Part 8 - Kubernetes Real-Time Troubleshooting

Introduction 🌐

Welcome to the world of Kubernetes troubleshooting, where every challenge is an opportunity to sharpen your skills and emerge victorious. Join us as we embark on a journey through common real-time scenarios, unraveling mysteries, and uncovering solutions along the way.

Scenario 36: Kubernetes Component Version Mismatch

```
muhammad_khabbab@cloudshell:~ (utopian-precept-406712)$ kubectl version --output=yaml
clientVersion:
  buildDate: "2023-09-13T09:35:49Z"
  compiler: gc
  gitCommit: 89a4ea3e1e4ddd7f7572286090359983e0387b2f
  gitTreeState: clean
  gitVersion: v1.28.2
  goVersion: go1.20.8
  major: "1"
  minor: "28"
  platform: linux/amd64
kustomizeVersion: v5.0.4-0.20230601165947-6ce0bf390ce3
serverVersion:
  buildDate: "2023-10-25T09:31:59Z"
  compiler: gc
  gitCommit: 5e57f46af978babbaddc685365fd47e18625a32b
  gitTreeState: clean
  gitVersion: v1.27.7-gke.1056000
  goVersion: go1.20.10 X:boringcrypto
  major: "1"
  minor: "27"
  platform: linux/amd64
```

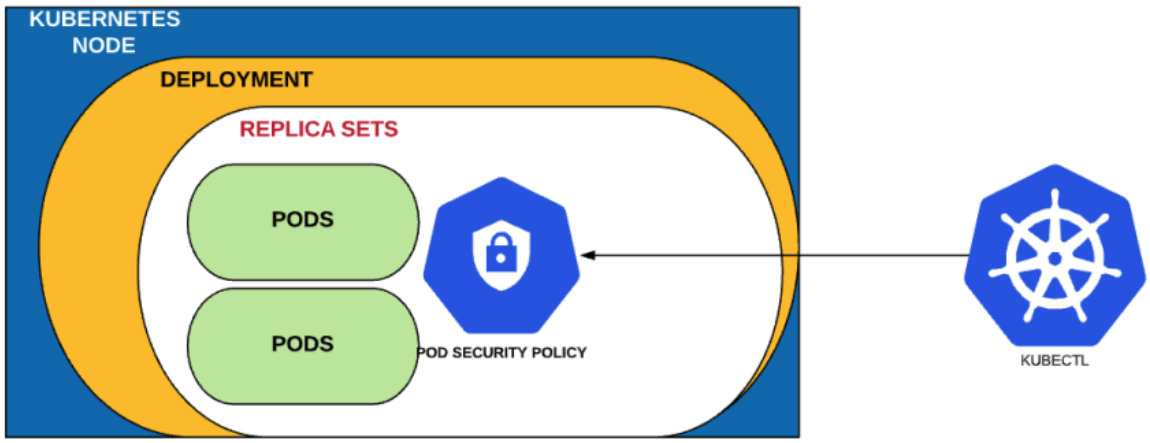
Symptoms: Kubernetes components (e.g., kubelet, kube-proxy, etcd) are running different versions across cluster nodes, causing compatibility issues and potential service disruptions.

Diagnosis: Compare Kubernetes component versions (kubectl version) across cluster nodes and review system logs (journalctl, /var/log/kubelet.log) for any version-related errors or inconsistencies.

Solution:

1. Upgrade Kubernetes components to the same version across all cluster nodes to ensure compatibility and consistency in cluster behaviour and functionality.
2. Implement automated version management and deployment strategies (e.g., Kubernetes rolling upgrades, Kubeadm upgrade) to streamline the process of upgrading Kubernetes components and maintaining version consistency.
3. Monitor Kubernetes component versions using version control systems (e.g., Git, Helm charts) and configuration management tools (e.g., Ansible, Puppet) to detect and remediate version discrepancies proactively.
4. Perform regression testing and validation of Kubernetes upgrades in a staging environment before rolling out changes to production clusters to identify and mitigate any potential compatibility issues or regressions.

Scenario 37: Pod Security Policy Violations



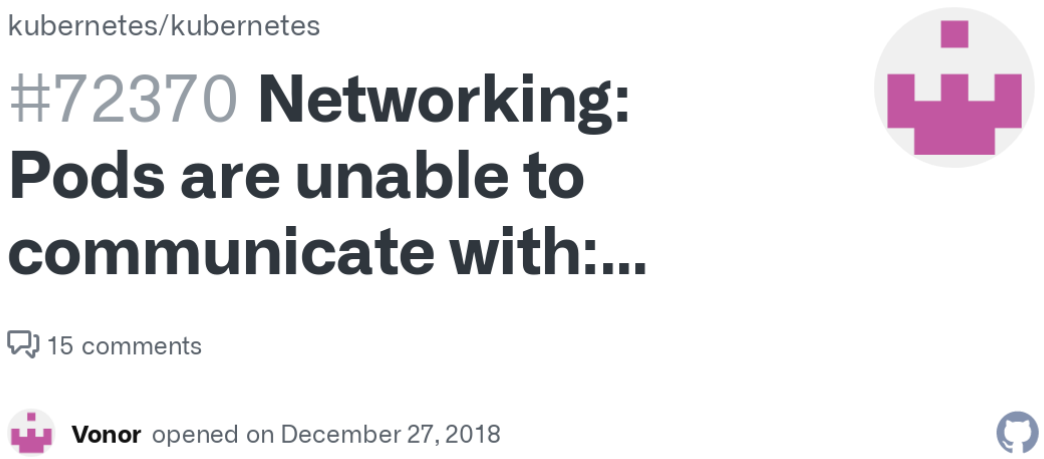
Symptoms: Pods are prevented from starting or running due to security policy violations, such as forbidden container capabilities, privileged access, or insecure volume mounts.

