管理 Admission Webhook 的 TLS 证书

本文介绍如何管理 Admission Webhook 的 TLS 证书。

前面我们学习了如何开发自己的准入控制器 Webhook，这些准入 Webhook 控制器调用自定义配置的 HTTP 回调服务来进行其他检查。但是，APIServer 仅通过 HTTPS 与 Webhook 服务进行通信，并且需要 TLS 证书的 CA 信息。所以对于如何处理该 Webhook 服务证书以及如何将 CA 信息自动传递给 APIServer 带来了一些麻烦。

前面我们是通过 openssl（cfssl）来手动生成的相关证书，然后手动配置给 Webhook 服务的，除此之外，我们也可以使用 cert-manager 来处理这些 TLS 证书和 CA。但是，cert-manager 本身是一个比较大的应用程序，由许多 CRD 组成来处理其操作。仅安装 cert-manager 来处理准入 webhook TLS 证书和 CA 不是一个很好的做法。

另外一种做法就是我们可以使用自签名证书，然后通过使用 Init 容器来自行处理 CA，这就消除了对其他应用程序（如 cert-manager）的依赖。接下来我们就来重点介绍下如何使用这种方式来管理相关证书。

初始化容器

这个初始化容器的主要功能是创建一个自签名的 Webhook 服务证书，并通过 mutate/验证配置将 caBundle 提供给 APIServer。Webhook 服务如何使用该证书（通过 Secret Volumes 或 emptyDir），取决于实际情况。这里我们这个初始化容器将运行一个简单的 Go 二进制文件来执行这些功能。核心代码如下所示：

```
package main

import (
  "bytes"
  cryptorand "crypto/rand"
  "crypto/rsa"
  "crypto/x509"
  "crypto/x509/pkix"
  "encoding/pem"
  "fmt"
  log "github.com/sirupsen/logrus"
  "math/big"
  "os"
```

```
  "time"
)

func main() {
  var caPEM, serverCertPEM, serverPrivKeyPEM *bytes.Buffer
  // CA config
  ca := &x509.Certificate{
    SerialNumber: big.NewInt(2021),
    Subject: pkix.Name{
      Organization: []string{"ydzs.io"},
    },
    NotBefore:             time.Now(),
    NotAfter:              time.Now().AddDate(1, 0, 0),
    IsCA:                  true,
    ExtKeyUsage:           []x509.ExtKeyUsage{x509.ExtKeyUsageClientAuth, x509.ExtKeyUsageServerAuth},
    KeyUsage:              x509.KeyUsageDigitalSignature | x509.KeyUsageCertSign,
    BasicConstraintsValid: true,
  }

  // CA private key
  caPrivKey, err := rsa.GenerateKey(cryptorand.Reader, 4096)
  if err != nil {
    fmt.Println(err)
  }

  // Self signed CA certificate
  caBytes, err := x509.CreateCertificate(cryptorand.Reader, ca, ca, &caPrivKey.PublicKey, caPrivKey)
  if err != nil {
    fmt.Println(err)
  }

  // PEM encode CA cert
  caPEM = new(bytes.Buffer)
  _ = pem.Encode(caPEM, &pem.Block{
    Type:  "CERTIFICATE",
    Bytes: caBytes,
  })

  dnsNames := []string{"admission-registry",
    "admission-registry.default", "admission-registry.default.svc"}
  commonName := "admission-registry.default.svc"

  // server cert config
  cert := &x509.Certificate{
    DNSNames:     dnsNames,
    SerialNumber: big.NewInt(1658),
    Subject: pkix.Name{
      CommonName:   commonName,
      Organization: []string{"ydzs.io"},
    },
    NotBefore:    time.Now(),
    NotAfter:     time.Now().AddDate(1, 0, 0),
    SubjectKeyId: []byte{1, 2, 3, 4, 6},
    ExtKeyUsage:  []x509.ExtKeyUsage{x509.ExtKeyUsageClientAuth, x509.ExtKeyUsageServerAuth},
    KeyUsage:     x509.KeyUsageDigitalSignature,
  }

  // server private key
  serverPrivKey, err := rsa.GenerateKey(cryptorand.Reader, 4096)
  if err != nil {
    fmt.Println(err)
  }

  // sign the server cert
  serverCertBytes, err := x509.CreateCertificate(cryptorand.Reader, cert, ca, &serverPrivKey.PublicKey, caPrivKey)
  if err != nil {
    fmt.Println(err)
  }

  // PEM encode the server cert and key
  serverCertPEM = new(bytes.Buffer)
  _ = pem.Encode(serverCertPEM, &pem.Block{
    Type:  "CERTIFICATE",
    Bytes: serverCertBytes,
```

```
  })

  serverPrivKeyPEM = new(bytes.Buffer)
  _ = pem.Encode(serverPrivKeyPEM, &pem.Block{
    Type:  "RSA PRIVATE KEY",
    Bytes: x509.MarshalPKCS1PrivateKey(serverPrivKey),
  })

  err = os.MkdirAll("/etc/webhook/certs/", 0666)
  if err != nil {
    log.Panic(err)
  }
  err = WriteFile("/etc/webhook/certs/tls.crt", serverCertPEM)
  if err != nil {
    log.Panic(err)
  }

  err = WriteFile("/etc/webhook/certs/tls.key", serverPrivKeyPEM)
  if err != nil {
    log.Panic(err)
  }

}

// WriteFile writes data in the file at the given path
func WriteFile(filepath string, sCert *bytes.Buffer) error {
  f, err := os.Create(filepath)
  if err != nil {
    return err
  }
  defer f.Close()

  _, err = f.Write(sCert.Bytes())
  if err != nil {
    return err
  }
  return nil
}
```

