Índice

Sessão 6: Segurança no Kubernetes

1) Visualizando certificados digitais

1. Remova os objetos criados na atividade anterior com os comandos que se seguem.

```
# kubectl delete pod,rs,deploy,hpa --all
```

```
# kubectl delete svc -l provider!=kubernetes
```

2. Qual o certificado x509 padrão utilizado pelo `kube-apiserver` para servir requisições HTTPS? E qual a chave privada correspondente?

▼ *Visualizar resposta*

Ambas as informações constam no comando de invocação do `kube-apiserver`. Este pode ser visto de duas formas: primeiro, descrevendo o pod:

```
# kubectl -n kube-system describe pod kube-apiserver-s2-master-1 | grep 'tls-
cert-file'
      --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
```

Ou, segundo, visualizando o arquivo de definição do pod estático:

```
# cat /etc/kubernetes/manifests/kube-apiserver.yaml | grep 'tls-private-key-file'
    - --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
```

3. Agora, identifique o certificado, chave privada e certificado de CA (*Certificate Authority*, ou autoridade certificadora) usados para autenticar o `kube-apiserver` como um cliente do servidor `etcd`.

   ▼ *Visualizar resposta*

   De igual forma, basta consultar as opções de invocação do `kube-apiserver`, pesquisando pelas entradas relevantes.

   ```
   # kubectl -n kube-system describe pod kube-apiserver-s2-master-1 | grep 'etcd'
         --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
         --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
         --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
         --etcd-servers=https://127.0.0.1:2379
   ```

4. Aponte o par de chaves pública/privada utilizadas para autenticar o `kube-apiserver` junto aos `kubelets` dos diversos *nodes* do *cluster*.

   ▼ *Visualizar resposta*

   Novamente, é bastante simples obter a informação solicitada:

   ```
   # kubectl -n kube-system describe pod kube-apiserver-s2-master-1 | grep 'kubelet-
   client'
         --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet
   -client.crt
         --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
   ```

5. Já descobrimos quais os certificados e chaves usados para autenticar o `kube-apiserver` como um cliente do servidor `etcd`.

   Agora, indique o certificado, chave privada e CA utilizados **pelo** servidor `etcd` para disponibilizar seus serviços a clientes (como o `kube-apiserver`, por exemplo).

   ▼ *Visualizar resposta*

   As informações solicitadas encontram-se entre as opções de invocação do *daemon* `etcd` dentro do pod `etcd-s2-master-1`. Vamos visualizá-las, uma a uma:

   ```
   # kubectl -n kube-system describe pod etcd-s2-master-1 | grep '\-\-cert-file='
         --cert-file=/etc/kubernetes/pki/etcd/server.crt
   ```

   ```
   # kubectl -n kube-system describe pod etcd-s2-master-1 | grep '\-\-key-file='
   ```

```
      --key-file=/etc/kubernetes/pki/etcd/server.key
```

```
# kubectl -n kube-system describe pod etcd-s2-master-1 | grep '\-\-trusted-ca-
file='
      --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
```

6. Vamos agora investigar detalhes sobre esses certificados. Indique, para o certificado usado pelo `kube-apiserver`, as seguintes informações:
   - *Common Name* (CN)
   - CA emissora
   - Algoritmo utilizado e tamanho da chave
   - Nomes alternativos configurados para o serviço
   - Validade do certificado (início e final)

   ▼ *Visualizar resposta*

   Podemos visualizar informações sobre um certificado no terminal usando o comando `openssl x509`. Em conjunção com o `grep`, é trivial selecionar apenas os campos objetivados por cada um dos itens da lista acima. Primeiro, vejamos o CN:

   ```
   # openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout | grep
   'Subject: CN'
          Subject: CN = kube-apiserver
   ```

   A seguir, o *issuer* (ou CA emissora), que também possui a *string* CN em sua linha:

   ```
   # openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout | grep 'Issuer:
   CN'
          Issuer: CN = kubernetes
   ```

   Agora, o algoritmo:

   ```
   # openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout | grep
   'Signature Algorithm' | sed 's/^[[:space:]]*//g' | uniq
   Signature Algorithm: sha256WithRSAEncryption
   ```

   O tamanho da chave fica destacado na linha `RSA Public-Key`:

   ```
   # openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout | grep 'RSA
   Public-Key'
                  RSA Public-Key: (2048 bit)
   ```

E quanto aos nomes alternativos do serviço? Vejamos:

```
# openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout | grep
'Alternative Name' -A1
            X509v3 Subject Alternative Name:
                DNS:kubernetes, DNS:kubernetes.default,
DNS:kubernetes.default.svc, DNS:kubernetes.default.svc.cluster.local, DNS:s2-
master-1, IP Address:10.96.0.1, IP Address:192.168.68.20
```

Finalmente, vamos à validade:

```
# openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout | grep
'Validity' -A2
        Validity
            Not Before: Mar 23 12:20:15 2022 GMT
            Not After : Mar 23 12:20:16 2023 GMT
```

Note que a validade do certificado é de um ano, fato confirmado ao consultarmos a data e hora correntes no servidor.

```
# date
Sat 26 Mar 2022 02:02:08 PM UTC
```

7. Compare a validade do certificado analisado no passo anterior com o da CA utilizada para autenticar clientes do `kube-apiserver`. Há alguma diferença?

   ▼ *Visualizar resposta*

   Primeiro, vamos descobrir qual é esse certificado:

   ```
   # kubectl -n kube-system describe pod kube-apiserver-s2-master-1 | grep '\-\-
   client-ca-file'
         --client-ca-file=/etc/kubernetes/pki/ca.crt
   ```

   Ao verificar sua validade, constatamos que ela é significativamente superior: dez anos!

   ```
   # openssl x509 -in /etc/kubernetes/pki/ca.crt -text -noout | grep 'Validity' -A2
           Validity
               Not Before: Mar 23 12:20:15 2022 GMT
               Not After : Mar 20 12:20:15 2032 GMT
   ```

8. Antes de prosseguir, execute o comando abaixo:

   ```
   # lab-6.1.1
   ```

9. Tente executar qualquer comando `kubectl`. O que ocorre?

   ▼ *Visualizar resposta*

   Vejamos:

   ```
   # kubectl get pod
   The connection to the server 192.168.68.20:6443 was refused - did you specify the
   right host or port?
   ```

   Não funcionou... por que será?

10. Aparentemente, o problema está afetando o pod `kube-apiserver`. Investigue sua causa-raiz e solucione-o.

    ▼ *Visualizar resposta*

    Note que, ao tentarmos visualizarmos eventos do pod via `kubectl`, encontramos um erro. Da mesma forma que no passo anterior, a indisponibilidade do `kube-apiserver` impede que tomemos o caminho típico para solucionar esse problema.

    ```
    # kubectl -n kube-system describe pod kube-apiserver-s2-master-1
    The connection to the server 192.168.68.20:6443 was refused - did you specify the
    right host or port?
    ```

    E agora? Bem, sabemos que o `kube-apiserver` é um pod estático operando dentro do *node* `s2-master-1`. Adicionalmente, sabemos que o *container runtime* utilizado para executar os containers em nossa instalação do Kubernetes é o Docker.

    Podemos, então, consultar diretamente os logs do container via comando `docker`. Para tanto, vamos primeiramente descobrir seu *container ID*:

    ```
    # docker ps -a | grep kube-apiserver
    5e86c83397f2        607331163122           "kube-apiserver --ad…"   About a
    minute ago   Exited (1) 39 seconds ago                         k8s_kube-
    apiserver_kube-apiserver-s2-master-1_kube-
    system_385b165efbf056419463126763d02159_3
    4d5b810938d9        k8s.gcr.io/pause:3.2    "/pause"                 2 minutes
    ago        Up 2 minutes                                          k8s_POD_kube-
    apiserver-s2-master-1_kube-system_385b165efbf056419463126763d02159_0
    ```

    De posse deste, vamos visualizar seus eventos com o comando `docker logs`:

    ```
    # docker logs 5e86c83397f2 --tail 1
    W1007 09:12:37.888428       1 clientconn.go:1223] grpc: addrConn.createTransport
    failed to connect to {https://127.0.0.1:2379  <nil> 0 <nil>}. Err :connection
    error: desc = "transport: authentication handshake failed: x509: certificate
    signed by unknown authority". Reconnecting...
    ```

Uhm... o certificado apresentado ao `kube-apiserver` quando ele se conecta com o serviço na porta 2379 é assinado por uma autoridade certificadora desconhecida. Caso você não se recorde, esse serviço é o `etcd`. Vamos ver como está a configuração de certificados do `kube-apiserver` — especificamente o que define a CA do `etcd`:

```
# cat /etc/kubernetes/manifests/kube-apiserver.yaml | grep etcd-cafile
    - --etcd-cafile=/etc/kubernetes/pki/ca.crt
```

No passo (e) desta atividade observamos que o certificado da CA do `etcd` é definido pela opção `trusted-ca-file`. Naquela ocasião, seu valor era `/etc/kubernetes/pki/etcd/ca.crt`, diferente do utilizado acima. Vamos alterá-lo com o comando `sed`, modificando o arquivo YAML que define o pod estático `kube-apiserver`:

```
# sed -i 's/\(etcd-cafile=\).*/\1\/etc\/kubernetes\/pki\/etcd\/ca.crt/'
/etc/kubernetes/manifests/kube-apiserver.yaml
```

Após algum tempo, o pod estático do `kube-apiserver` volta a ficar operacional. Evidentemente, o comando `kubectl` também fica disponível — veja:

```
# kubectl -n kube-system get pod kube-apiserver-s2-master-1
NAME                          READY   STATUS    RESTARTS   AGE
kube-apiserver-s2-master-1    1/1     Running   0          24s
```

2) API de certificados

2.1) Criando novos usuários

1. Um novo colaborador, `mario`, foi adicionado à equipe; vamos conceder acesso no *cluster* Kubernetes a esse usuário.

   O primeiro passo, evidentemente, é criar sua chave privada e CSR (*Certificate Signing Request*). Faça isso utilizando o comando `openssl`, gerando uma chave RSA de 2048 bits.

   Ao gerar o CSR serão feitas algumas perguntas. Responda-as da seguinte forma:

   - *Country Name*: `BR`
   - *State or Province Name*: Código de 2 letras do seu estado (p.ex. `DF`)
   - *Locality Name*: Sua cidade
   - *Organization Name*: `Contorq`
   - *Organizational Unit Name*: `IT`
   - *Common Name*: `mario`
   - *Email Address*: `mario@contorq.com`
   - *A challenge password*: deixe vazio

- *An optional company name*: deixe vazio

Verifique que ambos os arquivos foram criados com sucesso.

▼ *Visualizar resposta*

Vamos começar gerando a chave privada do usuário:

```
# openssl genrsa -out mario.key 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
..........................++
.....................................................................................
..++
e is 65537 (0x010001)
```

A seguir, criamos seu CSR (*Certificate Signing Request*). Nas perguntas, basta digitar as informações providas no enunciado da atividade.

```
# openssl req -new -key mario.key -out mario.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.

(...)

A challenge password []:
An optional company name []:
```

```
# ls mario.*
mario.csr  mario.key
```

2. Crie um objeto do tipo CertificateSigningRequest usando o CSR criado no passo anterior. Comece pelo arquivo YAML, e em seguida adicione-o ao sistema via `kubectl create`.

   Verifique o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   O arquivo YAML que descreve esse objeto é bastante simples. Como não foram solicitadas permissões adicionais ao usuário iremos manter apenas o grupo padrão, `system:authenticated`.

```yaml
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: mario
spec:
  groups:
  - system:authenticated
```

```
 8   request: MARIO_CSR_BASE64
 9   signerName: kubernetes.io/kube-apiserver-client
10   usages:
11   - client auth
```

Veja que o atributo `request`, acima, não está preenchido. Ele deve conter o CSR do usuário a ser adicionado, codificado em base64. Vamos fazer a codificação e substituição com o comando abaixo:

```
# sed -i "s/MARIO_CSR_BASE64/$( cat mario.csr | base64 | tr -d '\n' )/"
mario_csr.yaml
```

A seguir, criamos o objeto com `kubectl create`.

```
# kubectl create -f mario_csr.yaml
certificatesigningrequest.certificates.k8s.io/mario created
```

Terá funcionado? Vamos ver:

```
# kubectl get csr mario
NAME    AGE   SIGNERNAME                            REQUESTOR          CONDITION
mario   26s   kubernetes.io/kube-apiserver-client   kubernetes-admin   Pending
```

3. Note que o CSR não é aprovado automaticamente. Faça isso.

   Em seguida, imprima na tela o certificado assinado, que deve ser repassado ao usuário `mario` — este, juntamente com a chave privada, serão os recursos utilizados pelo usuário para autenticar-se no *cluster*.

