Setting Up Prometheus Monitoring and Slack Alerts in Kubernetes

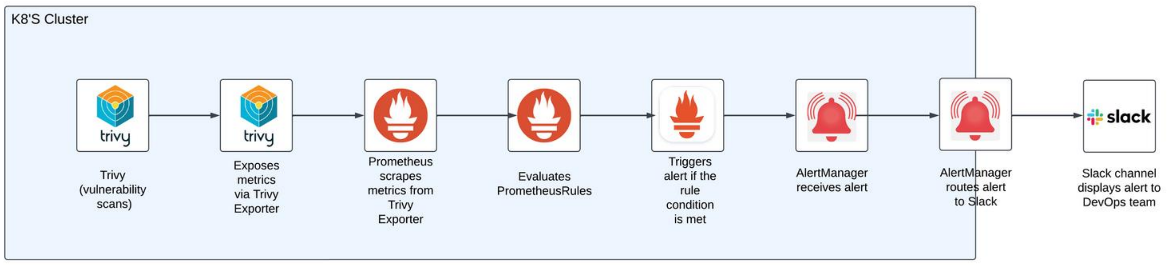
Introduction

Monitoring and alerting are essential for maintaining the health, performance, and security of Kubernetes clusters. By integrating **Prometheus** for monitoring, **AlertManager** for alerting, and **Slack** for notifications, you can proactively address critical issues, such as vulnerabilities detected by **Trivy**. This guide simplifies and deepens the process of configuring these tools, ensuring clarity at every step.

Objective

Goals

1. Install and configure the **kube-prometheus-stack** Helm chart for Prometheus and AlertManager.
2. Set up **PrometheusRule CRDs** to define conditions for alerts.
3. Configure **AlertManager** to route alerts to a Slack channel.
4. Test and verify the setup end-to-end, resolving common issues encountered in real-world scenarios.


K8'S Cluster
trivy
Trivy (vulnerability scans)
trivy
Exposes metrics via Trivy Exporter
Prometheus scrapes metrics from Trivy Exporter
Evaluates PrometheusRules
Triggers alert if the rule condition is met
AlertManager receives alert
AlertManager routes alert to Slack
slack
Slack channel displays alert to DevOps team

Step-by-Step Setup

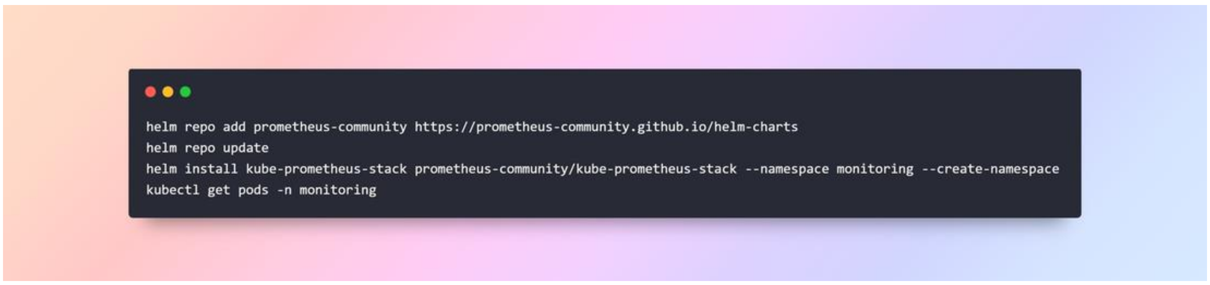
1. Install kube-prometheus-stack Helm Chart

Prerequisites

- **Kubernetes cluster** with sufficient resources.
- Helm installed on your local machine.
- kubectl is configured to access your cluster.

Installation

Add the Prometheus community Helm repository and install the stack:



```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
helm install kube-prometheus-stack prometheus-community/kube-prometheus-stack --namespace monitoring --create-namespace
kubectl get pods -n monitoring
```

Key components include:

- **Prometheus**: Collects and stores metrics.
- **AlertManager**: Sends alerts to destinations like Slack.
- **Grafana**: Provides dashboards for visualization.

