PLATFORM9

Securing Kubernetes: A Primer

Platform9 Kubernetes Journey eBook series

- Understanding Kubernetes
- Architecting Kubernetes Deployments
- Operating Kubernetes
- Scaling Kubernetes
- Securing Kubernetes 🡐 This eBook

Download at: https://www.platform9.com/k8s-journey

Table of Contents

Introduction

Security is perhaps the most complex dimension of Kubernetes – which is saying a lot, because Kubernetes as a whole is quite complex.

Security in Kubernetes is complex not only because there are so many types of threats to consider and layers to secure, but also because Kubernetes has an especially complicated security model. Kubernetes includes some built-in security features, but they address only certain categories of risk. As a result, Kubernetes admins must deploy additional tools to manage some of the potential vulnerabilities in a Kubernetes environment, such as those associated with container images and registries.

On top of this, the methodologies that admins use to configure Kubernetes's native security features – as well as to integrate it with third-party security tools – play a key role in shaping overall security outcomes. There are best practices to follow when working with security functionality in Kubernetes, as well as pitfalls to avoid.

Navigating Kubernetes security challenges

This eBook serves as a guide to help admins navigate the complicated Kubernetes security landscape. It begins in the first chapter by describing which types of security features and functionality Kubernetes does and does not offer, out-of-the-box. Subsequent chapters discuss how to integrate external tools to help fill the gaps left by Kubernetes's native security regime in critical areas, such as the encryption of network traffic for Kubernetes applications and the secure management of secrets in a Kubernetes cluster.

In addition to walking through the nuts and bolts of various security considerations in Kubernetes, this eBook also discusses best practices for placing security front-and-center when working with Kubernetes configuration files and managing integrations between Kubernetes and external tools. In this way, it aims to instill good security habits in Kubernetes admins, in addition to teaching them about certain technologies that can help mitigate security vulnerabilities in Kubernetes environments.

On balance...

The information presented in the following chapters is not a totally comprehensive guide to Kubernetes security. There are a variety of additional tools that could be used. There are also security considerations that this eBook does not address but that may be relevant in some Kubernetes deployments, such as managing roles and access control using a cloud provider's IAM framework when running managed Kubernetes on a public cloud.

We should note, too, that certain Kubernetes distributions are more "opinionated" than the core Kubernetes open-source project about which security tools and methodologies to use. If you deploy Kubernetes using one of these distributions instead of a distribution that aligns more closely with the open-source version of Kubernetes, you may discover that your distribution features more built-in security features than those described here.

Nonetheless, for admins seeking a concise but meaningful guide to Kubernetes security, we hope you find it in the pages that follow.

Should you have questions about managing security or any other dimension of your Kubernetes environments, the Kubernetes experts at Platform9, who specialize in providing the tools and support needed to make Kubernetes a turnkey experience for companies large and small, are always happy to help. Contact us anytime.

Kubernetes Security: What (and What Not) to Expect Out-of-the-Box

When it comes to security, there is a lot that Kubernetes does. There is also a lot that it doesn't do.

To secure Kubernetes effectively for real-world deployment, you must understand which built-in security features Kubernetes offers and which it doesn't, and how to leverage Kubernetes's security features at scale.

In this section, we walk through Kubernetes's security architecture, and discuss best practices for securing production Kubernetes deployments.

Kubernetes components overview

At first glance, Kubernetes security might seem simple. But Kubernetes is a complex tool, and a Kubernetes deployment involves many layers and moving parts. Securing Kubernetes is actually a bit complicated.

A Kubernetes deployment consists of many different components, including:

- The Kubernetes master
- The Kubernetes nodes
- The server that hosts Kubernetes
- The container runtime you use with Kubernetes
- Networking and transport layers within your cluster
- Any interface, logging, monitoring and other tools you use in conjunction with Kubernetes
- The applications that run inside containers hosted on Kubernetes

Securing Kubernetes requires you to address security challenges associated with each of these components.

Built-in Kubernetes security features

Fortunately, Kubernetes offers several built-in security features to help secure the components described above.

RBAC

Role-based access control, or RBAC, lets you specify which users are able to perform certain actions (like reading pods) using the Kubernetes API. The access policies you specify can be applied within a given namespace, or (if you define a ClusterRole) across multiple namespaces.

Pod security policies

Think of Kubernetes pod security policies as sort of the opposite of RBAC policies. While RBAC lets you control which actions certain users can perform on pods (or other resources), pod security policies are used to restrict the actions that pods are allowed to perform. You can create pod security policies that do things like prevent containers from running as root, or disallow a pod to share the process ID namespace with its host.

