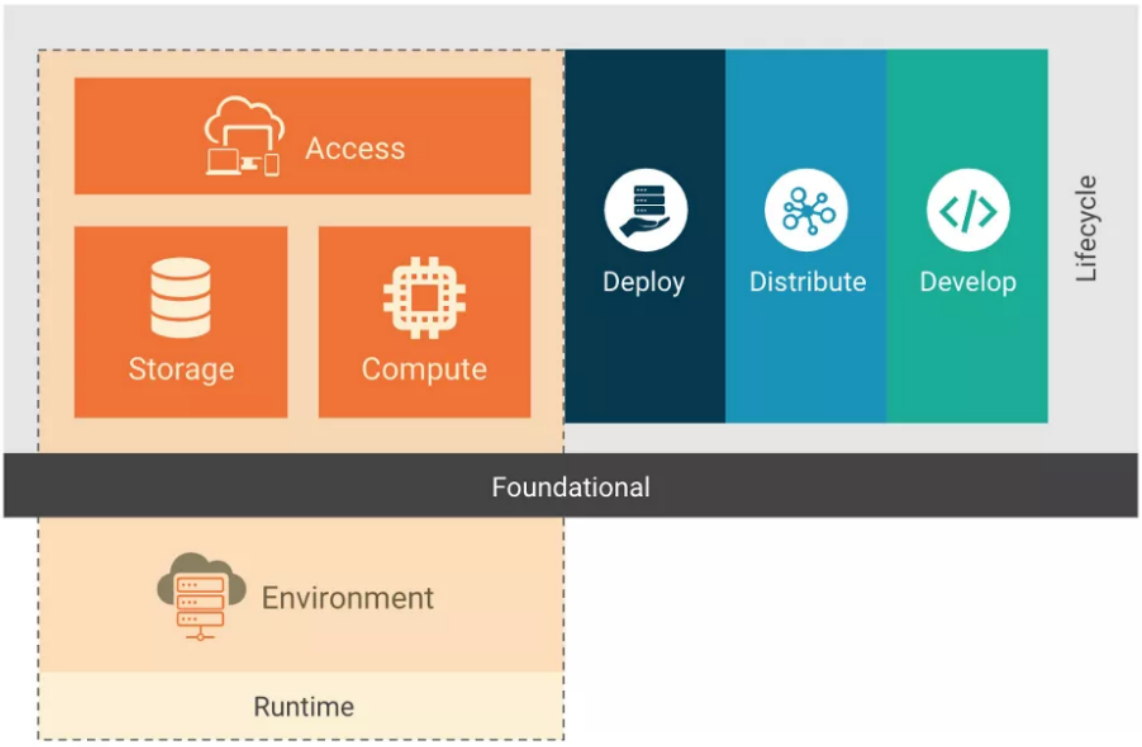
# 模块14 基于Kubernetes和Istio的安全保证

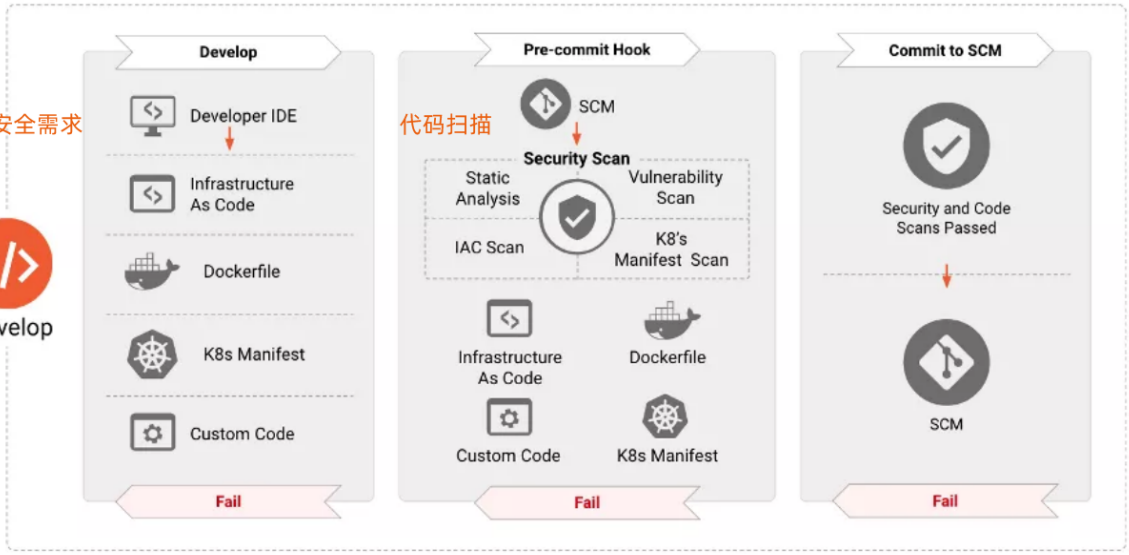
## 云原生语境下的安全保证

### 层次模型



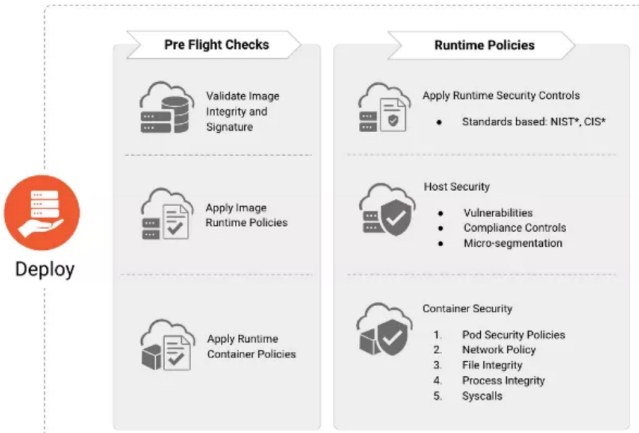