2. Configure Prometheus and AlertManager

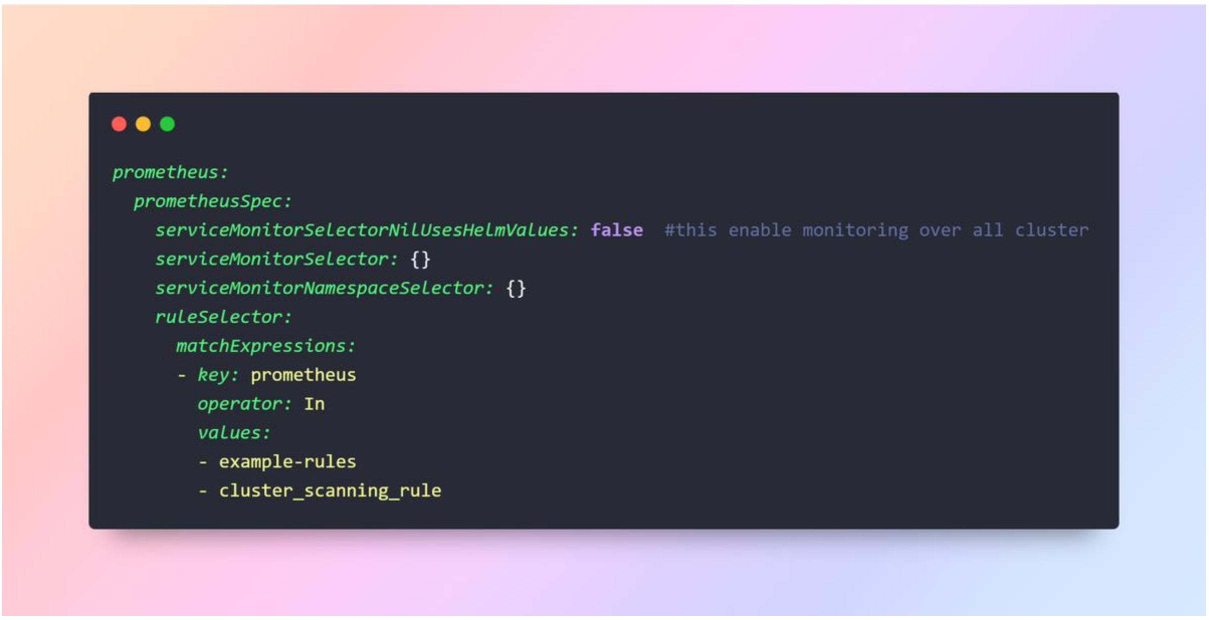
Modify, observability-conf\prom-values-alertmanager.yaml

- To make Prometheus discover custom rules and configure Slack notifications, update the values.yaml file. This step is critical and often overlooked in existing documentation clone Github repo and update, observability-conf\prom-values-alertmanager.yaml

https://github.com/sarvadnyaJawale/monitor-security-olly-in-Kubernetes-.git

Prometheus **RuleSelector** and **ServiceMonitorSelector**

- This ensures Prometheus picks up the custom alert rules and monitors:

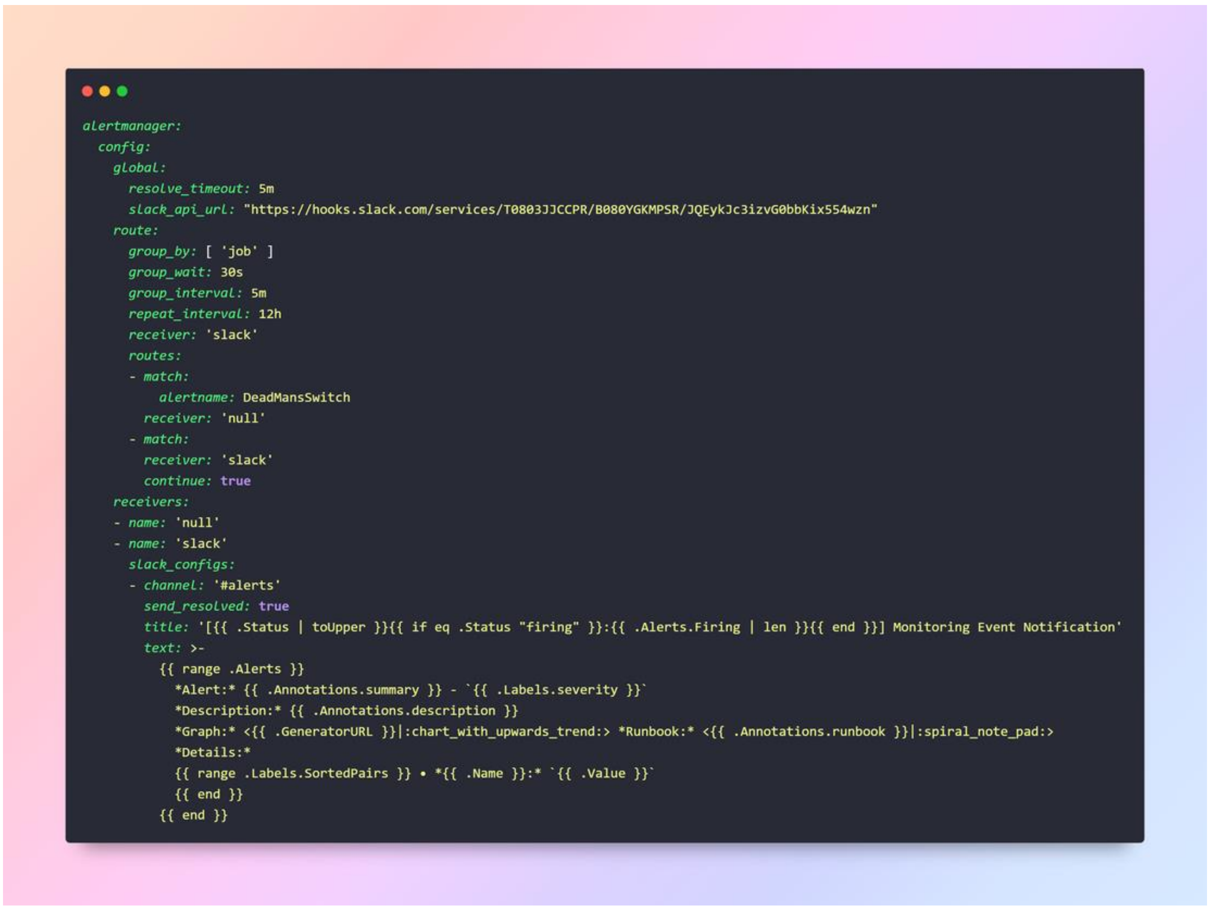


```yaml
prometheus:
  prometheusSpec:
    serviceMonitorSelectorNilUsesHelmValues: false  #this enable monitoring over all cluster
    serviceMonitorSelector: {}
    serviceMonitorNamespaceSelector: {}
    ruleSelector:
      matchExpressions:
      - key: prometheus
        operator: In
        values:
        - example-rules
        - cluster_scanning_rule
```

