# Zero Trust Infrastructure implementation in Kubernetes using Calico and Istio

Author – Twesha Thakur, 12223061, 33, Course Code – INT334

***Executive Summary* — This project demonstrates the implementation of a zero-trust security architecture on a containerized web application using industry-standard tools: Calico for network-level security and Istio for service mesh management. The implementation showcases defense-in-depth security principles by combining multiple security layers—network policies (Layer 3/4) and service mesh encryption (Layer 7)—to create a production-grade secure environment.**

**Technologies Used:** Kubernetes, Docker, Calico, Istio, Python Flask, Nginx, Prometheus, Grafana, Kiali

***GitHub Link* — https://github.com/TweshaThakur/Zero-Trust-Architecture-in-K8s-Calico-Istio**

## I. Introduction

### *1.1 Background*

Modern cloud-native applications require robust security measures that go beyond traditional perimeter-based security. Zero-trust architecture operates on the principle of "never trust, always verify," assuming that threats can exist both inside and outside the network perimeter.

### *1.2 Project Objectives*

1. Deploy a multi-tier web application on Kubernetes
2. Implement network-level security using Calico network policies
3. Integrate service mesh security using Istio with mTLS
4. Demonstrate defense-in-depth security architecture
5. Achieve observability of secure communications
6. Resolve integration challenges between network policies and service mesh

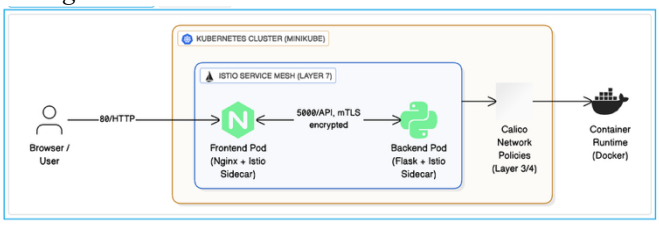
### *1.3 Problem Statement*

Traditional monolithic security approaches fail in microservices architectures. This project addresses:

- **Challenge 1:** How to secure pod-to-pod communication at the network level
- **Challenge 2:** How to implement application-level encryption without code changes
- **Challenge 3:** How to make Calico and Istio work together (known conflict)
- **Challenge 4:** How to visualize and verify security measures in real-time

## II. System Architecture

### *2.1 High-Level Architecture*




### *2.2 Component Description*

Application Components**:**

- **Frontend Service:** Nginx web server serving HTML/JavaScript interface
- **Backend Service:** Python Flask REST API with health and data endpoints
- **LoadTest Pod:** Traffic generator for demonstrating service mesh behavior

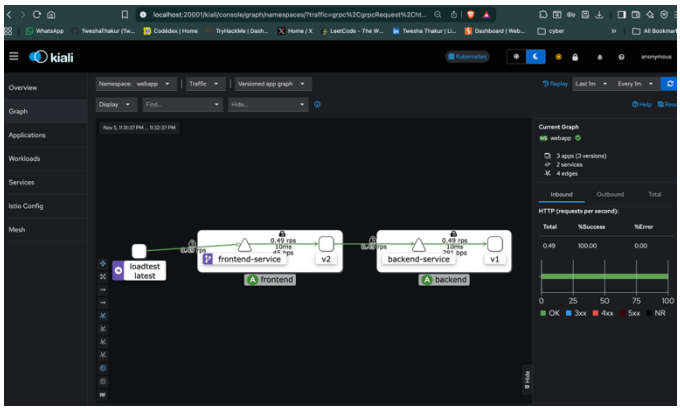
Security Components**:**

- **Calico CNI:** Container Network Interface providing network policy enforcement
- **Istio Service Mesh:** Traffic management with automatic mTLS encryption
- **Istio Sidecars:** Envoy proxies injected into each pod for traffic interception

Observability Components**:**

- **Kiali:** Service mesh topology visualization
- **Prometheus:** Metrics collection and storage
- **Grafana:** Metrics visualization dashboards

*2.3 Network Flow*




## III. IMPLEMENTATION

*3.1 Environment Setup*

Infrastructure Requirements:

- Minikube cluster with 6GB RAM, 4 CPUs
- Docker driver for container runtime
- CNI disabled initially (for Calico installation)
- Pod network CIDR: 192.168.0.0/16

Key Configuration:

- minikube start --driver=docker --memory=6144 --cpus=4 \ --cni=false --extra-config=kubeadm.pod-network-cidr=192.168.0.0/16

