EPScan: Automated Detection of Excessive RBAC Permissions in Kubernetes Applications

Yue Gu*, Xin Tan*, Yuan Zhang, Siyan Gao, Min Yang
*Fudan University, * co-first authors*

***Abstract*—As the dominant container orchestration system, Kubernetes has a large ecosystem of third-party applications. The third-party Kubernetes applications access various cluster resources to extend the cluster functionality and Kubernetes adopts the RBAC mechanism to manage the resource access permissions. Recently, researchers revealed that third-party applications are granted excessive permissions and proposed an excessive permission attack. The attacker can exploit some critical excessive permissions to escape from the worker node and take over the whole Kubernetes cluster. However, this attack assumes that the attacker has compromised a worker node via container escape, which is difficult to realize in real scenarios.**

Therefore, we propose a new excessive permission attack with simpler attack conditions in this paper. We reveal that an attacker who has compromised one pod (less difficult than compromising a worker node) can exploit some other excessive privileges to take over worker nodes or break the availability and data confidentiality of other pods. Although excessive permissions of third-party applications pose a great threat to the security of Kubernetes clusters, there is no effective approach for detecting them.

In this paper, we propose a novel approach, namely `EPScan`, which automatically detects exploitable excessive permissions in third-party applications. To achieve this, `EPScan` employs a novel pod-oriented program analysis, which utilizes several new techniques to accurately identify the resource access behavior of the programs running in each pod. `EPScan` then compares the permissions required for these behaviors with those requested by the pod in its configuration file and finally reports the exploitable permissions that can be abused to launch an excessive permissions attack. We applied `EPScan` on 108 third-party applications from the CNCF projects and discovered previously unknown exploitable excessive permissions in 106 pods across 50 applications with a precision of 94.6% and 9 CVE identifiers assigned.

1. Introduction

Kubernetes is an open-source container orchestration platform for managing containerized workloads and services [1]. In recent years, Kubernetes [1] has been widely used by a great many companies and cloud vendors [2], [3] and has become the de-facto standard for container orchestration. The great success of Kubernetes is due in large part to its excellent and rapidly growing ecosystem. The Kubernetes ecosystem comprises various tools, technologies, and platforms from third-party organizations that help organizations manage their Kubernetes clusters and enhance their DevOps practices [4]. More importantly, the community also provides a large number of third-party plug-ins and applications running on Kubernetes control plane which extends the functionality of Kubernetes. According to our investigation, over 100 of the 187 projects hosted by CNCF [5] belong to Kubernetes applications. Additionally, there are over 1,000 third-party Kubernetes applications on the Artifact Hub [6] (the largest cloud-native package distribution platform). Some top Kubernetes applications have been widely used. For example, KEDA [7], an autoscaling application that provides event-driven scale for containers in Kubernetes, has more than 7.9k stars [8] and has been deployed to production environments of well-known cloud vendors such as AWS [9] and Azure [10].

To realize their functionality, the third-party applications utilize various Kubernetes cluster resources [11] via the Kubernetes API [12]. The Kubernetes API server employs Role-based access control (RBAC) to control resource access from applications. Specifically, the Kubernetes cluster grants permissions (i.e., what operations can be performed on what resources) to applications based on the permissions specified in the deployment configurations. At runtime, the API server authenticates the application that launched the request. As these third-party applications are granted critical permissions, the security flaws related to RBAC permissions would lead to serious consequences to the cluster.

Unfortunately, the security of permission management in these third-party applications has not yet been fully explored. Most existing research on Kubernetes security targets the co-residency attacks [13] and misconfigurations [14], [15], [16] and does not study the RBAC permissions. Recently, Yang *et al.* [17] proposed a new attack surface called excessive permission attack. That is, the third-party application is granted critical RBAC permissions that are not required for its functionality, called excessive permissions (EP for short). Yang *et al.* [17] demonstrated that excessive permissions can be abused to escape from the worker node and gain control over the whole Kubernetes cluster, resulting in privilege escalation in Kubernetes. However, the proposed attack requires a strong condition,

i.e., assuming that the attacker has compromised a worker node, which is usually achieved by container escape. In this paper, we propose a new attack model for abusing excessive permissions, which assumes the attacker has compromised the application within a container. We propose three attack paths for the model that lead to three different security consequences: controlling worker nodes, information leakage, and denial of service.

In summary, excessive permissions in third-party applications can lead to serious consequences. However, there is no approach that can effectively detect excessive permissions in Kubernetes applications. Yang *et al.* [17] developed a tool [18] to analyze configuration files and identify applications that request critical permissions. However, this tool does not analyze the application program, so it cannot determine if the requested permission is really being used by the application. In addition, there are many efforts to detect excessive permissions in Android system [19], [20], [21], [22], [23]. Unfortunately, due to the unique nature of Kubernetes and the Go programming language, there are many technical challenges in migrating these work to Kubernetes applications. First, the third-party application often consists of multiple pods, each running different executables. The complex compilation and image-building process of the application makes it difficult to identify the specific programs running in each pod and their code-level entries for program analysis. Second, the resource access APIs and libraries have never been systematically studied. This poses a challenge for program analysis in identifying and modeling resource access behavior in source code. Third, the function call chains in third-party applications are very long and complicated, containing virtual calls, external library calls, and Go goroutines. This leads to significant challenges in accurately analyzing what resource access behaviors can be reached by each code entry. In conclusion, analyzing resource access behavior in Kubernetes applications is challenging, which hinders the automatic detection of excessive permissions in third-party applications.

In this paper, we propose a novel end-to-end excessive RBAC permission analysis approach, named `EPScan` (Excessive Permission Scan). `EPScan` consists of three components. First, `EPScan` performs RBAC-centered configuration analysis to identify the permissions requested by the application. Then it employs a novel pod-oriented program analysis to determine the permissions required by the application's functionality. The proposed program analysis leverages the script understanding capability of the large language model (termed LLM) as well as a new reachability approximation method for Kubernetes applications to achieve accurate behavior analysis at the pod granularity. At last, it identifies the inconsistency between the requested permissions and the required permissions, as well as reports excessive permissions that can be exploited to launch the attacks proposed by Yang *et al.* [17] or the attack proposed in this paper.

We implement a prototype of `EPScan` based on the CodeQL framework [24] and leverage *gpt-4o* as the distillation LLM. We then applied `EPScan` to 108 third-party applications from the CNCF project list [5] to evaluate its effectiveness. It turns out that `EPScan` successfully identified excessive permissions in 106 pods across 50 applications, with only 6 false positives. As of now, the developers have confirmed 39 of them and assigned 9 CVEs. In addition, we evaluated the efficiency of `EPScan`. The results show that it takes an average of 14 minutes to analyze an application. Therefore, we believe `EPScan` can be extended to a large range of third-party applications, which has great practical value for permission management of Kubernetes applications. We also evaluated the effectiveness of each key component of the program analysis of `EPScan`.

In summary, we make the following contributions:

- We propose a new attack model for abusing excessive RBAC permissions. We reveal that an attacker inside a container can exploit excessive RBAC permissions to make attacks and cause severe consequences without container escape.
- We propose an automated approach to detect excessive RBAC permissions in third-party applications statically. Our proposed approach utilizes several new techniques to overcome the challenges of analyzing the resource access behaviors of Kubernetes applications.
- We implement a prototype of `EPScan` and evaluate its effectiveness in 108 third-party Kubernetes applications. As a result, `EPScan` successfully identified exploitable excessive permissions in 106 pods across 50 applications, with 9 CVE identifiers assigned.

Available Artifact. We open-source the prototype of `EPScan` under the Apache-2.0 license at `https://github.com/seclab-fudan/EPScan`.

2. Background

2.1. Kubernetes

Kubernetes Architecture. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications [25]. From an architectural point of view, a Kubernetes cluster mainly consists of worker machines called nodes (also known as the data plane) and a control plane [26], [27].

A node may be a virtual or physical machine, where workloads are executed as pods. Pods are the smallest deployable units of computing that are created and managed by Kubernetes [28]. In particular, a pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.

The control plane is responsible for managing and maintaining the Kubernetes cluster. In particular, the control plane consists of various official management components and some third-party extensions, which extend the cluster's functionalities. One of the core components is the Kubernetes API server, which serves as the API front end of the cluster. Through the interface exposed by the API server, worker nodes can access kinds of Kubernetes cluster resources (e.g., creating pods) to realize their functionalities.

Third-party Kubernetes Applications. While Kubernetes provides powerful features out of the box, the community further provides a rich ecosystem of third-party applications [29]. These third-party applications are a series of software that can be installed and run in the Kubernetes cluster, which provides external functionalities for interacting with underlying resources or improving the effectiveness of the Kubernetes cluster [17]. To implement special features, third-party applications often need to invoke various APIs to make use of Kubernetes resources.

With the growth of the Kubernetes community and ecosystem, there are lots of third-party applications in the real world.

- There are 187 projects on the CNCF project list [5], most of which are third-party applications for Kubernetes. Especially, some projects are marked as Graduated, which means these applications are used successfully in production environments. For example, Rook [30] is a graduated project to provide cloud-native storage for Kubernetes and has over 11k stars on GitHub.
- ArtifactHub [6] is a large open-source platform for publishing cloud native packages. Until now, there are 12,546 Kubernetes-related packages in the platform [31], including many third-party Kubernetes applications. For example, Kubernetes provides an operator pattern for developing Kubernetes applications. There are around one thousand operator-pattern applications in the platform [32].
- Top cloud vendors including Google [33], Amazon [34], Azure [35] develop third-party applications themselves and deploy these applications in their business environments to provide extra features for their users. For example, the Google Kubernetes engine supports the users running their workloads on a separate, dedicated node pool, which reduces the risk of privilege escalation attacks in the Kubernetes cluster [36].