- **ruleSelector**: Filters PrometheusRules by label. Here, only rules labeled prometheus: cluster_scanning_rule are considered.
- **serviceMonitorSelectorNilUsesHelmValues: false**: Ensures only explicitly defined ServiceMonitors are used, avoiding unwanted defaults.

AlertManager Configuration for Slack

Configure Slack as the alert receiver:

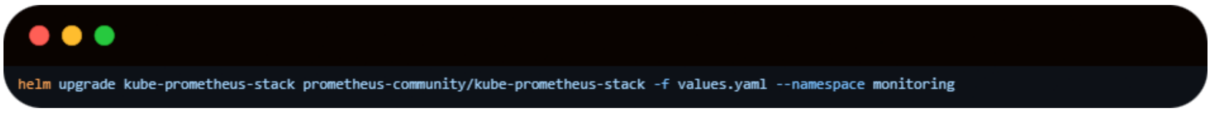


```yaml
alertmanager:
  config:
    global:
      resolve_timeout: 5m
      slack_api_url: "https://hooks.slack.com/services/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"
    route:
      group_by: [ 'job' ]
      group_wait: 30s
      group_interval: 5m
      repeat_interval: 12h
      receiver: 'slack'
      routes:
      - match:
          alertname: DeadMansSwitch
        receiver: 'null'
      - match:
        receiver: 'slack'
        continue: true
    receivers:
    - name: 'null'
    - name: 'slack'
      slack_configs:
      - channel: '#alerts'
        send_resolved: true
        title: '[{{ .Status | toUpper }}{{ if eq .Status "firing" }}:{{ .Alerts.Firing | len }}{{ end }}] Monitoring Event Notification'
        text: >-
          {{ range .Alerts }}
            *Alert:* {{ .Annotations.summary }} - `{{ .Labels.severity }}`
            *Description:* {{ .Annotations.description }}
            *Graph:* <{{ .GeneratorURL }}|:chart_with_upwards_trend:> *Runbook:* <{{ .Annotations.runbook }}|:spiral_note_pad:>
            *Details:*
            {{ range .Labels.SortedPairs }} • *{{ .Name }}:* `{{ .Value }}`
            {{ end }}
          {{ end }}
```

- **resolve_timeout**: Time after which unresolved alerts are dropped.
- **slack_api_url**: Webhook URL for Slack integration.
- **route**: Defines how alerts are grouped and forwarded.
- **receivers**: Configures Slack as the destination for alerts.