### 开发环节安全保证

- 传统安全的CIA三要素
- 基础设施即代码



## 分发环节安全保证

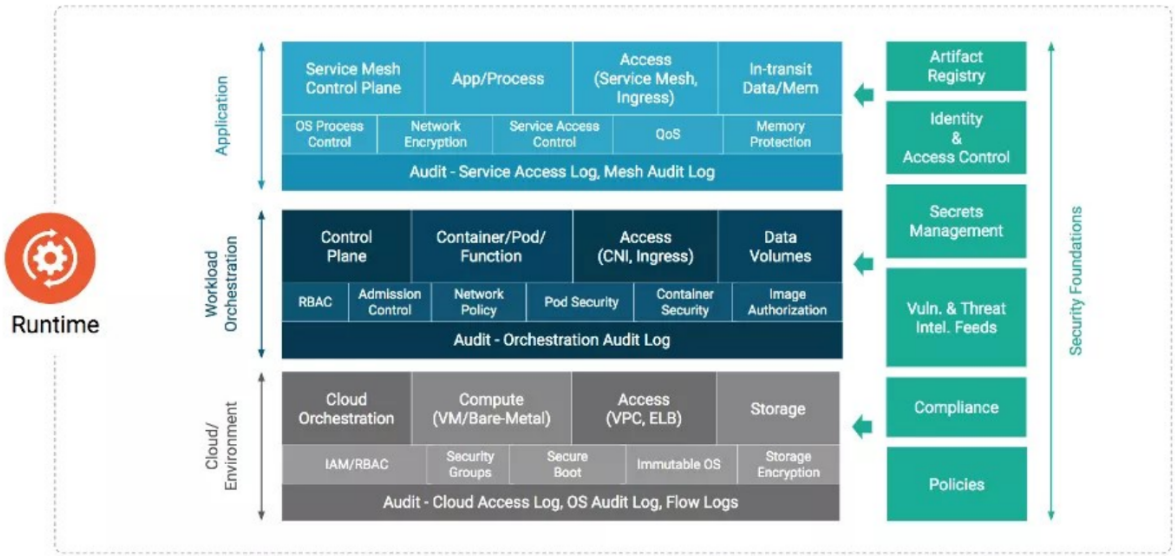

## 部署环节安全保证




运行前检查

- 镜像签名及完整性
- 镜像运行策略（如没有恶意软件或严重的漏洞）
- 容器运行策略（如无过度权限）
- 主机安全漏洞和合规性控制
- 工作负载、应用程序和网络安全策略

