Prescriptive Model for Software Supply Chain Assurance in Private Cloud Environments

Robert Wood
Prescriptive Model for Software Supply Chain Assurance in Private Cloud Environments

GIAC (GCCC) Gold Certification

ISSE 5901

Author: Robert Wood, woodrobe@gmail.com
Advisor: David Fletcher

Accepted: September 1, 2020

Abstract

As companies embrace Continuous Integration/Continuous Deployment (CI/CD) environments, automated controls are critical for safeguarding the Software Development Life Cycle (SDLC). The ability to vet and whitelist container images before installation is vitally important to ensuring the security of corporate networks. Google Cloud offers the Container Registry in combination with Binary Authorization to understand the container footprint in the environment and provide a mechanism for enforcing policies. Grafeas and Kritis are open-source alternatives. This paper evaluates Grafeas and Kritis and provides specific recommendations for using these tools or equivalents in private cloud environments.
1. Introduction

Companies recognize that software is a crucial differentiator in our competitive world. As a result, they focus significant time and energy protecting the software development supply chain. Simply put, the software development supply chain is the sequence of actions designed to bring new functionality to the market quickly.

Continuous Integration/Continuous Deployment (CI/CD) models allow companies to build, test quickly, and deploy the vital software products on which their businesses depend. These companies deploy code many times each day, and ensuring the stability of these frequent deployments highlights the importance of thorough testing, trusted third-party code, and automated deployment. Further, securing the software supply chain is a challenging endeavor as companies embrace microservices, open-source tools, and hybrid cloud environments (Elliot, 2017). With a wide array of metadata to track, identifying, and understanding the current environment is no trivial task. Without this metadata, companies struggle to understand what is truly happening in their software supply chain.

Software supply chain management allows companies to protect the critical software assets they create, ensuring that they use only trusted images, authorized components, and safe third-party tools. Companies must know everything that has happened from "the time code is written (until it goes) to production" (Greenberg, 2019). With this knowledge, they can proactively block suspect code and ensure the integrity of their products. Google leads the pack with commercial tools that help manage the software supply chain. Container Registry houses metadata and provides customers insight into what is happening in the CI/CD pipeline. Binary Authorization leverages the data in Container Registry, providing a checkpoint for images and components before allowing them into the Kubernetes cluster.

Open-source tools provide an alternative for companies hosting internal cloud environments, which is essential for companies operating in industries like financial services. These companies may be hesitant to use public cloud resources due to concerns over data confidentiality. Grafeas and Kritis are open-source alternatives to Container Registry and Binary Authorization, respectively. Currently, these active open-source projects offer companies the ability to track metadata about their internally-hosted CI/CD pipelines and prevent the deployment of unauthorized and potentially damaging code within Google Cloud Platform.
(GCP). However, while they are open-source, the usefulness of Kritis and Grafeas, both optimized for GCP, is limited.

A challenge for companies desiring the flexibility to move between public cloud providers and private cloud environments is determining how to implement metadata tracking and image vetting tools in such a way that they achieve portability, scalability, and security. As a result, a prescriptive model for deploying these critical capabilities in private cloud environments is both warranted and needed.

2. Background

2.1 Software Supply Chain and CI/CD

2.1.1 Software Supply Chain Defined

The software supply chain includes several critical gates for managing the deployment of software. Aysylu Greenberg discusses the following steps in her 2019 talk for InfoQ Brasil: write code, code check-in, build image, test & verification, QA, and deploy to production (Greenberg, 2019). Like the food supply chain, problems in any step of the process can be catastrophic, resulting in the release of corrupt or unreliable software.

2.1.2 CI/CD Defined

CI/CD provides a mechanism for controlling the software supply chain. Static and dynamic code scanning, protecting secrets, and vulnerability scanning are some of the critical activities provided as part of the CI/CD pipeline. As a result, company support for CI/CD continues to increase. This support is evident as "conventional software development and delivery methods are rapidly becoming obsolete as deployment frequency increases." (Azeri, 2020). Because CI/CD focuses on the critical processes of software updates, including "build, deploy, test and release processes" (Humble, 2011), companies yield significant power in controlling software updates by using CI/CD processes.

2.2 CIS Controls

The Center for Internet Security (CIS) has long published its top 20 critical controls that organizations can follow to enhance security. Security practitioners subject these controls to
constant vetting. They continually evaluate and adjust these controls to address the changing needs of the business.

