教程（五）jenkins 集成k8s 的认证原理与配置总结

配置流程

之前说了jenkins与k8s整合后pipeline如何使用k8s来动态创建slave节点。 这次总结一下在jenkins中如何填写与k8s的通信配置。

1. 在插件管理中安装kubernetes插件
2. 在系统配置中，添加一个云
3. 在云配置中填写kubernetes的配置，其中包括
    a. kubernetes api server的ip地址和端口号
    b. kubernetes 根证书key
    c. 如果jenkins master部署到k8s中的话只需要填写service name即可。 如果jenkins master部署在k8s外则比较麻烦，需要在k8s中为jenkins创建一个service account， 然后为这个service account配置rbac以便让jenkins有权限创建相关资源。 然后service account下的secret下复制它的token。 最后在jenkins中添加一个secret text类型的凭据，把token内容复制进去。 这样才能让jenkins与k8s通信

k8s安全机制详解

这里主要讲下上面第三步如何配置kubernetes的方法。 这里涉及到k8s的安全机制。
之前在研究的时候觉得这块内容非常好，所以在这里记录一下。

k8s有很多种安全认证机制，可以说这方面它做的还是很复杂的。这里主要涉及的是基于数据证书和service account token的认证以及rbac的鉴权机制。

x509数据证书认证

x509认证是默认开启的认证方式，
api-server启动时会指定k8s的根证书(ca)以及ca的私钥，只要是通过ca签发的客户端x509证书，则可认为是可信的客户端。
这里提一下k8s是启用https进行通信的， 而且是双向加密。 也就是说不仅仅需要客户端信任服务端的根证书(ca)来验证服务端的身份，也需要服务端信任客户端的数字证书来验证客户端的身份。

数据证书的原理

这里我先用一点时间来介绍一下数字证书的双向认证方式(尽量简单易懂)。 互联网在通信时如果使用tls或者ssl协议作为安全加密方式的话，基本认证用户信息都是采用非对称加密方式(只有身份认证这么做， 因为非对称加密性能差) 也就是我们常见的一个私钥一个公钥。
自己留一个私钥，公钥公开给所有人。 那么从自己这里发送给别人的报文只有对应的公钥才能解密，
所以如果对方用你公开的公钥来解密，解密成功，那就能证明你是你， 如果解密不成功，那这个请求就是有风险的。
而如果对方要发消息给自己， 对方就要拿着公开的公钥加密它的小心然后发送过来， 如果你自己可以用私钥解密成功，
就说明这个报文没有经过篡改，如果解密失败那就说这个报文被人篡改过。 这就是非对称加密，
私钥和公钥都能加密数据，但是只有私钥才能解密公钥加密的数据，也只有公钥才能解密私钥加密的数据。

所以数据证书就是基于非对称加密来的。 它包含公钥、名称以及证书授权中心的数字签名。
客户端下载服务端的数字证书并选择信任这个证书。 那么就可以正常通信， 如果没有服务端的证书会怎么样。
还记得我们在浏览器中访问带有https的网站会蹦出来的的安全提示信息么 ----- 这是一个不受信任的网址，是否继续访问。
因为我们没有下载服务端证书并信任它， 所以我们手中没有对应的公钥能解密，所以这就是个不安全的访问(无法确认对方的身份)。
我们可以选择继续访问， 当然这时候会有人问既然没有公钥解密那是怎么能选择继续访问的呢，
我们刚才也说了，非对称算法只用来做身份的验证，它不会加密真正的报文数据。报文加密是对称算法做的事这里不细讲了。
所以我们做接口测试的时候如果遇到https的请求，需要下载数据证书并用代码加载， 或者就直接用代码的方式忽略这个身份认证的风险。