## 运行环境




# 容器运行时安全保证

- 以Non-root身份运行容器

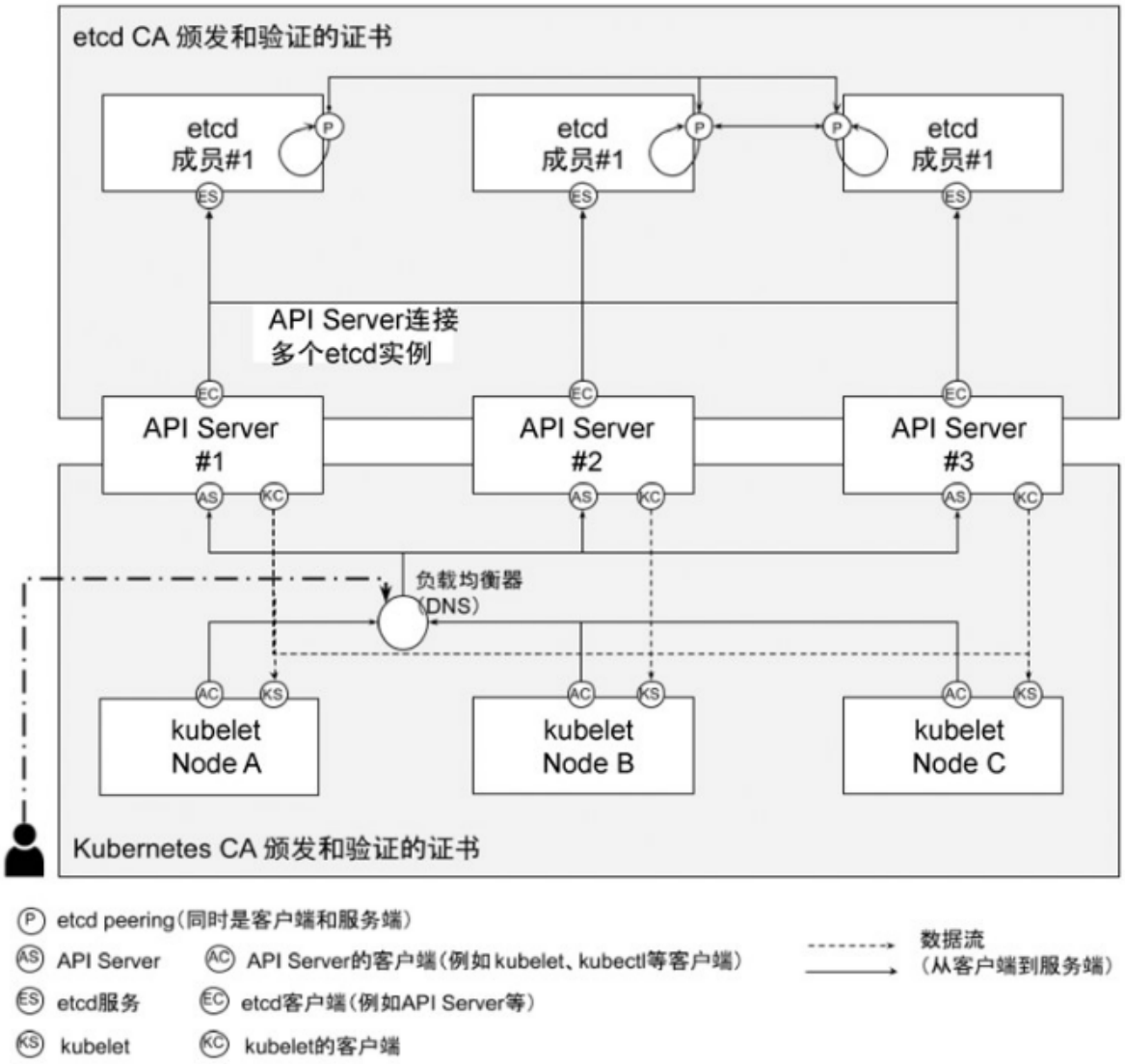
```
FROM ubuntu
RUN user add patrick
USER patrick
```

- User Namespace
  - 依赖于 User namespace，任何容器内部的用户都会映射成主机的非 root 用户。
  - 但该功能未被默认 enable，因其引入配置复杂性，比如系统不知道主机用户和容器用户的映射关系，在 mount 文件的时候无法设置适当的权限。
- rootless container
  - 指**容器运行时**以非root身份启动。即使容器被突破，在主机层面获得的用户权限也是非root身份的用户，确保了安全
  - Docker和其它运行时本身的后台Deamon需要以root身份运行，这样其它用户的容器才能以rootless身份运行
  - 无需Daemon的运行时，完全不需要root身份
- 集群的安全性保证
  - 保证容器与容器之间、容器与主机之间隔离，限制容器对其他容器和主机的消极影响。
  - 保证组件、用户及容器应用程序都是最小权限，限制它们的权限范围。
  - 保证集群的敏感数据的传输和存储安全。
  - 常用手段
    - Pod安全上下文（Pod Security Context）
    - API Server的认证、授权、审计和准入控制
    - 数据的加密机制等

