Automate Security Configuration of Multiple Pods using Kubernetes

Somchart Fugkeaw, Sirapitch Boonyasampan, Ittiwat Nimitliupanit, Thanapat Thaipakdee
School of ICT, Sirindhorn International Institute of Technology
Thammasat University, Pathum Thani, Thailand
somchart@siit.tu.ac.th, sirapitchboonyasampan@gmail.com, ittiwat0812@gmail.com, name22077@gmail.com

The dynamic and evolving landscape of cloud computing has propelled Kubernetes as a pivotal platform for orchestrating containerized applications. This project introduces an innovative application to enhance the security of Kubernetes pods by automating security configurations. The proposed solution addresses three critical areas: Isolation, Role-Based Access Control (RBAC), and Ingress-Specific IP. Our approach introduces a granular isolation framework at the container level, ensuring that unauthorized access is prevented between pods, thereby reducing the attack surface and maintaining integrity. We implement a sophisticated RBAC system at the pod level, defining and enforcing access permissions based on user roles and specific policies. This ensures that only authorized entities interact with sensitive pod resources, striking a balance between operational efficiency and security. The project also introduces network policies to regulate ingress traffic between pods, allowing only specified IP addresses to communicate with designated containers, reducing the risk of unauthorized access and safeguarding sensitive data from eavesdropping. By integrating these security measures, our solution establishes a robust security posture for Kubernetes deployments, securing containerized applications from a broad spectrum of internal and external threats while facilitating compliance with industry standards and regulations, ensuring secure, scalable, and reliable Kubernetes environments.

Keywords: Kubernetes, pod security, policy enforcement, container isolation, RBAC, ingress traffic control, secure cloud management

1. Introduction

In the dynamic landscape of cloud computing, Kubernetes has emerged as a crucial platform for orchestrating containerized applications. The scale and complexity of Kubernetes deployments have grown, making robust security measures essential. This project aims to automate the security configuration of Kubernetes pods by focusing on three key areas: isolation, Role-Based Access Control (RBAC), and ingress-specific IP controls.

The motivation behind this project is twofold. Firstly, the growing sophistication of cyber threats necessitates advanced security measures to safeguard Kubernetes deployments. Secondly, regulatory requirements for data protection and privacy call for stringent protocols that ensure compliance and mitigate unauthorized access to sensitive resources.

The project addresses these challenges by developing a comprehensive security framework. This framework implements:

1. Isolation: A granular isolation framework at the container level, preventing unauthorized access between pods, reducing the attack surface, and maintaining integrity.
2. RBAC: A dynamic RBAC system at the pod level, defining and enforcing access permissions based on user roles and specific policies. The system allows for granular control, ensuring that only authorized entities interact with sensitive pod resources, balancing security with operational efficiency. The RBAC policies can be dynamically managed, adapting to different apps in different pods, with examples provided to illustrate their functionality.
3. Ingress-Specific IP: Network policies regulate ingress traffic between pods, allowing only specified IP addresses to communicate with designated containers. This reduces the risk of unauthorized access and safeguards sensitive data from eavesdropping.

By integrating these security measures, the project aims to establish a robust security posture for Kubernetes deployments. The solution secures containerized applications from both internal and external threats, while facilitating compliance with industry standards and regulations, ensuring secure, scalable, and reliable Kubernetes environments.

2. Related work

In our project, we're focusing on making sure data and resources are safe within the Kubernetes pod. In this section, we'll look at what others have done to improve security between pods. We'll explore how they've set rules, controlled access, and encrypted data. By learning from their methods, we can make our project even more secure.

[1] ITNEXT Article by Shazad Brohi proposed implementing Pod Security Policies (PSPs) to secure Kubernetes clusters. These policies manage cluster operations by setting conditions for pods, covering aspects such as host access, container UIDs, and volumes.

[2] Alawala & Ray on Amazon EKS proposed pod security standards for Amazon EKS to safeguard clusters against unwanted changes. They highlight the role of PSPs in enforcing security settings but note their deprecation in Kubernetes 1.21 and removal in 1.25, urging for alternative strategies.

[3] Rostami on RBAC in Kubernetes proposed Role-Based Access Control (RBAC) to secure Kubernetes control plane layers, including etcd objects. RBAC allows fine-grained access control over components, including PODs, namespaces, and deployments, with roles defined at the namespace or cluster level.

