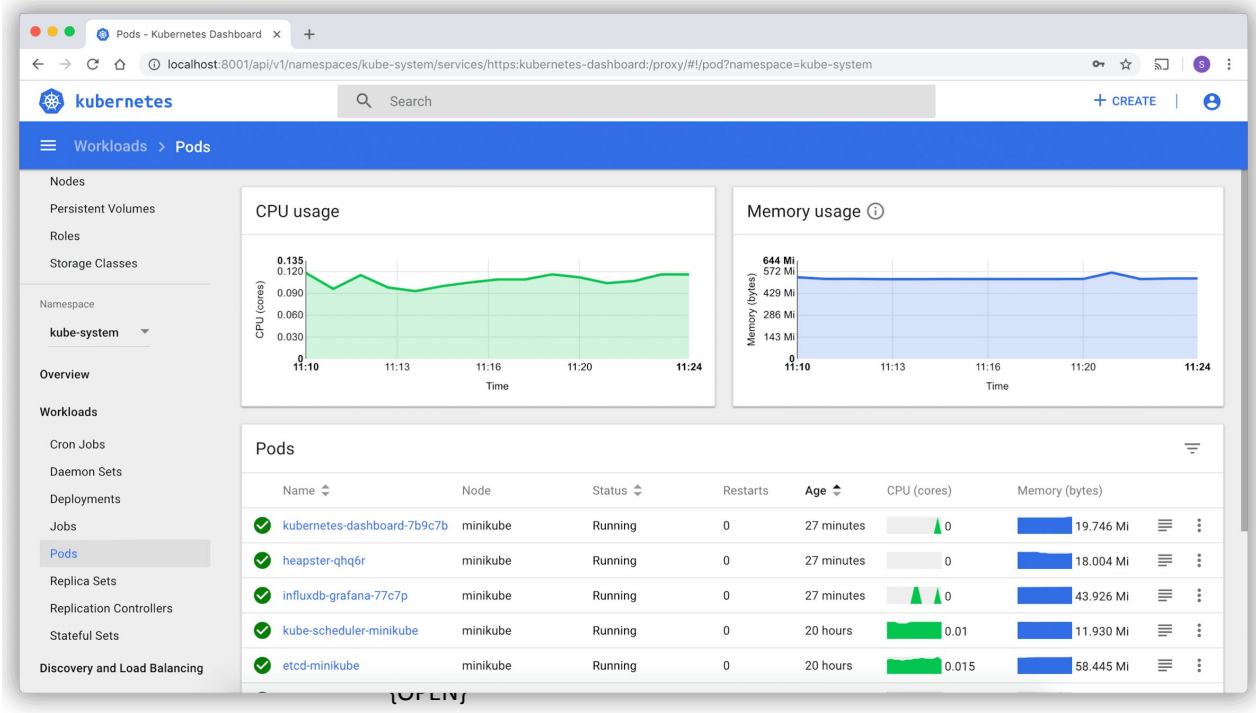
Cybersécurité des infrastructures pour le cloud

Agathe Blaise (Thales), Jacopo Bufalino (CNAM)

Kubernetes monitoring

Detection and Prevention

- In Kubernetes, features like Network Policies and RBAC are primarily used for **prevention-related** capabilities.
- It is equally **important to implement detection measures** that provide visibility into the environment, allowing us to monitor and understand ongoing activities.



Runtime Monitoring

Monitoring: Metrics and alerts – check the health and performance of platform and deployed applications.

- Kube-state-metrics (built-in): metrics about the state of the objects: node status, node capacity (CPU and memory), number of desired/(un)available/updated replicas per deployment, pod status, …
- Metrics Server (built-in): exposes statistics about the resource utilization of Kubernetes objects.
- Prometheus: open-source systems monitoring and alerting toolkit, scrapping metrics via exporters (Kube-state-metrics, node-exporter).
- Grafana dashboard: open-source platform for monitoring and observability of Prometheus metrics.
- Jaeger: open-source distributed tracing platform. Helps monitor and troubleshoot transactions across microservices-based architectures, providing insight into request flows and performance bottlenecks.

{OPEN}

Kube-state-metrics (native Kubernetes)

- **kube-state-metrics** is a simple service that listens to the Kubernetes API server and generates metrics about the state of the objects.
- Focused on the **health of the various objects** inside, such as deployments, nodes and pods.

- Metrics are **exported on the HTTP endpoint** /metrics on the listening port (default 8080) and served as plaintext.
- Designed to be consumed either by **Prometheus** itself or by a **scraper** that is compatible.

Kube-state-metrics (native Kubernetes)

Examples:

```
kube_state_metrics_list_total{resource="*v1.Node",result="success"} 1
kube_state_metrics_list_total{resource="*v1.Node",result="error"} 52
kube_state_metrics_watch_total{resource="*v1beta1.Ingress",result="success
"} 1
```

For HTTP:

```
http_request_duration_seconds_bucket{handler="metrics",method="get",le="2.
5"} 30
http_request_duration_seconds_sum{handler="metrics",method="get"}
0.02111391999999998
http_request_duration_seconds_count{handler="metrics",method="get"} 30
```

Metrics Server (Kubernetes-sigs)

- Metrics Server is a **scalable, efficient source of container resource** metrics for Kubernetes built-in autoscaling pipelines.

- Metrics Server collects resource metrics from Kubelets and exposes them in Kubernetes apiserver through Metrics API for use by **Horizontal Pod Autoscaler** and **Vertical Pod Autoscaler**. Metrics API can also be accessed by kubectl top, making it easier to debug autoscaling pipelines.

{OPEN}

Core Monitoring Pipeline

“Core monitoring pipeline” or “resource metrics pipeline” – installed with every distribution

- **cAdvisor** collects metrics about containers and nodes that on which it is installed
- **Kubelet** exposes these metrics (default is one-minute resolution) through Kubelet APIs
- **Metrics Server** discovers all available nodes and calls Kubelet API to get containers and nodes resources usage.
- Metrics Server **exposes** these metrics through Kubernetes aggregation API.

{OPEN}

Runtime Monitoring

Logging: Aggregation, Searching & Filtering

- Fluent Bit: open-source log processor collecting metrics and logs from different sources (Kubernetes container logs from the file system or Systemd/Journald) and enrich them with Kubernetes metadata.
- Centralization and visualization: ElasticSearch and Kibana.
- Elastic Cloud for Kubernetes (ECK).

{OPEN}

Runtime Monitoring

Auditing: Audits the activities generated by the users, the applications, and the control plane itself.

- Auditing (built-in): Kubernetes API audit
- Many other sources: Kubeaudit, Kube-bench, Kube-hunter, Calico, …

{OPEN}

Prometheus & Grafana

1. **Prometheus** scrapes metrics from Kubernetes objects (Pods, Nodes Services) via exporters (e.g., kube-state-metrics, node-exporter)
2. It **stores time-series data** in its database, enabling real-time and historical analysis.
3. **Alertmanager** handles alerts based on defined thresholds
4. **Grafana** queries Prometheus and visualizes data with custom dashboards.

{OPEN}

Elasticsearch

ELK stack:

- **Elasticsearch**: stores and indexes logs efficiently, 1) functions like a database but is schema-free and JSON-based, 2) provides a RESTful API for searching and querying logs.
- **LogStash**: ingests, processes, and forwards logs. Accepts data from various sources (log file, Kafka, or a S3 bucket). Transforms data before forwarding it to Elasticsearch for long-term storage.
- **Kibana**: creates dashboards with charts, graphs, and time-series data analytics. Allows real-time log exploration for monitoring and troubleshooting.

