Breaking the Bulkhead: Demystifying Cross-Namespace Reference Vulnerabilities in Kubernetes Operators

Andong Chen*, Ziyi Guo†¶, Zhaoxuan Jin†, Zhenyuan Li*¶ and Yan Chen†
*Zhejiang University †Northwestern University
chenandong@zju.edu.cn, {n7l8m4, zhaoxuanjin2025}@u.northwestern.edu,
lizhenyuan@zju.edu,cn, ychen@northwestern.edu

***Abstract*—Kubernetes Operators, automated tools designed to manage application lifecycles within Kubernetes clusters, extend the functionalities of Kubernetes, and reduce the operational burden on human engineers. While Operators significantly simplify DevOps workflows, they introduce new security risks. In particular, Kubernetes enforces namespace isolation to separate workloads and limit user access, ensuring that users can only interact with resources within their authorized namespaces. However, Kubernetes Operators often demand elevated privileges and may interact with resources across multiple namespaces. This introduces a new class of vulnerabilities, the Cross-Namespace Reference Vulnerability. The root cause lies in the mismatch between the declared scope of resources and the implemented scope of the Operator's logic, resulting in Kubernetes being unable to properly isolate the namespace. Leveraging such vulnerability, an adversary with limited access to a single authorized namespace may exploit the Operator to perform operations affecting other unauthorized namespaces, causing Privilege Escalation and further impacts.**

To the best of our knowledge, this paper is the first to systematically investigate Kubernetes Operator attacks. We present Cross-Namespace Reference Vulnerability with two strategies, demonstrating how an attacker can bypass namespace isolation. Through large-scale measurements, we found that over 14% of Operators in the wild are potentially vulnerable. Our findings have been reported to the relevant developers, resulting in 8 confirmations and 7 CVEs by the time of submission, affecting vendors including the inventor of Kubernetes - Google and the inventor of Operator - Red Hat, highlighting the critical need for enhanced security practices in Kubernetes Operators. To mitigate it, we open-source the static analysis suite and propose concrete mitigation to benefit the ecosystem.

I. Introduction

Kubernetes has emerged as the dominant platform for container orchestration, playing a central role in the deployment, scaling, and management of containerized applications in modern cloud-native environments [1, 2, 3, 4]. As a highly extensible and open-source system, Kubernetes facilitates the automation of complex operations such as container deployment, scheduling, and management across clusters. Its flexibility and wide adoption have made it the cornerstone of many enterprise-level infrastructure solutions, offering efficient ways to handle diverse and dynamic workloads in a scalable manner.

Kubernetes organizes resources into namespaces [5], which allow users to divide a single cluster into multiple virtual clusters. Each namespace serves as a logical boundary, isolating resources like pods, services, and secrets from other namespaces within the same cluster. This isolation is essential for managing different applications or services within the same Kubernetes environment, enabling users to work independently without interfering with each other. Namespaces also provide a way to scope access to resources, ensuring that certain actions can be confined to specific namespaces and reducing the risk of accidental or malicious interference between services.

To achieve namespace isolation, a crucial security mechanism is Role-Based Access Control (RBAC) [6]. RBAC defines roles and permissions for users, service accounts, and other entities within the cluster, helping to control which actions are allowed within the system. For example, a staff member of a team may only be allowed to manipulate resources within the namespace assigned to their team, while the cluster administrator would be assigned all permissions across namespaces. By configuring RBAC policies, administrators can limit access to resources within specific namespaces, ensuring that users or services can only interact with the resources they are authorized to access. This granularity of access control reinforces the isolation between namespaces and helps to prevent unauthorized access to sensitive resources.

While Kubernetes provides robust tools and security mechanisms to manage and secure applications, the native platform has limitations in automating the lifecycle of complex applications. Kubernetes requires significant manual intervention for tasks like scaling, upgrades, and configuration management, which can be time-consuming and error-prone [7]. Kubernetes Operators [8] were introduced to address these limitations. Operators are programs that extend Kubernetes' capabilities by automating the management of applications. They encapsulate the operational knowledge required to manage complex Kubernetes applications, automating critical tasks such as deployment, scaling, and lifecycle management. Users can then easily request Operators to conduct complex operational tasks in contrast to manually manipulating raw Kubernetes

¶ Corresponding Authors: Ziyi Guo, Zhenyuan Li

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240761
www.ndss-symposium.org

resources. By automating these processes, Operators reduce the operational burden on DevOps teams and enable more consistent and reliable application management.

However, Kubernetes Operators require significant privileges to carry out their tasks. Due to the broad range of operations they need to perform, these Operators are usually granted substantial permissions across namespaces. While these permissions are necessary for the proper functioning of the Operator, they also introduce a significant security risk. Due to improper security practices in Operator implementation, adversaries may forge malicious requests toward Operators, exploit vulnerabilities to escalate their own permissions, break namespace isolation, and perform unauthorized operations within the cluster.

Although extensive research has been conducted on Kubernetes security, Operator-specific attacks remain largely unexplored. Previous studies have focused on misconfiguration in Kubernetes, especially excessive RBAC permissions, highlighting the risks of overly permissive access controls [9, 10, 11, 12]. Practitioners thus suggested limiting RBAC permissions for Kubernetes Operators [13, 14, 15]. While these works have led to improvements in reducing permissions, they have neither addressed the inherent risks that remain even when permissions are minimized, nor presented attacks specific to Operators. Specifically, necessary permissions required by Operator business logics, which cannot be further minimized, may still be exploited due to improper security design within the Operator logic. Furthermore, existing attacks assume attackers have compromised containers (e.g., get a shell) in prior, leaving critical gaps in how to compromise application containers.

Other research has focused on bugs in Kubernetes Operators [16, 17, 18], yet these studies primarily concentrated on the functional bugs of Operators rather than their security vulnerabilities. Furthermore, existing security tools [19, 20, 21, 22, 23, 24] do not adequately address the security concerns specific to Operators, leaving a significant gap in the ecosystem.

Thus, in this paper, we present the first systematic research on Kubernetes Operator attacks, unveiling cross-namespace reference vulnerabilities. The root cause of cross-namespace reference vulnerability lies in the mismatch between the declared scope of a resource and the effective scope of the Operator's logic. A resource may be declared as namespace-scoped, allowing users with limited access to a single namespace to deploy it, while the Operator's logic may perform actions that affect other namespaces, breaking the intended isolation. We present a novel practical threat model without assuming that attackers have already compromised containers, allowing exploits to be made from scratch. We propose two distinct tactics for exploiting Kubernetes Operators to elevate an attacker's privileges, both of which exploit the scope mismatch in Operator implementation and break the isolation between Kubernetes namespaces.

To measure the new kind of vulnerabilities, we designed and implemented a static analysis suite that can identify scope mismatch in Operators. We conducted large-scale measurements of 2,268 Kubernetes Operators in the wild, revealing that over 14% of the Operators are potentially vulnerable to these attacks. We responsibly disclosed our findings to their developers, and, by the time of submission, 8 vulnerabilities had been confirmed and 7 CVEs were assigned or under assignment in response to our reports, affecting vendors including the inventor of Kubernetes - Google and the inventor of Operator - Red Hat, highlighting the critical need for enhanced security practices in Kubernetes Operators.

All in all, our contributions can be summarized as follows:

- **New Attack**. We present the first systematic research on Kubernetes Operator attacks. We unveil the new type of vulnerability specific to Operators, Cross-Namespace Reference Vulnerability, detailing two distinct tactics and their root cause, both of which can be leveraged to escalate privileges.
- **Large-scale Measurement**. We design and implement tools for the measurement of real-world Cross-Namespace Reference Vulnerabilities. We conduct large-scale measurements of Kubernetes Operators in the wild, demonstrating that over 14% of Operators are susceptible to these vulnerabilities.
- **Real-World Impact**. We responsibly disclosed our findings to the developers, and, by the time of submission, 8 vulnerabilities were confirmed and 7 CVEs were assigned in response to our reports, affecting leading vendors including Google and Red Hat.
- **Benefic Ecosystem**. We open-source our analyzer, which covers detailed modeling of major Kubernetes libraries, enabling analysis of not only Operators but also other Kubernetes applications. To facilitate remediation, we propose concrete mitigations, open-source mitigation samples, and a patch generator for practitioners. Artifacts can be found at https://doi.org/10.17605/OSF.IO/PWVC4.

II. Background

A. Kubernetes Namespace and RBAC

Kubernetes is a powerful container orchestration platform that automates the deployment, scaling, and management of containerized applications. It is designed to manage large-scale complex applications, where multiple teams or applications may share a single cluster. To help organize and isolate resources within the cluster, Kubernetes provides a mechanism called *Namespaces* [5]. A namespace[1] is a logical partition or a virtual cluster within a physical cluster. Each namespace acts as a boundary, ensuring that resources in one namespace do not conflict with those in another.

Namespaces are particularly useful in multi-tenant environments, where different teams or applications share the same Kubernetes cluster [25]. By isolating resources in separate namespaces, Kubernetes prevents one team from accessing or interfering with another team's resources. This isolation is vital for security and resource management, ensuring that users and applications can only access the resources assigned to

[1]We use ns as short for namespace for the remaining parts.

their namespace, preventing unauthorized access or potential conflicts between resources.

Kubernetes employs multiple security mechanisms to help ensure that the cluster remains secure and that resources are properly isolated. One of the most important mechanisms for securing access to resources within a namespace is RBAC (Role-Based Access Control) [6]. RBAC allows administrators to define roles with specific permissions and bind those roles to users or service accounts, ensuring that only authorized entities can perform certain actions. Cluster administrators can grant both ns-specific permissions and cluster-level permissions. Specifically:

- *Role* and *RoleBinding*: Define and grant resource permissions within a specific namespace to a user, group, or service account.
- *ClusterRole* and *ClusterRoleBinding*: Define and grant resource permissions across all namespaces to a user, group, or service account.

B. Kubernetes Resource

At its core, Kubernetes organizes the cluster's state using resources, which are data objects encapsulating configuration and runtime information. Kubernetes manages many types of resources within a cluster, which are fundamental components that define the desired state of applications and services. Common built-in resource types include pods [26] and deployments [27]. A pod is the smallest deployable unit in Kubernetes and typically represents one or more containers that share the same network and storage resources. A deployment is a higher-level abstraction that manages the lifecycle of pods, specifying the desired number of pod replicas. In addition to built-in resources, Kubernetes allows users to define *Custom Resources* (CRs) [28], extending Kubernetes to manage domain-specific requirements beyond its default capabilities. Each type of Custom Resource is described by a Custom Resource Definition (CRD) [28], which specifies the resource's schema.

In Kubernetes, each type of resource, whether a built-in resource or a Custom Resource, is bound with an explicit scope, indicating its accessibility and impact within the cluster. Resources can be either *Namespace-scoped* or *Cluster-scoped*. In Kubernetes, the scope of built-in resources is embedded within the Kubernetes implementation, whereas the scope of Custom Resources is explicitly defined in their associated Custom Resource Definitions. NS-scoped resources must reside within a specific namespace, meaning they are logically isolated and can be accessed or manipulated by users with only NS-specific Roles. Conversely, Cluster-scoped resources exist at the cluster level and are not confined to any single namespace. These Cluster-scoped resources may affect or interact with all namespaces across the cluster. Due to their broad impact, accessing or manipulating cluster-scoped resources requires users to possess a cluster-wide ClusterRole, reflecting elevated privileges.