[4] Budigiri et al. on Network Policies proposed network policies in Kubernetes, particularly for 5G cloud environments. They demonstrate that these policies impose minimal performance overhead and highlight potential security challenges for container isolation.

[5] Binnie & McCune on Kubernetes Authorization proposed using RBAC for Kubernetes authorization, emphasizing user access control over API objects. They cover various authorization mechanisms and tools for RBAC auditing, highlighting both cluster and namespace-level roles, as well as built-in roles and groups.

[6] Microsoft Learn discusses developer best practices for securing pods in Azure Kubernetes Services (AKS). The article emphasizes the importance of pod security context and how it can be configured to manage aspects such as container privileges, capabilities, and security policies. It also highlights how these practices contribute to safeguarding Kubernetes deployments against unauthorized access and vulnerabilities.

[7] Red Hat provides a guide to Kubernetes Security Context and Pod Security Policy (PSP). It explains how security context configurations manage container privileges, user IDs, and capabilities, enhancing cluster security. PSPs enforce these settings across the cluster, providing a cohesive strategy to manage security risks.

[8] Packet Pushers presents four methods of Kubernetes isolation, highlighting how various isolation strategies can help secure Kubernetes clusters. These methods include network segmentation, namespace isolation, runtime restrictions, and RBAC configurations, each contributing to a layered security approach.

[9] AWS introduces security groups for pods, offering fine-grained controls to restrict ingress traffic. This feature helps manage communication between pods, minimizing the risk of unauthorized access and providing a secure communication framework for Kubernetes clusters.

[10] Wiz's 2023 Kubernetes Security Report provides key takeaways on the state of Kubernetes security, identifying common vulnerabilities and offering insights into securing Kubernetes deployments. The report also outlines strategies for improving security posture, including securing the control plane and managing container configurations.

[11] Aqua offers ten steps for securing Kubernetes clusters, focusing on securing data transmission, managing access control, and monitoring deployments. These steps include enforcing RBAC policies, implementing network policies, and securing pod configurations, providing a comprehensive strategy for securing Kubernetes environments.

[12] An IEEE publication discusses various security measures for Kubernetes, including network policies and RBAC configurations【12】, emphasizing their role in securing Kubernetes clusters against internal and external threats.

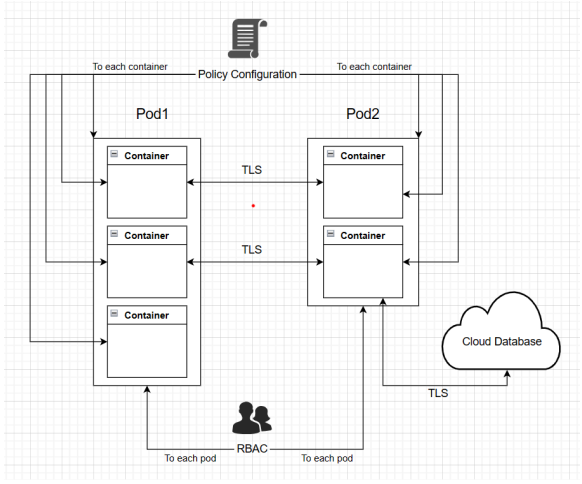
[13] ARMO provides insights into Kubernetes security best practices, covering topics such as access control, data encryption, and monitoring. The blog offers strategies to secure Kubernetes deployments, including enforcing network policies and RBAC configurations, minimizing security risks.

[14] Trilio's resources focus on Kubernetes security monitoring, discussing tools and techniques for tracking Kubernetes deployments. The guide highlights strategies for identifying and mitigating security threats, ensuring secure Kubernetes operations.

[15] SentinelOne offers a guide on defending modern cloud-based workloads, including Kubernetes security strategies. The guide covers aspects such as securing data transmission, managing access control, and implementing monitoring solutions, providing a comprehensive strategy for securing Kubernetes deployments.

3. Our Proposed Scheme

A. System Model



System Model:

1. Container-Level Policies:

- Policy Management: Each container is configured with its own set of security policies, defining access controls, network restrictions, and encryption requirements specific to the container's functionality and sensitivity of data it handles.

- Policy Enforcement: Policies are enforced at the container level to ensure isolation and security within the pod, preventing unauthorized access and ensuring compliance with organizational security requirements.