### 2.2.1 CIS Control 18, Application Software Security

CIS Control 18 addresses application software security. This control recommends that analysts "manage the security life cycle of all in-house developed and acquired software in order to prevent, detect, and correct security weaknesses" (CIS, 2020). CI/CD plays into this requirement as it gives practitioners a toolset for protecting the software build and release process.

### 3. Container Metadata and Container Policy Enforcement Solutions

#### 3.1 Commercial Options

##### 3.1.1 Container Registry – Google Cloud

Companies must understand metadata about their environment to protect the software supply chain. Details about vulnerability scans, image management, and access controls are critical to securing software assets. Further, this metadata contains the minutia often used by malicious actors to breach corporate protections. Container Registry is Google's solution to these challenges. Built into GCP, Container Registry is now one of Google's core offerings.

##### 3.1.2 Binary Authorization – Google Cloud

Companies need to control the images that make it into the software supply chain. Vulnerable or malicious images can cause disruption, or worse, data breaches. Binary Authorization is the commercial solution provided by Google in GCP. Natively integrated into the platform, Binary Authorization is easy to configure and minimally disruptive to the software supply chain process. The tool aims to be simple to both configure and use.

Binary Authorization supports whitelisting of images, which allows system owners to identify approved images and proactively and tightly control the environment. Further, the module supports break-glass functionality, which allows for emergency deployments when
needed. This capability is necessary in cases where rigid controls prevent the deployment of recovery resources needed to meet critical deadlines or address time-sensitive client demands. Further, cryptographic tracking of images helps to ensure that only approved images get into the environment.

Another critical feature of Binary Authorization is that it integrates with several third-party solutions. Google touts this feature and continues to enhance the offering. Its tight integration with GCP and GCP's Kubernetes components makes it seamless for Google's customers. Further, its inclusion in the GCP graphical interface makes it an easily used feature for GCP users.
3.2 Open-Source Options for Container Metadata and Container Policy Enforcement

3.2.1 Grafeas

Google identifies the following requirements for securing software supply chains. Firstly, infrastructure is immutable to protect against advanced persistent threats. Secondly, controls,
including attestations, integrate with the software supply chain. Lastly, the system works with
developer tools and open-source software (Elliot, 2017).

Grafeas is an open-source tool that captures and stores metadata about artifacts and related
vulnerabilities (Martin, 2018). This metadata allows cluster owners to control the use of those
artifacts within the software supply chain. Grafeas has two primary constructs: notes and
occurrences. Notes define pieces of information identified through analysis, while occurrences
are instances of notes discovered through analysis.

Additionally, projects are namespaces used to house metadata, and attestations are the part of
Grafeas that enable cryptographic evidence collection. The use of cryptography provides
transparency and supports regulated environments. While Grafeas is open-source, it currently
works best in the GCP environment. However, during this research, Grafeas proved easy to
deploy in a private Kubernetes environment.

3.2.2 Kritis

Kritis is the open-source answer to Google's Binary Authorization. It uses Image Security
Policies (ISP) to approve images for use in the software supply chain. Kritis also validates and
mutates webhooks to vet images introduced into GCP Kubernetes clusters. Like Binary
Authorization, Kritis defines Attestation Authorities to provide evidence of an image's validity.
Further, it uses encryption to sign approved images and provide proof that images are safe for
use in the software supply chain.

Like Binary Authorization, Kritis supports whitelisting of images and break-glass
functionality in case of emergency. Notably, Kritis currently works best in GCP environments.
The researcher did not find any working code for quick deployment to non-GCP Kubernetes
clusters. Consequently, cluster owners working in non-GCP environments may find it
challenging to design an appropriate solution that meets audit and security requirements.

3.2.3 Admission Controllers

Because Kritis has such a tie to GCP, this research delves more deeply into admission
controllers, which allow cluster owners to intercept, validate, and accept or reject requests to the
cluster. Admission controllers run in two phases: mutating and validating (Isberner, 2019). They
provide advanced security features such as security baseline enforcement across namespaces.
Most importantly, mutating and validating webhooks offer functionality similar to what is available commercially.

