviews v2 → details 通信（HTTP 403 NG）

```
$ kubectl exec -n book-info $(kubectl get pod -n book-info ¥
  -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}') ¥
  -c reviews -- curl -s -o /dev/null  -w "HTTP %{http_code}¥n" http://details:9080/details/0
```

演習４　サービス間通信の暗号化とシークレット管理

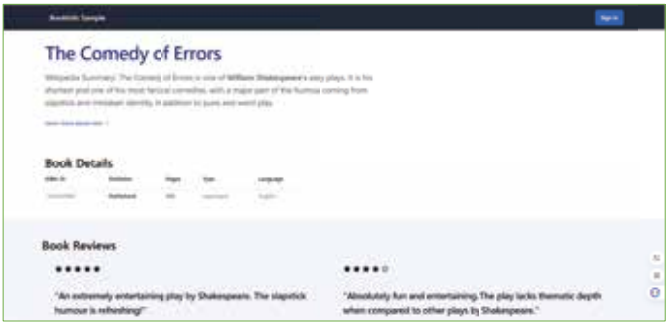
4-4：AuthorizationPolicyの設定と動作検証②

ブラウザからbook-infoにアクセスしてください。変わらずWeb画面が表示されれば、4-4は完了です。

検証手順

3 **外部通信確認**

再度ブラウザからbook-infoを確認してみましょう。
mTLS、Authorizationを適用しても、ブラウザから確認できていればOKです。



book-infoアプリ（<DNS名>/productpage）



CLB

演習４　サービス間通信の暗号化とシークレット管理

合格判定基準

演習４　サービス間通信の暗号化とシークレット管理

合格判定基準

演習4の合格判定基準は以下の通りです。

4-1：シークレット情報のSecret管理

- Kubernetes Secretが正しく作成されていること
- Podが正常に起動し、環境変数およびボリュームに Secret がマウントされていること
- Secret情報がBase64で暗号化されていること（復号した値がキー情報と一致すること）

4-2：Istioインストール

- Istioのインストールが完了していること
- サンプルアプリケーションbook-infoのPodが起動していること（istio-proxyが2/2 READYで注入されていること）
- ブラウザから、book-infoのアプリケーションを確認できること

4-3：mTLS通信による認証

- PeerAuthenticationの設定にmode: STRICTが適用されていること
- PeerAuthentication適用（mTLS有効）時、外部NamespaceのPodからアクセスが拒否されること
- PeerAuthenticationの適用時でも同一Namespaceの通信が許可されていること

4-4：AuthorizationPolicyによる認可

- AuthorizationPolicyが適用されていること（Namespace全体：拒否、正規経路：許可）
- 正規経路のみ通信可能で、他経路は拒否されていること

演習５　Kialiによる可視化と通信分析

問題編

演習５　Kialiによる可視化と通信分析

演習概要

サービスメッシュ環境（Istio＋book-infoアプリケーション）の可観測性（Observability）を確立し、異常の早期検知と迅速な原因特定により安定稼働を実現するための手法を学習します。

Before：可観測性が不十分な環境

マイクロサービス間通信がブラックボックス化
サービス間のトポロジーと通信の把握が困難で、依存関係が不透明

エラー検知と原因特定の遅延
障害発生から対応までの時間が長く、影響範囲の把握が困難

セキュリティインシデントの検知困難
異常な通信パターンの発見が遅れ、対応が後手に回る

After：可観測性を確立した環境

Kialiによるメッシュ可視化
サービス間の依存関係と通信状態をリアルタイム表示

PrometheusとGrafanaによる監視
メトリクス収集とダッシュボード化

エラー率と遅延の分析
異常の早期検知と根本原因の迅速な特定が可能に

演習５　Kialiによる可視化と通信分析

演習内容

サービスメッシュ環境（Istio＋book-infoアプリケーション）の可観測性（Observability）を確立し、異常の早期検知と迅速な原因特定により安定稼働を実現するための手法を学習します。



利用するツール

Prometheus
メトリクス収集

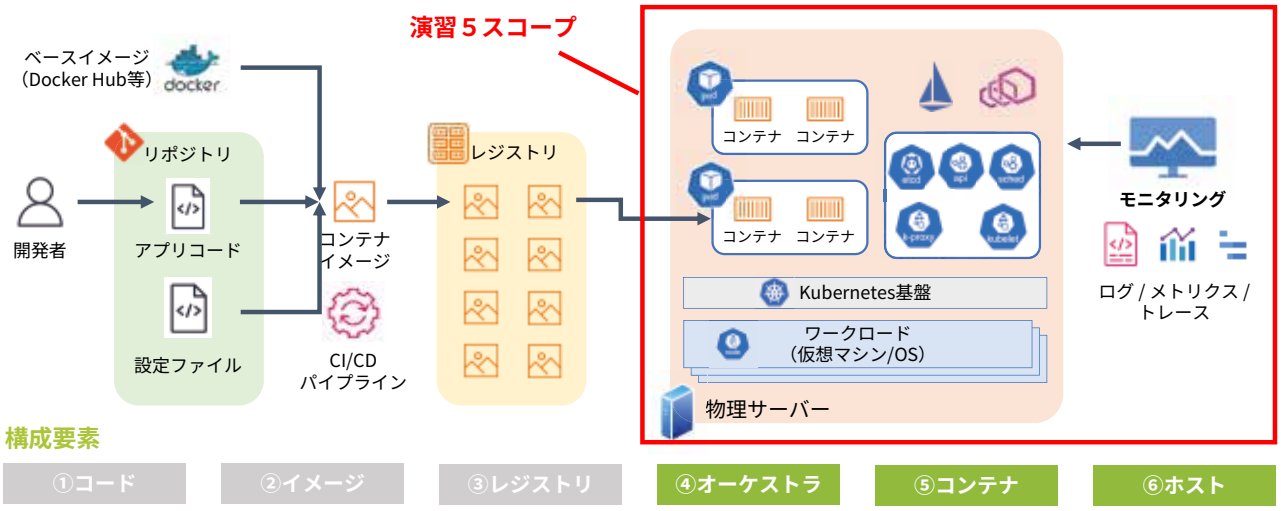
Grafana
メトリクス可視化・分析

Kiali
サービストポロジー可視化

演習５　Kialiによる可視化と通信分析

演習５の位置づけ

演習５ではオーケストレーション、コンテナ、ホストをターゲットに可観測性を実現します。



演習５　Kialiによる可視化と通信分析

可観測性の課題

サービスメッシュ環境でのトラブルは、考慮すべき要素が多く、原因特定に時間を要し、サービス影響が大きくなる可能性があります。課題4で取り組んだセキュリティ対策だけでは不十分です。