2.2. Kubernetes Resources & API

Type of Resource. In order to extend the functionalities of the Kubernetes cluster, the third-party applications require various cluster resources. In all, the Kubernetes contains 58 resource types [11]. Most Kubernetes API resource types are objects – they represent a concrete instance in the cluster, like a pod or secret data [37]. Third-party applications rely on these resource instances for their own functionality. For example, *submariner* [38], a Kubernetes application facilitating network communication across multiple clusters, leverages the *daemonsets* resource to deploy and manage the network proxy pods. Besides, a smaller number of resource types represent operations on objects, rather than objects, such as a permission check.

Resource-based API. For accessing the resources, Kubernetes provides a resource-based (RESTful) programmatic interface, named Kubernetes API, which is implemented via HTTP. Kubernetes resources are all stored as API objects, and modified via RESTful calls to the API [37]. The Kubernetes API enables different operations on resources by accepting different API verbs. In particular, it supports retrieving, creating, updating, and deleting resources via the standard HTTP verbs. In addition, Kubernetes also implements its own verbs, including listing and watching resources.

Library for Resource Access. From within the pod, applications can craft HTTP requests and access the API URL for resource access directly. However, such a method is very cumbersome and the best practice [12] of accessing the API server is using the official libraries. By manually inspecting the implementation of 20 CNCF projects [5], we found that there are currently two commonly used libraries for accessing the API.

- **client-go library** [39] provides several official Go clients for communicating with a Kubernetes cluster. Specifically, it provides several clients that enable users to access arbitrary Kubernetes APIs.
- **controller-runtime Project** [40] is a set of Go libraries for building Controllers. Its client package also provides functionality for interacting with Kubernetes API servers.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
subjects:
- kind: ServiceAccount
  name: read-example
roleRef:
  kind: Role
  name: Pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Figure 1: A RoleBinding example. It binds the “pod-reader” role with the “read-example” serviceAccount.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Figure 2: A Role example. It has three verbs of the “pods” resource.

2.3. RBAC in Kubernetes

Kubernetes leverages role-based access control (RBAC) for the authorization of resource access (i.e., accessing the Kubernetes API) from third-party applications [41]. When a pod (hosting a third-party app) sends resource access requests to the API server, it authenticates as a particular service account. Only the requests that comply with the RBAC permissions of the *ServiceAccount* will be allowed

and processed by the API server. Specifically, in the RBAC mechanism, configuring RBAC permissions for a pod involves three steps. ❶ Developers should define *Role* objects which contain rules that represent a set of permissions. Rules have two important attributes, *resources* and *verbs*, that define what kind of actions can be taken on what kind of resources. ❷ After grouping permissions into *Roles*, developers could grant the *Roles* to *ServiceAccounts* through the RoleBinding object. ❸ When creating a pod object, the developers could configure *ServiceAccounts* for pods by setting the *spec* attribute. When the cluster starts, it automatically mounts a token to the application running inside the pod as its service account credentials to access Kubernetes API.

For example, as shown in Figure 1, the "read-example" service account has a *RoleBinding* named "read-pods" which binds the "pod-reader" *Role* to the account. As shown in Figure 2, the "pod-reader" *Role* has three verbs of the "pods" resource, "get", "watch" and "list", which grant read access to pods to the "read-example" service account. Finally, the "read-example" service account is assigned to a pod, which enables the pod to read all pod objects in the namespace.

Besides, the *Role* and *RoleBinding* objects are used to set and bind RBAC permissions within a particular namespace [42], which provides a mechanism for isolating resources within a single cluster. By contrast, Kubernetes provides *ClusterRole* and *ClusterRoleBinding* to define cluster-scoped permissions.

3. Understanding Excessive Permission in Kubernetes

In this section, we first illustrate what is excessive permission in Kubernetes third-party applications. Then we discuss attacks that can be launched using such excessive permissions, including a new attack model proposed for the first time in this paper.

3.1. Problem Statement

The problem of excessive permissions, i.e., applications have been granted more critical permissions than they need to function properly, is a classic problem in the security research of permission systems [20], [43], [44], [45], [46], [47]. This work focuses on excessive RBAC permission of third-party applications running in the Kubernetes cluster, which has not yet been fully explored.

Specifically, we assume the RBAC permission in Kubernetes is a (verb, resource) tuple. The verb specifies the type of operation the permission allows. The resource specifies the kind of resource on which the permission can operate. The Kubernetes cluster grants RBAC permissions to each pod (the basic unit of the Kubernetes cluster) according to the permissions requested in its configuration file. At runtime, the program running in the pod invokes Kubernetes API to access cluster resources and the API server authenticates the API request based on the granted permissions of the pod. If a pod requests more permissions in its configuration file than the least permissions required by the executables running in the pod, we assume the pod, as well as the third-party application, has an excessive permissions problem.

3.2. Attack Model

A Kubernetes cluster has 58 types of resources with a large number of RBAC permissions involved, but not all of them affect the security of the cluster. This paper focuses on the critical excessive permissions that can be exploited to cause harm to the Kubernetes cluster. Specifically, this paper takes into account two types of excessive permission attack models.

TABLE 1: Lists of exploitable EPs and security impacts.

Impact[a]	Permission	Scope
1	(list/get, secrets)	$=$ cluster
1	(update, clusterrolebindings)	$\geq$ namespace
1	(update/patch/escalate/bind, clusterroles)	$\geq$ namespace
1	(update/patch, nodes)	$=$ cluster
2	(create/update/patch, workload[b])	$\geq$ resource-specific
3	(list/get, secrets)	$\geq$ namespace
3	(create/update/patch, services)	$=$ cluster
3	(update/patch/delete, networkpolicies)	$=$ cluster
4	(delete, workload)	$\geq$ resource-specific
4	(update/patch/delete, services)	$\geq$ resource-specific
4	(delete, nodes)	$=$ cluster
4	(delete, ingresses)	$\geq$ resource-specific

[a] 1-Take over the whole cluster, 2-Take over a worker node, 3-Leak sensitive information, 4-Perform DoS attack.
[b] *workload* is an alias of resources: *pods*, *deployments*, *statefulsets*, *daemonsets*, *jobs*, *cronjobs*, and *replicasets*.

Attack-Model-I: *The worker node has been compromised.* It is widely believed that Kubernetes implements robust isolation mechanisms for securing the cluster [48], [49], [36], which includes controlling accesses to the Kubernetes APIs and resources, restricting the capabilities of a workload or user at runtime, and protecting cluster components from being compromised. Even if a worker node is compromised by a malicious user, it is difficult to compromise other nodes within the cluster. However, Yang *et al.* [17] reveal that an attacker can exploit some specific excessive permissions to compromise the isolation mechanism and eventually take control of the whole Kubernetes cluster, resulting in privilege escalation in Kubernetes.

In particular, the attack model proposed by Yang *et al.* assumes that an attacker controls one worker node of Kubernetes (a.k.a, attacker-controlled worker node). The attacker on a worker node can exploit excessive permissions of third-party apps' DaemonSet [50] to directly steal the cluster admin permission. In addition, the attacker can leverage excessive permissions to hijack the critical components of the same application or another third-party application and then use them to indirectly steal the cluster admin permission. As a result, an attacker who controls a worker node can escalate to the cluster admin and take over the whole cluster. The permissions involved in this attack model are listed in the first four rows of Table 1.

Attack-Model-II: *The pod has been compromised.* Although Yang *et al.* are the first to propose the excessive permission attack of Kubernetes applications, we notice that their attacks require a strong condition. That is, they assume the attacker has controlled a worker node. It is often achieved by exploiting multiple vulnerabilities to perform container escape, which requires strong vulnerability exploitation skills.

By further investigating Kubernetes permissions and their corresponding resources, we propose a new attack model. We assume that an attacker leverages some vulnerability to compromise an application running in a container, without taking control of the entire worker node. The attacker can then abuse excessive permissions to perform multiple attacks. Specifically, we propose three attack paths depending on the different attack consequences.

- **Take over a worker node.** In Kubernetes, the workloads (e.g., pods) are also wrapped as resource objects. If the compromised application has excessive permissions for creating or modifying other workloads, the attacker can construct a malicious workload object to deploy malicious privileged workloads on any worker node. For instance, the *Pod* object allows setting the target worker node through the *nodeSelector* attribute [51]. At last, the attacker can execute container escape and privilege escalation easily in the privileged container, thereby taking over the worker node. We cover the detailed procedure of this attack path with a real-world case in §6.7.
- **Perform DoS Attack.** Similar to the previous attack path, if the compromised application has excessive deletion permissions for other workloads, the attacker can delete the workloads, resulting in denial-of-service attacks on other services in the cluster.
- **Leak sensitive information.** In Kubernetes, there is a type of *Secret* object that contains some sensitive data such as passwords, tokens, or keys [52]. If the compromised application has excessive read permissions to the *Secret* objects of other pods, the attacker could exploit the excessive permissions to leak sensitive data, causing great property loss to users [53].

Difference between the models. First of all, both the attack models uncover that excessive permissions in third-party applications can cause serious harm to Kubernetes clusters, which motivates an effective approach for exploitable excessive permission detection. Second, the two attack models have different attack objectives. Yang *et al.* aim to utilize excessive permissions to break the isolation between worker nodes. That is, to control all worker nodes through one compromised worker node. In contrast, the attack proposed in this paper is dedicated to breaking the isolation between pods and worker nodes, as well as the isolation among pods. Finally, due to different objectives, the two models require different prerequisites. Our attack model assumes the attacker has controlled an application running in a container, which is easier to achieve in real scenarios.

4. Approach Overview

4.1. Overall Idea

The overall idea of detecting excessive permissions in Kubernetes is straightforward. Considering that Kubernetes takes pods as its basic unit, we need to perform the analysis with pod granularity. Given a third-party application, we first determine the permissions used by each pod based on the source code analysis and extract the permissions requested by each pod from the configurations. Through comparing the two permission sets, we can surely determine if the application has requested unnecessary critical permissions.