Apply Updated Configuration

- Upgrade the Helm chart with the modified prom-values-alertmanager.yaml

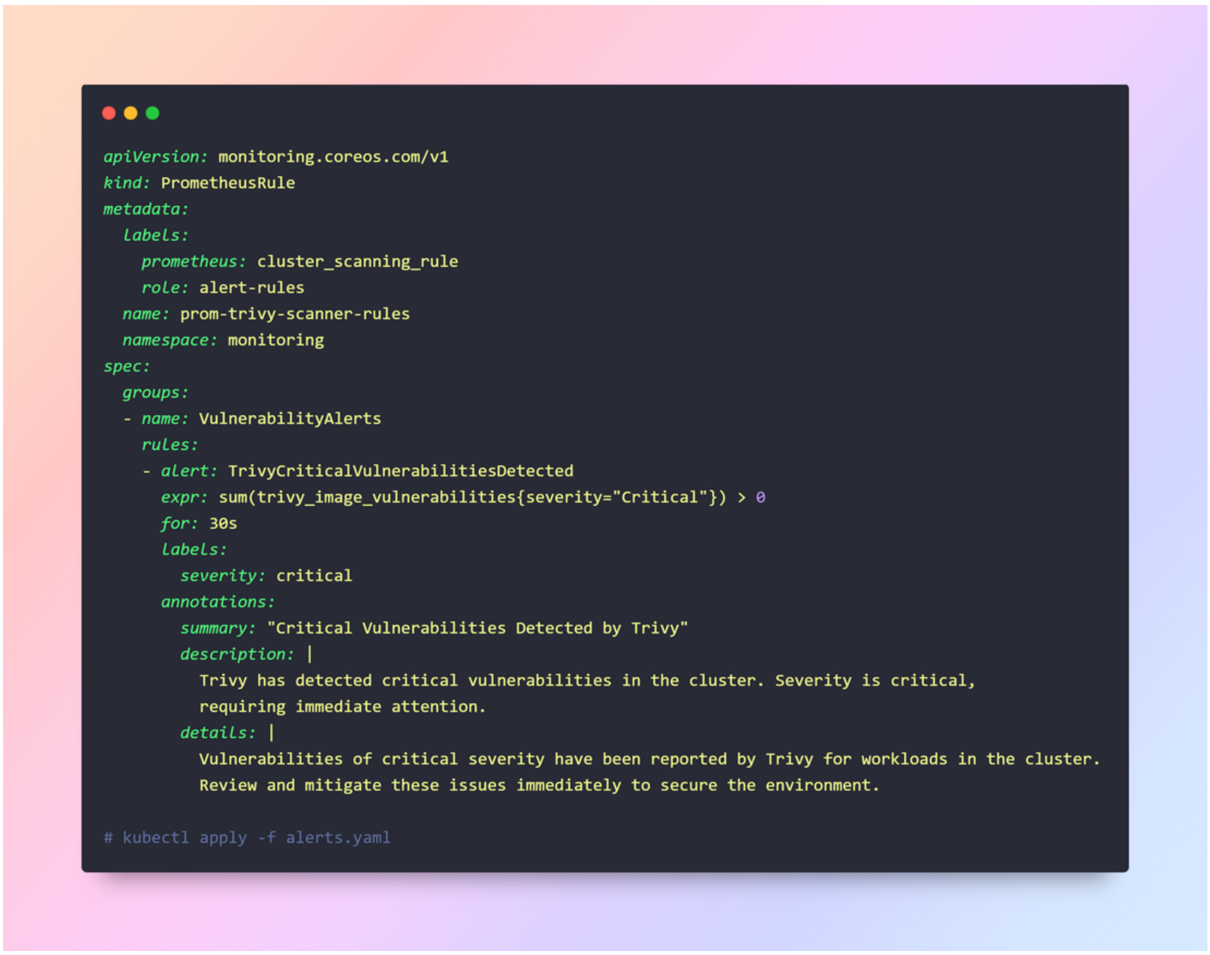


```
helm upgrade kube-prometheus-stack prometheus-community/kube-prometheus-stack -f values.yaml --namespace monitoring
```

This step applies your configurations to Prometheus and AlertManager. Though seemingly simple, this step is often omitted in documentation.

Create PrometheusRule CRDs

- PrometheusRules define alert conditions.

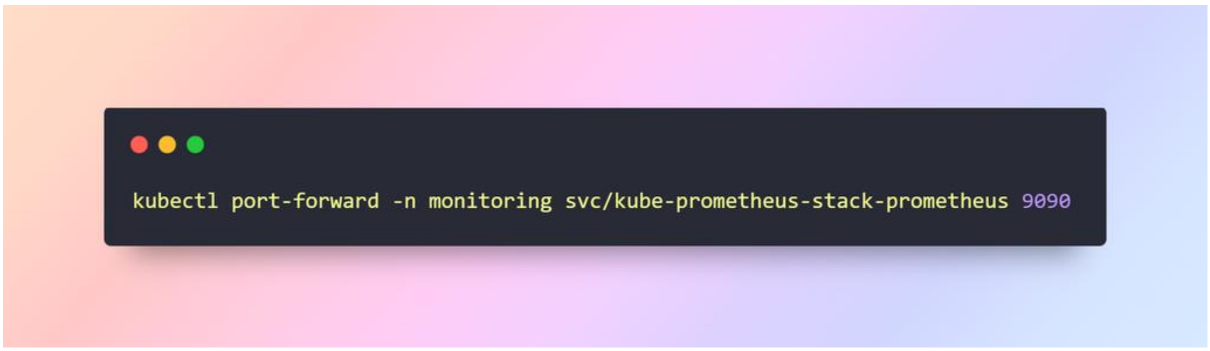


```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    prometheus: cluster_scanning_rule
    role: alert-rules
  name: prom-trivy-scanner-rules
  namespace: monitoring
spec:
  groups:
  - name: VulnerabilityAlerts
    rules:
    - alert: TrivyCriticalVulnerabilitiesDetected
      expr: sum(trivy_image_vulnerabilities{severity="Critical"}) > 0
      for: 30s
      labels:
        severity: critical
      annotations:
        summary: "Critical Vulnerabilities Detected by Trivy"
        description: |
          Trivy has detected critical vulnerabilities in the cluster. Severity is critical,
          requiring immediate attention.
        details: |
          Vulnerabilities of critical severity have been reported by Trivy for workloads in the cluster.
          Review and mitigate these issues immediately to secure the environment.

# kubectl apply -f alerts.yaml
```