Importantly, resources themselves merely represent desired configurations or states. To realize these desired states, each type of resource is managed by an associated controller, a program responsible for monitoring resources and taking actions to align the actual state with the desired state described by resources. For example, the *deployment controller* monitors *deployment* resources and ensures that the desired number of pod replicas are running. If a pod fails or is deleted, the *deployment controller* automatically creates a new pod to meet the desired state. For built-in resources, Kubernetes provides native controllers. For Custom Resources, users should develop custom controllers.

```yaml
apiVersion: example.com/v1
kind: DatabaseInstance
metadata:
  name: my-database
spec:
  replicas: 3
  storageSize: "10Gi"
```

Fig. 1. Custom Resource Example

C. Kubernetes Operator

Kubernetes Operator is a method of automating and managing the lifecycle of complex applications on top of Kubernetes by extending the platform's native capabilities. Originally introduced by CoreOS (now part of Red Hat), it emerged from a recognition that while Kubernetes excels at automatically orchestrating workloads, many organizations need a more powerful automation pattern to handle full lifecycle management, such as database management, application upgrades, or failure recovery, that requires specific operational knowledge. Operator is thus introduced to extend Kubernetes by embedding human operational expertise into software, enabling automated management of complex, stateful applications.

An Operator consists of one or more Custom Resource Definitions (CRDs) and their corresponding Custom Resource Controllers. CRD defines the schema of a custom resource type that will be processed by the Operator. Controllers work with CRDs by continually monitoring custom resources and taking actions to fulfill operational tasks requested by users.

To use an Operator, users manipulate custom resources that represent the operational task, along with the related parameters they want to conduct. The Operator controller reads these custom resources, takes actions listed in the custom resource, and ensures that the operational task behaves as expected. Considering an Operator for database management tasks, users may create a custom resource listed in Figure 1, including arguments like the number of database replicas and storage settings. The Operator controller then reads the custom resource, automatically provisions, scales, and maintains the database according to these specifications.

Since Operators typically manage multiple kinds of resources across namespaces, they often run with elevated RBAC privileges, allowing them to create, modify, and delete resources on behalf of users. This makes them powerful but also

introduces security risks. If an Operator does not adopt proper security measures, attackers may exploit the vulnerabilities of the Operator to gain unauthorized access or manipulate resources beyond their intended scope.

III. Threat Model

Our threat model aligns with real-world Kubernetes deployments where multiple tenants, teams, or applications share the same cluster while being isolated within their respective namespaces [25]. The adversary aims to break Kubernetes namespace isolation and achieve cross-ns privilege escalation by exploiting security weaknesses in Operator implementations. Their objectives are performing operations in unauthorized namespaces (i.e., namespaces that they have no *Roles*) and thus escalating privileges.

We assume the Kubernetes cluster deploys vulnerable Operators, and the adversary has legitimate access to a Kubernetes cluster but can only access their authorized namespaces. Thus, they cannot access or manipulate cluster-scoped resources and can only interact with Operators by manipulating ns-scoped resources in their authorized namespace. They seek to leverage vulnerable Kubernetes Operators to execute unauthorized operations in other namespaces. The adversary may be:

- A malicious tenant in a multi-tenant cluster who is only authorized to access their assigned namespace.
- A compromised application running in a namespace with ns-level permissions mounted.
- An attacker who steals credentials of Kubernetes accounts with ns-level permissions.

Our threat model is practical. For example, many real-world cloud services are operated on multi-tenant Kubernetes. One of them, Red Hat Developer Sandbox [29], is provided by assigning a namespace on multi-tenant Kubernetes to a user, where Operators are deployed. This implies that attackers may subscribe Kubernetes-based service to conduct our attacks.

It is notable that our threat model is significantly different from previous works [9, 10, 11, 12]. Specifically, existing works assume that the adversaries have compromised vulnerable application containers in prior, which is a strong assumption in the real world, leaving critical gaps in how to compromise application containers. In contrast, within our threat model, adversaries don't have to gain control of Operators in prior, since these Operators may be deployed in adversary-unauthorized namespaces. In extreme cases, Operators can even be deployed outside the Kubernetes cluster [30]. So the threat model of previous works is relatively infeasible, but our threat model is more feasible and aligned with real-world scenarios.

In this paper, the terms *Namespace* and *Cross-Namespace* refer specifically to Kubernetes Namespaces, which are used to isolate resources within a Kubernetes cluster. They are distinct from similarly named concepts in other systems, such as Linux Namespaces, which isolate system resources at the OS kernel level. They isolate resources at different layers and have no direct linkage despite sharing the same term.

The vulnerability we presented acts as a strategy for attackers to access or manipulate unauthorized resources. The final concrete impact depends on the accessed unauthorized resources and differs between cases.

IV. Cross-Namespace Attacks

A. Attack Overview

In Kubernetes clusters, namespaces act as virtual boundaries, restricting user access and isolating resources. Kubernetes Operators manage applications and resources and perform essential operational tasks. While these Operators simplify application management, their inherent privileges and operational flexibility create potential security vulnerabilities that can be exploited for cross-namespace reference attacks.

The high-level attack flow is as follows: an attacker, who has legitimate but restricted access to one namespace, manipulates a maliciously crafted ns-scoped resource instance within their authorized namespace. The Operator, continuously watching for ns-scoped resource events, detects this malicious resource instance and processes it with privileged operations that impact namespaces beyond the attacker's authorized scope, effectively breaking the intended namespace isolation enforced by Kubernetes.

Root Cause. The core enabling cross-ns reference attacks stems from a mismatch between the declared scope of a resource and the actual scope of its process logic. Specifically, the vulnerability arises when the scope of a resource is defined as *Namespaced*, indicating that each instance should strictly reside within its assigned namespace. Thus, an adversary only with *Role* in a single namespace is allowed by RBAC to manipulate such a resource in their own namespace. However, despite this ns-scoped definition, the Operator may actually perform operations across namespaces, inadvertently allowing manipulation of resources in namespaces beyond the intended scope. As a result, an adversary without *Role* in other namespaces may invoke the Operator to escalate their permissions and access unauthorized namespaces.

The root leading to such implementation lies in the improper trade-off at the design level between security and convenience. Kubernetes grants significant flexibility for controllers. Built-in controllers in Kubernetes prioritize security to eliminate attack surfaces. For instance, a Pod can only reference Secrets within the same namespace to avoid cross-ns. However, many developers [31, 32, 33, 34, 35] want cross-ns to avoid manually duplicating Secrets in each namespace. Since Operators are proposed to reduce human intervention, the community may prioritize convenience, overlooking the underlying Kubernetes's security model and enabling cross-ns reference.

Cross-Namespace Features. There are two primary scenarios that enable cross-namespace reference actions. First, when processing ns-scoped resources in one namespace, an Operator may access or manipulate other ns-scoped resources in a different namespace (§IV-B). Second, when processing ns-scoped resources, an Operator might access or manipulate cluster-scoped resources, leading to impacts on the whole cluster across all namespaces (§IV-C). Both scenarios allow

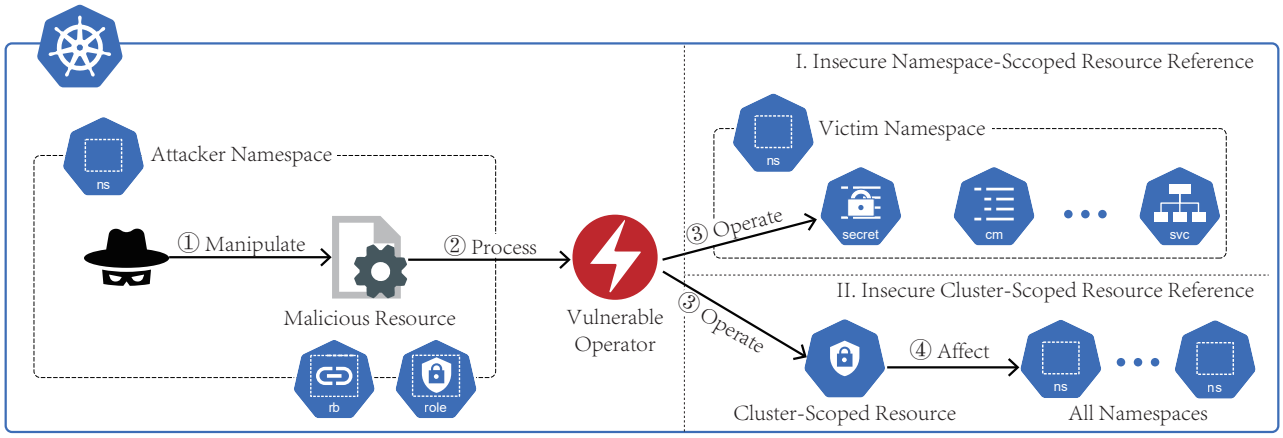

Fig. 2. Attack Flow

adversaries to trick the Operator into performing unintended, privileged operations beyond the adversary's RBAC scope.

B. Insecure Namespace-Scoped Resource Reference

Insecure NS-Scoped Resource Reference vulnerability arises when an Operator processing ns-scoped resources, the fields of which are then used by the Operator to reference resources in other namespaces. This vulnerability fundamentally undermines Kubernetes namespace isolation by enabling attackers to indirectly access resources from namespaces they are otherwise restricted from accessing.

Attack Flow. As illustrated in Figure 2, consider two namespaces: an attacker namespace and a victim namespace containing sensitive resources. The Roles and RoleBindings claim that the attacker can only access resources in their namespace and cannot access those in the victim's namespace.

To bypass the restriction of RBAC and access resources in unauthorized namespaces, the attacker first crafts and deploys a malicious resource instance within their namespace. This resource includes fields leveraged by the Operator to reference resources located in the victim namespace. From the perspective of Kubernetes, the deployment of the malicious resource should be allowed because it only knows that the attacker has created a resource under their authorized namespace, but does not know if the resource leads to privilege escalation.

The Operator then processes the created malicious resource. The Operator extracts the fields in the malicious resource, operates the victim resource located in the victim namespace, and inadvertently leaks or tampers with sensitive information. Thus, the attacker effectively escalates their privileges, bypassing Kubernetes' namespace isolation, gaining unauthorized access to resources that should have remained secure.

Example. A common real-world scenario occurs when an Operator manages applications consuming credentials (e.g., API Secret Key) stored in Kubernetes Secrets. As per Kubernetes official security practice [36], Secrets should only be referenced strictly within the same namespace to maintain proper isolation.

```yaml
1  apiVersion: example.com/v1
2  kind: Example
3  metadata:
4    name: malicious
5    # Deploy resource in attacker's namespace
6    namespace: attacker
7  spec:
8    secretRef:
9      name: sensitive-secret
10     # Reference secret in victim's namespace
11     namespace: victim
```

```go
1  func (r *Example) Reconcile(...) (...) {
2      // secretRef is attacker-controlled
3      secretRef := example.Spec.SecretRef
4
5      secret := &corev1.Secret{}
6      namespacedName := types.NamespacedName{
7          Name:      secretRef.Name,
8          // Cross-namespace Reference
9          Namespace: secretRef.Namespace
10     }
11
12     // Retrieve the secret
13     err := r.Get(ctx, namespacedName, secret)
14
15     // Further operation using secret
16 }
```

Fig. 3. Insecure Namespace-Scoped Resource Reference Sample

However, the vulnerable Operator implements cross-namespace references by setting up a *secretRef.namespace* field in its custom resource definition. Given this insecure implementation, an attacker restricted to a namespace could craft the malicious custom resource as illustrated in the upper YAML file of Figure 3. He deploys a resource with *metadata.namespace* setting to *attacker*, which means the resource is deployed in the *attacker* namespace. This deployment is allowed since the RBAC authorized the attacker to work in his own namespace. In the specification of the resource, the attacker defines the value of *secretRef* at Lines 8-11, referencing a Secret named *sensitive-secret* in his unauthorized namespace *victim*.

The Operator notices the malicious resource deployed by the attacker, reads the *secretRef* field at Line 3 of the Reconcile function illustrated in Figure 3. The name and namespace of the referenced victim Secret are then loaded into the *namespacedName* object, which is used to query and retrieve the specified Secret into the *secret* object at Line 13. The remaining parts of the Operator will consume the content of the Secret to perform further operations.