k8s的数字证书认证

k8s在部署时会自己创建一个CA（证书颁发）并产生CA的私钥和数字证书。
k8s其他服务的服务也都会生成自己的私钥并申请给CA，让CA办法服务自己的证书。 所以客户端，比如我们这次将的jenkins要与k8s整合，要填写的证书的key的配置。其实是CA的证书(也就是跟证书) 而不是api-server的证书。 因为证书的机制是你信任了CA的证书，
那么也就顺带新人了CA颁发的所有其他的证书了。 那么在k8s中怎么查看根证书信息呢？
如果你能得到kubeconfig文件的话，那么它就在kubeconfig文件中的cluster信息里。 如果你不知道kubeconfig文件在哪，
可以使用命令kubectl config view --raw 来查看.

```
]root@m7-prod-ssd001 [12:26:21] {~}
-->$ k config view --raw
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUR2akNDQXFhZ0F3SUJBZ0lVWDg0SFB0UzgrUnRiTTV6UXN5ZTFIVE9kVTQwd0RRWUpLb1pJaHZjTkFRRUwKQlFBd1pURUxNQWtHQTFVRUJ
hTUNRMDR4RURBT0JnTlZCQWdUQjBKbGFVcHBibWN4RURBT0JnTlZCQWNUQjBKbAphVXBwYm1jeEREQUtCZ05WQkFvVVEEyczRjekVQTUEwR0ExVUVDeE1HVTNsemRHVnRNUk13RVFZRFZRUURFd3ByZmRXSmxjbTVsZEdWek1CNFhEVEU1TU
hFeE1EQTNNemN3TUZvWERUSTBNREV3T1RBM016Y3dNRm93WlRFTE1Ba0cKQTFVRUJoTUNRMDR4RURBT0JnTlZCQWdUQjBKbGFVcHBibWN4RURBT0JnTlZCQWNUQjBKbGFVcHBibWN4RERBSwpCZ05WQkFvVEEyczRjekVQTUEwR0ExVUVDe
1HVTNsemRHVnRNUk13RVFZRFZRUURFd3ByZFdkbGNtNWxkR1Z6Ck1JSUJJakFOQmdrcWhraUc5dzBCQVFFRkFBT0NBUThBTUlJQkNnS0NBUUVBclUrTWVJaTlUcnN3dWNRSHVRY1cKVmZxaTZ4WXQ3S0UxQmhubjI4aVFNeEI3TGZDamdk
UowMGdPQ3RJUFBYYmNYK0dTU1VCRE5FeEcvNmVEbmFkZgpyY2NOeU5IY2p0anhCbmdZamYwRmNuTWMwK2ZBcFg0cGUxd1lSUzhscXNrWVZVdzNOT0ZvR2VqREx4QZWxEanR4Clg1dUFjeUpaMEtuM25WOFFvWnNIaGJ6enk4Q0hOVG5Zb0Z
WmJrVGxBb0YxWGl1ajd4Q2FRa2ZIMTJxdklNQUYKUDNUcVR1eUtYZ1pkTktUdC8rZXdiNWJRRzVnQlZJTldsUXNER0xrYUdjZWNGMVlJWDRSb1E3TjZhdWxPdVpHQQpMWlkyOVJZcUJUSEpaanBIVkhoMXJpenhGeWROdDM0MTJtVm1Cem
FSHZIV0ErcGh1Mnkzc2FCOHBORktyMFl4CnJRSURBUUFCbzJZd1pEQU9CZ05WSFE4QkFmOEVCQU1DQVFZd0VnWURWUjBUQVFIL0JBZ3dCZ0VCL3dJQkFqQQMQKQmdOVkhRNEVGZ1FVc0pPdmI1aEFWNSt1N3h4RFZMaVR0bWtvRU5Fd0h3W
RWUjBqQkJnd0ZvQVVzSk92YjVoQQpWNSt1N3h4RFZMaVR0bWtvRU5Fd0RRWUpLb1pJaHZjTkFRUxCUUFEZ2dFQkFKS25FdUNIT0c5SU8xZkQyVTZzClFhbEp2UnMyVi9Vd3lCMm1FUzBkQ3gyWk9uYjYxM0VIdGJQNUhYU290ZGE2dE0v
GlaYWRnLzJSWlowRHdtbDMKcGUxVDB1WEdjN1FPQGRvNUVIWFBtSHZBREUyUVRmTZ50RUJ5Z0RJdXR1M0M2aE52b01WM3RkWi82M21GOWFXQwp0MjBWVlkyQ0JvbkFyYjJueUVwS1VPMUdSbUlwL1VSNm1LejFibHNZaW5aUm5nanZsRWJ
TWIwM3NZQTFIeno3CnEyazV6dnE1WTNla0cwV20vRFgwNmRCdmRzQ1JLRFdQaDZWZUMvUjBtU25QSEVYcFNkaVBBeCsrT1JKOXlXd2cKQjRiZGxQT2lzbWlWZVk1TS94SVBtMFJ4S09GdDdkNEUzZG1oa0t0bzZsT1hReDEzVmpQRmdNWU
jNDcwRGZ3ZQpFZGc9Ci0tLS0tRU5EIENFUlRJRklDQVRFLS0tLS0K
    server: https://172.27.128.190:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: admin
  name: kubernetes
current-context: kubernetes
kind: Config
preferences: {}
```