Admission controllers serve several vital functions. By proxy, they greatly enhance security, governance, and configuration management. By controlling image sources, limiting the ability to run as root, and checking images against known vulnerabilities, admission controllers enhance security. By enforcing labels and encouraging the use of annotations, admission controllers promote good governance. Finally, by allowing for the validation of object configuration, enforcing resource limits, and checking for labels like "latest," admission controllers support configuration management.

![Diagram of Admission Control Process](image)

**Figure 2:** Location of mutating and validating webhooks.

Mutating webhooks are just one of thirty admission controllers that ship with Kubernetes. They provide cluster owners great flexibility and allow them to introduce custom logic when creating, updating, or deleting resources (Isberner, 2019). Mutating webhooks are unique because they can change the objects they are vetting, making them sturdy, but also risky. For example, developers may use a mutating webhook to add a sidecar container to a process in certain situations. This flexibility is useful, but the risk introduced is not acceptable to more conservative organizations.

Unlikemutating webhooks, validating webhooks cannot change the object they are checking. Organizations that are audit-focused and prefer to only work with immutable objects may prefer this structure. Further, validating webhooks run after mutating webhooks, and therefore have visibility into the final version of objects passed to etcd.
Because admission controllers are not beholden to GCP like Kritis, they can be used to provide an appropriate level of control for the environment. Notably, companies can make use of mutating and validating webhooks to build in the controls provided by a Kritis implementation using the tools already built into Kubernetes. That said, admission controllers, like Kritis, are still relatively new. Any solution pursued requires some level of creativity and flexibility on the part of the cluster owner.

4. Research Methodology

The goal of this research is to evaluate commercial and open-source metadata and image vetting tools and provide a recommendation for configuring and installing open-source solutions in a private cloud environment. The first part of this research involved a Google Cloud account for configuring both Container Registry and Binary Authorization, and Grafeas and Kritis services in a public cloud setting. With the knowledge gained in the public cloud environment, the researcher configured a standalone Kubernetes cluster to capture procedures and provide recommendations for running metadata tracking and image vetting tools in a private cloud environment.

The researcher vetted several Kubernetes options to use for private deployment. The first environment considered was Minikube, which deploys a standalone, single-node cluster. While an excellent platform for testing, Minikube did not make sense because of its single-node structure. Because most Kubernetes production deployments contain both master and slave nodes, testing against an environment with only a master node seemed incomplete and not representative of a typical real-world deployment.

Next, the researcher considered a three-box, three-node Kubernetes cluster using CentOS7 servers with one master node and two worker nodes. The researcher successfully used this structure during a previous research effort. However, changes in Kubernetes in the last year and the inherent overhead of managing three servers were too much effort given the goals of this research. It was more important that the researcher be able to build and reset the environment easily to test a variety of code with minimal overhead. Appendix VII contains the procedures used to deploy this cluster.
Ultimately, Kind proved the most useful. Kind is a self-contained Kubernetes cluster that allows for quick deployment of multiple-node clusters to a single server. Kind is particularly appealing as it supports multi-node clusters, works on Linux, Mac, and Windows, and has tight integration with Kubernetes. Notably, clusters can be torn down and rebuilt in just a few minutes.

To become familiar with Kind, the researcher set up a Kind cluster and deployed a simple WordPress site. The procedures used to stand up this WordPress instance came from the researcher’s previous work and provided a benchmark by which to determine the relative difficulty of building and destroying clusters with Kind.

For this machine, the researcher used a CentOS7 server with 100 GB of storage, 4 GB of RAM, and two processors. This environment allowed for quick deployment and teardown of Kubernetes clusters for testing of GitHub code. Because the researcher evaluated several sets of code, this environment saved time in managing the cluster and associated servers.

5. Analysis of Findings

5.1 Container Registry – Analysis

Container Registry in GCP seamlessly integrates with the other modules of the Google platform. Setting up a Kubernetes cluster is easy, both from the graphical user interface (GUI) and the command line. Once built, kubectl, a command-line tool built into Kubernetes, works efficiently for managing clusters.

The GCP menus provide insight into useful metadata about images deployed to the cluster as well as details about identity and access management (IAM), volume configuration, and network configuration, as seen in Figure 3. The system is well-designed for use from either the GUI or command line.
Figure 3: GCP menus
5.2 Binary Authorization – Analysis

Binary Authorization is simple to configure in GCP. Further, whitelisting is easily set up using the list of exempt image paths in the GUI. The researcher did several tests in the system and found the interface both simple and effective.