Impact. Insecure NS-Scoped Resource Reference vulnerabilities fundamentally enable attackers to escalate privileges by allowing them to reference and manipulate resources in namespaces beyond their legitimate access. Further impact of this vulnerability heavily depends on how the Operator processes and utilizes the referenced resources, as well as the nature of the referenced resources themselves. For instance, if the referenced resource is a Kubernetes Secret containing sensitive credentials like API Tokens, an attacker may obtain unauthorized access to applications, databases, or cloud infrastructure. If the Operator not only reads but also modifies referenced resources, attackers might disrupt service availability, modify application configurations, or inject malicious workloads. Thus, the severity and scope of the impact are highly context-dependent, ranging from sensitive information leakage to complete cluster compromise, based on the type and usage of the improperly referenced resource.

C. Insecure Cluster-Scoped Resource Reference

Insecure Cluster-Scoped Resource Reference occurs when a Kubernetes Operator processes a ns-scoped resource and interacts with cluster-scoped resources. Unlike ns-scoped resources that remain isolated within specific namespaces, cluster-scoped resources affect the entire Kubernetes cluster. If an Operator allows users to influence these cluster-scoped resources through ns-scoped resources, it creates a pathway for attackers to escalate privileges and potentially compromise the entire cluster.

Attack Flow. As illustrated in Figure 2, consider a namespace controlled by an attacker named *attacker*, and all the other victim namespaces. The Roles and RoleBindings in the cluster define that the attacker can only access resources in their namespace and cannot access any victim's namespace.

The basic attack workflow for this vulnerability starts with an attacker creating a malicious resource in their authorized namespace, whose fields are leveraged by the Operator to reference a cluster-scoped resource. Kubernetes RBAC accepts the deployment of the malicious resource because it only knows that the attacker has created a resource under their authorized namespace, but does not know if the resource leads to privilege escalation. The Operator, running with elevated cluster-level privileges, processes this malicious input and subsequently performs operations on the referenced cluster-scoped resource. As cluster-scoped resources inherently affect the entire Kubernetes environment, these unauthorized accesses and manipulations enable attackers to escalate privileges beyond their initial namespace boundaries and gain control or influence over all the other namespaces.

```yaml
1  apiVersion: example.com/v1
2  kind: App
3  metadata:
4    name: malicious
5    # Namespace-scoped Resource
6    namespace: attacker
7  spec:
8    # ...
```

```
1  func (r *App) Reconcile(...) (...) {
2      crb := &rbacv1.ClusterRoleBinding{
3          Subjects: []rbacv1.Subject{{
4              Kind:      "ServiceAccount",
5              Name:      app.name,
6              // Attacker-controlled input
7              Namespace: app.Namespace
8          }}
9      }
10     // Create CRB to attacker's namespace
11     r.Create(ctx, crb)
12 }
```

Fig. 4. Insecure Cluster-Scoped Resource Reference Sample

Example. ClusterRole and ClusterRoleBinding are two cluster-scoped built-in resources in Kubernetes. It can grant cluster-wide permissions to a ServiceAccount, a built-in namespace-scoped resource that provides an identity for applications to access the Kubernetes API. Some Kubernetes applications require mounting a ServiceAccount with cluster-wide permissions to access the Kubernetes API and operate correctly. Thus, when deploying such an application in a specified namespace, their Operators will create a ServiceAccount in that namespace, mount it on the application, and create a ClusterRoleBinding to grant cluster-wide permission to that ServiceAccount. An insecure implementation occurs when an Operator accepts ns-scoped resources and creates a ClusterRoleBinding to a ServiceAccount in the requesting namespace. An attacker restricted to a namespace could thus craft the malicious resource illustrated in the YAML file of Figure 4. He deploys a resource with *metadata.namespace* setting to *attacker*, which means the resource is deployed in the *attacker* namespace.

The Operator monitors *App* custom resources. It finds the malicious resource deployed by the attacker, creates a ServiceAccount in the attacker's namespace, and then creates a ClusterRoleBinding towards that ServiceAccount to grant it cluster-wide permissions. Since the ServiceAccount is created in the attacker's namespace, the attacker can impersonate the ServiceAccount to escalate his privilege, gaining cluster-wide permissions granted by ClusterRoleBinding. In short, such a vulnerability can be leveraged to directly elevate the permissions of attackers.

Impact. Insecure Cluster-Scoped Resource References allow attackers to escalate privileges and affect resources across all namespaces in a cluster. The specific severity and effect of this vulnerability depend on which cluster-scoped resources the Operator interacts with. For instance, if an Operator

insecurely creates a ClusterRole or ClusterRoleBinding as dictated by ns-scoped resources, an attacker can gain cluster-level permissions. The detailed permissions assigned depend on the implementation of Operators.

V. Cross-Namespace In The Wild

To assess the prevalence of the vulnerabilities described in Section IV, we conducted a large-scale measurement of real-world Kubernetes Operators and disclosed our findings to affected vendors.

A. Measurement Methodology

1) Overview: To understand how widespread the vulnerabilities are in real-world Kubernetes Operators, we perform a systematic measurement illustrated in Figure 5, which consists of the following steps:

1) Operator Collection: A large set of publicly available Kubernetes Operator repositories is collected from GitHub.
2) Resource Type Identification: Resource types (either Kubernetes built-in resources or custom resources) used by each Operator are extracted, and their declared scopes (either ns-scoped or cluster-scoped) are identified based on the code.
3) Vulnerability Detection: The analysis identifies whether Operators process ns-scoped resources but conduct insecure cross-ns reference behavior, as depicted in Section IV-B and Section IV-C.
4) Summary: Identified vulnerabilities are further aggregated based on the types of referenced resources and operation verbs to evaluate the impacts in the real world.

Among all Kubernetes development frameworks recommended by the Kubernetes official [8], Golang frameworks (i.e. Kubebuilder [37] and Operator SDK [38]) own the highest GitHub Stars and dominates with around 2.4k Operators, followed by Python (331 Operators), Shell (167 Operators), Java (134 Operators), Rust (96 Operators), and .NET (29 Operators). Given the overwhelming amount of Golang-based Operators, the collection specifically targets Operators implemented in Golang. We adopted CodeQL [39] v2.17.4 to analyze Operators. The entire CodeQL query suite uses around 1,500 lines of QL rules, covering detailed modeling of major Kubernetes Go libraries, enabling analysis of Kubernetes applications beyond Operators.

TABLE I
Common Operator-Related Libraries

Library	Description
k8s.io/api [40]	K8s Built-In Resource Specifications
k8s.io/apimachinery [41]	K8s Metadata Specifications
client-go [42]	K8s Official Client
client-gen [43]	K8s Official Client Generator
controller-runtime [44]	Controller Client

To enhance the measurement process, 5 commonly used libraries listed in Table I were modeled to accurately resolve and track Kubernetes interactions within collected Operator implementations. They contain specifications of native Kubernetes resources, namespace-related data structures, and functions for Operators to manipulate resources. The detail is elaborated later.

2) Operator Collection: The dataset of Kubernetes Operators analyzed was collected by crawling GitHub repositories. To achieve this, GitHub Search API [45] was utilized with the query string *"Kubernetes Operator language: go"*. The collection process strictly adhered to GitHub's API usage policies to responsibly retrieve relevant Operator repositories.

After collecting Operators from GitHub, we set up CodeQL databases for each Operator. 13 Operators that cannot be compiled to generate the CodeQL database due to errors, like syntax and dependency errors, are eliminated, and the final set contains 2,268 Operators.

3) Resource Type Identification: The first critical step in detecting vulnerabilities is Resource Type Identification, as the attack requires the attacker to initiate operations using a ns-scoped resource they are authorized to create in their namespace. The analysis separately handles Kubernetes Built-in Resources and Operator-defined Custom Resources.

Custom Resource Identification. For Custom Resources defined by the Operators, their data structures can be explicitly extracted from the source code. In Kubernetes, each resource structure must contain a field of type TypeMeta (defined by the *Apimachinery* library [41]), which acts as the unique identifier of a resource type. Thus, the analyzer extracts all struct types in Operators and filters those with TypeMeta fields. This outputs all custom resource types in Operators.

To further identify the scope of each resource type (ns or cluster), common Kubernetes frameworks like Kubebuilder [37] and Operator SDK [38], as well as Kubernetes' official client code generator [43], require developers to explicitly decorate cluster-scoped resource structs using special marker annotations "+*genclient:nonNamespaced*" or "+*kubebuilder:resource:scope:Cluster*". By detecting these markers in the Custom Resource struct definitions, the analyzer reliably identifies Custom Resource types in operators and their scopes.

Kubernetes Built-in Resources. Unlike Custom Resources, the built-in Kubernetes resource specifications are imported from the external *k8s.io/api* [40] library to Operators, thus their source code and scope markers are not directly accessible for CodeQL. Therefore, the analyzer adopts an alternative method. Specifically, Operators would ultimately depend on the *client-go* library [42]. Each type of built-in Kubernetes resource is uniquely associated with a typed client provided by the *client-go* library [42]. Each typed client is constructed by methods in its corresponding *Getter* interfaces under the *k8s.io/client-go/kubernetes/typed* package. For example, considering the built-in resource type *Pod*, there is a uniquely associated Pod client. The Pod client is constructed by the only method in the *PodGetter* interface. By extracting the return types of methods in all *Getter* interfaces, the analyzer identifies all built-in types and their clients.

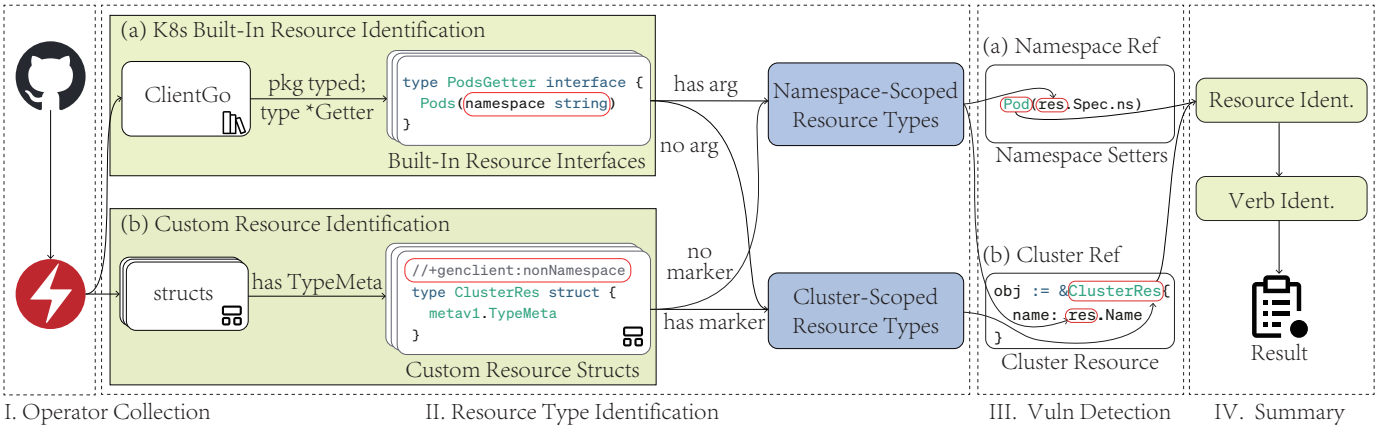

Fig. 5. Measurement Workflow

To determine the scope of built-in resources, the analysis leverages the only method in the *Getter* interface of each typed client. Specifically, ns-scoped resource clients in client-go require a *namespace* parameter in their constructor to specify the target namespace. In contrast, constructors for cluster-scoped resource clients do not require such a namespace argument. By counting and verifying constructor parameters, the analysis distinguishes ns-scoped from cluster-scoped built-in resources.

