

拉勾教育

—互联网人实战大学—

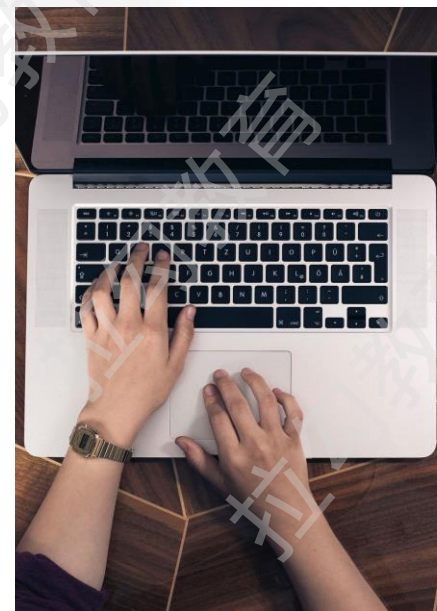
# 《Kubernetes 原理剖析与实战应用》

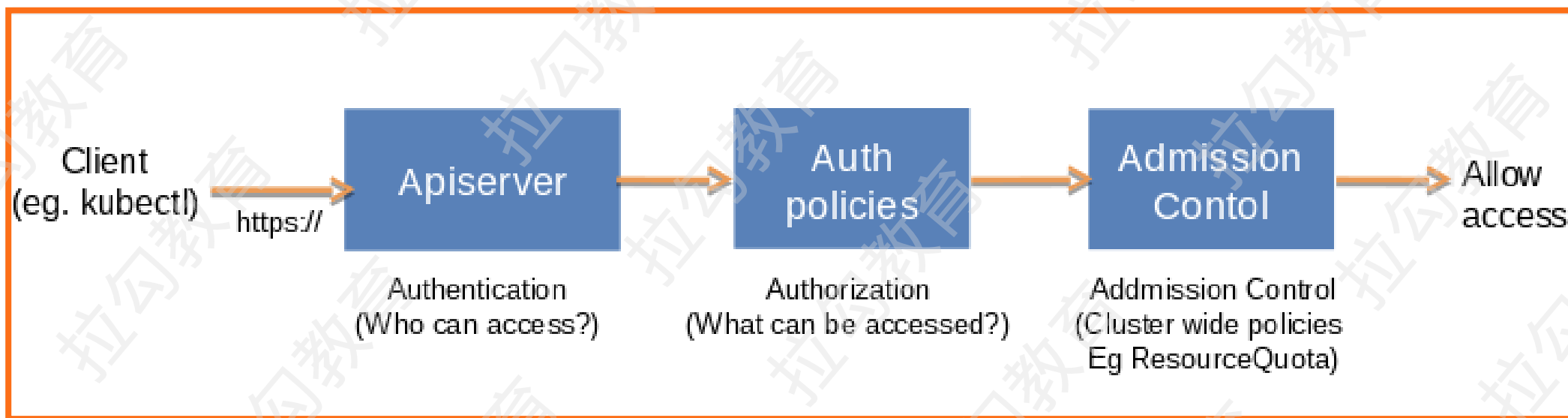
正范

— 拉勾教育出品 —

# 18 | 权限分析：Kubernetes 集群权限管理那些事儿

那么你是否知道在这其中 Kubernetes 是如何对这些请求进行认证、授权的呢？





## 授权

负责做权限管控

解决“你能干什么？”的问题

给予你能访问 Kubernetes 集群中的哪些

资源以及做哪些操作的权利

## 准入控制

从字面上你可能不太好理解

其实就是由一组控制逻辑级联而成

对对象进行拦截校验、更改等操作

## x509 证书

Kubernetes 使用 mTLS 进行身份验证，通过解析请求使用的 客户端（client）证书  
将其中的 subject 的通用名称（Common Name）字段（例如"/CN=bob"）作为用户名

<https://kubernetes.io/zh/docs/concepts/cluster-administration/certificates/>

```
openssl req -new -key zhangsan.pem -out zhangsan-csr.pem -subj  
"/CN=zhangsan/O=app1/O=app2"
```