{OPEN}

Elasticsearch

ELK stack:

- Can be deployed in the Kubernetes cluster with a Helm chart containing all configurations (e.g., Pods, roles and role bindings, secrets)
- /!\ requires a **unified logging layer** to collect and normalize logs from multiple sources → **fluentd**

{OPEN}

Fluentd

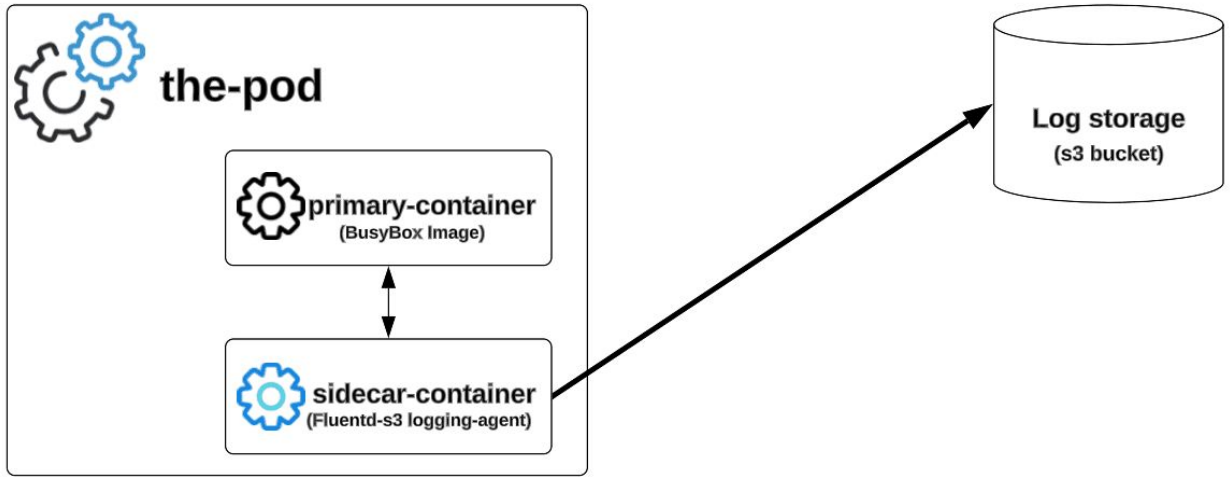
- Open-source **log collector and aggregator** that unifies data collection and forwarding for different sorts of logs (e.g., access logs, web logs, system logs, database logs, infrastructure logs)
- Runs in the cluster as a **DaemonSet** (one Pod running on each Kubernetes node)
  - **Input plugins**: define the **data source** (e.g., using s3 plugin if you use s3 as your input source)
  - **Parser plugins**: transform raw log data into structured formats (e.g., JSON to parse JSON-formatted logs)
  - **Filter plugins**: modify or enrich logs before forwarding (e.g., adding custom fields as Kubernetes labels and timestamps)
  - **Output plugins**: define where logs are sent (e.g., elasticsearch or kafka)

Fluent Bit

- **High-speed log processor and forwarder** designed for cloud-native environments with minimal resource usage.
- Shares the same **plugin-based architecture** than Fluentd with input, filter, and output plugins.
- **Zero dependencies** – runs independently without requiring external libraries.
- **SQL stream processing** – enables real-time filtering, transformation, and aggregation of logs using SQL queries.
- **Custom logging architecture** – allows flexible log processing and forwarding across multiple destinations.

Fluentd and Fluent Bit as a sidecar container

- **Decouples logging logic** from the main application
- **Collects and forwards logs** in real time
- **Ensure log persistence** even if the main container crashes
- **Enables per-pod log processing** with custom filtering & enrichment



Fluentd runs as a sidecar container inside the same pod as the application

{OPEN}

Elastic Cloud for Kubernetes (ECK)

- Automates the deployment and management of the Elastic stack (Elasticsearch, Logstash, Kibana, and Beats) within Kubernetes
- Centralized logging, search, analytics, and security monitoring
- Beats deployed to collect four kinds of data:
  - **PacketBeat**: network traffic
  - **AuditBeat**: Kubernetes audit logs (including system calls)
  - **FileBeat**: container logs
  - **MetricBeat** : memory, CPU, disk usage at both node and pod level

{OPEN}

Auditing

Auditing provides a security-relevant, chronological set of records documenting the sequene of actions in a cluster.

- What happened?
- When did it happen?
- Who initiated it?
- On what did it happen?
- From where was it initiated?
- To where was it going?

```yaml
apiVersion: v1
kind: Event
metadata:
  name: mypod.1692ef7d8a7b5b78
  namespace: default
involvedObject:
  kind: Pod
  name: mypod
  namespace: default
  uid: 1692ef7d8a7b5b78
  apiVersion: v1
type: Warning
reason: Failed
message: 'Failed to pull image "nginx:invalidtag": rpc error: code = NotFound desc'
source:
  component: kubelet
  host: node-1
firstTimestamp: "2024-03-18T12:00:00Z"
lastTimestamp: "2024-03-18T12:00:10Z"
count: 3
eventTime: null
reportingComponent: ""
reportingInstance: ""
```


{OPEN}

Auditing

Audit policy defines rules about what events should be recorded and what data they should include.

Audit levels	Description
None	Don't log events that match this rule.
Metadata	Log request metadata (user, timestamp, resource, ver, etc) but not request or response body.
Request	Log event metadata and request body but not response body.
RequestResponse	Log event metadata, request and response bodies.

{OPEN}

Auditing

Example audit policy file:

```yaml
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
# Don't generate audit events for all requests in RequestReceived stage.
omitStages:
  - "RequestReceived"
rules:
  # Log pod changes at RequestResponse level
  - level: RequestResponse
    resources:
    - group: ""
      # Resource "pods" doesn't match requests to any subresource of pods,
      # which is consistent with the RBAC policy.
      resources: ["pods"]
  # Log "pods/log", "pods/status" at Metadata level
  - level: Metadata
    resources:
    - group: ""
      resources: ["pods/log", "pods/status"]

  # Don't log requests to a configmap called "controller-leader"
  - level: None
    resources:
    - group: ""
      resources: ["configmaps"]
      resourceNames: ["controller-leader"]
  # Log the request body of configmap changes in kube-system.
  - level: Request
    resources:
    - group: "" # core API group
      resources: ["configmaps"]
    # This rule only applies to resources in the "kube-system" namespace.
    # The empty string "" can be used to select non-namespaced resources.
    namespaces: ["kube-system"]

  # Log configmap and secret changes in all other namespaces at the Metadata level.
  - level: Metadata
    resources:
    - group: "" # core API group
      resources: ["secrets", "configmaps"]
```

Dynamic-time security

{OPEN}

Signature-based vs. ML-based detection

Signature-based Detection

Advantages:

- **Relies on known patterns** (hashes, byte sequences, rules)
- Fast and **low false positives** for known threats

Disadvantages:

- Ineffective against **new, unknown (zero-day)** threats
- Requires constant **manual updates**

Example: Detecting a virus by matching it to a database of known malware signatures

{OPEN}

Signature-based vs. ML-based detection

ML-based Detection

Advantages:

- Uses **algorithms trained on behaviour, patterns, and anomalies**
- Can detect **zero-day** malware
- Learns and adapts over time

Disadvantages:

- Requires **training data** and can have **false positives**
- More **resource-intensive**