- kubectl get nodes

*3.2 Application Deployment*

1. **Backend Service (Python Flask App)**
   i. REST API with three endpoints: `/`, `/api/data`, `/health`
   ii. Returns JSON responses with hostname
   iii. timestamp - Containerized with Docker (image: `simple-backend:v1`)
2. **Frontend Service (Nginx)**
   i. Serves HTML/JavaScript web interface
   ii. Proxies `/api/backend/*` requests to backend service
   iii. Containerized with Docker (image: `simple-frontend:v2`)
3. **Kubernetes Resources**
   i. Namespace: `webapp` - 2
   ii. Deployments (frontend, backend) - 2
   iii. Services (ClusterIP for backend, NodePort for frontend)
   iv. Resource limits configured for both pods

```
tweshathakur@Tweshas-MacBook-Air ~ % kubectl get pods -n webapp
NAME                        READY   STATUS    RESTARTS      AGE
backend-75898d6fcc-49sxs    2/2     Running   2 (8d ago)    8d
frontend-6c795679bf-jmc57   2/2     Running   2 (8d ago)    8d
loadtest                    2/2     Running   2 (8d ago)    8d
tweshathakur@Tweshas-MacBook-Air ~ %
```

*3.3 Calico Implementation*

1. **Installation**
   - Installed Tigera operator and Calico custom resources
     - Configured with VXLAN encapsulation for pod networking
     - Enabled network policy enforcement
     - Network Policy
2. **Network Policy**
   - Default Deny Policy: Blocks all ingress and egress by default
   - Backend Policy: Allows specific traffic patterns
   - Frontend Policy: Permits user access and backend communication
   - LoadTest Policy: Enables traffic generation for demonstration

*3.4 Istio Implementation*

1. Installation
   - Installed using demo profile (includes ingress/egress gateways)
   - Enabled automatic sidecar injection for webapp namespace
   - Configured Gateway and VirtualService for traffic routing
2. mTLS Configuration
   By default, Istio enforces automatic mutual TLS between services:
   - Certificate generation handled by Citadel/Istiod
   - Automatic certificate rotation
   - No application code changes required
3. A challenge was – Injected Istio sidecars injected couldn't start because of deny-all network policy of calico.
4. Solution - Allowing specific Istio components ports
   port: 15012 # *Pilot/Istiod XDS*
   port: 15021 # *Sidecar health*
   port: 15090 # *Envoy Prometheus*
   port: 15020 # *Merged metrics*

3.5 *Obeservability*

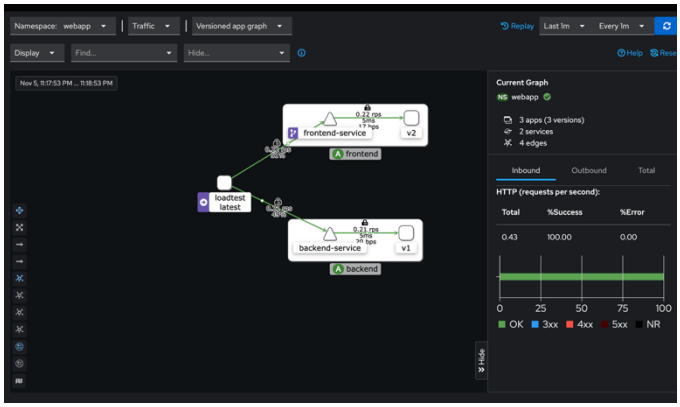
Kiali Service Mesh Dashboard:

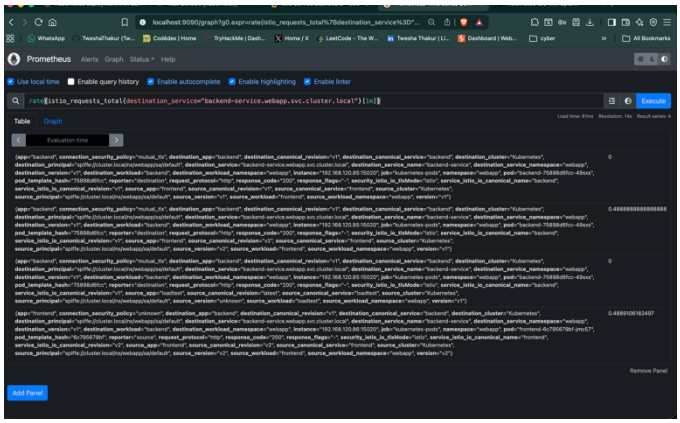
- Installed as Istio addon
- Provides real-time traffic topology visualization
- Displays mTLS status with padlock indicators
- Shows request rates, latencies, and error rates

Prometheus & Grafana:

- Prometheus collects metrics from Istio sidecars
- Grafana provides pre-built Istio dashboards
- Metrics include request rates, latencies, resource usage

**Traffic Generation:** LoadTest pod continuously sends requests through the service mesh to generate observable traffic patterns.