## Token

用户自己提供的静态 Token

```
token1,user1,uid1,"group1,group2,group3"
```



## ServiceAccount Token

是 Kubernetes 中使用最为广泛的认证方式之一

主要用来给 Pod 提供访问 API Server 的权限

通过使用 Volume 的方式挂载到 Pod 中

<https://kubernetes.io/zh/docs/tasks/configure-pod-container/configure-service-account/>

```
$ kubectl create sa demo
serviceaccount/demo created
→ ~ kubectl get sa demo
NAME SECRETS AGE
demo 1 6s
→ ~ kubectl describe sa demo
Name: demo
Namespace: default
Labels: <none>
Annotations: <none>
Image pull secrets: <none>
Mountable secrets: demo-token-fvsjg
Tokens: demo-token-fvsjg
Events: <none>
```

```
$ kubectl get secret demo-token-fvsjg
NAME          TYPE          DATA  AGE
demo-token-fvsjg  kubernetes.io/service-account-token  3    27s
$ kubectl describe secret demo-token-fvsjg
Name:         demo-token-fvsjg
Namespace:    default
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: demo
              kubernetes.io/service-account.uid: f8fe4799-9add-4a36-8c29-a6b2744ba9a2

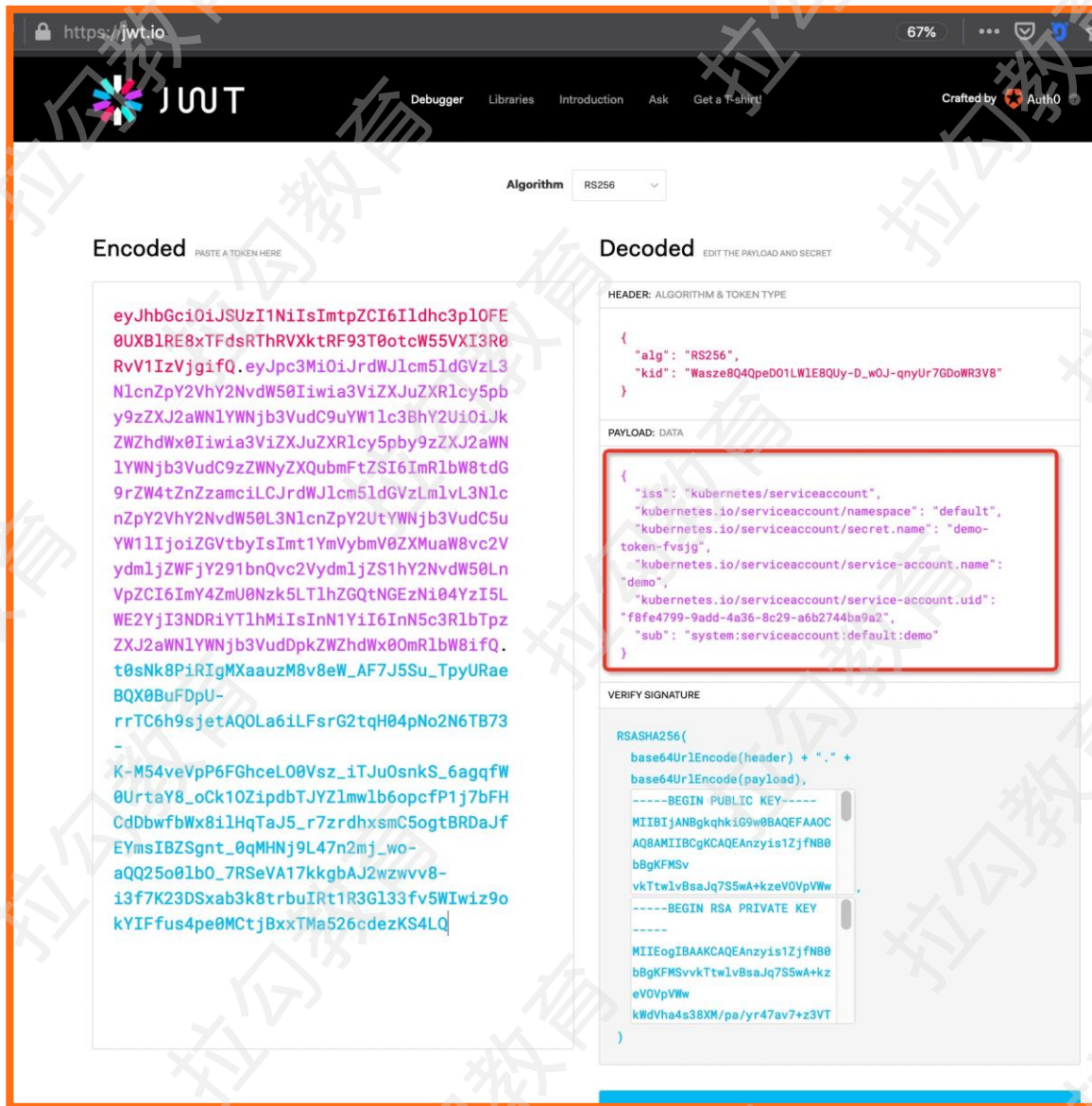
Type: kubernetes.io/service-account-token
```