⚠ 可観測性の課題

複雑なサービス間通信

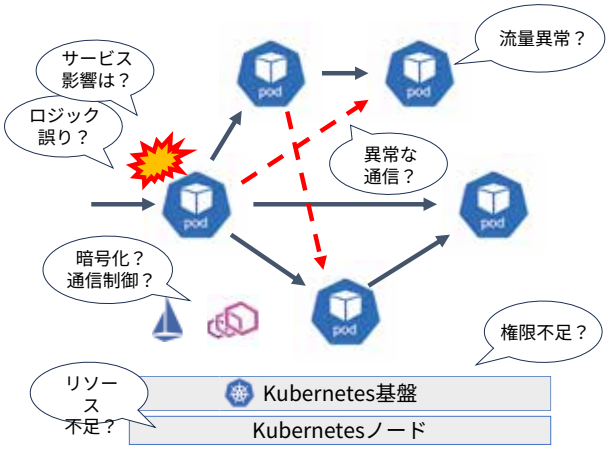
マイクロサービス化により経路・依存関係が不明瞭になり
全体把握が困難

異常検知の遅延、分析の難しさ

通信エラーや処理遅延に伴う、検知・対応が遅れる
障害や性能劣化の原因特定に時間を要する

SLOやセキュリティ監視の不足

SLOなどが未管理で運用品質が不安定になり、異常なトラフィックなどのインシデント兆候の把握が困難



Pod障害時に考えられる要因

演習５　Kialiによる可視化と通信分析

サービスメッシュ環境における可観測性

サービスメッシュ環境の可観測性を実現することで、サービスの可視化だけでなく、異常検知と原因究明を迅速化することが可能となります。

⚠ 可観測性の課題

複雑なサービス間通信

マイクロサービス化により経路・依存関係が不明瞭になり全体把握が困難

異常検知の遅延、分析の難しさ

通信エラーや処理遅延に伴う、検知・対応が遅れる
障害や性能劣化の原因特定に時間を要する

SLOやセキュリティ監視の不足

SLOなどが未管理で運用品質が不安定になり、異常なトラフィックなどのインシデント兆候の把握が困難

✔ 実施するセキュリティ対策

可観測性（Observability）による監視

- ログ
- メトリクス
- トレーシング

演習５　Kialiによる可視化と通信分析

前提知識：可観測性（Observability）

可観測性とはシステムの内部状態を外部から理解できる状態にすることです。従来の閾値やログによる監視から、システム全体を把握して予兆検知、障害対応、性能最適化を可能にします。

Metrics（メトリクス）

- 数値化された計測値（CPU使用率、レイテンシ等）
- 時系列データベースに格納され、集計・可視化

メトリクス種類

インフラ系
CPU、メモリ、ディスクIO
AP系
リクエスト数、エラー率、レイテンシ
Kubernetes/EKS
Pod CPU、メモリ、HPAスケールメトリクス

Logs（ログ）

- イベント発生時の詳細な記録
- エラー内容、リクエスト情報、スタックトレース

ログ種類

- アプリログ
- Kubernetesログ
- セキュリティログ
- インフラログ

Traces（トレース）

- マイクロサービス間の1リクエストの流れを追跡
- 分散システムにおける処理のボトルネック特定

構成

トレース
1つのリクエストの全体像

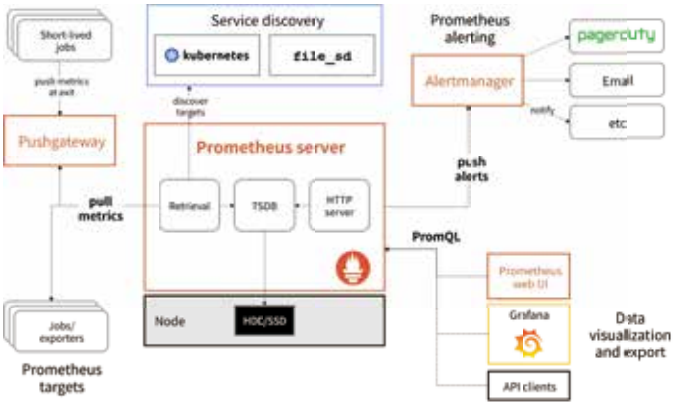
スパン
各サービスでの処理のまとまり
開始〜終了時間、タグ、ログを持つ

「メトリクス」×「ログ」×「トレース」で複合的にシステムを分析して原因を特定します

演習５　Kialiによる可視化と通信分析

前提知識：Prometheus（メトリクス収集）

Prometheusは、アプリやインフラの状態（CPU使用率・メモリ・リクエスト数など）を自動で収集し、監視や可視化に利用できるオープンソースのモニタリングツールです。



https://prometheus.io/docs/introduction/overview/

Prometheusの特徴

オープンソースの監視ツール
コンテナ環境の監視に最適なOSS

時系列データベース
メトリクスの収集・保存・クエリに特化した高性能データベースを内蔵

Pullモデル
ターゲットからメトリクスを収集
エクスポーターの種類が豊富

サービスディスカバリ
動的環境でも自動的にモニタリングターゲットを検出

強力なクエリ言語
PromQLによる柔軟なメトリクスの検索・集計・分析

演習５　Kialiによる可視化と通信分析

前提知識：Grafana（メトリクス可視化）

Grafanaは可視化と監視のためのプラットフォームです。サーバーやコンテナ、アプリケーションなどから収集したメトリクスを、グラフや表などで表現し、アラート通知します。Prometheusと連動して動かすことが多いです。