4) Vulnerability Detection: This step determines whether an attacker-controlled input can influence sensitive operations that cross namespace boundaries or impact the whole cluster. To achieve this, the analysis uses interprocedural taint tracking, tracing the propagation of data originating from the ns-scoped resource objects to insecure reference sites in the controller logic.

Insecure Namespace-Scoped Resource Reference. The goal of this detection is to determine whether attacker-controlled values can be used to specify the namespace of another resource by the Operator. This is essential because if the attacker can influence which namespace a referenced resource belongs to, they can trick the Operator into accessing or modifying resources beyond their authorized scope.

TABLE II
RESOURCE NAMESPACE SETTERS

Field	Library
ObjectMeta.Namespace	k8s.io/apimachinery
NamespacedName.Namespace	k8s.io/apimachinery
ObjectMetaApplyConfiguration.Namespace	client-go

Function	Library
ApplyConfiguration.WithNamespace()	client-go
*.SetNamespace()	k8s.io/apimachinery
Constructor of Typed Client	client-go
Constructor of Typed Client	client-gen

Thus, the analysis tracks data flow from ns-scoped resource objects to namespace setters (listed in Table II) that are leveraged to specify the namespace of a resource. By systematic code reviews of common Operator libraries, we identify 3 struct fields that can store namespace values in 2 libraries and identify 4 types of functions in 3 libraries that can be used to set the namespace field of a resource object or set up a typed client towards a specific namespace.

It is worth noting that the 3 fields (listed in Table II) of ns-scoped resource objects are excluded from the taint source, as these fields denote the namespace where this resource is deployed. Since the attackers are only authorized to access their namespace, the namespace fields of attacker-controlled resources are always the attacker-authorized namespace. If these fields sink in the referenced resources' namespace fields, it means the referenced resources are also in the attacker-authorized namespace. Thus, no cross-ns operation is conducted.

Taking the above key points into consideration, if the tainted data flows into any of these namespace setters, the Operator is flagged as potentially vulnerable to insecure namespace-scoped resource references.

Insecure Cluster-Scoped Resource Reference. This analysis aims to detect whether attacker-controlled input can influence cluster-scoped resources. Since cluster-scoped resources affect the entire Kubernetes cluster, any modification to them based on ns-scoped input represents a significant privilege escalation risk. Thus, the analysis tracks data flow from the identified ns-scoped resource objects into any cluster-scoped resource objects. If tainted input is used to construct cluster-scoped resource objects, the Operator is flagged as vulnerable to insecure cluster-scoped resource references.

5) Summary: To understand what an adversary can do to which kind of resource, the measurement further identifies insecurely referenced resource types and operations towards these resources.

Affected Resource Type Identification. This step discovers which resource can be referenced by an adversary. This identification is trivial for insecure cluster-scoped resource references, as their sink site in the vulnerability detection phase is set to cluster-scoped resource objects. Thus, the

affected cluster-scoped resource types can be directly extracted from sink objects.

For insecure ns-scoped resource references, the affected resource type identification depends on the type of sink site in the previous step. If the previous data flow sinks at the three fields, *WithNamespace()*, or *SetNamespace()* methods, then the analyzer further tracks interprocedural data flow from the previous sinks to any ns-scoped resource objects to identify affected resource objects and types. If the previous data flow sinks at the constructor of a typed client, then the resource type is the one associated with that typed client.

Verb Identification. To understand what an adversary can do to the insecurely referenced resources, the analyzer identifies the Kubernetes API Verbs (e.g., Get, Create, Update, Delete, etc.) related to insecurely referenced resources. If an insecurely referenced resource is found to be related to a Verb, like Create, then the adversary can exploit the vulnerable Operator to create the insecurely referenced resource in the Kubernetes cluster, which they should not.

Verb identification is achieved by identifying client method invocations that accept insecure references. For *controller-runtime* library, it processes all resource types by a unified typeless client in the *sigs.k8s.io/controller-runtime/pkg/client* package. For *client-go* and *client-gen* libraries, they process each type of resource with a specific typed client. Each Kubernetes API Verb corresponds to the client method with the same name. The analyzer thus performs interprocedural taint tracking from the reference site to these client methods to identify the related verbs.

B. Measurement Result

We conducted measurements on 2,268 Operators crawled from GitHub to assess the real-world impacts of insecure cross-ns references. The measurement suite was run on a Windows 11 machine with an Intel i7-10700K CPU (3.80GHz) and 32GB RAM. The suite cost 40026 seconds, with 17.6 seconds per Operator on average. In this part, we answered the following research questions:

- **RQ1:** How many operators are potentially vulnerable to insecure cross-namespace reference?
- **RQ2:** What resources can be cross-namespace referenced by attackers?
- **RQ3:** What can attackers do towards cross-namespace referenced resources?
- **RQ4:** How can insecure cross-namespace references impact the real world?

1) RQ1: How Many Operators Are Potentially Vulnerable To Cross-Namespace Reference?: To assess the prevalence of insecure cross-ns reference vulnerabilities, we analyzed a dataset of 2,268 real-world Kubernetes Operators collected from GitHub. Each Operator was examined to determine whether attacker-controlled ns-scoped resources can influence operations across namespace boundaries. The results illustrated that 318 Operators were potentially vulnerable, affecting 282 Operator providers. Specifically:

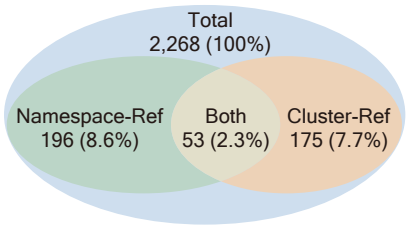

Fig. 6. Percentage of Affected Operators

- 196 Operators (8.6%) include insecure ns-scoped resource references that can specify or influence operations on other namespaces.
- 175 Operators (7.7%) contain insecure cluster-scoped resource references, where attacker-controlled ns-scoped resources can affect cluster-scoped resources.
- 53 Operators (2.3%) allow both types of references, posing risks of privilege escalation at both the namespace and cluster level.

These findings illustrate that a non-negligible portion (over 14%) includes logic that may lead to privilege escalation, highlighting a widespread but largely overlooked security concern in the Kubernetes ecosystem.

TABLE III
OPERATOR TYPE SUMMARY

Type	Validated?	# of Flagged	# of Validated
Cluster Management	✓	56	11
Database	✓	52	9
Networking	✓	34	6
Cloud Management	✓	25	3
CI/CD	✓	23	3
Metrics	✓	22	3
Storage	✓	17	4
Gadgets	✓	12	3
Secret Management	✓	10	1
Device	✓	9	2
Workflow	✓	8	1
Testing	✓	7	4
Identity	×	7	0
AI	✓	7	2
OS Management	✓	6	1
Searching	✓	4	1
Security	×	4	0
Framework	×	3	0
Config Management	✓	3	1
CMS	×	2	0
Financial	×	2	0
Registry	×	2	0
Backup	×	1	0
Collaboration	×	1	0
Streaming	×	1	0
Total	**16✓**	**318**	**55**

Table III illustrates the type distribution of flagged Operators. These Operators cover a wide range of usage types in the real world, where Cluster Management, Database, and Networking are most prevalent.

We further assessed the accuracy of our measurement. Due to the lack of existing datasets or detection tools for this new

class of vulnerabilities, it is infeasible to validate all Operators at scale manually. Thus, we primarily focused on false positives, evaluating if flagged Operators were truly vulnerable. We randomly reviewed 55 flagged Operators without any specific tailored criteria, covering major types of Operators as listed in Table III. Although this sampling may not cover all potential Operator variations or directly conclude the perfect representative, it provides a practical basis to estimate the accuracy of our detection by covering the major types (or most prevalent types) of Operators, as we shown in Table III. Among the 55 cases, only 5 were false positives. This result indicates that the vast majority of cases flagged are indeed vulnerable, which supports the reliability of our measurement concerning the prevalence of vulnerability. While false negatives are difficult to quantify due to the lack of ground truth, our analysis focused on a range of commonly observed cross-namespace patterns, which may not capture all potential variations but still reflect realistic threats and led to confirmed vulnerabilities. Overall, the measurement provides strong evidence that Cross-NS Reference Vulnerabilities are non-negligible in the real-world ecosystem.

TABLE IV
MAJOR INSECURELY REFERENCED RESOURCE TYPE

Scope	Resource Type	Ref By #Op.
Namespace	Secret	102
	ConfigMap	29
	Deployment	29
	Service	22
	StatefulSet	12
Cluster	Namespace	62
	ClusterRoleBinding	40
	ClusterRole	26
	Node	25
	PersistentVolume	15

2) RQ2: What Resources Can Be Cross-Namespace Referenced By Attackers? : To understand the attack surface exposed by insecure cross-ns references, we investigate the types of resources that Operators allow attackers to reference across namespaces. For each resource type, we count the number of Operators that insecurely reference it and analyze which types are most frequently involved.

Among ns-scoped resources, the most commonly insecurely referenced types (listed in Table IV) are Secret (referenced by 102 Operators), ConfigMap (29 Operators), and Deployment (29 Operators). In Kubernetes, Secrets store highly sensitive data such as API keys, credentials, and TLS certificates. ConfigMaps often contain important application configurations that control applications' behavior, like API endpoints and performance arguments. Deployments define and manage the application workloads by controlling replica sets and pods.

For cluster-scoped resources, the most common insecurely referenced types are Namespace (62 Operators), ClusterRoleBinding (40 Operators), and ClusterRole (26 Operators). In Kubernetes, Namespace is the resource that defines a namespace in a Kubernetes cluster. ClusterRoles and ClusterRoleBindings define and grant cluster-level permissions that apply to the whole cluster.

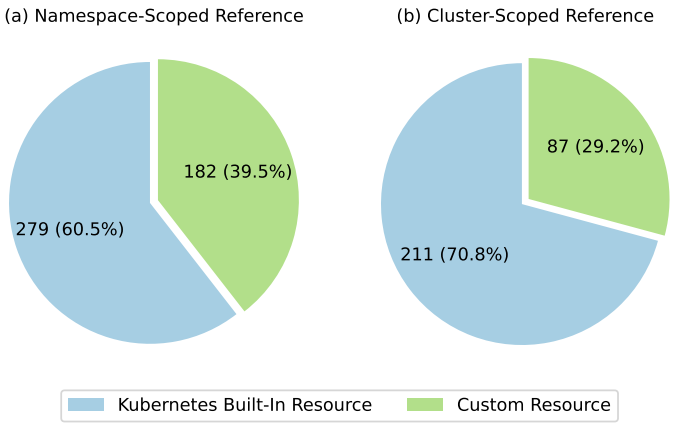

Fig. 7. Reference of Built-In and Custom Resources

We also distinguish between insecure references to built-in Kubernetes resources and custom resources. To this end, we aggregated the type-#Operator result above based on built-in resource type or custom resource type. For insecure ns-scoped references, 279 cases involved built-in resources and 182 involved custom resources. For insecure cluster-scoped references, 211 targeted built-in resources and 87 involved custom resources. These results indicate that insecure references can affect both built-in resources and custom resources. And the insecure references are more commonly associated with Kubernetes built-in resources.