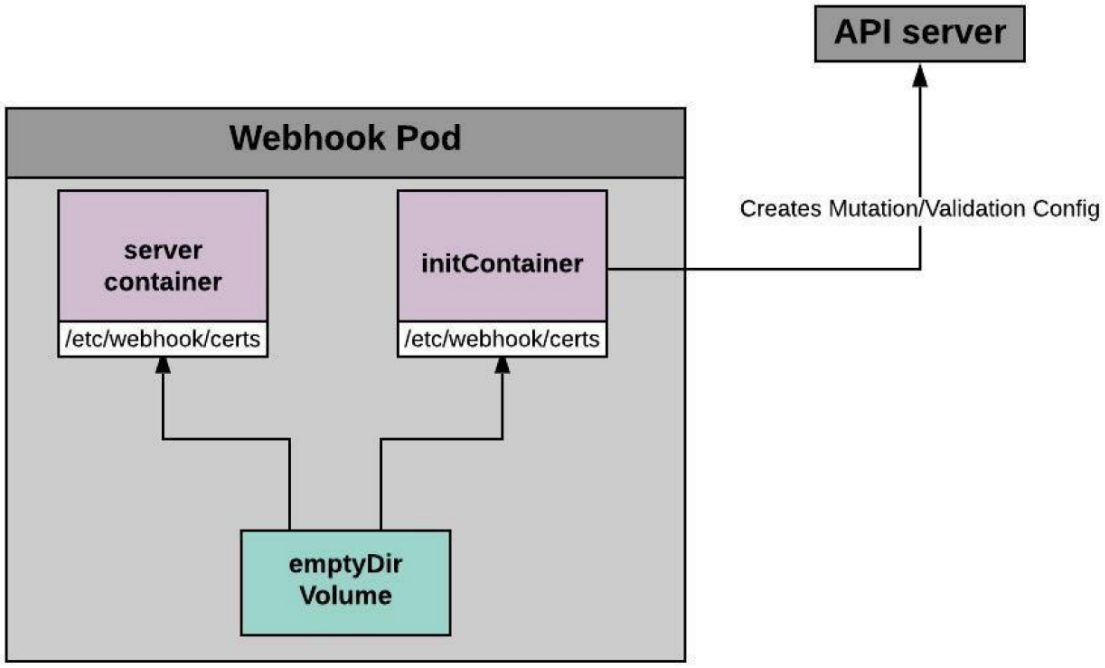
在上面的代码中我们通过生成自签名的 CA 并签署 Webhook 服务证书来提供服务：

- 首先为 CA 创建一个配置 ca
- 为该 CA 创建一个 RSA 私钥 caPrivKey
- 生成一个自签名的 CA、caByte 和 caPEM，在这里，caPEM 是 PEM 编码的 caBytes，将是提供给 APIServer 的 CA_BUNDLE 数据
- 创建 webhook 服务证书的配置，即上面代码中的 cert。该配置中的重要属性是 DNSNames 和 commonName，要注意的是该名称必须是到达 Webhook 服务的完整地址名称
- 然后为 Webhook 服务创建一个 RS 私钥 serverPrivKey
- 使用上面代码中的 ca 和 caPrivKey 创建服务端证书 serverCertBytes
- 然后用 PEM 对 serverPrivKey 和 serverCertBytes 进行编码，这个 serverPrivKeyPEM 和 serverCertPEM 就是 TLS 证书和密钥了，将由 Webhook 服务使用。