Diagnosis: Review PodSecurityPolicy (PSP) configurations (kubectl get psp) and inspect pod security context (kubectl describe pod <pod_name>) for any policy violations or security-related errors.

Solution:

1. Update PodSecurityPolicy definitions to align with organizational security requirements and best practices for container runtime security, including restricting privileged access, limiting container capabilities, and enforcing least privilege principles.
2. Configure admission controllers (e.g., PodSecurityPolicy admission controller, OPA Gatekeeper) to enforce PSPs and prevent deployment of pods that violate security policies or pose security risks.
3. Enable pod security context settings (e.g., securityContext, runAsUser, readOnlyRootFilesystem) to enforce security controls and mitigate common attack vectors, such as privilege escalation, container breakout, or file system tampering.
4. Implement container image scanning and vulnerability management solutions to detect and remediate security vulnerabilities in container images before deployment and runtime execution.

Scenario 38: Network Policy Configuration Errors



Symptoms: Pods are unable to communicate with each other or with external services due to misconfigured network policies, resulting in network connectivity issues and service disruptions.

Diagnosis: Review Kubernetes network policy configurations (kubectl get networkpolicy) and inspect network traffic flows (kubectl exec -it <pod_name> -- /bin/sh) to identify any policy violations or connectivity problems.

Solution:

1. Validate network policy definitions to ensure that they accurately reflect the intended network segmentation and access control requirements for pod-to-pod or pod-to-service communication.

2. Use network policy validation tools (e.g., kube-score, kyverno) to identify potential misconfigurations or security gaps in network policy definitions and remediate them accordingly.
3. Implement network policy logging and monitoring solutions to track network traffic patterns and policy enforcement actions, helping to diagnose and troubleshoot connectivity issues more effectively.
4. Conduct regular network policy reviews and audits to verify compliance with organizational security policies and industry best practices for network segmentation and isolation.

Scenario 39: Ingress Controller Configuration Errors

Symptoms: Ingress resources fail to route incoming traffic to backend services or exhibit unexpected behavior due to misconfigured or malfunctioning ingress controller configurations.

Diagnosis: Review Ingress controller logs (kubectl logs -n <ingress_controller_namespace> <ingress_controller_pod>) and inspect Ingress resource definitions (kubectl describe ingress <ingress_name>) for any errors or inconsistencies.

Solution:

1. Verify Ingress controller deployment configurations (e.g., annotations, ingress class, service backends) and ensure that they align with the desired routing and traffic management requirements for incoming requests.
2. Use Ingress controller health checks and readiness probes to monitor the availability and responsiveness of the Ingress controller instances and detect any failures or performance issues.
3. Test Ingress routing rules and configurations using HTTP request simulation tools (e.g., curl, Postman) to validate that traffic is being routed correctly to the intended backend services and endpoints.

4. Update and reload Ingress controller configurations dynamically using configuration management tools (e.g., kubectl apply, Helm charts) to apply changes and updates to routing rules and settings without disrupting existing traffic flows.

Scenario 40: Cluster DNS Resolution Issues

Symptoms: Pods are unable to resolve DNS names or domain names due to DNS resolution failures or misconfigured cluster DNS settings, causing service discovery and communication problems.

Diagnosis: Check cluster DNS configuration (kubectl get cm -n kube-system coredns) and inspect pod DNS configurations (kubectl exec -it <pod_name> -- cat /etc/resolv.conf) for any DNS-related errors or inconsistencies.

Solution:

1. Verify that CoreDNS or kube-dns pods are running and healthy (kubectl get pods -n kube-system) and inspect their logs (kubectl logs -n kube-system coredns-<pod_id>) for any DNS resolution errors or warnings.
2. Troubleshoot DNS name resolution issues by testing DNS queries (nslookup, dig) from within pods and nodes to verify connectivity to DNS servers and resolve DNS records accurately.
3. Check firewall and network policies to ensure that DNS traffic is allowed and not blocked by network security rules or restrictions, both within the cluster and with external DNS servers.
4. Update DNS configuration settings (e.g., cluster DNS domain, upstream DNS servers) and restart CoreDNS or kube-dns pods to apply changes and propagate DNS configuration updates across the cluster.

In the up-coming parts, we will discussion on more troubleshooting steps for the different Kubernetes based scenarios. So, stay tuned for the and follow @Prasad Suman Mohan for more such posts.

Follow - https://www.linkedin.com/in/prasad-suman-mohan (for updates)