Kubernetes Pod Internet Connectivity Issue - Reverse Path Filtering

Date: September 29, 2025

Environment: Contabo VPS, Ubuntu, Kubernetes with Calico/Flannel CNI

Status: ✅ RESOLVED

Keywords: `kubernetes`, `networking`, `rp_filter`, `calico`, `flannel`, `packet-filtering`, `iptables`, `pod-connectivity`, `CNI`, `linux-kernel`, `firewalld`, `UFW`

Executive Summary

Pods in Kubernetes cluster could not access external internet despite correct DNS resolution and routing table configuration. The root cause was Linux kernel's Reverse Path Filtering (`rp_filter`) dropping packets from pod source IPs that didn't match expected routing paths. Resolution required disabling `rp_filter` on all network interfaces and resolving firewall conflicts.

Problem Statement

Symptoms

- ✅ Pods could resolve DNS queries (e.g., `nslookup google.com` worked)
- ✅ Routing table showed correct default gateway
- ✅ TCP port tests succeeded (`nc -zv 8.8.8.8 53`)
- × ICMP ping failed with "Packet filtered" message from host IP
- × HTTP/HTTPS requests timed out
- × No internet connectivity from pods

Initial Observations

bash

```
# From inside pod
ping 8.8.8.8
# Result: From 155.133.23.116 icmp_seq=1 Packet filtered

# DNS worked
nslookup google.com
# Result: Successful resolution

# TCP connectivity worked
nc -zv 8.8.8.8 53
# Result: Connection succeeded
```

Infrastructure Context

Network Architecture

```
┌─────────────────────────────────────────────────────┐
│      Physical Host (155.133.23.116)                 │
│  ┌───────────────────────────────────────────┐      │
│  │ Kubernetes Node                           │      │
│  │  ┌─────────────────────────────────┐      │      │
│  │  │ Pod (10.244.41.52)              │      │      │
│  │  │                                 │      │      │
│  │  │ Needs: Internet Access          │      │      │
│  │  └────────────────┬────────────────┘      │      │
│  │                   │                       │      │
│  │     ┌─────────────▼─────────────┐         │      │
│  │     │ Calico veth               │         │      │
│  │     │ (cali*)                   │         │      │
│  │     └─────────────┬─────────────┘         │      │
│  │                   │                       │      │
│  │     ┌─────────────▼─────────────┐         │      │
│  │     │ flannel.1                 │         │      │
│  │     └─────────────┬─────────────┘         │      │
│  │                   │                       │      │
│  │     ┌─────────────▼─────────────┐         │      │
│  │     │ eth0                      │         │      │
│  │     └─────────────┬─────────────┘         │      │
│  └─────────────────────┼─────────────────────┘      │
└──────────────────────────┼──────────────────────────┘
                           │
```

```
        ▼
    Internet (8.8.8.8)
```

Environment Details

- **OS:** Ubuntu Server
- **Kubernetes:** Single/Multi-node cluster
- **CNI:** Calico + Flannel (dual CNI setup)
- **Pod CIDR:** 10.244.0.0/16
- **Host IP:** 155.133.23.116
- **Firewalls:** UFW (enabled) + firewalld (conflicting)

Root Cause Analysis

1. Reverse Path Filtering (Primary Issue)

What is `rp_filter`?

- Linux kernel security feature that validates packet source addresses
- Located at `/proc/sys/net/ipv4/conf/<interface>/rp_filter`
- Three modes:
  - `0` = Disabled (no filtering)
  - `1` = Strict mode (RFC 3704 strict)
  - `2` = Loose mode (RFC 3704 loose)

The Problem:

```bash
# Initial state showed
/proc/sys/net/ipv4/conf/cali*/rp_filter = 2  # Loose mode
/proc/sys/net/ipv4/conf/flannel.1/rp_filter = 2
/proc/sys/net/ipv4/conf/docker0/rp_filter = 2
```

Packet Flow Analysis:

```
1. Pod sends packet:
   Source: 10.244.41.52 (pod IP)
   Destination: 8.8.8.8

2. Calico/iptables does MASQUERADE:
   Source: 155.133.23.116 (host IP) ← NAT translation
   Destination: 8.8.8.8

3. Packet exits via eth0

4. Reply comes back:
   Source: 8.8.8.8
   Destination: 155.133.23.116

5. Kernel's rp_filter check on eth0:
   "Would I route to 10.244.41.52 via eth0?"
   Answer: NO (10.244.41.0/16 is not in eth0 routing table)

6. Result: PACKET DROPPED ×
```

Why it failed: The kernel's reverse path filter checked if the **original pod source IP** (10.244.41.52) would be routable via the interface receiving the reply (eth0). Since pod IPs are only in internal routing tables, `rp_filter` considered them invalid and dropped the packets.