   ▼ *Visualizar resposta*

   A aprovação do certificado é bastante simples, via comando `kubectl certificate`:

   ```
   # kubectl certificate approve mario
   certificatesigningrequest.certificates.k8s.io/mario approved
   ```

   Para obter o certificado basta solicitar o objeto em formato YAML; o atributo `certificate` contém a informação que procuramos.

   ```
   # kubectl get csr/mario -o yaml | grep '^ *certificate:'
     certificate:
   LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURRVENDQWltZ0F3SUJBZ0lRUzVOMmxyUVZreWVXZ
   GpFNXoyTGtNREFOQmdrcWhraUc5dzBCQVFzRkFEQVYKTVJNd0VRWURWUVFERXdwcmRXSmxjbTVsZEdWek
   1CNFhEVEl3TVRBd056QTVOVEV6TTFvWERUSXhNVEF3TnpBNQpOVEV6TTFvd1hERUxNQWtHQTFVRUJoTUN
   RbEl4Q3pBSkJnTlZCQWdUQWtSR01SRXdEd1lEVlFRSEV3aENjbUZ6CmFXeHBZVEVRTUE0R0ExVUVDaE1I
   UTI5dWRHOXljVEVMTUFrR0ExVUVDeE1DU1ZReERqQU1CZ05WQkFNVEJXUmgKZG1sa01JSUJakFOQmdrc
   ```

```
WhraUc5dzBCQVFFRkFBT0NBUThBTUlJQkNnS0NBUUVBdTlvZ1BjVUFoRW1UczBTZApPK1dGNzd5dUlYWE
RERVJteEFKc1lDTHhYdzhCWlR0ZXV1VGlFSEthdFlXQm1aUGNVNTcyUE9EOXlYclVObWZLCi9PM2N5TjN
aMXRjQ2RFL2IxOEZCa0YwU3BFcTY2UlZSaVdhejBKejdvVnl1QUlqTGZ0b3E3U1FLVEV1cnhuYjIKWTds
aEI2UURMM0M4b3NRQVFxdWpNZUVMajNiSlZuTndGSzBkUnR6SkZoK1Rmbm1BYXo0Tk9wVTdPUUMxTlRyY
QpWaHZDTjdTTjFNRi8vR0N2dGczd0NFcndLYjVXRXpTL09Tc3J5eUdidXVJMDVTbmV2eE9TbTBzZW16V1
M3MmpWCkpJMzVTdE1ZL1o4UExsUXN2clNUS09Ua2Rvd1hRWk5Ma2g4aUo1NERCdlU1YW5IcVhYQ0lBZUZ
yQWlGZmx3SjAKMDRXNmx3SURBUUFCbzBZd1JEQVRCZ05WSFNVRUREQUtCZ2dyQmdFRkJRY0RBakFNQmdO
VkhSTUJBZjhFQWpBQQpNQjhHQTFVZEl3UVlNQmFBRkxMT1RySlk4b1loNUtvb0psMnp1d2RXU2VxOE1BM
EdDU3FHU0liM0RRRUJDd1VBCkE0SUJBUUFSTTNRZVNJNFdSckJBNWhtUklvRVV2blFqRlQ5dVZwcS9TVz
hkNU1UOVVDTUN6UXZwTTh4c0JQMk4KZkI0U0VQQk1uUUpCVzZiakN6Mmo5ZEpCZnJneWlzRVhqUXplMlp
scE5UVlhvNGViZGRBb0daTmF0dkF5MWxIMgpOWmpCWDRlTUR4dHB3Zlh3dTQzcE42UW52ZmE0VEgrbmtC
SjcvTHZ0cUJKTVYzUzdWNDNGUURhREZ5UjY0WXBtCjVxRDZSZG9hS1J0d1o4amkzdVhvYTFQa3UwNDVLb
0VDbmhRd202RDZ6aktMTjJXODlhWGJiMXl0dm5GUGxBRS8KOXp3RE16Uno1UXRIdEJtS0VJaGV4bzBaYW
M1cE5MZzkwM3phd0wzNllRZFlIODArMUFyUGJEWnNyWkMrTnNsVwpTOEZVZ2loUkNyU2xLa1h4Y1QzT1A
2RWRUSm5JCi0tLS0tRU5EIENFUlRJRklDQVRFLS0tLS0K
```

Note que o certificado acima está codificado em `base64` — para utilizá-lo, é necessário primeiro realizar o *decode* dessa *string*.

4. Crie, se não existir, a pasta `/root/.kube/users`. A seguir, copie para dentro dessa pasta a chave privada e o certificado assinado do usuário `mario`, utilizando os nomes de arquivo `mario.key` e `mario.crt`, respectivamente.

   ▼ *Visualizar resposta*

   Vamos primeiramente criar a pasta:

   ```
   # mkdir -p /root/.kube/users
   ```

   A seguir, copiamos a chave privada para a pasta.

   ```
   # mv /root/mario.key /root/.kube/users
   ```

   Iremos usar um comando semelhante ao utilizado na atividade anterior para extrair o certificado assinado — desta vez, contudo, iremos usar o comando `base64` para decodificar a *string* e redirecionar o resultado para o arquivo `mario.crt`.

   ```
   # kubectl get csr/mario -o jsonpath='{.status.certificate}' | base64 -d >
   /root/.kube/users/mario.crt
   ```

   Vejamos se tudo funcionou a contento:

   ```
   # ls -1 /root/.kube/users/
   mario.crt
   mario.key
   ```

2.2) Detectando inconsistências

1. Antes de prosseguir, execute o comando abaixo:

```
# lab-6.2.2
```

2. Um novo CSR foi adicionado. Qual é ele?

▼ *Visualizar resposta*

Vejamos:

```
# kubectl get csr | grep 'Pending'
bowser      118s   kubernetes.io/kube-apiserver-client          kubernetes-admin
Pending
```

3. Que estranho, não adicionamos nenhum CSR com esse nome até aqui... verifique a quais grupos esse CSR solicita acesso.

▼ *Visualizar resposta*

Visualize o CSR em formato YAML, buscando o atributo `groups`. Assim:

```
# kubectl get csr bowser -o yaml | grep '^ *groups:' -A2
  groups:
  - system:masters
  - system:authenticated
```

Note que além do grupo padrão `system:authenticated`, o usuário também seria adicionado a `system:masters`: esse grupo efetivamente possui permissões totais sobre o *cluster*.

4. Perigoso, não é mesmo? Caso esse CSR fosse aprovado, o usuário em questão poderia ter acesso irrestrito ao *cluster*. Negue a solicitação, e em seguida remova o CSR.

▼ *Visualizar resposta*

Primeiro, negamos a solicitação:

```
# kubectl certificate deny bowser
certificatesigningrequest.certificates.k8s.io/bowser denied
```

E, depois, removemos o CSR:

```
# kubectl delete csr bowser
certificatesigningrequest.certificates.k8s.io "bowser" deleted
```

3) KubeConfig

3.1) Lidando com arquivos KubeConfig simples

1. Há três diferentes modos de definição do arquivo `kubeconfig` a ser utilizado em um comando `kubectl`. Quais são eles?

   ▼ *Visualizar resposta*

   Segundo a documentação do Kubernetes, disponível em https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/ , as três formas são:

   1. Arquivo com o nome `config`, dentro do diretório `$HOME/.kube` do usuário.
   2. Arquivo apontado pela variável de ambiente `$KUBECONFIG` no *shell* corrente.
   3. Arquivo especificado diretamente através da *flag* `--kubeconfig` passada ao comando `kubectl`.

2. Considere o arquivo `kubeconfig` sendo utilizado no momento, pelo usuário `root`. Quantos *clusters*, usuários e contextos estão definidos nesse arquivo, e quais são eles? Qual é o contexto padrão configurado?

   ▼ *Visualizar resposta*

   Vamos ver se o arquivo `kubeconfig` padrão, descrito pelo item (1) da resposta ao passo anterior, existe.

   ```
   # ls -d ${HOME}/.kube/config
   /root/.kube/config
   ```

   Poderíamos visualizar as informações solicitadas diretamente no arquivo, e contar "no olho". Mas, muito melhor, podemos utilizar uma ferramenta de *parsing* especializada em arquivos YAML, como é o caso da `yq`. Instale-a:

   ```
   <strong># wget
   https://github.com/mikefarah/yq/releases/download/3.4.1/yq_linux_amd64 -O
   /usr/bin/yq && chmod +x /usr/bin/yq</strong>
   ```

   Para utilizar o `yq`, basta invocá-lo com o parâmetro `read`, ou `r`, juntamente com a *string* de busca. Por exemplo, vamos solicitar os nomes de todos os *clusters* existentes no arquivo, e em seguida contar o número de linhas — efetivamente, respondendo à pergunta de quantos *clusters* estão definidos.

   ```
   # yq r /root/.kube/config 'clusters[*].name' | wc -l
   1
   ```

   Removendo o `wc` podemos visualizar os nomes desses *clusters*:

```
# yq r /root/.kube/config 'clusters[*].name'
kubernetes
```

Qual será o endereço do servidor desse *cluster*?

```
# yq r /root/.kube/config 'clusters[*].cluster.server'
https://192.168.68.20:6443
```

Façamos o mesmo procedimento para contar o número, e definir quais são, os usuários definidos no `kubeconfig`:

```
# yq r /root/.kube/config 'users[*].name' | wc -l
1
```

```
# yq r /root/.kube/config 'users[*].name'
kubernetes-admin
```

E, finalmente, vejamos as informações dos contextos:

```
# yq r /root/.kube/config 'contexts[*].name' | wc -l
1
```

```
# yq r /root/.kube/config 'contexts[*].name'
kubernetes-admin@kubernetes
```

```
# yq r /root/.kube/config 'contexts[*].context'
cluster: kubernetes
user: kubernetes-admin
```

O `current-context` define o contexto padrão ao utilizar o `kubeconfig` em questão. Para visualizá-lo, execute:

```
# yq r /root/.kube/config 'current-context'
kubernetes-admin@kubernetes
```

3. Vamos retomar a configuração do usuário `mario`, iniciada na atividade (2).

   Adicione as credenciais (chave privada e certificado assinado) desse usuário ao arquivo `/root/.kube/config`. A seguir, adicione um novo contexto, `mario@kubernetes`, relacionando esse usuário ao *cluster* `kubernetes` (sugestão: utilize o comando `kubectl config`).

Finalmente, altere o contexto padrão para o recém-criado `mario@kubernetes` e verifique seu funcionamento com um comando `kubectl` qualquer. Note que, como ainda não atribuímos permissões a esse usuário, você verá um erro: **isso é esperado**, e será corrigido na próxima atividade.