# Kubernetes的安全保证

## 集群的安全通信

期望所有的API通信在默认情况下都使用TLS加密




## 控制面安全保证

- 认证
  - 小型的单用户集群可能希望使用简单的证书或静态承载令牌方法。
  - 更大的集群则可能希望整合现有的、OIDC、LDAP 等允许用户分组的服务器。
  - 所有 API 客户端都必须经过身份验证，即使它是基础设施的一部分，比如节点、代理、调度程序和卷插件。 这些客户端通常使用服务帐户或 X509 客户端证书，并在集群启动时自动创建或是作为集群安装的一部分进行设置。
- 授权
  - 与身份验证一样，简单而广泛的角色可能适合于较小的集群，但是随着更多的用户与集群交互，可能需要将团队划分成有更多角色限制的单独的命名空间。
- 配额
  - 资源配额限制了授予命名空间的资源的数量或容量。这通常用于限制命名空间可以分配的 CPU、内存或持久磁盘的数量，但也可以控制每个命名空间中有多少个 Pod、服务或卷的存在。

## NodeRestriction

对kubelet的权限进行非RBAC的限制

- 准入控制器限制了 kubelet 可以修改的 Node 和 Pod 对象，kubelet 只可修改自己的 Node API 对象，只能修改绑定到节点本身的 Pod 对象。
- NodeRestriction 准入插件可防止 kubelet 删除 Node API 对象。
- 防止 kubelet 添加/删除/更新带有 node-restriction.kubernetes.io/ 前缀的标签。
- 将来的版本可能会增加其他限制，以确保 kubelet 具有正常运行所需的最小权限集。

## etcd存储加密

```
apiVersion: API Server.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    providers:
    - identity: {}
    - aesgcm:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
    - aescbc:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
    - secretbox:
        keys:
        - name: key1
          secret:
YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0
NTY=
    - kms:
        name : myKmsPlugin
        endpoint: unix:///tmp/socketfile.sock
        cachesize: 100
```

## Security Context

Pod 定义包含了一个安全上下文，用于描述允许它请求访问某个节点上的特定 Linux 用户（如 root）、获得特权或访问主机网络，以及允许它在主机节点上不受约束地运行的其它控件。

Pod 安全策略可以限制哪些用户或服务帐户可以提供危险的安全上下文设置。 例如：Pod 的安全策略可以限制卷挂载，尤其是 hostpath，这些都是 Pod 应该控制的一些方面。

一般来说，大多数应用程序需要限制对主机资源的访问，他们可以在不能访问主机信息的情况下成功以根进程（UID 0）运行。但是，考虑到与 root 用户相关的特权，在编写应用程序容器时，你应该使用非 root 用户运行。

类似地，希望阻止客户端应用程序逃避其容器的管理员，应该使用限制性的 Pod 安全策略。

Kubernetes 提供了三种配置 Security Context 的方法：

- Container-level Security Context：仅应用到指定的容器。
- Pod-level Security Context：应用到 Pod 内所有容器以及 Volume。
- Pod Security Policies（PSP）：应用到集群内部所有 Pod 以及 Volume。

### Container-level Security Context

仅应用到指定的容器上，并且不会影响volume。下面的例子设置容器运行在特权模式（privileged）

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
    - name: hello-world-container
      # The container definition
      # ...
      securityContext:
        privileged: true
```

**Pod-level Security Context**

应用到Pod内所有容器，并影响Volume（包括fsGroup（就是建立目录是的用户和组）和selinuxOptions（孟老师也从来没有实践过））

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
  # specification of the pod's containers
  # ...
  securityContext:
    fsGroup: 1234
    supplementalGroups: [5678]
    seLinuxOptions:
      level: "s0:c123,c456"
```

### Pod Security Policy (PSP)

集群级pod安全策略，自动为集群内的Pod和Volume设置Security Context

| 控制项 | 说明 |
| --- | --- |
| privileged | 运行特权容器 |
| defaultAddCapabilities | 可添加到容器的 Capabilities |
| requiredDropCapabilities | 会从容器中删除的 Capabilities |
| volumes | 控制容器可以使用哪些 volume |
| hostNetwork | host 网络 |
| hostPorts | 允许的 host 端口列表 |
| hostPID | 使用 host PID namespace |
| hostIPC | 使用 host IPC namespace |
| seLinux | SELinux Context |
| runAsUser | user ID |
| supplementalGroups | 允许的补充用户组 |
| fsGroup | volume FSGroup |
| readOnlyRootFilesystem | 只读根文件系统 |