Verify Setup

- Expose Prometheus UI
- Port-forward Prometheus to access its web UI:



```
kubectl port-forward -n monitoring svc/kube-prometheus-stack-prometheus 9090
```

Verify:

1. **Metrics**: Check if Trivy metrics are scraped under **Targets → trivy-exporter**.
2. **Rules**: Ensure trivy-alerts.yaml rules are loaded under **Status → Rules**.

Simulate an Alert

Induce a test condition (e.g., deploy a vulnerable container). Check:

- Alerts are displayed under **Alerts** in Prometheus.
- Notifications are sent to the Slack channel.

Workflow: Trivy to Slack Alerts

Trivy Scans and Metrics Export:

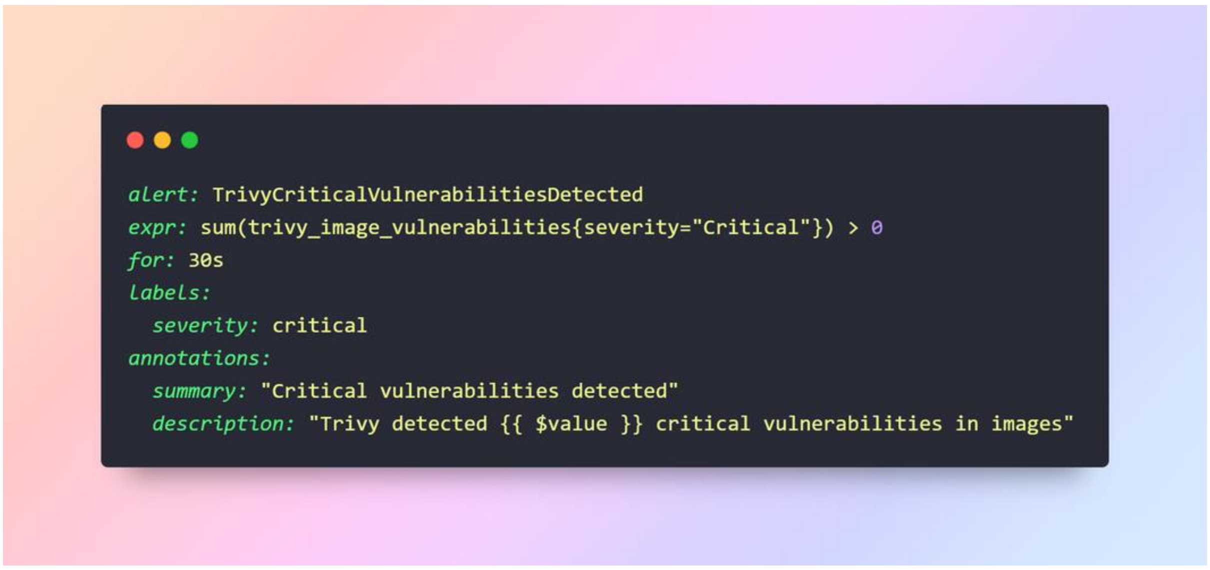
- **Process**:
  - Trivy runs as a **Kubernetes DaemonSet or CronJob** to scan container images in the cluster for vulnerabilities.
  - The results are converted into **Prometheus-compatible metrics** and exposed via the **Trivy Exporter** at an HTTP endpoint (/metrics).
- **Technical Note**:
  - Trivy's metrics include fields like trivy_image_vulnerabilities{severity="Critical"}.
  - The **Trivy Exporter Service** in the trivy-system namespace provides these metrics.

Prometheus Scrapes Metrics:

- **Process**:
  - Prometheus, deployed via the kube-prometheus-stack, scrapes the Trivy metrics from the **Trivy Exporter Service**.
  - Metrics scraping is configured via a **ServiceMonitor CRD**, which specifies the namespace, service, and endpoints.
- **Key Configuration**:
  - serviceMonitorSelector in Prometheus's values.yaml ensures the discovery of the Trivy ServiceMonitor.

Rule Evaluation in Prometheus:

- **Process**:
  - Prometheus evaluates custom **PrometheusRules** (e.g., trivy-alerts.yaml) defined for Trivy metrics.