▼ *Visualizar resposta*

Podemos adicionar as credenciais manualmente ou, muito melhor, fazê-lo automaticamente com o comando `kubectl config set-credentials`. Para tanto, basta apontar os arquivos de certificado e chave com as *flags* apropriadas:

```
# kubectl config set-credentials mario --client-key=/root/.kube/users/mario.key
--client-certificate=/root/.kube/users/mario.crt --embed-certs=true
User "mario" set.
```

Vamos validar se o comando funcionou usando o `yq`.

```
# yq r /root/.kube/config 'users[*].name'
kubernetes-admin
mario
```

A seguir, criamos um contexto para o novo usuário com o *cluster* preexistente:

```
# kubectl config set-context mario@kubernetes --cluster=kubernetes --user=mario
Context "mario@kubernetes" created.
```

Terá funcionado?

```
# yq r /root/.kube/config 'contexts[*].name'
kubernetes-admin@kubernetes
mario@kubernetes
```

Perfeito. Vamos agora alterar o contexto padrão — primeiro, verificando seu valor atual.

```
# yq r /root/.kube/config 'current-context'
kubernetes-admin@kubernetes
```

Alterando-o:

```
# kubectl config use-context mario@kubernetes
Switched to context "mario@kubernetes".
```

E validando o funcionamento da configuração:

```
# yq r /root/.kube/config 'current-context'
mario@kubernetes
```

Ao tentar interagir com o *cluster*, obtemos o erro aludido pelo enunciado.

```
# kubectl get pod
Error from server (Forbidden): pods is forbidden: User "mario" cannot list
resource "pods" in API group "" in the namespace "default"
```

4. Finalmente, retorne ao contexto anterior com o comando:

```
# kubectl config use-context kubernetes-admin@kubernetes
Switched to context "kubernetes-admin@kubernetes".
```

3.2) Lidando com arquivos KubeConfig complexos

1. Antes de prosseguir, execute o comando abaixo:

```
# lab-6.3.2
```

2. Vamos agora considerar um arquivo `kubeconfig` mais complexo, acessível em `/root/custom-kubeconfig`. Quantos *clusters*, usuários e contextos estão definidos nesse arquivo, e quais são eles?

▼ *Visualizar resposta*

Usando o `yq`, é trivial responder as perguntas realizadas. Primeiro, as informações de *clusters*:

```
# yq r /root/custom-kubeconfig 'clusters[*].name' | wc -l
3
```

```
# yq r /root/custom-kubeconfig 'clusters[*].name'
diehard
predator
bloodsport
```

Agora, de usuários:

```
# yq r /root/custom-kubeconfig 'users[*].name' | wc -l
3
```

```
# yq r /root/custom-kubeconfig 'users[*].name'
```

```
mcclane
dutch
dux
```

E, finalmente, de contextos:

```
# yq r /root/custom-kubeconfig 'contexts[*].name' | wc -l
3
```

```
# yq r /root/custom-kubeconfig 'contexts[*].name'
mcclane@diehard
dutch@predator
dux@bloodsport
```

3. Quais são os arquivos de certificado e chave privada utilizados pelo usuário `mcclane`?

▼ *Visualizar resposta*

Essas informações podem ser visualizadas na seção `users.user` do usuário em questão. Iremos referenciá-lo pelo índice do usuário, com a ferramenta `yq`.

```
# yq r /root/custom-kubeconfig 'users[2].user.client-certificate'
/etc/kubernetes/pki/users/mcclane.cert
```

```
# yq r /root/custom-kubeconfig 'users[2].user.client-key'
/etc/kubernetes/pki/users/mcclane.key
```

4. Qual é o contexto padrão configurado? Se necessário, altere-o para que esse contexto seja `mcclane@diehard`.

Verifique o funcionamento de sua configuração.

▼ *Visualizar resposta*

Vejamos o contexto padrão:

```
# yq r /root/custom-kubeconfig 'current-context'
dutch@predator
```

A seguir, alteramos o contexto e verificamos a efetivação da configuração.

```
# kubectl config use-context mcclane@diehard --kubeconfig=/root/custom-kubeconfig
Switched to context "mcclane@diehard".
```

```
# yq r /root/custom-kubeconfig 'current-context'
mcclane@diehard
```

5. Ter que especificar a *flag* `--kubeconfig` a cada comando `kubectl` utilizado é bastante inconveniente. Faça com que o arquivo `kubeconfig` `/root/custom-kubeconfig` seja utilizado como o `kubeconfig` padrão **apenas** na sessão corrente do *shell*.

   Não sobrescreva o arquivo `/root/.kube/config` existente.

   ▼ *Visualizar resposta*

   Além de utilizar a *flag* `--kubeconfig` e via arquivo `$HOME/.kube/config`, é também possível definir o arquivo `kubeconfig` a ser utilizado através da variável de ambiente `$KUBECONFIG`. Defina-a:

   ```
   # export KUBECONFIG=/root/custom-kubeconfig
   ```

6. Ao tentar utilizar qualquer comando `kubectl` para interagir com o *cluster*, um erro é encontrado. Qual é ele?

   Determine sua razão, e solucione o problema.

   ▼ *Visualizar resposta*

   Vamos ver qual é esse erro:

   ```
   # kubectl get pod
   error: unable to read client-cert /etc/kubernetes/pki/users/mcclane.cert for
   mcclane due to open /etc/kubernetes/pki/users/mcclane.cert: no such file or
   directory
   ```

   O arquivo `/etc/kubernetes/pki/users/mcclane.cert` não foi encontrado. Quais arquivos existem nesse diretório?

   ```
   # ls -1 /etc/kubernetes/pki/users/mcclane.*
   /etc/kubernetes/pki/users/mcclane.crt
   /etc/kubernetes/pki/users/mcclane.key
   ```

   Note que o nome correto do arquivo é `mcclane.crt`, e não `mcclane.cert`. Vamos corrigir a informação via `sed`:

   ```
   # sed -i 's/mcclane.cert/mcclane.crt/' /root/custom-kubeconfig
   ```

   E, em seguida, testar o funcionamento do comando `kubectl`:

   ```
   # kubectl get node
   ```

```
NAME          STATUS   ROLES    AGE     VERSION
s2-master-1   Ready    master   3d23h   v1.19.2
s2-node-1     Ready    <none>   3d23h   v1.19.2
```

7. Finalmente, volte a utilizar o arquivo `kubeconfig` padrão com o comando:

```
# unset KUBECONFIG
```

Valide o funcionamento de sua configuração com o comando:

```
# kubectl get node
NAME          STATUS   ROLES    AGE     VERSION
s2-master-1   Ready    master   3d23h   v1.19.2
s2-node-1     Ready    <none>   3d23h   v1.19.2
```

4) Controle de acesso baseado em papéis

4.1) Papéis-padrão do sistema

1. O Kubernetes suporta diversos métodos de autorização, documentados em https://kubernetes.io/docs/reference/access-authn-authz/authorization/#authorization-modules . Quais modos de autorização estão configurados no *cluster*, no momento?

▼ *Visualizar resposta*

Essa informação é definida através da opção `--authorization-mode`, quando da invocação do `kube-apiserver`.

```
# kubectl -n kube-system describe pod kube-apiserver-s2-master-1 | grep
authorization-mode
      --authorization-mode=Node,RBAC
```

2. Quais são as *roles* (papéis) existentes em cada um dos namespaces presentes no *cluster*?

▼ *Visualizar resposta*

Pode-se utilizar o comando `kubectl get role`, circulando por cada um dos namespaces e contando os *roles* existentes em cada um deles. Alternativamente, podemos fazer isso em um único comando, como no formato mostrado abaixo:

```
# for ns in $( kubectl get ns -o custom-columns=NAME:.metadata.name --no-headers
); do echo -e "\nNamespace: ${ns}\n-----\n"; kubectl get role -n ${ns}; done

Namespace: default
-----

No resources found in default namespace.
```

```
Namespace: kube-node-lease
-----

No resources found in kube-node-lease namespace.

Namespace: kube-public
-----

NAME                                  CREATED AT
kubeadm:bootstrap-signer-clusterinfo   2022-03-23T12:20:34Z
system:controller:bootstrap-signer     2022-03-23T12:20:32Z

Namespace: kube-system
-----

NAME                                                 CREATED AT
extension-apiserver-authentication-reader            2022-03-23T12:20:32Z
kube-proxy                                           2022-03-23T12:20:35Z
kubeadm:kubelet-config-1.23                          2022-03-23T12:20:32Z
kubeadm:nodes-kubeadm-config                         2022-03-23T12:20:32Z
system::leader-locking-kube-controller-manager       2022-03-23T12:20:32Z
system::leader-locking-kube-scheduler                2022-03-23T12:20:32Z
system:controller:bootstrap-signer                   2022-03-23T12:20:32Z
system:controller:cloud-provider                     2022-03-23T12:20:32Z
system:controller:token-cleaner                      2022-03-23T12:20:32Z
weave-net                                            2022-03-23T12:20:37Z
```

3. Quais recursos, e ações sobre esses recursos, são garantidos à *role* `weave-net`, no namespace `kube-system`?

   ▼ *Visualizar resposta*

   Para tanto, basta utilizar `kubectl describe`. Veja que a *role* possui permissão de criar ConfigMaps, e também visualizar e alterar o ConfigMap `weave-net`, especificamente.

   ```
   # kubectl -n kube-system describe role weave-net | grep 'PolicyRule:' -A5
   PolicyRule:
     Resources   Non-Resource URLs  Resource Names  Verbs
     ---------   -----------------  --------------  -----
     configmaps  []                 []              [create]
     configmaps  []                 [weave-net]     [get update]
   ```

4. Quais RoleBindings existem em cada um dos namespaces presentes no *cluster*?

   ▼ *Visualizar resposta*

   Assim como no passo (b), realizamos o mesmo procedimento com os objetos do tipo RoleBinding.

```
# for ns in $( kubectl get ns -o custom-columns=NAME:.metadata.name --no-headers
); do echo -e "\nNamespace: ${ns}\n-----\n"; kubectl get rolebinding -n ${ns};
done

Namespace: default
-----

No resources found in default namespace.

Namespace: kube-node-lease
-----

No resources found in kube-node-lease namespace.

Namespace: kube-public
-----

NAME                                  ROLE
AGE
kubeadm:bootstrap-signer-clusterinfo   Role/kubeadm:bootstrap-signer-clusterinfo
3d2h
system:controller:bootstrap-signer     Role/system:controller:bootstrap-signer
3d2h

Namespace: kube-system
-----

NAME                                                ROLE
   AGE
kube-proxy                                          Role/kube-proxy
   3d2h
kubeadm:kubelet-config-1.23                         Role/kubeadm:kubelet-config-
1.23
   3d2h
kubeadm:nodes-kubeadm-config                        Role/kubeadm:nodes-kubeadm-
config
   3d2h
metrics-server-auth-reader                          Role/extension-apiserver-
authentication-reader        26h
system::extension-apiserver-authentication-reader   Role/extension-apiserver-
authentication-reader        3d2h
system::leader-locking-kube-controller-manager      Role/system::leader-locking-
kube-controller-manager   3d2h
system::leader-locking-kube-scheduler               Role/system::leader-locking-
kube-scheduler
   3d2h
system:controller:bootstrap-signer
Role/system:controller:bootstrap-signer
   3d2h
system:controller:cloud-provider                    Role/system:controller:cloud-
```

```
provider
   3d2h
system:controller:token-cleaner                    Role/system:controller:token-
cleaner
   3d2h
weave-net                                          Role/weave-net
   3d2h
```

5. Qual é a conta associada à *role* `weave-net`, no namespace `kube-system`?

   ▼ *Visualizar resposta*

   Para determinar esse informação, basta utilizar `kubectl describe`. Veja que trata-se de uma conta do tipo ServiceAccount.

   ```
   # kubectl -n kube-system describe rolebinding weave-net | grep 'Role:' -A6
   Role:
     Kind:  Role
     Name:  weave-net
   Subjects:
     Kind            Name       Namespace
     ----            ----       ---------
     ServiceAccount  weave-net  kube-system
   ```

4.2) Criando novos papéis

1. Vamos voltar à configuração do usuário `mario`. Utilizando o comando `kubectl auth`, verifique se esse usuário possui permissão para visualizar a lista de pods no namespace *default*.

   ▼ *Visualizar resposta*

   De forma similar ao comando `sudo`, no *shell* Bash, pode-se utilizar a *flag* `--as` para simular a execução de um comando como outro usuário. Para validar se é possível executar uma ação no *cluster*, pode-se informar o comando `can-i` ao `kubectl auth`. Assim:

   ```
   # kubectl auth can-i list pods --as mario
   no
   ```

2. Crie a *role* `pod-admin`, com permissões de listagem, criação e deleção de pods no namespace *default*. Visualize-a em detalhes, com `kubectl describe`.

   Em seguida, crie o RoleBinding `pod-admin-mario` ligando a *role* `pod-admin` ao usuário `mario`. Visualize-a em detalhes, com `kubectl describe`.