Example: Using an ML model to classify a process as malicious based on features like API calls, file access patterns, and entropy

Signature-based vs. ML-based detection

Feature	Signature-based	ML-based
Known threats	Effective	Effective
Unknown threats (or zero-day attacks)	Limited	Adaptive
Performance	Fast	Slower (usually)
Maintenance	Frequent updates	Self-learning
False positives	Low (unknown threats)	Can be higher

{OPEN}

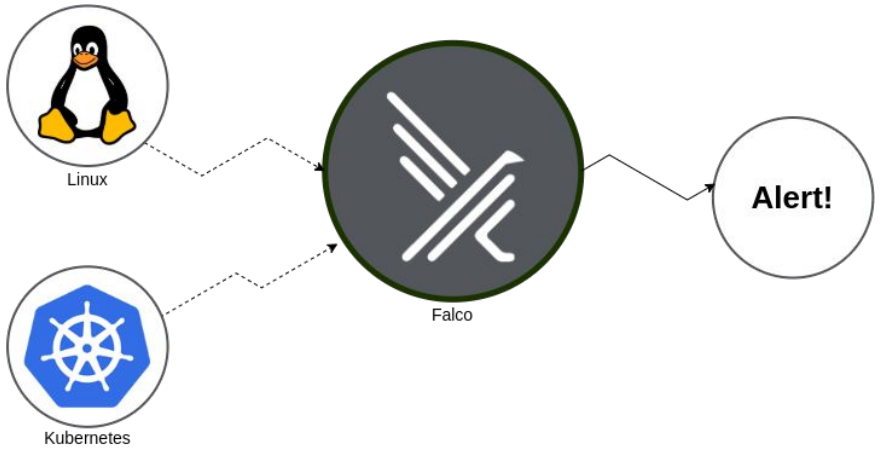
Kubernetes security solutions at runtime

Feature / Solution	Falco	Calico Cloud	Cilium	Sysdig Secure
System calls monitoring	✓ Yes	✓ Yes	✓ Yes	✓ Yes
Network traffic	✗ No	✓ Yes	✓ Yes	✓ Yes
Technique used	Signature-based detection	Machine Learning + policies	Signature-based detection	Machine Learning + rules
Remediation (active blocking)	✗ No (detection-only)	✗ (policy enforcement only)	✗ (policy enforcement only)	✓ Yes (real-time response and blocking)
Enforcement type	Detection via syscalls	Policy-based network security	eBPF-based network & system security	Detection + response
Runtime threat detection	✓ Yes (via kernel syscalls)	✓ Yes	✓ Yes (via eBPF)	✓ Yes (deep visibility into processes, files, and network)

Falco (from Sysdig)

Focus: Runtime threat detection for containers and K8s using **syscalls**

- Uses **Linux system calls** to monitor container behavior
- Uses **rule-based detection** (e.g., detecting unexpected shell execution inside a container)
- **Example:** can alert when a container runs an unexpected binary (e.g., bash inside an Nginx pod).



Limitations

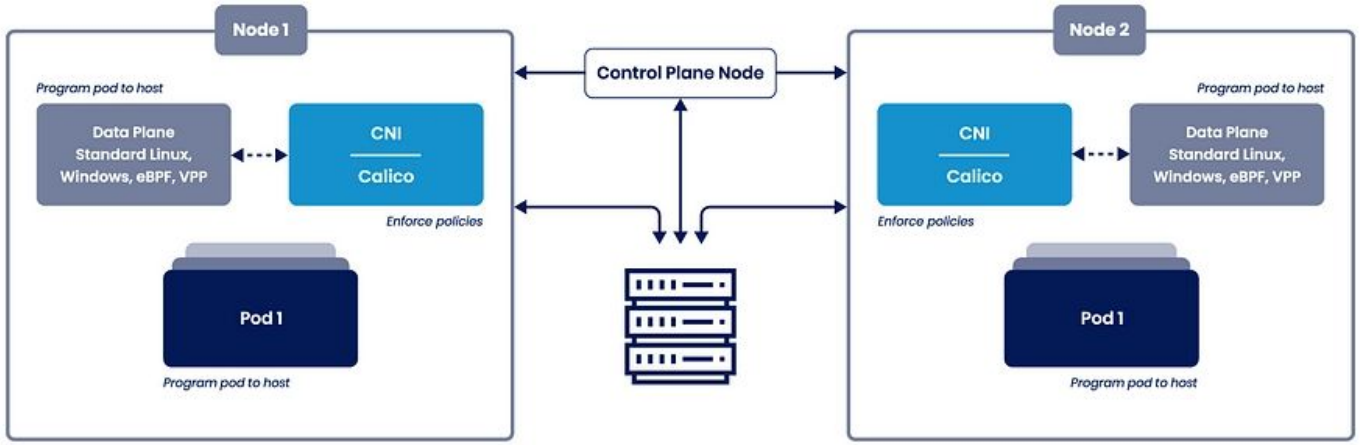
- **No active blocking** - it only **detects** threats, does not prevent them
- No built-in **network security** capabilities

{OPEN}

Calico Cloud (from Tigera)

Focus: Network security for Kubernetes with machine-learning (ML) driven anomaly detection

- Uses **network policies** to enforce micro segmentation
- Analyzes **network traffic** to detect anomalies
- Uses **ML to detect suspicious activity** based on traffic patterns



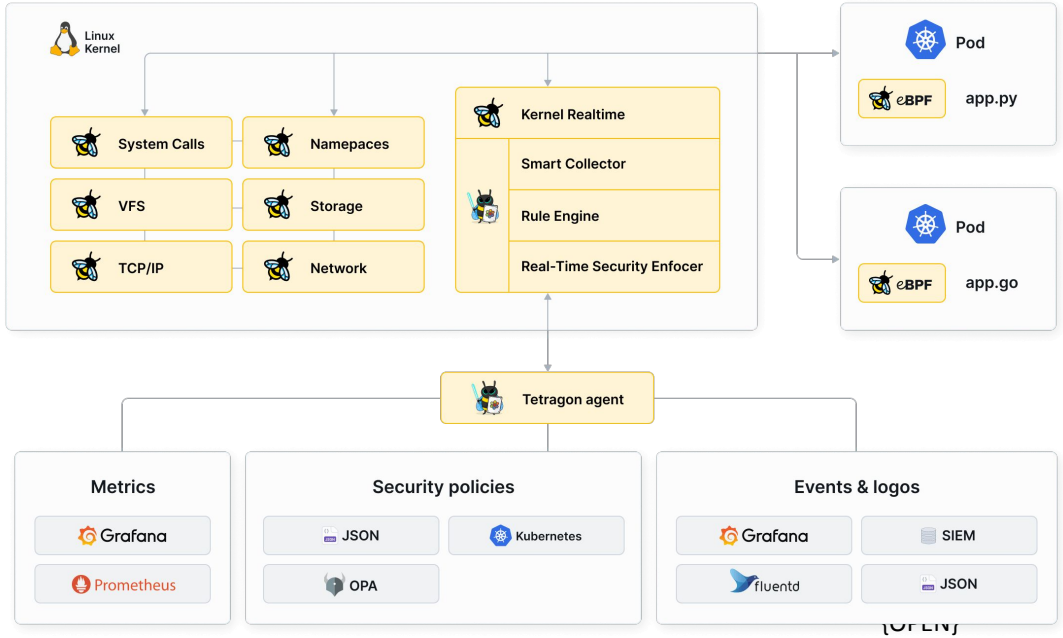
Limitations

- **No deep syscall monitoring** – does not inspect process-level activity inside containers
- **No real-time remediation** – enforces policies but do not dynamically nlock threats at the host level

Cilium

Focus: eBPF-powered network & system security for Kubernetes

- Uses **eBPF (extended Berkeley Packet Filter)** for deep observability and security enforcement at the kernel level.
- Can enforce identity-based network policies (rather than just IP-based)
- Detects network anomalies and restricts unauthorized traffic dynamically.