2. Firewall Conflicts (Secondary Issue)

```bash
# Both firewalls running simultaneously
systemctl status ufw       # active
systemctl status firewalld # active ← CONFLICT!
```

Having two firewall systems created unpredictable rule interactions and blocked ICMP packets.

3. Interface-Specific Settings

Setting `net.ipv4.conf.all.rp_filter=0` didn't propagate to **existing** interfaces:

- Calico veth pairs (cali*)
- Flannel overlay (flannel.1)
- Docker bridge (docker0)

These interfaces retained their default `rp_filter=2` setting.

Solution Implementation

Step 1: Disable Conflicting Firewall

```bash
# Stop and disable firewalld
sudo systemctl stop firewalld
sudo systemctl disable firewalld
```

Rationale: Running UFW and firewalld simultaneously creates rule conflicts.

Step 2: Disable rp_filter on All Interfaces

```bash
# Set global defaults
sudo sysctl -w net.ipv4.conf.all.rp_filter=0
sudo sysctl -w net.ipv4.conf.default.rp_filter=0
sudo sysctl -w net.ipv4.conf.eth0.rp_filter=0

# Fix existing Calico interfaces
for i in /proc/sys/net/ipv4/conf/cali*/rp_filter; do
  echo 0 | sudo tee $i
done

# Fix Flannel interface (note: flannel.1 has a dot)
echo 0 | sudo tee /proc/sys/net/ipv4/conf/flannel.1/rp_filter

# Fix Docker bridge
sudo sysctl -w net.ipv4.conf.docker0.rp_filter=0

# Fix any custom bridges
for i in /proc/sys/net/ipv4/conf/br-*/rp_filter; do
  echo 0 | sudo tee $i
done
```

Step 3: Allow ICMP Traffic

```bash
```

```
# Add iptables rules for ICMP
sudo iptables -A OUTPUT -p icmp --icmp-type echo-request -j ACCEPT
sudo iptables -A INPUT -p icmp --icmp-type echo-reply -j ACCEPT
```

Step 4: Make Changes Permanent

```bash
# Update sysctl.conf
sudo bash -c 'cat >> /etc/sysctl.conf << EOF
# Disable rp_filter for Kubernetes pod networking
net.ipv4.conf.all.rp_filter = 0
net.ipv4.conf.default.rp_filter = 0
net.ipv4.conf.eth0.rp_filter = 0
EOF'

# Create systemd service for interface-specific settings
sudo bash -c 'cat > /etc/systemd/system/fix-rp-filter.service << EOF
[Unit]
Description=Disable rp_filter for Kubernetes networking
After=network.target

[Service]
Type=oneshot
ExecStart=/bin/bash -c "for i in /proc/sys/net/ipv4/conf/*/rp_filter; do echo 0 > \\$i; done"
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
EOF'

# Enable the service
sudo systemctl enable fix-rp-filter.service
sudo systemctl start fix-rp-filter.service
```

Step 5: Verification

```bash
```

```
# From inside a pod
kubectl exec -it <pod-name> -- /bin/bash

# Test connectivity
ping -c 3 8.8.8.8        # ✅ Should succeed
ping -c 3 google.com     # ✅ Should succeed
curl https://ifconfig.me  # ✅ Should return host IP
curl -I https://google.com # ✅ Should return HTTP 200
```

Result:

```
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=114 time=11.6 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=114 time=11.5 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=114 time=11.5 ms
```