2. RBAC for Pod Access:

- Role-Based Access Control (RBAC): Role-Based Access Control (RBAC) in Kubernetes dynamically adapts access controls to fit the changing needs of applications across different pods. For instance, RBAC policies can be designed to automatically adjust permissions based on factors such as the deployment environment or application state. This dynamic adjustment ensures that permissions are as permissive as necessary and as restrictive as feasible, aligning with the principle of least privilege.Here are examples of how RBAC policies can be applied:

- Development vs. Production Access: Define separate roles and role bindings that restrict developers to manage pods only in a development namespace, while stricter controls are enforced in production.
- Temporary Privilege Escalation: Implement policies allowing operations staff temporary access to sensitive operations during incident resolution, with automatic revocation after a set period.

3. Data Encryption:

- Pod-to-Pod Communication: Data exchanged between pods is encrypted using robust encryption algorithms such as TLS

-Pod-to-Cloud Communication: Communication between pods and cloud services is encrypted to protect sensitive information from unauthorized access and eavesdropping, maintaining data confidentiality and security.

4. Access Control:

- RBAC for Pod Access: RBAC policies are enforced at the pod level to regulate access to pod resources based on user roles and permissions, ensuring that only authorized entities can interact with the pod and its containers.

- Network Policies: Network policies are implemented to restrict network communication between containers within the same pod and between pods, allowing only authorized traffic according to defined rules and access controls.

5. Integration Interfaces:

- Kubernetes API Integration: Interfaces with the Kubernetes API server to retrieve pod metadata, manage security policies, and enforce access controls within the Kubernetes cluster.

- Cloud Service Integration: Integrates with cloud service providers to establish secure communication channels, exchange cryptographic keys, and enforce

encryption policies for communication between pods and cloud services.

B. Methodology

Phase 1: Pod Listing and Selection Phase From Cloud

This phase introduces a systematic approach to enforce policies within Kubernetes pods at the container level, focusing on root access restriction, ingress traffic control, isolation enforcement, and overall policy enforcement.

Function listPods():
  Get the element with ID 'podList'
  If element is not found:
    Log error message 'Element with ID "podList" not found.'
    Return

  Try:
    Call coreV1Api.listPodForAllNamespaces() to get the list of pods
    Extract the items from the response body as pods

    Clear existing content in 'podList'
    Set innerHTML of 'podList' to 'Select : '

    Create a select element
    Set the width and overflow style of the select element
    Set the ID of the select element to 'podSelect'
    Set the size of the select element to 1 (dropdown)

    Create a default option element
    Set the text content of the default option to 'Select a pod'
    Disable and select the default option
    Append the default option to the select element

    For each pod in pods:
      If pod's namespace is 'default':
        Log pod information to console
        Create an option element
        Set the value of the option element to 'pod.metadata.labels.app|pod.metadata.namespace'
        Set the text content of the option element to 'pod.metadata.name (pod.metadata.namespace)'
        Append the option element to the select element

    Append the select element to 'podList'

  Catch error:
    Log error message 'Error listing pods: ' and the error
    Set the text content of 'podList' to 'Failed to load pods. Error: ' and the error message

Phase 2: Enforce The Policy To Each Pod

In this phase, the goal is to apply security policies to each pod within the Kubernetes cluster to enhance security. This includes implementing pod isolation policies, specific pod ingress policies, and enforcing non-root policies between pods.

Algorithm 1: Pod isolation policy

In Kubernetes, network policies are used to control the communication between pods. The following pseudo code demonstrates the creation of a network isolation policy for a specified pod within a given namespace. This policy denies both ingress and egress traffic to ensure complete isolation of the pod.

Function createIsolationPolicy(podName, namespace) {
  set apiVersion to 'networking.k8s.io/v1'
  set kind to 'NetworkPolicy'
  set metadata to {
    name: concatenate 'isolate-' with podName
    namespace: namespace
  }
  set spec to {
    podSelector: {
      matchLabels: {
        'app': podName
      }
    }
    policyTypes: ['Ingress', 'Egress']
    ingress: []

```
    egress: []
  }
  return {
    apiVersion: apiVersion
    kind: kind
    metadata: metadata
    spec: spec
  }
}
```

