Integrity Protection for Kubernetes Resource Based on Digital Signature

1st Ruriko Kudo
IBM Research - Tokyo
IBM Japan
Tokyo, Japan
rurikudo@ibm.com

2nd Hirokuni Kitahara
IBM Research - Tokyo
IBM Japan
Tokyo, Japan
hirokuni.kitahara1@ibm.com

3rd Kugamoorthy Gajananan
IBM Research - Tokyo
IBM Japan
Tokyo, Japan
gajan@jp.ibm.com

4th Yuji Watanabe
IBM Research - Tokyo
IBM Japan
Tokyo, Japan
muew@jp.ibm.com

2021 IEEE 14th International Conference on Cloud Computing (CLOUD) | 978-1-6654-0060-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/CLOUD53861.2021.00042

***Abstract*—Integrity of the cloud is the most important requirement for mission-critical enterprise workloads. NIST SP 800-53 states that information systems must prevent the installation of any components that have not been verified digitally with a signed certificate that is recognized and approved by the organization's information system. On a Kubernetes cluster, the admission controller can control requests for application installation, and it would be a powerful protection tool if it could control requests for Kubernetes resources based on signature verification. However, there are various technical challenges when it comes to verifying the signature for a Kubernetes resource at the admission controller because a signed resource is rewritten automatically by internal cluster work and many requests that include internal mutation without a signature are generated. In this work, we propose an approach to protect the integrity of a Kubernetes resource with signature verification at the admission controller. Our approach addresses the issue that the differences between the signed resource in the admission request and the signature message occur automatically in Kubernetes and conducts signature verification properly by using DryRun. We also propose a profile framework to address the internal mutation request that cannot be attached to the signature. Our experimental results demonstrate that standard applications can be protected by our approach.**

***Index Terms*—application integrity, Kubernetes, signature**

I. Introduction

A. Background

Security, compliance, and integrity of the cloud are the most important requirements for mission-critical enterprise workloads. For cloud computing that requires a high level of protection, such as for governments and financial institutions, various standards have been established to strengthen the security. For example, NIST SP 800-53 [17], which is a security management standard required for cloud services introduced by the government, states that the information system must prevent the installation of any components that have not been verified digitally with a signed certificate that is recognized and approved by the organization's information system.

Kubernetes [11] is a container orchestration platform containing a mechanism called an admission controller (A.C.) [12] that controls the requests for operations on a Kubernetes resource. If we could prevent the installation of unauthorized requests by conducting signature verification and enforcement at the admission controller, it would be a strong integrity protection tool. On a Kubernetes cluster, an application consists of container images and Kubernetes resources that define the configuration of the application. In order to protect the integrity of the entire application, both the container image and the Kubernetes resource should be validated with signatures. While a signature verification at the admission controller for container images has already been developed [15], there is currently no mechanism to control the request for the Kubernetes resource based on the signature verification.

There are various challenges when it comes to protecting a Kubernetes resource with a signature at the admission controller. First, it is not easy to verify the signature correctly at the admission controller because the Kubernetes resource is rewritten internally before the admission controller receives the signed Kubernetes resource. The Kubernetes resource is represented by YAML [20], and the entire content of YAML is used as a message to create a signature for protecting the Kubernetes resource. The signed Kubernetes resource is passed on to the admission controller as an admission request when the Kubernetes resource is installed in a cluster. However, the resource in the admission request is different from the original resource at the time of signature generation. In order to verify the signature correctly based on the original resource, we need to handle the difference between the original resource and the resource in the admission request that occurs due to the internal Kubernetes operation.

Second, it is difficult to handle all admission requests properly with a signature because the requests originate from not only outside but also inside a cluster. On a Kubernetes, various internal processes are performed as part of the normal operation of the cluster, and related admission requests are generated. These requests that the cluster generates as an internal operation are an expected mutation and cannot be attached to a signature. If all unsigned requests from the cluster are blocked by the admission controller because of no signature, it may affect the cluster. It is necessary to filter out admission requests caused by internal cluster operation from the large number of requests in order to validate and control requests without any negative impacton the cluster.

2159-6190/21/$31.00 ©2021 IEEE
DOI 10.1109/CLOUD53861.2021.00042

B. Our contribution

In this paper, we propose a method to protect the integrity of application configuration by using signature at an admission controller. The contributions of this work are as follows.

1) The development of an admission controller that verifies the signature of a Kubernetes resource in an admission request based on the object at the time of signature generation.
2) The development of a flexible profile to efficiently filter requests and changes to the resource based on the internal cluster behavior from the signature verification and to control the admission request based on the profile.

In the proposed method, we utilize the DryRun [18] function of Kubernetes to calculate the consistency between the resource in the admission request and the resource at the time of signature creation so that the signature can be verified correctly based on the original resource. In addition, for filtering mutations generated internally from the cluster, we created a profile framework that can flexibly define admission requests from multiple perspectives. This filtering is essential for the admission controller to achieve signature verification of Kubernetes resources. The profile enables the admission controller to determine whether a received request has been internally generated or not and whether changes to a resource have occurred internally.