比如

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: permissive
spec:
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  hostPorts:
  - min: 8000
    max: 8080
  volumes:
  - '*'
```

## Taint

### 为节点增加Taint

使用命令 kubectl taint 给节点增加一个 Taint：

```
kubectl taint nodes node1 key=value:NoSchedule
```

运行如下命令删除 Taint：

```
kubectl taint nodes node1 key:NoSchedule-
```

在 PodSpec 中为容器设定容忍标签：

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

```
tolerations:
- key: "key"
  operator: "Exists"
  effect: "NoSchedule"
```

> 详细可参见模块7笔记。

可以以租户为粒度，为不同租户的节点增加taint，使得节点彼此隔离。

Taint的作用是让租户独占节点，无对应Toleration的Pod无法调度到Taint节点上，实现了应用部署的隔离。

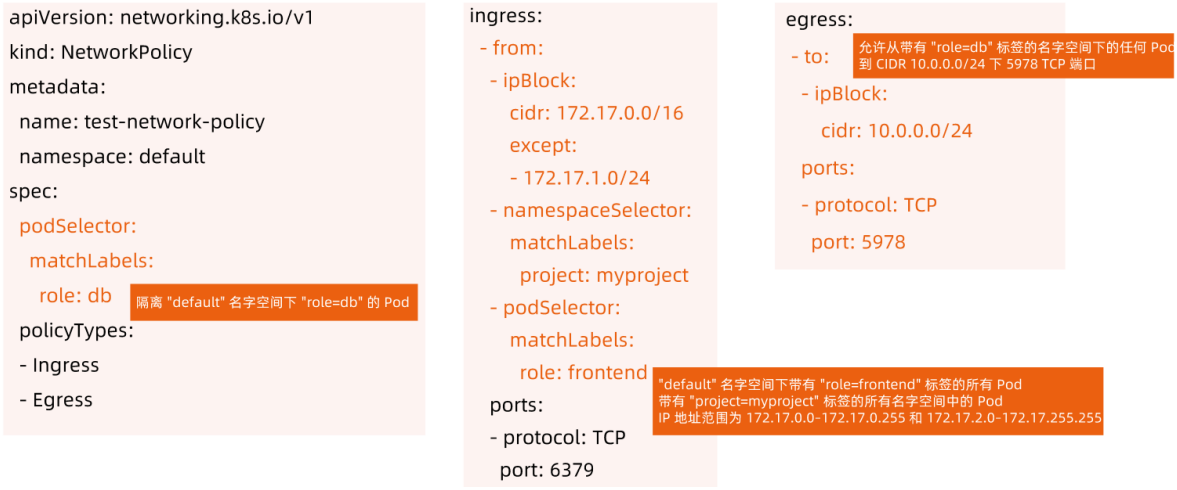
# NetworkPolicy

NetworkPolicy用于控制3层或者4层的网络流量。默认情况下，K8S集群中的POD都是互通的，可以指定一个白名单，在白名单中的POD才可以于当前POD通信（白名单的意思就是如果建立了白名单，默认就是不通的，必须在“墙上开洞”才可以通讯）

- 直接指定POD、
- 指定命名空间
- 指定的IP地址或地址段
- 策略类型policyTypes分为两类：Ingress或者Egress
  - 如果未指定类型，则默认是Ingress
  - 如果有任何出口规则的话则设置Egress

> NetworkPolicy通过网络插件实现，比如Calico或Cilium。多个规则之间形成了一个并集。

例子如下




## Ingress默认规则的设置语法

默认拒绝所有入站流量

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: default-deny-ingress
spec:
 podSelector: {}
 policyTypes:
 - Ingress
```

默认允许所有入站流量

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: allow-all-ingress
spec:
 podSelector: {}
 ingress:
 - {}
 policyTypes:
 - Ingress
```

当ingress有一个空的规则时就表示允许，如果啥也没有就表示拒绝

## Egress默认规则的设置语法

默认允许所有出站流量

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: allow-all-egress
spec:
 podSelector: {}
 egress:
 - {}
 policyTypes:
 - Egress
```

