c't Teil 5: Container-Sicherheitsbegehung

Wenn der erste Cluster läuft und die Kubectl-Befehle leicht von der Hand gehen, möchte man als Kubernetes-Einsteiger am liebsten mit den ersten produktiven Anwendungen beginnen. Doch vorher sollte man sich etwas Zeit nehmen für die Sicherheit von Cluster und Anwendern.

Von Jan Mahn

c't 3/2023 S. 154

c't kompakt

- Webseiten will man heute per HTTPS veröffentlichen. Mit Traefik in Kubernetes ist das kein Problem.
- Damit Pods im Cluster nur die Dienste erreichen, die Sie freigeben, gibt es NetworkPolicies.
- Sobald mehrere Nutzer am Cluster arbeiten, sollten Sie Rollen vergeben und sich um Rechteverwaltung kümmern.

Geschafft – in den ersten vier Teilen dieser Reihe haben wir Sie auf dem Weg vom Docker-Nutzer zum Kubernetes-Cluster-Betreiber begleitet und als Beispiel Schritt für Schritt eine WordPress-Instanz zusammengebaut [1]. Bevor Sie das Wissen auf eigene Projekte anwenden, sollten Sie aber noch ein paar Sicherheitskonzepte kennenlernen und gleich von Anfang an einsetzen – das ist immer erfolgversprechender, als Security nachträglich an eine fertige Anwendung dranzubasteln. Auf den folgenden Seiten lernen Sie, wie Sie Ihren Cluster in drei Schritten sicherer machen: mit Transportverschlüsselung, Netzwerkregeln für Pods und Zugriffsberechtigungen für Admins.

Am Ende des vierten Teils dieser Reihe meldete sich eine WordPress-Instanz auf Port 80 per HTTP. Vor 15 Jahren hätte man damit noch Benutzer und Admins begeistern können, heute fehlt ein entscheidendes Detail: HTTPS mit einem gültigen Zertifikat, das von einer vertrauenswürdigen Stammzertifizierungsstelle stammt und dem die Browser vertrauen. Das muss man heute nicht mehr kaufen, Let’s Encrypt liefert es kostenlos und der HTTP-Router Traefik, den Sie im Laufe der Reihe schon installiert haben, beschafft Zertifikate von diesem Anbieter und verlängert sie automatisch. Die erste Aufgabe, um diese Beschaffung einzurichten, hat nichts mit Kubernetes zu tun. Damit Sie ein Zertifikat bekommen können, müssen Sie einen A- und optional einen AAAA-DNS-Eintrag für eine Domain oder eine Subdomain setzen, der auf eine beliebige, externe IP-Adresse Ihres Clusters verweist. Erledigen Sie diese Aufgabe am besten mit etwas zeitlichem

Abstand, damit sich der Eintrag in allen DNS-Caches herumspricht. Danach können Sie sich an die Vorbereitungen im Cluster machen.

Traefik mit Speicher

Wenn Sie den Beschreibungen dieser Kubernetes-Reihe gefolgt sind, haben Sie Traefik im dritten Teil mit folgenden Zeilen über den Kubernetes-Paketmanager Helm installiert:

```
helm repo add traefik https://helm.traefik.io/traefik
helm repo update
helm install traefik traefik/traefik
```

Helm legt ein Deployment mit dem Namen `traefik` aus dem Chart `traefik/traefik` an. Diese Installation müssen Sie für die automatische Zertifikatsbeschaffung etwas erweitern. Traefik braucht persistenten Speicherplatz für die Zertifikate und die Zertifikatsbeschaffung über Let’s Encrypt braucht eine minimale Konfiguration. Immer, wenn es in einer Installation per Helm etwas einzurichten gibt, erledigt man das über eine sogenannte Values-Datei. Das ist ein Stück YAML, das man Helm mit einem Install- oder Upgrade-Befehl unterschiebt. Wie das YAML aussieht, definieren die Entwickler des jeweiligen Helm-Pakets – man befüllt damit Variablen, die im Helm-Chart verwendet werden. Wie das im Detail funktioniert, müssen Sie erst durchdringen, wenn Sie eigene Anwendungen in Helm-Charts verpacken wollen. Als Anwender eines Charts reicht es, die benötigten Konfigurationen aus der Dokumentation in eine neue Datei zu legen und anzupassen. Für Traefik mit TLS legen Sie auf Ihrer lokalen Maschine eine Datei traefik-values.yaml an (den Namen können Sie frei vergeben). In die Datei kommen folgende Zeilen:

```
persistence:
  enabled: true
  name: data
  accessMode: ReadWriteOnce
  size: 128Mi
  path: /data
  annotations: {}

certResolvers:
  letsencrypt:
    email: <Ihre Mailadresse>
    tlsChallenge: true
    httpChallenge:
      entryPoint: "web"
    storage: /data/acme.json

ports:
  websecure:
    tls:
      certResolver: "letsencrypt"
  web:
    redirectTo: websecure
```