Grafana - Node Exporterダッシュボード
https://grafana.com/grafana/dashboards/

Grafanaの特徴

リアルタイム可視化
CPUやメモリなどのリソース使用状況をリアルタイムで表示

カスタムダッシュボード
必要なメトリクスのみをグループ化して一覧表示可能。正常/警告/危険状態を視覚的に色分けすることも可能。

インタラクティブなグラフ
ズームイン・アウトや時間範囲変更が可能

多様なグラフタイプ
折れ線グラフ、ゲージ、ヒートマップ、テーブルなど

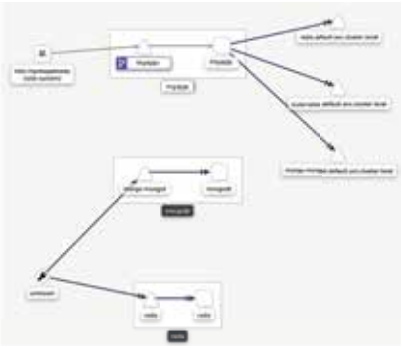
演習５　Kialiによる可視化と通信分析

前提知識：Kiali（サービストポロジー可視化）

Kialiはサービスメッシュ可視化ツールです。サービスメッシュ内の通信、遅延、エラー率をリアルタイムにグラフ表示します。mTLS状態・トレース・サービス依存関係を自動マップ化することも可能です。



Kiali アーキテクチャ
https://kiali.io/docs/architecture/architecture/



Kiali トラフィックグラフ
https://kiali.io/docs/faq/graph/

Kialiの特徴

サービス間通信のリアルタイム可視化
マイクロサービス同士の通信経路やリクエスト数、エラー率、レイテンシをリアルタイムで表示

サービスマップ（依存関係の自動生成）
サービス間のつながりをトラフィックグラフとして表示。サービス間の通信方向を可視化

mTLS状態の可視化
サービス間の通信が mTLS で保護されているかをアイコンで表示

トラフィック分析（詳細レベル）
成功/失敗リクエストの割合、p90/p99レイテンシ、RPSなどを細かく表示。ボトルネック分析や性能チューニングに役立つ

演習５　Kialiによる可視化と通信分析

演習５の進め方

マイクロサービス環境（Istio + book-infoアプリケーション）の可観測性（Observability）を確立し、異常の早期検知と迅速な原因特定により安定稼働を実現するための手法を学習します。

5-1	**Prometheus / Grafana / Kialiのインストール**	Helm経由でPrometheus / Grafana / Kialiをインストール
5-2	**Kialiトラフィックグラフによる可視化**	Kialiのトラフィックグラフで、book-infoアプリケーションのトポロジーを可視化
5-3	**Kialiを用いた通信分析**	複数パターンでPod間通信を発生させ、Kialiによる通信状況やエラーを分析
5-4	**Grafanaダッシュボード作成**	Grafanaのダッシュボードを作成し、Kubernetes環境の状況をメトリクスで可視化

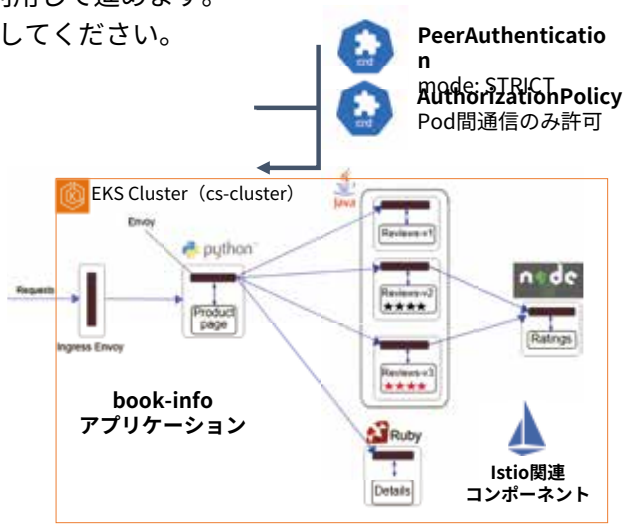
演習５　Kialiによる可視化と通信分析

演習前提

演習5は、EKS、Istio、book-infoアプリケーションを利用して進めます。
EKSクラスターを作成の上、演習4実施後の環境を用意してください。

EKSクラスター（演習4-4実施後）

- **Istio**
  istio-system Namespaceにインストール済
- **book-infoアプリケーション**
  book-info Namespaceに、
  Istioのサンプルアプリ「book-info」をデプロイ済
- **PeerAuthentication**
  mode: STRICTを適用済
- **AuthorizationPolicy**
  以下ルールを適用済
  - Namespace全体の通信は拒否
  - book-infoのPod間通信のみ許可