Algorithm 2: Specific Pod ingress policy

In Kubernetes, network policies are used to control the communication between pods. The following pseudo code demonstrates the creation of a specific ingress policy for a specified pod within a given namespace. This policy allows ingress traffic only from specified IP addresses, while denying all other ingress and egress traffic.

```
Function createSpecificIsolationPolicy(podName,
namespace, allowedIps) {
  allowedIps = Array.isArray(allowedIps) ?
allowedIps : [allowedIps];
  set apiVersion to 'networking.k8s.io/v1'
  set kind to 'NetworkPolicy'
  set metadata to {
    name: concatenate 'isolate-' with podName
    namespace: namespace
  }
  set spec to {
    podSelector: {
      matchLabels: {
        'app': podName
      }
    }
    policyTypes: ['Ingress', 'Egress']
    ingress: [
      {
        from: allowedIps.map(ip => ({
          ipBlock: {
            cidr: concatenate ip with '/32'
          }
        }))
      }
    ]
    egress: []
  }
  return {
    apiVersion: apiVersion
    kind: kind
    metadata: metadata
    spec: spec
  }
}
```

Algorithm 3: Can't run as root policy

In Kubernetes, it's important to enforce security policies to prevent containers from running as root. The following pseudo code demonstrates the application of a non-root policy to deployments within a namespace, based on a specified pod name filter.

```
Function applyNonRootPolicy(podNameFilter,
namespace)
  Load the Kubernetes configuration
  Initialize kc as new KubeConfig()
  kc.loadFromDefault()
  Create a CoreV1Api client
  Initialize k8sApi as
kc.makeApiClient(CoreV1Api)

Try
  List all pods in the specified namespace
podsResponse = k8sApi. listNamespacedPod
(namespace)
 pods = podsResponse.body.items

  Filter pods based on the provided name filter
filteredPods = Filter pods where pod.metadata.name
includes podNameFilter

  Extract unique deployment names from the filtered
pods
deploymentNames = Get unique deployment names
from filteredPods

   Iterate over each unique deployment name
 For each deploymentName in deploymentNames

   Apply non-root policy to each unique deployment
Print "Applying non-root policy to deployment",
deploymentName, "in namespace", namespace
```

Note: The actual implementation would use k8sAppsApi.patchNamespacedDeployment(...) here

Catch error
Handle errors and print detailed error information
Print "Failed to retrieve or update deployments in namespace", namespace, ":", error
If error.response exists
Print 'HTTP Status:', error.response.statusCode
Print 'Response Headers:', error.response.headers
Print 'Response Body:', error.response.body
Else
Print 'Error details:', error.message

Phase 3 : Revoke The Policy From Each Pod

Algorithm 1: Discard Networkpolicy

This algorithm is designed to delete a specific NetworkPolicy associated with a pod in a given namespace.

Function deleteIsolatePolicy(podName, namespace):
Try:
Attempt to delete the NetworkPolicy
Await netV1Api.deleteNamespacedNetworkPolicy (isolate-' + podName, namespace)

Log deletion success
Print 'Network policy deleted: isolate-' + podName

Catch error:
Handle errors
(omitting the handling)

Algorithm 2: Discard Can't run as root policy

This algorithm is designed to remove the "runAsNonRoot" policy from all containers within a specified pod.

Function deleteNonRootPolicy(podName, namespace):
Try:
Read the pod details
const response = await coreV1Api. readNamespacedPod(podName, namespace);
const pod = response.body;

Remove runAsNonRoot from security context for each container
For each container in pod.spec.containers:
If container.securityContext and container.securityContext.runAsNonRoot:
Delete container.SecurityContext. runAsNonRoot

Prepare the patch
const patch = [
{ op: 'replace', path: '/spec/containers', value: pod.spec.containers }
];

Patch the pod to remove runAsNonRoot policy
Await coreV1ApipatchNamespacedPod (podName, namespace, patch, undefined, undefined, undefined, {
headers: { 'Content-Type': 'application/json-patch+json' }
});

Log success message
Print 'Pod "', podName, '" in namespace "', namespace, '" runAsNonRoot policy removed successfully.'

Catch error:
Handle errors
Print 'Error:', error
If error.response:
Print 'Status Code:', error.response.statusCode
Print 'Response Body:', error.response.body
Print 'Failed to delete non-root policy for pod "', podName, '" in namespace "', namespace, '".'