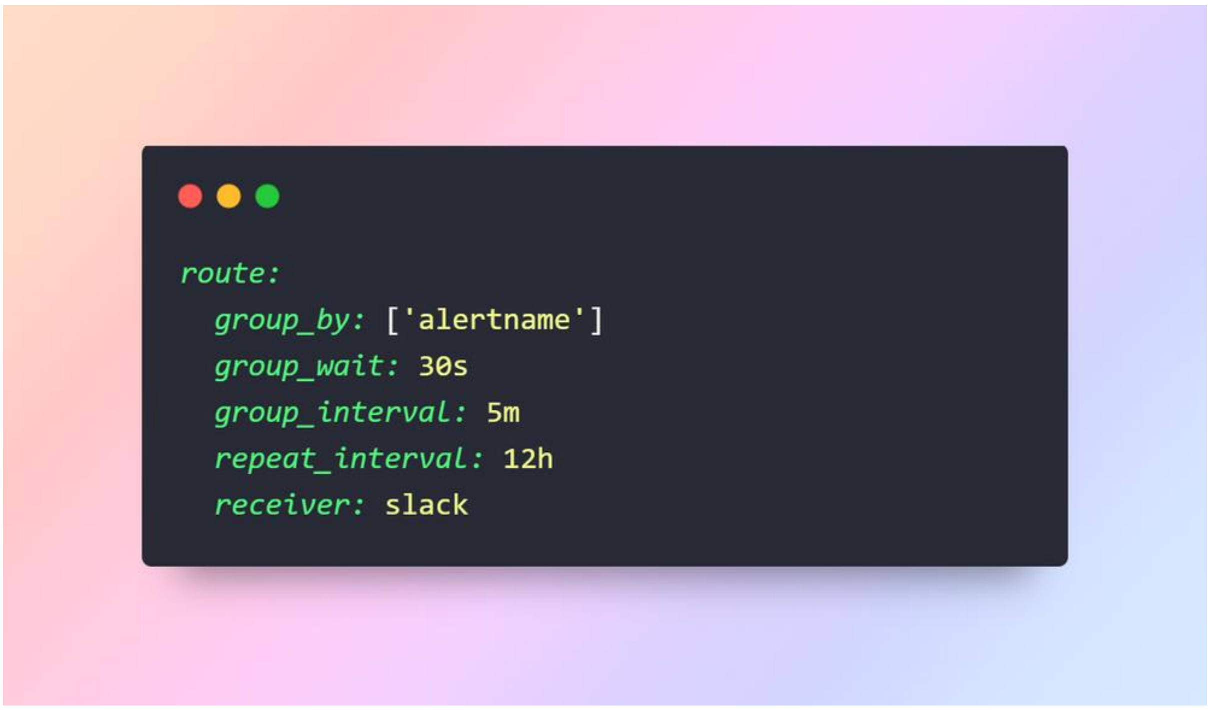


```yaml
alert: TrivyCriticalVulnerabilitiesDetected
expr: sum(trivy_image_vulnerabilities{severity="Critical"}) > 0
for: 30s
labels:
  severity: critical
annotations:
  summary: "Critical vulnerabilities detected"
  description: "Trivy detected {{ $value }} critical vulnerabilities in images"
```

- If the condition (expression) evaluates to true, an alert is triggered.

Alert Forwarding to AlertManager:

- **Process**:
  - Prometheus forwards triggered alerts to **AlertManager**, which is configured to handle alert routing.
- **Configuration**
  alertmanager.config specifies alert grouping and forwarding rules:

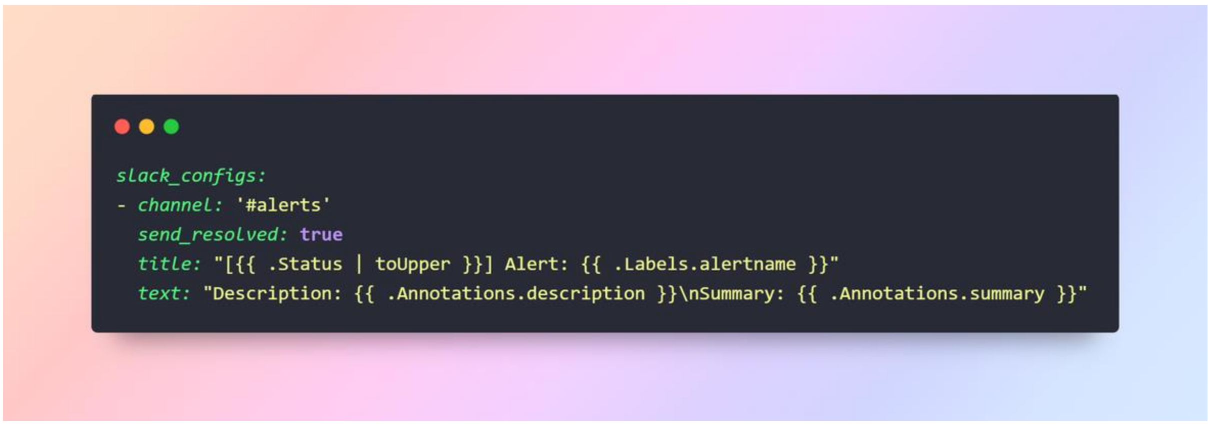


```yaml
route:
  group_by: ['alertname']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 12h
  receiver: slack
```

- Alerts are routed to the slack receiver.