   Valide sua configuração através da criação, listagem e subsequente remoção de um pod qualquer no namespace *default*.

   ▼ *Visualizar resposta*

   Primeiramente, criaremos a *role*.

```
# kubectl create role pod-admin --resource=pods --verb=create,list,delete
role.rbac.authorization.k8s.io/pod-admin created
```

Vamos verificar se os objetos e ações desejados são de fato os afetados pela *role*:

```
# kubectl describe role pod-admin
Name:         pod-admin
Labels:       <none>
Annotations:  <none>
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  pods       []                 []              [create list delete]
```

Perfeito. Vamos agora criar o RoleBinding, associando a *role* ao usuário `mario`.

```
# kubectl create rolebinding pod-admin-mario --role=pod-admin --user=mario
rolebinding.rbac.authorization.k8s.io/pod-admin-mario created
```

Novamente, verificamos se a ação surtiu o efeito esperado:

```
# kubectl describe rolebinding pod-admin-mario
Name:         pod-admin-mario
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  Role
  Name:  pod-admin
Subjects:
  Kind  Name   Namespace
  ----  ----   ---------
  User  mario
```

Retomemos o uso do comando `kubectl auth can-i`. Como ficou o permissionamento do usuário?

```
# kubectl auth can-i list pods --as mario
yes
```

Agora, o usuário `mario` deve ter permissão de listar, criar e deletar pods. Testemos:

```
# kubectl run test --image=nginx:alpine --as=mario
pod/test created
```

```
# kubectl get pod --as=mario
NAME   READY   STATUS    RESTARTS   AGE
test   1/1     Running   0          26s
```

```
# kubectl delete pod test --as=mario
pod "test" deleted
```

3. Antes de prosseguir, execute o comando abaixo:

```
# lab-6.4.2
```

4. O usuário `mario` está tentando visualizar informações sobre os pods `et` e `indiana` criados em um novo namespace, `spielberg`. É possível? E, se sim, ambos os pods são investigáveis?

   Observe as *roles* e RoleBindings criados no namespace que justificam o funcionamento dos comandos.

   ▼ *Visualizar resposta*

   Vamos tentar visualizar as informações dos pods informados. Primeiro, o pod `et`:

```
# kubectl -n spielberg get pod et --as mario
NAME   READY   STATUS    RESTARTS   AGE
et     1/1     Running   0          4m33s
```

   Tudo certo. E quanto ao pod `indiana`?

```
# kubectl -n spielberg get pod indiana --as mario
Error from server (Forbidden): pods "indiana" is forbidden: User "mario" cannot
get resource "pods" in API group "" in the namespace "spielberg"
```

   Ahá! Aí está o erro. Vamos ver quais roles existem dentro do namespace:

```
# kubectl -n spielberg get role
NAME         CREATED AT
pod-reader   2022-03-26T14:23:40Z
```

   Quais seriam as permissões garantidas por essa *role*?

```
# kubectl -n spielberg describe role pod-reader
Name:         pod-reader
Labels:       <none>
Annotations:  <none>
PolicyRule:
```

```
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  pods       []                 [et]            [get watch list]
```

Note que apenas o pod `et` é afetado pela *role* acima, como indicado pela coluna *Resource Names*. Como conseguimos visualizar informações sobre o pod `et` anteriormente (mas não `indiana`), é de se imaginar que o usuário `mario` tenha *binding* sob essa *role* — vamos verificar.

```
# kubectl -n spielberg get rolebinding
NAME                ROLE               AGE
pod-reader-mario    Role/pod-reader    5m16s
```

```
# kubectl -n spielberg describe rolebinding pod-reader-mario
Name:         pod-reader-mario
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  Role
  Name:  pod-reader
Subjects:
  Kind  Name   Namespace
  ----  ----   ---------
  User  mario
```

5. Edite a *role* `pod-reader` de forma que o usuário `mario` possa investigar também o pod `indiana`. Não ofereça mais permissões que o estritamente necessário para que o usuário possa realizar essa tarefa.

   ▼ *Visualizar resposta*

   Vamos lá. Primeiro, invoque a edição da *role*:

   ```
   # kubectl -n spielberg edit role pod-reader
   role.rbac.authorization.k8s.io/pod-reader edited
   ```

   Na seção `.rules.resourceNames`, adicione o nome do pod `indiana` ao do pod `et`, que já consta na lista de `resourceNames`.

   ```
   resourceNames:
   - et
   - indiana
   ```

   Agora sim, deve funcionar. Vamos ver:

   ```
   # kubectl -n spielberg get pod indiana --as mario
   ```

```
NAME      READY   STATUS    RESTARTS   AGE
indiana   1/1     Running   0          8m49s
```

6. O usuário `mario` agora irá necessitar de permissões para criar, listar e deletar deployments no namespace `spielberg`. Caso você tente editar a *role* preexistente, `pod-reader`, irá notar que não é possível alterar alguns de seus parâmetros-chave, como `apiVersion`, `kind` e `name`.

   Exporte a *role* `pod-reader` em formato YAML. A seguir edite o arquivo, alterando o nome da *role* para `deployadm-podrdr` e adicionando as permissões mencionadas no parágrafo anterior. Tenhas especial atenção com o atributo `apiGroups`.

   Remova a *role* `pod-reader` e seu RoleBinding associado.

   Finalmente, adicione via arquivo YAML a *role* `deployadm-podrdr` e crie um RoleBinding com o nome `deployadm-podrdr-mario`, associando a *role* ao usuário `mario`. Crie, liste e remova um deployment qualquer para validar sua configuração.

   ▼ *Visualizar resposta*

   Como a *role* deve ter seu nome alterado, iremos exportá-la e editar seu nome e permissionamento conforme solicitado. Vamos começar pela exportação:

   ```
   # kubectl -n spielberg get role pod-reader -o yaml > deployadm-podrdr.yaml
   ```

   Agora, edite seu conteúdo, que deve ficar como mostrado abaixo. Note que as seção não-relevantes para a configuração da *role* foram omitidas aqui.

   ```
   1 apiVersion: rbac.authorization.k8s.io/v1
   2 kind: Role
   3 metadata:
   4   namespace: spielberg
   5   name: deployadm-podrdr
   6 rules:
   7 - apiGroups: [""]
   8   resources: ["pods"]
   9   resourceNames: ["et", "indiana"]
  10   verbs: ["get", "watch", "list"]
  11 - apiGroups: ["apps", "extensions"]
  12   resources: ["deployments"]
  13   verbs: ["create", "delete", "list"]
   ```

   Excelente. Vamos remover a *role* e RoleBinding...

   ```
   # kubectl -n spielberg delete role pod-reader
   role.rbac.authorization.k8s.io "pod-reader" deleted
   ```

   ```
   # kubectl -n spielberg delete rolebinding pod-reader-mario
   ```

```
rolebinding.rbac.authorization.k8s.io "pod-reader-mario" deleted
```

E recriar a *role* via arquivo YAML.

```
# kubectl apply -f deployadm-podrdr.yaml
role.rbac.authorization.k8s.io/deployadm-podrdr created
```

```
# kubectl -n spielberg describe role deployadm-podrdr
Name:         deployadm-podrdr
Labels:       <none>
Annotations:  <none>
PolicyRule:
  Resources               Non-Resource URLs  Resource Names  Verbs
  ---------               -----------------  --------------  -----
  deployments.apps        []                 []              [create delete list]
  deployments.extensions  []                 []              [create delete list]
  pods                    []                 [et]            [get watch list]
  pods                    []                 [indiana]       [get watch list]
```

A seguir, refazemos a associação do usuário via RoleBinding.

```
# kubectl -n spielberg create rolebinding deployadm-podrdr-mario --role=deployadm
-podrdr --user=mario
rolebinding.rbac.authorization.k8s.io/deployadm-podrdr-mario created
```

E, finalmente, o teste. Vamos criar, listar e remover um deployment qualquer para validar nossa configuração.

```
# kubectl -n spielberg create deploy test --image=nginx:alpine --replicas=2
--as=mario
deployment.apps/test created
```

```
# kubectl -n spielberg get deploy --as mario
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
test   2/2     2            2           16s
```

```
# kubectl -n spielberg delete deploy test --as mario
deployment.apps "test" deleted
```

7. Remova o namespace `spielberg` para liberar recursos do *cluster*.

```
# kubectl delete ns spielberg
```

```
namespace "spielberg" deleted
```

Note que, ao remover o namespace, todos os recuros nele contidos (como pods, Roles e RoleBindings) também são removidos.

5) Papéis *cluster-wide*

1. Quantos ClusterRoles e ClusterRoleBindings existem no *cluster*, no momento?

   ▼ *Visualizar resposta*

   Com o uso da *flag* `--no-headers` e o comando `wc`, é trivial descobrir essas informações. Veja:

   ```
   # kubectl get clusterrole --no-headers | wc -l
   67
   ```

   ```
   # kubectl get clusterrolebinding --no-headers | wc -l
   52
   ```

2. Em qual namespace está inserido o ClusterRole `cluster-admin`?

   ▼ *Visualizar resposta*

   Como documentado em https://kubernetes.io/docs/reference/access-authn-authz/rbac/#role-and-clusterrole, ClusterRoles são recursos *non-namespaced* (isto é, não inseridos em nenhum namespace), mas sim *cluster-wide*.

   ClusterRoles podem ser utilizados para garantir acesso a:

   - Recursos no escopo do *cluster*, como *nodes*.
   - *Endpoints* que não configuram recursos *per se*, como `/healthz`.
   - Recursos dentro do escopo de namespaces, como pods, garantindo acesso através de todos os namespaces (por exemplo, o comando `kubectl get pods --all-namespaces`).

3. Quais usuários ou grupos são associados ao ClusterRole `cluster-admin`? Que tipo de permissionamento esses agentes possuem sobre o *cluster*?

   ▼ *Visualizar resposta*

   A associação a ClusterRoles pode ser determinada através do exame de seu(s) ClusterRoleBindings correspondentes, via `kubectl describe`:

   ```
   # kubectl describe clusterrolebinding cluster-admin
   Name:         cluster-admin
   Labels:       kubernetes.io/bootstrapping=rbac-defaults
   Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
   Role:
     Kind:  ClusterRole
   ```

```
  Name:  cluster-admin
Subjects:
  Kind   Name            Namespace
  ----   ----            ---------
  Group  system:masters
```

Quais seriam essas permissões? Vamos ver:

```
# kubectl describe clusterrole cluster-admin
Name:         cluster-admin
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  *.*        []                 []              [*]
             [*]                []              [*]
```

Portanto, os usuário associados ao grupo `system:masters` possuem permissão total sobre todos os recursos do *cluster*.

4. O Kubernetes, como sabemos, possui diversos objetos e recursos à disposição do administrador. Utilizando o comando `kubectl api-groups`, determine quais recursos são *cluster-wide* e quais deles não o são. Exiba apenas seus nomes, em ordem alfabética.

   ▼ *Visualizar resposta*

   A *flag* `--namespaced` pode ser usada para mostrar aqueles recursos que não estão associados a namespaces (isto é, *cluster-wide*). Via `--sort-by` podemos organizar a lista alfabeticamente, ainda.

