le cnam

Software Supply Chain Security for the Cloud

Final lecture

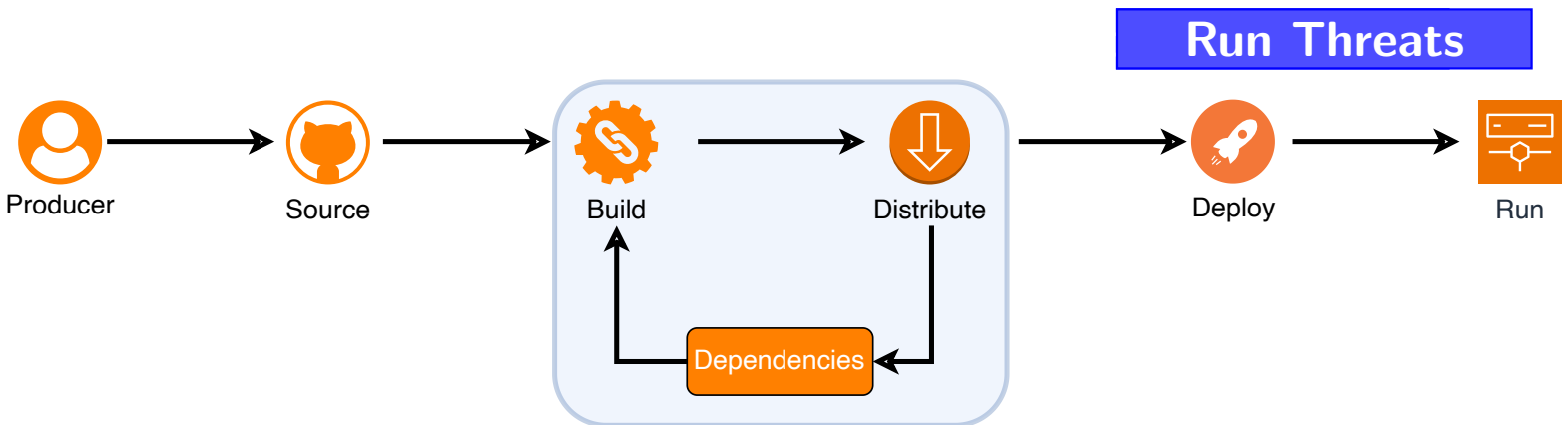
Jacopo Bufalino (CNAM)

Introduction

Where we left

- New code has been developed
- Merged to a main branch
- ...CI pipelines ...
- Ensure dependencies are correct and without known vulnerabilities
- Produced an artifact that is ready to be distributed

Post-build security



Post-build step of the SSC

What will our cloud infrastructure include:

- ➜ Our containers
  - Source code
  - Package dependencies
  - Operating system dependencies
- ➜ Our infrastructure configuration
  - IaC (orchestrating Kubernetes + cloud resources)
- ➜ Third-party containers
  - Source code
  - Package dependencies
  - Operating system dependencies
- ➜ Third party configuration
  - IaC (orchestrating Kubernetes + cloud resources)

IaC misconfigurations

IaC can bring misconfigurations too

- ➜ Recall last year lecture on cloud security
- ➜ Misconfigurations can be of different types:
  - Network
  - Host
  - Permissions
  - Secrets
  - Resources

IaC misconfigurations (example)

```
apiVersion: v1
kind: Pod
metadata:
  name: unsafepod
spec:
  hostNetwork: true
  hostPID: true
  containers:
  - name: container1
    image: alpine
    securityContext:
      privileged: true
    volumeMounts:
    - name: root
      mountPath: /host
```

Issue of IaC misconfigurations

- ➜ Often IaC components are imported
- ➜ e.g., Kubernetes manifests, Helm Charts
- ➜ Need to find misconfigurations in imported components too!

How to find misconfigurations in IaC

Very similar to code and CI static analysis.

- ➜ Scan IaC files (YAML)
- ➜ Search for patterns generated by OSS community
- ➜ Trivy, Checkov, Terrascan
- ➜ Effective because of declarative configuration!

Many tools can check for IaC, containers and CI files at the same time.

Cloud network misconfigurations

Cloud network misconfigurations

- Among the most frequent cloud security failures
- Particularly critical in containerized environments
  - There are many microservices communicating over the network
- Some of the causes:
  - Privileged containers escape the Kubernetes network (hostNetwork)
  - Lack of network policies
  - Mismatch between declared and actual behavior

Mismatch between declared and actual behavior

This is a common problem in the container and orchestrator virtualization layers.