Wie auch in den ersten Teilen dieser Reihe finden Sie alle Inhalte zum Download in einem GitHub-Repository (siehe ct.de/yqsq), abtippen müssen Sie also nichts. Der erste Abschnitt `persistence:` aktiviert persistenten Speicherplatz. In der Folge legt Helm einen PersistentVolumeClaim an und bindet ihn an den Traefik-Container. 128 MByte reichen für die Zertifikate und Metadaten, die Traefik anlegt, absolut aus.

Der zweite Abschnitt konfiguriert die ACME-Funktion von Traefik. Über diesen Standard bezieht die Software das Zertifikat bei Let's Encrypt und legt dafür auf Port 80 (`entryPoint: "web"`) eine HTTP-Challenge ab, mit der Let's Encrypt prüft, ob Sie diese Domain kontrollieren. Ändern müssen Sie in diesem Beispiel nur Ihre Mailadresse. Sie wird nicht im Zertifikat veröffentlicht, Let's Encrypt nutzt sie nur, um Sie zu warnen, wenn das Zertifikat ausläuft – das sollte aber nicht passieren, wenn Ihr Server läuft: Traefik beschafft kommentarlos neue Zertifikate.

Der dritte Abschnitt `ports:` aktiviert schließlich den zuvor definierten Zertifikatsdienst namens `letsencrypt` für den HTTPS-Port, den Traefik `websecure` nennt. Der Endpunkt `web` wird mit `redirectTo` angewiesen, sämtlichen Verkehr auf seinen HTTPS-Kollegen umzuleiten. Ihre Nutzer surfen also immer mit TLS.

Liegt die Datei traefik-values.yaml auf Ihrer lokalen Maschine mit installiertem Helm bereit, navigieren Sie auf der Kommandozeile in den Ordner mit dieser Datei und aktualisieren Sie das Helm-Deployment mit diesen Werten:

```
helm upgrade -f traefik-values.yaml traefik traefik/traefik
```

Wenig später sollten Sie mit `kubectl get pvc` einen PersistentVolumeClaim für Traefik sehen. Damit sind die Voraussetzungen geschaffen, um eine IngressRoute auf HTTPS umzustellen. Als Beispiel dient die Wordpress-Route aus Teil 3 und 4 der Reihe. Nur zwei Änderungen an der YAML-Datei sind nötig:

```
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: wordpress-ingress
  namespace: frontend
spec:
  entryPoints:
    - websecure # vorher: web
  routes:
      # vorher: match: PathPrefix(`/`)
    - match: Host(`www.example.org`)
      kind: Rule
      services:
        - name: wp-external
          port: 80
```

Die erste Änderung betrifft den `entryPoint`. Statt auf Port 80 soll die Seite künftig auf Port 443 antworten. Damit der Zertifikatsdienst von Traefik weiß, für welche Domains er ein Zertifikat ordern soll, brauchen Sie immer eine Regel vom Typ `Host` mit dem Namen. Das muss nicht nur eine sein:

Mit dem Operator `||` können Sie auch mehrere Host-Einträge angeben (zum Beispiel www.example.org und example.org).

Um Ihre WordPress-Instanz umzustellen, ändern Sie die Zeilen in der YAML-Datei und setzen einen Kubectl-Apply-Befehl ab. Es dauert nur rund 60 Sekunden, bis Sie Ihre Seite mit vorangestelltem https:// erreichen. Wenn Sie der Browser auch nach Minuten noch mit einer Zertifikatswarnung begrüßt, klemmt irgendetwas. Besorgen Sie sich mit `kubectl get pods` den Namen des Traefik-Pods und schauen mit `kubectl logs`, was Traefik daran hindert, ein Zertifikat zu besorgen.