AlertManager Sends Notifications to Slack:

- **Process**:
  - AlertManager uses the configured **Slack API webhook** to send alert notifications to the specified Slack channel.
- **Key Slack Configuration**:
  - slack_configsin AlertManager's receiver definition:
  - yaml
  - Copy code

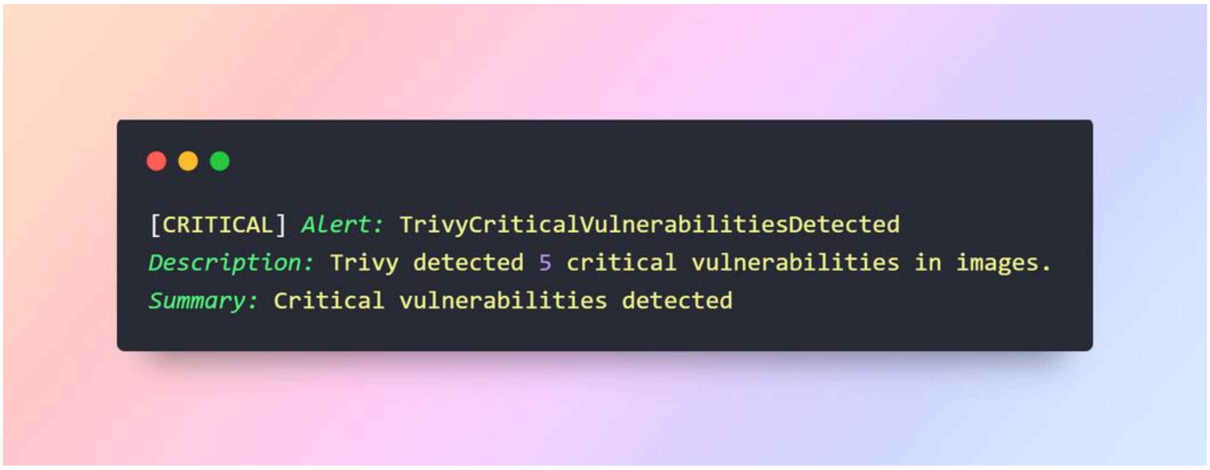


```yaml
slack_configs:
- channel: '#alerts'
  send_resolved: true
  title: "[{{ .Status | toUpper }}] Alert: {{ .Labels.alertname }}"
  text: "Description: {{ .Annotations.description }}\nSummary: {{ .Annotations.summary }}"
```

Notification Reaches Slack Channel:

- **Outcome**:
  - The Slack channel (e.g., #alerts) receives a detailed alert message, enabling DevOps teams to act quickly.

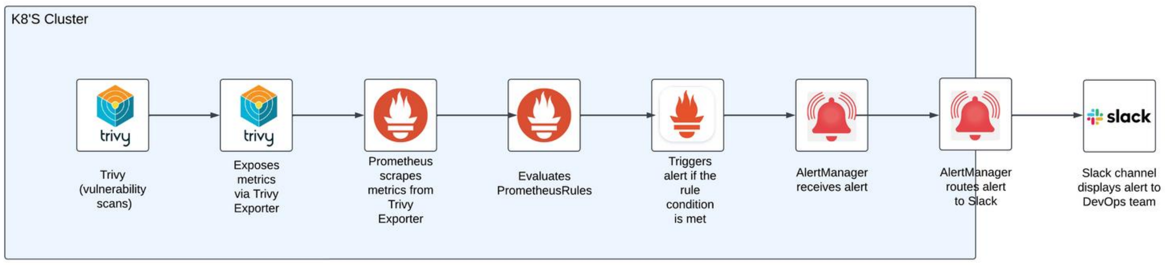
Sample Slack Message:



```
[CRITICAL] Alert: TrivyCriticalVulnerabilitiesDetected
Description: Trivy detected 5 critical vulnerabilities in images.
Summary: Critical vulnerabilities detected
```

Technical Flow Summary

1. **Trivy (Vulnerability Scanner)**:
   - Runs scans → Generates metrics → Exposes metrics via /metrics.
1. **Prometheus**:
   - Scrapes metrics from Trivy → Evaluates rules → Triggers alerts for critical conditions.
1. **AlertManager**:
   - Receives alerts from Prometheus → Routes alerts to Slack based on routing rules.
1. **Slack**:
   - Displays notifications to the configured channel for operational visibility.