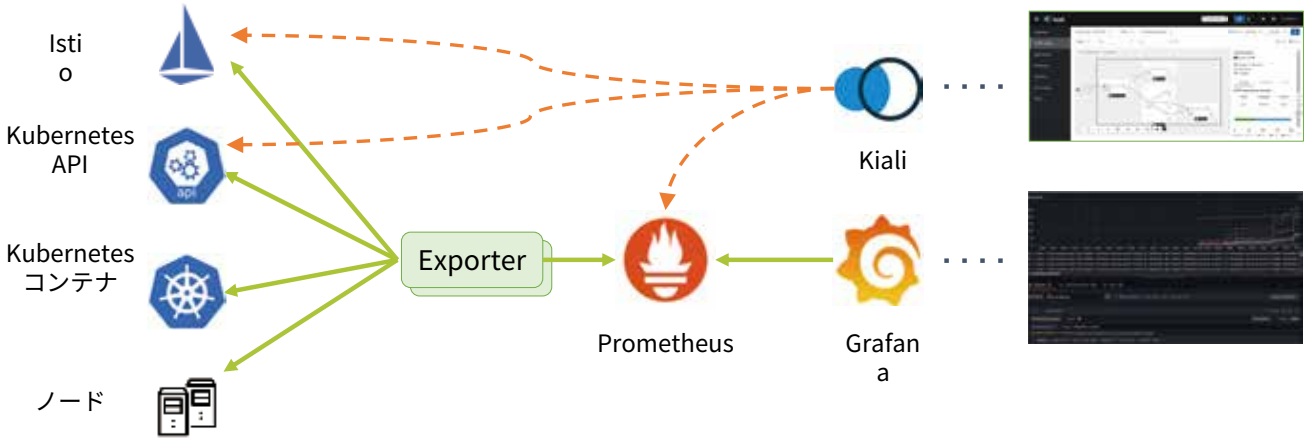
5-1：Prometheus / Grafana / Kialiのインストール

問題編

演習５　Kialiによる可視化と通信分析

5-1：期待される挙動

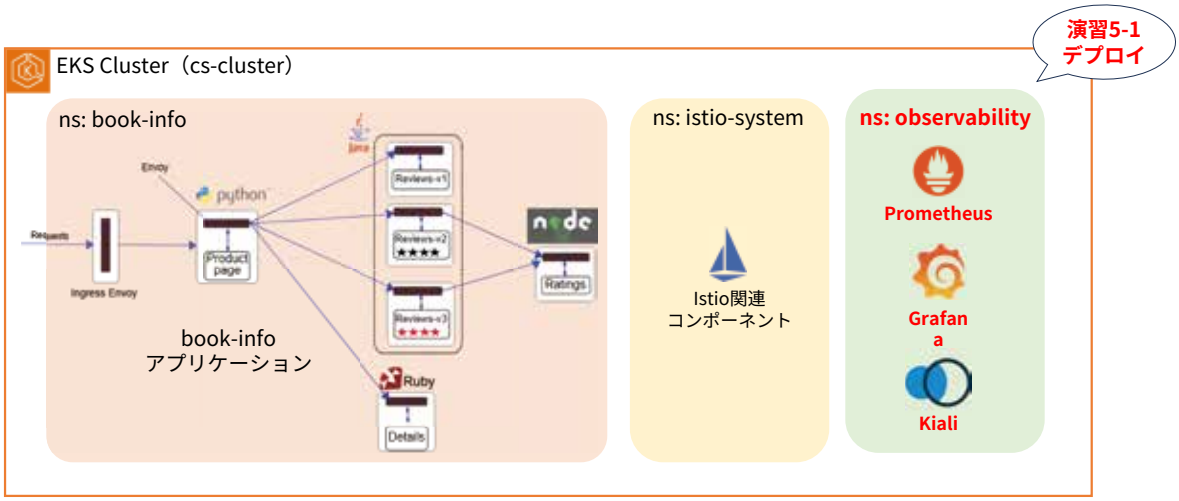
EKSクラスターにPrometheus / Grafana / Kialiをインストールし、可観測性の準備をします。



演習５　Kialiによる可視化と通信分析

5-1： Prometheus / Grafana / Kialiのインストール

EKSクラスターにPrometheus / Grafana / Kialiをインストールしてください。
これらのツールは新規Namespaceにインストールします。



演習５　Kialiによる可視化と通信分析

5-1： Prometheus / Grafana / Kialiのインストール

EKSクラスターにPrometheus / Grafana / Kialiをインストールしてください。
これらのツールは新規Namespaceにインストールします。

演習5-1手順

1. **Namespaceの作成**
   新規Namespace「observability」を作成してください。

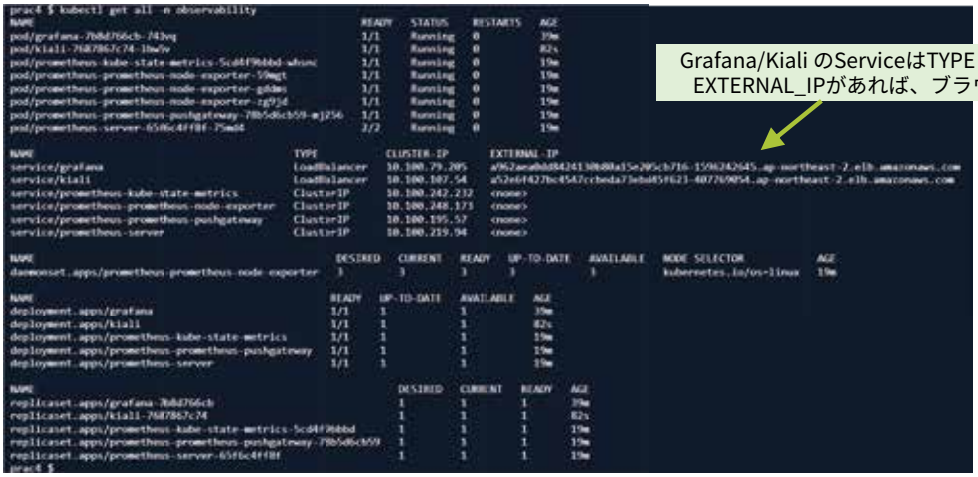
以下手順は、 Helm経由でのインストールを推奨します。

2. **Prometheusのインストール**
   observability NamespaceにPrometheusをインストールしてください。
   ⚠ 動作軽量化のため、今回はPrometheus ServerはPVCを利用しない、Alertmanagerは無効化することを推奨
3. **Grafanaのインストール**
   observability NamespaceにGrafanaをインストールしてください。
4. **Kialiのインストール**
   observability NamespaceにKialiをインストールしてください。
   ⚠ Grafana / Kialiは外部ブラウザからアクセスできるよう、Serviceのタイプを「LoadBalancer」にしてください

演習５　Kialiによる可視化と通信分析

5-1：動作確認

インストールが完了したら、各ツールの起動状態を確認してください。
observability Namespaceにリソースが正常起動していれば、演習5-1は完了です。