默认拒绝所有入口和所有出站流量

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: default-deny-all
spec:
 podSelector: {}
 policyTypes:
 - Ingress
 - Egress
```

# 依托于Calico的NetworkPolicy

K8S内置的NetworkPolicy使用起来不够灵活，所以Calico也提供了一个同名对象，但是位于不同的组

## NetworkPolicy

命名空间级别资源，不限制节点和Pod之间的通信，只能限制POD（被成为Endpoint ）之间的

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
 name: allow-tcp-90
spec:
 selector: app == 'envoy' # 应用此策略的 endpoint
 types: # 应用策略的流量方向
   - Ingress
   - Egress
```

```
ingress: # 入口的流量规则
  - action: Allow # 流量的行为
    protocol: ICMP # 流量的协议
    notProtocol: TCP # 匹配流量协议不为值的流量
    source: # 流量的来源 src 与 dst 的匹配关系为与，所有的都生效即生效
      nets: # 有效的来源 IP
      selector: # 标签选择器
      namespaceSelector: # 名称空间选择器
      ports: # 端口
      - 80 # 单独端口
      - 6040:6050     # 端口范围
    destination: # 流量的目标
egress: # 出口的流量规则
  - action: Allow
serviceAccountSelector: # 使用与此规则的 serviceAccount
```

## GlobalNetworkPolicy

功能与NetworkPolicy一样，但是为集群级别的资源，而且能限制主机（HostEndpoint）

## 理解Calico的防火墙规则

NetworkPolicy其实就是防火墙，也是利用Iptables实现的。Calico的原理是创建很多Veth pair设备，这些设备都以“cali-”开头

```
:cali-FORWARD - [0:0]
:cali-INPUT - [0:0]
:cali-OUTPUT - [0:0]
-A INPUT -m comment --comment "cali:Cz_u1IQiXIMmKD4c" -j cali-INPUT
-A cali-INPUT -p udp -m comment --comment "cali:w7ud0UgQSEi_zKuQ" -m comment --comment "Allow VXLAN packets from whitelisted hosts" -m multiport --dports 4789 -m set --match-set cali40all-vxlan-net src -m addrtype --dst-type LOCAL -j ACCEPT
-A cali-INPUT -p udp -m comment --comment "cali:4cgmbdWsLmozYhJh" -m comment --comment "Drop VXLAN packets from non-whitelisted hosts" -m multiport --dports 4789 -m addrtype --dst-type LOCAL -j DROP
-A cali-INPUT -i cali+ -m comment --comment "cali:t45BUBhpu3Wsmi1_" -g cali-wl-to-host
-A cali-INPUT -m comment --comment "cali:NOmsycyknYZaGOFf" -m mark --mark 0x10000/0x10000 -j ACCEPT
-A cali-INPUT -m comment --comment "cali:Or0B7eoenKO2p8Bf" -j MARK --set-xmark 0x0/0xf0000
-A cali-INPUT -m comment --comment "cali:AmIfvPGG2lYUK6mj" -j cali-from-host-endpoint
-A cali-INPUT -m comment --comment "cali:79fWWn1SpufdO7SE" -m comment --comment "Host endpoint policy accepted packet." -m mark --mark 0x10000/0x10000 -j ACCEPT
-A cali-wl-to-host -m comment --comment "cali:Ee9Sbo10IpVujdIY" -j cali-from-wl-dispatch
-A cali-wl-to-host -m comment --comment "cali:nSZbcOoG1xPONxb8" -m comment --comment "Configured DefaultEndpointToHostAction" -j ACCEPT
-A cali-to-wl-dispatch -o cali2+ -m comment --comment "cali:ZJts1ClWTzhqZEeH" -g cali-to-wl-dispatch-2
-A cali-to-wl-dispatch-2 -o cali23a582ef038 -m comment --comment "cali:3OpBnQAsm2jeqKzt" -g cali-tw-cali23a582ef038
```

### 创建规则允许ICMP

```yaml
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: allow-ping-in-cluster
spec:
  selector: all()    #类似于一个表达式
  types:
    - Ingress
```

```yaml
  ingress:
    - action: Allow
      protocol: ICMP
      source:
        selector: all()
      icmp:
        type: 8 # Ping request
    - action: Allow
      protocol: ICMPv6
      source:
        selector: all()
      icmp:
        type: 128 # Ping request
```