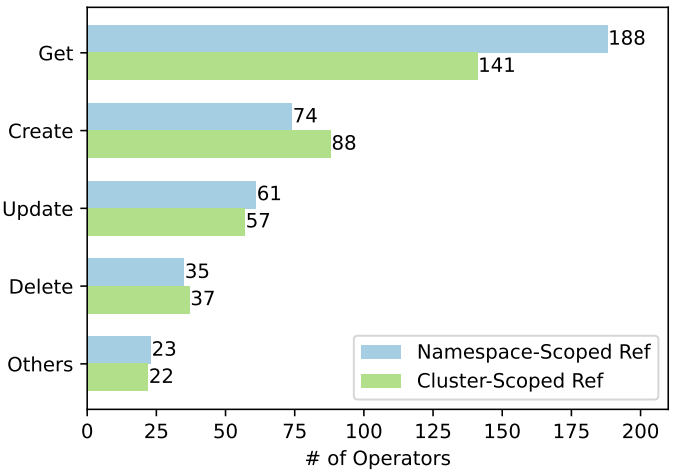

Fig. 8. Verbs Used By # Operators Towards Insecurely Referenced Resources

3) RQ3: What Can Attackers Do Towards Cross-Namespace Referenced Resources?: To understand the potential impact of insecure cross-ns references, we analyze the operations (Kubernetes API Verbs) that Operators perform on the referenced resources. For each identified insecurely referenced resource, we extract its related verbs. We then count how many Operators apply each verb to each insecurely referenced resource type. The result is illustrated in Figure 8.

For insecurely referenced ns-scoped resources, the top three most common verbs are *Get* (used by 188 Operators), *Create* (74 Operators), and *Update* (61 Operators), with *Get* being the most prevalent. In Kubernetes, *Get* retrieves the current state of a resource, *Create* instantiates a new resource, and *Update* modifies an existing resource. The predominance of the *Get* operation indicates that a large number of vulnerable Operators retrieve data from resources in other namespaces based on attacker-controlled inputs, exposing unauthorized data to attackers.

For insecurely referenced cluster-scoped resources, the top three verbs are *Get* (used by 141 Operators), *Create* (88 Operators), and *Update* (57 Operators), with *Get* accounting for the largest proportion. This suggests that in many cases, Operators may use attacker-influenced data to get cluster-wide resources, which may expose sensitive cluster-level information to attackers.

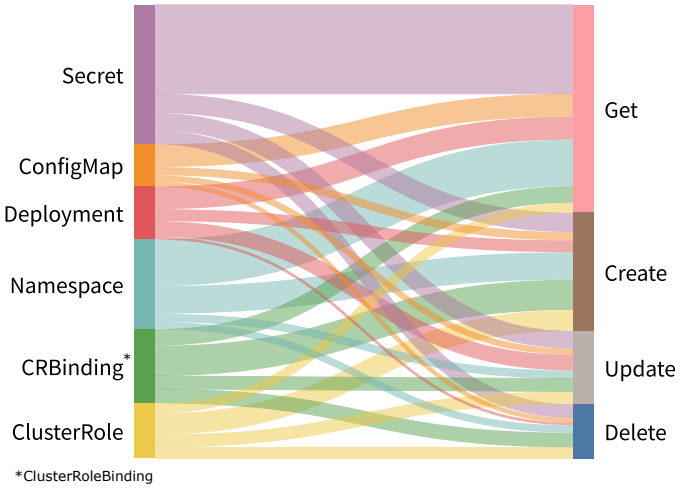

Fig. 9. Verbs of Major Insecurely Referenced Resources

TABLE V
MAJOR INSECURELY USED VERB-RESOURCE PAIRS

Scope	Verb - Ref.Resource Type	Used By #Op.
Namespace	Get - Secret	97
	Get - ConfigMap	25
	Get - Deployment	25
	Create - Secret	21
	Update - Secret	19
Cluster	Get - Namespace	51
	Create - ClusterRoleBinding	33
	Create - Namespace	29
	Create - ClusterRole	23
	Get - Node	20

To understand the practical implications, we investigate the verb-resource pairs (i.e., combinations of verbs and insecurely referenced resource types) to determine which verbs Operators typically perform on specific insecurely referenced resource types. The result is illustrated in Table V and Figure 9.

For insecurely referenced ns-scoped resource types, the most prevalent pairs are *Get-Secret* (used by 97 Operators), *Get-ConfigMap* (25 Operators), and *Get-Deployment* (25 Operators). These indicate that a substantial number of Operators can be exploited to read data from Secrets, configuration files, or application deployments in attackers' unauthorized namespaces, causing information exposure.

For insecurely referenced cluster-scoped resource types, the most prevalent pairs are *Get-Namespace* (used by 51 Operators), *Create-ClusterRoleBinding* (33 Operators), and *Create-Namespace* (29 Operators). These patterns suggest that Operators may be exploited to reveal other namespaces in the cluster, assign cluster-wide permissions, or provision new namespaces, leading to privilege elevation.

Together, these findings highlight that insecure cross-ns references are not only present but often tied to high-impact operations on sensitive or privileged Kubernetes resources.

C. RQ4: Case Study

To validate the practical significance of the identified vulnerabilities, we conducted in-depth inspections of vulnerable Operators. Combined with static analysis and manual exploits, the detailed impacts of their vulnerabilities are identified. We responsibly disclosed our findings to their vendors. By the time of submission, 8 vulnerabilities have been confirmed by the affected vendors as listed in Table VI, where all vulnerable resources are custom resources, as Operators are typically triggered by their custom resources and then reference other resources. 7 CVEs have either been assigned or are currently under processing, reflecting community acknowledgment and real-world relevance of these issues. Notably, even Operators maintained by Google, the inventor of Kubernetes, and Red Hat, the inventor of Operators, were confirmed to be vulnerable to insecure cross-ns references, highlighting that the problem is systemic and not limited to less mature projects.

We introduced 2 vulnerabilities to illustrate how these insecure references impact the real world.

1) Grafana/tempo-operetor Insecure ClusterRole Assignment: Grafana [46], a member of CNCF, is a widely adopted open-source analytics and visualization platform, renowned for transforming complex data into interactive dashboards. Among their products, Grafana Tempo [47] stands as a distributed tracing backend for the Grafana Observability Stack, gaining over 4,000 stars on GitHub. The tempo-operator [48], developed by both Grafana and Red Hat, is the official solution for deploying and managing Grafana Tempo on Kubernetes clusters. It defines a ns-scoped Custom Resource, *TempoStack*, for users to deploy Tempo.

Specifically, when enabling *JaegerQuery* and *MonitorTab* functions in a *TempoStack* resource, tempo-operator would create a ClusterRoleBinding, granting cluster-level permissions to a ServiceAccount in the namespace specified by the attacker-controlled *TempoStack.Namespace* field. An attacker can thus deploy a *TempoStack* in his namespace, where the *JaegerQuery* and *MonitorTab* functions were enabled and *TempoMonolithic.Namespace* field was set to the name of his

TABLE VI
VENDOR-CONFIRMED VULNERABILITIES

Operator	Vendor	Status	Vulnerable Resource	Referenced Resource	Verb
elcarro-oracle-operator	Google	Confirmed	Instance	Backup	Get
gateway-operator	Kong	CVE Assigning	AIGateway	Secret	Get
baremetal-operator	Metal3-io	CVE-2025-29781	BMCEventSubscription	Secret	Get
observability-operator	Red Hat	CVE-2025-2843	MonitorStack	ClusterRoleBinding	Create/Update/Delete
gateway-operator	Kong	CVE Assigning	ControlPlane	ClusterRoleBinding	Create
tempo-operator	Grafana & Red Hat	CVE-2025-2842	TempoStack	ClusterRoleBinding	Create/Delete
tempo-operator	Grafana & Red Hat	CVE-2025-2786	TempoMonolithic	ClusterRoleBinding	Create/Delete
ais-k8s	NVIDIA	CVE-2025-23260	AIStore	ClusterRoleBinding	Create/Update/Delete

authorized namespace. The Operator will then grant cluster-level permissions to a ServiceAccount in his namespace. The attacker can then impersonate the ServiceAccount to gain access to resources in the whole cluster.

In terms of impacts, the operator would grant the *cluster-monitoring-view* ClusterRole, an OpenShift-specific role enabling access to Prometheus and Thanos APIs, thus allowing the attacker to observe metrics and monitor cluster-wide resource states. We validated the bug and reported it to the vendors. They responded that *"We confirmed the vulnerability and will start the process to assign a CVE shortly. "*. CVE-2025-2842 was then assigned to us.

2) GoogleCloudPlatform/elcarro-oracle-operator Insecure Cross-Namespace Database Backup Reference: Google, the creator of Kubernetes and a member of CNCF, significantly influenced the evolution of cloud-native computing. Google develops multiple Operators to help users deploy common applications on Google Kubernetes Engine (GKE). Among them, elcarro-oracle-operator [49] is the one for deploying and managing lifecycles of Oracle databases on GKE, providing functions including database backup and restore.

The elcarro-oracle operator was found vulnerable to Insecure NS-Scoped Resource Reference when processing the custom resource *Instance*. *Instance*, a ns-scoped resource, was used to specify the parameters for an Oracle database instance. It defined a field *Instance.Spec.Restore.BackupRef* for users to restore the database instance from a specified database backup in the cluster. However, the field allowed users to reference a backup in other namespaces. An attacker who was only authorized in his namespace can thus restore any database backups from unauthorized namespaces into his own namespace, causing information leakage.

Moreover, the backup storage of elcarro-oracle-operator was based on Google Cloud Storage (GCS). The Operator thus required the GKE cluster to have read, write, and delete permissions to the GCS buckets and paths where backups were stored. However, when a user requested a database restore with the *Instance*, the database restore tools would run in the user's namespace and produce logs including the GCS buckets and paths where the target backup was stored. An attacker can then acquire the real GCS path of victim backups. Since the GKE cluster had both write and delete permissions to the backup GCS path, the attacker in the GKE cluster was naturally authorized to modify and delete victim database backups, causing data tampering and loss of data.

We validated the bug and responsibly reported this issue to Google. They have confirmed this vulnerability with the rationale *"Vulnerabilities where the only precondition is the attacker having a role in, or belonging to, a Google Cloud organization, project, or resource, with no interaction between attacker and victim"*.

VI. MITIGATION & DISCUSSION

During our inspection of vulnerable Operators, we realized that Operators aim to simplify user operations as much as possible, thus may embed cross-ns reference functionality to spare users the repetitive, manual task of duplicating resources like Secrets across namespaces—a practice necessitated by Kubernetes' namespace isolation. However, this convenience can inadvertently introduce vulnerabilities if not properly implemented, as attackers may exploit such helpful behavior to perform unauthorized cross-ns actions and privilege escalations. Thus, we suggest the following mitigations to eliminate the cross-ns reference vulnerability.

Carefully Using Multi-Tenant Kubernetes. Practitioners are discouraged from sharing Kubernetes across untrusted users, ensuring all tenants in multi-tenant Kubernetes do no evil. Developers should eliminate vulnerabilities of applications running on multi-tenant clusters, which may be exploited to gain initial Kubernetes access and affect other tenants.

Scope Alignment. Developers are advised to ensure that the declared scope of resources accurately reflects the scope of their operational effect. If a resource is defined as ns-scoped but its process logic performs actions across multiple namespaces at the cluster level, it creates a dangerous mismatch between the resource's access control boundary and its actual impact. In such cases, the resource should be explicitly declared as cluster-scoped, ensuring that only privileged users can create or manipulate it.

Mitigate with Kubernetes Admission Control. In cases where cross-ns is necessary or modifying Operators is painful, we designed a mitigation approach based on *Validating Admission Webhook* [50] to reject harmful requests before proceeding to the original Operators. The mitigation works independently, demands no modification to existing Operators, and can be easily adapted to any project.