Note that we adopt static analysis to identify excessive permissions rather than dynamic analysis based on two main considerations. First, the dynamic analysis cannot cover all the code and therefore misses some critical issues. Second, dynamic analysis requires environmental setting up and well-configured APP installation, which causes usability issues in our scenario. For example, the *submariner* project [38] only deploys certain pods under specific non-default configurations, which requires massive configuring effort. Yang *et al.* reported that it takes one hour to complete the environment setup [17]. This raises unacceptable costs for scanning over 100 applications in CNCF project lists and over 1,000 in ArtifactHub.

4.2. Challenges & Solution

Though the overall idea seems simple, it is quite challenging to realize such an analysis. In particular, there are the following three technical challenges in analyzing resource access behavior and required permissions from the source code at pod granularity, which are due to the nature of Kubernetes and Go.

Challenge-I: *How to identify the source code entry for executables running in the pod?* The third-party application may contain multiple pods, each running different executables that correspond to different main entries in the code repository. In order to perform code-level program analysis for the executables running in each pod, we need to identify the main entry for each executable. However, determining which executables are run by each pod and their main entries is challenging. One challenge is that the startup process of the pod is complex, involving various styles of startup commands and complex shell scripts. There is no general analysis technique to determine the main program that the pod is running. Another challenge is that the third-party application has a complex compilation and image-packaging process, both preventing us from recognizing the main entries for the executables.

To address the challenge, we propose an LLM-assisted method. One reason is that LLMs are trained on extensive datasets that include a considerable amount of shell script code, enabling them to comprehend and process shell scripts effectively. Another reason is that recent research [54], [55] proves that the Language Learning Model has a good ability in comprehending and processing program [54], [55]. We

therefore introduce the large language model to help us understand the container startup process and identify the main executables it runs. We then use the runtime information embedded in the Go executable to locate its main entry in the source code. This information is unaffected by the packaging and compilation process in most situations.

Challenge-II: *How to identify and model resource access behavior in the program?* As introduced in §2.2, Kubernetes applications tend to perform resource access through official libraries. However, these libraries consist of a great many functionalities, involving hundreds of available APIs. There is currently no systematic study about the APIs used for resource access in these libraries.

Therefore, we perform a deep investigation to understand which APIs are used specifically for resource access and how they are being used. In particular, we manually examined all APIs in the **client-go library** [39] and **controller-runtime Project** [40]. In all, we collected 1498 resource accessing APIs across the two libraries as shown in Table 2. In addition, we further investigated how these APIs specify the types of operations and the types of resources to be operated on. Based on the investigation, we propose a customized static analysis approach to model the resource access behavior corresponding to API call sites.

Challenge-III: *How to determine the reachable resource access for each program?* After identifying the resource access behavior, an accurate reachability analysis is required to identify the resource access reachable from each main entry, otherwise, it will lead to false positives or false negatives in the subsequent excessive permission detection. However, the traditional call graph construction method for object-oriented languages (e.g., RTA [56], VTA [57]) is ineffective on these third-party applications. First, most function call chains of third-party applications are very deep, containing a large number of external library functions, which greatly affect the accuracy of traditional call-graph analysis. Second, third-party applications sometimes use callback and function pointers, which are not well supported by existing call graph analysis.

To address this challenge, we propose a tailored reachability approximation method for the main function of Kubernetes' third-party applications. The key observation is that whenever a main function (or functions reachable from main) holds a reference to another function, then the function is reachable from the main function. Specifically, the main function can hold the reference to a function directly by creating a function pointer, or instantiate a concrete object to hold the reference to the object's virtual functions. It is worth stating that even in cases like virtual calls and callbacks, the main function (or functions reachable from main) must hold a reference to the call target, either directly or indirectly, before the call occurs. For example, the program needs to register the target function of callbacks before the callback really takes place. In light of this, we could build a RefGraph that represents the reference relationship of all the functions. Then, given a main entry and a specific function, we can approximate the reachability between them by querying whether the main function holds a reference to the specific function.

This approach establishes the link between main entry and reachable functions with a simple analysis and avoids analyzing callbacks, virtual calls, and other call conventions that require complex data flow information and modeling to determine. By contrast, this proposed method has three limitations. Firstly the key observation only works for main entries. Secondly, the method can't give the entire call graph, it can only qualitatively determine the reachability. Finally, the method is not sound. Fortunately, the first two limitations have no impact on our analysis and we think a small number of false positives of the reachability analysis are acceptable.

5. Design of EPScan

Drawing on the above ideas, we propose `EPScan`, the first automated solution for analyzing the behavior of third-party Kubernetes applications and detecting excessive permissions among them. In this section, we first describe the overall architecture of `EPScan` and then introduce the design of each main component in detail.

5.1. Architecture

Figure 3 presents the overall architecture of `EPScan`. `EPScan` takes the source code and configuration files of the Kubernetes application as input and detects the security-sensitive excessive permissions. In particular, `EPScan` consists of three main components.

1) **RBAC-centered configuration Analysis** extracts RBAC permission requested by each pod of the application and unifies their format.
2) **Pod-oriented behavior analysis** leverages static analysis to diagnose the resource access behavior of the programs running in each pod and obtains the minimal permissions required by each pod.
3) **Excessive permission detection** compares requested permissions with required permissions to identify excessive permissions, and further picks out excessive privileges that bring out security risks.

5.2. RBAC-centered Configuration Analysis

This phase extracts all the pods as well as the permissions granted to each pod from the configuration files. Specifically, `EPScan` first extracts the permission rules of pods from the configuration files and then transforms them into the form of permission tuples described in §3.1 for subsequent excessive permission checking.

RBAC rule extraction. As introduced in §2.3, permissions requested by pods are defined through the permission rules in *Role* and *ClusterRole* objects and then bound to pods via *RoleBinding/ClusterRoleBinding* and *ServiceAccount* objects. Thus `EPScan` first parses all *Pod* objects as well as *RoleBinding*, *ClusterRoleBinding* and *ServiceAccount* objects to recover the binding between the *Pod* objects and the *Role/ClusterRole* objects. Then, for each pod,

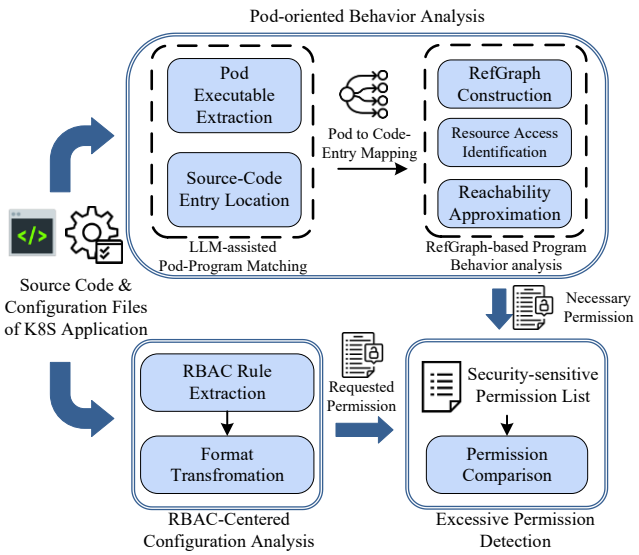

Figure 3: Architecture of EPScan

EPScan further parses the definition of corresponding *Role* and *ClusterRole* objects to extract the permission rules. In particular, EPScan directly collects the rules from the *rules* attribute.

Format transformation of rules. The format of the extracted permission rules does not lend itself to later analysis. First, Kubernetes allows the use of wildcard symbols (*) to match any resources or verbs. Second, raw permission rules may overlap, creating obstacles in excessive permission analysis. Therefore, EPScan further unifies the permission rules into permission tuples as defined in §3.1. Specifically, for each permission rule, EPScan extracts the set of resources and the set of verbs from it and then computes the Cartesian product of the two sets, thus converting the permission rule into a list of (verb, resource) tuples. If the rules use wildcard symbols (*), EPScan replaces them with concrete resources or verbs according to the Kubernetes official documents[58], [59]. Eventually, EPScan merges and de-duplicates the analysis results of all the permission rules of a pod.

5.3. LLM-assisted Pod-Program Matching

As introduced in §4.2, a Kubernetes application may consist of multiple pods, each running its own program that requests varied Kubernetes resources. To realize permission analysis at the pod granularity, EPScan needs to determine which programs are being run by the pods and locate the source-code-level entry points for later program analysis.

Pod Executable Extraction. Initially, EPScan downloads the container image and attempts to examine which executables might be run as the main service of the pod. However, as introduced in §4.2, it is challenging to determine the real executable running by the pod due to the complicated and varied startup commands and scripts. Considering the Language Learning Model is proven to be effective in comprehending and processing programs [54], [55], EPScan utilizes an LLM-based method to determine the executable. This method generates a prompt based on the contents of the startup commands and scripts, and then queries LLM what is the main executable started by the commands and scripts.

In particular, EPScan adopts the few-shot in-context learning approach, as outlined in existing studies [60], to construct the prompts. A typical few-shot prompt consists of three components, task instructions, K example queries with answers, and the actual query. EPScan constructs the prompt in the following ways:

- **Task instructions** provide a detailed description of the task, including the type of information provided, how inputs should be processed, and the desired format and content of responses. Following best practices in prompt engineering [61], we specified a persona of *startup script analyzer* for the LLM in the prompt and defined the response format and content through a series of rules, e.g., present the executable list in JSON format.
- **K examples**, also referred to as *K shots*, serve as typical task examples that enable the LLM to (1) generate responses following the format of the example answers and (2) learn to perform the tasks without modifying model parameters by observing typical patterns in shell scripts. The format of the examples is consistent with that of the actual query, except with each query followed by an answer.
- **The actual query** includes contextual information about the task, which the LLM uses to execute the tasks described in the instructions. EPScan gathers the container's startup commands and environment variables as the query. Furthermore, if the startup command includes a path to a shell script, EPScan extracts the content of the shell script from the container image to enrich the query.