   ```
   # kubectl api-resources --namespaced=false --sort-by=name -o=name
   apiservices.apiregistration.k8s.io
   certificatesigningrequests.certificates.k8s.io
   clusterrolebindings.rbac.authorization.k8s.io
   clusterroles.rbac.authorization.k8s.io
   componentstatuses
   csidrivers.storage.k8s.io
   csinodes.storage.k8s.io
   customresourcedefinitions.apiextensions.k8s.io
   flowschemas.flowcontrol.apiserver.k8s.io
   ingressclasses.networking.k8s.io
   mutatingwebhookconfigurations.admissionregistration.k8s.io
   namespaces
   nodes
   nodes.metrics.k8s.io
   persistentvolumes
   podsecuritypolicies.policy
   priorityclasses.scheduling.k8s.io
   ```

```
prioritylevelconfigurations.flowcontrol.apiserver.k8s.io
runtimeclasses.node.k8s.io
selfsubjectaccessreviews.authorization.k8s.io
selfsubjectrulesreviews.authorization.k8s.io
storageclasses.storage.k8s.io
subjectaccessreviews.authorization.k8s.io
tokenreviews.authentication.k8s.io
validatingwebhookconfigurations.admissionregistration.k8s.io
volumeattachments.storage.k8s.io
```

Alterando a *flag* `--namespaced` para `true`, descobrimos os recursos que são associados a namespaces.

```
# kubectl api-resources --namespaced=true --sort-by=name -o=name
bindings
configmaps
controllerrevisions.apps
cronjobs.batch
csistoragecapacities.storage.k8s.io
daemonsets.apps
deployments.apps
endpoints
endpointslices.discovery.k8s.io
events
events.events.k8s.io
horizontalpodautoscalers.autoscaling
ingresses.networking.k8s.io
jobs.batch
leases.coordination.k8s.io
limitranges
localsubjectaccessreviews.authorization.k8s.io
networkpolicies.networking.k8s.io
persistentvolumeclaims
poddisruptionbudgets.policy
pods
pods.metrics.k8s.io
podtemplates
replicasets.apps
replicationcontrollers
resourcequotas
rolebindings.rbac.authorization.k8s.io
roles.rbac.authorization.k8s.io
secrets
serviceaccounts
services
statefulsets.apps
```

5. O usuário `mario` está agora responsável pelo monitoramento de *nodes* do *cluster*. Naturalmente, permissões apropriadas devem ser dadas ao usuário de forma que ele consiga realizar seu

trabalho.

Primeiramente, determine as ações (`verbs`) suportadas pelo objeto-alvo (`nodes`).

A seguir, crie o ClusterRole `node-monitor` que garanta todas as permissões não-invasivas (isto é, que permitem criação, deleção ou edição de objetos) ao objeto-alvo `nodes`.

Crie, também, o ClusterRoleBinding `node-monitor-mario` que associe o ClusterRole `node-monitor` ao usuário `mario`.

Finalmente, teste o funcionamento de sua configuração.

▼ *Visualizar resposta*

A opção `-o=wide` do comando `kubectl api-resources` permite mostrar as ações associadas a um objeto. Vamos verificar quais são essas ações para *nodes*:

```
# kubectl api-resources --namespaced=false --sort-by=name -o=wide | grep ' Node '
| grep -o '\[.*\]'
[create delete deletecollection get list patch update watch]
```

Evidentemente, as ações `create`, `delete`, `deletecollection`, `patch` e `update` permitem ativa alteração do estado de objetos. Vamos criar a ClusterRole removendo esses `verbs`:

```
# kubectl create clusterrole node-monitor --resource=nodes --verb=get,list,watch
clusterrole.rbac.authorization.k8s.io/node-monitor created
```

Verifiquemos:

```
# kubectl describe clusterrole node-monitor
Name:         node-monitor
Labels:       <none>
Annotations:  <none>
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  nodes      []                 []              [get list watch]
```

Agora, ao ClusterRoleBinding:

```
# kubectl create clusterrolebinding node-monitor-mario --clusterrole=node-monitor
--user=mario
clusterrolebinding.rbac.authorization.k8s.io/node-monitor-mario created
```

```
# kubectl describe clusterrolebinding node-monitor-mario
Name:         node-monitor-mario
```

```
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  ClusterRole
  Name:  node-monitor
Subjects:
  Kind  Name   Namespace
  ----  ----   ---------
  User  mario
```

Via `kubectl auth`, testemos se o usuário possui permissão para realizar as ações adicionadas ao ClusterRole.

```
# kubectl auth can-i list nodes --as mario
Warning: resource 'nodes' is not namespace scoped
yes
```

E, claro, vamos verificar também com o comando "ao vivo":

```
# kubectl get nodes --as mario
NAME          STATUS   ROLES                  AGE    VERSION
s2-master-1   Ready    control-plane,master   3d2h   v1.23.5
s2-node-1     Ready    <none>                 3d2h   v1.23.5
```

6. O usuário `mario` também ficará responsável pela gestão de armazenamento do *cluster*, incluindo criação, remoção e quaisquer outras ações sobre objetos relacionados ao `apiGroup` `storage.k8s.io`.

   Além desses objetos, as mesmas permissões deverão ser garantidas a PersistentVolumes, discutidos em maior detalhe no capítulo dedicado à gestão de armazenamento no Kubernetes.

   Crie, via arquivos YAML, o ClusterRole e ClusterRoleBinding que atingem os objetivos acima. A seguir, teste sua configuração.

   ▼ *Visualizar resposta*

   Primeiro, determine quais recursos *cluster-wide* pertencem ao `apiGroup` em questão.

```
# kubectl api-resources --namespaced=false | grep 'storage.k8s.io'
csidrivers                                      storage.k8s.io/v1
false        CSIDriver
csinodes                                        storage.k8s.io/v1
false        CSINode
storageclasses                    sc            storage.k8s.io/v1
false        StorageClass
volumeattachments                               storage.k8s.io/v1
false        VolumeAttachment
```

Além desses, temos também o recurso `PersistentVolume`:

```
# kubectl api-resources --namespaced=false | grep ' PersistentVolume$'
persistentvolumes                 pv           v1
false        PersistentVolume
```

Edite o arquivo YAML que descreve o ClusterRole objetivado, com o conteúdo que se segue.

```
 1 kind: ClusterRole
 2 apiVersion: rbac.authorization.k8s.io/v1
 3 metadata:
 4   name: storage-admin
 5 rules:
 6 - apiGroups: [""]
 7   resources: ["persistentvolumes"]
 8   verbs: ["*"]
 9 - apiGroups: ["storage.k8s.io"]
10   resources: ["*"]
11   verbs: ["*"]
```

A seguir, crie o ClusterRole via `kubectl create`.

```
# kubectl create -f storage-admin.yaml
clusterrole.rbac.authorization.k8s.io/storage-admin created
```

Agora, para o ClusterRoleBinding:

```
 1 apiVersion: rbac.authorization.k8s.io/v1
 2 kind: ClusterRoleBinding
 3 metadata:
 4   name: storage-admin-mario
 5 roleRef:
 6   apiGroup: rbac.authorization.k8s.io
 7   kind: ClusterRole
 8   name: storage-admin
 9 subjects:
10 - apiGroup: rbac.authorization.k8s.io
11   kind: User
12   name: mario
```

```
# kubectl create -f storage-admin-mario.yaml
clusterrolebinding.rbac.authorization.k8s.io/storage-admin-mario created
```

Vamos testar! Execute alguma ação sobre os objetos delineados no ClusterRole — por exemplo, `csinodes`:

```
# kubectl get csinodes --as mario
NAME          DRIVERS   AGE
s2-master-1   0         4d19h
s2-node-1     0         4d19h
```

7. Antes de continuar, para manter a integridade do *cluster*, remova todos os elementos criados durante os passos (e) e (f) desta atividade.

```
# kubectl delete clusterrole node-monitor               ; \
  kubectl delete clusterrolebinding node-monitor-mario  ; \
  kubectl delete clusterrole storage-admin              ; \
  kubectl delete clusterrolebinding storage-admin-mario
clusterrole.rbac.authorization.k8s.io "node-monitor" deleted
clusterrolebinding.rbac.authorization.k8s.io "node-monitor-mario" deleted
clusterrole.rbac.authorization.k8s.io "storage-admin" deleted
clusterrolebinding.rbac.authorization.k8s.io "storage-admin-mario" deleted
```

6) Segurança de imagens

STOPHERE

6.1) Deployment de *registry* privado

O uso de um *registry* privado permite à uma organização manter suas imagens de container em um ambiente seguro e controlado. Através de um *registry* privado é possível ter maior controle sobre a gestão, permissionamento e backup de imagens produzidas para uso interno.

Vamos, nesta atividade, fazer o deployment de um *registry* privado usando a imagem https://hub.docker.com/_/registry . Adicionalmente, iremos configurar criptografia TLS e autenticação para maior segurança no uso do ambiente.

1. Garanta que você está logado na máquina `s2-master-1`, como o usuário `root`.

```
# hostname ; whoami
s2-master-1
root
```

2. Crie um diretório de trabalho onde os artefatos do *registry* privado serão criados. A seguir, entre nesse diretório.

```
# mkdir -p ~/registry/{images,certs,auth}
```

```
# cd ~/registry
```

3. Vamos, agora, criar o par de chaves assimétricas usados para conexão TLS com o *registry* privado. Note a configuração do *hostname* do *registry* no comando abaixo, `registry.contorq.com`:

```
<strong># openssl req \
  -newkey rsa:4096 -nodes -sha256 -keyout certs/domain.key \
  -subj "/C=BR/ST=DF/L=Brasilia/O=Contorq/OU=IT/CN=registry.contorq.com" \
  -x509 -days 365 -out certs/domain.crt \
  -extensions EXT -config <( \
   printf "[dn]\nCN=registry.contorq.com\n[req]\ndistinguished_name =
dn\n[EXT]\nsubjectAltName=DNS:registry.contorq.com\nkeyUsage=digitalSignature\nexte
ndedKeyUsage=serverAuth")</strong>
```

4. Para que *daemon* Docker possa operar com um *registry* utilizando um certificado auto-assinado (como o criado no passo anterior) é necessário copiar sua chave pública para o diretório `/etc/docker/certs.d`. Vamos fazer isso para o *host* `s2-master-1`:

```
# mkdir -p /etc/docker/certs.d/registry.contorq.com:30500
```

```
# cp certs/domain.crt /etc/docker/certs.d/registry.contorq.com:30500/ca.crt
```

Como o serviço será acessado por nome e fora do contexto do Kubernetes (via *daemon* Docker, de fato), vamos configurar o *hostname* `registry.contorq.com` estaticamente via arquivo `/etc/hosts`:

```
# echo '192.168.68.20 registry.contorq.com' >> /etc/hosts
```

Os mesmos passos devem ser realizados para o *host* `s2-node-1`. Vamos lá:

```
# sudo -u vagrant scp -i /home/vagrant/.ssh/tmpkey certs/domain.crt vagrant@s2-
node-1:~/ca.crt
domain.crt                                                    100% 2041
519.8KB/s   00:00
```

```
<strong># sudo -u vagrant ssh -i /home/vagrant/.ssh/tmpkey s2-node-1 /bin/bash <<
EOF
sudo mkdir -p /etc/docker/certs.d/registry.contorq.com:30500
sudo mv /home/vagrant/ca.crt /etc/docker/certs.d/registry.contorq.com:30500
sudo bash -c 'cat << EOS >> /etc/hosts
192.168.68.20 registry.contorq.com
EOS'
EOF</strong>
```

5. Iremos utilizar autenticação HTTP `BASIC` para validar acesso ao *registry* privado — de fato, via arquivo `.htpasswd`. Vamos criar um arquivo com um único usuário `stanley`, com senha igual a `swordfish`:

```
# docker run \
  --entrypoint htpasswd \
  registry:2.7.0 -Bbn stanley swordfish > auth/htpasswd
```

   Verique seu conteúdo antes de prosseguir:

```
# cat auth/htpasswd
stanley:$2y$05$.zeHA1z2hxhv6kUoeIcKOee/oQrQ7mBr34TAhQ.TE5wPSIGx4MPR6
```

6. Tudo pronto! Vamos fazer o deployment da aplicação seguindo as instruções em https://docs.docker.com/registry/deploying/ , fazendo adaptações para um ambiente Kubernetes (e não Docker, puramente).