创建上面的规则后，防火墙规则如下

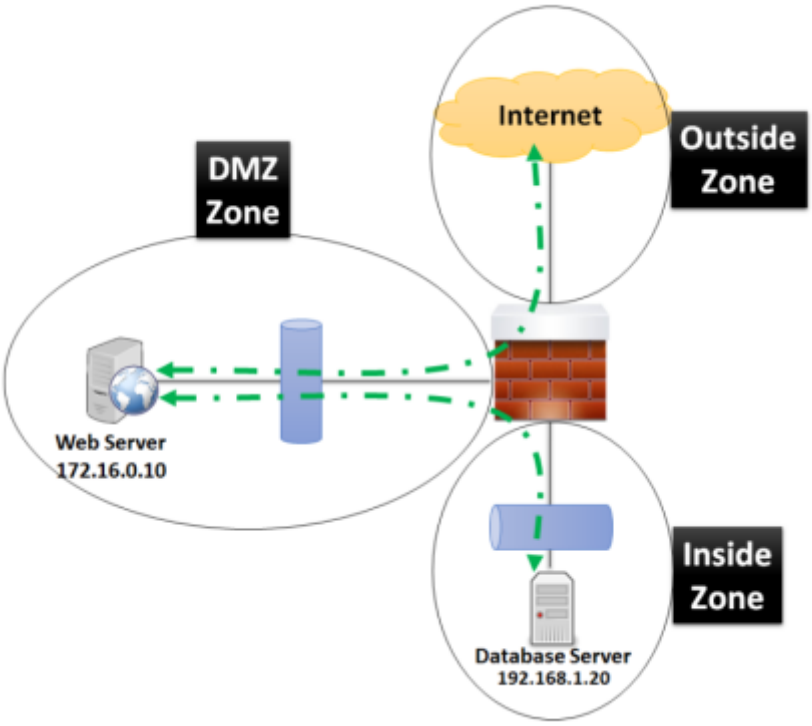
```
-A cali-tw-cali23a582ef038 -m comment --comment "cali:BCIjDQUbOGHklAW3" -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A cali-tw-cali23a582ef038 -m comment --comment "cali:yw1bAkXtcDncZYnn" -m conntrack --ctstate INVALID -j DROP
-A cali-tw-cali23a582ef038 -m comment --comment "cali:Q8lAtOCV68p7YEgj" -j MARK --set-xmark 0x0/0x10000
-A cali-tw-cali23a582ef038 -m comment --comment "cali:MlIFD0RDk3GLGBw_" -m comment --comment "Start of policies" -j MARK --set-xmark 0x0/0x20000
-A cali-tw-cali23a582ef038 -m comment --comment "cali:nuuVD6tedS2uGLLB" -m mark --mark 0x0/0x20000 -j cali-pi-_6MPzWUO0Ko_GzNULnUn
-A cali-tw-cali23a582ef038 -m comment --comment "cali:1kxCb6jIwvgJ1o-0" -m comment --comment "Return if policy accepted" -m mark --mark 0x10000/0x10000 -j RETURN
-A cali-tw-cali23a582ef038 -m comment --comment "cali:8x_ka5R_adJgPMpR" -m mark --mark 0x0/0x20000 -j cali-pi-_1GuvcYJ2yW4bHzL8wH-
-A cali-tw-cali23a582ef038 -m comment --comment "cali:Rgn8KAkSp5qogm_L" -m comment --comment "Return if policy accepted" -m mark --mark 0x10000/0x10000 -j RETURN
-A cali-tw-cali23a582ef038 -m comment --comment "cali:-MPNeMXPojG9o70Z" -m comment --comment "Drop if no policies passed packet" -m mark --mark 0x0/0x20000 -j
DROP
-A cali-tw-cali23a582ef038 -m comment --comment "cali:NQY_ueeL5deBM7Gr" -j cali-pri-kns.ns-calico-01
-A cali-tw-cali23a582ef038 -m comment --comment "cali:uXJa3c-DvyNMk4k2" -m comment --comment "Return if profile accepted" -m mark --mark 0x10000/0x10000 -j RETURN
-A cali-tw-cali23a582ef038 -m comment --comment "cali:Fvd66tSiXb2mjtmi" -j cali-pri-_sZa7qKPN6oyJdl6bLA
-A cali-tw-cali23a582ef038 -m comment --comment "cali:BFE1X5N6HK2ULI3F" -m comment --comment "Return if profile accepted" -m mark --mark 0x10000/0x10000 -j RETURN
-A cali-tw-cali23a582ef038 -m comment --comment "cali:1ROdF_sc6w3fUpLs" -m comment --comment "Drop if no profiles matched" -j DROP
-A cali-pi-_1GuvcYJ2yW4bHzL8wH- -p icmp -m comment --comment "cali:--CtA5sGB7G86H8e" -m set --match-set cali40s:5y5I3VdRZfDU01O--xXAPx2 src -m icmp --icmp-type 8 -
j MARK --set-xmark 0x10000/0x10000
-A cali-pi-_1GuvcYJ2yW4bHzL8wH- -m comment --comment "cali:1Sl0swLbnXZNBbt9" -m mark --mark 0x10000/0x10000 -j RETURN
```