Network policies

Kubernetes allows you to define a Network policy to control what traffic is allowed to flow between different pods and endpoints in your cluster.

Network policies aren't designed primarily as a security tool; they are mainly a way to manage network traffic and avoid unnecessary network load. However, because they allow you to block traffic to or from certain pods or endpoints, they can essentially function as a sort of firewall if you want them to. Network policies are a handy way to help lock down the networks inside your cluster in order to reduce security risks associated with the network.

Secrets management

If you are running an application on Kubernetes that needs to access secrets (like passwords or an SSH key), you can keep those secrets secure by using Kubernetes's built-in secrets management framework. To do this, you use the kubectl command to translate your secret information into a special object (which Kubernetes refers to as a Secret, with a capital S) that you can later make accessible to your pods.

Thanks to Secrets, you don't have to do insanely insecure things, like store passwords as plain text files inside your container images.

What Kubernetes doesn't secure (on its own)

Although the features described in the preceding section address some of the security needs of a Kubernetes deployment, they don't cover everything. There are a number of notable gaps, including:

- **Host security**: Kubernetes host security is on you. Kubernetes does nothing to keep host servers secure.
- **Images**: Your application is only as secure as the images you use to run it. Kubernetes won't check your images for insecure code or other vulnerabilities.
- **Container runtime**: No matter which runtime you choose to use with Kubernetes, Kubernetes won't do anything to detect breaches that exploit the runtime or known security vulnerabilities within the runtime.

You'll either have to manage these Kubernetes security needs manually, or use a dedicated security tool that provides these features.

Keeping Kubernetes secure at scale

It's easy to talk about Kubernetes security features and best practices when you're dealing with an experimental cluster or a small-scale deployment where you can manage everything by hand.

But what if you have a large, production-grade Kubernetes deployment that comprises dozens of pods (or more) spread across a sprawling infrastructure? In that environment, it's critical to streamline your approach to Kubernetes security in order to keep it manageable and consistent.

The following strategies can help:

- **Focus on reusable policies**: You don't want to have to recreate or update RBAC or pod security policy configurations every time your cluster changes or you redeploy an app. Strive to write policies that can be reused indefinitely.
- **Write broad policies**: For efficiency's sake, it's often preferable to write a policy that applies to your entire cluster rather than one that applies only to one specific resource. You can't do this in all situations, of course, but when possible, write policies that are as broad as possible.
- **Secure your images and hosts**: As we noted, Kubernetes itself doesn't do anything to help secure images or hosts. Fortunately, many third-party tools do. Use frameworks like SELinux or AppArmor to harden your hosts, and make sure you scan your container registries for known vulnerabilities.
- **Consider a managed Kubernetes service**: Although the security services provided by managed Kubernetes platforms vary from vendor to vendor, most managed offerings will provide at least some security support and reduce the amount of security considerations that you need to manage yourself.

Chapter Summary

Like most things that involve Kubernetes, Kubernetes security is a complex, multi-faceted discipline. Fortunately, there is no shortage of solutions and best practices for helping to secure Kubernetes deployments, even at scale. Once you wrap your mind around what is available, keeping Kubernetes secure, while still not simple, is much less difficult than you might think.

Kubernetes Secrets Management

As noted in the previous chapter, Kubernetes provides a built-in feature for managing secrets (such as SSH keys, API keys and passwords that a Kubernetes application or service may require) through the kubectl tool. However, there is more to consider regarding Kubernetes secrets management than this. It's also possible to use third-party secrets management tools in conjunction with Kubernetes, and this approach can be preferable in many scenarios.

In this chapter, we walk through Kubernetes secrets management, and explain some of the options beyond the native functionality in this regard.

An Introduction to Kubernetes Secrets

Kubernetes provides users with the ability to create secrets objects and provide them securely to pods within the cluster. Let's consider a scenario where we need to provide access credentials for a data store to pods within our cluster. We'll need to create and store the secrets within the cluster, and then make them available to the applicable pods.

You can create secrets manually or use the *kubectl create secret* command to create them. The *create secret* command can create secrets from string literals, or files containing the values. Using the file approach eliminates the need to escape special characters from the operating system. If we have our database credentials loaded into files named *database_user.txt* and *database_password.txt*, then we could load them into the cluster's secret store with the following command.

```
$ kubectl create secret generic database-credentials --from-
file=./database_user.txt --from-file=./database_password.txt
```

This command creates a secret named ***database-credentials*** in the secret store. The system should respond with a ***secret "database-credentials" created*** response.