Figure 4: Binary Authorization - exempt image path list
The researcher used a NGINX image for some of this testing. In this initial test, the researcher executed a YAML file to deploy the NGINX image to the Kubernetes cluster with a Binary Authorization policy configured to disallow all images. As expected, GCP rejected the image with a message from the Binary Authorization system. The message stated that the image policy webhook denied the request because it violated the admission rule. It confirms that Binary Authorization checked the policy and enforced that policy as part of the process.

```
    kind: Pod
    metadata:
      name: nginx
    spec:
      containers:
      - name: nginx
        image: "gcr.io/$PROJECT_ID/nginx:latest"
        ports:
          - containerPort: 80
    EOF
```

Figure 5: Binary Authorization – command line denial message

5.3 Grafeas - Analysis

Grafeas proved challenging to deploy. The researcher reviewed several deployment procedures during testing before successfully confirming a working set of procedures. Appendix I shows the detailed instructions needed to install both Grafeas and Kritis in GCP. The researcher used Helm to install Grafeas in GCP.

Helm is a robust package manager designed specifically for Kubernetes. Helm uses the concept of charts to allow users to define, install, and upgrade Kubernetes applications. The researcher examined several Helm charts, trying to find a simple way to spin up Grafeas and Kritis. Another nice feature of Helm is that it can easily rollback deployments as necessary.

Notably, available Helm scripts required modification due to a problematic name flag. While there were multiple Grafeas-related Helm scripts in GitHub, most did not work on the initial attempt. Because Grafeas is a recently released product, no consensus exists on best practices for deployment. The result was much experimentation and eventually a working set of procedures. Once deployed, Kubernetes ran a single Grafeas pod and functioned as expected.
The Grafeas Helm installation scripts were straightforward. With the slight modifications mentioned above, the researcher successfully ran Grafeas in a non-GCP Kubernetes cluster. In both environments, Grafeas ran as a single pod in the Kubernetes cluster.

![Figure 6: Grafeas pod running in GCP Kubernetes Cluster](image)

### 5.4 Kritis – Analysis

Setting up Kritis was a more challenging endeavor. As with Grafeas, there was not a single, working set of instructions for installation. Appendix I contains the detailed installation instructions used to deploy Kritis in GCP. As with Grafeas, the running Kritis deployment consists of one running pod, the validation hook. However, Kritis requires pre-install and post-install pods, which causes the installation process to be longer and more complex.
Using Kritis, the cluster owner establishes an attestation authority. Using this authority, the owner creates attestation policies and specifies the trusted authority for signature verification. This is important, as a trusted authority must sign images before they can gain admittance into the cluster.

Once running, Kritis functioned as expected. Whitelisting and break-glass features mirrored those of Binary Authorization.

Unlike Grafeas, the researcher was not able to run Kritis outside of GCP. Instead, the best solution for including image vetting in a private Kubernetes cluster was to implement webhooks directly in Kubernetes. In the process, the researcher reviewed several GitHub repositories with code related to both validating and mutating webhooks. Most code was either outdated, relied on decommissioned Kubernetes features, or simply did not work.