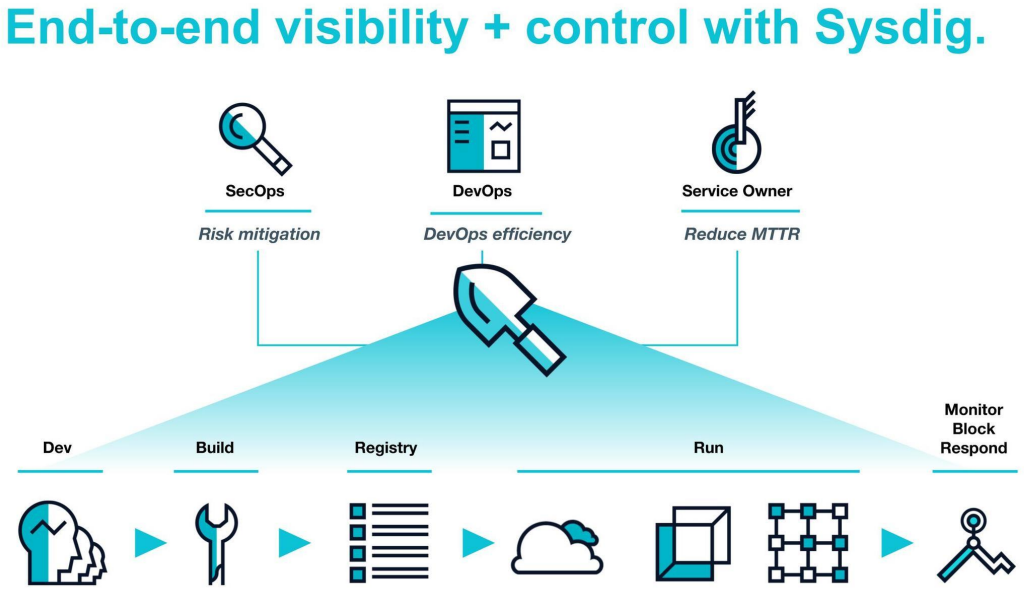
Limitations

- **Signature-based detection** – may miss advanced unknown threats
- **No active runtime remediation** – can enforce policies but does not actively mitigate threats.

Sysdig Secure (Falco + active remediation)

Focus: Full-stack container security, including Falco-based detection with **real-time response and blocking.**

- Uses **Falco for syscall monitoring + ML-driven analytics** for anomaly detection
- Includes **real-time remediation:** automatically killing suspicious containers or isolating them
- Supports **network security policies, image scanning,** and **runtime enforcement**



{OPEN}

Kubernetes security solutions at runtime

Feature / Solution	Falco	Calico Cloud	Cilium	Sysdig Secure
System calls monitoring	✓ Yes	✓ Yes	✓ Yes	✓ Yes
Network traffic	✗ No	✓ Yes	✓ Yes	✓ Yes
Technique used	Signature-based detection	Machine Learning + policies	Signature-based detection	Machine Learning + rules
Remediation (active blocking)	✗ No (detection-only)	✗ (policy enforcement only)	✗ (policy enforcement only)	✓ Yes (real-time response and blocking)
Enforcement type	Detection via syscalls	Policy-based network security	eBPF-based network & system security	Detection + response
Runtime threat detection	✓ Yes (via kernel syscalls)	✓ Yes	✓ Yes (via eBPF)	✓ Yes (deep visibility into processes, files, and network)

{OPEN}

Machine learning solutions

Example: Learning state machine models from network data

- Gather NetFlow data from cluster using ElasticSearch
- NetFlow data contains only normal network behaviour
- NetFlow data is encoded to sequences
- State machine models network behaviour of cluster
- Anomalies (potential attacks) are detected as deviations from the normal behavior

{OPEN}

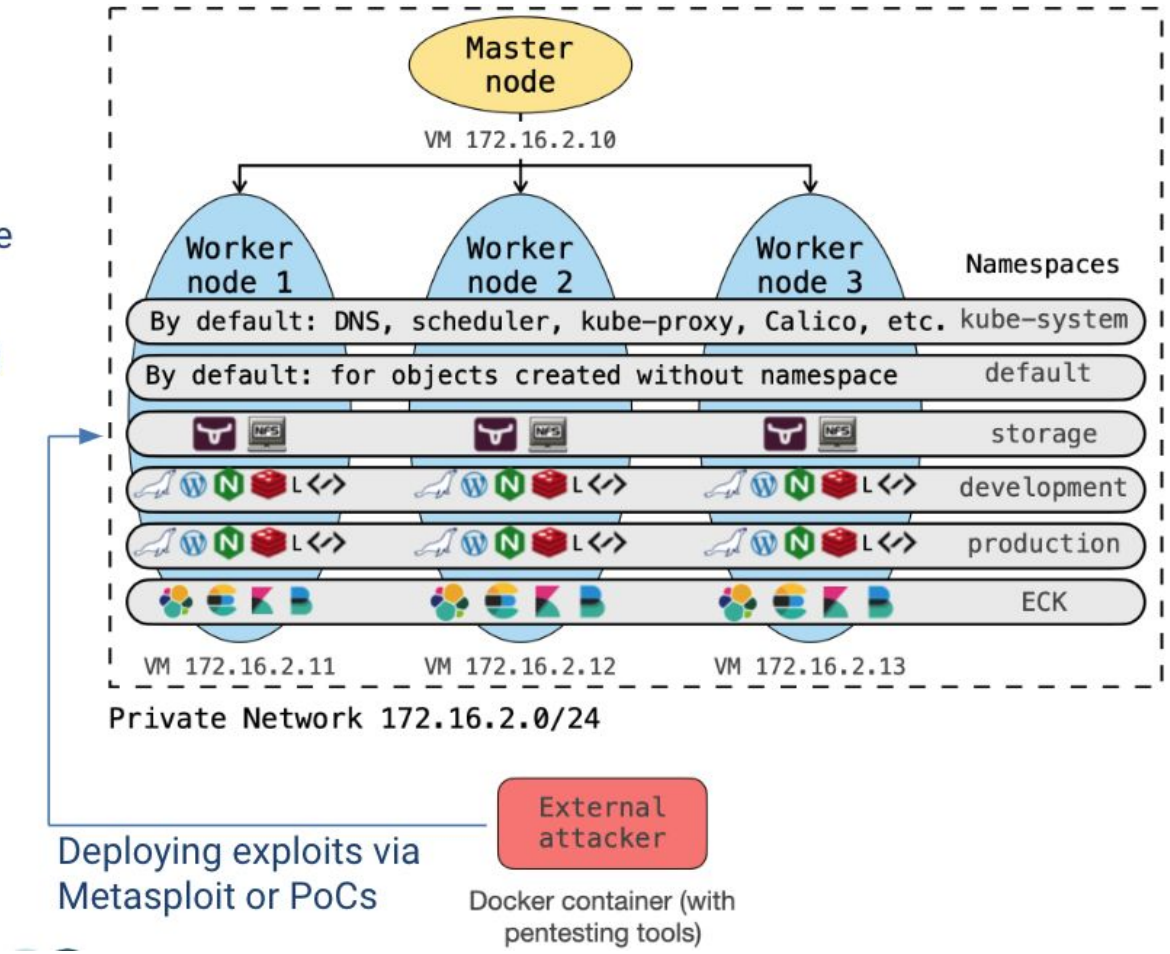
Kubernetes testbed and attack scenarios

{OPEN}

Kubernetes testbed components

K8s cluster composed of one master node and several workers nodes - with several namespaces:

- **Storage**: handling distributed and persistent storage
- **Development** and **production:** Wordpress and MariaDB, nginx server, guestbook (Redis leader and followers, front end)
- **ECK** (Elastic Cloud on Kubernetes):
  - Elasticsearch, Kibana, Logstash
  - Filebeat: container logs
  - Auditbeat: K8s audit logs, syscalls
  - Packetbeat: traffic
  - Metricbeat: CPU and memory resources usage
  - Prometheus/Grafana



{OPEN}

Kubernetes testbed components

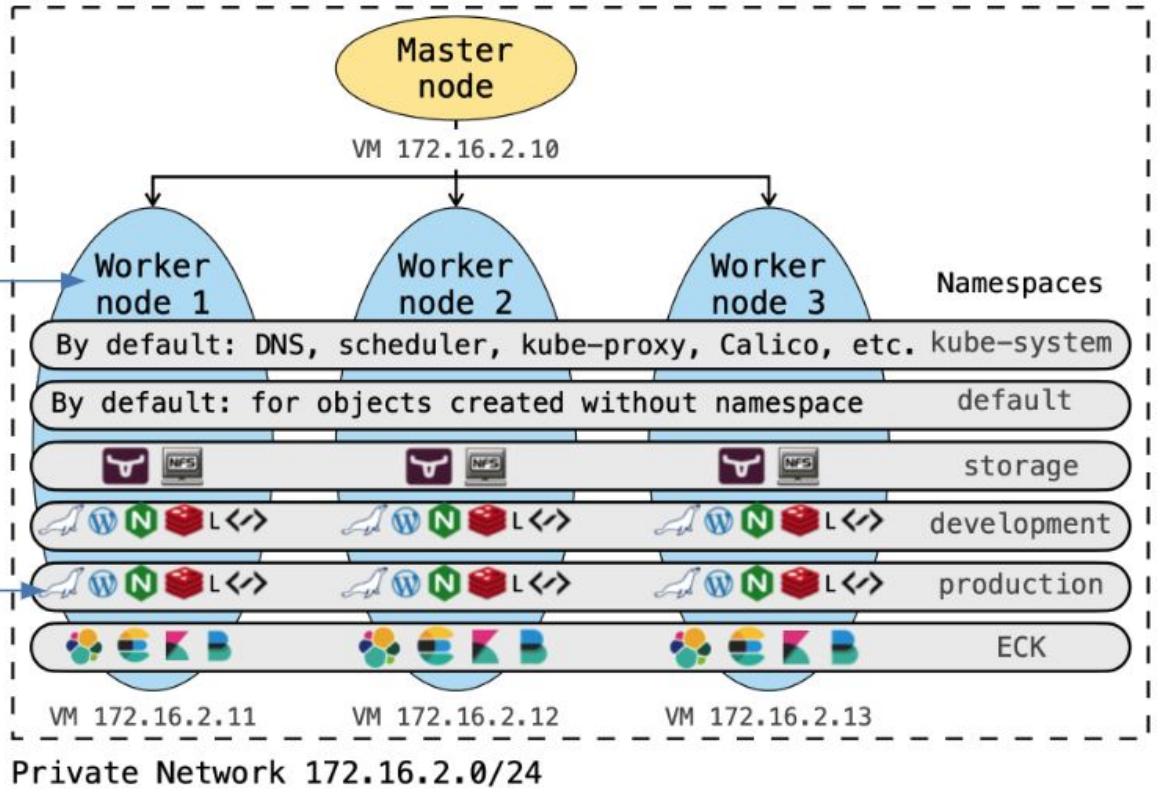
Two ***main categories of vulnerabilities***:

- **Platform-related**: vulnerable cluster configuration, kubelet's unauthenticated, fake worker node, Role-Based Access Control issues, vulnerable network endpoints, vulnerable service tokens

→ *Attacker on the network - can move within the cluster and across containers*

- **Application-related**: misconfigured Docker, malicious Docker image, vulnerable software application within pod

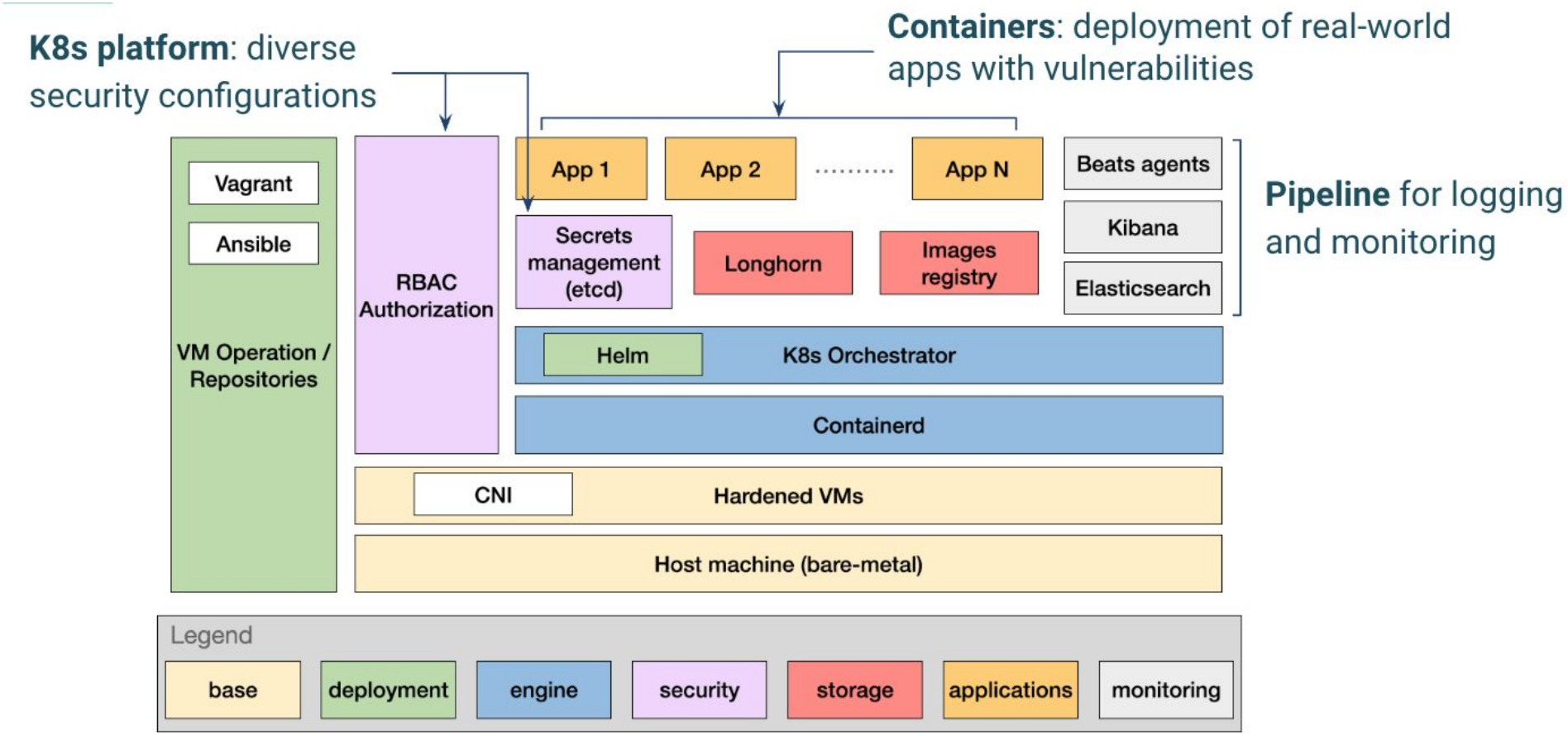
→ *Compromised application - lead to foothold in container*