4. Implementation And Experiment

This section describes the implementation of our security configuration system for Kubernetes pods, detailing its design and architecture. It also outlines the experiments conducted to evaluate the

system's functional modules, security mechanisms, and overall performance.

A.Prototype System

The system prototype for our Kubernetes security configuration system is developed based on JavaScript, HTML, and CSS. We use NPM packages in Node.js to implement system APIs and services for service integration. Additionally, we use Docker Desktop and Google Cloud as a Docker engine, with Kubernetes to manage Docker containers, images, and Kubernetes clusters. In this section, we provide example screenshots demonstrating the GUI of our system, showcasing its interface and functionalities.

.

Figure 1 shows the interface of a "Kubernetes Pod Configuration Manager" application. The user can select a pod from a dropdown menu and then apply various security configurations. Options include "Isolate selected pod," "Cannot Access Root," and "Allow Ingress from Specific IP," with a "Select" button and an "Apply Changes" button for applying configurations to the chosen pod. The terminal section shows a command output, highlighting an "isolate-service" pod configuration.

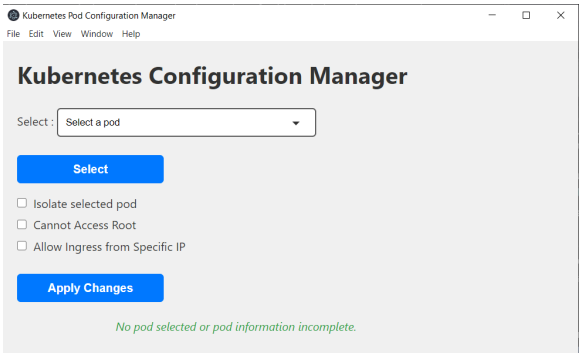


Fig 1. Configuration Page

B.Experiment

In this experiment, we demonstrate the creation and use of an application designed to configure security policies for Kubernetes pods. This application focuses on two key aspects:

- Isolation of Specific Pods: Ensuring that designated pods are completely isolated from other pods within the Kubernetes cluster, preventing any unauthorized communication.
- Ingress Traffic Control: Defining and enforcing policies that allow only specified pods to communicate with the designated isolated pods.

Isolation of Specific Pods

Table 1 present a summarizes results of communication tests between Service A, Service B, and Service C before and after applying an isolation policy to Service A. It demonstrates the impact of the isolation policy on inter-service communication within the Kubernetes cluster.

TABLE 1. Result of isolation

Communication Test	From	To	Result Before Isolation	Result After Isolation
External Communication	Service A	google.com	Success	Failed
Internal Communication	Service A	Service B	Success	Failed
Internal Communication	Service A	Service C	Success	Failed
Ingress Communication	Service B	Service A	Success	Failed
Ingress Communication	Service C	Service A	Success	Failed

```
User@LAPTOP-T5IBK3AO MINGW64 ~/OneDrive/Desktop/k8s-config-app
$ kubectl exec --stdin --tty service-a-f5757c648-9gqt8  -- sh
# curl google.com
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
# curl 10.36.0.27
Hello from Service B!# curl 10.36.0.28
Hello from Service C!#
```

Before applying the isolation policy to Service A. It shows that Service A can freely communicate with other services in the Kubernetes cluster, as well as with external websites like Google. This means there are no network policies restricting the ingress or egress traffic for Service A at this stage.

```
User@LAPTOP-T5IBK3AO MINGW64 ~/OneDrive/Desktop/k8s-config-app
$ kubectl get networkpolicy
NAME               POD-SELECTOR    AGE
isolate-service-a  app=service-a   37s
```

The terminal output shows the result of applying an isolation policy to Service A using the Configuration application. This confirms that Service A has been successfully isolated.

```
User@LAPTOP-T5IBK3AO MINGW64 ~/OneDrive/Desktop/k8s-config-app
$ kubectl exec --stdin --tty service-a-f5757c648-9gqt8  -- sh
# curl google.com
curl: (6) Could not resolve host: google.com
# curl 10.36.0.27
curl: (28) Failed to connect to 10.36.0.27 port 80 after 130353
 ms: Couldn't connect to server
# curl 10.36.0.28
curl: (28) Failed to connect to 10.36.0.28 port 80 after 131561
 ms: Couldn't connect to server
#
```