However, the researcher did identify a set of working sample code by Yathi Naik and Malte Misberner (Naik and Misberner, 2019). The sample code included a working mutating webhook. The code consisted of several YAML files and scripts to handle the challenge of certificate generation for use with the webhooks.
The researcher took this initial code and built a working validating webhook for use in non-GCP Kubernetes clusters to provide image vetting. By changing the following sequence of code to reference validating webhooks rather than mutating webhooks, the researcher was also able to spawn a validating webhook in the Kubernetes cluster.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: webhook-server
  namespace: webhook-demo
spec:
  selector:
    app: webhook-server
  ports:
    - port: 443
      targetPort: webhook-api
---
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
  name: demo-webhook
webhooks:
  - name: webhook-server.webhook-demo.svc
    clientConfig:
      service:
        name: webhook-server
        namespace: webhook-demo
      path: "/validate"
      caBundle: $CA_PEM_B64
    rules:
      - operations: ["CREATE"]
        apiGroups: [""
        apiVersions: ["v1"]
```

Figure 9: Updated deployment YAML file (code updated to support validating webhooks)
6. Synthesis, Results, and Implications

Rapid deployment of code makes protecting the software supply chain critical for companies building tools for competitive advantage. Bad images, whether malicious or not, can cause reputational, legal, or regulatory problems.

In reviewing both commercial and open-source solutions available for metadata tracking and image vetting in Kubernetes, the researcher concluded that commercial offerings in GCP are ahead of open-source tools for the non-GCP environment. While Container Registry, Binary Authorization, Grafeas, and Kritis work natively in GCP, similar tools are not readily available for non-GCP Kubernetes clusters.

While the researcher initially looked to deploy Grafeas and Kritis in a private Kubernetes cluster, he ultimately decided to first focus on deploying Grafeas, which worked well in non-GCP Kubernetes clusters. The researcher then spent most of his research time studying validating and mutating webhooks, which are available in native Kubernetes clusters. Much of the research involved downloading code from GitHub and attempting to run it in a Kind Kubernetes cluster. The speed and flexibility by which Kind spins up and tears down Kubernetes clusters allowed the researcher to assess several sets of code, many built to address different technical challenges. Dissecting and understanding the goals of that code helped the researcher better understand how both validating and mutating webhooks work. Further, the researcher learned how to deploy them to achieve additional protection.

7. Further Research

The software supply chain continues to be a risk to organizations that depend on development for competitive advantage. Companies operating in areas that are not yet entirely comfortable with public cloud environments need open-source solutions that they can deploy and manage in private cloud environments. GCP, Azure, and AWS work well for many companies. However, companies operating in the financial sector, as part of government agencies, or in healthcare, are often unable to use these rich public cloud offerings.

While Kritis is coming for non-GCP environments, companies hosting internal clouds or running on non-GCP platforms need tools now to ensure software supply chain integrity. As
such, the definition of validating and mutating webhook rules needs more research. Companies today can use a GitHub repository that contains a variety of webhooks to provide these required checks.

The researcher plans to take the working validating webhook above and experiment with it in his corporate environment. Because he works in the financial space, validating webhooks are most appropriate. By building upon this proof-of-concept code, the researcher can help his organization achieve functional checks in the corporate Kubernetes environment.

### 8. Conclusion

Companies must continue to protect the software development supply chain. Global markets and intense competition make a secure software development supply chain a requirement. The challenge is not new, but the demand for solutions continues to grow.

The continued rise of CI/CD emphasizes the need for thorough testing, trusted third-party code, and automated deployment in software supply chains. Capturing the metadata needed to track and identify reliable sources of truth is no trivial task. Without this metadata, companies struggle to understand what is truly happening in their software supply chain.

Software supply chain management allows companies to protect the critical software assets they create. It ensures that they use only trusted images, authorized components, and reputable third-party tools. Google's commercial tools that help in managing the software supply chain lead the market. Container Registry houses metadata and provides customers insight into what is happening in the CI/CD pipeline. Binary Authorization leverages the data in the Container Registry and provides a checkpoint for images and components before allowing them into production.

Open-source tools provide alternatives for companies hosting internal cloud environments. Grafeas and Kritis are open-source alternatives to Container Registry and Binary Authorization, respectively. These active open-source projects continue to progress and offer companies the ability to track metadata about their CI/CD pipelines and prevent the deployment of unauthorized and potentially damaging code. However, Kritis is limited in that procedures for using the tool outside of GCP are not currently available. As such, security-minded companies hosting internal Kubernetes environments must use built-in tools like admissions controls.
A challenge for companies desiring the flexibility to move between public cloud providers and private cloud environments is to determine how to implement Grafeas and Kritis, or similar tools, in such a way that they achieve the desired mobility and transparently simply. This paper aimed to provide a prescriptive model for deploying tools like Grafeas and Kritis in private cloud environments. The hope is that companies can take these recommendations and build tailored solutions that help them secure their software supply chains, regardless of location.
References


Appendix I – Install both Grafeas and Kritis in GCP

**Grafeas:** Check out your fork of the Kritis repository. Then, navigate to the standalone folder.

```
cd ${GOPATH}/src/github.com/grafeas/kritis/docs/standalone
```

- Set up a GCP project where Kubernetes Engine API is enabled. You'll need to create a new project in GCP.