到这里我们就可以生成所需的证书，密钥和 CA_BUNDLE 数据了。然后我们将与同一 Pod 中的实际 Webhook 服务容器共享该服务器证书和密钥。

- 一种方法是事先创建一个空的 Secret 资源，通过将该 Secret 作为环境变量传递来创建 Webhook 服务，初始化容器将生成服务器证书和密钥，并用证书和密钥信息来填充该 Secret。此 Secret 将安装到 Webhook 服务容器上，以使用 TLS 来启动 HTTP 服务器。
- 第二种方法（在上面的代码中使用）是使用 Kubernete 的本地 Pod 特定的 emptyDir 卷。该数据卷将在两个容器之间共享，在上面的代码中，我们可以看到 init 容器将这些证书和密钥信息写入特定路径的文件

中，该路径就是其中的一个 emptyDir 卷，并且 Webhook 服务容器将从该路径读取用于 TLS 配置的证书和密钥，并启动 HTTP Webhook 服务器。请参考下图：



Webhook 的 Pod 规范如下所示：

```
spec:
  initContainers:
    - image: <webhook init-image name>
      imagePullPolicy: IfNotPresent
      name: webhook-init
      volumeMounts:
        - mountPath: /etc/webhook/certs
          name: webhook-certs
  containers:
    - image: <webhook server image name>
      imagePullPolicy: IfNotPresent
      name: webhook-server
      volumeMounts:
        - mountPath: /etc/webhook/certs
          name: webhook-certs
          readOnly: true
  volumes:
    - name: webhook-certs
      emptyDir: {}
```

处理 CA Bundle

然后剩下的就只有使用 mutate/验证配置将 CA_BUNDLE 信息提供给 APIServer，这可以通过两种方式完成：

- 使用 init 容器中的 client-go 在现有 `MutatingWebhookConfiguration` 或 `ValidatingWebhookConfiguration` 中来修补 CA_BUNDLE 数据。
- 另一种方式使用配置中的 CA_BUNDLE 数据在 init 容器本身中直接创建 `MutatingWebhookConfiguration` 或 `ValidatingWebhookConfiguration` 即可。

在这里，我们将通过 init 容器来创建配置，通过动态获取某些参数，例如 mutate 配置名称，Webhook 服务名称和 Webhook 命名空间，我们都可以直接从 init 容器的环境变量中来获取这些值：

```
initContainers:
- image: <webhook init-image name>
  imagePullPolicy: IfNotPresent
  name: webhook-init
  volumeMounts:
    - mountPath: /etc/webhook/certs
      name: webhook-certs
  env:
    - name: MUTATE_CONFIG
      value: admission-registry-mutate
    - name: VALIDATE_CONFIG
      value: admission-registry
    - name: WEBHOOK_SERVICE
      value: admission-registry
    - name: WEBHOOK_NAMESPACE
      value:  default
```