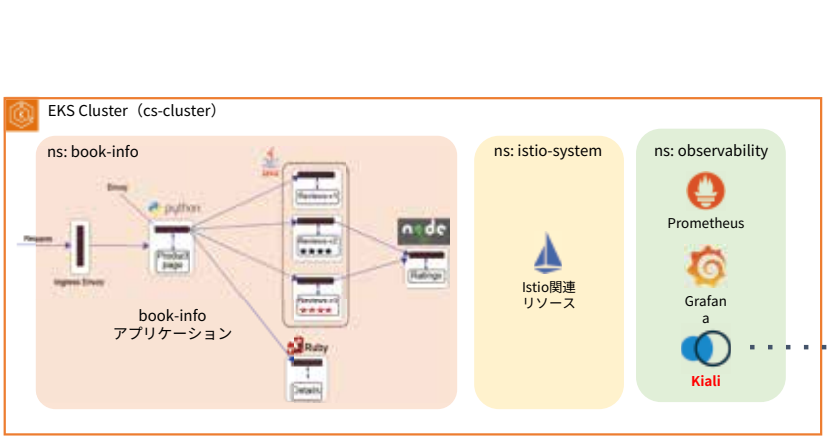
5-2：Kialiトラフィックグラフによる可視化

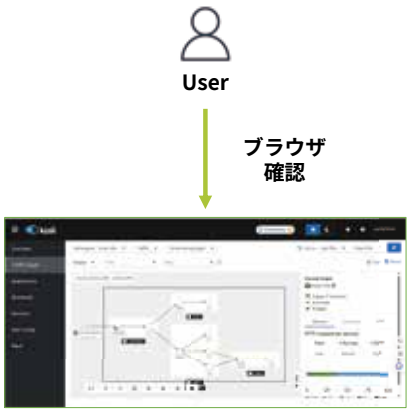
問題編

演習５　Kialiによる可視化と通信分析

5-2：期待される挙動

Kialiのトラフィックグラフにアクセスし、book-infoアプリケーションのトポロジーが表示されることを確認してください。サービスメッシュの可視化を実現します。





Kiali
トラフィックグラフ

演習５　Kialiによる可視化と通信分析

5-2： Kialiトラフィックグラフによる可視化

Kialiにアクセスして、トラフィックグラフを表示し、book-infoアプリケーションのトポロジーが可視化されることを確認してください。サービスのトポロジーを確認できたら、演習5-2は完了です。

演習5-2 手順

1　Kialiの接続先を確認

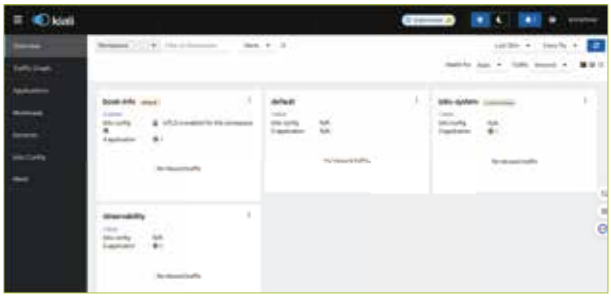
KialiのService（Type：LoadBalancer）を確認し、EXTERNAL-IPを確認してください。
⚠ ServiceのTypeがClusterIPの場合は、kubectl patchでLoadBalancerに変更してください。

```
$ kubectl get service -n observability
```

2　ブラウザからKialiにアクセス

ブラウザから以下にアクセスし、
Kialiが表示されることを確認してください。

http://<EXTERNAL-IP>:20001



演習５　Kialiによる可視化と通信分析

5-2： Kialiトラフィックグラフによる可視化

Kialiにアクセスして、トラフィックグラフを表示し、book-infoアプリケーションのトポロジーが可視化されることを確認してください。サービスのトポロジーを確認できたら、演習5-2は完了です。

3　トラフィックグラフの確認

Kialiのトラフィックグラフ（Traffic Graph）を操作して、book-infoアプリケーションのトポロジーが表示されることを確認してください。またその通信が暗号化されていることを確認してください。
⚠ 事前にブラウザからbook-infoに何度かアクセスし、トラフィックを発生させてください。





5-3：Kialiを用いた通信分析

問題編

演習５　Kialiによる可視化と通信分析

5-3：期待される挙動

book-infoアプリケーションに対して、様々なトラフィックを発生させ、Kialiのトラフィックグラフで検知できることを確認してください。今回は5つのパターンで通信状態を確認します。

パターン１　定常負荷
正常時のベースライン取得
通常運用時のリクエスト数、レイテンシ、エラー率の基準値を取得します。

パターン２　高負荷スパイク
急激なトラフィック増加の観測
高負荷ツールを使用して、急激なリクエスト増加時のシステム挙動を観測します。

パターン３　非認証Pod通信
istio-proxyなしPodからのアクセス
サイドカーを持たないPodからサービスメッシュ内サービスへのアクセスに失敗することを確認します。

パターン４　非認可経路の通信
PeerAuthenticationで許可されていない経路
許可されていない経路で通信し、アクセスに失敗することを確認します。

パターン５　Fault Injection（遅延）
VirtualServiceによる遅延注入
特定のサービスに意図的な遅延を注入し、システムの挙動を観測します。

演習５　Kialiによる可視化と通信分析

5-3： Kialiを用いた通信分析

book-infoアプリケーションに対して、幾つかのパターンでトラフィックを発生させてください。
各通信パターンにおけるトポロジー、レスポンス、リクエスト量などを確認してください。

作業内容

1　パターン１　定常負荷

book-infoアプリケーションのproductpageに定常負荷を掛け、Kialiで通信状態を確認してください。

サンプルコマンド　$ hey -z 60s -q 5 -c 10 http://<EXTERNAL-IP>/productpage

2　パターン２　高負荷

book-infoアプリケーションのproductpageに高負荷を掛け、Kialiで通信状態を確認してください。

サンプルコマンド　$ hey -z 60s -q 40 -c 40 http://<EXTERNAL-IP>/productpage

3　動作確認

定常負荷時、高負荷時を掛けた際の変化を確認してください。

- サービストポロジー
- レスポンスタイム（p95 Percentile）
- リクエスト量（Traffic Rate）

演習５　Kialiによる可視化と通信分析

5-3： Kialiを用いた通信分析

book-infoアプリケーションに対して、幾つかのパターンでトラフィックを発生させてください。
各通信パターンにおけるトポロジー、レスポンス、リクエストなどを確認してください。

作業内容

4　パターン３　非認証Pod通信

一時的にout-mesh Pod（istio-proxyなし）を立て、book-infoアプリ（ratings）にアクセスしてください。
Kialiで一時的なPodからratingsへのトラフィックが100%エラーになっていることを確認してください。

サンプルコマンド
$ kubectl run out-mesh --rm -it -n book-info --image=curlimages/curl -- sh
~ $ curl -v http://ratings.book-info.svc.cluster.local:9080/ratings/1

5　パターン４　非認可経路の通信

AuthorizationPolicyで許可されていないPod間通信を発生させます。
Kialiでreivews v2からratingsへのトラフィックが100%エラーになっていることを確認します。

サンプルコマンド
$ kubectl exec deploy/reviews-v2 -n book-info -it -- curl -v http://details:9080/details/1