✅ **SUCCESS!**

Technical Deep Dive: Why rp_filter Matters for Kubernetes

Kubernetes NAT Flow

```
┌─────────────────────────────────────────────────────┐
│ Step 1: Pod initiates connection                    │
│  ┌──────────┐                                       │
│  │  Pod     │ SRC: 10.244.41.52                     │
│  │10.244    │ DST: 8.8.8.8                          │
│  │.41.52    │                                       │
│  └────┬─────┘                                       │
│       │                                             │
│       ▼                                             │
│  ┌──────────────────────────────────────────┐       │
│  │ Step 2: iptables MASQUERADE              │       │
│  │ (in POSTROUTING chain)                   │       │
│  │                                          │       │
│  │ Changes packet:                          │       │
│  │ SRC: 10.244.41.52 → 155.133.23.116       │       │
│  │ DST: 8.8.8.8 (unchanged)                 │       │
│  └─────────────┬────────────────────────────┘       │
│                │                                    │
│                ▼                                    │
```

```
│    ┌──────────────────┐            │
│    │ eth0       │ Packet exits     │
│    │ 155.133.23.116 │ to internet  │
│    └─────────┬────────┘            │
└──────────────┼───────────────────────────────────┘
     │
     ▼
    ┌────────────────┐
    │ Internet   │
    │ 8.8.8.8   │
    └───────┬────────┘
     │
     │ Reply packet
     │ SRC: 8.8.8.8
     │ DST: 155.133.23.116
     ▼
┌──────────────────────────────────────────────────┐
│    ┌──────────────────┐            │
│    │ eth0       │ Reply arrives    │
│    │ 155.133.23.116 │               │
│    └─────────┬────────┘            │
│      │              │
│      ▼              │
│  ┌────────────────────────────────────────┐      │
│  │ Step 3: rp_filter CHECK (if enabled) │     │
│  │              │     │
│  │ Question: "Would I route packets   │     │
│  │ to 10.244.41.52 via eth0?"     │     │
│  │              │     │
│  │ Routing Table Check:         │     │
│  │ - 10.244.0.0/16 → caliXXX, flannel │     │
│  │ - NOT via eth0          │     │
│  │              │     │
│  │ Answer: NO            │     │
│  │ Action: DROP PACKET ❌        │     │
│  └─────────────────────────────────────────┘     │
│                    │
│ With rp_filter=0:              │
│  ┌─────────────────────────────────────────┐     │
│  │ Step 3: NO rp_filter check     │     │
│  │ Packet passes through ✅       │     │
│  └────────┬────────────────────────────────┘     │
│     │               │
│     ▼               │
```

```
│ ┌───────────────────────────────────────────┐     │
│ │ Step 4: Connection tracking        │      │
│ │ (conntrack) restores original flow   │      │
│ │                     │      │
│ │ Changes packet back:          │      │
│ │ SRC: 8.8.8.8             │      │
│ │ DST: 155.133.23.116 → 10.244.41.52  │      │
│ └─────────┬─────────────────────────────────┘     │
│    │                │
│    ▼                │
│   ┌─────────┐               │
│   │ Pod  │ Packet delivered! ✅        │
│   │10.244 │              │
│   │.41.52 │              │
│   └─────────┘               │
└───────────────────────────────────────────────────────────┘
```

Why Default rp_filter Doesn't Work

When you set:

```bash
net.ipv4.conf.all.rp_filter = 0
net.ipv4.conf.default.rp_filter = 0
```

This affects:

- ✅ Future interfaces created after this setting
- × Existing interfaces already created (like calico veth pairs)

The kernel checks the **most specific** setting:

```bash
# Priority order (most specific wins):
1. net.ipv4.conf.<interface>.rp_filter  # Highest priority
2. net.ipv4.conf.all.rp_filter
3. net.ipv4.conf.default.rp_filter     # Lowest priority
```

Diagnostic Commands Reference

Check rp_filter on all interfaces

```bash
for i in /proc/sys/net/ipv4/conf/*/rp_filter; do
  echo "$i = $(cat $i)"
done
```

Check routing table

```bash
# From inside pod
ip route

# From host
kubectl exec -it <pod-name> -- ip route
```

Check iptables NAT rules

```bash
sudo iptables -t nat -L POSTROUTING -v -n | grep MASQUERADE
```

Check firewall status

```bash
sudo systemctl status ufw
sudo systemctl status firewalld
sudo ufw status
```

Test connectivity levels

```bash
```

```bash
# Layer 3 (ICMP)
ping -c 3 8.8.8.8

# Layer 4 (TCP)
nc -zv 8.8.8.8 53
nc -zv 1.1.1.1 443

# Layer 7 (HTTP)
curl -I https://google.com
curl https://ifconfig.me
```

Check connection tracking

```bash
sudo conntrack -L | grep <pod-ip>
```

Monitor packet filtering in real-time

```bash
# Watch iptables counters
watch -n1 'sudo iptables -L -v -n | head -30'

# Check for dropped packets
sudo iptables -L -v -n | grep DROP
```

Key Learnings

1. CNI Networking Complexity

Kubernetes CNI plugins create complex virtual networking requiring careful kernel parameter tuning. Default security settings may conflict with pod networking requirements.

2. rp_filter Behavior

- Setting `all` and `default` doesn't retroactively update existing interfaces
- Each interface maintains its own independent setting
- Wildcard patterns don't work with `sysctl` for interface names containing special characters (like `flannel.1`)

3. Firewall Interactions

Running multiple firewall systems (UFW + firewalld) creates unpredictable behavior. Choose one and disable others.

4. Diagnostic Approach

- Start with packet flow analysis
- Test at each OSI layer (ICMP, TCP, HTTP)
- Check kernel parameters, not just iptables rules
- Verify settings on ALL network interfaces, not just main interface

5. Kubernetes-Specific Networking

- Pod networking uses NAT/MASQUERADE for external connectivity
- CNI plugins create dynamic network interfaces that need special configuration
- Default Linux security settings aren't always compatible with container networking

---

Prevention & Best Practices

1. Pre-Configure Host for Kubernetes

```bash
# Before installing Kubernetes
cat >> /etc/sysctl.d/99-kubernetes.conf << EOF
net.ipv4.conf.all.rp_filter = 0
net.ipv4.conf.default.rp_filter = 0
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF

sysctl --system
```

2. Choose One Firewall

```bash
```

```
# Disable firewalld if using UFW
systemctl stop firewalld
systemctl disable firewalld
systemctl mask firewalld
```

3. Create Systemd Service for Interface Settings

Use the systemd service approach to handle dynamically created interfaces.