To determine the effectiveness of the proposed method, we conducted an experiment in which we installed actual applications in a Kubernetes cluster and evaluated how well our method could handle the requests that are generated during the application installation. The results showed that our method is able to control the requests appropriately based on the profiles and signature verification: in total, 98.3% of the requests generated by the application installations were handled correctly. Since we experimented with multiple applications with general Kubernetes application installers (e.g., Helm Charts [2], Operator [4] with Operator Lifecycle Manager [19], and Operator (manual installation)), we expect the above results to be similar for almost all applications. Furthermore, for the resources that could not be protected properly, we found we could enable protection after re-configuring the profiles. The additional approach will also be described.

In Section II of this paper, we describe relevant technologies and define the problem. In Section III, we present our proposed approach. Section IV evaluates our prototype system. Related work is introduced in Section V. We conclude in Section VI with a summary of our findings.

II. Background and Challenge

We briefly describe the technologies used in this work and define some relevant terminologies.

A. Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services that facilitates both declarative configuration and automation. All applications running on Kubernetes are composed of Kubernetes resources. If you want to create a Kubernetes resource, you need to adequately describe the object's specs in terms of the desired state, along with basic information about the object (e.g., name). For example, Deployment is a Kubernetes resource that represents an application running on a cluster. We can represent Deployment as shown in Fig. 1 if we want to install an nginx application on a cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
    labels:
    app: nginx
spec:
    replicas: 3
    selector:
    matchLabels:
        app: nginx
    template:
    metadata:
        labels:
        app: nginx
    spec:
        containers:
        - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Fig. 1. Example of Kubernetes resource.

The core of Kubernetes' control plane is the API server. We need to use a Kubernetes API to operate (create, update, delete, etc.) a Kubernetes resource. Most operations can be performed through the kubectl command-line interface. When adding a Kubernetes resource to a cluster, the definition written in a YAML file is applied with kubectl command-line.

On the cluster side, when the Kubernetes API server receives a request, it authenticates and authorizes it. Kubernetes can be configured flexibly. Admission controller, which controls the request control of the Kubernetes API server, is one of these flexible configurations. The admission controller is a piece of code that intercepts requests to the Kubernetes API server prior to the creation of the object, but after the request is authenticated and authorized. An admission request, which is an object received by the admission controller, includes various information such as Kubernetes resource object and username (i.e., who created the request). This information can be used to control various aspects of the request. Thus, by extending the API functions, it is possible to hook code to modify the request (MutatingAdmissionWebhook) or to decide whether or not to allow it to be executed (ValidatingAdmissionWebhook).

B. Admission controller

As described in chapter II-A, Kubernetes is highly configurable and extensible. There are many enhancements that can be implemented by using admission controllers.

Sysdig Secure [13] proposed an admission controller that is integrated with an image scanner. Sysdig's Admission Controller builds upon Kubernetes and enhances the capacity of the

image scanner to check images for Common Vulnerabilities and Exposures (CVEs), misconfigurations, outdated images, etc., elevating the scan policies from detection to actual prevention. Container images that do not fulfill the configured admission policies will be rejected from the cluster before being assigned to a node and allowed to run.

Open Policy Agent (OPA) [14] and OPA Gatekeeper [16] are other extensions that use an admission controller. With OPA, you can enforce custom policies on Kubernetes objects without recompiling or reconfiguring the Kubernetes API server or even the Kubernetes admission controllers. By installing OPA Gatekeeper as an admission controller, you can check a policy that can describe rate limits, names of trusted servers, the clusters an application should be deployed to, permitted network routes, or accounts a user can withdraw money from.

Portieris [15] is a Kubernetes admission controller for the enforcement of image security policies. You can create image security policies for each Kubernetes namespace or at the cluster level, and enforce different rules for different images.

C. Survey of Kubernetes admission request

1) Difference between the resource at signature creation and at admission request: In this section, we explain the difference between the message when a Kubernetes resource is signed and the message in the admission request that the admission controller actually receives. Figure 2 shows the flow of creating a Kubernetes resource. When a user applies a Kubernetes resource to a cluster, the admission controller receives an admission request. Figure 3 shows an example of Deployment before applying the cluster. When the admission controller verifies a signature on the Kubernetes resource, the application owner creates a signature with the message of the YAML. However, the request received by the admission controller is rewritten to be different from the original resource that has been signed, as new fields and values are inserted by the internal operation of Kubernetes. In fact, when the Deployment in Fig. 3 is deployed in a Kubernetes cluster, the message in the admission request is rewritten as shown in Fig. 4. In addition, the fields and values added by the Kubernetes internal behavior differ depending on the resource.

Thus, in order to achieve signature verification of Kubernetes resources at the admission controller, we should not verify the signature of the resource included in the admission request directly. Instead, we must take into account the parts that have been changed by the internal operation of the cluster.