... leading to compromise of the whole cluster

{OPEN}

Kubernetes testbed



Kubernetes testbed and multi-step attack scenarios

Objective: to provide an environment that can be replicated and exploited to deploy example **attack scenarios**

Presentation of the **Kubernetes testbed** with several components

- Kubernetes platform with diverse security configurations including insecure ones
- Deployment of real-world applications, including some with vulnerabiliites
- Pipeline for logging and monitoring (network traffic, system calls, logs from containers, Kubernetes audit logs, CPU/memory metrics)

Kubernetes testbed and multi-step attack scenarios

Description of the **4 multi-step attack scenarios** and mapping to tactics from MITRE ATT&CK framework

- Combination of **different tactics:** initial access, execution, discovery, lateral movement, privilege escalation
- Diverse **end goals** from an attacker: DoS attack, malicious workload, foothold outside the K8s cluster, access to sensitive data

Description of **data collected during an experiment**

- Labeled datasets including background traffic and attack scenarios launched simultaneously
- Preliminary results to detect attack scenarios: suspicious syscalls, abnormal traffic patterns, increase of CPU/RAM usage, …

Kubernetes testbed and multi-step attack scenarios

MITRE ATT&CK matrix for Kubernetes – design of 4 attack scenarios

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Mouvement	Collection	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8s secrets	Access the K8s API server	Access cloud resources	Image from a private registry	Data destruction
Compromised images in registry	Bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8s events	Mount service principal	Access Kubelet API	Container service account		Resource Hijacking
Kubeconfig file	New container	Kubernetes Cronjob	hostPath mount	Pod/container name similarity	Access container service account	Network Mapping	Cluster internal networking		Denial of Service
Application vulnerability	Application exploit (RCE)	Malicious admission controller	Access cloud resources	Connect from proxy server	Application credentials in config files	Access Kubernetes Dashboard	Applications credentials in config files		
Exposed Dashboard	SSH server running inside container				Access managed identity credential	Instance Metadata API	Writable volume mounts on the host		
Exposed sensitive interfaces	Sidecar injection				Malicious admission controller		Access Kubernetes Dashboard		
							Access tiller endpoint		
							CoreDNS poisoning		
							ARP poisoning and IP spoofing		

Titre

Scenario 1 Scenario 2 Scenario 3 Scenario 4

Scenario-based view

Scenario 1: establish persistence	Scenario 2: denial of service	Scenario 3: malicious code execution	Scenario 4: read sensitive data
Execution: application exploit (RCE) on guestbook frontend	**Initial access**: OpenSSH server version < 7.3	**Initial access**: CVE on the Wordpress WebUI	**Discovery**: kubelet's unauthenticated
Discovery: fingerprint the system and discover pods	**Execution**: send simultaneous requests with very long passwords	**Execution**: RCE attack by an unauthenticated remote attacker	**Execution**: scan to get service account tokens
Lateral movement: move to a pod with root privileges	**Impact**: CPU overloading (remote DoS)	**Discovery**: fingerprint the system and discover pods	**Privilege escalation**: break out of the container and get root on the host
Persistence: daemon set with writable hostPath mounts, Kubernetes cron job		**Lateral movement**: access a privileged pod	**Collection**: read from file system on Kubernetes host
		Privilege escalation: break out of the container and get root on the host	
		Execution: deploy a bitcoin miner on the host with Docker	

Sidecar containers for monitoring & security

- A **sidecar** container is a **secondary container** that runs alongside the main application container in the **same Pod**, adding extra functionality **without modifying** the main application.
- Firewall sidecar container for network traffic monitoring
  - Monitors, filters, and controls network traffic **before it reaches the main container**.

/!\ Pods share the same network namespace, so all containers within a pod use the same IP and network interface.

→ **Netfilter Queue** can be used to redirect network traffic to the firewall sidecar before it reaches the application.

1. Network packets are stored in a queue
2. The firewall sidecar processes & inspects the traffic
3. Only approved packets are forwarded to the main application container.

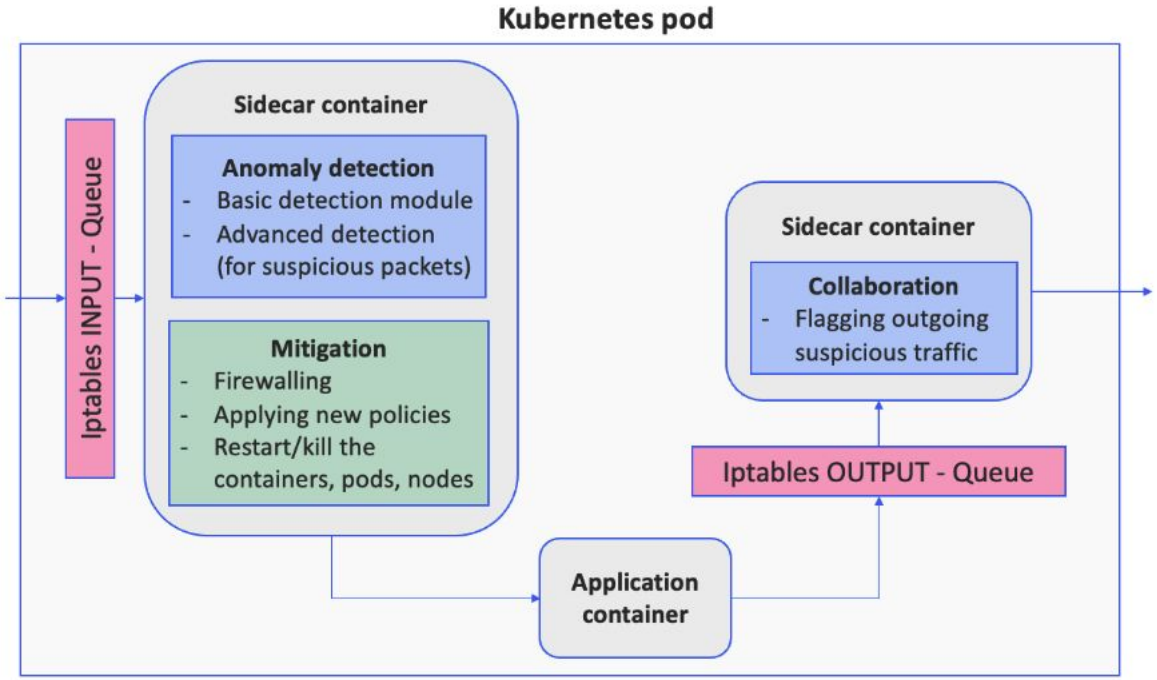
Sidecar containers for monitoring & security

Step 0: Automatic sidecar injection

- Modify Kubernetes Pods to include sidecar containers dynamically with a Kubernetes Job