After generating these components, EPScan assembles them to the prompt for the LLM to complete the queried answer. Figure 7 in §A.2 includes an example prompt used by EPScan. Ultimately, EPScan extracts executable files from the container image based on the list of executables provided by the LLM.

Source-code Entry Location. After obtaining the executable running in the pod, EPScan locates the code entry of the executable in the source code repository via the runtime information within the Go executables. Go embeds the runtime information of all functions into the executables during the linking phase [62] to support mechanisms such as unwinding. The runtime information of the function includes the file path of the function's source code. This allows us to extract the source file path of the *main* function from the executable, and find the corresponding file in the source code. Note that this information is retained even when the executable binary is stripped, enabling us to locate the code location of the compiled function easily. To be specific, EPScan first extracts the source code file path of the *main*

function. Afterward, EPScan leverages the path to locate the corresponding source file in the project source code. EPScan then identifies the *main* function in the source file as the code entry of the executable.

5.4. RefGraph-based Program Behavior Analysis

After determining the code entry of the executable that each pod runs, EPScan performs static program analysis to determine the resource access behavior of the program, thereby obtaining the necessary permissions required by the pod. EPScan first locates the call sites for all resource access APIs and further models access behaviors, including access types (i.e., verbs) and resource types. Subsequently, EPScan constructs the RefGraph of the whole application and employs a RefGraph-based algorithm to determine whether these API call sites are reachable from the code entries.

Resource Access Modeling & Identification. As described in §2.2, Kubernetes provides two libraries [39], [40] for accessing cluster resources in Go. We manually examined all APIs in these two libraries by reviewing the documents and related source code and filtered out those not accessing cluster resources. In all, we collected 1498 resource accessing APIs across the two libraries as shown in Table 2. EPScan identifies all call sites of these collected APIs, models the resource access behavior of each call site, and further determines the permissions required for the behavior.

EPScan models each call site via the type of access operation and the type of resource to operate on. In particular, EPScan takes the function name of the API as the access operation type. For resource type, we summarize three ways that the API determines the input resource type, with an example for each way in Figure 4. We therefore use different methods to analyze each type of API and use different representations to record the resource types.

- **Type-I: Via Receiver Type.** The type of resource to operate on can be directly determined by the type of its receiver (akin to the *this* pointer in other languages). For example, the API *DeploymentInterface.Delete* operates on *deployments* resources, as inferred from its receiver type, *DeploymentInterface*. EPScan captures the type of the receiver to represent the resource type.
- **Type-II: Via Parameter Type.** The resource type is determined through the type of the incoming resource parameter. To achieve polymorphism, some APIs accept an object of a common interface type as its resource parameter and then perform different actions at runtime depending on the actual type of the incoming parameter. For example, the API *Client.Delete* uses the parameter *obj* to intake the object to be deleted, with the parameter type in API signature being interface type *client.Object*. EPScan conducts data flow analysis on the resource parameters to trace how the resource parameter is created and passed. EPScan then locates the first place on the data flow where the object exists as a non-common interface type and treats that type as the resource type.
- **Type-III: Via String in Parameter.** The resource type could also be ascertained through the constant string value of a specific parameter. For instance, the API *NewListWatchFromClient* takes a constant string as the parameter to retrieve the operated resource type, such as *"deployments."* EPScan models these APIs by performing data flow analysis on the parameters to verify if the input parameters are constant strings, collecting these constant strings to represent the resource type.

After locating and modeling the resource access behaviors, EPScan further analyzes the permissions each behavior requires (i.e., a tuple of verbs and resources) based on heuristic rules. For resources, the modeling results of EPScan contain different representations, including receiver types, non-common interface types, and constant strings. Basically, the constant string asserts the corresponding Kubernetes resource. In addition, the naming of receiver/non-common interface types indicates their corresponding Kubernetes resources. For example, the receiver type *DeploymentInterface* corresponds to the deployment resource. Therefore, EPScan infers the standard Kubernetes resources for the modeling results based on the literals, including the type name and the constant string value. For verbs, we constructed a mapping from API functions to permission verbs by auditing the code of each API function. When given a modeled resource access, EPScan queries the mapping to convert it to a (verb, resource) tuple for later excessive permission detection.

```
// Specify the resource type via the receiver type
dep := clientset.AppsV1().Deployments(ns)
dep.Delete(ctx, "demo-deployment", opts)

// Specify the resource type via the parameter type
obj := &appsv1.Deployment{
    ...
}
client.Delete(ctx, obj, opts)

// Specify the resource type via the constant string in
the parameter
lw := cache.NewListWatchFromClient(client,
"deployments", ns, field)
```

Figure 4: Example call sites of three API types.

RefGraph Construction. As introduced in §4.2, EPScan utilizes the RefGraph, which captures the reference relationships between Go functions to approximate the reachability analysis of Kubernetes applications. RefGraph is a directed graph whose nodes represent top-level code elements of Go, including functions, global variables, and type definitions [63]. A function can hold a reference to another function via a directed edge, or via a path with some global variable or type definition on it.

TABLE 2: Modeled resource accessing APIs.

Library	API Type	# of APIs
k8s.io/client-go	Type-I	1478[a]
	Type-II	5
	Type-III	5
sigs.k8s.io/ controller-runtime	Type-II	10

[a] We modeled 311 receiver types with each type having 5 resources accessing API declared approximately.

`EPScan` constructs the RefGraph using the algorithm shown in algorithm 1. The algorithm takes the source code of a Go project as input and is performed on its Abstract Syntax Tree (AST). As demonstrated in lines 1-4, the algorithm initially collects all top-level language elements from the source code to serve as nodes of the RefGraph. Subsequently, the algorithm iterates through each node to create their outgoing edges. For function nodes, the algorithm connects an edge to all functions and global variables referenced in the function body. Moreover, as shown in lines 14-18, the algorithm also connects an edge to the type of object constructed within the function. For global variable nodes, similar to function nodes, the algorithm connects an edge to all functions and global variables referenced in the variable initializer. For type nodes, the algorithm connects an edge to all methods defined on the type, as these methods would be referenced in the virtual function table or called through reflection API.

Reachability Approximation. Subsequently, `EPScan` utilizes the RefGraph to approximate the reachability from resource access API call sites to code entry points, determining the resource access behaviors performed by each pod and the required permissions. Following our insights discussed in the §4.2, we consider that if function A holds a reference to another function B, B is reachable from A. That is, if there exists a directed path in the RefGraph from a given entry point to the function where a resource access occurs, `EPScan` deems the resource access site as reachable from the entry point.

In this way, `EPScan` identifies all resource access sites reachable from each source-code entry identified in §5.3, labeling the required permissions for each pod in a Kubernetes application.

5.5. Excessive Permissions Detection

This phase compares the permissions requested by each pod in the configurations against those required by the program running in the pod, thereby deducing Excessive Permissions (EPs). That is, if some permission is requested in the configuration but no API call sites within the program utilize this permission, `EPScan` labels such permissions as EPs. In addition, considering that not all EPs pose a security risk to the application, `EPScan` further refines the results by filtering and reporting on exploitable EPs.

We determine whether excessive permission is exploitable based on two aspects, the permission type and permission scope. First, we focus only on those permissions that can be exploited to launch the two attacks described in §3.2. Second, the permission scope should be large enough so that it causes harm to the whole cluster or other pods. In particular, Kubernetes defines three levels of permission scope. For *cluster* scope, the permission can operate on all resources within the cluster; for *namespace* scope, it can operate on all resources within a specific namespace; for *resource-specific* scope, it can operate only on resources with specific names. In conclusion, we summarize the list of

Algorithm 1: Construction of RefGraph

Input: The source code $source$ of the project.
Output: The reference graph (V, E), which consists of a vertex set V and an edge set E.

```
1  functions ← GetAllFunctions(source);
2  globalvars ← GetAllGlobalVars(source);
3  types ← GetAllStructTypes(source);
4  V ← functions ∪ globalvars ∪ types;
5  E ← ∅;
6  for func in functions do
7      ast ← GetFunctionBody(func);
8      for rfunc in GetReferenced_Function(ast) do
9          E ← E ∪ {(func, rfunc)};
10     end
11     for rgvar in GetReferenced_GlobalVar(ast) do
12         E ← E ∪ {(func, rgvar)};
13     end
14     for callee in GetCallee(func) do
15         if callee is a constructor of struct type T
            then
16             E ← E ∪ {(func, T)};
17         end
18     end
19 end
20 for gvar in globalvars do
21     ast ← GetInitializer(gvar);
22     for rfunc in GetReferenced_Function(ast) do
23         E ← E ∪ {(gvar, rfunc)};
24     end
25     for rgvar in GetReferenced_GlobalVar(ast) do
26         E ← E ∪ {(gvar, rgvar)};
27     end
28 end
29 for T in types do
       /* Iterates all methods defined
          on type T.                          */
30     for method in GetTypeMethod(T) do
31         E ← E ∪ {(T, method)};
32     end
33 end
```

exploitable EPs in Table 1, specifying the type and scope for each exploitable EP. EPScan extracts the scope of each EP from the configuration file and filters the harmless EPs from the detected results based on the list.

To be specific, the scope of EP is identified based on the bound method and its rule definition. The EPs bound via *ClusterRoleBinding* are identified as *cluster* scope, and those bound via *RoleBinding* are identified as *namespace* scope. Moreover, if the rules definition of EPs specifies the names of resource objects, the EP is identified as *resource-specific* scope regardless of bound method.

6. Evaluation

In this section, we evaluate EPScan in various aspects. First, we evaluate the effectiveness and efficiency of EPScan in detecting excessive RBAC permissions in real Kubernetes apps. Then, we evaluate the effectiveness of each key component of EPScan's pod-oriented behavior analysis. At last, we present two case studies to demonstrate the security risks of the excessive permissions discovered by EPScan.