4. Document Network Architecture

Maintain clear diagrams of your network topology, CNI configuration, and traffic flow.

5. Test After Changes

Always verify pod connectivity after infrastructure changes:

```bash
kubectl run test-pod --image=busybox --rm -it -- sh
# Then test: wget -O- http://google.com
```

References & Further Reading

Official Documentation

- **Kubernetes Networking:** https://kubernetes.io/docs/concepts/cluster-administration/networking/
- **Calico Documentation:** https://docs.projectcalico.org/
- **Flannel Documentation:** https://github.com/flannel-io/flannel

Linux Kernel Documentation

- **rp_filter:** https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt
- **iptables/netfilter:** https://netfilter.org/documentation/

RFC Standards

- **RFC 3704:** Ingress Filtering for Multihomed Networks (BCP 84)
- **RFC 1918:** Address Allocation for Private Internets

Related Issues

- Kubernetes GitHub: Issues tagged with `networking` and `CNI`
- Common pod connectivity problems in bare-metal clusters

Appendix: Complete Command Sequence

Initial Diagnosis

```bash
# Check interface rp_filter settings
for i in /proc/sys/net/ipv4/conf/*/rp_filter; do echo "$i = $(cat $i)"; done

# Check firewalls
systemctl status ufw
systemctl status firewalld

# Check iptables
sudo iptables -L -v -n | grep DROP
sudo iptables -t nat -L POSTROUTING -v -n
```

Full Resolution

```bash
```

```
# 1. Stop conflicting firewall
sudo systemctl stop firewalld
sudo systemctl disable firewalld

# 2. Set global defaults
sudo sysctl -w net.ipv4.conf.all.rp_filter=0
sudo sysctl -w net.ipv4.conf.default.rp_filter=0
sudo sysctl -w net.ipv4.conf.eth0.rp_filter=0

# 3. Fix all interfaces
for i in /proc/sys/net/ipv4/conf/cali*/rp_filter; do echo 0 | sudo tee $i; done
for i in /proc/sys/net/ipv4/conf/br-*/rp_filter; do echo 0 | sudo tee $i; done
echo 0 | sudo tee /proc/sys/net/ipv4/conf/flannel.1/rp_filter
sudo sysctl -w net.ipv4.conf.docker0.rp_filter=0
sudo sysctl -w net.ipv4.conf.tunl0.rp_filter=0

# 4. Allow ICMP
sudo iptables -A OUTPUT -p icmp --icmp-type echo-request -j ACCEPT
sudo iptables -A INPUT -p icmp --icmp-type echo-reply -j ACCEPT

# 5. Make permanent
sudo bash -c 'cat >> /etc/sysctl.conf << EOF
net.ipv4.conf.all.rp_filter = 0
net.ipv4.conf.default.rp_filter = 0
net.ipv4.conf.eth0.rp_filter = 0
EOF'

# 6. Create systemd service
sudo bash -c 'cat > /etc/systemd/system/fix-rp-filter.service << EOF
[Unit]
Description=Disable rp_filter for Kubernetes networking
After=network.target

[Service]
Type=oneshot
ExecStart=/bin/bash -c "for i in /proc/sys/net/ipv4/conf/*/rp_filter; do echo 0 > \$i; done"
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
EOF'

sudo systemctl enable fix-rp-filter.service
```

```
sudo systemctl start fix-rp-filter.service

# 7. Verify
kubectl exec -it <pod-name> -- ping -c 3 8.8.8.8
```

Conclusion

This incident demonstrates the importance of understanding Linux kernel networking parameters when deploying Kubernetes clusters on bare-metal or VPS infrastructure. While `rp_filter` provides legitimate security benefits for traditional servers, it conflicts with the NAT-based networking model used by Kubernetes CNI plugins.

The resolution required:

1. Recognizing that packet filtering was occurring at the kernel level, not just iptables
2. Understanding the interaction between `rp_filter` and NAT/MASQUERADE
3. Systematically checking and updating all network interfaces
4. Resolving firewall conflicts
5. Making configuration persistent across reboots

Time to Resolution: ~2 hours of systematic debugging
Impact: All pods in cluster gained internet connectivity
Permanence: Configuration persists across reboots via systemd service

---

Document Version: 1.0
Last Updated: September 29, 2025
Author: Infrastructure Team
Next Review: Before next Kubernetes upgrade