2) Distribution of admission requests: We investigated what kind of admission requests occur on the Kubernetes cluster in order to enable the signature verification of Kubernetes resources at the admission controller. The surveyed data are all the requests that occurred on the OpenShift Container Platform [10] (a Kubernetes enterprise platform) during the period of October 30, 2020 to November 4, 2020. Since this cluster is close to the initialized state, the requests that occurred during this period are the requests generated internally by the platform to run the Kubernetes cluster.

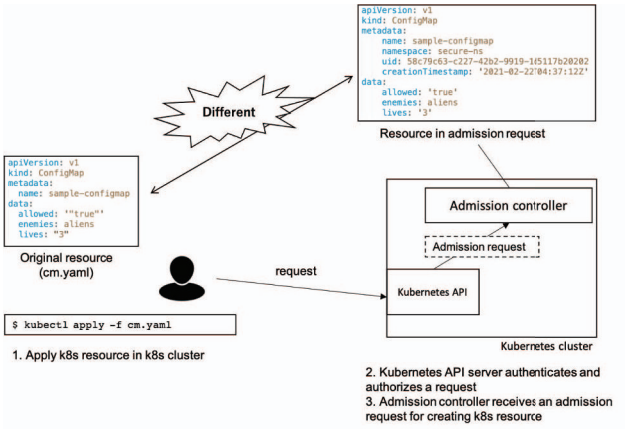

Fig. 2. Process flow for k8s resource creation.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-app
  labels:
    app: test-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-app
  template:
    metadata:
      labels:
        app: test-app
    spec:
      imagePullSecrets:
        - name: regkey
      containers:
        - name: test-app
          image: test-image:0.0.1
          resources:
            requests:
              cpu: 100m
          command:
            - sleep
            - infinity
```

Fig. 3. Original Deployment.

Figure 5 shows the numbers of created, updated, and deleted requests every three hours. As we can see, more than 160,000 requests were internally generated every three hours even in the steady state when there was no external operation on the Kubernetes cluster. In order to achieve signature verification of Kubernetes resources on the admission controller, it is essential to handle this large number of requests appropriately.

Table I shows the results of investigating the requests that occurred in which namespace, for which kind, and by which service account. Examples with the high number of requests are shown. We can see that requests that occur in the steady state come from various resources connected to various service accounts, and that they occur in various namespaces. It is therefore necessary to judge which requests occurring during the steady state are not mutations so that they are not blocked when protecting the namespace in which the application is installed.

TABLE I
EXAMPLE OF ADMISSION REQUESTS IN STEADY STATE

Namespace	*Kind*	*UserName*	*Total*
openshift-kube-scheduler	ConfigMap	kube-scheduler	196,853
openshift-cloud-credential-operator	ConfigMap	openshift-cloud-credential-operator:default	196,810
kube-system	ConfigMap	kube-controller-manager	131,481
hive	ConfigMap	hive:hive-controllers	98,686
openshift-cluster-storage-operator	Lease	openshift-cluster-storage-operator:csi-snapshot-controller	79,029
openshift-monitoring	ConfigMap	openshift-monitoring:cluster-monitoring-operator	62,825
policies	PlacementBinding	rh-acm:multicluster-operators	43,464
policies	PlacementRule	rh-acm:multicluster-operators	43,464
policies	Policy	rh-acm:multicluster-operators	43,464
kube-node-lease	Lease	node:ip-xx-x-xxx-xx.xxxxx.compute.internal	39,570

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: test-app
  namespace: secure-ns
  uid: 9ceccc8c-4765-4266-b057-774866417734
  generation: 1
  creationTimestamp: '2021-02-22T04:33:36Z'
  labels:
      app: test-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-app
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: test-app
    spec:
      containers:
      - name: test-app
        image: 'test-image:0.0.1'
        command:
          - sleep
          - infinity
        resources:
          requests:
            cpu: 100m
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        imagePullPolicy: IfNotPresent
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      securityContext: {}
      imagePullSecrets:
      - name: regkey
      schedulerName: default-scheduler
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
status: {}
```

Fig. 4. Deployment in admission request.

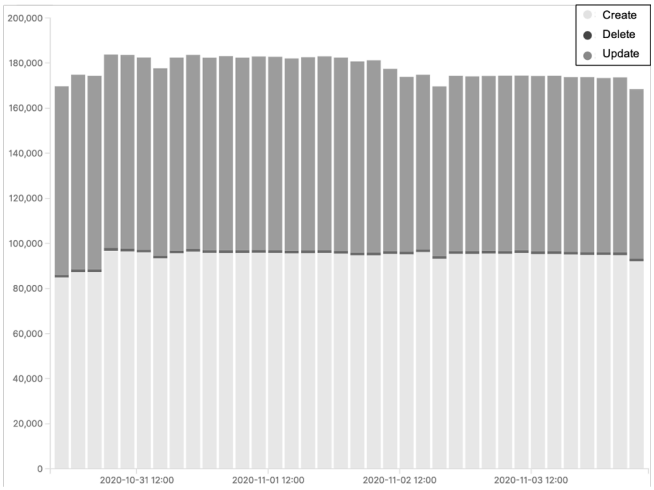

Fig. 5. Admission requests in steady state.