演習５　Kialiによる可視化と通信分析

5-3： Kialiを用いた通信分析

book-infoアプリケーションに対して、幾つかのパターンでトラフィックを発生させてください。
各通信パターンにおけるトポロジー、レスポンス、リクエストなどを確認してください。

作業内容

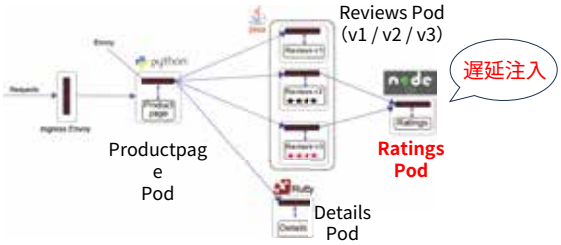
6　パターン５　Fault Injection

Fault injection（VirtualService）で意図的に5秒の処理遅延を注入し、reviews v2/v3からratingsへの通信をタイムアウトさせてください。
またKialiで該当トラフィックがエラーになっていることを確認してください。

⚠ ratingsへの通信はproductpage側の設定により、3秒でタイムアウトします。

⚠ VirtualServiceのサンプルマニフェストは次ページ。

```
$ nano fault-injection.yaml
$ kubectl apply -f fault-injection.yaml
```





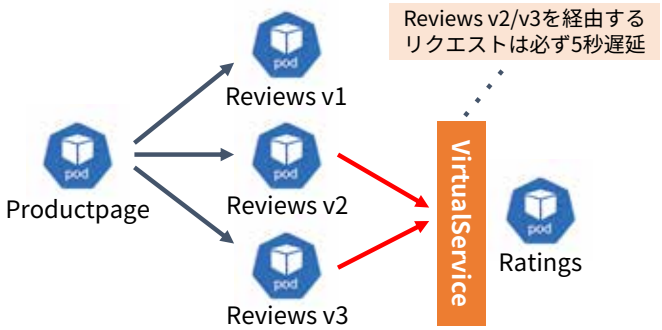
演習５　Kialiによる可視化と通信分析

5-3：Virtual Serviceのサンプルマニフェスト

Fault Injection（遅延注入）を実現するVirtualServiceのサンプルマニフェストです。
このマニフェストを適用することで、Reviews v2/v3 からRatingsへの通信が一律5秒遅延します。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: ratings
  namespace: book-info
spec:
  hosts:
  - ratings
  http:
  - fault:
      delay:
        fixedDelay: 5s
        percentage:
          value: 100
    route:
    - destination:
        host: ratings
```

fault-injection-delay.yaml



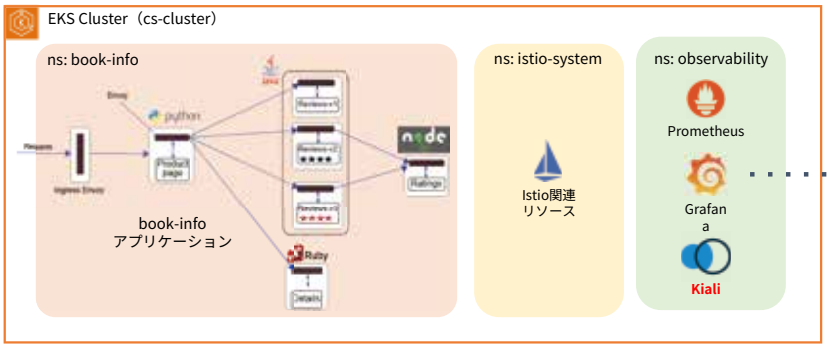
5-4：Grafanaダッシュボード作成

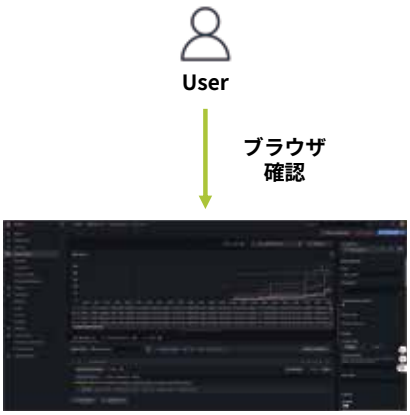
問題編

演習５　Kialiによる可視化と通信分析

5-4：期待される挙動

Grafanaにアクセスし、EKSクラスターのメトリクスを使ったダッシュボードを作成してください。
Prometheusでメトリクスを収集し、Grafanaで参照することで実現します。





Grafana
ダッシュボード

演習５　Kialiによる可視化と通信分析

5-4：Grafanaダッシュボード作成

GrafanaにアクセスしEKSクラスターのメトリクスを使ったダッシュボードを作成してください。
Prometheusでメトリクスを収集し、Grafanaで参照することで実現します。

演習5-4 手順

1 **Grafanaの接続先を確認**

GrafanaのService（Type：LoadBalancer）を確認し、EXTERNAL-IPを確認してください。