Essentially, Validating Admission Webhook is a mechanism to extend the native Kubernetes access control. When a user

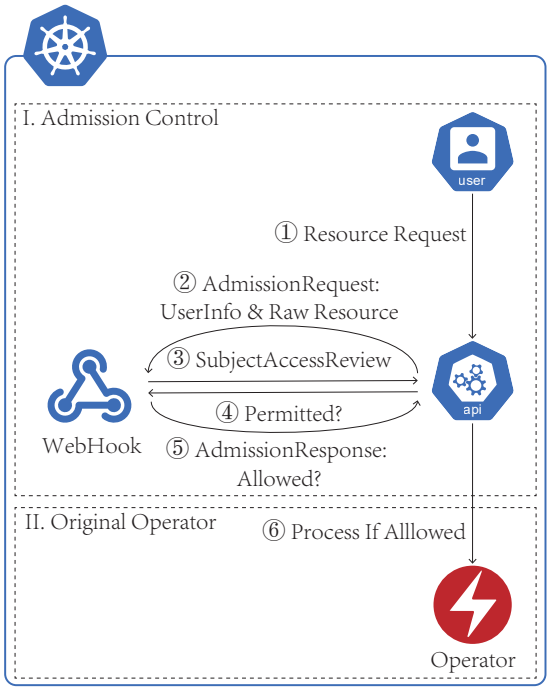

Fig. 10. Mitigate with Validating Admission Webhook

```
1  func (v *Validator) ValidateCreate(ctx, obj) {
2    // Raw Resource and Requests
3    aigateway, ok := obj.(*gateway.AIGateway)
4    req, err := admission.RequestFromContext(ctx)
5    // Craft SubjectAccessReview
6    secretNs := aigateway.Spec.CPC.Namespace
7    review := SubjectAccessReview{
8      Spec: SubjectAccessReviewSpec{
9        User:   req.UserInfo.user.Username,
10       ResourceAttributes: &ResourceAttributes{
11         Resource:  "secrets",
12         Verb:      "get",
13         Namespace: *secretNs,
14       },
15     },
16   }
17   // Check User Permission and Respond
18   err = v.client.Create(ctx, &review)
19   if !review.Status.Allowed {
20     return nil, fmt.Errorf("no permission")
21   }
22 }
```

Fig. 11. Simplified Admission WebHook Example For Mitigating gateway-operator Vulnerability

manipulates a resource, besides RBAC, Kubernetes will also invoke Webhooks to decide whether to allow or reject the request. In addition, Kubernetes provides the *SubjectAccessReview* [51] API, which can be used to check whether a user has specific RBAC permissions on a given resource. We leverage these mechanisms to further check whether the user has permissions for referenced resources and inform Kubernetes to reject requests if the user is unauthorized. This fills the gap of the native Kubernetes RBAC, which is not aware of whether a resource has cross-ns references.

Figure 10 gives the workflow of our proposed mitigation. The basic flow of a Validating Admission Webhook is that the Kubernetes API Server intercepts a user request, sends an *AdmissionReview* object containing the raw resource and user identity to the Webhook, and waits for an *AdmissionResponse* object from the Webhook indicating whether the request should be accepted or rejected. Upon receiving an *AdmissionReview* from the Kubernetes API Server, the Webhook reads the object and evaluates whether the user has cross-namespace permissions by issuing a *SubjectAccessReview* query to the API Server. If the result indicates that the user lacks permissions, the Webhook responds with an *AdmissionResponse* with the *allowed* field set to *false*, causing the API Server to reject the request before it reaches the Operator.

We implemented a Validating Admission Webhook with Kubebuilder [37]. Figure 11 illustrates a simplified version of our Webhook to mitigate the cross-ns secret reference vulnerability of the Kong gateway-operator. The Line 3-4 extracts the raw resource and raw request. The Line 6 gets the namespace of the referenced secret from the raw resource. The Line 7-16 crafts a *SubjectAccessReview* query, which checks whether the requesting user has permission to Get Secrets in the referenced namespace. The Webhook issues the query at Line 18, checks the result at Line 19, and rejects the unauthorized user at Line 20. With the scaffolding generated by Kubebuilder, tens of lines of code are required to implement a basic mitigation. And developers or cluster administrators can further implement other finer-grained validations in complex scenarios if needed.

TABLE VII
VALIDATING ADMISSION WEBHOOK MITIGATION EVALUATION

Operator	Mitigated	Before (ms)	After (ms)
elcarro-oracle-operator	✓	95.7	99.3 (+3.6)
gateway-operator	✓	94.5	97.0 (+2.5)
baremetal-operator	✓	91.6	94.5 (+2.9)
observability-operator	✓	96.5	101.2 (+4.7)
tempo-operator	✓	106.3	113.7 (+7.4)
ais-k8s	✓	101.7	105.2 (+3.5)
Average		97.7	101.8 (+4.1)

We evaluate its effectiveness and overhead on the vulnerable Operators listed in Table VII. The evaluation is conducted on a Kubernetes cluster with 1 control plane node and 3 worker nodes. Each node is equipped with 8 cores of Intel E5-2680 v4 CPU (2.40GHz) and 32GB RAM. To evaluate the effectiveness, we created two ServiceAccounts in the cluster: one with cluster-level permissions and another limited to a single namespace. Before deploying the mitigation, both ServiceAccounts were able to invoke all listed vulnerable Operators to perform cross-ns references. After deploying the mitigation, only the ServiceAccount with legitimate cross-ns permissions could invoke the affected Operators, while

the ServiceAccount with single-ns permissions was denied in all cases. This demonstrates that the mitigation successfully enforces namespace boundaries and prevents unauthorized privilege escalation.

To evaluate the overhead, since the Webhook only works at admission time and operates independently before the Operator logic, we measured the time between the point at which a new resource request was initiated and the point the Kubernetes API Server accepted the resource. For each test case, we repeated it 10 times and calculated the average overhead. Experimental results show an average increase of 4.1 ms after applying the mitigation. Given that the added overhead is a one-time overhead for a single resource request and Operator logic typically takes seconds to minutes, this added delay is negligible in practice.

To facilitate the mitigation, we open-sourced our Webhook for reference and a lightweight automatic Webhook generator, enabling quick adoption for practitioners.

Difference with Related Kubernetes Attacks. The cross-ns attack is fundamentally different from existing attacks [9, 12, 11] regarding threat models, root causes, and defenses.

For threat models, as illustrated in Section III, the precondition of launching existing attacks is to assume attackers have compromised containers (e.g., get a shell) before conducting overprivilege attacks, leaving critical gaps in how to compromise containers. In contrast, we presented a practical threat model without such strong assumptions, detailing two practical strategies to exploit Operators from scratch.

For root causes, existing attacks stem from granting applications unused permissions. Once attackers compromise applications (e.g., gain shells), they can conduct existing attacks by using over-granted privileges in follow-up actions. Thus, they [9, 12, 11, 2, 14, 15] suggest the Principle of Least Privilege (PoLP). Since only removing unused permissions, PoLP will not affect their functionality and is an ideal defense.

In contrast, our attack arises even when Operators follow PoLP. Supposing an Operator reading user input to get Secrets in a specified namespace (rather than assuming gain shells), it typically implies the Operator needs secret information for its operation, thus requiring certain privileges to function correctly. Removing such privileges would break the Operator's functionality, even for legitimate intra-ns references. In other words, our vulnerability stems from insecure use of legitimate privilege, which existing defenses cannot address. An ideal defense should preserve necessary privileges while mitigating vulnerabilities; therefore, we propose new mitigations.

More Cross-Tenant Attacks. Similar cross-tenant attacks can also happen in various cloud scenarios beyond Kubernetes. Security researchers have discovered multiple services provided by Microsoft Azure [52, 53, 54, 55, 56], AWS [57, 58], Google Cloud Platform [59, 60], and Oracle [61] were vulnerable to cross-tenant attacks, allowing attackers to manipulate unauthorized resources of other tenants. Our work further extends the landscape to Kubernetes and emphasizes cross-tenant risks in modern cloud architectures.

VII. Related Work

Kubernetes Operators and Controllers. Existing research on Kubernetes operators and controllers focuses on functional bugs instead of attacks. Acto [18] proposes an automatic end-to-end testing technique for validating the operational correctness of Kubernetes Operators. Acto continuously generates desired state declarations and verifies whether the Operator correctly reconciles the system to those states. Sieve [17] presents an automatic reliability testing framework for cluster-management controllers. By injecting faults, Sieve uncovers deep semantic bugs by observing how controllers behave under fault conditions they are expected to tolerate. Anvil [62] presents the formal verification framework for Kubernetes controllers via TLA-style temporal reasoning, validating whether controllers eventually bring the cluster to the desired state and maintain it. Kivi [63] verifies Kubernetes controllers and their configurations by modeling controller behaviors and checking for violations of user-defined intent properties using model checking. It detects issues like imbalance and lifecycle bugs, focusing on functional correctness. Xu et al. [16] systematically summarize historical functional bugs of Operator. Red Hat [13], Synk [14], and KubeOps [15] suggest several good practices for Kubernetes Operators. However, their core guidance, limiting RBAC scope, is a general Kubernetes practice, not specific to Operator, and does not mitigate our proposed attack.

To the best of our knowledge, our work is the first comprehensive study on Kubernetes Operator attacks.

Kubernetes Security. In terms of attack and exploitation techniques, MITRE [64] and Microsoft [65] summarize tactics to compromise containers and container orchestration systems like Kubernetes. Pecka et al. [66] investigate privilege escalation scenarios for DevOps pipelines on Kubernetes. He et al. [67] present cross-container attacks on Kubernetes with eBPF. Page Spray attack [68] can lead to container escaping, which can be mitigated by memory safety hardening and repair [69, 70]. Abbas et al. [71] design Cross-Linux-Namespace defenses to detect container escape attacks. Spahn et al. [72] set up honeypots on Kubernetes and analyze the attacks towards containers and container orchestration systems. Shringarputale et al. [73] present a co-residency attack towards container orchestration systems. Zeng et al. [74] comprehensively analyze 30 vulnerabilities in Kubernetes stacks. Avrahami and Hai [12, 75] identify the threat of trampoline pods which can be leveraged to gain escalated privileges. However, these works have not revealed or addressed the security issues brought about by Kubernetes operators.

Kubernetes offers extensive configuration options for managing applications, including access controls and specifying security contexts. Any misconfigurations can lead to severe security vulnerabilities. Thus, another theme of Kubernetes security research is eliminating misconfiguration. Shamim et al. [76, 77, 78] systematically reveal the risks of misconfiguration regarding best practice. Rahman et al. [79] design static analysis tools and conduct a large-scale empirical

study on Kubernetes manifests, revealing the landscape of misconfiguration. Ul Haque et al. [80] leverage knowledge graphs to detect and mitigate Kubernetes misconfiguration. Recent work Yang et al. [9] identify the security risk of excessive Kubernetes RBAC permissions, which may lead to whole cluster takeover. EPScan [10] follows up on the research and designs systems to automatically minimize RBAC permissions. The industry also presents numerous tools for Kubernetes security, including Trivy [19], Kubescape [20], KubeSec [21], KubeArmor [22], Open Policy Agent [23], and Kyverno [24], providing functions like misconfiguration detection and runtime policy enforcement.

While the existing works try to address misconfiguration and achieve the Principle of Least Privilege (PoLP) for applications, the vulnerability we present is not simply misconfigurations or violations of PoLP. They arise from inherent flaws in how Operators process user-controlled resources. These vulnerabilities exist even when the permissions of Operators are minimal, highlighting a deeper design-level security gap in the Operator model itself.

VIII. Conclusion

In this paper, we presented the first in-depth research on Kubernetes Operator attacks, unveiling a long-neglected Cross-Namespace Reference Vulnerability with two strategies, demonstrating how an attacker can bypass namespace isolation. We designed and implemented a static analysis suite to conduct large-scale measurements, illustrating that over 14% of Operators in the wild are potentially vulnerable. Our findings have been reported to the relevant developers, resulting in 8 confirmations and 7 CVEs by the time of submission, highlighting the critical need for enhanced security practices in Kubernetes Operators. We proposed concrete mitigation solutions and open-sourced our code to benefit the ecosystem.