为了创建 `MutatingWebhookConfiguration` 或者 `ValidatingWebhookConfiguration` 资源对象，我们将以下代码添加到上面的 init 容器代码中。

```
package main

import (
  "bytes"
  "context"
  "os"

  admissionregistrationv1 "k8s.io/api/admissionregistration/v1"
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
  "k8s.io/client-go/kubernetes"
  "k8s.io/client-go/rest"
)

func initKubeClient() (*kubernetes.Clientset, error) {
  var (
    err error
    config *rest.Config
  )
  if config, err = rest.InClusterConfig(); err != nil {
    return nil, err
  }

  // 创建 Clientset 对象
  clientset, err := kubernetes.NewForConfig(config)
  if err != nil {
    return nil, err
  }
  return clientset, nil
}

func CreateAdmissionConfig(caCert *bytes.Buffer) error {
  var (
    webhookNamespace, _ = os.LookupEnv("WEBHOOK_NAMESPACE")
    mutationCfgName, _  = os.LookupEnv("MUTATE_CONFIG")
    validateCfgName, _ = os.LookupEnv("VALIDATE_CONFIG")
    webhookService, _ = os.LookupEnv("WEBHOOK_SERVICE")
    validatePath, _ = os.LookupEnv("VALIDATE_PATH")
    mutationPath, _ = os.LookupEnv("MUTATE_PATH")
  )

  clientset, err := initKubeClient()
  if err != nil {
    return err
  }

  ctx := context.Background()
```

```
if validateCfgName != "" {
  validateConfig := &admissionregistrationv1.ValidatingWebhookConfiguration{
    ObjectMeta: metav1.ObjectMeta{
      Name: validateCfgName,
    },
    Webhooks: []admissionregistrationv1.ValidatingWebhook{
      {
        Name: "io.ydzs.admission-registry",
        ClientConfig: admissionregistrationv1.WebhookClientConfig{
          CABundle: caCert.Bytes(),
          Service: &admissionregistrationv1.ServiceReference{
            Name: webhookService,
            Namespace: webhookNamespace,
            Path: &validatePath,
          },
        },
        Rules: []admissionregistrationv1.RuleWithOperations{
          {
            Operations: []admissionregistrationv1.OperationType{admissionregistrationv1.Create},
            Rule: admissionregistrationv1.Rule{
              APIGroups:   []string{""},
              APIVersions: []string{"v1"},
              Resources:   []string{"pods"},
            },
          },
        },
        FailurePolicy: func() *admissionregistrationv1.FailurePolicyType{
          pt := admissionregistrationv1.Fail
          return &pt
        }(),
        AdmissionReviewVersions: []string{"v1"},
        SideEffects: func() *admissionregistrationv1.SideEffectClass {
          se := admissionregistrationv1.SideEffectClassNone
          return &se
        }(),
      },
    },
  }

  validateAdmissionClient := clientset.AdmissionregistrationV1().ValidatingWebhookConfigurations()
  _, err := validateAdmissionClient.Get(ctx, validateCfgName, metav1.GetOptions{})
  if err != nil {
    if errors.IsNotFound(err) {
      if _, err = validateAdmissionClient.Create(ctx, validateConfig, metav1.CreateOptions{}); err != nil {
        return err
      }
    } else {
      return err
    }
  } else {
    if _, err = validateAdmissionClient.Update(ctx, validateConfig, metav1.UpdateOptions{}); err != nil {
      return err
    }
  }

}

if mutationCfgName != "" {
  mutateConfig := &admissionregistrationv1.MutatingWebhookConfiguration{
    ObjectMeta: metav1.ObjectMeta{
      Name: mutationCfgName,
    },
    Webhooks: []admissionregistrationv1.MutatingWebhook{{
      Name: "io.ydzs.admission-registry-mutate",
      ClientConfig: admissionregistrationv1.WebhookClientConfig{
        CABundle: caCert.Bytes(), // CA bundle created earlier
        Service: &admissionregistrationv1.ServiceReference{
          Name:      webhookService,
          Namespace: webhookNamespace,
          Path:      &mutationPath,
        },
      },
      Rules: []admissionregistrationv1.RuleWithOperations{{Operations: []admissionregistrationv1.OperationType{
```

```
            admissionregistrationv1.Create},
            Rule: admissionregistrationv1.Rule{
              APIGroups:   []string{"apps", ""},
              APIVersions: []string{"v1"},
              Resources:   []string{"deployments", "services"},
            },
          }},
          FailurePolicy: func() *admissionregistrationv1.FailurePolicyType{
            pt := admissionregistrationv1.Fail
            return &pt
          }(),
          AdmissionReviewVersions: []string{"v1"},
          SideEffects: func() *admissionregistrationv1.SideEffectClass {
            se := admissionregistrationv1.SideEffectClassNone
            return &se
          }(),
        }},
      }

      mutateAdmissionClient := clientset.AdmissionregistrationV1().MutatingWebhookConfigurations()
      _, err := mutateAdmissionClient.Get(ctx, mutationCfgName, metav1.GetOptions{})
      if err != nil {
        if errors.IsNotFound(err) {
          if _, err = mutateAdmissionClient.Create(ctx, mutateConfig, metav1.CreateOptions{}); err != nil {
            return err
          }
        } else {
          return err
        }
      } else {
        if _, err = mutateAdmissionClient.Update(ctx, mutateConfig, metav1.UpdateOptions{}); err != nil {
          return err
        }
      }

    }

    return nil
  }
```

这里首先我们读取环境变量，例如 webhookNamespace，接下来，我们将使用 CA bundle 信息（先前创建）和其他必需信息来定义配置的资源对象结构。最后，我们使用 client-go 来创建配置资源对象。对于 Pod 重新启动或删除的情况，我们可以在 init 容器中添加额外的逻辑，例如首先删除现有配置，然后再仅在创建或更新 CA bundle（如果配置已存在）之前删除它们。