```
PROJECT=<project ID assigned to you>
gcloud config set project $PROJECT
gcloud components update
gcloud config set compute/zone us-central1-a
gcloud container clusters create kritis-test --num-nodes=2
gcloud container clusters get-credentials kritis-test
```

- Create and upload the Service Account Key:

```
gcloud iam service-accounts create kritis-ca-admin 
   --display-name "Kritis Service Account"
gcloud iam service-accounts keys create gac.json 
   --iam-account kritis-ca-admin@${PROJECT}.iam.gserviceaccount.com
kubectl create secret generic gac-ca-admin --from-file=gac.json
```

Install Grafeas to the cluster with the following script. The script also generates TLS certificates that the Grafeas server uses.

**NOTE:** update `setup_grafeas.sh` as follows

```
helm install command – remove "--name" and save the file.
```

**WARNING:** Make sure to set Common Name to grafeas-server when prompted during the certificate creation.

```
./setup_grafeas.sh
```

**Kritis:**

```
curl -LO https://storage.googleapis.com/resolve-tags/latest/resolve-tags-linux-amd64.tar.gz && 
   RESOLVE_TAGS_DIR=$HOME/.kube/plugins/resolve && 
   mkdir -p $RESOLVE_TAGS_DIR && tar -C $RESOLVE_TAGS_DIR -xzf resolve-tags-linux-amd64.tar.gz && 
   mv $RESOLVE_TAGS_DIR/resolve-tags-linux-amd64 $RESOLVE_TAGS_DIR/resolve-tags && 
   sudo cp $RESOLVE_TAGS_DIR/resolve-tags /usr/local/bin/
```

Install kritis to your cluster:

```
helm install kritis https://storage.googleapis.com/kritis-charts/repository/kritis-charts-0.2.2.tgz
```
Figure 1: Binary Authorization Menus in GCP – Set to not allow any images
Figure 2: Pull down an image

Figure 3: Image denied based on Binary Authorization configuration
Figure 4: Policy changes to "Allow All Images"

Figure 5: Added specific image path to allow list
Figure 6: Pod successfully deployed

Figure 7: Create attestor from gcloud command line
Figure 8: Create Note

Figure 9: Create attestation note payload
Figure 10: Note created

Syntax to create a note attestor:

gcloud beta container binauthz attestations create --artifact-url="${IMAGE_PATH}@${IMAGE_DIGEST}" --attester="projects/${PROJECT_ID}/attestors/${ATTESTOR}" --signature-file=${GENERATED_SIGNATURE} --public-key-id="${PGP_FINGERPRINT}"
Appendix III – Kubernetes Cluster Installation – Kind

1. Create a cluster using Kind
   a. `sudo kind delete cluster`
   b. `sudo kind create cluster --config kind-config.yaml`
      i. Syntax of kind-config.yaml

         ```yaml
         kind: Cluster
         apiVersion: kind.x-k8s.io/v1alpha4
         nodes:
           - role: control-plane
           - role: worker
           - role: worker
         1. Determines the number of nodes in the cluster
         2. Configurable
         ii. Nodes
         iii. Control plane
         iv. CNI
         v. Storage Class
         vi. Joins worker nodes
         ```

2. Install app using kubectl
   a. `sudo kubectl apply -k ./`
      i. Example here was a WordPress instance
         1. `kubectl get services WordPress`
         2. `sudo kubectl port-forward services/wordpress 7676:80`
Appendix IV – Installing Grafeas and Kritis in a private Kubernetes cluster

Install Go:
1. wget https://dl.google.com/go/go1.13.linux-amd64.tar.gz
2. sudo tar -C /usr/local -xzf go1.13.linux-amd64.tar.gz
3. export PATH=$PATH:/usr/local/go/bin

Install Helm:
- $ git clone https://github.com/helm/helm.git
- $ cd helm
- $ make
- Add helm to path
  - o export PATH="/home/woodrobe/helm/helm/bin:$PATH"

Grafeas: cd /home/woodrobe/go/src/github.com/grafeas/kritis/docs/standalone
- Create Secret using JSON file (see appendix 5 for JSON syntax):
  kubectl create secret generic gac-ca-admin --from-file=gac.json

Install Grafeas to the cluster with setup_grafeas.sh.

NOTE: update setup_grafeas.sh as follows

helm install command – remove "–name" and save the file.

WARNING: Make sure to set Common Name to grafeas-server when prompted during the certificate creation.

/setup_grafeas.sh

Kritis:

curl -LO https://storage.googleapis.com/resolve-tags/latest/resolve-tags-linux-amd64.tar.gz && \
  RESOLVE_TAGS_DIR=$HOME/.kube/plugins/resolve && \
  mkdir -p $RESOLVE_TAGS_DIR && tar -C $RESOLVE_TAGS_DIR -xf resolve-tags-linux-amd64.tar.gz && \
  mv $RESOLVE_TAGS_DIR/resolve-tags-linux-amd64 $RESOLVE_TAGS_DIR/resolve-tags && \
  sudo cp $RESOLVE_TAGS_DIR/resolve-tags /usr/local/bin/
Install kritis to your cluster:

```
helm install kritis https://storage.googleapis.com/kritis-charts/repository/kritis-charts-0.2.2.tgz
```

Figure 1: grafeas and kritis running on a local Kubernetes cluster
Appendix V – Contents of gac.json

This researcher uses this file, which comes from the Kritis deployment in GCP, to create keys needed in the private cloud Kubernetes cluster. The researcher was able to start the Kritis service in a private cloud. However, modification is needed as Kritis could not see the ValidatingWebHook that was created by Kritis using the certificates generated with this file.

```json
{
    "type": "service_account",
    "project_id": "woodise5901a",
    "private_key_id": "3f50d3d2db4059c520c6d62f7bba8bd7276769a",
    "private_key": "-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAAOCKCBcgSjAgEAoIBAQCgRt6Y4icA4MGz\nw2q3lRo0nBjYOrq+EbNnxX3qyho0OckFqWf4WV/VP7mCXAE6C9qKistemueQbSJEh\n\nMn5shwqiyP5+jOg5+R5lWzy9eeVFCroagJ0pHJdImBasm7mTK7kD2zn59vEaaun\nxuyWBOBNJSSQ5BI6yfMQrbQ3nPzQ6o6K4D1Qwaei1l7fmfmcq+JK0d6hKNeNhe4\n1Av\n\n7NPL4qscvg6l6eq2G+nY0yADszGR18hUdQDBdFyuH+UC2N+BEV3APjvoZnm/\n\nO/tYNcT422WlLzQZh7NT/3k45R4Q3hdeKhrp2YwpyrGybmnnvPKh9S\nG12YFyN7q\n\n------END PRIVATE KEY-----",
    "client_email": "kritis-ca-admin@woodise5901a.iam.gserviceaccount.com",
    "client_id": "100392186007034736626",
    "auth_uri": "https://accounts.google.com/o/oauth2/auth",
    "token_uri": "https://oauth2.googleapis.com/token",
    "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
    "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/kritis-ca-admin@woodise5901a.iam.gserviceaccount.com"
}
```

Appendix VI – Summary of GitHub and related research

- **Command to interact with images in a Kubernetes cluster**
  
kubectl exec --stdin --tty wordpress-99787fff5-6z84s -- /bin/bash

- **Attempt to use a validating webhook outside of Kritis – did not work**
  
  [https://banzaicloud.com/blog/k8s-admission-webhooks/](https://banzaicloud.com/blog/k8s-admission-webhooks/)
  
  [https://github.com/banzaicloud/admission-webhook-example](https://github.com/banzaicloud/admission-webhook-example)

  This site and GitHub repository walked through setting up a validating webhook in a Kubernetes cluster. The process finished without error, but the webhook did not function as advertised. The researcher reviewed the code did not find the issue.

  Here are the steps that the researcher followed:

  1. ./deployment/webhook-create-signed-cert.sh
  2. kubectl get secret admission-webhook-example-certs
  3. kubectl create -f deployment/deployment.yaml
  4. kubectl create -f deployment/service.yaml
  5. cat ./deployment/validatingwebhook.yaml | ./deployment/webhook-patch-ca-bundle.sh > ./deployment/validatingwebhook-ca-bundle.yaml
  6. cat deployment/validatingwebhook-ca-bundle.yaml
  7. kubectl label namespace default admission-webhook-example=enabled
  8. kubectl create -f deployment/validatingwebhook-ca-bundle.yaml namespace default -o yaml
  9. kubectl create -f deployment/sleep.yaml (should have thrown an error)
  10. kubectl create -f deployment/sleep-with-labels.yaml

- **Kelsey Hightower Validating Webhooks - Failed**
  
  [https://github.com/kelseyhightower/denyenv-validating-admission-webhook](https://github.com/kelseyhightower/denyenv-validating-admission-webhook)

  This repository also contained sample code for a validating admission webhook. While the code did not work as expected, the site had some excellent information about validating webhooks.

- **Giant Swarm Example**
  
  [https://docs.giantswarm.io/guides/creating-your-own-admission-controller/](https://docs.giantswarm.io/guides/creating-your-own-admission-controller/)
Here is another set of code that seemed to make use of older Kubernetes functionality.

- **Silverlock Example:**
  
  [https://blog.questionable.services/article/kubernetes-admission-control/](https://blog.questionable.services/article/kubernetes-admission-control/)

  Silverlock's example was promising, but the researcher was not able to get the code working as described.

- **Stackrox Example:**
  

  The Stackrox example was a simple set of code for deploying a mutating webhook. The code worked as described and serves as a basis for the validating webhook shown in Appendix VII. The researcher plans to use this as a basis for future validation hooks in his private cloud environment.

Here are the steps taken to test the code in the researcher's test environment:

1. Deploy a clean Kind Kubernetes cluster
2. Run deploy.sh (creates CA, cert and private key, and launches pod in a webhook-server namespace)
   a. Creates namespace called "webhook-demo"
   b. Creates TLS secrets
   c. Updates YAML with generated secrets
   d. Creates demo-webhook MutatingWebhookConfiguration from deployment/deployment.yaml.template

**TODO:**

1. Use this base to modify what rules are checking.
2. Attempt to use code to introduce a ValidatingWebhookConfiguration
Appendix VII – Distributed Kubernetes cluster installation

Steps to set up the environment

1. Install Kubernetes on Master and all Nodes

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
```