Ethics Considerations

We conducted our research with strict adherence to ethical guidelines. To collect Operator projects, we followed GitHub's rate limits and usage policies. We validated vulnerabilities on our own Kubernetes clusters, ensuring that no third-party users or environments were affected. We responsibly disclosed vulnerabilities to all affected vendors. Thus, each affected project has at least 90 days to fix before publication. All vulnerability disclosures complied with the security policies of the respective vendors. All vulnerabilities in Table VI, including Google and Red Hat, were validated and then reported to the vendors according to their vulnerability policies. To facilitate mitigation, we suggested available approaches, released mitigation samples and a lightweight mitigation generator, enabling quick adoption for practitioners. We will ensure all vulnerabilities discussed in the case studies are fixed or authorized to mention by the time of the final publication. We are actively coordinating with vendors and will publish full datasets at the proper time (expected conference date) to ensure developers have enough time to fix and minimize impacts.

Acknowledgment

We thank our reviewers and shepherd for their valuable feedback and comments. For authors, Ziyi Guo is supported by Google PhD Fellowship. Zhenyuan Li is supported by the National Natural Science Foundation of China under 62402419 and by the CCF-Tencent Rhino-Bird Open Research Fund. Any opinions, findings, and conclusions or recommendations expressed in this work are those of the author(s) and do not necessarily reflect the views of the institutions above.

References

[1] "The voice of kubernetes experts report 2024: the data trends driving the future of the enterprise — cncf," 6 2024, [Online; accessed 2025-04-04]. [Online]. Available: https://www.cncf.io/blog/2024/06/06/the-voice-of-kubernetes-experts-report-2024-the-data-trends-driving-the-future-of-the-enterprise/

[2] "Kubernetes adoption, security, and market trends report 2024," 2024, [Online; accessed 2025-04-04]. [Online]. Available: https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview

[3] "Latest kubernetes adoption statistics: Global insights," 5 2024, [Online; accessed 2025-04-12]. [Online]. Available: https://edgedelta.com/company/blog/kubernetes-adoption-statistics

[4] D. A. Team, "Kubernetes: Adoption and market trends," 6 2024, [Online; accessed 2025-04-04]. [Online]. Available: https://datahubanalytics.com/kubernetes-adoption-and-market-trends/

[5] "Namespaces — kubernetes," 2024, [Online; accessed 2025-04-04]. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

[6] "Using rbac authorization — kubernetes," 2025, [Online; accessed 2025-04-04]. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/rbac/

[7] Vmware, "Top 5 kubernetes operations challenges and how to mitigate," 2022, [Online; accessed 2025-04-04]. [Online]. Available: https://www.vmware.com/docs/white-paper-top-5-kubernetes-operations-challenges-and-how-to-mitigate

[8] "Operator pattern — kubernetes," 2024, [Online; accessed 2025-04-04]. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[9] N. Yang, W. Shen, J. Li, X. Liu, X. Guo, and J. Ma, "Take over the whole cluster: Attacking kubernetes via excessive permissions of third-party applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 3048–3062. [Online]. Available: https://doi.org/10.1145/3576915.3623121

[10] Y. Gu, X. Tan, Y. Zhang, S. Gao, and M. Yang, " EPScan: Automated Detection of Excessive RBAC Permissions in Kubernetes Applications ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA,

USA: IEEE Computer Society, May 2025, pp. 11–11. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00011

[11] D. Mosyan, "Kubernetes — pod escaping privilege escalation — by david mosyan — medium," 7 2023, [Online; accessed 2025-10-27]. [Online]. Available: https://medium.com/@dmosyan/kubernetes-pod-escaping-privilege-escalation-ff45cffd96f1

[12] Y. Avrahami and S. B. Hai, "Kubernetes privilege escalation: Container escape== cluster admin," *Black Hat USA*, 2022.

[13] "Kubernetes operators: good security practices," 2022, [Online; accessed 2025-07-10]. [Online]. Available: https://www.redhat.com/en/blog/kubernetes-operators-good-security-practices

[14] C. Laux, "Security implications of kubernetes operators — snyk," 2 2022, [Online; accessed 2025-10-27]. [Online]. Available: https://snyk.io/blog/security-implications-of-kubernetes-operators/

[15] J. Keller, "Concerned about security in kubernetes operators? here are 6 security practices you should follow," 10 2025, [Online; accessed 2025-10-27]. [Online]. Available: https://kubeops.net/blog/concerned-about-security-in-kubernetes-operators-here-are-6-security-practices-you-should-follow

[16] Q. Xu, Y. Gao, and J. Wei, "An empirical study on kubernetes operator bugs," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1746–1758. [Online]. Available: https://doi.org/10.1145/3650212.3680396

[17] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 143–159. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/sun

[18] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu, "Acto: Automatic end-to-end testing for operation correctness of cloud system management," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 96–112. [Online]. Available: https://doi.org/10.1145/3600006.3613161

[19] "aquasecurity/trivy: Find vulnerabilities, misconfigurations, secrets, sbom in containers, kubernetes, code repositories, clouds and more," 2025, [Online; accessed 2025-04-04]. [Online]. Available: https://github.com/aquasecurity/trivy

[20] "kubescape/kubescape," 2025, [Online; accessed 2025-04-04]. [Online]. Available: https://github.com/kubescape/kubescape

[21] "controlplaneio/kubesec: Security risk analysis for kubernetes resources," 2025, [Online; accessed 2025-04-04]. [Online]. Available: https://github.com/controlplaneio/kubesec

[22] "kubearmor/kubearmor: Runtime security enforcement system. workload hardening/sandboxing and implementing least-permissive policies made easy leveraging lsms (bpf-lsm, apparmor)." 2025, [Online; accessed 2025-04-04]. [Online]. Available: https://github.com/kubearmor/kubearmor

[23] "Open policy agent," 2025, [Online; accessed 2025-04-04]. [Online]. Available: https://www.openpolicyagent.org/

[24] "Kyverno," 2025, [Online; accessed 2025-04-04]. [Online]. Available: https://kyverno.io/

[25] "Multi-tenancy — kubernetes," 2024, [Online; accessed 2025-04-06]. [Online]. Available: https://kubernetes.io/docs/concepts/security/multi-tenancy/

[26] "Pods — kubernetes," 2025, [Online; accessed 2025-04-14]. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/

[27] "Deployments — kubernetes," 2025, [Online; accessed 2025-04-14]. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[28] "Custom resources — kubernetes," 2024, [Online; accessed 2025-04-14]. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[29] "Developer sandbox — red hat developer," 9 2021, [Online; accessed 2025-10-27]. [Online]. Available: https://developers.redhat.com/developer-sandbox

[30] "Go operator tutorial — operator sdk," 2024, [Online; accessed 2025-04-10]. [Online]. Available: https://sdk.operatorframework.io/docs/building-operators/golang/tutorial/#1-run-locally-outside-the-cluster

[31] frittentheke, "Allow referencing a tls certificate / secret (secretname) residing in another namespace · issue #57325 · kubernetes/kubernetes," 12 2017, [Online; accessed 2025-10-29]. [Online]. Available: https://github.com/kubernetes/kubernetes/issues/57325

[32] neilharris1, "allow imagepullsecrets to reference secrets in other namespaces · issue #104415 · kubernetes/kubernetes," 8 2021, [Online; accessed 2025-10-27]. [Online]. Available: https://github.com/kubernetes/kubernetes/issues/104415

[33] Mangaal, "Kubernetes ingress not able to reference a secret that is in another different namespace · issue #107495 · kubernetes/kubernetes," 1 2022, [Online; accessed 2025-10-27]. [Online]. Available: https://github.com/kubernetes/kubernetes/issues/107495

[34] "Is there a way to share secrets across namespaces in kubernetes? - stack overflow," 9 2017, [Online; accessed 2025-10-29]. [Online]. Available: https://stackoverflow.com/questions/46297949/is-there-a-way-to-share-secrets-across-namespaces-in-kubernetes

[35] "azure - k8s use secret from different namespace - stack overflow," 12 2021, [Online; accessed 2025-10-29]. [Online]. Available: https://stackoverflow.com/questions/70492607/k8s-use-secret-from-different-namespace

[36] "Good practices for kubernetes secrets — kubernetes," 2024, [Online; accessed 2025-03-26]. [Online]. Available: https://kubernetes.io/docs/concepts/security/secrets-good-practices/#restrict-access-for-secrets

[37] "kubernetes-sigs/kubebuilder: Kubebuilder - sdk for building kubernetes apis using crds," 2025, [Online; accessed 2025-03-29]. [Online]. Available: https://github.com/kubernetes-sigs/kubebuilder

[38] "Operator sdk," 2025, [Online; accessed 2025-03-29]. [Online]. Available: https://sdk.operatorframework.io/

[39] "Codeql," 2025, [Online; accessed 2025-03-29]. [Online]. Available: https://codeql.github.com/

[40] "kubernetes/api: The canonical location of the kubernetes api definition." 2025, [Online; accessed 2025-03-29]. [Online]. Available: https://github.com/kubernetes/api

[41] "kubernetes/apimachinery," 2025, [Online; accessed 2025-03-29]. [Online]. Available: https://github.com/kubernetes/apimachinery

[42] "kubernetes/client-go: Go client for kubernetes." 2025, [Online; accessed 2025-03-29]. [Online]. Available: https://github.com/kubernetes/client-go

[43] "kubernetes-client/gen: Common generator scripts for all client libraries," 2025, [Online; accessed 2025-03-29]. [Online]. Available: https://github.com/kubernetes-client/gen

[44] "kubernetes-sigs/controller-runtime: Repo for the controller-runtime subproject of kubebuilder (sig-apimachinery)," 2025, [Online; accessed 2025-03-29]. [Online]. Available: https://github.com/kubernetes-sigs/controller-runtime

[45] "Rest api endpoints for search - github docs," 2022, [Online; accessed 2025-04-15]. [Online]. Available: https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28

[46] "Cloud native computing foundation," 2025, [Online; accessed 2025-04-11]. [Online]. Available: https://www.cncf.io/

[47] "Grafana tempo oss — distributed tracing backend," 2025, [Online; accessed 2025-04-12]. [Online]. Available: https://grafana.com/oss/tempo/

[48] "grafana/tempo-operator: Grafana tempo kubernetes operator," 2025, [Online; accessed 2025-04-15]. [Online]. Available: https://github.com/grafana/tempo-operator

[49] "Googlecloudplatform/elcarro-oracle-operator," 2025, [Online; accessed 2025-04-08]. [Online]. Available: https://github.com/GoogleCloudPlatform/elcarro-oracle-operator

[50] "Dynamic admission control — kubernetes," 2025, [Online; accessed 2025-06-26]. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

[51] "Subjectaccessreview — kubernetes," 2025, [Online; accessed 2025-06-26]. [Online]. Available: https://kubernetes.io/docs/reference/kubernetes-api/authorization-resources/subject-access-review-v1/

[52] "Chaosdb: Unauthorized privileged access to microsoft azure cosmos db," [Online; accessed 2025-10-29]. [Online]. Available: https://chaosdb.wiz.io/

[53] "Wiz research discovers "extrareplica"— a cross-account database vulnerability in azure postgresql — wiz blog," 4 2022, [Online; accessed 2025-10-29]. [Online]. Available: https://www.wiz.io/blog/wiz-research-discovers-extrareplica-cross-account-database-vulnerability-in-azure-postgresql

[54] "Microsoft azure synapse pwnalytics - blog — tenable®," 6 2022, [Online; accessed 2025-10-29]. [Online]. Available: https://www.tenable.com/blog/microsoft-azure-synapse-pwnalytics