上面的certificate-authority-data中的内容就是根证书了， 我们只需要复制下来经过base64转码就可以复制到jenkins中了。
这样就做到了k8s 发送给jenkins的认证了， 因为jenkins获取并信任了根证书。 但是这样是不够的，因为k8s要的是双向认证，
所以光是客户端认证了k8s的身份不够，还需要k8s能够认证客户端的身份。
那么在这里有两个认证方式，我们首先说数据证书的方式。也就是说套路是客户端自己也要生成一个数字证书，然后发请求给CA，
CA再给客户端办法证书。 因为k8s服务端也是信任CA证书的，所以也就连带信任了客户端的证书。
但是这种方式比较麻烦而且无法解决权限的问题(创建k8s资源的权限)。所以我们不能使用这种方式。
下面只给出这种方式步骤不会继续验证它的作用：

1. 使用openssl命令生成私钥
2. 使用openssl命令生成一个csr请求文件(csr就是要发送给CA用来请求颁发一个证书的请求文件)
3. 使用kubectl命令创建一个k8s的csr(CertificateSigningRequest) 对象，对象里要把刚才生成的csr文件内容粘贴上去。
4. 使用 kubectl certificate approve 命令颁发证书(这一步也就是让CA给客户端颁发证书了)。我们使用这个证书就可以。

代码参考：

```
csrName=${service}.${namespace}
tmpdir=$(mktemp -d)
echo "creating certs in tmpdir ${tmpdir} "

cat <<EOF >> ${tmpdir}/csr.conf
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name
[req_distinguished_name]
[ v3_req ]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names
[alt_names]
DNS.1 = ${service}
DNS.2 = ${service}.${namespace}
DNS.3 = ${service}.${namespace}.svc
EOF

openssl genrsa -out ${tmpdir}/server-key.pem 2048
openssl req -new -key ${tmpdir}/server-key.pem -subj "/CN=${service}.${namespace}.svc" -out
${tmpdir}/server.csr -config ${tmpdir}/csr.conf

# clean-up any previously created CSR for our service. Ignore errors if not present.
kubectl delete csr ${csrName} 2>/dev/null || true

# create  server cert/key CSR and  send to k8s API
cat <<EOF | kubectl create -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: ${csrName}
spec:
  groups:
  - system:authenticated
  request: $(cat ${tmpdir}/server.csr | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - server auth
EOF

# verify CSR has been created
while true; do
    kubectl get csr ${csrName}
    if [ "$?" -eq 0 ]; then
        break
    fi
done

# approve and fetch the signed certificate
kubectl certificate approve ${csrName}
# verify certificate has been signed
for x in $(seq 10); do
    serverCert=$(kubectl get csr ${csrName} -o jsonpath='{.status.certificate}')
    if [[ ${serverCert} != '' ]]; then
        break
    fi
    sleep 1
done
if [[ ${serverCert} == '' ]]; then
    echo "ERROR: After approving csr ${csrName}, the signed certificate did not appear on the
resource. Giving up after 10 attempts." >&2
    exit 1
fi
echo ${serverCert} | openssl base64 -d -A -out ${tmpdir}/server-cert.pem
```

Service Account Token认证

这种认证方式是我们推荐的jenkins与k8s整合中用来给k8s验证jenkins身份的机制。

什么是service account

service account是Kubernetes唯一由自己管理的账号实体，意味着service account可以通过Kubernetes创建，每个namespace会有一个默认的service account(以下简称sa)，pod在启动的时候如果你不去指定自己创建的sa的话， 就会被分配到这个默认的sa。 sa是使用Bearer Token认证的，基于JWT(JSON Web Token)认证机制，JWT原理和x509证书认证其实有点类似，都是通过CA根证书进行签名和校验，只是格式不一样而已，所以认证流程参考刚才说的数字证书的方式即可。 所以在配合jenkins与k8s整合的时候， 我们需要为jenkins创建一个sa。 如下：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: qa-jenkins
  namespace: default
```

一个很简单的配置， 我们通过kubectl create 命令创建以后我们就在default名称空间看到这个sa。 但是我们可以查看一下它的内容。

```
-->$ k describe sa qa-jenkins
Name:         qa-jenkins
Namespace:    default
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1","kind":"ServiceAccount","metadata":{"annotations":{},"name":"qa-jenkins","namespace":"default"}}

Tokens:              qa-jenkins-token-mh5dh

Image pull secrets:  <none>

Mountable secrets:   qa-jenkins-token-mh5dh

Events:  <none>
```

我们可以看到k8s自动为sa创建了一个token，我们上面说sa的认证是基于token的， 所以这个token的内容就非常重要了。事实上，k8s为sa专门创建了一个secret对象来保存这个token。 如下：