III. PROPOSED APPROACH

A. Overview

In this chapter, we present an overview of the proposed approach. As discussed in chapter II-A, Kubernetes has an extension called "Admission Controller" that allows you to add arbitrary controls to requests when creating or modifying Kubernetes resources. In this work, we utilize this feature and propose a signature verification method for Kubernetes resources as follows.

Figure 6 shows the overview of the proposed approach. First, an application owner generates a signature for the application's Kubernetes resource written in the original YAML file (A) and attaches the signature (B) and the message (C) to the file. The message is the entire original YAML. When the signed YAML file (D) is deployed in a cluster, the Kubernetes API creates an admission request. The admission controller receives this admission request, which includes the Kubernetes resource (E), and the verification server (which runs with the admission controller) performs filtering and signature verification based on the profile, and finally determines whether to allow the request.

The details of the signature verification and the profile are described in chapters III-B and III-C, respectively.

B. Signature verification at Admission controller

This section describes the signature verification at the Admission controller.

First, as shown in Fig. 6, a signature (B) for the original YAML file (A) is created and then attached to the file. A message (C), which is an encoded version of the entire resource, is also attached to the resource. The signature is generated by the commonly used GNU Privacy Guard (GPG)

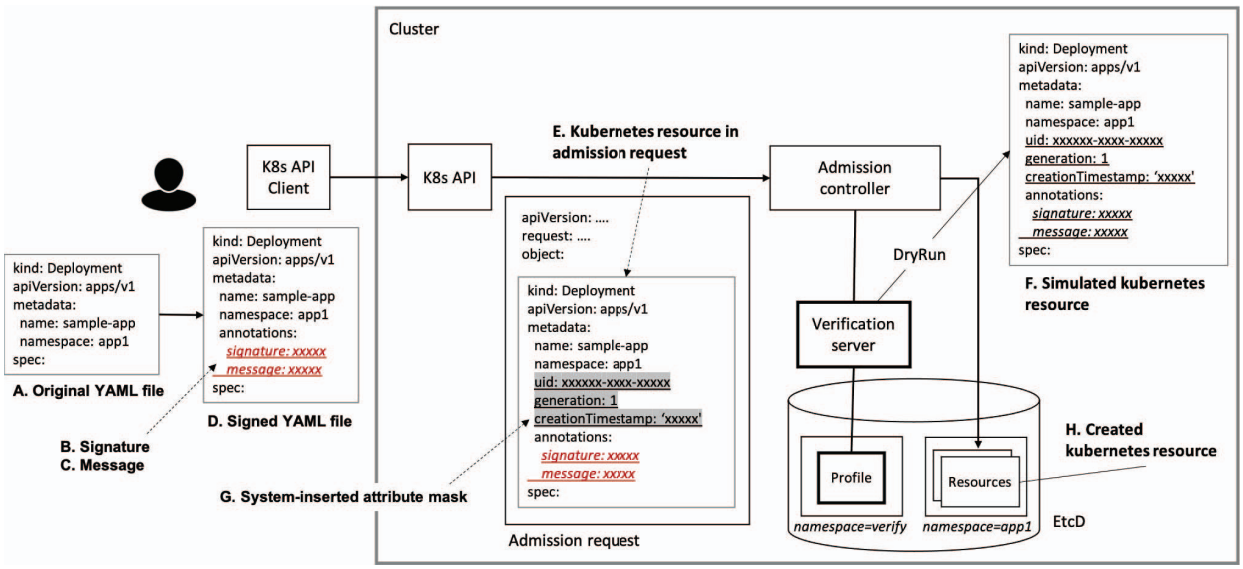

Fig. 6. Overview of proposed approach

or Rivest-Shamir-Adleman (RSA). When this signed resource (D) is applied to the cluster, the admission controller receives an admission request that contains the resource (E). This object has been rewritten by the Kubernetes cluster, as explained in section II-C1.

In order to control the request at the admission controller based on signature verification, the verification server performs the following operations.

1) Get the signature (B) and message (C) from the Kubernetes resource in the admission request (E).
2) Verify that the signature is correct with signature (B) and message (C).
   - If the signature verification fails, the request is not allowed.
   - If the signature verification succeeds, move on to step 3.
3) Perform a DryRun using the decoded message (C) and receive a simulated Kubernetes resource (F) from the API server (Fig. 7).
4) Compare the original message (C) with the simulated Kubernetes resource (F), and calculate the fields added by the cluster (Fig. 8).
5) Check if the Kubernetes resource (E) masked by the system-inserted attribute (G) matches with the simulated Kubernetes resource (F) masked by the system-inserted attribute (G).
   - If these resources match, the Kubernetes resource is consistent with the message (C), and the request is allowed.
   - If these resources do not match, the object has been tampered with from the message (C), and the request is not allowed.

At step 3, we create a request on DryRun mode to obtain simulated Kubernetes resources. DryRun is one option of Kubernetes. When a request is created on DryRun mode, it is processed as a typical request without being persisted to storage. It therefore responds as a body for the request that is as close as possible to a non-dry-run response. As shown in Fig. 7, the proposed method obtains the simulated Kubernetes resource from the original YAML file by using DryRun to compute the system-inserted fields.