## IV. Security Analysis

*4.1 Defense-in-Depth Implementation*

This project implements multiple security layers:

**Layer 1 - Network (Calico):**

- Controls which pods can communicate (pod-to-pod isolation)
- Enforces ingress/egress rules at the network level
- Blocks unauthorized access attempts

**Layer 2 - Service Mesh (Istio):**

- Automatic mutual TLS encryption between services
- Identity-based authentication using SPIFFE certificates
- Traffic policies and routing rules

**Layer 3 - Application:**

- Health check endpoints
- API endpoint validation
- Resource limits preventing DoS

4.2 Zero-Trust Principles Applied

1. **Verify Explicitly:** Every connection authenticated via mTLS certificates
2. **Least Privilege Access:** Network policies allow only required communication
3. **Assume Breach:** Multiple security layers prevent lateral movement

*4.3 Security Testing*

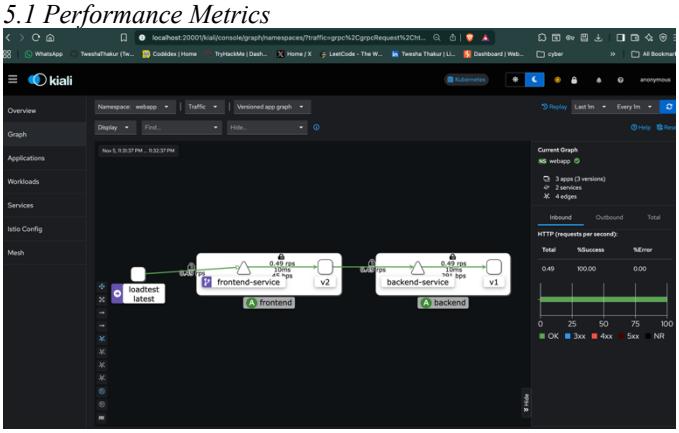
Test 1 - **Unauthorized Access Attempt:** Created a test pod without proper labels attempting to access backend:
bash
kubectl run unauthorized --image=nginx -n webapp
*Attempt to access backend - BLOCKED by network policy*
**Result:** Connection timeout - network policy successfully blocked unauthorized access.
Test 2 - **mTLS Verification:** Using Istio diagnostic tools to verify mutual TLS:
bash
istioctl proxy-config secret [pod-name] -n webapp
*Shows valid certificates issued by Istio CA*

Result:Valid certificates present, mTLS active between all services.

Test 3 - Traffic Interception:
All traffic between pods passes through Envoy proxies, visible in logs:
[Envoy] mTLS connection established
[Envoy] Certificate validation successful

## V. Results and Observations

*5.1 Performance Metrics*




*5.2 Functionality Verification*

Application Endpoints Tested:
- ✅ Frontend web interface accessible
- ✅ Backend health check responding
- ✅ Backend data API returning JSON

- ✅ Frontend → Backend proxy working
- ✅ Traffic flowing through Istio Gateway

Security Verification:
- ✅ Network policies enforced (unauthorized access blocked)
- ✅ mTLS active between all services (verified via Kiali padlock icons)
- ✅ Certificates auto-rotated (verified via Istio config dumps)
- ✅ Traffic observable in real-time (Kiali graph)

## VI. Conclusions

### 6.1 Project Accomplishments

This project successfully demonstrates:

1. **Multi-Layer Security:** Implementation of defense-in-depth with Calico (network) and Istio (application) security layers working in harmony.
2. **Zero-Trust Architecture:** Every connection authenticated and encrypted; no implicit trust based on network location.
3. **Production-Ready Patterns:** Use of industry-standard tools (Calico, Istio) configured according to best practices.
4. **Complex Integration:** Resolved known conflicts between network policies and service mesh—a common real-world challenge.
5. **Observability:** Achieved real-time visibility into secure communications, essential for security operations.

### 6.2 Key Takeaways

**Technical Insights:**

- Network policies and service meshes address different security concerns and should be used together
- Careful port management is critical when integrating multiple security tools
- Observability is essential for verifying security implementations

**Best Practices Applied:**

- Default-deny network policies
- Automatic mTLS without application changes
- Comprehensive health checks and monitoring
- Structured approach to troubleshooting integration issues

### 6.3 Real-World Applicability

This implementation demonstrates patterns directly applicable to production environments:

- **Microservices Security:** Same approach scales to hundreds of services
- **Compliance Requirements:** mTLS and network isolation meet many regulatory standards
- **Zero Downtime Security:** Security added without application code changes
- **Audit Trail:** All traffic logged and observable

### 6.4 Future Enhancements

Potential extensions to this project:

1. **Multi-Namespace Architecture:** Deploy frontend and backend in separate namespaces for stricter isolation
2. **Authorization Policies:** Implement Istio authorization policies for fine-grained access control
3. **Rate Limiting:** Add Istio rate limiting to prevent DoS attacks
4. **Egress Control:** Restrict external API access using Istio egress gateway
5. **Continuous Security:** Integrate with CI/CD for automated security policy testing