Verkehrssteuerung

Das nächste Problem, das Sie auf dem Weg zu einem sichereren Cluster angehen sollten, ist der ungezügelte Netzwerkverkehr von Pods. Unternimmt man nichts, kann jeder Pod jeden Service im Cluster ansprechen, auch über Namespace-Grenzen hinweg. Außerdem kommt er ungehemmt ins Internet. In der Theorie ist das kein Problem, wenn man davon ausgeht, dass man als Administrator alle Pods selbst kontrolliert und weiß, was darin passiert. Doch mit dieser Einstellung macht man es Angreifern wahnsinnig leicht. Sobald die es schaffen, ihren Schadcode auf einem einzigen Pod auszuführen, können sie mühelos weiteren Code aus dem Internet nachladen und sich im Cluster oder gleich im ganzen Netzwerk ausbreiten. Das muss nicht sein.

Die allermeisten Pods brauchen keinen Weg ins Internet; bei Containern kann und sollte man da noch strenger sein als bei klassisch installierter Software. Bei letzterer gibt es manchmal noch das Szenario, dass sie regelmäßig einen Herstellerserver nach neuen Updates fragen müssen. Bei Containern fällt auch das weg, weil Software im Container sich nicht selbst aktualisieren soll. Das funktioniert in Containern anders: Gibt es Updates, wird der Container durch einen mit frischem Image ersetzt.

Nicht zuletzt durch die schwere Sicherheitslücke in der Java-Logging-Bibliothek Log4j haben viele Serverbetreiber erkannt, dass sie in der Vergangenheit zu großzügig mit dem Recht umgegangen sind, Kontakt mit dem Internet aufzunehmen. Der Log4Shell-Angriff war darauf angewiesen, dass Log4J Code aus dem Internet nachlud.

Schränken Sie den Verkehr rigoros ein, damit es so weit nicht kommt. Dafür kennt Kubernetes das Konzept der NetworkPolicies, die wie Cluster-interne Firewallregeln funktionieren. Wie bei anderen Firewalls gibt es zwei Arten von Regeln; solche für eingehenden (Ingress) und solche für ausgehenden Verkehr (Egress). Eingerichtet werden die Policies, wie alles in Kubernetes, mit einer YAML-Definition, die wie die folgende aussieht:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: example-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
```

```
    example.org/network-rule: strict
  ingress:
    - from:
        - podSelector: {}
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: database
      ports:
        - port: 3306
    - to:
        - namespaceSelector: {}
          podSelector:
            matchLabels:
              k8s-app: kube-dns
      ports:
        - port: 53
          protocol: UDP
    - to:
        - ipBlock:
            cidr: 12.34.56.0/24
      ports:
        - port: 443
```

Wie jedes Objekt bekommt eine NetworkPolicy im Abschnitt `metadata:` einen Namen. Unter `spec:` folgt der Inhalt der Regel. Zunächst legt man fest, für welche Pods sie gelten soll. Zum Einsatz kommt ein `podSelector`, der zum Beispiel auch bei Services genutzt wird. Kubernetes sucht damit nach allen Pods, an die man ein bestimmtes Label angeheftet hat. Labels können Sie als Administrator grundsätzlich recht frei vergeben. Es ist jedoch ausdrücklich empfohlen, selbst ausgedachte Labels mit einem eigenen Präfix in Form der eigenen Domain zu kennzeichnen. Dann kann man sichergehen, dass sie nicht mit Kubernetes-eigenen Labels kollidieren (auch nicht mit einem, das sich die Kubernetes-Entwickler in Zukunft noch ausdenken). Im Beispiel sollen alle Pods eine Regel bekommen, die mit dem Label `example.org/network-rule` und dem Wert `strict` markiert sind.

Es folgt eine recht kurze Ingress-Regel mit einem leeren Objekt, das Einschränkungen enthalten könnte, im Beispiel aber leer ist. Die Folge: Alle Pods im Cluster dürfen Pods mit dieser NetworkPolicy kontaktieren. Möchte man dagegen, dass kein anderer Pod auf einen Pod zugreifen darf, muss die Liste der Ingress-Regeln leer sein:

```
ingress: []
```

Ausgehende Regeln gibt es gleich drei. Die erste erlaubt Verkehr zu Pods mit dem Label `app: database`. Ohne die zweite Regel funktioniert sie aber nicht – ein typischer Fehler, mit dem sich NetworkPolicy-Einsteiger teils lange herumärgern. Die zweite Regel erlaubt dem Pod, den Kubernetes-internen DNS-Server auf Port 53 zu nutzen, um die Adresse der Datenbank aufzulösen. Ohne DNS geht nicht viel. Die dritte Egress-Regel ist ein Beispiel für externen Verkehr. Der Pod darf damit alle IP-Adressen zwischen 12.34.56.1 und 12.34.56.254 auf Port 443 ansprechen.

> Mit dem NetworkPolicy-Editor von Cilium hat man passende Regeln ohne Tipperei schnell zusa
> Mit dem NetworkPolicy-Editor von Cilium hat man passende Regeln ohne Tipperei schnell zusammengebaut.

Die Syntax der NetworkPolicies ist zu Beginn etwas verwirrend und schnell hat man sich verkonfiguriert, was entweder zu unnötig laxen Regeln oder gescheiterten Verbindungen führt.

Sehr empfehlenswert (nicht nur für Einsteiger) ist der Onlinegenerator von Cilium (editor.cilium.io), in dem man seine Regeln im Browser zusammenklickt und an roten und grünen Pfeilen sieht, was erlaubt und verboten ist. Ganz uneigennützig stellt Cilium den Editor nicht bereit: Cilium ist ein Open-Source-Projekt, das eine zusätzliche Netzwerkschicht für Kubernetes entwickelt, die unter anderem noch genauer filtern kann. Und so weist der Editor an einigen Stellen darauf hin, was mit Kubernetes-Bordmitteln (noch) nicht möglich ist und wie übersichtlich die Cilium-Syntax im Vergleich aussieht. Cilium ist reizvoll für große Organisationen, aber kein Projekt für Einsteiger.

In NetworkPolicies kann man viel Zeit investieren. Ein sicherer und effizienter Weg: Man verbietet zunächst mal alles und gibt dann gezielt die wenigen Ports und Verbindungen frei, die wirklich nötig sind. Wenn Sie so vorgehen, erfinden Sie passende NetworkPolicies bald genauso routiniert, wie Sie Deployments, Services und IngressRoutes anlegen. Ausgangspunkt für das Absicherungsprojekt ist eine strikte Regel, die alle Schotten dicht macht:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: default-deny
  namespace: default
spec:
  policyTypes:

  - Ingress
  - Egress
  podSelector: {}
  ingress: []
  egress: []
```

Weil eine NetworkPolicy immer zu einem Namespace gehört, braucht man diese Regel für jeden Namespace. Ist das erledigt, geht erst mal nichts mehr – damit das WordPress-Beispiel wieder läuft, brauchen Sie Regeln, die WordPress Kontakte zur Datenbank gestatten und Traefik Kontakt zu WordPress. Wenn Sie diese Herausforderung als Übungsaufgabe nutzen wollen, ein Tipp: Zum erfolgreichen Verbindungsaufbau gehören `ingress` und `egress`. Eine mögliche Lösung finden Sie im GitHub-Repository zum Artikel (siehe ct.de/yqsq) – es führen aber viele Wege zum Ziel.

Rechte verwalten

Beim Einrichten des Clusters im ersten Teil der Reihe hat K3S stumm einen Benutzer angelegt, der mit vollen Rechten ausgestattet ist. Er meldet sich mit einem Schlüsselpaar aus öffentlichem und privatem Schlüssel an – mit diesen Feinheiten mussten Sie sich bislang nicht beschäftigen, weil K3S

die Erzeugung erledigt und alles in eine fertige YAML-Datei gelegt hat, die Sie als ~/.kube/config auf die lokale Maschine kopiert haben.

Solange Sie allein sind und auf dieses Schlüsselpaar aufpassen, ist das nicht grundsätzlich unsicher – auf Ihrer lokalen Maschine liegen ja oft auch SSH-Schlüssel für diverse Server. Problematisch wird der Kubernetes-Superuser aber spätestens, wenn Sie Ihren Cluster mit Kollegen teilen wollen. Denen wollen Sie nicht unbedingt Vollzugriff und Ihren einzigen Schlüssel geben. Hier kommt die sehr granulare Rollenverwaltung von Kubernetes zum Zug. Um beim WordPress-Beispiel zu bleiben, könnten sich die Frontend-Entwickler einen Zugang wünschen, um im Namespace Frontend Änderungen an Pods vornehmen zu können. Sie wollen vielleicht mal das Image austauschen. Mit der IngressRoute und anderen Ressourcen haben sie aber nichts zu tun.