Once you have created the secret object, you can add it to the container as volume attached to the pod, or you can load the values into environment variables when initializing a new container in the pod. Let's look at an example configuration that includes the secrets as a volume on the pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: apppod
spec:
  containers:
  - name: apppod
    image: application
  volumes:
  - name: secrets
    secret:
        secretName: database-credentials
```

Assuming that you used the original file names to load the database credentials into the secrets object and attached the secrets into a volume named *secrets,* the credentials could now be accessed at */etc/secrets/database_user* and */etc/secrets/database_password*.

Essential Considerations With Kubernetes Secrets

While using a Kubernetes secret object is more secure than including credentials within the image, there are some critical characteristics of secrets that you should keep in mind. From a positive perspective, the use of a secret object simplifies administration if credentials need to be updated. An updated secret is available to all of the containers using it as soon as it is updated.

Additionally, because secrets are created independently of the pods, they are less likely to be exposed during the process of creating and configuring the pods. Administrators can also enable encryption at rest if the pod is running on a recent version of Kubernetes in order to protect the data further.

But there are also some potential pitfalls that you should be aware of when using a secret object. Below, I'll list some of the primary concerns and risks that you should keep in mind when deciding whether to use Kubernetes Secrets to hold your sensitive information. This is by no means an exhaustive list, and you should consult the official Kubernetes documentation on Secrets for a full and current list of potential risks and best practices.

First of all, secrets in a cluster may be available to all users of the cluster, so be careful about uploading personal credentials if you're working in a shared environment. Secondly, if you load the secret into environmental variables, you should ensure that these logging statements don't include the variables, as this exposes the secret in plain text in the logs. Finally, when creating secrets, you might use base64 encoded data or input files with the plaintext values. You should ensure that you delete these resources following the creation of the secrets so that they don't inadvertently end up in source control.

Using A Non-Native Solution for Secrets Management

Kubernetes Secrets is an adequate solution for most secrets. Still, there are more robust security systems available from providers with a long pedigree of secrets management and focused investment in their solutions. The integration of one of these systems might be overkill for a small application footprint. Still, for larger organizations as well as organizations that support a hybrid of Kubernetes and other platforms, they may provide a more secure option.

Solutions like Vault from HashiCorp and CyberArk offer highly secure systems for secrets management and provide integrations with Kubernetes as well as all of the primary cloud providers. These systems usually include secrets management strategies, auditing, and role-based access control, and they might even integrate seamlessly with your corporate data center.

HashiCorp recently announced additional features that will allow Vault to integrate easily with a Kubernetes environment. CyberArk offers similar features with its flagship secrets management product, and they also offer an open-source product, Conjur Open Source, which is container-native and can integrate effortlessly with your Kubernetes environment.

In addition to these options, there are other offerings from other providers of security management systems. Before deciding on the right solution for your organization, you should be sure to examine the available options carefully so that you can select the best solution for your security needs as well as your budget.

Learning More

If you would like to learn more about Kubernetes Secrets, the official Kubernetes documentation provides a thorough look at Secrets and how to implement them. Similar information is also available in the documentation on how to Distribute Credentials Securely Using Secrets.

Using Let's Encrypt with Route 53 to Secure Kubernetes

The general public's awareness of application security has never been higher. Thankfully, with cloud-native applications, microservices, and similar technologies being deployed more often as containers in Kubernetes clusters, having everything managed by Kubernetes has given administrators an easier way to have a publicly validated Transport Layer Security (TLS) certificate presented for all of their applications through the Kubernetes ingress controller.

Doing so requires the integration of some external tools to encrypt traffic from Kubernetes applications. In this chapter, we explain how to do this using two popular tools: Let's Encrypt and Route 53.

The Rise and Mass Adoption of TLS

Security has always been an important part of providing reliable services. It used to be quite expensive, though, because it required what was often seen as excessive hardware capacity, and even application delivery controllers started to bundle custom hardware components to offload the encryption workload from the main CPU. The introduction of cloud computing in the early 2010's enabled almost everyone to achieve secure communications, and web sites and web browsers soon began to flag traffic that was not encrypted.

In 2014, the availability of computing resources, along with the public's increasing demand for more security, highlighted a problem that the Internet Security Research Group (ISRG) needed to address. The problem was that there were no automated, free, and transparent agencies generating TLS certificates for the general public -- there were only a few free (but painfully slow) providers and a plethora of expensive automated providers to choose from. Consequently, many organizations relied on self-signed certificates, which made their sites difficult to work with and often caused web browsers to flag them as potentially fraudulent. Self-signed certificates have other issues, such as essentially training your users to ignore invalid certificates and indiscriminately hit accept/continue, which is how most man-in-the-middle attacks happen, and also how personal and corporate information can be stolen.

In order to help every site reach the minimum level of TLS security, the ISRG started an organization in conjunction with the Linux Foundation (and initially sponsored by the EFF) called Let's Encrypt. Let's Encrypt provides an open and transparent way to automatically generate certificates that expire in 90 days. The ISRG has also persuaded major browser vendors and projects to include their root certificate as a recognized authority, making it as good as any other from a consumer-visibility point of view. Everyone loves that green bar in their browser that says "secure."

Using Let's Encrypt - The Basics

Let's Encrypt is built around the Automatic Certificate Management Environment (ACME and rfc8555) protocol and provides backend security and certificate authority. It relies on client-side requests from one of three major avenues: a manual mode, a hosting provider that has partnered with them to provide client-side tools, or one of many ACME client utilities (primarily Certbot).

When the certificate is requested by a client, Let's Encrypt has its backend services validate that the domain is under the control of the requestor by executing an HTTP request on port 80 to the domain listed in the certificate.

Certbot, for example, starts up a basic web server on port 80 and listens for the event; after processing the request, it shuts down the web server, writes the certificates to file, and retains its configuration, which you will reuse to renew the certificate before it expires in 90 days. Then, when Certbot is reactivated, it does it all over again.

So what happens if port 80 is already in use on that domain, or if the domain is part of a cluster where you can't control which machine the request will go to? In both of these cases, Let's Encrypt can validate the domain through DNS. It does this by providing a unique and custom text string which will be added to a specific subdomain as a TXT record. Then, Let's Encrypt will query for those records and issue the certificates after the records are found. This provides a way to renew certificates with absolutely no impact on running applications, since this process is completely out of bounds of the normal HTTP request flow.

Hosting a Domain in AWS Route 53

If you have any resources in the cloud, you likely have some (if not all) of them in an AWS region. One of the best services available on AWS is its DNS service Route 53. EKS from AWS has native integration with Route 53 as part of the service life cycles for all of the subdomains that it manages.

You can add a domain either through the Web Console or via the AWS CLI. Below, we will demonstrate the process using CLI:

The first step is to point a domain at Route 53's DNS servers. This can be done by redirecting an existing domain, or by registering a domain directly using route53domains.

Buying a domain is as simple as running one command; well, two if you set contact information as a variable:

```
$ export CONTACT='{"FirstName": "string","LastName": "string","ContactType":
"PERSON","OrganizationName": "string","AddressLine1": "string","AddressLine2":
"string","City": "string","State": "string","CountryCode":”US”,"ZipCode":
"string","PhoneNumber": "+1.1234567890","Email": "string","Fax":
"string","ExtraParams":[]}'

$ aws route53domains register-domain --domain-name example.com --duration-in-years 1 -
-admin-contact $CONTACT --registrant-contact $CONTACT --tech-contact $CONTACT
{
    "OperationId": "abcdef01-2345-6789-0098-87654321fedc"
}
```

Domains registered through AWS are registered automatically. If you are using an existing domain, then you will need to register it as a hosted zone:

```
aws route53 create-hosted-zone --name example.com --caller-reference 2019-10-10-20:20
--hosted-zone-config Comment="command-line version"
```

Creating and Validating a Certificate Using Let's Encrypt

To show all of the steps required for creating and validating a certificate using Let's Encrypt, we will be using an Ubuntu 18 instance. Ubuntu 18 has all of the required utilities available in software repositories, including the AWS tools. They just need to be configured using "`aws configure`".

Next, we install Certbot and AWS CLI to work with Let's Encrypt and Route 53.

```
$ sudo apt-get install software-properties-common
$ sudo add-apt-repository universe
$ sudo add-apt-repository ppa:certbot/certbot
$ sudo apt-get update
$ sudo apt-get install certbot awscli python3-certbot-dns-route53
```

Now we run the configuration for the AWS CLI:

```
$ aws configure
AWS Access Key ID [None]: ABCDEF
AWS Secret Access Key [None]: GHIJKLMNOPQRTUVWXYZ
Default region name [None]: us-west-2
Default output format [None]:
```

Then it's time to create the actual certificate:

```
$ certbot certonly --dns-route53 -d example.com -d '*.example.com' --logs-dir
/home/ubuntu/letsencrypt/log/ --config-dir /home/ubuntu/letsencrypt/config/ --work-dir
/home/ubuntu/letsencrypt/work/ --server https://acme-v02.api.letsencrypt.org/directory
-m email@example.com --agree-tos --non-interactive
```

Option	Description
certonly	means certbot will only process the certificate and not try to automatically add it to any web servers
-d	the different domains that will be included in the certificate
-m	the email address that will be used by Let's Encrypt for emergencies and to remind you about renewals
--dns-route53	the validation plug-in that will be used
--logs-dir --config-dir --work-dir	where the configuration and working files will be stored. The default is under /var/logs/letsencrypt and /etc/letsencrypt
--agree-tos --non-interactive	stop certbot from prompting to accept the terms and conditions
--server	points to the explicit version of Let's Encrypt that supports wildcard domains

Here's the output of the successful certbot command; note the lines where it automatically creates and cleans up the actual records in Route 53:

```
Saving debug log to /home/ubuntu/letsencrypt/log/letsencrypt.log
Found credentials in shared credentials file: ~/.aws/credentials
Plugins selected: Authenticator dns-route53, Installer None
Obtaining a new certificate
Performing the following challenges:
dns-01 challenge for example.com
dns-01 challenge for example.com
Waiting 10 seconds for DNS changes to propagate
Waiting for verification...
Cleaning up challenges
Non-standard path(s), might not work with crontab installed by your operating system
package manager

IMPORTANT NOTES:
 - Congratulations! Your certificate and chain have been saved at:
   /home/ubuntu/letsencrypt/config/live/hybridcloud.ninja/fullchain.pem
   Your key file has been saved at:
   /home/ubuntu/letsencrypt/config/live/hybridcloud.ninja/privkey.pem
   Your cert will expire on 2020-02-19. To obtain a new or tweaked
   version of this certificate in the future, simply run certbot
   again. To non-interactively renew *all* of your certificates, run
   "certbot renew"
 - Your account credentials have been saved in your Certbot
   configuration directory at /home/ubuntu/letsencrypt/config. You
   should make a secure backup of this folder now. This configuration
   directory will also contain certificates and private keys obtained
   by Certbot so making regular backups of this folder is ideal.
 - If you like Certbot, please consider supporting our work by:

   Donating to ISRG / Let's Encrypt:   https://letsencrypt.org/donate
   Donating to EFF:                    https://eff.org/donate-le
```

The certificates are created under the config directory, specifically /home/ubuntu/letsencrypt/config/live/example.com/, and it contains four files:

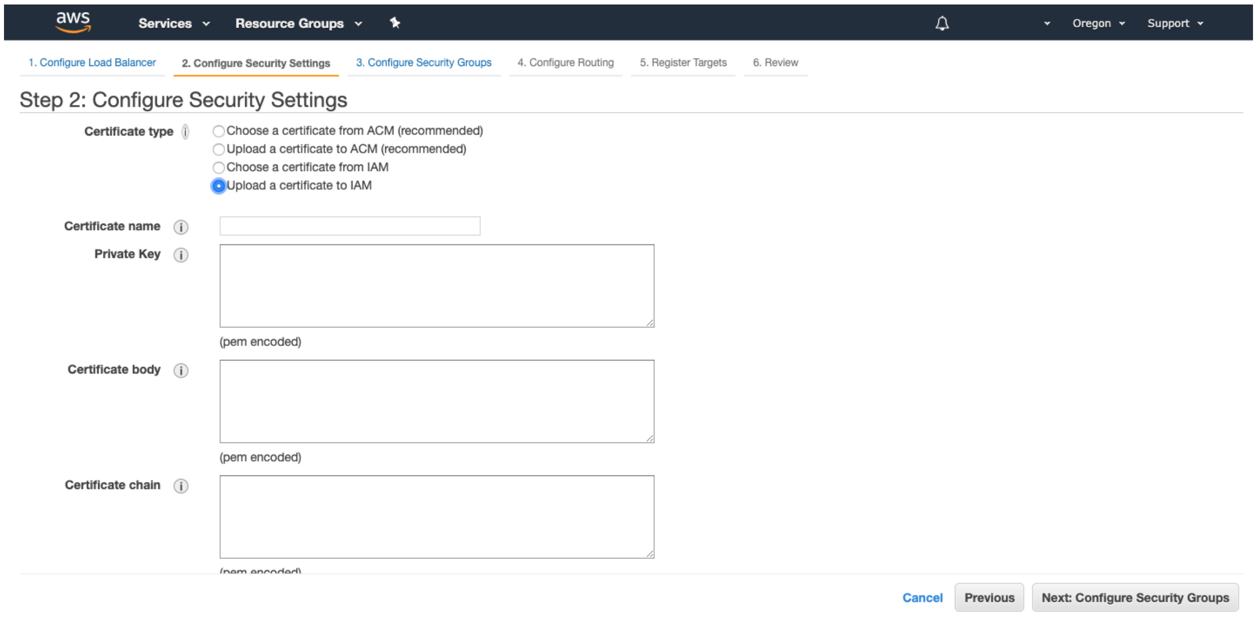
privkey.pem	the private key for your certificate
fullchain.pem	the certificate file used in most server software
chain.pem	used for OCSP stapling in Nginx >=1.3.7
cert.pem	will break many server configurations, and should not be used without reading further documentation

Adding the Certificate to a Node Group in EKS

If you are using EKS as your Kubernetes distribution on AWS, you won't actually add the certificate to EKS; rather, you'll add the Let's Encrypt certificate to the Elastic Load Balancer (ELB) that is fronting the node group that you are going to use to host the domain. The domain name is also associated with the ELB. When creating or updating the configuration of the ELB, there will be an option to either directly import the TLS certificate from Let's Encrypt or attach it using IAM. The former configuration is easier.

During the security configuration for ELB, you will need the content of the privkey.pem, cert.pem, and chain.pem files for the appropriate fields.

The Web UI has three fields that use the cert.pem, chain.pem, and privkey.pem files that you created:


aws
Services
Resource Groups
Oregon
Support
1. Configure Load Balancer
2. Configure Security Settings
3. Configure Security Groups
4. Configure Routing
5. Register Targets
6. Review

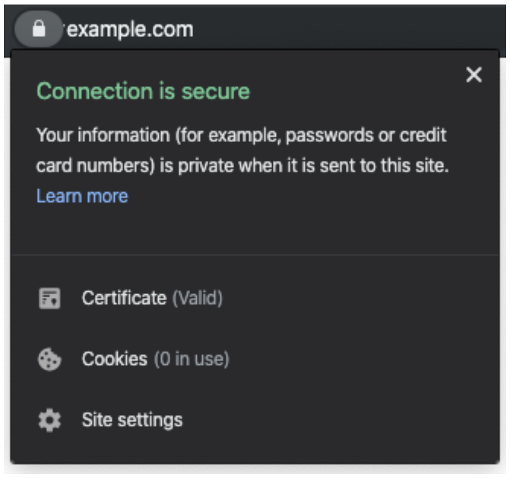
Step 2: Configure Security Settings

Certificate type
Choose a certificate from ACM (recommended)
Upload a certificate to ACM (recommended)
Choose a certificate from IAM
[✓ Upload a certificate to IAM]

Certificate name
Private Key (pem encoded)
Certificate body (pem encoded)
Certificate chain (pem encoded)

Cancel
Previous
Next: Configure Security Groups

Then, when a modern web browser connects, the certificate will be fully validated and accepted:



Connection is secure

Your information (for example, passwords or credit card numbers) is private when it is sent to this site.

Certificate (Valid)

Cookies (0 in use)

Site settings

Adding the Certificate to a Kubernetes Ingress Controller

Let's Encrypt is often used in non-EKS Kubernetes distributions where the certificate will be loaded directly into the ingress controller in the cluster. To perform this operation using the standard Kubernetes ingress resource, the actual product that performs the ingress function must support TLS encryption. The TLS certificate is loaded in a Secret with a specific format that is referenced either from the ConfigMap used by the ingress controller (for example, haproxy) or from the ingress configuration (for example, GKE and nginx).

Import the TLS Secret using the following command:

```
$ kubectl create secret tls exampledotcom-cert --key privkey.pem --cert fullchain.pem
```

A TLS Secret used by the ingress controller would look like the following:

```
apiVersion: v1
kind: Secret
metadata:
  name: exampledotcom-cert
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
```

If TLS will be referenced by the ingress controller, the yaml spec will likely be part of a larger and more complex ingress configuration:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress-controller
spec:
  tls:
  - secretName: exampledotcom-cert
  rules:
  - http:
      paths:
      - path: /*
        backend:
          serviceName: wordpress
          servicePort: 80
```