In summary, our evaluation is organized by answering the following research questions:

- **RQ1:** How effective is EPScan at detecting excessive permissions, and what are the security impacts of the detected issues? ((in §6.2))
- **RQ2:** How efficient is EPScan in performing the end-to-end analysis? ((in §6.3))
- **RQ3:** How effective is EPScan at matching pods with program entries? ((in §6.4))
- **RQ4:** How effective is EPScan at locating and modeling resource access in Kubernetes apps? ((in §6.5))
- **RQ5:** How effective is EPScan at approximating the reachability of resource access? ((in §6.5))

6.1. Experimental Setup

Prototype Implementation. We implemented a prototype of EPScan targeting third-party Kubernetes applications written in Go. Specifically, the program analysis of EPScan is built upon CodeQL [24], which is a scalable static analysis framework supporting various languages. The other parts of EPScan such as parsing the configuration and detecting excessive permissions are realized in Python. In all, the prototype consists of 928 lines of CodeQL code and 7,958 lines of Python code. In addition, EPScan leverages *gpt-4o* as the distillation LLM to perform Pod-Program and we consumed about 280k tokens during the evaluation. All the experiments are run on a Ubuntu 20.04 machine with an Intel Xeon Gold 6242 processor and 245 GB memory.

CNCF App Dataset. Considering that CNCF is currently the largest (with 240k+ contributors) and most influential open-source community of cloud-native computing, we select the third-party Kubernetes apps in CNCF projects[5] as the main dataset to evaluate EPScan. Specifically, after manually checking the project's introduction, we identified 121 third-party apps for Kubernetes out of the 187 projects in the CNCF project list [5]. In addition, since EPScan requires source code and configuration files as input, we further reviewed the project's homepage to locate the project's source code repository, as well as reading the project's deployment documentation to obtain the complete application configuration. Finally, we collect the configuration files and source code of 108 apps as the dataset, as shown in Table 3.

TABLE 3: Breakdown of CNCF App dataset.

Stars	**Apps**	**Pods**[a]	**LoCs**(K)
High ($\geq$5000)	39	120	4426
Medium ($<$5000)	44	112	6292
Low ($<$1000)	25	50	1882
Total	108	282	12600

[a] We only count the pods running programs written in Go.

6.2. RQ1: Excessive Permission Detection

We applied EPScan to the CNCF App datasets to detect excessive permissions within the applications. In total, EPScan identified exploitable EPs in 112 Pods across 55 Apps.

Verification. We manually inspected whether the exploitable EPs reported by EPScan are indeed unnecessary for the application. We examined the related application configurations and source code for each reported pod. We first inspected if the relevant pod had requested these permissions in the configuration. Subsequently, we examined the source code for resource accessing API usage that utilizes these permissions. If such usage was found, we further examined whether these API call sites were reachable from the pod. Overall, we confirm that 106/112 (94.6%) pods do have true exploitable EPs.

Security Impacts. As introduced in §3.2, by exploiting the EPs, attackers could potentially gain control of the entire cluster, control worker nodes, leak sensitive information, or launch DoS attacks. We categorized the 106 pods according to the security impact of their EPs, as shown in Table 4. Note that existing work [17] only targets the EPs that could be exploited to take over the whole cluster and misses a lot of EPs that also lead to severe consequences.

EP Disclosure. We responsibly reported all the discovered EPs to the developers. As shown in Table 5, the developers have confirmed the issues in 39 pods across 16 Apps and we will keep in constant communication with the developers about other issues. We also worked with the developers to report the confirmed issues to CVE, and to date, 9 CVEs have been assigned.

False Positive Analysis. Overall, EPScan tends to produce false positives rather than false negatives. This is mainly because the permissions requested by the pod are retrieved through configuration analysis, which is precise. However, determining whether these permissions are excessive is based on the potentially failing pod behavior

TABLE 4: Security Impact of the discovered EPs.

Model	Impact	# of Pod	# of App
I	Take over the whole cluster	82	36
I, II	Take over worker node	77	36
I, II	Leak sensitive information	56	27
I, II	Perform DoS attack	69	34

TABLE 5: Confirmation status of developer report.

Confirmation Status	Apps	Pods
Confirmed	16	39
Pending	9	12
No Respond	27	55
Total	50[a]	106

[a] The sum of Apps exceeds the total number because the confirmation status may differ for pods in the same App.

analysis. Specifically, unrecognized resource access behavior will result in permissions being incorrectly marked as excessive, leading to false positives. To further understand the false positives reported by EPScan, we examined the false positives in the remaining 6 pods. We categorize the reasons as follows:

- **Data flow analysis failures** affected four Pods. Some resource access APIs in these projects construct resource objects in a highly complex manner, which prevents data flow analysis from determining the type of operated resources. For example, *vineyard* project [64] stores the configurations of resources to be created in YAML format. At runtime, *vineyard* dynamically parses these YAML files into resource objects and passes them to the API. It is impossible for static analysis to determine the resource type from this kind of API usage.
- **Imprecision of CodeQL** affected two Pods. CodeQL may miss references to a global variable or function in corner-case scenarios. As a result, some global variables and functions are not recognized by EPScan, causing several edges to be lost in the RefGraph.

6.3. RQ2: Efficiency

We evaluated the efficiency of EPScan on the CNCF App Dataset. Overall, EPScan required approximately 27 hours to perform sensitive EP detection across 108 apps, averaging 14 minutes of analysis per application. As shown in Table 6, the pod-oriented behavior analysis is the most time-consuming among all the components. Specifically, before running our analysis, EPScan first builds the code database from the source code repository with the CodeQL toolchain, which takes an average of 13 minutes. Once the code database is built, thanks to the intermediate results cached by CodeQL, our analysis is finished in a short time, averaging 1 minute per app.

Compared to manual analysis, EPScan significantly reduces the time overhead of excessive permission detection. For instance, Yang *et al.* [17] conducted a similar analysis on 153 CNCF projects, which took two months. Since Yang *et al.* only automated the analysis of configuration files, they had to manually audit the source code to identify EPs, which is labor-intensive. Moreover, during the process of reporting the detected EPs to the community, several developers pointed out that analysis of the necessary permissions requires significant effort for code review. We believe that EPScan eases this problem and greatly helps researchers and developers to identify EPs in a large scale of applications.

6.4. RQ3: Pod-Program Matching

We evaluated the effectiveness of the pod-program matching of EPScan. We sampled 80/282 (28.4%) pods from the CNCF App dataset and manually constructed the ground-truth for pod-program matching. Specifically, we devoted great efforts to reviewing the entire application compilation and deployment process, including compilation files, image build scripts, pod configuration files, and container startup scripts, to track the main function entries in the source code of the executables running in each pod.

Besides, to further demonstrate the contribution of our proposed LLM-assisted pod-program matching, we have implemented a variant of EPScan, named $\texttt{EPScan}_{rule}$, which employs simple heuristic rules to match pods with programs. Since the dynamic approach suffers from the usability issue described in §4.1, we implemented $\texttt{EPScan}_{rule}$ based on the static approach. In particular, $\texttt{EPScan}_{rule}$ models the common patterns of container startup commands and locates executables from startup commands based on regular matches. Then $\texttt{EPScan}_{rule}$ searches the project repository for a path containing the executable's name and takes the main function under that file path as the program entry of the executable. This heuristic rule is based on the fact that developers usually incorporate component names (often also used as executable names) into file paths, such as *cmd/chaos-daemon/main.go*. It is worth mentioning that the compile command can accurately match executables and program entries. However, applications usually use complex CI/CD toolchains for compilation, making it difficult to match compile commands by text. In our dataset, only 35 apps (32.4%) matched at least one compile command. Therefore, we build $\texttt{EPScan}_{rule}$ with the aforementioned heuristic rules for matching.

We evaluated EPScan and $\texttt{EPScan}_{rule}$ on the ground-truth dataset. As presented in Table 7, EPScan successfully

TABLE 6: Time overhead of each component of EPscan.

	Pod-Oriented Program Analysis		Excessive Permission Detection	Configuration Analysis	Total
	CodeQL database Build	Other Analysis			
Average	13min18s	1min26s	<1s	<1s	14min45s
Median	9min38s	54s	<1s	<1s	10min8s

identified the source-code entry points of 71 pods, achieving an accuracy rate of 88.8%. In contrast, $\texttt{EPScan}_{rule}$ identified the source-code entry points for 39 pods, marking a 45% decrease in accuracy compared to EPScan. This indicates that our LLM-assisted method effectively improves the matching between pods and programs. Note that the precision of $\texttt{EPScan}_{rule}$ is slightly higher than that of EPScan. This is primarily due to that $\texttt{EPScan}_{rule}$'s heuristic rules sometimes extract no executable program from the startup command, thus giving no results. Consequently, this leads to a significantly lower recall rate for $\texttt{EPScan}_{rule}$ compared to EPScan. We further analyzed the causes of the failed cases of EPScan. We found that the 7 false positives due to a special form of code organization. In particular, the target application consists of multiple programs, each having its own repository. The main function of each program is located in the same path in its repository, so EPScan cannot distinguish between them with the file path extracted from the executable. For the 2 false negatives, EPScan fails to extract the startup shell script from the container image since the application mounts them at runtime.

TABLE 7: The effectiveness of EPScan in Pod-Program Matching.

	EPScan	**$\texttt{EPScan}_{rule}$**
TP / FP / FN	71 / 7 / 2	39 / 1 / 40
Acc(%)	**88.8**	48.8
Prec(%)	91.0	**97.5**
Recall(%)	**97.2**	49.3
F1	**94.0**	65.5

6.5. RQ4: Resource Access Modeling & Identification

We evaluated the effectiveness of the resource access modeling & identification of EPScan. We manually construct the ground-truth dataset by reviewing the source code and labeling the program position that triggers resource access as long as the RBAC permission the access requires. Considering such code reviewing demands a significant amount of human effort, we sampled 10/108 applications from the CNCF App dataset and labeled at most 50 program positions for each app. We finally labeled 427 program positions among the sampled apps.