对于证书轮换的情况，对于向服务器容器提供此证书所采用的每种方法，方法将有所不同：

- 如果我们使用的是 emptyDir 卷，则方法将是仅重新启动 Webhook Pod。由于 emptyDir 卷是临时的，并且绑定到 Pod 的生命周期，因此在重新启动时，将生成一个新证书并将其提供给服务器容器。如果已经存在配置，则将在配置中添加新的 CA bundle。
- 如果我们正在使用 Secret 卷，则在重新启动 Webhook Pod 时，可以检查 Secret 中现有证书的有效期，以决定是将现有证书用于服务器还是创建新证书。

在这两种情况下，都需要重新启动 Webhook Pod 才能触发证书轮换/续订过程。何时需要重新启动 Webhook 容器以及如何重新启动 Webhook 容器，将取决于实际情况。可能的几种方法可以使用Cronjob、controller 等来实现。

到这里我们的自定义 Webhook 已注册，APIServer 可以通过 config 读取到 CA bundle 信息，并且 Webhook 服务已准备好按照 configs 中定义的规则处理 mutate/验证请求。

部署

最后将上面的证书生成应用打包成一个 Docker 镜像，将上节课部署的 Webhook 服务删除，重新使用如下所示的资源对象进行部署即可：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admission-registry-sa
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: admission-registry-role
rules:
- verbs: ["*"]
  resources: ["validatingwebhookconfigurations", "mutatingwebhookconfigurations"]
  apiGroups: ["admissionregistration.k8s.io"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admission-registry-rolebinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: admission-registry-role
subjects:
- kind: ServiceAccount
  name: admission-registry-sa
  namespace: default
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: admission-registry
  labels:
    app: admission-registry
spec:
  selector:
    matchLabels:
      app: admission-registry
  template:
    metadata:
      labels:
        app: admission-registry
    spec:
      serviceAccountName: admission-registry-sa
      initContainers:
      - image: cnych/admission-registry-tls:v0.0.3
        imagePullPolicy: IfNotPresent
        name: webhook-init
        env:
        - name: WEBHOOK_NAMESPACE
          value: default
        - name: MUTATE_CONFIG
          value: admission-registry-mutate
        - name: VALIDATE_CONFIG
          value: admission-registry
        - name: WEBHOOK_SERVICE
          value: admission-registry
        - name: VALIDATE_PATH
          value: /validate
        - name: MUTATE_PATH
          value: /mutate
        volumeMounts:
          - mountPath: /etc/webhook/certs
            name: webhook-certs
      containers:
      - name: webhook
        image: cnych/admission-registry:v0.1.4
        imagePullPolicy: IfNotPresent
        env:
        - name: WHITELIST_REGISTRIES
          value: "docker.io,gcr.io"
        ports:
        - containerPort: 443
        volumeMounts:
```

```
        - name: webhook-certs
          mountPath: /etc/webhook/certs
          readOnly: true
      volumes:
        - name: webhook-certs
          emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: admission-registry
  labels:
    app: admission-registry
spec:
  ports:
    - port: 443
      targetPort: 443
  selector:
    app: admission-registry
```

现在我们就不需要自己手动去创建包含证书的 Secret 资源对象了，也不需要手动去替换准入控制器配置对象中的 CA bundle 信息了，这些都将通过 Init 初始化容器来帮我们自动完成。

由于初始化容器需要访问 `MutatingWebhookConfiguration` 和 `ValidatingWebhookConfiguration` 这两个资源对象，所以我们需要声明对应的 RBAC 权限。创建完成后的资源对象如下所示：

```
$ kubectl get pods -l app=admission-registry
NAME                                   READY   STATUS    RESTARTS   AGE
admission-registry-64f6b46cdc-vqbrl   1/1     Running   0          96s
$ kubectl exec -it admission-registry-64f6b46cdc-vqbrl -- ls /etc/webhook/certs
tls.crt  tls.key
$ kubectl get validatingwebhookconfiguration
NAME                             WEBHOOKS   AGE
admission-registry               1          20s
➜  admission-registry git:(main) ✗ kubectl get mutatingwebhookconfigurations
NAME                         WEBHOOKS   AGE
admission-registry-mutate   1          24s
```

然后同样再去测试一次即可，到这里我们就完成了使用初始化容器来管理 Admission Webhook 的 TLS 证书的功能。