   Crie o arquivo `/root/registry/registry.yaml` com o conteúdo que se segue:

```
 1 apiVersion: apps/v1
 2 kind: Deployment
 3 metadata:
 4   labels:
 5     app: registry
 6   name: registry
 7 spec:
 8   replicas: 1
 9   selector:
10     matchLabels:
11       app: registry
12   template:
13     metadata:
14       labels:
15         app: registry
16     spec:
17       nodeName: s2-master-1
18       tolerations:
19       - key: "node-role.kubernetes.io/master"
20         operator: "Exists"
21         effect: "NoSchedule"
22       containers:
23       - image: registry:2.7.0
24         name: registry
25         env:
26         - name: REGISTRY_AUTH
27           value: "htpasswd"
28         - name: REGISTRY_AUTH_HTPASSWD_REALM
29           value: "Contorq Registry"
```

```
        - name: REGISTRY_AUTH_HTPASSWD_PATH
          value: /auth/htpasswd
        - name: REGISTRY_HTTP_TLS_CERTIFICATE
          value: /certs/domain.crt
        - name: REGISTRY_HTTP_TLS_KEY
          value: /certs/domain.key
        ports:
        - name: registry-port
          containerPort: 5000
        volumeMounts:
        - name: auth
          mountPath: /auth
        - name: certs
          mountPath: /certs
        - name: images
          mountPath: /var/lib/registry
      volumes:
        - name: auth
          hostPath:
            path: /root/registry/auth
        - name: certs
          hostPath:
            path: /root/registry/certs
        - name: images
          hostPath:
            path: /root/registry/images
---
apiVersion: v1
kind: Service
metadata:
  name: registry
spec:
  ports:
  - port: 5000
    protocol: TCP
    targetPort: 5000
    nodePort: 30500
  selector:
    app: registry
  type: NodePort
```

A seguir, faça o deployment via `kubectl apply`:

```
# kubectl apply -f registry.yaml
deployment.apps/registry created
service/registry created
```

7. Verifique se o deployment funcionou a contento:

```
# kubectl get deployment registry
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
registry   1/1     1            1           29m
```

```
# kubectl get svc registry
NAME       TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
registry   NodePort   10.111.173.190   <none>        5000:30500/TCP   30m
```

Observe que, devido ao uso do atributo `nodeName` e à tolerância aplicada ao container, o pod `registry` está executando no *node* `s2-master-1`:

```
# kubectl get pod -l app=registry -o custom-
columns=NAME:.metadata.name,NODE:.spec.nodeName
NAME                        NODE
registry-69b58fcf58-g9vs6   s2-master-1
```

6.2) Fazendo o upload de imagens para um *registry* privado

1. Antes de utilizar o *registry* para criar pods e deployments devemos, evidentemente, fazer o upload de imagens para esse *registry*.

   Primeiro, garanta que você está logado na máquina `s2-master-1`, como o usuário `root`. Usando o comando `docker`, faça o download da imagem `fbscarel/myapp-color` (não execute um container, apenas realize o download).

   ▼ *Visualizar resposta*

   Vamos garantir que estamos na máquina, e com o usuário correto:

   ```
   # hostname ; whoami
   s2-master-1
   root
   ```

   Para fazer o download da imagem, executamos `docker pull`.

   ```
   # docker pull fbscarel/myapp-color
   Using default tag: latest
   latest: Pulling from fbscarel/myapp-color

   (...)

   Status: Downloaded newer image for fbscarel/myapp-color:latest
   docker.io/fbscarel/myapp-color:latest
   ```

2. Usando o comando `docker`, aplique uma *tag* à imagem recém-baixada, utilizando o *registry* privado `registry.contorq.com:30500`.

   ▼ *Visualizar resposta*

   ```
   # docker tag fbscarel/myapp-color:latest registry.contorq.com:30500/myapp-color
   ```

3. Usando o comando `docker`, realize o login no *registry* privado. Atente-se ao usuário e senha configurados durante a atividade anterior.

   ▼ *Visualizar resposta*

   ```
   # docker login registry.contorq.com:30500
   Username: stanley
   Password: swordfish

   (...)

   Login Succeeded
   ```

4. Faça o upload da imagem cuja *tag* foi aplicada no passo (b) para o *registry* privado.

   ▼ *Visualizar resposta*

   ```
   # docker push registry.contorq.com:30500/myapp-color
   The push refers to repository [registry.contorq.com:30500/myapp-color]

   (...)

   sha256:0e8179db14378826d62d1934fe56d23f78d3f4c18cd97ae1642cb56eb553bf23 size:
   1788
   ```

5. Verifique que a imagem foi recebida com sucesso, usando o comando `curl` para consultar o catálogo do *registry*. Sugestão: verifique a funcionalidade das *flags* `--insecure` e `--user`.

   ▼ *Visualizar resposta*

   Basta utilizar as *flags* informadas da seguinte forma, ao invocar o `curl`:

   ```
   # curl --insecure -u stanley:swordfish
   https://registry.contorq.com:30500/v2/_catalog
   {"repositories":["myapp-color"]}
   ```

6.3) Utilizando um *registry* privado

1. Crie um deployment com o nome `private-app` e 2 réplicas utilizando a imagem armazenada no *registry* privado durante a atividade anterior.

Verifique o estado dos pods, e determine: a aplicação está operacional?

▼ *Visualizar resposta*

Primeiro, vamos criar o deployment:

```
# kubectl create deploy private-app --image=registry.contorq.com:30500/myapp
-color --replicas=2
deployment.apps/private-app created
```

Qual seria o estado dos pods? Chequemos:

```
# kubectl describe pod -l app=private-app | tail -n5
  Normal   Pulling    27s (x3 over 69s)  kubelet            Pulling image
"registry.contorq.com:30500/myapp-color"
  Warning  Failed     27s (x3 over 69s)  kubelet            Failed to pull image
"registry.contorq.com:30500/myapp-color": rpc error: code = Unknown desc = Error
response from daemon: Get https://registry.contorq.com:30500/v2/myapp-color/
manifests/latest: no basic auth credentials
  Warning  Failed     27s (x3 over 69s)  kubelet            Error: ErrImagePull
  Normal   BackOff    1s (x4 over 69s)   kubelet            Back-off pulling
image "registry.contorq.com:30500/myapp-color"
  Warning  Failed     1s (x4 over 69s)   kubelet            Error:
ImagePullBackOff
```

Não deu certo... ao verificar a razão, fica claro o motivo: não existem credenciais de acesso informadas para conexão com o *registry* privado `registry.contorq.com`. Vamos corrigir isso.

2. Crie um objeto do tipo Secret, `registry-secret`, contendo as credenciais requetidas para acessar o *registry* privado. Verifique o funcionamento de sua configuração.

▼ *Visualizar resposta*

Note que o tipo do Secret deve ser `docker-registry`, como no comando mostrado abaixo.

```
# kubectl create secret docker-registry registry-secret --docker
-server=https://registry.contorq.com:30500 --docker-username=stanley --docker
-password=swordfish
secret/registry-secret created
```

Verifique seu conteúdo:

```
# kubectl describe secrets registry-secret
Name:         registry-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>
```

```
Type:  kubernetes.io/dockerconfigjson

Data
====
.dockerconfigjson:  128 bytes
```

3. Edite o deployment `private-app` e configure-o para utilizar as credenciais de login do *registry* privado `registry.contorq.com:30500`.

   A seguir, verifique que os pods do deployment estão operacionais. Caso afirmativo, crie um serviço do tipo `NodePort` que exponha a aplicação e verifique seu funcionamento usando os comandos `curl` ou `wget`.

   ▼ *Visualizar resposta*

   Invoque a edição do deployment:

   ```
   # kubectl edit deploy private-app
   deployment.apps/private-app edited
   ```

   Na seção `.spec.template.spec.imagePullSecrets`, insira o nome do Secret criado anteriormente.

   ```
   imagePullSecrets:
   - name: registry-secret
   ```

   Imediatamente, o deployment é atualizado e seus pods, recriados. Em pouco tempo, eles ficam operacionais:

   ```
   # kubectl get deploy private-app
   NAME          READY   UP-TO-DATE   AVAILABLE   AGE
   private-app   2/2     2            2           11m
   ```

   ```
   # kubectl describe pod -l app=private-app | tail -n3
     Normal  Pulled     51s   kubelet            Successfully pulled image
   "registry.contorq.com:30500/myapp-color" in 53.084356ms
     Normal  Created    51s   kubelet            Created container myapp-color
     Normal  Started    51s   kubelet            Started container myapp-color
   ```

   Para testar o deployment, basta criar o serviço do tipo `NodePort` e usar o `curl` para acessá-lo.

   ```
   # kubectl create svc nodeport private-app --tcp 80 --node-port=30080
   service/private-app created
   ```

```
# curl http://localhost:30080/color
Hostname: private-app-f4486fb8d-lfnlg ; Color: blue
```

4. Antes de prosseguir, remova os objetos criados durante essas atividades, de forma a reduzir o uso de recursos do *cluster*.

```
# kubectl delete pod,rs,deploy --all         ; \
  kubectl delete svc -l provider!=kubernetes ; \
  kubectl delete secret registry-secret
pod "private-app-f4486fb8d-fnfz4" deleted
pod "private-app-f4486fb8d-lfnlg" deleted

(...)

service "registry" deleted
secret "registry-secret" deleted
```

7) Contextos de segurança

7.1) Alterando usuário e grupo efetivos

1. Crie o pod `aurora` com a imagem `busybox`, executando o comando `sleep 600`. Feito isso, responda: qual é o usuário utilizado para rodar esse processo?

   ▼ *Visualizar resposta*

   Criar o pod é bastante fácil:

   ```
   # kubectl run aurora --image=busybox -- sleep 600
   pod/aurora created
   ```

   Para verificar o usuário executando o comando, podemos usar o `ps`:

   ```
   # kubectl exec -it aurora -- ps auxmw | grep [s]leep
       1 root      0:00 sleep 600
   ```

2. Edite as configurações do pod, de forma que o processo execute com o *user ID* 1000. O que ocorre?

   Em caso de erro, exporte e edite a configuração do pod, e em seguida recrie-o.

   ▼ *Visualizar resposta*

   Ao tentar editar o pod, encontramos um erro, como já aconteceu em outras atividades.

```
# kubectl edit pod aurora
```

```
# pods "aurora" was not valid:
# * spec: Forbidden: pod updates may not change fields other than
`spec.containers[*].image`, `spec.initContainers[*].image`,
`spec.activeDeadlineSeconds` or `spec.tolerations` (only additions to existing
tolerations)
```

Vamos recriar o pod via arquivo YAML. Use o conteúdo abaixo.

```
 1 apiVersion: v1
 2 kind: Pod
 3 metadata:
 4   name: aurora
 5 spec:
 6   securityContext:
 7     runAsUser: 1000
 8   containers:
 9   - args:
10     - sleep
11     - "600"
12     image: busybox
13     name: aurora
```

Em um *one-liner*, removemos o pod e o recriamos via YAML.

```
# kubectl delete pod aurora ; kubectl create -f aurora.yaml
pod "aurora" deleted
pod/aurora created
```

Ao verificar o usuário que invocou o comando `sleep`, fica claro que a configuração surtiu o efeito esperado.

```
# kubectl exec -it aurora -- ps auxmw | grep [s]leep
    1 1000      0:00 sleep 600
```

3. Por que o *user ID* do processo é mostrado em formato numérico?

▼ *Visualizar resposta*

Para verificar os usuários mapeados no sistema podemos consultar o arquivo `/etc/passwd`. Note que nenhum dos usuários nesse arquivo possui o UID 1000.

```
# kubectl exec -it aurora -- cat /etc/passwd
root:x:0:0:root:/root:/bin/sh
```

```
daemon:x:1:1:daemon:/usr/sbin:/bin/false
bin:x:2:2:bin:/bin:/bin/false
sys:x:3:3:sys:/dev:/bin/false
sync:x:4:100:sync:/bin:/bin/sync
mail:x:8:8:mail:/var/spool/mail:/bin/false
www-data:x:33:33:www-data:/var/www:/bin/false
operator:x:37:37:Operator:/var:/bin/false
nobody:x:65534:65534:nobody:/home:/bin/false
```

De fato, do ponto de vista do pod, é como se esse usuário "não existisse":

```
# kubectl exec -it aurora -- whoami
whoami: unknown uid 1000
```

Mas, de fato, há um usuário com esse UID dentro do *host*: o `vagrant`, como verificado pelo comando abaixo.

```
# getent passwd vagrant
vagrant:x:1000:1000:vagrant,,,:/home/vagrant:/bin/bash
```

O mapeamento de UIDs e execução de pods não-privilegiados é um tema complexo, que pode ser tratado de diferentes formas. A distribuição Kubernetes OpenShift, da Red Hat, possui uma abordagem interessante para o problema: o uso de UIDs aleatórios e arbitrariamente assinalados. Este artigo (https://access.redhat.com/articles/4859371) é bastante interessante, e provê informações úteis também para instalações "puras" do Kubernetes.