We evaluated EPScan on the ground-truth dataset. As presented in Table 8, EPScan successfully located and modeled the resource usage at 410 labeled program positions, achieving a recall of 96.0%. We further checked the 17 program positions EPScan failed to model. Two of these resource usages are from *dex* project [65]. Since *dex* only accesses a few resources, the developers implement their own wrappers to send HTTP requests accessing the Kubernetes API, instead of using libraries like client-go. For the other 15 resource accesses, EPScan failed to model them because of the inaccurate data flow analysis, similar to what we discussed in §6.2. This indicates the limitations of EPScan in resource access analysis. Future work may improve by using more sophisticated static analysis approaches.

TABLE 8: The effectiveness of EPScan in Resource Usage Locating&Modeling.

Project	**Sample**	**EPScan**	**Recall(%)**
bfe	32	32	100.0
chaos-mesh	50	43	86.0
dapr	50	49	98.0
dex	2	0	0.0
hwameistor	50	49	98.0
kubewarden	43	42	97.7
longhorn	50	50	100.0
prometheus	50	50	100.0
submariner	50	45	90.0
volcano	50	50	100.0
Total	427	410	96.0

6.6. RQ5: Reachability Approximation

We evaluated the effectiveness of the reachability approximation of EPScan in identifying whether a resource access site is reachable from a specific main entry. To construct the ground-truth dataset, we take the resource access site from the dataset labeled in §6.5 as targets and manually pinpoint the main entry point that reaches the targets by carefully reviewing the source code. Considering such code reviewing demands great human efforts, we labeled at most ten target sites for each application. In all, the ground-truth dataset consists of 92 target sites, with each function associated with 1.13 main entries on average.

To highlight the benefits of our RefGraph-based reachability analysis, we compared it with traditional callgraph-based analysis. In particular, we employed two *state-of-the-art* callgraph construction algorithms for Go, RTA [56] and VTA [57]. Then a reachability analysis is performed on the constructed call graph to determine the main entries that can reach the target resource access.

We evaluated EPScan and the baseline methods on the ground-truth dataset. As shown in Table 9, it turns out

that the RefGraph-based analysis of EPScan surpasses the callgraph-based analysis in terms of both precision and recall. Both RTA and VTA perform type and data flow analysis to extract the information needed for virtual call analysis. However, their analysis is inaccurate and incomplete in Kubernetes applications written in Go, resulting a low recall and precision. We further checked the case that EPScan failed to analyze and found these errors were primarily due to the imprecision of CodeQL, similar to what we discussed in §6.2.

TABLE 9: The effectiveness of EPScan in Reachability Approximation.

	EPScan	RTA	VTA
Prec(%)	**98.9**	50.0	77.8
Recall(%)	**98.9**	39.4	33.3
TP	90	26	28
FP	1	26	8
FN	1	40	56

6.7. Case Study

In this section, we demonstrate how the excessive permissions detected by EPScan would compromise the security of a Kubernetes cluster through a case study. The case is a real-world exploitable excessive permission reported by EPScan in the *submariner* project [38]. *submariner* is a Kubernetes application that aims to facilitate network communication across multiple clusters. It has garnered 2.4k stars on GitHub to date. As illustrated in Figure 6 in §A.1, *submariner* created a distinct *Role* object for the *route-agent* component, which grants all verbs for the *deployments* resource. However, EPScan discovered that the *route-agent* never created deployment objects and identified the *(create, deployments)* as an excessive permission. We reported our findings to the project maintainers and received confirmation as well as one CVE assignment.

Then we present how to exploit this excessive permission to control the worker nodes within the cluster from a compromised application as described in §3.2 Model-II. The overall exploitation procedure is shown in Figure 5. The procedure consists of three steps. ❶ First the attacker can directly grab the service account token from the token file mounted by Kubernetes from the compromised application. ❷ Once the attacker grabs the SA token, he could exploit the *(create, deployments)* permission to create privileged containers with malicious images. In particular, the attacker can specify *securityContext.privileged* in the *deployment* object to create privileged containers and set *containers.image* to a malicious image prepared by the attacker. ❸ Once the attacker creates a privileged container, he can follow the exploitation proposed by previous researchers [66] to achieve container escape and privilege escalation without any vulnerabilities. This exploitation process demonstrates that an attacker occupying a container can easily abuse an exploitable EP to compromise most of the worker nodes in the cluster, fully illustrating the great harm that EP can cause. A more detailed introduction about the exploitation is available at §A.3.

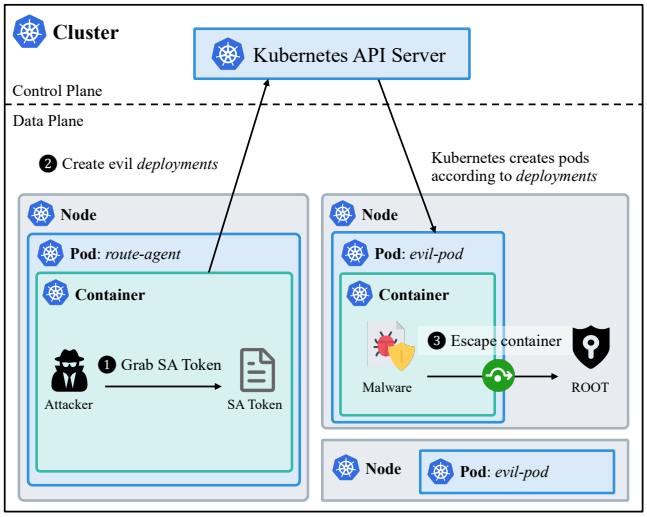

Figure 5: Attack steps of *submariner* case.

7. Related Work

7.1. Attacks in Kubernetes

In recent years, as Kubernetes has become the dominant tool for automated container orchestration, a lot of research on Kubernetes attacks has arisen. Shamim *et al.* [67], [16] study the best practices of Kubernetes and explore how to exploit the violations of Kubernetes security best practices. Sushring *et al.* [13] showed that attackers can launch co-residency attacks with a victim application from inside state-of-the-art containers running on Kubernetes. Minna *et al.* [68] discussed possible attack scenarios against Kubernetes networking. Zeng *et al.* [69] presented a full-stack vulnerability analysis of the cloud-native platform, including 30 vulnerabilities of Kubernetes. All these works do not consider the attack surface of Kubernetes permissions. Recently, Yang *et al.* [17] proposed that attackers could take over the whole Kubernetes cluster by exploiting excessive permissions of third-party applications. However, the attack proposed by Yang *et al.* requires a strong condition (i.e., control one worker node). Our work extends this attack surface to allow the attacker to abuse EPs under a weaker condition (i.e., control one pod) and proposes more exploitable EPs. To identify excessive permission, Yang *et al.* proposed a semi-automatic approach and spent two months examining 53 applications. However, the tool developed does not analyze the application program and requires application installation, resulting in huge human effort. Our work proposes a novel static approach that can automatically complete the analysis of 108 applications in 27 hours.

7.2. Kubernetes Security Analysis Tool

Given that Kubernetes faces numerous security threats, many security analysis works have emerged in recent years to secure Kubernetes. Haque *et al.* [14] presented a learning-based approach to build a secured configuration knowledge graph, which could be utilized for Kubernetes misconfiguration mitigation. Rahman *et al.* [15] constructed a static analysis tool to identify security misconfigurations in Kubernetes manifests while Dell'Immagin *et al.* [70] aim to detect security smells in Kubernetes-deployed microservices.

In addition, there exist lots of production-ready tools for Kubernetes security analysis: Kubescape [71], Checkov [72], [73], Datree [74], [75], KICS [76], [77], Kube-bench [78], Kube-linter [79], kube-score [80], kubeaudit [81], KubiScan [82], Kubesec.io [83], Kube-hunter [84] and Sonarqube [85]. These tools implement kinds of static analysis of Kubernetes configuration files to detect known security issues. However, none of the existing security analysis tools can systematically analyze the correctness of Kubernetes RBAC configuration. Our work fills that gap.

7.3. Excessive Permission Detection

In the past decade, *Principle of least privilege* has been well discussed on kinds of modern software systems [86], [87], [88], [20]. There is a body of work [43], [44], [45], [46], [47] that points out excessive permissions given to apps can lead to serious security risks, including enabling privilege escalation [45] , breaking the Android privacy guarantees [46] and acquiring critical system capabilities [47]. In order to mitigate such security risks, many recent studies have focused on detecting excessive permissions (a.k.a., least privilege violations). Some works [19], [20], [21], [22], [23] identify unnecessary permissions requests of mobile apps based on the app descriptions and their API usage. Wang *et al.* [89] verify if the capabilities of mobile apps in the cloud exceed the legitimate needs of the apps. While most work focused on the mobile apps and the Android permission system, our work targets the unnecessary permissions of Kubernetes apps which breaks the access control mechanism in Kubernetes. In addition, we propose several new techniques to facilitate the analysis based on the characteristics of Kubernetes apps.

8. Conclusion

In this paper, we proposed `EPScan`, a novel approach designed for detecting exploitable excessive permissions in third-party applications of Kubernetes. `EPScan` employs a pod-oriented program analysis to accurately identify permissions required by the executables running in each pod. Overall, `EPScan` identified exploitable excessive permissions among 50 third-party applications from the CNCF projects, with a precision of 94.6%. In addition, we propose a new attack model that requires easier-to-achieve conditions than the existing excessive permission attacks. We reveal that attackers who compromise an application in a container can exploit excessive permissions to break Kubernetes isolation, which allows them to take control of worker nodes or steal sensitive information from other containers.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (62172105, 62202106, 62102093, 62302101, 62172104, 62102091, 62472096, 62402114, 62402116). Yuan Zhang and Min Yang are the corresponding authors. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] Overview of kubernetes. https://kubernetes.io/docs/concepts/overview/ [Accessed on 12 April 2024].

[2] Case studies of kubernetes. https://kubernetes.io/case-studies/ [Accessed on 12 April 2024].