After applying an isolation policy at Service A. The isolation policy restricts Service A's ability to communicate with both external websites and other services within the Kubernetes cluster. This confirms that the isolation policy is effectively blocking all network traffic to and from Service A.

```
User@LAPTOP-T5IBK3AO MINGW64 ~/OneDrive/Desktop/k8s-config-app
$ kubectl exec --stdin --tty service-b-796f77f9df-qhgj2  -- sh
# curl 10.36.0.28
Hello from Service C!# curl 10.36.0.58
curl: (28) Failed to connect to 10.36.0.58 port 80 after 128728 ms: Coul
dn't connect to server
#
```

```
User@LAPTOP-T5IBK3AO MINGW64 ~/OneDrive/Desktop/k8s-config-app
$ kubectl exec --stdin --tty service-c-86877bdd98-7fk7d  -- sh
# curl 10.36.0.27
Hello from Service B!# curl 10.36.0.58
curl: (28) Failed to connect to 10.36.0.58 port 80 after 129682 ms: Couldn'
t connect to server
#
```

After applying specific network policies. The results show that Service B and Service C can communicate with each other but are unable to communicate with Service A due to the applied isolation policy on Service A.

Ingress Traffic Control

Table 2 summarizes the ingress communication capabilities between Service A, Service B, and Service C before and after applying an ingress-specific policy to Service A. It shows how the ingress-specific policy affects the ability of Service B and Service C to interact with Service A.

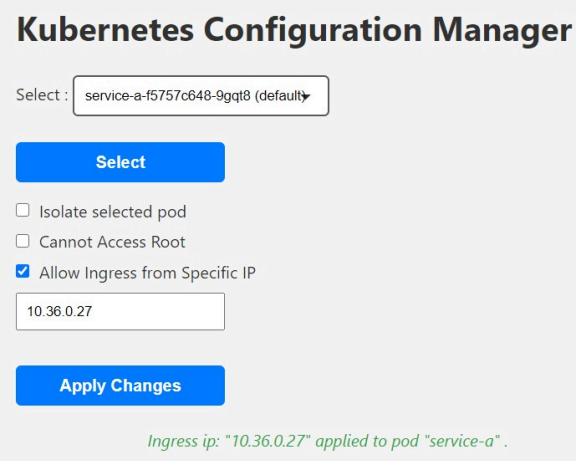
TABLE 2. Result of Ingress Traffic Control

Communication Test	From	To	Result Before Set Ingress	Result After Set Ingress
Ingress Communication	Service B	Service A	Success	Success
Ingress Communication	Service C	Service A	Success	Failed

```
User@LAPTOP-T5IBK3AO MINGW64 ~/OneDrive/Desktop/k8s-config-app
$ kubectl exec --stdin --tty service-b-796f77f9df-qhgj2 -- sh
# curl 10.36.0.58
Hello from Service A!#
```

```
User@LAPTOP-T5IBK3AO MINGW64 ~/OneDrive/Desktop/k8s-config-app
$ kubectl exec --stdin --tty service-c-86877bdd98-7fk7d -- sh
# curl 10.36.0.58
Hello from Service A!#
```

Before applying an ingress-specific policy to Service A, other services in the Kubernetes cluster, such as Service B and Service C, can freely communicate with Service A. This means that pods can send and receive network requests without any restrictions.




```
User@LAPTOP-T5IBK3AO MINGW64 ~/OneDrive/Desktop/k8s-config-app
$ kubectl describe networkpolicy isolate-service-a
Name:         isolate-service-a
Namespace:    default
Created on:   2024-05-19 21:33:09 +0700 +07
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     app=service-a
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      IPBlock:
        CIDR: 10.36.0.27/32
        Except:
  Allowing egress traffic:
    <none> (Selected pods are isolated for egress connectivity)
  Policy Types: Ingress, Egress
```

After applying the ingress-specific policy to Service A, only Service B is allowed to communicate with Service A. This ensures that Service A is protected from any unauthorized access from other services within the Kubernetes cluster. The provided terminal output demonstrates that Service B can still communicate with Service A, confirming that the ingress-specific policy is functioning correctly.

The provided terminal output demonstrates after applying an ingress-specific policy that allows only Service B to communicate with Service A. This means that Service C cannot communicate with Service A, while Service B can still successfully communicate with it.