Dafür brauchen Sie zunächst zwei neue Ressourcen: eine Role und ein RoleBinding. Erstere definiert, was ein Rolleninhaber genau darf. Per RoleBinding wird die Rolle an einen Nutzer oder eine Nutzergruppe gebunden – den Nutzer mit dem Nutzernamen `frontend-guy` erzeugen Sie erst danach.

Folgender YAML-Block definiert die Rolle `frontend-dev` im Namespace `frontend` und berechtigt dazu, Pods zu erstellen und zu löschen:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: frontend
  name: frontend-dev
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs:
      [
        "create",
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "update",
        "watch",
      ]
```

Das Beispiel zeigt, wie granular Sie Berechtigungen steuern dürfen. `apiGroups` enthält nur einen leeren String, damit gilt die Regel für Core-Kubernetes-Objekte, zu denen Pods gehören. Die `apiGroup` eines Kubernetes-Objekts erkennen Sie mit einem Blick auf die Angabe `apiVersion` in der YAML-Datei. Die oben vorgestellte NetworkPolicy gehört beispielsweise zur Gruppe `networking.k8s.io`.

Unter `resources:` geben Sie eine Liste mit Objekten an, mit denen interagiert werden darf. Die Frontend-Kollegen im Beispiel müssen sich mit Pods begnügen. Dafür dürfen sie mit Pods in ihrem Namespace alles anstellen – freigegeben sind alle `verbs`, die Kubernetes kennt. Diese Liste dient

hier nur der Anschauung, anstatt sie auszuschreiben, würde man in der Praxis die Wildcard `[*]` nutzen. Per Wildcard könnten Sie zum Beispiel auch eine Rolle bauen, die in einem Namespace alle Ressourcen sehen darf.

Damit der Benutzer `frontend-guy` diese Rolle bekommt, brauchen Sie noch eine Zuweisung:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: frontend-dev
  namespace: frontend
subjects:
  - kind: User
    name: frontend-guy
    apiGroup: rbac.authorization.k8s.io
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: frontend-dev
```

In diesem Objekt werden Benutzer und Rolle miteinander verknüpft. Legen Sie eine Datei mit den beiden Objekten an und bringen sie per Kubectl-Apply in den Cluster.

Benutzer backen

Zum Erfolg fehlt nur noch der Benutzer `frontend-guy` selbst – doch einen Befehl wie `kubectl get users` oder eine User-Ressource, für die Sie eine YAML-Datei anlegen könnten, sucht man in der Kubernetes-Dokumentation vergeblich. So etwas kennt Kubernetes nicht, weil das System überhaupt keine Benutzer in seiner Datenhaltung speichert. Stattdessen arbeitet das Kubernetes-API nur mit Zertifikaten: Zum „Anlegen“ eines Nutzers erzeugen Sie lokal ein Zertifikat mit einem öffentlichen und privaten Schlüssel und schreiben den Benutzernamen hinein. Dieses Zertifikat lassen Sie von der Zertifizierungsstelle von Kubernetes signieren. Diese Zertifizierungsstelle bringt Kubernetes immer mit und nutzt sie intern auch für andere Aufgaben, meist bekommen Sie davon nicht viel mit.

Mithilfe des privaten Schlüssels und des Zertifikats können Sie sich fortan ausweisen. Auch wenn der Benutzername nicht im Cluster gespeichert wurde, klappen Authentifizierung (Anmeldung) und Autorisierung (Rechtevergabe). Das Kubernetes-API weiß: „Ich habe dieses Zertifikat signiert, also kann ich den Angaben darin trauen“.

Das Prinzip leuchtet ein, wenn man es einmal durchgespielt hat. Wenn Sie die Befehle nicht abtippen wollen, finden Sie sie ebenfalls im Repository. Die Reise beginnt auf einer lokalen Maschine mit dem Erzeugen eines RSA-Schlüsselpaars. Dafür kommt das klassische Werkzeug `openssl` zum Einsatz. Es funktioniert mittlerweile auch unter Windows, wir empfehlen Windows-Nutzern dennoch, die folgenden Schritte unter Linux oder im WSL nachzuspielen. Alle Dateinamen im folgenden Beispiel können Sie frei vergeben:

```
openssl genrsa -out user.pem
```

Damit liegt das Schlüsselpaar bereit, daraus wird jetzt eine Zertifikatsbestellung (Certificate Signing Request, CSR):

```
openssl req -new -key user.pem -out user.csr -subj "/CN=frontend-guy"
```