```
root@m7-prod-ssd001 [13:01:57] {~/gaofei/tmp}
-->$ k describe secret qa-jenkins-token-mh5dh
Name:         qa-jenkins-token-mh5dh
Namespace:    default
Labels:       <none>
Annotations:  kubernetes.io/service-account.name=qa-jenkins
              kubernetes.io/service-account.uid=47755f4e-44f8-11ea-8034-fe0324ba10ee

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1359 bytes
namespace:  7 bytes
token:      eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6InFhLWplbmtpbnMtdG9rZW4tbWg1ZGgiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoicWEtamVua2lucyIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6IjQ3NzU1ZjRlLTQ0ZjgtMTFlYS04MDM0LWZlMDMyNGJhMTBlZSIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDpkZWZhdWx0OnFhLWplbmtpbnMifQ.U_yKk5Faiufskl79Gg8AHYQtfDdbzdDItrbMMpGS7N1mBt-GRiyChdBnLneaaXMaN14gHmtUfcvLbhnbWQJsn6zCSpev7VULm5vLFnaOFTjx5s1yqWKTz1xfOMvDEwiZIyfse9GAVGV7t6SyhHStGYriQ7Ns8wDGrRQoV0yS_4YeTnh9CpPIXsIxZU8vE-ax2qYV5A5zKIrZhk8WSz_G9ImwP7P0zpWLIiPrdUFdJ95ARxGFBSgQwXtF-6Gwi8NxDaVmJGHovdKSku3vBcaek1pm57n9DCXc0j--4MNzVDJLRImYTYt4SgTTgXaOLij4q-auA6xk8FMPiQh96O6pA
root@m7-prod-ssd001 [13:04:28] {~/gaofei/tmp}
-->$
```

可以看到这个secret对象里分别保存了ca的根证书，namespcel以及token。 这些我们认证身份需要的信息就都有了。
而事实上这3个文件会被挂载在pod的/run/secrets/kubernetes.io/serviceaccount 这个目录下。 每个Pod启动的时候都会有这些文件，而我们在使用client-go的时候，如果不额外指定kubeconfig， 它会默认读取这里面的内容来初始化。

OK， 到了这里我们只需要复制token的内容， 在jenkins上创建一个secret text类型的凭据后，在jenkins上的云配置中设置一下即可。
至此， 我们的双向认证就解决了。这时候你在UI上点击测试， 就可以看到连接成功的字样。

下面我们要解决最后一个问题，那就是权限， 虽然身份得到了认证，那并不代表你就有权限做各种各样的事情。
尤其是jenkins与k8s整合后需要动态创建pod作为slave节点， 所以它需要获取创建，更新，删除，查询k8s各种资源的权限。
接下来就讲RBAC

基于RBAC的权限管理机制

RBAC-- 基于角色的权限管理机制。 RBAC可以作用在service account上， 规定这个sa所有用的权限。 不多说，直接看demo。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: qa-jenkins
  namespace: default

---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: qa-jenkins
  namespace: default
rules:
- apiGroups: ["", "apps", "autoscaling", "batch"]
  resources: ["pods/exec", "services", "endpoints",
"pods","secrets","configmaps","crontabs","deployments","jobs","nodes","rolebindings","clusterroles
","daemonsets","replicasets","statefulsets","horizontalpodautoscalers","replicationcontrollers","c
ronjobs"]
  verbs: ["*"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: qa-jenkins
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: qa-jenkins
subjects:
- kind: ServiceAccount
  name: qa-jenkins
  namespace: default
```

在k8s中Role规定了一个角色， Role中rules这个字段中规定了apigroups（上面定了所有的4个group），
resources（所能操作的资源对象）和verbs（所拥有的权限，*代表所有权限）
也就是说上面我定义的这个Role能基本上拥有所有对象的所有权限（我懒，不想去一个一个的筛选jenkins到底需要哪些权限了，
所以干脆直接给他所有权限）。 这里要注意的是Pod/exec也是一个资源，这个当初坑到我了，
jenkins切换container执行命令的时候走的是类似kubetl exec这个命令的方式。 它是需要在pod/exec这个资源上有权限的，
而我当初不知道pod/exec这个资源对象， 单纯的以为只要在Pod上有exec权限就可以了。 结果悲剧了， 卡了我好一会才知道这个坑。

而RoleBinding则负责把Role和一个sa绑定， 也就是赋予sa一个角色，上面我就把这个有所有权限的角色给这样的sa上。
而我们在jenkins上配置了这个sa的token作为认证。 所以jenkins也就有了权限了。 详细的RBAC内容不细展开了~~~。这篇文章就到这。
至此我们jenkins和k8s的整合也就结束了。