5.Conclusions

This project presents a comprehensive solution for enhancing the security of Kubernetes deployments through automated pod configurations. Our system introduces granular isolation measures at the container level and ingress-specific IP controls. These features collectively establish a robust security posture, safeguarding Kubernetes clusters from internal and external threats.

Through our implementation and experiments, we've demonstrated the system's ability to effectively configure pods, manage security settings, and ensure compliance with industry standards. By integrating these security mechanisms, the project contributes to securing Kubernetes deployments, providing a secure, scalable, and reliable environment for containerized applications.

Future work will explore further enhancements, such as integrating additional security measures, refining dynamic RBAC policies, optimizing performance, encrypting the data in transit and ensuring continued protection for Kubernetes deployments.

Reference

[1] S. Brohi, “Implementing a Secure-First Pod Security Policy Architecture | by Shazad Brohi | Nov, 2020 | Medium | ITNEXT,” *Medium*, Dec. 16, 2021. https://itnext.io/implementing-a-restricted-first-pod-security-policy architecture-af4e906593b0

[2] “Implementing Pod Security Standards in Amazon EKS | Amazon Web Services,” *Amazon Web Services*, Oct. 27, 2022. https://aws.amazon.com/blogs/containers/implementing-pod-security-standards-in-amazon-eks/

[3] “Role-Based Access Control (RBAC) Authorization in Kubernetes,” *River Publishers Journals & Magazine | IEEE Xplore*, 2023. https://ieeexplore.ieee.org/document/10255393

[4] “Network Policies in Kubernetes: Performance Evaluation and Security Analysis,” *IEEE Conference Publication | IEEE Xplore*, Jun. 08, 2021. https://ieeexplore.ieee.org/document/9482526

[5] “Kubernetes Authorization With RBAC,” *part of Cloud Native Security | Wiley Data and Cybersecurity books | IEEE Xplore*. https://ieeexplore.ieee.org/document/9932381

[6] Schaffererin, “Developer best practices - Pod security in Azure Kubernetes Services (AKS) - Azure Kubernetes Service,” *Microsoft Learn*, Jan. 12, 2024. https://learn.microsoft.com/en-us/azure/aks/developer-best-practices-pod-security

[7] W. L. Dang, “Guide to Kubernetes Security Context & Pod Security Policy (PSP),” Mar. 31, 2021. https://www.redhat.com/en/blog/guide-to-kubernetes-security-context-pod-security-policy-psp

[8] M. Levan, “4 Methods Of Kubernetes Isolation,” *Packet Pushers Interactive LLC*, Jan. 26, 2024. https://packetpushers.net/blog/4-methods-of-kubernetes-isolation/

[9] “Introducing security groups for pods | Amazon Web Services,” *Amazon Web Services*, Jan. 28, 2021. https://aws.amazon.com/blogs/containers/introducing-security-groups-for-pods/

[10] S. Berkovich and R. Lipowitch, “Key takeaways from the Wiz 2023 Kubernetes Security Report,” *wiz.io*, Nov. 08, 2023. https://www.wiz.io/blog/key-takeaways-from-the-wiz-2023-kubernetes-security-report

[11] “Kubernetes Security Best Practices: 10 Steps to Securing K8s,” *Aqua*, Jan. 16, 2024. https://www.aquasec.com/cloud-native-academy/kubernetes-in-production/kubernetes-security-best-practices-10-steps-to-securing-k8s/

[12]“‘Under-reported’ Security Defects in Kubernetes Manifests,” *IEEE Conference Publication | IEEE Xplore*, Jun. 01, 2021. https://ieeexplore.ieee.org/document/9476056

[13] B. Hirschberg, “Kubernetes Security Best Practices for Security Professionals,” *ARMO*, Apr. 07, 2024. https://www.armosec.io/blog/kubernetes-security-best-practices/

[14] K. Jackson, “Kubernetes Security Monitoring in 2024: Tools, Techniques, & Trends,” *OpenStack Backup and Recovery | Kubernetes Backup and Recovery*, Apr. 24, 2024. https://trilio.io/resources/kubernetes-security-monitoring/

[15] M. Aslaner, “Defending Cloud-Based Workloads: A Guide to Kubernetes Security,” *SentinelOne*, Apr. 03, 2024. https://www.sentinelone.com/blog/defending-modern-cloud-based-workloads-a-guide-to-kubernetes-security/