Step 1: Monitoring & Detection

- Collection of time series data on network traffic volume.
- Identifies sudden spikes, drops, and unusual patterns in network activity
- Uses statistical models & ML-based approaches for real-time detection



Modified Z-score :

More robust than standard Z-score because using the median.

$$M_i = \frac{0.6745(x_i - \tilde{x})}{MAD}$$

Forwarding – intermediate queues

Collaborative Intrusion Detection

Mitigation

Sidecar containers for monitoring & security

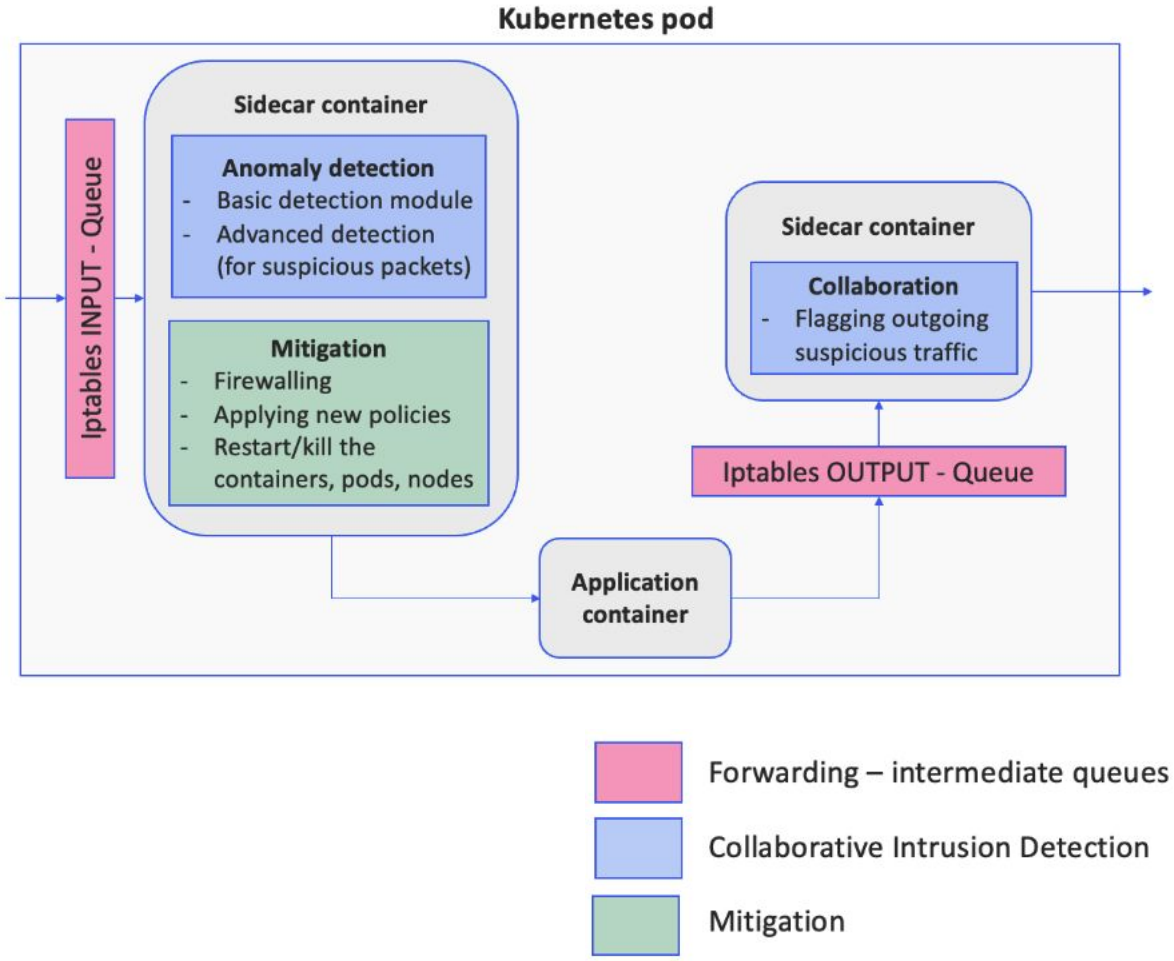
Step 2: Characterisation

/!\ Triggered upon anomaly detection from Step 1

- Collection of key metrics:

  1) At network-level: unique source IP addresses, unique source/destination port numbers, number of active connections
  2) System resource: CPU usage spike, memory consumption

- Anomaly scoring with modified Z-score → classifying the type of anomaly
  E.g., denial of service, network scan, port scan, reverse shell, …

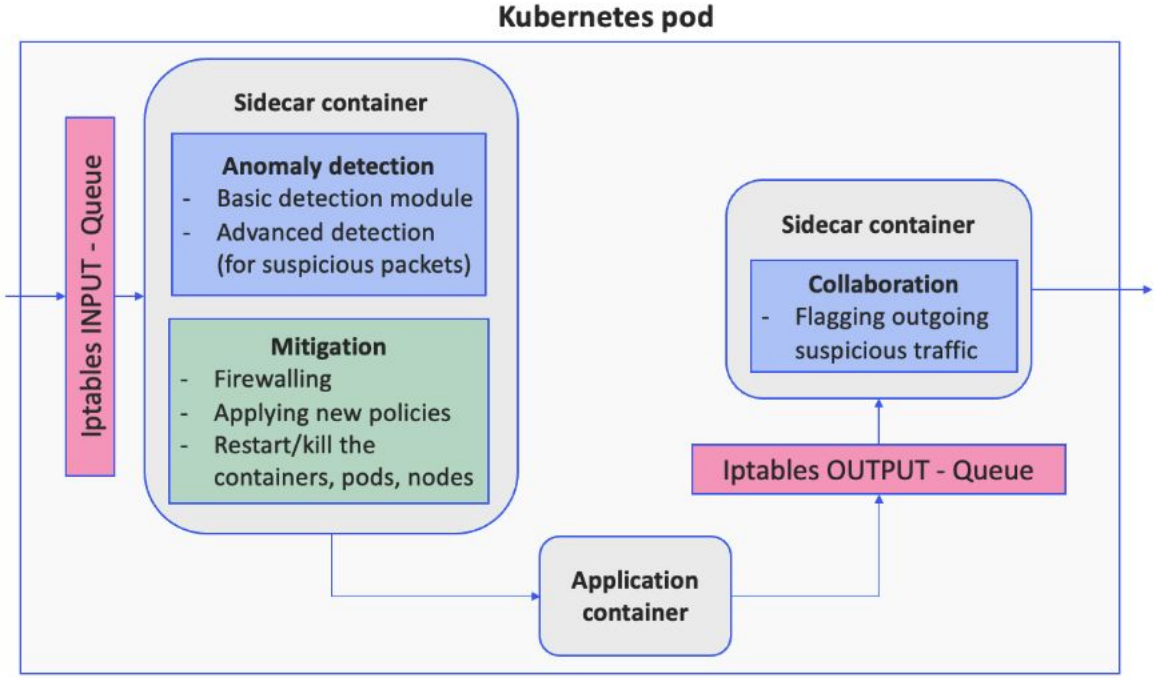


Sidecar containers for monitoring & security

Step 3: Collaboration

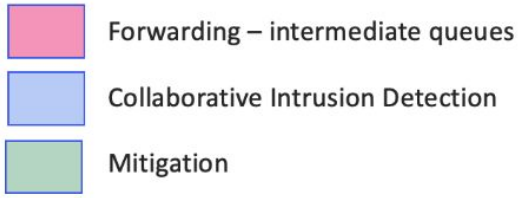
/!\ Triggered when Step 2 does not confirm an anomaly and needs further analysis through collaboration

- Sending “ping” alerts between pods
- Enables cross-pod collaboration to verify abnormal traffic patterns
- Tags network traffic to trace suspicious activity and warns other Pods about a possible ongoing attack



Marking :

Using the 32-bit *timestamp* field inside the Timestamp option of the IP protocol to communicate the pod ID as well as various metrics detected as abnormal at Step 2.