```
$ kubectl get service -n observability
```

2 **ブラウザからGrafanaにアクセス**

ブラウザから以下にアクセスし、
Grafanaが表示されることを確認してください。
http://<EXTERNAL-IP>

3 **Grafanaにログイン**

ID/PW：admin/adminでログインしてください。



Grafana ログイン後トップページ

演習５　Kialiによる可視化と通信分析

5-4：Grafanaダッシュボード作成

GrafanaにアクセスしEKSクラスターのメトリクスを使ったダッシュボードを作成してください。
Prometheusでメトリクスを収集し、Grafanaで参照することで実現します。

演習5-4 手順

4 **PrometheusとGrafanaの統合**

左ペイン > Connections > Data Source > Add data source > Prometheusを選択
Connectionに、プロメテウスサーバーのURLを入力して「Save & Test」を押下
　URL：http://prometheus-server.observability.svc.cluster.local

5 **ダッシュボード作成**

左ペイン > Dashboards > New Dashboardを選択
K8sクラスター、Istio、アプリケーション関連のメトリクスを使って複数パネルを作ってください。

メトリクス例
- node_cpu_seconds_total
- istio_request_duration_milliseconds_bucket
- istio_requests_total



Grafana ダッシュボード作成例

演習５　Kialiによる可視化と通信分析

5-4：Grafanaダッシュボード

Prometheusで収集したメトリクスをGrafanaのダッシュボードで表示してください。
ダッシュボードに複数パネルを作成し、何かしらグラフを表示できれば、演習5-4は完了です。



ダッシュボード作成例

演習５　Kialiによる可視化と通信分析

合格判定基準

演習５　Kialiによる可視化と通信分析

合格判定基準

演習5の合格判定基準は以下の通りです。

5-1：Prometheus / Grafana / Kialiのインストール

- observability Namespaceに、Prometheus / Grafana / Kiali 関連のリソースが正常に起動している
- Grafana / KialiのServiceがLoadBalancerタイプでEXTERNAL-IPを持っている

5-2：Kialiトラフィックグラフによる可視化

- ブラウザからKialiにアクセスできる（http://<EXTERNAL-IP>:20001）
- Kialiのトラフィックグラフでbook-infoのトポロジーが表示される
- サービス間の依存関係と通信方向が正しく可視化されている

5-3：Kialiを用いた通信分析

- パターン１：定常負荷　正常なトラフィックフローが可視化される
- パターン２：高負荷　リクエスト量増加とレスポンスタイム変化を確認できる
- パターン３：非認証Pod　istio-proxyなしPodからの通信が100%エラーになる
- パターン４：非認可経路　AuthorizationPolicyで拒否された通信がエラー表示される
- パターン５：Fault Injection　遅延注入によりreviews→ratingsがタイムアウトする

5-4：Grafanaダッシュボード作成

- ブラウザからGrafanaにアクセスできる（http://<EXTERNAL-IP>）
- GrafanaにPrometheusデータソースが正しく設定できている
- EKSクラスターまたはIstioのメトリクスを含むダッシュボードが作成され、データが正常に表示される

確認テスト

概要 確認テスト

【問題1】クラウドネイティブセキュリティの 4C に含まれないものはどれか？

A. Cloud
B. Cluster
C. Container
D. ControlPlane

【問題2】4C モデルで「Cloud」レイヤーの主な役割として最も適切なのは？

A. コンテナ間通信の暗号化
B. インフラ基盤（IAM・ネットワーク・暗号化）で外部侵入を防ぐ
C. コンテナイメージの脆弱性解析
D. コードの静的解析を行う

【問題3】多層防御の目的として正しいものは？

A. 一つの防御層が突破されても全体の影響を最小化する
B. セキュリティ設定を一箇所にまとめる
C. マイクロサービス化を防ぐため
D. システムの設定をなるべく複雑にするため

【問題4】「継続的なセキュリティテスト」の代表例として最も適切なのは？

A. 本番環境でのみテストを実施する
B. 手動でログを目視チェックする
C. CI/CD に脆弱性チェックを組み込み、Shift Left（早期検出）を実現する
D. 一年に一度セキュリティ診断を行う

【問題5】「ホスト（Host）」レイヤーで最も重要なセキュリティ対策は？

A. OS・カーネル・ランタイムの防御
B. イメージ署名
C. AuthorizationPolicy の設定

D. ServiceAccount の最小権限設定

==========

【問題1】

正解：D

【問題2】

正解：B

【問題3】

正解：A

【問題4】

正解：C

【問題5】

正解：A

演習1 確認テスト

【問題1】RBAC における RoleBinding の役割はどれか？

A. Role を ServiceAccount に紐づける
B. Pod の起動を制御する
C. NetworkPolicy を適用する
D. Namespace を作成する

【問題2】NetworkPolicy を適用しない場合のデフォルト挙動は？

A. 全通信拒否
B. PodごとにACL設定が必要
C. すべてのPod間通信が許可
D. DNSのみ許可

【問題3】PSS（Pod Security Standards）の restricted が防ぐものはどれか？

A. Pod の自動スケール
B. root ユーザでの実行や特権昇格
C. Service の利用
D. Namespace の作成

【問題4】次の Role の resources: ["pods"] が意味するものは？

A. Pod のメトリクス取得
B. Pod リソースに対する操作権限の対象
C. Pod のスケジューリング設定
D. Pod のイベントログのみ

【問題5】NetworkPolicy を適用後、Pod が名前解決に失敗する最も典型的な原因はどれか？

A. kube-system Namespace への Egress を許可していない
B. Pod のイメージが古い
C. Pod のメモリ不足

D. Ingress のみ設定して Egress を設定していない

【問題6】PSSの baseline と restricted の違いとして正しいのはどれか？

A. baseline は root 実行が必須、restricted は root 禁止

B. baseline はホストPath を必ず許可する

C. restricted は seccomp を無効化する必要がある

D. baseline は特権コンテナ禁止だが root 実行は許容、restricted は両方禁止

==========

【問題1】

正解：A

【問題2】

正解：C

【問題3】

正解：B

【問題4】

正解：B

【問題5】

正解：A

【問題6】

正解：D

演習2 確認テスト

【問題1】脆弱なイメージ nginx:latest をそのまま使うリスクとして最も適切なのは？

A. Pod の再起動が早くなる
B. イメージのビルド時間が短くなる
C. Service の作成が失敗する
D. 脆弱性が多く、攻撃者に悪用される可能性が高い

【問題2】Trivy による脆弱性診断の結果を改善する一般的な方法は？

A. Pod 数を増やす
B. ベースイメージを alpine / distroless に変更する
C. Deployment の replicas を 1 にする
D. Service を ClusterIP に変更する

【問題3】Cosign で署名する際に必要なファイルは？