4. Será possível executar dois containers em um mesmo pod com usuários diferentes? Vamos testar.

   Crie um multi-container pod `sleepers` com dois containers, `slp1` e `slp2`. Ambos devem usar a imagem `busybox` e executar o comando `sleep 600`. Faça com que o *user ID* padrão (i.e., aplicável a todos os containers do pod) seja 2000. Para um dos containers, altere o *user ID* para 3000.

   Valide o funcionamento de sua configuração.

   ▼ *Visualizar resposta*

   A criação do multi-container pod em si é relativamente simples. É importante notar, porém, que o atributo `securityContext` pode ser aplicado tanto na seção `.spec` quanto `.spec.containers` — e, ainda, que a hierarquia dessas diretivas é respeitada.

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: sleepers
5 spec:
6   securityContext:
7     runAsUser: 2000
```

```
 8  containers:
 9  -  image: busybox
10     name: slp1
11     command: ["sleep", "600"]
12     securityContext:
13      runAsUser: 3000
14
15  -  image: busybox
16     name: slp2
17     command: ["sleep", "600"]
```

Vamos criar o pod e verificar:

```
# kubectl create -f sleepers.yaml
pod/sleepers created
```

Note que, no caso do container `slp1`, o UID utilizado é o 3000...

```
# kubectl exec -it sleepers -c slp1 -- ps auxmw | grep '[s]leep'
    1 3000      0:00 sleep 600
```

Já no caso do container `slp2`, ele é o 2000.

```
# kubectl exec -it sleepers -c slp2 -- ps auxmw | grep '[s]leep'
    1 2000      0:00 sleep 600
```

5. Qual é o grupo efetivo dos containers do pod criado no passo anterior? Ele é o mesmo para ambos?

   ▼ *Visualizar resposta*

   O grupo efetivo pode ser determinado com o comando `id -g`. Veja:

   ```
   # kubectl exec -it sleepers -c slp1 -- id -g
   0
   ```

   ```
   # kubectl exec -it sleepers -c slp2 -- id -g
   0
   ```

6. Vamos resolver isso: crie o pod `slpgroup` usando a imagem `busybox`, e executando o comando `sleep 600`. Faça com que o usuário efetivo desse pod seja 1000, e que ele opere com o grupo primário 1000 e grupos suplementares 1001 e 1002.

   Valide o funcionamento de sua configuração.

▼ *Visualizar resposta*

Para atingir os objetivos delineados basta usar os atributos `runAsGroup` e `supplementalGroups`, da seguinte forma:

```
apiVersion: v1
kind: Pod
metadata:
  name: slpgroup
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 1000
    supplementalGroups: [1001, 1002]
  containers:
  -  image: busybox
     name: slpgroup
     command: ["sleep", "600"]
```

A seguir, cria-se o pod:

```
# kubectl apply -f slpgroup.yaml
pod/slpgroup created
```

E com o comando `id`, verificamos o funcionamento da configuração.

```
# kubectl exec -it slpgroup -- id
uid=1000 gid=1000 groups=1001,1002
```

7.2) Habilitando *capabilities*

1. Tente alterar a data e hora correntes do pod `aurora` para 21 de outubro de 2015, às 7:28 da manhã. O que ocorre?

   ▼ *Visualizar resposta*

   A operação não é autorizada. Veja:

   ```
   # kubectl exec -it aurora -- date -s '2015-10-21 07:08'
   date: can't set date: Operation not permitted
   ```

   Para fazer isso, necessitamos da *capability* `CAP_SYS_TIME` adicionada ao container.

2. Adicione a *capability* `SYS_TIME` ao pod. Recrie-o se necessário.

   ▼ *Visualizar resposta*

   Basta utilizar um arquivo YAML com o seguinte conteúdo:

```
apiVersion: v1
kind: Pod
metadata:
  name: aurora
spec:
  containers:
  - args:
    - sleep
    - "600"
    image: busybox
    name: aurora
    securityContext:
      capabilities:
        add: ["SYS_TIME"]
```

```
# kubectl delete pod aurora ; kubectl create -f aurora.yaml
pod "aurora" deleted
pod/aurora created
```

3. Tente novamente alterar a data com o comando utilizado no passo (a) desta atividade.

   ▼ *Visualizar resposta*

   Vamos ver:

   ```
   # kubectl exec -it aurora -- date -s '2015-10-21 07:08'
   Wed Oct 21 07:08:00 UTC 2015
   ```

   A data é efetivamente alterada. Muito bom!

4. A lista de *capabilities* disponíveis em um sistema Linux é bastante extensa. Qual recurso pode ser consultado para visualizar a lista completa, bem como explicações sobre cada um desses *capabilities*?

   ▼ *Visualizar resposta*

   Além da consulta de recursos na Internet, é também possível visualizar a lista completa de *capabilities* acessando a página de manual:

   ```
   # man 7 capabilities
   ```

5. Antes de prosseguir, remova os objetos criados durante esta atividade, de forma a reduzir o uso de recursos do *cluster*.

   ```
   # kubectl delete pod --all
   pod "aurora" deleted
   pod "sleepers" deleted
   ```

```
pod "slpgroup" deleted
```

8) Políticas de rede

Network Policies, ou políticas de rede, são construtos que permitem controlar com quais entidades de rede um pod poderá (ou não) comunicar-se. Por "entidades" nos referimos a outros pods, namespaces com o qual o pod pode conversar ou faixas de endereçamento IP.

Nesta atividade iremos fazer o deployment de uma aplicação típica, com a seguinte topologia:

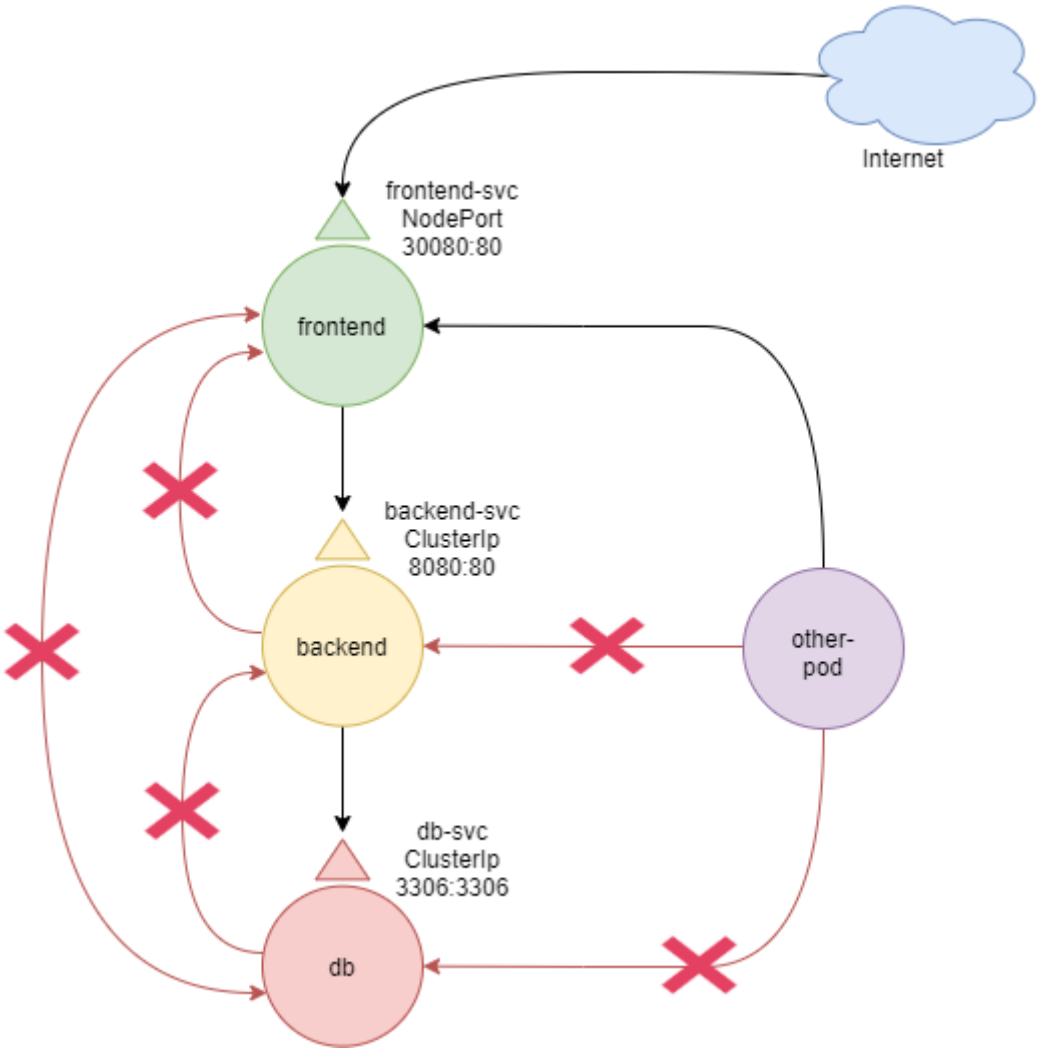

Figura 1. Topologia da aplicação

Note que a aplicação possui um caminho esperado de comunicação:

- O pod `frontend` deve receber requisições vindas de qualquer origem, através do serviço do tipo `NodePort` `frontend-svc`, expondo a porta 80 do pod e a porta 30080 dos *nodes*.

  O pod `frontend` deve, ainda, poder comunicar-se com o pod `backend`, descrito abaixo.

- O pod `backend` deve receber requisições vindas do pod `frontend`, através do serviço do tipo `ClusterIP` `backend-svc`, expondo a porta 80 do pod.

  O pod `backend` deve, ainda, poder comunicar-se com o pod `db`, descrito abaixo.

- O pod `db` deve receber requisições vindas do pod `backend`, através do serviço do tipo `ClusterIP` `db-svc`, expondo a porta 3306 do pod e a mesma porta, externamente.
- Todas as demais vias de comunicação não devem ser autorizadas; isso será garantido através da implementação de *Network Policies*. Note que iremos também criar um pod não-relacionado à aplicação, `other-pod`, para validar que as políticas estão funcionando como esperado.

O pod `db` deve utilizar a imagem `mysql:5.6`, com senha de acesso à base para o usuário `root` configurada como `password`. Todos os demais pods deverão utilizar a imagem `nginx:alpine`.

Todos os objetos acima mencionados devem ser criados em um namespace dedicado, com o nome `myapp`.

1. Vamos começar do princípio: crie todos os pods e serviços descritos acima, via comandos imperativos ou arquivos YAML (como preferir).

   Não se preocupe com as *Network Policies* ainda: elas serão implementas nos passos a seguir.

   ▼ *Visualizar resposta*

   Pode-se utilizar um único arquivo YAML para criar todos os objetos especificados no enunciado. Para isso, utilizamos o divisor `---`, que indica o começo de um novo documento YAML.

```yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: myapp
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: myapp
spec:
  containers:
  - image: nginx:alpine
    name: frontend
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: backend
  name: backend
  namespace: myapp
spec:
  containers:
  - image: nginx:alpine
```

```
    name: backend
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: db
  name: db
  namespace: myapp
spec:
  containers:
  - env:
    - name: MYSQL_ROOT_PASSWORD
      value: password
    image: mysql:5.6
    name: db
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: other-pod
  name: other-pod
  namespace: myapp
spec:
  containers:
  - image: nginx:alpine
    name: other-pod
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: frontend-svc
  name: frontend-svc
  namespace: myapp
spec:
  ports:
  - nodePort: 30080
    port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: frontend
  type: NodePort
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: backend-svc
```

```
  name: backend-svc
  namespace: myapp
spec:
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: backend
  type: ClusterIP
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: db-svc
  name: db-svc
  namespace: myapp
spec:
  ports:
  - port: 3306
    targetPort: 3306
    protocol: TCP
  selector:
    app: db
---
```

Ao invocar a criação, note que todos os objetos são criados em sequência.