Figure 9 shows the process flow of the Kubernetes resource in the proposed method. The original YAML file is signed by application owner and then deployed in a Kubernetes cluster and passed on to the admission controller as an admission request. If the request to create the Kubernetes resource is validated by the admission controller, the resource is created.

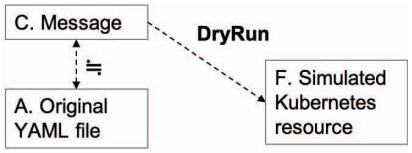

Fig. 7. DryRun

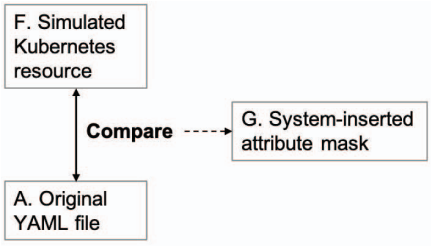

Fig. 8. Acquire system-inserted attribute mask

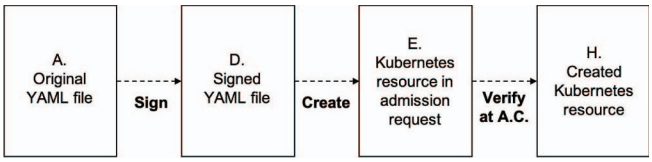

Fig. 9. Request process flow

C. Profile for request control

In proposed method, the request is filtered by a profile before proceeding to the signature verification.

An admission request contains not only the Kubernetes resource (=object) but also various information such as who made the request (=userName), where the request will be installed (=Namespace), and the Kubernetes resource before the change (=oldObject). The proposed profile is configured to allow flexible filtering of requests by combining this information. You can create "application profile" for each Kubernetes namespace for each application. The following rules can be defined in a profile.

1) Protect Rules: The protect rules are used to define the resources we would like to protect with signatures. Requests that match the rules defined here will be verified with a signature by the verification server. The following is an example of a protect rule that protects a secret named " creds-secret " with a signature. As shown in this example, we can define the resources to be protected on the basis of conditions such as kind and name.

```
protectRules:
- match:
    - kind: Secret
    name: creds-secret
```

Listing 1. Example of Protect Rules

2) Ignore Rules: In ignore rules, the requests that are generated by the internal operation of the cluster are defined to skip signature verification. It is also possible to define resources to be explicitly excluded from protection by signature. Requests that match the rules defined here are not identified as mutations, even if they are not signed. Here is an example of an ignore rule.

In this example, it is defined that replicaset-controller, which is a ServiceAccount of the Kubernetes cluster, is allowed to operate the pod. In Kubernetes, a pod is automatically created when a replicaset is created by replicaset-controller. These requests occur internally and are not mutations. Therefore, it is necessary to allow the operation on the pod from this ServiceAccount. There are many operations that occur internally, similar to the above example. The ignore rules to allow these operations are thus common on Kubernetes. In the proposed method, the "common profile" is available. Other rules are also defined in the "common profile" to determine whether the behavior is internal or not.

```
ignoreRules:
- match:
    - kind: Pod
    username:  kube-system:replicaset-controller
```

Listing 2. Example of Ignore Rules in common profile

3) Ignore Attrs: In ignore attrs, fields that are internally rewritten by the cluster are listed. We can also define fields that are allowed to be modified, such as application settings for scalability. The fields specified here are allowed even if the values are changed during signature verification.

In the following example, the parameters are excluded from the signature verification because these fields are added automatically to the resource when the application is installed using Helm Chart.

```
ignoreAttrs:
- attrs:
    - metadata.annotations.meta.helm.sh/release-
        namespace
    - metadata.labels.app.Kubernetes.io/managed-by
    - metadata.annotations.meta.helm.sh/release-name
    match:
    - kind: "*"
```

Listing 3. Example of Ignore Attrs in common profile

IV. Evaluation

We conducted an experiment in which we installed a signed application in a Kubernetes cluster and evaluated whether requests generated during the application installation were processed properly by our proposed approach with signature verification and profiles.

The requests generated during the signed application installation are categorized as follows.

1) Request that is outside the scope of our proposed method (e.g., Event, Lease)
2) Request that matches the condition in the common profile
3) Request that matches the condition in the application profile
4) Request subject to signature verification and that has a valid signature
5) Request blocked because signature verification failed
6) Request that is neither out of scope nor matches the condition in the profiles

The proposed method deals with requests 2 to 6 above. However, in the case of requests in category 5, the signature verification might fail even if the signature is correctly signed because the request has a difference between the original resource and the resource in the admission request. In the case of requests in category 6, the request is not signed and the conditions defined in the profile are not met. This means these requests are not protected by the proposed method. Ideally, requests in categories 5 or 6 will not occur. In this experiment, we evaluated how accurately the proposed method could process the requests by the profile and signature verification.