# Set SELinux in permissive mode (effectively disabling it)
setenforce 0

```
setenforce 0
```

```
sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config
```

```
yum install -y kubelet kubeadm kubectl --disableexcludes=kubernetes
```

```
systemctl enable --now kubelet
```

2. Turn off Swap and update /etc/fstab on Master and all Nodes

```
swapoff -a
vim /etc/fstab (Comment out last line
```
3. Set hostname for each machine (Master and all Nodes)

4. Ensure that Docker and Kubernetes are started upon boot (Master and all Nodes)

   systemctl enable docker.service
   systemctl enable --now kubelet

   REBOOT

5. Download overlay network YAML files (Master only)

   wget https://docs.projectcalico.org/v3.3/getting-started/kubernetes/installation/hosted/kubernetes-datastore/calico-networking/1.7/calico.yaml
   wget https://docs.projectcalico.org/v3.3/getting-started/kubernetes/installation/hosted/rbac-kdd.yaml

6. Set up Pod Network (Master only)

   service firewalld stop

   # Setup daemon.
   cat > /etc/docker/daemon.json <<EOF
   {
     "exec-opts": ["native.cgroupdriver=systemd"],
     "log-driver": "json-file",
     "log-opt": {
       "max-size": "100m"
     },
     "storage-driver": "overlay2",
   }
   EOF
"storage-opts": [  "overlay2.override_kernel_check=true"
]
EOF

mkdir -p /etc/systemd/system/docker.service.d

# Restart Docker
systemctl daemon-reload
systemctl restart docker

sudo kubeadm init --pod-network-cidr=192.168.0.0/16

7. Run as Regular User
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

8. Apply YAML files
kubectl apply -f RBAC-kdd.yaml
kubectl apply -f calico.yaml

Verify with:
kubectl get pods --all-namespaces
kubectl get nodes
sudo systemctl status kubelet.service
ls /etc/kubernetes
ls /etc/kubernetes/manifests

9. Useful checks

kubeadm token list

#to create a new token
kubeadm token create

#if you need to get the discovery token
openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der 2>/dev/null | openssl dgst -sha256

EXAMPLE

kubeadm join 192.168.50.143:6443 --token vcycku.z1f2u2bel8l1zba5 \\n  --discovery-token-ca-cert-hash
sha256:779e1742c77e2707001a12fe38ffd4abbd3df21e9166327596949d7f733da596

10. If get nodes show "Not Ready," rerun the following

kubectl apply -f calico.yaml