Data

====

eyJhbGciOiJSUzI1NiIsImtpZCI6Ildhc3plOFE0UXBlRE8xTFdsRThRVXktRF93T0otcW55VXI3R0RvV1IzV  
jgifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYW  
Njb3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWwNjb3VudC9zZW  
NyZXQubmFtZSI6ImRlbW8tdG9rZW4tZnZzZamciLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3  
NlcnZpY2UtYWNjb3VudC5uYW1lIjoieZGVtbyIsImt1YmVybmV0ZXMuaW8vc2VydmljZWJY291bnQvc2V  
ydmljZS1hY2NvdW50LnVpZCI6ImY4ZmU0Nzk5LTlhZGQtNGEzNi04YzI5LWE2YjI3NDRIYTlhMilsInN1Yi  
I6InN5c3RlbTpozZXJ2aWNlYWwNjb3VudDpkZWZhdWx0OmRlbW8ifQ.t0sNk8PiRlgMXaauzM8v8eW\_AF  
7J5Su\_TpyURaeBQX0BuFDpU-rrTC6h9sjetaQOLa6iLFsrG2tqH04pNo2N6TB73-K-  
M54veVpP6FGhceLO0Vsz\_iTJuOsnkS\_6agqfW0UrtaY8\_oCk1OZipdbTJYZlmwlb6opcfP1j7bFHCdDbw  
fbWx8ilHqTaJ5\_r7zrdhxsmC5ogtBRDaJfEYmsIBZSgnt\_0qMHNj9L47n2mj\_wo-  
aQQ25o0lbO\_7RSeVA17kkgbAJ2wzwvv8-  
i3f7K23DSxab3k8trbulRt1R3Gl33fv5Wlwiz9okYIFfus4pe0MCtjBxxTMa526cdezKS4LQ

namespace: 7 bytes



The screenshot shows the JWT.io website interface. The top navigation bar includes links for Debugger, Libraries, Introduction, Ask, and Get a T-shirt. The main content area is divided into two sections: Encoded and Decoded. The Encoded section contains a long string of base64-encoded characters. The Decoded section shows the decoded payload, which is a JSON object representing a Kubernetes service account. The payload is highlighted with a red box. The payload contains the following information:

```
{  "iss": "kubernetes/serviceaccount",  "kubernetes.io/serviceaccount/namespace": "default",  "kubernetes.io/serviceaccount/secret.name": "demo-token-fvsg",  "kubernetes.io/serviceaccount/service-account.name": "demo",  "kubernetes.io/serviceaccount/service-account.uid": "f8fe4799-9add-4a36-8c29-a6b2744ba9a2",  "sub": "system:serviceaccount:default:demo"}
```

The Decoded section also includes a section for the RSASHA256 signature, which is a base64-encoded string of the header and payload concatenated with a dot, followed by the signature itself.

```
kubeadm join --discovery-token abcdef.1234567890abcdef
```



```
kubeadm join --discovery-token abcdef.1234567890abcdef
```

Authorization: Bearer this-is-a-very-very-very-long-token





## ABAC

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1", "kind": "Policy", "spec":  
  {"user": "zhangsan", "namespace": "*", "resource": "pods", "readOnly": true}}
```

[https://raw.githubusercontent.com/kubernetes/kubernetes/  
master/pkg/auth/authorizer/abac/example\\_policy\\_file.jsonl](https://raw.githubusercontent.com/kubernetes/kubernetes/master/pkg/auth/authorizer/abac/example_policy_file.jsonl)

## RABC

通过 Kubernetes 的对象就可以直接进行管理，也便于动态调整权限

引入了角色，所有的权限都是围着这个角色进行展开的

每个角色里面定义了可以操作的资源以及操作方式

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # 空字符串""表明使用core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

```
kind: ClusterRole
```

```
apiVersion: rbac.authorization.k8s.io/v1
```

```
metadata:
```

```
#鉴于ClusterRole是集群范围对象，所以这里不需要定义?"namespace"字段
```

```
name: pods-reader
```

```
rules:
```

```
- apiGroups: [""]
```

```
resources: ["pods"]
```

```
verbs: ["get", "watch", "list"]
```

```
apiVersion: rbac.authorization.k8s.io/v1
#此角色绑定使得用户"jane" 或者 "manager"组 (Group) 能够读取"default"命名空间中的 Pods
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane #名称区分大小写
  apiGroup: rbac.authorization.k8s.io
- kind: Group
  name: manager #名称区分大小写
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role #这里可以是 Role, 也可以是 ClusterRole
  name: pods-reader #这里的名称必须与你想要绑定的 Role 或 ClusterRole 名称一致
  apiGroup: rbac.authorization.k8s.io
```

准入控制可以帮助我们在 APIServer 真正处理对象前做一些校验以及修改的工作

```
https://kubernetes.io/zh/docs/reference/access-authn-authz/admission-controllers/#%E6%AF%8F%E4%B8%AA%E5%87%86%E5%85%A5%E6%8E%A7%E5%88%B6%E5%99%A8%E7%9A%84%E4%BD%9C%E7%94%A8%E6%98%AF%E4%BB%80%E4%B9%88
```



## Kubernetes

Kubernetes 在 Auth 方面设计得很完善

支持多种后端身份认证授权系统

比如 LDAP (Lightweight Directory Access Protocol)、Webhook 等

Next: 《19 | 资源限制：如何保障你的 Kubernetes 集群资源不会被打爆》



# 拉勾教育

— 互联网人实战大学 —



关注拉勾「教育公众号」  
获取更多课程信息