A. Experimental Environment

The experiment was conducted on a kind cluster [1], which is a local Kubernetes cluster. The Kubernetes version was

v1.19.7. We deployed the following applications with application installers for Kubernetes. Helm Charts [2] is a package manager for Kubernetes applications and Artifact Hub [3] is a web-based application that enables finding, installing, and publishing packages. A Kubernetes operator [4] is a method of packaging, deploying, and managing a Kubernetes application, and OperatorHub.io [5] is a new home for the Kubernetes community to share operators. In this experiment, applications were randomly selected from Artifact Hub and OperatorHub.io. In addition to these two application sources, we used an open-source applications published on GitHub for manual operator installation.

B. Experimental Procedure

We configured the profile for each application and installed the applications with the following steps.

1) Configure an application profile to define which resource should be signed.
   - Helm Charts:
   a) Get resources from the package with Helm template command.
   b) Define the acquired resources in a protect rule in the application profile as the signature verification target.
   c) If a ServiceAccount is included in the package, define the ServiceAccount in an ignored rule to allow requests from it.
   - Operator:
   a) Get resources from operatorHub.io.
   b) Define the acquired resources in a protect rule in the application profile as the signature verification target.
   c) Define the ServiceAccount of the operator in an ignored rule to allow the request from it.
   - Manual operator installation:
   a) Get resources from GitHub.
   b) Define the acquired resources in a protect rule in the application profile as the signature verification target.
   c) Define the ServiceAccount of the operator in an ignored rule to allow the request from it.
2) Generate signatures.
3) Install the application profile in a namespace.
4) Install the signed application in the namespace by application installer.

C. Results

The classification results of requests that occurred during the installation are shown in Table II.

A total of 2298 requests were generated by these application installations. Signature verification failed with two resources, and 39 requests were not processed by the proposed method because they were neither out of scope nor matched with a condition in the profile. Therefore, 98.3% were handled correctly.

Looking at the requests in category 5, most of them could be installed without failing to verify the signature by using the profile of the proposed method. In the case of the NGINX Ingress Controller [6] installed by Helm chart, signature verification for the service resource failed because the healthCheckNodePort field in the admission request had a different value in the original resource. The difference was caused because the value in the healthCheckNodePort field was embedded by the Kubernetes specification when doing a DryRun. In order to install the service resource properly, the application profile should be re-configured as follows:

```
ignoreAttrs:
- match:
  - kind: Service
    name: my-release-nginx-ingress
  attrs:
  - spec.healthCheckNodePort
```

Listing 4. Ignore Attrs for NGINX Ingress Controller

Similarly for other applications, the field "-Port" in a service resource is changed automatically by Kubernetes. Therefore, we can add a rule that ignores the port field to the common profile for all service resources as follows:

```
ignoreAttrs:
- match:
  - kind: Service
  attrs:
  - spec.*Port
```

Listing 5. Ignore Attrs in common profile

In the Nutanix CSI Volume Driver [7], signature verification of StorageClass resource failed because a field "volumeBindingMode" is automatically inserted with the default value "Immediate" into the object in admission request. We can resolve the issue by updating the profile as below as is the case in NGINX Ingress Controller.

```
ignoreAttrs:
- match:
  - kind: StorageClass
    name: nutanix-volume
  attrs:
  - volumeBindingMode
```

Listing 6. Ignore Attrs for Nutanix CSI Volume Driver

Looking at the requests in category 6, most of them were classified correctly by using the profile of the proposed method. In Istio [8], Nginx [9] and Tektoncd [36], a few ServiceAccounts (e.g., "system:serviceaccount:my-nginx-ingress:my-nginx-ingress-controller") were generated by the operator. In addition to the operator's ServiceAccount, these ServiceAccounts that are created by the operator, also generated requests. Since these ServiceAccount are not listed in the profile for the application, these requests generated by the ServiceAccounts are classified as 6. However, the fact that the ServiceAccount created by the operator operates the application resource is not a mutation. Therefore, we can modify the filtering of the proposed method to the appropriate result by adding a rule to allow requests generated by the ServiceAccount to the application profile. The following example is re-configured Ignore Rules for Nginx.

TABLE II
CLASSIFICATION RESULTS OF REQUESTS

application	*installer type*	*1)out of scope*	*2)common profile*	*3)app profile*	*4)valid signature*	*5)invalid signature*	*6)unprocessed*
Kubeview	Helm Chart	31.3%	53.1%	0.0%	15.6%	0.0%	0.0%
NGINX Ingress Controller	Helm Chart	32.5%	35.0%	10.0%	20.0%	2.5%	0.0%
Jenkins	Helm Chart	44.6%	33.9%	0.0%	21.4%	0.0%	0.0%
Nutanix CSI Volume Driver	Helm Chart	51.2%	25.6%	0.0%	22.0%	1.2%	0.0%
Yourls	Helm Chart	45.1%	40.8%	0.0%	14.1%	0.0%	0.0%
Minio	Operator	17.2%	73.9%	6.7%	2.2%	0.0%	0.0%
Jenkins	Operator	14.0%	36.0%	48.3%	1.7%	0.0%	0.0%
Argocd	Operator	23.1%	64.0%	12.0%	0.9%	0.0%	0.0%
Istio	Operator	39.3%	27.5%	28.1%	0.4%	0.0%	4.7%
Grafana	Operator	25.9%	33.0%	39.8%	1.4%	0.0%	0.0%
Nginx Ingress	Operator Manual	25.0%	16.7%	37.5%	9.7%	0.0%	11.1%
Tektoncd	Operator Manual	52.6%	7.7%	36.3%	2.0%	0.0%	1.4%

```
ignoreRules:
- match:
  - username: nginx-ingress-operator
  - username: my-nginx-ingress-controller
```