[3] From serverless containers to serverless ubernetes. https://www.alibabacloud.com/blog/from-serverless-containers-to-serverless-kubernetes_596533 [Accessed on 12 April 2024].

[4] Kubernetes ecosystem: A comprehensive overview of kubernetes tools and technologies. https://sidglobalsolutions.medium.com/kubernetes-ecosystem-a-comprehensive-overview-of-kubernetes-tools-and-technologies-c15da06d8320 [Accessed on 12 April 2024].

[5] Graduated and incubating projects. https://www.cncf.io/projects/ [Accessed on 12 April 2024].

[6] Artifacthub. https://artifacthub.io/ [Accessed on 12 April 2024].

[7] Project website of keda. https://keda.sh/ [Accessed on 12 April 2024].

[8] Github of keda. https://github.com/kedacore/keda [Accessed on 12 April 2024].

[9] Proactive autoscaling of kubernetes workloads with keda using metrics ingested into amazon managed service for prometheus. https://aws.amazon.com/cn/blogs/mt/proactive-autoscaling-kubernetes-workloads-keda-metrics-ingested-into-aws-amp/ [Accessed on 12 April 2024].

[10] Simplified application autoscaling with kubernetes event-driven autoscaling (keda) add-on. https://learn.microsoft.com/en-us/azure/aks/keda-about [Accessed on 12 April 2024].

[11] Kubernetes api. https://kubernetes.io/docs/reference/kubernetes-api/ [Accessed on 12 April 2024].

[12] Accessing the kubernetes api from a pod. https://kubernetes.io/docs/tasks/run-application/access-api-from-pod/ [Accessed on 12 April 2024].

[13] 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

[14] M. U. Haque, M. M. Kholoosi, and M. A. Babar, "Kgsecconfig: A knowledge graph based approach for secured container orchestrator configuration," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 420–431. [Online]. Available: https://doi.org/10.1109/SANER53432.2022.00057

[15] 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

[16] 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

[17] 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

[18] Github of excessivepermissionattack. https://github.com/XDU-SysSec/ExcessivePermissionAttack/ [Accessed on 12 April 2024].

[19] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 217–228. [Online]. Available: https://doi.org/10.1145/2382196.2382222

[20] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 627–638. [Online]. Available: https://doi.org/10.1145/2046707.2046779

[21] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: towards automating risk assessment of mobile applications," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13. USA: USENIX Association, 2013, p. 527–542.

[22] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1354–1365. [Online]. Available: https://doi.org/10.1145/2660267.2660287

[23] S. Zhang, H. Lei, Y. Wang, D. Li, Y. Guo, and X. Chen, "Apimind: Api-driven assessment of runtime description-to-permission fidelity in android apps," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023, pp. 427–438.

[24] The Open Source of CodeQL in Github. https://codeql.github.com/ [Accessed on 12 April 2024].

[25] Production-grade container orchestration. https://kubernetes.io/ [Accessed on 12 April 2024].

[26] Kubernetes – architecture. https://www.geeksforgeeks.org/kubernetes-architecture/ [Accessed on 12 April 2024].

[27] Cluster architecture. https://kubernetes.io/docs/concepts/architecture/ [Accessed on 12 April 2024].

[28] Pods. https://kubernetes.io/docs/concepts/workloads/pods/ [Accessed on 12 April 2024].

[29] Exploring 10 essential 3rd party tools for kubernetes management. https://medium.com/@vinoji2005/exploring-10-essential-3rd-party-tools-for-kubernetes-management-115c7ab5e17f [Accessed on 12 April 2024].

[30] Rook. https://rook.io/ [Accessed on 12 April 2024].

[31] Search packages on artifacthub with helm-charts filter. https://artifacthub.io/packages/search?kind=0&sort=relevance&page=1 [Accessed on 12 April 2024].

[32] Search packages on artifacthub with only-operators filter. https://artifacthub.io/packages/search?operators=true&sort=relevance&page=1 [Accessed on 12 April 2024].

[33] Google kubernetes engine (gke). https://cloud.google.com/kubernetes-engine?hl=en [Accessed on 12 April 2024].

[34] Amazon eks add-ons. https://docs.aws.amazon.com/eks/latest/userguide/eks-add-ons.html [Accessed on 12 April 2024].

[35] Add-ons, extensions, and other integrations with azure kubernetes service (aks). https://learn.microsoft.com/en-us/azure/aks/integrations [Accessed on 12 April 2024].

[36] Isolate your workloads in dedicated node pools. https://cloud.google.com/kubernetes-engine/docs/how-to/isolate-workloads-dedicated-nodes [Accessed on 12 April 2024].

[37] Kubernetes api concepts. https://kubernetes.io/docs/reference/using-api/api-concepts/ [Accessed on 12 April 2024].

[38] Project website of submariner. https://submariner.io/ [Accessed on 12 April 2024].

[39] client-go. https://github.com/kubernetes/client-go/ [Accessed on 12 April 2024].

[40] Kubernetes controller-runtime project. https://github.com/kubernetes-sigs/controller-runtime [Accessed on 12 April 2024].

[41] Using rbac authorization. https://kubernetes.io/docs/reference/access-authn-authz/rbac/ [Accessed on 12 April 2024].

[42] Namespaces. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ [Accessed on 12 April 2024].

[43] Y. Aafer, X. Zhang, and W. Du, "Harvesting inconsistent security configurations in custom android roms via differential analysis," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16. USA: USENIX Association, 2016, p. 1153–1168.

[44] S. A. Gorski and W. Enck, "Arf: identifying re-delegation vulnerabilities in android system services," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 151–161. [Online]. Available: https://doi.org/10.1145/3317549.3319725

[45] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, "Android custom permissions demystified: From privilege escalation to design shortcomings," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 70–86.

[46] A. Aldoseri, D. Oswald, and R. Chiper, "A tale of four gates: Privilege escalation and permission bypasses on android through app components," in *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 233–251. [Online]. Available: https://doi.org/10.1007/978-3-031-17146-8_12

[47] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "Hare hunting in the wild android: A study on the threat of hanging attribute references," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1248–1259. [Online]. Available: https://doi.org/10.1145/2810103.2813648

[48] Multi-tenancy. https://kubernetes.io/docs/concepts/security/multi-tenancy/#isolation [Accessed on 12 April 2024].

[49] Tenant isolation. https://aws.github.io/aws-eks-best-practices/security/docs/multitenancy/#isolating-tenant-workloads-to-specific-nodes [Accessed on 12 April 2024].

[50] Kubernetes api concepts. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/ [Accessed on 12 April 2024].

[51] Assigning pods to nodes. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/ [Accessed on 12 April 2024].

[52] Secrets. https://kubernetes.io/docs/concepts/configuration/secret/ [Accessed on 12 April 2024].

[53] Tesla cloud resources are hacked to run cryptocurrency-mining malware. https://arstechnica.com/information-technology/2018/02/tesla-cloud-resources-are-hacked-to-run-cryptocurrency-mining-malware/ [Accessed on 12 April 2024].

[54] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[55] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[56] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 324–341. [Online]. Available: https://doi.org/10.1145/236337.236371

[57] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 264–280, 2000.

[58] Kubernetes api reference. https://kubernetes.io/docs/reference/kubernetes-api/ [Accessed on 12 April 2024].

[59] Request verbs and authorization. https://kubernetes.io/docs/reference/access-authn-authz/authorization/#determine-the-request-verb [Accessed on 12 April 2024].

[60] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, and A. Askell, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[61] Prompt engineering. https://platform.openai.com/docs/guides/prompt-engineering [Accessed on 12 April 2024].

[62] Go 1.2 runtime symbol information. http://golang.org/s/go12symtab [Accessed on 12 April 2024].

[63] The go programming language specification. https://go.dev/ref/spec [Accessed on 12 April 2024].

[64] Project website of vineyard. https://v6d.io/ [Accessed on 12 April 2024].

[65] Project website of dex. https://dexidp.io/ [Accessed on 12 April 2024].

[66] Bad pods: Kubernetes pod privilege escalation. https://bishopfox.com/blog/kubernetes-pod-privilege-escalation/ [Accessed on 12 April 2024].

[67] M. S. I. Shamim, F. A. Bhuiyan, and A. A. U. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," *2020 IEEE Secure Development (SecDev)*, pp. 58–64, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:220250875

[68] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the security implications of kubernetes networking," *IEEE Security & Privacy*, vol. 19, no. 5, pp. 46–56, 2021.

[69] 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

[70] G. Dell'Immagine, J. Soldani, and A. Brogi, "Kubehound: Detecting microservices' security smells in kubernetes deployments," *Future Internet*, vol. 15, no. 7, 2023. [Online]. Available: https://www.mdpi.com/1999-5903/15/7/228

[71] Kubescape. https://kubescape.io/f [Accessed on 12 April 2024].

[72] Github repo of checkov. https://github.com/bridgecrewio/checkov [Accessed on 12 April 2024].

[73] Checkov. https://www.checkov.io/ [Accessed on 12 April 2024].

[74] Github repo of datree. https://github.com/datreeio/datree [Accessed on 12 April 2024].

[75] Datree. https://hub.datree.io/ [Accessed on 12 April 2024].

[76] Github repo of kics. https://github.com/Checkmarx/kics [Accessed on 12 April 2024].

[77] Kics - keeping infrastructure as code secure. https://www.kics.io/index.html [Accessed on 12 April 2024].

[78] kube-bench. https://aquasecurity.github.io/kube-bench/v0.6.15/ [Accessed on 12 April 2024].

[79] kube-linter. https://github.com/stackrox/kube-linter [Accessed on 12 April 2024].

[80] kube-score. https://github.com/zegl/kube-score [Accessed on 12 April 2024].

[81] kubeaudit. https://github.com/Shopify/kubeaudit [Accessed on 12 April 2024].

[82] Kubiscan. https://github.com/cyberark/KubiScan [Accessed on 12 April 2024].