{OPEN}

Sidecar containers for monitoring & security

Step 4: Mitigation

- Retrieving anomaly characteristics from previous steps
- **Immediate countermeasures:**
- /!\ Stop the attack:
  - Delete the compromised pod
  - Apply *iptables* rules to block malicious traffic
  - Isolate the pod using a queue
- **Long-term protection measures:**
- Strengthen security to prevent future attacks
  - Define stricter network policies and *iptables* rules
  - Adding a seccomp profile to restrict system calls
  - Enforce new Pod Security Policies (PSP)

{OPEN}

Rules for classification

Classification of different attacks according to different parameters.

↗↗: *Very high modified Z-score value.*

↗: *High modified Z-score value.*

~: *Stable modified Z-score value.*

✓: *Useful information learned.*

✗: *No useful information learned.*

		DoS	Reverse Shell	(Dirb) scan	DDoS	IP Spoofing	Port Scan
Modified Z-score	Traffic volume (# packets)	↗↗	↗	↗	↗↗	↗	↗
	# unique source IP addresses	~	~	~	↗↗	~	~
	# unique destination port numbers	~	~	~	~	~	↗↗
	# of connections	↗↗	~	~	↗↗	~	~
	% CPU usage	↗↗	~	↗	↗↗	~	~
Collaboration	Ping alerts		✓	✗		✗	
	Marking of all packets					✓	

Data collection for various scenarios

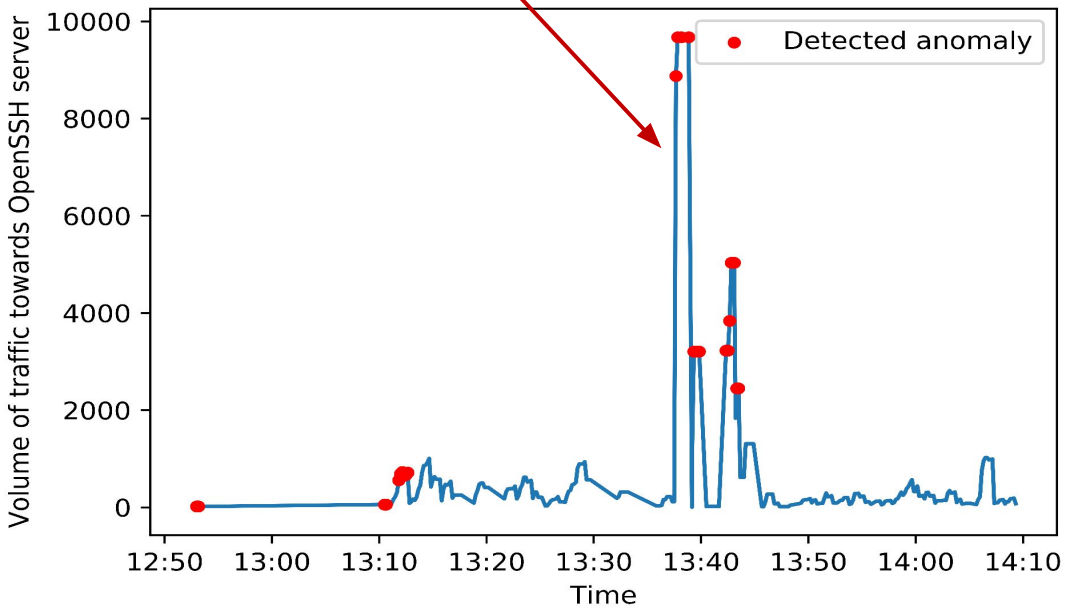
Data collection campaign with benign traffic generated and 3 multi-step attack scenarios have been launched:

- **Scenario 1**: DoS attack which consists in overloading a given service (container)
- **Scenario 2**: DIRB scan which consists in identifying the existing web objects of a service (webpages of a frontend)
- **Scenario 3**: reverse shell which consists in opening a terminal on a service (container) to remotely control it

This data collection campaign helped us getting a dataset of network traffic that includes malicious instances to evaluate our solution.

Detection

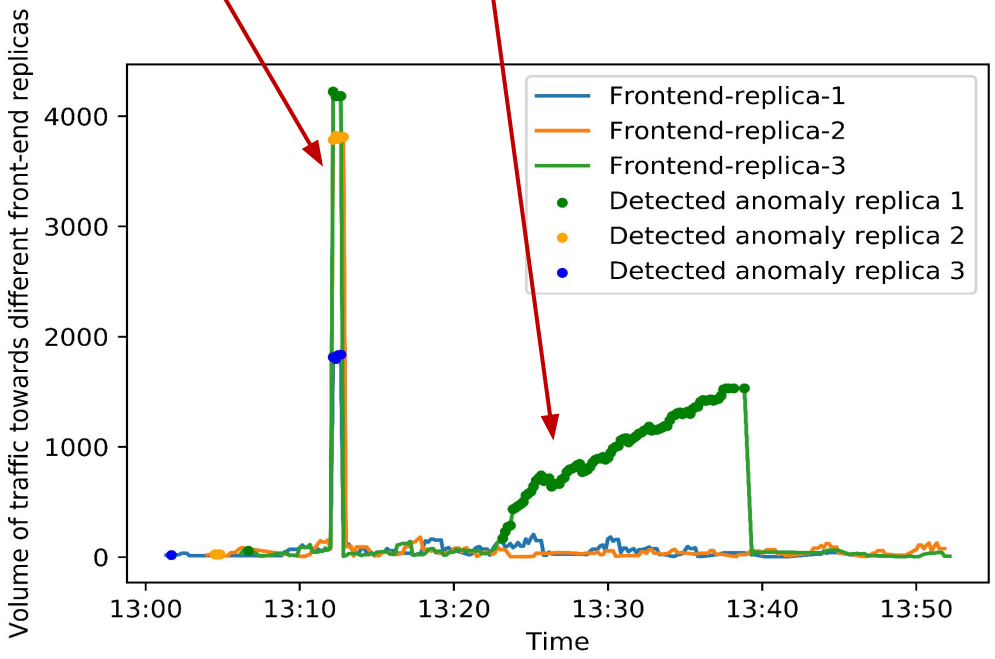
Scenario 1 :
DoS attack on a given pod



Volume of traffic over time during a DoS attack

Scenario 2 :
Dirb scan spread on the 3 replicas

Scenario 3 :
Reverse Shell on a single replica



Volume of traffic over time during a DIRB scan and a reverse shell

{OPEN}

Followed by mitigation

	DoS	DIRB	Reverse Shell
Detection	High increase in network traffic	High increase in network traffic distributed on all replicas of the frontend	Progressive increase of traffic volume only on a given replica.
Mitigation	Setting a iptables rule to drop packets.	- Traffic burst: Deploying new replicas to temporarily handle the load. - Scan DIRB scan : Setting a iptables rule to drop packets	- Short term : Suppressing the pod and creating a new one. - Long term : Adding Seccomp rule

{OPEN}