When referenced from the ConfigMap, it looks like this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: haproxy-configmap
  namespace: default
data:
...
  ssl-certificate: "default/exampledotcom-cert”
  ssl-numproc: "1"
  ssl-redirect: "ON"
  ssl-redirect-code: "302"
...
```

Certificate Signing Automation Including Renewals

As mentioned earlier, Let’s Encrypt certificates are only valid for 90 days, which is great for quick tests, proof of concepts, or other short-lived activities. Ninety days is not very long when dealing with production environments where certificates are typically good for one to three years. Let’s Encrypt allows renewals every 90 days, but it involves basically the same steps as the initial generation.

Because many people have this same issue, JetStack has backed an open source project named cert-manager that has been under development for almost two years, and is starting to gain traction as it gets closer to a stable release. Under the hood, cert-manager has multiple ways to handle certificate registration and renewal including the ACME technology that cerbot uses to interact with Let’s Encrypt.

The idea behind cert-manager is to have a Kubernetes native runtime that handles the creation and renewal of x509 certificates (SSL/TLS) within a single cluster. It can handle all the tasks listed above relating to creating certificates, and will handle renewing those same certificates automatically. Installation for cert-manager can be handled through a manual process or with a version 2 Helm chart. It still has quirks to work through, but once it is running, it works great.

As part of its configuration to talk to Let's Encrypt, it can use a yaml file that will request the certificate, use Route 53 to validate, and then renew the clusters ingress certificate as required.

Define the Issuer - In this case Let's Encrypt with Route 53

```
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-r53
spec:
  acme:
    ...
    solvers:
    # example: cross-account zone management for example.com
    # this solver uses ambient credentials
    # to assume a role in a different account
    - selector:
        dnsZones:
          - "example.com"
      dns01:
        route53:
          region: us-east-1
          hostedZoneID: DIKER8JEXAMPLE # optional, see bpolicy above
          role: arn:aws:iam::XXXXXXXXXXXX:role/dns-manager
```

Define a TLS Certificate with alternate names that will be stored in a Kubernetes secret