Listing 7. Ignore Rules for Nginx

Also, Istio creates Istio's resources (secrets with names starting with "istio.") in various Namespaces because istio is an open-source service mesh platform. Therefore, if all the secrets on the cluster are set to be verified with a signature, the resources created by istio will be blocked, and the functionality of istio would be affected. In order to protect an application that is close to the function of the infrastructure of a cluster that creates resources across multiple Namespaces (such as Istio) without affecting its function, we can add a rule to the common profile. The following example is for Istio.

```
ignoreRules:
- match:
  - username: istio-citadel-service-account
    name: istio.*
    kind: Secret
```

Listing 8. Ignore Rules for Istio

It turned out that the admission controller can classify the received requests appropriately by using the proposed profile. In addition, the evaluation results show that it is possible to define flexible profiles to efficiently exclude the changes based on the internal cluster behavior from the signature verification, and that the signature verification of Kubernetes resources at the admission controller works as planned.

V. RELATED WORK

There have many research studies that focused on integrity of cloud. In the file integrity protection, monitoring system changes is a major approach [27] [34] [28]. This is a method of monitoring changes and detecting whether the changes are anomalous or not. Hai Jin et al. [27] and Dragi et al. [34] proposed an approach to detect mutation based on rules and H.Kitahara et al. [28] proposes a mechanism filter out default behavioral events in containers and detect mutations events that are anomalous. There existed commercial products for file integrity protection. For instance, Solarwinds ' s Security Event Manager [21], Qualys ' s File Integrity Monitoring [23], Trustwave ' s Endpoint Protection [22], and Tripwire [24].

In this paper, we propose a method to prevent unauthenticated changes to Kubernetes resources with signature, which is different from the conventional detection methods.

Integrity protection with a signature has been studied. For container images, which are one of the components of applications on Kubernetes, there are some technology to attach signatures to a container image. Docker Content Trust(DCT) [25] offers a static and basic approach to integrity verification, as it can check the integrity of the image at load-time only. Simple Signing [26] is the other technology for signing container image developed by RedHat. For the image signature verification, Portieris [15] provides request control based on the signature verification at Kubernetes admission controller. In the OpenShift Container Platform 4, Machine Config [33] verify signature for container image which pulled to a cloud platform. De Benedictis et al. [30] presents a third party solution for security monitoring of a lightweight cloud infrastructure, which exploits Remote Attestation to verify image integrity.

The proposed method provides the signature of Kubernetes resources installed with YAML and the request control at an admission controller. Since a container image is not changed once it is built, the signature verification does not need to consider the internal changes by the cluster. To realize signature verification for Kubernetes resources, there are issues that did not occur with container images. In this work, we addressed the problem that the Kubernetes resource is rewritten internally by Kubernetes before the admission controller receives the signed Kubernetes resource.

There are also many technologies for integrity protection with signature for other resources. Ernesto et al. [29] presented an access control model to protect information distributed on the Web that, by exploiting XML's own capabilities, allows the definition and enforcement of access restrictions directly on the structure and content of the documents. Gugnani G. et al. [32] provides confidentiality to the user while using cloud-based web services. H. Zhu et al. [36] proposed a scheme of data integrity verification based on a short signature algorithm (ZSS signature), which supports privacy protection and public auditing by introducing a trusted third party (TPA) in order

to ensure data integrity and availability in the cloud and IoT storage system They proposed an approach for selective encryption of XML elements so as to provide confidentiality and prevent XML document form improper information disclosure.

In this paper, signature verification is performed at the admission controller, and we address a challenge that admission controller receive a large number of requests including targeted or untargeted requests for verification.

VI. Summary

In this work, we proposed an approach to protect Kubernetes resource integrity based on signature verification at the admission controller. We achieved an admission controller that verifies the signature of a Kubernetes resource in the admission request based on the original Kubernetes resource before it was internally modified. We also proposed a flexible profile to efficiently filter mutations from the internal cluster behavior and control the admission request based on the profile. We conducted an experiment in which we installed actual applications in a Kubernetes cluster to determine the effectiveness of the proposed method.

The results demonstrated that 98.1% of the requests that were generated during the application installation were handled properly by the proposed method. Since we experimented with multiple applications with general Kubernetes application installers such as Helm Charts and Operator, we expect the results to be similar for almost all applications. Furthermore, for the resources that could not be properly protected, we found we could protect them by re-configuring the profiles. It turned out that the proposed profile can be configured flexibly and that it is effective to correctly protect application integrity.