A. cosign.yaml
B. cosign.pub
C. config.json
D. cosign.key

【問題4】Kyverno が署名検証を行うタイミングは？

A. Pod 内でアプリ起動時
B. イメージビルド時
C. Kubernetesデプロイ時
D. ノード起動時

【問題5】セキュアなPodでunshare コマンドが失敗する理由として正しいのは？

A. Pod に CPU limit が設定されている
B. seccomp により dangerous syscalls がブロックされる
C. コンテナイメージの容量が小さいため

D. Pod が PodSecurityPolicy を利用しているため

【問題6】AppArmorの役割として正しいものは？

A. 利用可能なシステムコールの制御

B. root権限の分割付与・削除

C. ファイルの読み書き・ネットワークの制御

D. Pod のログ量を削減する

==========

【問題1】

正解：D

【問題2】

正解：B

【問題3】

正解：D

【問題4】

正解：C

【問題5】

正解：B

【問題6】

正解：C

演習3 確認テスト

【問題1】CI/CD パイプラインにセキュリティゲートを導入する最大のメリットは？

A. Kubernetes のリソースが増える
B. Node の CPU 使用率を安定させる
C. GitHub Actions のワークフローが高速化する
D. セキュリティ問題を「マージ前に」検知して流出を防げる

【問題2】GitHub Actions の workflow で複数のジョブを定義する理由として最も適切なものは？

A. 同じ処理を繰り返し実行するため
B. 異なる処理を分離・並列で実行するため
C. ジョブ数を増やすと高速化できるため
D. GitHub 仕様で複数ジョブが必須なため

【問題3】ruff と black を CI（GitHub Actions）に組み込む最大のメリットはどれか？

A. runner の料金が安くなるため
B. セキュリティ問題を自動で解決できるため
C. コード品質をPR段階で自動検証できるため
D. デプロイが高速になるため

【問題4】CodeQL を Pull Request 時に実行することが Shift Left に該当する理由は？

A. レビュワーの負担を減らせるため
B. デプロイ後に対応しやすいため
C. マージ前に脆弱なコードを検出できるため
D. ワークフロー速度が向上するため

【問題5】Trivy の CI 実行で最も避けるべき設定は？

A. 定期スキャン（cron）
B. Pull Request トリガー

C. exit-code=0 固定

D. レポート出力（sarif）

【問題6】Dependabot を CI で自動マージすべきではない理由は？

A. コストが高くなるから

B. PR のレビューができなくなるため、破壊的変更を含む可能性がある

C. GitHub の制限

D. DockerHub の制限

==========

【問題1】

正解：D

【問題2】

正解：B

【問題3】

正解：C

【問題4】

正解：C

【問題5】

正解：C

【問題6】

正解：B

演習4 確認テスト

【問題1】セキュリティ強度が最も高いシークレット情報の管理方法はどれか？

A. ConfigMap に保存する
B. Secret に保存し、EncryptionConfiguration を有効化する
C. Secret Manager / Parameter Store に外部化する
D. Pod の環境変数に直接書く

【問題2】Service Mesh を導入する最大のメリットはどれか？

A. コンテナイメージ容量が削減され、コンテナの起動時間が短縮する
B. アプリ側でセキュリティ・観測性・通信制御を実装する必要がなくなる
C. セキュリティチェック項目が減り、GitHub Actions の速度が向上する
D. CI/CDパイプラインが不要になり、サービスメッシュで管理可能になる

【問題3】PeerAuthenticationのmode:PERMISSIVE の特徴として正しいものはどれか？

A. mTLSのみ許可する
B. mTLS/平文の両方を許可する
C. 通信をすべて拒否する
D. 外部アクセスのみ許可する

【問題4】mTLS 有効化後に起こりやすいトラブルとして最も正しいのは？

A. Service の ClusterIP が変わる
B. Pod の再スケジューリングが頻発する
C. HPA が動かなくなる
D. mTLS に非対応のサービスからの通信が拒否される

【問題5】Istio の AuthorizationPolicy の ALLOW ルールが複数ある場合、どのように評価される？

A. いずれか1つのルールがマッチすれば許可
B. すべての条件を満たす場合のみ許可

C. ルールはランダムで選択される

D. 常に最後のルールが適用される

【問題6】Istio を導入していない環境で AuthorizationPolicy のような L7 制御ができない理由は？

A. Node に十分な CPU がないため

B. Kubernetes の API に制約があるため

C. ネットワークが IPv6 のため

D. L7 制御（HTTP Path/Method）はアプリ側で実装する必要があるため

==========

【問題1】

正解：C

【問題2】

正解：B

【問題3】

正解：B

【問題4】

正解：D

【問題5】

正解：A

【問題6】

正解：D

演習5 確認テスト

【問題1】可観測性（Observability）の目的として最も適切なのは？

A. Pod のスケジュール方法を決定すること
B. システム内部の状態を外部から理解し、異常を早期に検知すること
C. Kubernetes のバージョンを自動更新すること
D. コンテナイメージのサイズを最小化すること

【問題2】Kiali の最大の特徴として最も適切なのは？

A. Kubernetesサービスのメトリクスを可視化できる
B. mTLSによって暗号化された通信かが分かる
C. Service Mesh のトラフィックトポロジーをリアルタイム可視化する
D. コンテナイメージにセキュリティパッチを適用する

【問題3】Kiali の Graph 表示が反映されない最も一般的な原因は？

A. アプリケーションにアクセスしておらず、トラフィックが発生していない
B. KialiがServiceMesh の外部で動作している
C. Prometheus が PushGateway を使用している
D. Istio が DaemonSet で構成されている

【問題4】Fault Injection の主な目的として正しいものは？

A. CPU リソースを強制的に増加させるため
B. 通信遅延やエラーを発生させ、耐障害性を検証するため
C. トラフィックを常に均等に分散させるため
D. 自動スケールを高速化するため

【問題5】Grafana の役割として最も適切なのは？

A. Pod の再起動を自動化する
B. コンテナイメージをビルドする
C. Service Mesh を構築する

D. メトリクスデータをグラフ化しダッシュボードで可視化する

【問題6】レイテンシの監視を行う際に、平均値より 90p / 99p が重要とされる理由は？

A. 平均値はネットワークで使えないため

B. 平均値は異常値が隠れ、遅いリクエストの実態が見えなくなるため

C. 90p/99p は計算コストが低いため

D. Prometheus で平均値が計算できないため

==========

【問題1】

正解：B

【問題2】

正解：C

【問題3】

正解：A

【問題4】

正解：B

【問題5】

正解：D

【問題6】

正解：B

令和７年度「専門職業人材の最新技能アップデートのための専修学校リカレント教育（リ・スキリング）推進事業」
情報技術者の技能アップデートのためのリカレント教育推進事業

コンテナサーバーセキュリティ教材資料
問題編

令和８年２月

一般社団法人全国専門学校情報教育協会

〒164-0003 東京都中野区東中野 1-57-8 辻沢ビル３F

電話：03-5332-5081 FAX.03-5332-5083

●本書の内容を無断で転記、掲載することは禁じます。