[83] Kubesec.io: Security risk analysis for kubernetes resources. https: https://kubesec.io/ [Accessed on 12 April 2024].

[84] kube-hunter. https://aquasecurity.github.io/kube-hunter/ [Accessed on 12 April 2024].

[85] Sonarqube. https://www.sonarsource.com/products/sonarqube/ [Accessed on 12 April 2024].

[86] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2008.

[87] S. Motiee, K. Hawkey, and K. Beznosov, "Do windows users follow the principle of least privilege? investigating user account control practices," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*, 2010, pp. 1–13.

[88] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen, "Webjail: least-privilege integration of third-party components in web mashups," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 307–316.

[89] X. Wang, Y. Sun, S. Nanda, and X. Wang, "Credit karma: understanding security implications of exposed cloud services through automated capability inference," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.

[90] Project website of kubevela. https://kubevela.io/ [Accessed on 12 April 2024].

[91] Kubernetes admission controllers reference. https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/ [Accessed on 12 April 2024].

Appendix A.

A.1. Config of *submariner*

```
...
metadata:
  name: submariner-routeagent
rules:
  - apiGroups:
      - apps
    resources:
      - deployments
      - daemonsets
      - replicasets
      - statefulsets
    verbs:
      - '*'
...
```

Figure 6: Part of the configuration of *route-agent*.

A.2. Prompt example

Figure 7 illustrates the prompts used by `EPScan` when analyzing the *kubevela* project [90]. For simplicity, some examples and parts of lengthy code have been omitted.

Since *gpt4-o* uses roles to differentiate between system instructions (*system*), user input (*user*), and LLM responses (*assistant*). According to best practices [61], we assigned the *system* role to the task instructions, the *user* role to example queries in K examples and the actual query, and the *assistant* role to example answers in K examples.

A.3. A detailed case study of CVE-2024-5042

Here, we provide a detailed analysis of CVE-2024-5042, including its root cause and the complete exploitation process.

Root Cause. We investigated how the excessive permission was introduced into the *route-agent* component. Through inspecting the Git logs and communicating with the maintainers, we found that the permission rules presented in Figure 6 were originally introduced four years ago, and designed for the *operator* pod. At that time, *submariner* configured a sharing service account for all pods, which caused the *route-agent* to be mistakenly granted excessive permission, leading to the exploitable EPs `EPScan` detected. Three years ago, the community refactored the permission configuration in light of the potential security risks and created a separate service account for each pod. Unfortunately, during the refactoring process, the developers did not scrutinize the least permissions for each pod. Some permission rules in the *operator* were copied directly into the *route-agent* without detailed auditing, leading exploitable EPs kept after refactoring. This indicates that Kubernetes application developers do not pay enough resources to the security of permission management, which motivates our work to facilitate permission management in third-party applications.

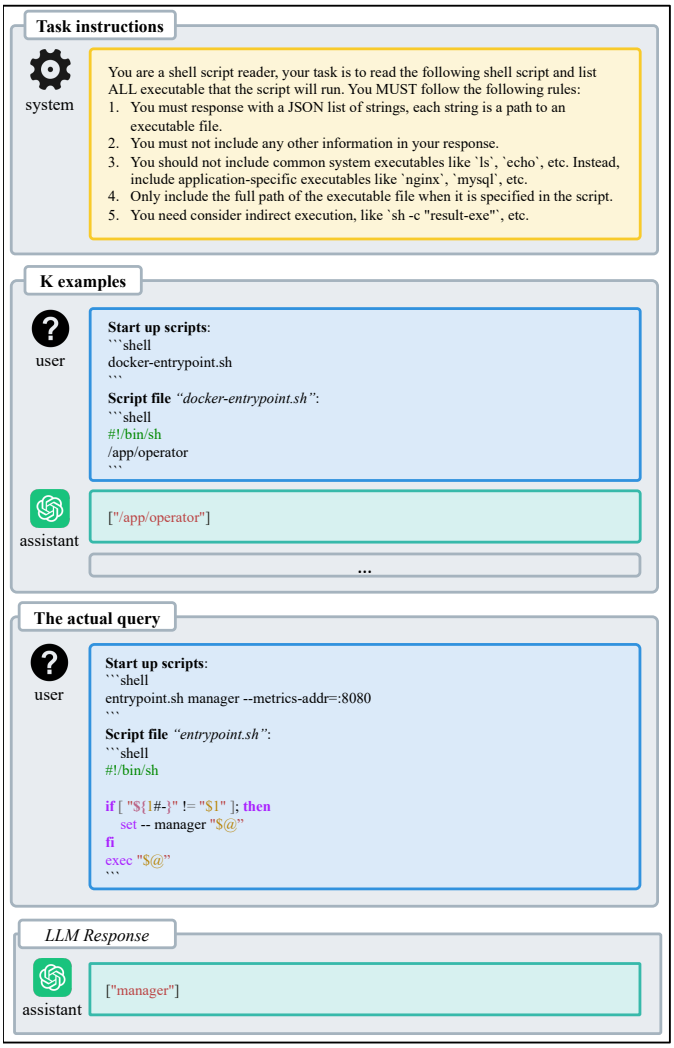

Figure 7: Example prompt and response of *kubevela*.

Exploitation. We then demonstrate how to exploit CVE-2024-5042 to control the worker nodes within the cluster under Model-II described in §3.2. To exploit this EP, an attacker must first obtain the service account token of the *route-agent*. In Model II, since the attacker has compromised one application running in a container, he can directly read the contents of the service account token file mounted by Kubernetes within the container. Once the token is obtained, the attacker obtains the *(create, deployments)* permission and is able to make requests to the Kubernetes API Server to create *deployment* resources.

Since *submariner* does not use admission controllers [91] to restrict deployment creation, the attacker can exploit the *(create, deployments)* permission to create privileged containers with malicious images by configuring two critical fields. First, the attacker can specify *securityContext.privileged* in the *deployment* object to create privileged containers. Second, when creating containers for *deployment* objects, Kubernetes pulls the image from Dockerhub based

on the *containers.image* field. Thus, the attacker can set the *containers.image* to a malicious image prepared for the attack, making the Kubernetes cluster deploy a container that the attacker has full control of. In addition, *deployment* resource has a *replicas* setting for creating multiple identical containers. Since Kubernetes tends to distribute replica containers across different nodes, an attacker can deploy the malicious container to most worker nodes by specifying a large number of replicas.

Once the attacker has full control of a privileged container, he can follow the exploitation process proposed by previous researchers [66] to achieve container escape and privilege escalation, thereby obtaining root access to the host worker nodes. All exploit scripts are publicly available and the process does not rely on any vulnerabilities, as privileged containers inherently can create root processes within the host PID namespace.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This work describes a system (`EPScan`) for automated static analysis of RBAC policies in Kubernetes pods. `EPScan` uses an LLM to identify a pod's entry points, and then uses the CodeQL framework to statically determine the resources that the pod may access.

The authors perform a large-scale measurement study towards both evaluating their tool and characterizing the prevalence of RBAC-related configuration weaknesses. The paper applies `EPScan` to 108 third-party applications and discovers excessive permissions in 50 with high precision. The paper also demonstrates how an attacker that has compromised an application can abuse excessive permissions to further compromise worker nodes without first needing a container escape.

B.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The work provides independent confirmation of important results. Although excessive permission vulnerabilities are well-known and exist by-design in Kubernetes, this study presents an automated analysis and measurement study of these issues in practice.
2) The work creates a new tool to enable future science. `EPScan` is a practical utility for analyzing the permission usage of applications, and could be incorporated by practitioners into automated analysis of their kubernetes pods. Such an automated approach presents a valuable step forward in the established field of Kubernetes security.

B.4. Noteworthy Concerns

1) The large language model approach described in the work does not meaningfully contribute to the paper's results. Roughly half of entrypoints discovered with the LLM are trivial examples and the remainder could be manually identified. As a tool for practitioners,

manually configuring container entrypoints is trivial and obviates the need for the LLM.
2) The problem of excess permissions is well-known and understood by Kubernetes developers.
3) The results of the paper are not vulnerabilities per se, but rather best practices failures that make exploiting other vulnerabilities easier.
4) The results of the study are based on a small sample size of highly-correlated kubernetes applications. As such, measurement findings may not generalize over developers as a whole.
5) `EPScan`'s LLM performance is evaluated against a strawman static analysis technique (matching paths in the container). Reviewers noted that semi-dynamic analysis and broader executable matching may outperform this baseline.

Appendix C. Response to the Meta-Review

We sincerely thank all the reviewers for providing constructive comments and suggestions. We would like to respond to the following noteworthy concerns.

For Noteworthy Concern #1, the LLM approach is important for applying `EPScan` in certain scenarios, especially for cluster maintenance personnel (or third-party security analyzers). These personnel may manage clusters with many apps installed, which may be affected by EP issues. Since these personnel usually lack professional knowledge of the apps (i.e., what main entry will pods run), the LLM approach can effectively help them scan for EP issues in these apps, allowing them to directly modify configurations for mitigating EP issues.

For Noteworthy Concern #3, we believe over-permission is widely recognized as a vulnerability. Several CWEs are related to EP issues (e.g., CWE-282, CWE-732), with hundreds of associated CVEs. In Kubernetes scenarios, Yang *et al.* [17] were the first to identify this attack surface, leading to multiple CVEs and vendor acknowledgments. Subsequently, other security teams have discovered similar CVEs, such as CVE-2024-33396. It is worth mentioning that we have developed exploit scripts for all 9 CVEs `EPScan` discovered. These scripts can perform attacks in clusters with only the vulnerable app installed, without needing to exploit any other vulnerabilities.

For Noteworthy Concern #5, our baseline method to evaluate the performance of LLM is designed based on the experience of manual analysis. In §6.4, we discussed the obstacles of using a semi-automatic approach or other strict executable matching approach in our scenario. These obstacles ultimately led us to design the baseline based on heuristic rules.