K8'S Cluster
Trivy (vulnerability scans)
Exposes metrics via Trivy Exporter
Prometheus scrapes metrics from Trivy Exporter
Evaluates PrometheusRules
Triggers alert if the rule condition is met
AlertManager receives alert
AlertManager routes alert to Slack
Slack channel displays alert to DevOps team

Common Challenges and Resolutions

1. Prometheus Rules Not Triggering Alerts

- **Cause**: Namespace mismatch or missing ruleSelector.
- **Solution**: Match PrometheusRule namespace (monitoring) and ensure proper ruleSelector.

2. Metrics Not Visible

- **Cause**: Trivy metrics not scraped.
- **Solution**: Ensure Trivy metrics are exposed and serviceMonitorSelector targets the namespace.

3. Slack Notifications Failing

- **Cause**: Invalid slack_api_url or routing issues.
- **Solution**: Verify Slack webhook URL and ensure AlertManager routes point to the Slack receiver.

Component Breakdown

1. **Prometheus**: Scrapes metrics and evaluates rules.
2. **AlertManager**: Routes alerts based on rules to configured receivers (e.g., Slack).
3. **Trivy**: A vulnerability scanner that generates metrics for container images.
4. **Slack Integration**: Ensures rapid visibility for critical alerts.

Best Practices

1. **Namespace Consistency**: Align namespaces across Prometheus, AlertManager, and Trivy.
2. **Rule Testing**: Validate all alert rules in a staging environment before production.
3. **Documentation**: Maintain clear documentation for rule configurations and alert workflows.
4. **Helm Upgrade**: Always use helm upgrade after modifying values.yaml

Takeaways and Further Resources

Lessons Learned

- Proper values.yaml configurations are critical for Prometheus and AlertManager.
- Namespace mismatches are a common source of issues; always verify configurations.
- Testing alert workflows ensures reliability.

Resources

- Prometheus Documentation
- AlertManager Configuration
- Trivy Documentation

This documentation provides a detailed yet easy-to-follow approach, ensuring even those new to Prometheus can succeed with the setup.

Conclusion

In this documentation, we walked through the comprehensive setup of **monitoring and alerting in Kubernetes** using Prometheus, AlertManager, and Slack, focusing on Trivy for vulnerability scanning. This workflow ensures real-time visibility into critical vulnerabilities and facilitates rapid incident response through seamless Slack notifications. By following these detailed steps, you can enhance the security and operational efficiency of your Kubernetes cluster, making it more robust and reliable.

A Note from My Experience

While Prometheus and its ecosystem are powerful tools for monitoring and alerting, my experience configuring this workflow exposed some significant pain points:

1. **Documentation Gaps**:
   - I found the **official Prometheus documentation** lacked clarity on certain configurations, especially for integrating custom rules like PrometheusRule and setting up serviceMonitorSelector. Important details—like the need to add these configurations to values.yaml and upgrade the Helm chart—were scattered or missing entirely.
2. **Configuration Pitfalls**:
   - Small oversights, such as namespace mismatches and omitted ruleSelector configurations, caused critical alerts to fail silently. These issues consumed unnecessary hours of troubleshooting and testing.
   - The lack of consolidated, practical examples for Slack integration further compounded the challenges.

Conclusion

In this documentation, we walked through the comprehensive setup of **monitoring and alerting in Kubernetes** using Prometheus, AlertManager, and Slack, focusing on Trivy for vulnerability scanning. This workflow ensures real-time visibility into critical vulnerabilities and facilitates rapid incident response through seamless Slack notifications. By following these detailed steps, you can enhance the security and operational efficiency of your Kubernetes cluster, making it more robust and reliable.

A Note from My Experience

While Prometheus and its ecosystem are powerful tools for monitoring and alerting, my experience configuring this workflow exposed some significant pain points:

1. **Documentation Gaps**:
   - I found the **official Prometheus documentation** lacked clarity on certain configurations, especially for integrating custom rules like PrometheusRule and setting up serviceMonitorSelector. Important details—like the need to add these configurations to values.yaml and upgrade the Helm chart—were scattered or missing entirely.
2. **Configuration Pitfalls**:
   - Small oversights, such as namespace mismatches and omitted ruleSelector configurations, caused critical alerts to fail silently. These issues consumed unnecessary hours of troubleshooting and testing.
   - The lack of consolidated, practical examples for Slack integration further compounded the challenges.

If You Need Help

If you find yourself stuck or facing similar difficulties while implementing Prometheus or Kubernetes monitoring workflows, don't hesitate to reach out! You can connect with me here:

- **LinkedIn**: Sarvadnya Jawle
- **Email**: sarvadnyajawle@gmail.com

I understand how frustrating these setups can be, and I'd be happy to assist or share insights from my journey.

Thank you for taking the time to read through this guide, and I hope it saves you the hours of trial and error that I endured. Happy monitoring!