References

[1] "Kind", Available: https://kind.sigs.k8s.io
[2] "Helm Charts", Available: https://helm.sh
[3] "Find, install and publish Kubernetes packages", Available: https://artifacthub.io
[4] "What is a Kubernetes operator?", Available: https://www.redhat.com/en/topics/containers/what-is-a-Kubernetes-operator
[5] "Welcome to OperatorHub.io", Available: https://operatorhub.io
[6] "NGINX Ingress Controller", Available: https://artifacthub.io/packages/helm/nginx/nginx-ingress
[7] "Nutanix CSI Volume Driver", Available: https://artifacthub.io/packages/helm/nutanix/nutanix-csi-storage
[8] "Istio", Available: https://operatorhub.io/operator/istio
[9] "NGINX Ingress Operator", Available: https://github.com/nginxinc/nginx-ingress-operator
[10] "Red Hat OpenShift Container Platform: Kubernetes for rapid innovation", Available: https://www.redhat.com/en/resources/openshift-container-platform-datasheet
[11] "What is Kubernetes?", Available: https://Kubernetes.io/docs/concepts/overview/what-is-Kubernetes/
[12] "Using Admission Controllers", Available: https://Kubernetes.io/docs/reference/access-authn-authz/admission-controllers/
[13] "Kubernetes admission controllers in 5 minutes", Kaizhe Huang, February 18, 2021, Available: https://sysdig.com/blog/Kubernetes-admission-controllers/
[14] "Policy-based control for cloud native environments", Available: https://www.openpolicyagent.org
[15] "Portieris", Available: https://github.com/IBM/portieris
[16] "Gatekeeper", Available: https://github.com/open-policy-agent/gatekeeper
[17] "NIST Risk Management Framework", Available: https://csrc.nist.gov/Projects/risk-management/sp800-53-controls/release-search#!/800-53
[18] "Dry-run", Available: https://Kubernetes.io/docs/reference/using-api/api-concepts/#dry-run
[19] "OPERATOR LIFECYCLE MANAGER", Available: https://olm.operatorframework.io
[20] "YAML: YAML Ain't Markup Language", Available: https://yaml.org/
[21] Solarwinds. Security event manager, 2020.
[22] Trustwave. Trustwave endpoint protection, 2020.
[23] Qualys. File integrity monitoring, 2020.
[24] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: A file system integrity checker. In Proceedings of the 2nd ACM Conference on Computer and Communications Security, CCS '94, page 18–29, New York, NY, USA, 1994. Association for Computing Machinery.
[25] "Docker Content Trust", Available: https://docs.docker.com/engine/security/trust/
[26] "Simple Signing", Available: https://www.redhat.com/en/blog/container-image-signing
[27] Hai Jin, Guofu Xiang, Deqing Zou, Feng Zhao, Min Li, and Chen Yu. A guest-transparent file integrity monitoring method in virtualization environment. Computers & Mathematics with Applications, 60(2):256 – 266, 2010. Advances in Cryptography, Security and Applications for Future Computer Science.
[28] H.Kitahara, K.Gajananan, and Y.Watanabe. Highly-scalable container integrity monitoring for large-scale Kubernetes cluster. In 2020 IEEE International Conference on Big Data (Big Data 2020), 2020.
[29] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. 2002. A fine-grained access control system for XML documents. ACM Trans. Inf. Syst. Secur. 5, 2 (May 2002), 169–202. DOI:https://doi.org/10.1145/505586.505590
[30] De Benedictis, Marco & Lioy, Antonio. (2019). Integrity verification of Docker containers for a lightweight cloud environment. Future Generation Computer Systems. 97. 10.1016/j.future.2019.02.026.
[31] H. C. Pöhls, "JSON Sensor Signatures (JSS): End-to-End Integrity Protection from Constrained Device to IoT Application," 2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, 2015, pp. 306-312, doi: 10.1109/IMIS.2015.48.
[32] Gugnani G, Ghrera SP, Gupta PK, Malekian R, Maharaj BT. Implementing DNA Encryption Technique in Web Services to Embed Confidentiality in Cloud. InProceedings of the Second International Conference on Computer and Communication Technologies 2016 (pp. 407-415). Springer India.
[33] "Verifying Red Hat container image signatures in OpenShift Container Platform 4", Available: https://access.redhat.com/verify-images-ocp4
[34] Dragi Zlatkovski, Aleksandra Mileva, Kristina Bogatinova, and Igor Ampov. A new real-time file integrity monitoring system for windows-based environments. ICT Innovations 2018, Web Proceedings ISSN 1857-7288, pages 243–258, 2018.
[35] H. Zhu et al., "A Secure and Efficient Data Integrity Verification Scheme for Cloud-IoT Based on Short Signature," in IEEE Access, vol. 7, pp. 90036-90044, 2019, doi: 10.1109/ACCESS.2019.2924486.
[36] "Tektoncd", Available: https://github.com/tektoncd/operator