[55] "Acsessed: Cross-tenant network bypass in azure cognitive search," [Online; accessed 2025-10-29]. [Online]. Available: https://www.mnemonic.io/resources/blog/acsessed-cross-tenant-network-bypass-in-azure-cognitive-search/

[56] "One token to rule them all - obtaining global admin in every entra id tenant via actor tokens - dirkjanm.io," 9 2025, [Online; accessed 2025-10-29]. [Online]. Available: https://dirkjanm.io/obtaining-global-admin-in-every-entra-id-tenant-with-actor-tokens/

[57] "Reported ecr public gallery issue," [Online; accessed 2025-10-29]. [Online]. Available: https://aws.amazon.com/cn/security/security-bulletins/AWS-2022-010/

[58] N. Frichette, "A confused deputy vulnerability in aws appsync — datadog security labs," 11 2022, [Online; accessed 2025-10-29]. [Online]. Available: https://securitylabs.datadoghq.com/articles/appsync-vulnerability-disclosure/

[59] "Google cloud platform (gcp) cross-tenant sql injection vulnerability on big query through native functions in looker studio - research advisory — tenable®," 9 2025, [Online; accessed 2025-10-29]. [Online]. Available: https://www.tenable.com/security/research/tra-2025-27

[60] "Google cloud platform (gcp) cross-tenant data sources leak with image rendering in looker studio - research advisory — tenable®," 9 2025, [Online; accessed 2025-10-29]. [Online]. Available: https://www.tenable.com/security/research/tra-2025-30

[61] E. Gabay, "Attachme: critical oci vulnerability allows unauthorized access to customer cloud storage volumes — wiz blog," 9 2022, [Online; accessed 2025-10-29]. [Online]. Available: https://www.wiz.io/blog/attachme-oracle-cloud-vulnerability-allows-unauthorized-cross-tenant-volume-access

[62] X. Sun, W. Ma, J. T. Gu, Z. Ma, T. Chajed, J. Howell, A. Lattuada, O. Padon, L. Suresh, A. Szekeres, and T. Xu, "Anvil: Verifying liveness of cluster management controllers," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp.

649–666. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/sun-xudong

[63] B. Liu, G. Lim, R. Beckett, and P. B. Godfrey, "Kivi: Verification for cluster management," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 509–527. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/liu-bingzhe

[64] "Matrix - enterprise - containers — mitre att&ck®," 2025, [Online; accessed 2025-04-13]. [Online]. Available: https://attack.mitre.org/matrices/enterprise/containers/

[65] "Tactics - threat matrix for kubernetes," 2025, [Online; accessed 2025-04-13]. [Online]. Available: https://microsoft.github.io/Threat-Matrix-for-Kubernetes/

[66] N. Pecka, L. Ben Othmane, and A. Valani, "Privilege escalation attack scenarios on the devops pipeline within a kubernetes environment," in *Proceedings of the International Conference on Software and System Processes and International Conference on Global Software Engineering*, ser. ICSSP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 45–49. [Online]. Available: https://doi.org/10.1145/3529320.3529325

[67] Y. He, R. Guo, Y. Xing, X. Che, K. Sun, Z. Liu, K. Xu, and Q. Li, "Cross container attacks: The bewildered eBPF on clouds," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5971–5988. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/he

[68] Z. Guo, D. K. Le, Z. Lin, K. Zeng, R. Wang, T. Bao, Y. Shoshitaishvili, A. Doupé, and X. Xing, "Take a step further: Understanding page spray in linux kernel exploitation," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1189–1206. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/guo-ziyi

[69] Z. Lin, Z. Yu, Z. Guo, S. Campanoni, P. Dinda, and X. Xing, "CAMP: Compiler and allocator-based heap memory protection," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4015–4032. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/lin-zhenpeng

[70] Z. Yu, Z. Guo, Y. Wu, J. Yu, M. Xu, D. Mu, Y. Chen, and X. Xing, "Patchagent: a practical program repair agent mimicking human expertise," in *Proceedings of the 34th USENIX Conference on Security Symposium*. USA: USENIX Association, 2025.

[71] M. Abbas, S. Khan, A. Monum, F. Zaffar, R. Tahir, D. Eyers, H. Irshad, A. Gehani, V. Yegneswaran, and T. Pasquier, "Paced: Provenance-based automated container escape detection," in *2022 IEEE International Conference on Cloud Engineering (IC2E)*, 2022, pp. 261–272.

[72] N. Spahn, N. Hanke, T. Holz, C. Kruegel, and G. Vigna, "Container orchestration honeypot: Observing attacks in the wild," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 381–396. [Online]. Available: https://doi.org/10.1145/3607199.3607205

[73] S. Shringarputale, P. McDaniel, K. Butler, and T. La Porta, "Co-residency attacks on containers are real," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, ser. CCSW'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 53–66. [Online]. Available: https://doi.org/10.1145/3411495.3421357

[74] Q. Zeng, M. Kavousi, Y. Luo, L. Jin, and Y. Chen, "Full-stack vulnerability analysis of the cloud-native platform," *Computers & Security*, vol. 129, p. 103173, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404823000834

[75] Y. Avrahami and S. B. Hai, "Trampoline pods: Node to admin privesc built into popular k8s platforms," 2022.

[76] M. S. Islam Shamim, F. Ahamed Bhuiyan, and A. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," in *2020 IEEE Secure Development (SecDev)*, 2020, pp. 58–64.

[77] S. I. Shamim, H. Hu, and A. Rahman, " On Prescription or Off Prescription? An Empirical Study of Community-prescribed Security Configurations for Kubernetes ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 707–707. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00170

[78] S. I. Shamim, "Mitigating security attacks in kubernetes manifests for security best practices violation," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1689–1690. [Online]. Available: https://doi.org/10.1145/3468264.3473495

[79] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, May 2023. [Online]. Available: https://doi.org/10.1145/3579639

[80] M. Ul Haque, M. M. Kholoosi, and M. A. Babar, "Kgsecconfig: A knowledge graph based approach for secured container orchestrator configuration," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 420–431.

IX. Artifact Appendix

A. Description & Requirements

1) How to access: The artifact can be accessed via the permanent URL at https://doi.org/10.17605/OSF.IO/PWVC4.

2) Hardware dependencies: Common Computers. Tested with a virtual machine equipped with 4 CPU Cores, 8 GB RAM, and 100 GB Disk.

3) Software dependencies: We package the whole environment into an OVF file for easy deployment. Users with virtual machine software, like VirtualBox and VMware, can directly start the virtual machine.

To deploy from scratch, the following requirements should be met:

- CodeQL v2.17.4: To run the static analyzer suite.
- Python 3 with tqdm and sarif-tools: To run the scripts.
- Go v1.24+, kubectl v1.11.3+, Docker v17.03+, and a Kubernetes v1.11.3+ cluster: To run mitigation samples.

4) Benchmarks: Datasets: CodeQL Databases of Kubernetes Operators, SARIF results generated by CodeQL.

B. Artifact Installation & Configuration

We suggest using our packaged virtual machine; then, only the virtual machine software is required. All artifacts are located at `/home/user/ArtifactEvaluation`. The username and password are both `user`.

Otherwise, install all dependencies listed above.

C. Major Claims

- (C1): 196 Operators include insecure namespace-scoped resource references. 175 Operators contain insecure cluster-scoped resource references. 53 Operators allow both types of references. This is proven by E1, whose result is presented in Figure 6.
- (C2): Among namespace-scoped resources, the most commonly insecurely referenced types are Secret (referenced by 102 Operators), ConfigMap (29 Operators), and Deployment (29 Operators). For cluster-scoped resources, the most common insecurely referenced types are Namespace (62 Operators), ClusterRoleBinding (40 Operators), and ClusterRole (26 Operators). This is proven by E1, whose result is presented in Table IV.
- (C3): For insecure namespace-scoped references, 279 cases involved built-in resources and 182 involved custom resources. For insecure cluster-scoped references, 211 targeted built-in resources and 87 involved custom resources. This is proven by E1, whose result is presented in Figure 7.
- (C4): For insecurely referenced namespace-scoped resources, the top three most common verbs are *Get* (used by 188 Operators), *Create* (74 Operators), and *Update* (61 Operators). For insecurely referenced cluster-scoped resources, the top three verbs are *Get* (used by 141 Operators), *Create* (88 Operators), and *Update* (57 Operators). This is proven by E1, whose result is presented in Figure 8.
- (C5): For insecurely referenced namespace-scoped resource types, the most prevalent pairs are *Get-Secret* (used by 97 Operators), *Get-ConfigMap* (25 Operators), and *Get-Deployment* (25 Operators). For insecurely referenced cluster-scoped resource types, the most prevalent pairs are *Get-Namespace* (used by 51 Operators), *Create-ClusterRoleBinding* (33 Operators), and *Create-Namespace* (29 Operators). This is proven by E1, whose result is presented in Table V.
- (C6): The webhook-based mitigation can successfully deny insecure cross-namespace operations, with only milliseconds of overhead. This is proven by E3, whose result is presented in Table VII.

D. Evaluation

Our paper conducted a large-scale measurement against real-world Kubernetes Operators. Due to the significant size of the raw data and the significant time required for full analysis, it may be challenging to conduct a full experiment for artifact evaluation. Thus, we designed scaled-down experiments for reviewers to reproduce.

To ensure that all reviewers can reproduce our measurement result, we provide the SARIF files of all Operators for reproduction. These SARIF files are intermediate results generated by running our CodeQL queries on measured Kubernetes Operators, which include information, like vulnerability locations, data flow, etc. These SARIF files can be efficiently analyzed using the scripts provided in the artifact to reproduce the measurement results presented in Section V-B.

Given that our measurement focuses more on vulnerable Operators, we provide CodeQL databases of all vulnerable Operators, which can be analyzed in a reasonable time for artifact evaluation. Reviewers can validate whether our CodeQL analysis suite is functional and generates the same results as we provide.

1) Experiment (E1): [10 Minutes Analysis]: Reproduce Measurement Results to Validate C1-C5.

[Preparation and Execution] Enter the `Experiments` folder of the artifact, and execute `bash ./run.sh`.

[Results] The full results will be printed on the console, reproducing results presented in Figure 6, Table IV, Figure 7, Figure 8, and Table V.

2) Experiment (E2): [4 Hours Analysis]: Validate Static Analyzer Functionality and Reproduce SARIF Results

[Preparation and Execution] Enter the `Analyzer` folder of the artifact. Execute `bash ./run.sh` to analyze all vulnerable Operators.

After analysis, execute `python ./diff.py` to validate that the freshly generated SARIF results are the same as those we provided in E1.

[Results] The final `diff.py` script will print the number of freshly generated SARIFs that are different from our provided SARIFs in E1. Ideally, the number should be zero.

3) Experiment (E3): [30 Minute]: Validate Mitigation Effectiveness and Measure Overhead

[Preparation and Execution] Enter the folder of the webhook, `Mitigation/webhook-examples`. Validate that attacks can be conducted and measure baseline overhead in a Kubernetes cluster without our mitigation by executing `python ./run_vulnerable.py`. Then, execute `python ./run_mitigated.py` to validate that attacks are blocked and measure the overhead in a Kubernetes cluster deployed with our mitigation.

[Results] It is expected that attacks can be successfully conducted in the vulnerable cluster. And the prompt, such as `xxxxxx Created`, will be printed, implying that malicious resources can be deployed to the cluster.

It is expected that attacks are denied in the mitigated cluster; the prompt like `webhook denied the request` will be printed, implying that malicious resources are blocked by our webhook-based mitigation.

Both scripts will print the average overhead for each Operator and the overall average overhead. It is expected that the average overhead of the mitigated cluster is around 5$\sim$15 milliseconds (may vary depending on hardware performance) over the baseline average overhead of the vulnerable cluster.

This experiment proves C6. The corresponding result in the paper is located at Table VII.