- 上面倒数第二条规则给ICMP的ping请求打了一个标签0x10000/0x10000
- 然后在第二条橙色规则中，对这样标记的流量做了放行

# 零信任架构（Zero Trust Architecture, ZTA）

传统的架构是基于边界的安全架构，默认内网比外网更安全，不法分子一旦突破边界进入内网，将会带来严重后果。传统的认证，具有如下特征：

- 以边界防护
- 静态访问控制
- **以网络为中心**




随着云计算、大数据、物联网、移动办公等新技术与业务的深度融合，网络安全边界逐渐变得更加模糊，传统边界防护安全理念面临巨大挑战。

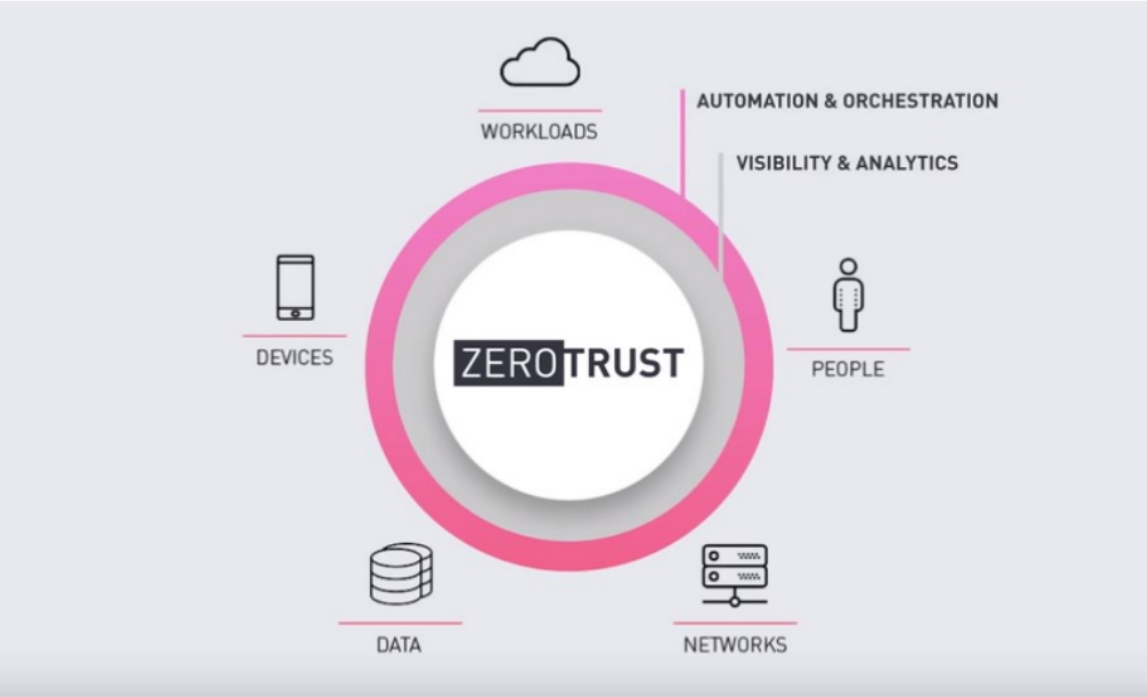
## 零信任核心原则

从不信任，始终验证

## 特征

- 以身份为中心（而不是以网络为中心）
- 以识别、持续认证、动态访问控制、授权、审计以及监测为链条
- 以最小化实时授权为核心
- 以多维信任算法为基础
- 认证达末端


## 安全模型




圆周围的图标表示零信任xxx，比如零信任设备等。

## 三大技术

SIM:

- 软件定义边界（SDP，Software Defined Perimeter）
- 身份识别与访问管理（IAM, Identity and Access Management）
    - 微隔离（MSG, Micro segmentation）