```
# kubectl apply -f app-topology.yaml
namespace/myapp created
pod/frontend created
pod/backend created
pod/db created
pod/other-pod created
service/frontend-svc created
service/backend-svc created
service/db-svc created
```

Podemos verificar a criação dos pods e serviços, para garantir seu funcionamento.

```
# kubectl -n myapp get pod
NAME        READY   STATUS    RESTARTS   AGE
backend     1/1     Running   0          41s
db          1/1     Running   0          41s
frontend    1/1     Running   0          41s
other-pod   1/1     Running   0          41s
```

```
# kubectl -n myapp get svc
NAME           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
backend-svc    ClusterIP   10.109.223.133   <none>        80/TCP       55s
db-svc         ClusterIP   10.102.249.150   <none>        3306/TCP       55s
frontend-svc   NodePort    10.104.180.205   <none>        80:30080/TCP   55s
```

2. Teste a conectividade entre os pods; no momento, como não há nenhuma *Network Policy* implementada, não haverá qualquer restrição de comunicação entre eles.

   Como o teste de conectividade envolve verificar o acesso à uma porta TCP específica, e não simplesmente acessibilidade ao pod via pacotes ICMP (como os enviados pelo comando `ping`, por exemplo) temos que usar uma ferramenta apropriada para esse cenário.

   O `netcat`, ou `nc`, é ideal para essas situações — e, ainda melhor, está disponível na imagem `nginx:alpine`. Pesquise como utilizar o `nc` para validar a conectividade entre *hosts*: tenha especial atenção às opções `-w` (que define o número de segundos até o *timeout* da conexão) e `-z` (*zero I/O mode*, que apenas verifica conectividade, encerrando em seguida).

   ▼ *Visualizar resposta*

   Vamos acessar um dos pods interativamente; por exemplo, `other-pod`:

   ```
   # kubectl -n myapp exec -it other-pod -- /bin/sh
   ```

   Dento dele, verificamos seu *hostname* e usuário efetivo.

   ```
   / # hostname ; whoami
   other-pod
   root
   ```

   A seguir, utilizamos o `nc` para nos conectarmos a um dos serviços publicados pelos demais pods; por exemplo, `db-svc`. Note que iremos utilizar o valor de retorno do comando em um bloco `if` para definir se houve sucesso ou falha na conexão. Ao utilizarmos a porta correta, 3306, obtemos sucesso:

   ```
   / # if nc -w1 -z db-svc 3306; then echo "success"; else echo "failure"; fi
   success
   ```

   E com uma porta incorreta, 3307, falha:

   ```
   / # if nc -w1 -z db-svc 3307; then echo "success"; else echo "failure"; fi
   failure
   ```

3. Com a conectividade entre pods validada, é hora de implementar as *Network Policies*. Vamos primeiramente fazer com que todo o tráfego entre pods, de saída e entrada, seja

bloqueado — algo equivalente a uma política *default deny* em um firewall.

Crie uma política com o nome `deny-netpol` que bloqueie todo o tráfego originado e oriundo aos pods do namespace `myapp`. Faça uma exceção ao tráfego egresso para as portas TCP/53 e UDP/53, necessárias para resolução de nomes dentro do contexto do *cluster*.

A seguir, verifique se sua configuração surtiu o efeito esperado: teste se a resolução de nomes via DNS permanece funcional (via `nslookup`), e se os demais acessos são negados.

▼ *Visualizar resposta*

Uma política *default deny* como especificada pelo enunciado pode ser definida da seguinte forma. Note a exceção ao tráfego egresso para as portas TCP/53 e UDP/53.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-netpol
  namespace: myapp
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
  egress:
  - to:
    ports:
    - protocol: TCP
      port: 53
    - protocol: UDP
      port: 53
```

Criamos a política com `kubectl apply` (ou `create`).

```
# kubectl apply -f deny-netpol.yaml
networkpolicy.networking.k8s.io/deny-netpol created
```

Vejamos se a resolução de nomes permanece funcional:

```
# kubectl -n myapp exec -it other-pod -- /bin/sh -c 'nslookup db-
svc.myapp.svc.cluster.local'
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:   db-svc.myapp.svc.cluster.local
Address: 10.102.249.150
```

Perfeito! E quanto à conectividade?

```
# kubectl -n myapp exec -it other-pod -- /bin/sh -c 'if nc -w1 -z db-svc 3306;
then echo "success"; else echo "failure"; fi'
failure
```

Diferentemente do que observamos no passo (b), agora o `nc` encontra falha ao conectar-se com o *host* `db`. Isso significa que a política está funcionando como esperado.

4. Vamos começar pela mais simples delas, a do pod `db`. Crie uma política com o nome `db-netpol` que implemente os controles de acesso delineados na topologia mostrada no início desta atividade.

   Os seguintes requisitos devem ser atendidos pela política:

   - O pod `backend` **deve** conseguir comunicar-se com o pod `db` através do serviço `db-svc`, na porta TCP/3306.
   - Os demais acessos devem ser negados.

   Observe que, ainda que sua política esteja corretamente configurada, a política *default deny* `deny-netpol` implementada no passo (c) irá impedir a conectividade entre os pods `backend` e `db`. Iremos solucionar isso no próximo passo.

   ▼ *Visualizar resposta*

   Primeiro, definimos a política via arquivo YAML:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-netpol
  namespace: myapp
spec:
  podSelector:
    matchLabels:
      app: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: backend
    ports:
    - protocol: TCP
      port: 3306
```

   E em seguida criamos o objeto.

```
# kubectl apply -f db-netpol.yaml
```

```
networkpolicy.networking.k8s.io/db-netpol created
```

Ainda não é possível testar se a configuração está correta, pelo motivos delineados no enunciado. Por ora, iremos prosseguir.

5. A seguir, vamos tratar do pod `backend`. Crie uma política com o nome `backend-netpol` que implemente os controles de acesso delineados na topologia mostrada no início desta atividade.

   Os seguintes requisitos devem ser atendidos pela política:

   - O pod `backend` **deve** conseguir comunicar-se com o pod `db` através do serviço `db-svc`, na porta TCP/3306.
   - O pod `frontend` **deve** conseguir comunicar-se com o pod `backend` através do serviço `backend-svc`, na porta TCP/80.
   - Os demais acessos devem ser negados.

   Após a implementação da política você pode, agora sim, validar a conectividade entre os pods `backend` e `db` através da ferramenta `nc`.

   ▼ *Visualizar resposta*

   Vamos à política:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-netpol
  namespace: myapp
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: db
    ports:
    - protocol: TCP
```

```
28       port: 3306
```

```
# kubectl apply -f backend-netpol.yaml
networkpolicy.networking.k8s.io/backend-netpol created
```

Em tudo estando correto, o pod `backend` deve conseguir conectar-se com `db` através do serviço `db-svc`. Será esse o caso?

```
# kubectl -n myapp exec -it backend -- /bin/sh -c 'if nc -w1 -z db-svc 3306; then
echo "success"; else echo "failure"; fi'
success
```

Perfeito! Isso significa que as políticas estabelecidas nos passos (d) e (e) estão corretas.

6. Agora, o pod `frontend`. Crie uma política com o nome `frontend-netpol` que implemente os controles de acesso delineados na topologia mostrada no início desta atividade.

   Os seguintes requisitos devem ser atendidos pela política:

   - Requisições vindas de qualquer origem **devem** conseguir comunicar-se com o pod `frontend` através do serviço `frontend-svc`, na porta TCP/80.
   - O pod `frontend` **deve** conseguir comunicar-se com o pod `backend` através do serviço `backend-svc`, na porta TCP/80.
   - Os demais acessos devem ser negados.

   Após a implementação da política você deve validar a conectividade de requisições externas ao pod `frontend`, bem como entre os pods `frontend` e `backend`.

   ▼ *Visualizar resposta*

   Primeiro, criamos a política:

```
 1 apiVersion: networking.k8s.io/v1
 2 kind: NetworkPolicy
 3 metadata:
 4   name: frontend-netpol
 5   namespace: myapp
 6 spec:
 7   podSelector:
 8     matchLabels:
 9       app: frontend
10   policyTypes:
11   - Ingress
12   - Egress
13   ingress:
14   - from:
15     ports:
```

```
16     - protocol: TCP
17       port: 80
18   egress:
19   - to:
20     - podSelector:
21         matchLabels:
22           app: backend
23     ports:
24     - protocol: TCP
25       port: 80
```

```
# kubectl apply -f frontend-netpol.yaml
networkpolicy.networking.k8s.io/frontend-netpol created
```

E agora, para os testes. Vamos verificar a conectividade externa com o serviço publicado pelo pod `frontend` — note que o comando abaixo não irá mostrar nenhuma saída, já que especificamos a *flag* `-s` (`--silent`) para o `curl` e a saída foi redirecionada para `/dev/null`.

```
<strong># curl -s http://localhost:30080 &> /dev/null</strong>
```

Como saber que deu certo, então? Simples: basta verificar os logs do pod `frontend`. Note que uma requisição `GET /` foi recebida:

```
# kubectl -n myapp logs frontend --tail=1
10.32.0.1 - - [11/Oct/2020:09:10:17 +0000] "GET / HTTP/1.1" 200 612 "-"
"curl/7.64.0" "-"
```

A seguir, vamos testar a conectividade entre os pods `frontend` e `backend`:

```
# kubectl -n myapp exec -it frontend -- /bin/sh -c 'if nc -w1 -z backend-svc 80;
then echo "success"; else echo "failure"; fi'
success
```

Perfeito, tudo funcionando como esperado.

7. Vamos finalizar com o pod `other-pod`. Crie uma política com o nome `other-pod-netpol` que implemente os controles de acesso delineados na topologia mostrada no início desta atividade.

   Como esse pod, diferentemente dos demais, não possui qualquer restrição de acesso, faça com que a política libere-o para conectar-se com qualquer outro pod.

   Após a implementação da política você deve validar o funcionamento das *Network Policies* implementadas anteriormente: ou seja, ainda que o pod `other-pod` possa originar conexões para quaisquer destinos, as políticas dos pods `db`, `backend` e `frontend` devem impedi-lo de acessar seus serviços.

▼ *Visualizar resposta*

A política do pod `other-pod` é bastante simples, quando comparada com as anteriores:

```
 1 apiVersion: networking.k8s.io/v1
 2 kind: NetworkPolicy
 3 metadata:
 4   name: other-pod-netpol
 5   namespace: myapp
 6 spec:
 7   podSelector:
 8     matchLabels:
 9       app: other-pod
10   policyTypes:
11   - Ingress
12   - Egress
13   ingress:
14   - {}
15   egress:
16   - {}
```

```
# kubectl apply -f other-pod-netpol.yaml
networkpolicy.networking.k8s.io/other-pod-netpol created
```

E agora, aos testes de conexão. Esperamos falha em todos eles, exceto com o pod `frontend` (cujo serviço deve receber conexões vindas de qualquer origem).

```
# kubectl -n myapp exec -it other-pod -- /bin/sh -c 'if nc -w1 -z db-svc 3306;
then echo "success"; else echo "failure"; fi'
failure
```

```
# kubectl -n myapp exec -it other-pod -- /bin/sh -c 'if nc -w1 -z backend-svc 80;
then echo "success"; else echo "failure"; fi'
failure
```

```
# kubectl -n myapp exec -it other-pod -- /bin/sh -c 'if nc -w1 -z frontend-svc
80; then echo "success"; else echo "failure"; fi'
success
```

De fato, as políticas criadas surtiram o efeito desejado.

8. Para concluir, remova o namespace `myapp` para liberar recursos do *cluster*.

```
# kubectl delete ns myapp
```

```
namespace "myapp" deleted
```

ENTREGA DA TAREFA

Para que seja considerada entregue você deve anexar a esta atividade no AVA uma imagem (nos formatos .png ou .jpg) do terminal mostrando a saída do comando `curl` ao contatar o *deployment* `private-app`, criado via imagem `myapp-color` hospedada no *registry* privado `registry.contorq.com:30500`.

Utilize como referência a saída de comando mostrada na atividade 6.6.3 (c) deste roteiro.