In dieser Zeile (und nirgends sonst) wird der Nutzername in der von LDAP bekannten Syntax mit vorangestelltem `CN=` (für Common Name) festgelegt. Heraus kommt eine Datei namens user.csr, die Sie direkt in Base64 wandeln müssen:

```
cat user.csr | base64
```

Die erzeugte Zeichenkette gehört in eine YAML-Datei, die einen CertificateSigningRequest für die Kubernetes-Zertifizierungsstelle definiert. Der Code sieht folgendermaßen aus:

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: csr-frontend-guy
spec:
  groups:
  - system:authenticated
  request: <Base64-String>
  signerName: kubernetes.io/kube-apiserver-client
  expirationSeconds: 157680000
  usages:
  - digital signature
  - key encipherment
  - client auth
```

Den Namen des Objekts dürfen Sie frei wählen und auch die `expirationSeconds` können Sie festlegen. Das Beispiel-Zertifikat gilt knapp 5 Jahre. Bereiten Sie diesen YAML-Schnipsel vor und bringen ihn in den Cluster:

```
kubectl certificate approve csr-frontentd-guy
```

Weil Sie ja momentan mit dem Superuser arbeiten, dürfen Sie Ihre eigene Anfrage direkt genehmigen. Im Cluster liegt jetzt ein signiertes Zertifikat, interessant ist dessen öffentlicher Schlüssel. Den bekommen Sie per Kubectl-Befehl:

```
kubectl get csr csr-frontend-guy  -o jsonpath='{.status.certificate}'
```

Der zugehörige private Schlüssel hat die ganze Zeit auf der lokalen Platte auf seinen Einsatz gewartet – `openssl` hat ihn zu Beginn in die Datei user.pem gelegt, er hat den Computer aber nie verlassen. Das soll auch so bleiben. Auch ihn müssen Sie sich einmal als Base64-String anzeigen:

```
cat user.pem | base64
```

Jetzt haben Sie alle wichtigen Bausteine zusammen, mit denen sich der Frontend-Entwickler namens `frontend-guy` anmelden kann. Er muss beide Base64-Zeichenketten in seine Datei ~/.kube/config unterhalb von `users:` einbauen, etwa so:

```
users:
  frontend-guy:
    client-certificate-data: <Pub Key>
    client-key-data: <Private Key>
```

Wenn Sie ausprobieren wollen, was der Beispielnutzer `frontend-guy` alles darf, bauen Sie diesen Block zusätzlich in Ihre Konfiguration ein (ohne die Zugangsdaten für Ihren Superuser zu löschen) und verweisen im Kontext auf diesen Nutzer. Der Aufruf `kubectl get pods` sollte eine Fehlermeldung auslösen, nur die Abfrage der Pods im Namespace `frontend` darf zum Erfolg führen:

```
  kubectl get pods -n frontend
```

Wenn Sie Administrator in einer größeren Organisation sind, wollen Sie die Schritte sicher automatisieren, um zügiger Zertifikate für das Team auszustellen. Die Schlüsselerzeugung mit `openssl` sollte bestenfalls auf der Entwicklermaschine des jeweiligen Nutzers passieren – den privaten Key müssen Sie als Administrator nicht kennen. Das ist der Reiz von asymmetrischer Kryptografie.

Zwischenfazit

Mit TLS und gültigem Zertifikat haben Sie Ihren Besuchern einen sicheren Weg zu Ihren Diensten bereitet, mit NetworkPolicies die Pods im Cluster voneinander abgeschottet und mit einem beschnittenen Account Ihre Kollegen gezielt berechtigt. Damit ist es um die Sicherheit im Cluster deutlich besser bestellt, der Kubernetes-Werkzeugkasten ist aber noch längst nicht ausgeschöpft und es gibt noch andere Sicherheitsmechanismen zu entdecken – eine Aufgabe für Fortgeschrittene ist beispielsweise die Anbindung von SELinux. Bevor Sie sich an diese Baustelle machen, ist es jetzt aber Zeit für den vergnüglichen Teil: Der Cluster ist bereit für richtige Anwendungen mit Speicherplatz und TLS. (*jam@ct.de*)

1. Literatur

2. Jan Mahn, Containerladeoffizier, Auf dem Lernpfad zum Kubernetes-Kenner, Teil 4, c’t 26/2022, S. 130

Beispiele und Dokumentation: ct.de/yqsq

Revision #1
Created 2026-09-26 13:20:53 UTC by ralf
Updated 2026-09-26 13:20:53 UTC by ralf