```
apiVersion: cert-manager.io/v1alpha2
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: example-com-tls
  issuerRef:
    name: letsencrypt-r53
  commonName: '*.example.com'
  dnsNames:
  - example.com
  - anotherdomain.com
```

Chapter Summary

Let's Encrypt is a basic TLS certificate that will successfully encrypt traffic between a client and the entrance to the Kubernetes cluster, and it will be verified and validated by any modern web browser. It does not offer centralized management of certificates or the extended validation features that many of the pay-to-play certificate vendors offer. However, Let's Encrypt enables internet sites to get rid of problematic self-signed certificates (which are far too common, especially in test and development environments) by providing them with publicly-validated certificates.

Kubernetes Container Registry and Image Scanning

As the first chapter of this guide noted, one critical security consideration that Kubernetes does not address natively is the contents of container images. Malware, insecure configurations or other security vulnerabilities that exist inside container images, or the container registries that host them, cannot be detected by Kubernetes. Nor can Kubernetes do much to mitigate the threats posed by security problems inside containers.

That's why taking steps to secure registries and images is so critical. This chapter explains how. It discusses the basics of container and image security, then identifies several tools that can be used alongside Kubernetes to address this challenge.

What is an Image Registry?

Kubernetes is an orchestration system that allows you to deploy and manage containers. A Kubernetes Pod holds related containers to support an application. When you create a new container within a Pod, you need to provide an image for the container to use. A container image is a self-contained piece of code that has everything it needs to run - including libraries, dependencies, tooling and configuration.

The Image Registry is where images are stored and can be retrieved to build containers. The Image Registry can be either a public or a private repository.

Public Container Registry

Currently, Docker Hub is the default and most widely used public container registry. Users can access many official images and can upload their images to use and to share with other developers. Using a public registry is not always a viable option for some organizations. It is a best practice to store proprietary images and sensitive configurations in a private image registry.

Using a Private Registry

Private registries provide a repository for both customized and commonly used images for an organization. The organization controls both access and the contents of the registry, protecting intellectual property, and preventing malicious injection of malware into the images the organization uses.

By using a private repository, you reduce the chance of exposing proprietary information or being attacked through a malicious change to an image. Container images also run the risk of having outdated dependencies and inadvertent vulnerabilities. Performing regular scans of your registry with a scanning tool can alert you if dependencies change, or if an image update introduces any vulnerabilities.

Comparing Popular Private Registry Options

As containers continue to grow in popularity, so do the numbers and quality of the tools that support their proliferation. The most popular options for private registries are also those with the most extended pedigree. Let’s review and compare four of the most popular options: Docker Hub private repositories, Google Cloud Registry, and Amazon Elastic Container Registry.

Docker Hub Private Repositories

As the default registry for Docker, most developers are already familiar with Docker Hub, and their private registry offering makes it easy to use Docker Hub for both private and public registries. Docker Hub is also relatively inexpensive, although supplemental services like scanning are available for an additional fee. An intuitive interface makes it easy to create an organization, groups, and users and control permissions to restrict and allow access.

Docker also offers the Docker Trusted Registry (DTR), which is an enterprise-grade registry that can be installed behind your corporate firewall or on a virtual private cloud (VPC). DTR includes an image scanning utility that can be added to the registry for a fee.

Google Cloud Registry (GCR)

As the originators of Kubernetes, you would expect Google to be a significant provider of related services, including an intuitive and inexpensive registry option. Google Cloud Registry is an affordable option, charging users only for storage and bandwidth costs.

GCR is also configured to allow access within your Google Cloud account, and with external resources as well. This flexibility makes it an excellent choice as a private registry provider, whether your containers are deployed on the Google Cloud, or elsewhere.

Amazon Elastic Container Registry (ECR)

If your organization is already using AWS, ECR uses a familiar interface and is simple to integrate into your existing Cloud infrastructure. The cost is relatively inexpensive and is limited to image storage fees and data transfer. Using ECR to support a Kubernetes cluster outside of AWS has the potential of being both more expensive and complicated.

Using AWS's highly configurable IAM offering affords a very secure access control model, which is scalable; and roles for human and non-human entities make management relatively simple. As with many of the AWS services, many of the details are abstracted from the AWS interface, which simplifies interactions, but can be problematic if you require insights into the inner workings of the registry.

JFrog Artifactory and Container Registry

JFrog's Artifactory is a universal binaries manager. JFrog Container Registry supports both Docker and Helm images, to enable you to build, store, and manage container images for all types of deployments. It is available as a self-hosted, dedicated solution, or as a SaaS service – enabling organizations to create local, remote and virtual image repositories.

The Importance of Image Scanning

One of the unique and advantageous characteristics of containers is how layers are used to build a container image. A service that you build might be in a layer added to a JVM layer, application server layer, and a Linux layer. When one of those layers is updated, you can rebuild your container and create a new and updated version with relative ease.

Unfortunately, the layered architecture also means it is easy to accidentally introduce vulnerabilities within one or more of the layers that comprise your container. A registry scanner can scan new images and periodically scan existing images to identify potential vulnerabilities and assist you in maintaining a secure and high-quality collection of images.

Selecting the Right Image Scanning Tool

At the heart of any image scanning tool is static analysis against a "Common Vulnerabilities and Exposures" (CVE) database. Each layer within the container image is analyzed and queried to discover known vulnerabilities. One such CVE, which is used by several providers, is the Clair Project.

In addition to vulnerability scanning, a comprehensive tool should compare the architecture of your application against best practices to identify potential vulnerabilities. Compliance scans examine your application to ensure that best practices are employed when managing secrets, ingress and egress points, and system configurations.

Image Scanning Options

All three of the registries we compared above offer CVE analysis as part of their offering, or as an add-on feature. CVE analysis is an essential aspect of your container security strategy; however, it should not be relied upon solely to secure your image registry. Let's look at and compare a few of the leading image security providers.

Twistlock

Recently acquired by Palo Alto Networks, Twistlock is one of the most comprehensive container security solutions available. Developed for the Cloud and explicitly focused on container security, Twistlock integrates seamlessly with Docker Hub, GCR, and ECR. Executing CVE based scans, as well as Security compliance and run-time defense, Twistlock provides comprehensive reports about all vulnerabilities within your Kubernetes ecosystem.

Aqua Security

Offering a scanning solution that can span multiple platforms, Aqua Security's Cyber Intelligence employs multiple vulnerabilities, malware detection, and threat mitigation services to ensure your container ecosystem is secure. While Twistlock focuses specifically on Container Security, Aqua offers additional security products outside of the container space.

SysDig

Finally, Sysdig Secure offers similar capabilities to both Aqua Security and Twistlock, including CVE, compliance, and run-time defense. An additional feature, which is offered by SysDig and by both Twistlock and Aqua Security, is the ability to perform post-mortem forensics on the environment.

System forensics allows you to see the before-and-after state of a system. This understanding enables both fully understanding the circumstances that led to the problem, and it's rapid resolution.

Learning More

You would be ill-advised to pursue a container-based strategy without integrating a comprehensive and well-supported image scanning solution into your CI/CD processes. Outsourcing security to a vendor who specializes in security and is continually improving their processes is the best way to ensure the security of your environment.

Each of the solutions listed above can meet your needs, but it's important to find the solution that best meets your needs, in terms of budget, support, and environment.

Conclusion

When it comes to Kubernetes security, there is certainly no turnkey, one-size-fits-all solution. Instead, securing Kubernetes requires understanding the many types of risks that exist across all of the components of a Kubernetes environment, then leveraging the tools and methodologies available to address each one. Some of these tools are built into Kubernetes, while others are third-party solutions.

Understanding Kubernetes security is challenging, even for seasoned developers and admins. If you need further guidance, the experts at Platform9 are always ready to help. Contact us to talk Kubernetes security anytime.