- ➜ Port information is only documentation
- ➜ No guarantee that the container will listen to a port
- ➜ This happens both in docker and in Kubernetes

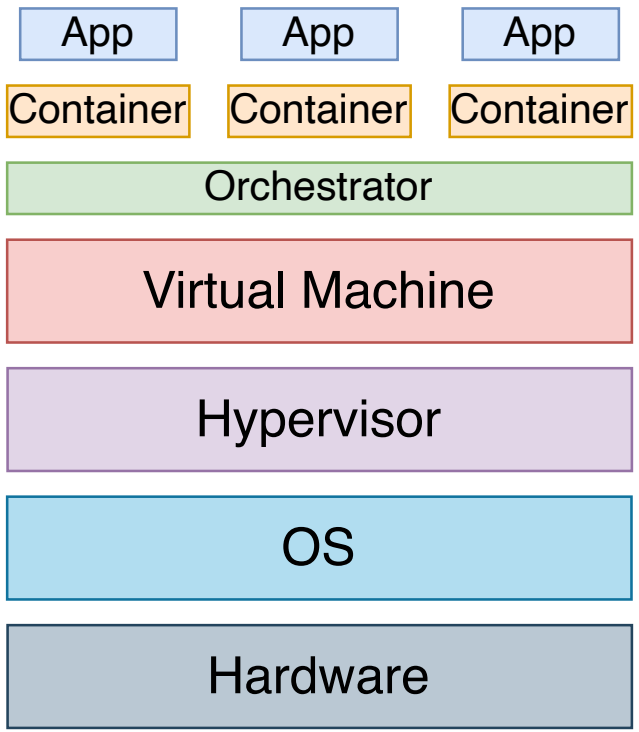


Figure: Virtualization Stack

Docker example

```
FROM alpine:3.18
RUN apk add --no-cache nginx
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

➜ In this case `EXPOSE 80` is pure metadata and does not open or bind the port

Kubernetes example

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

- ➜ `containerPort`: documentation hint
- ➜ Similar issues arise for Service and NetworkPolicy resources

Declarative vs Runtime

- Declarative: desired state (YAML)
- Runtime: actual sockets
- No built-in reconciliation for ports and protocols

This is unique

- CPU, memory → enforced
- Replicas → enforced
- Network policies → enforced
- Port bindings → NOT verified

Security Consequences

- ➜ Policies built on incorrect assumptions
- ➜ Impossible to do static analysis on the YAML

Attack Surface

- ➜ Undeclared / Unopen ports
- ➜ Unexpected services reachable
- ➜ Attackers can leverage these when scanning the network

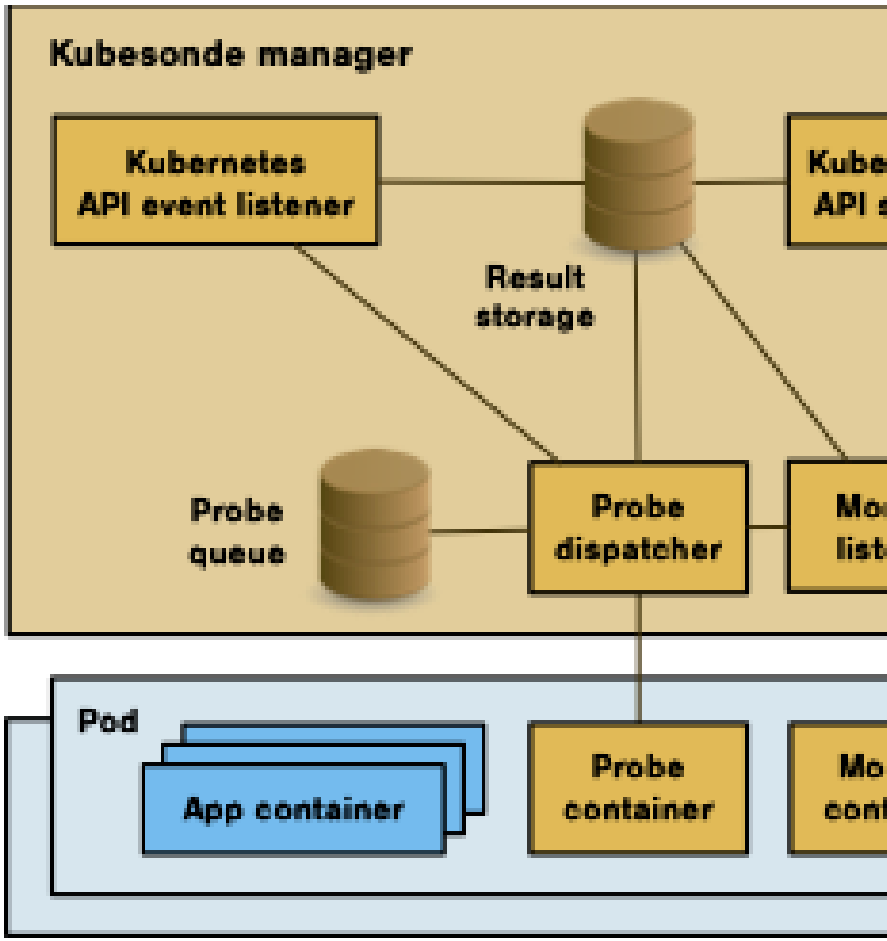
How about third-party components?

- ➜ Ports specified in Helm charts may be misconfigured
- ➜ How can we create network policies for applications we do not know?
- ➜ How can we know what is happening in the cluster?

How we solve these issues in our Research Lab

Kubesonde

- ➜ Act like an adversary
- ➜ Run a distributed nmap
- ➜ Collect open ports from each pod



HelmET

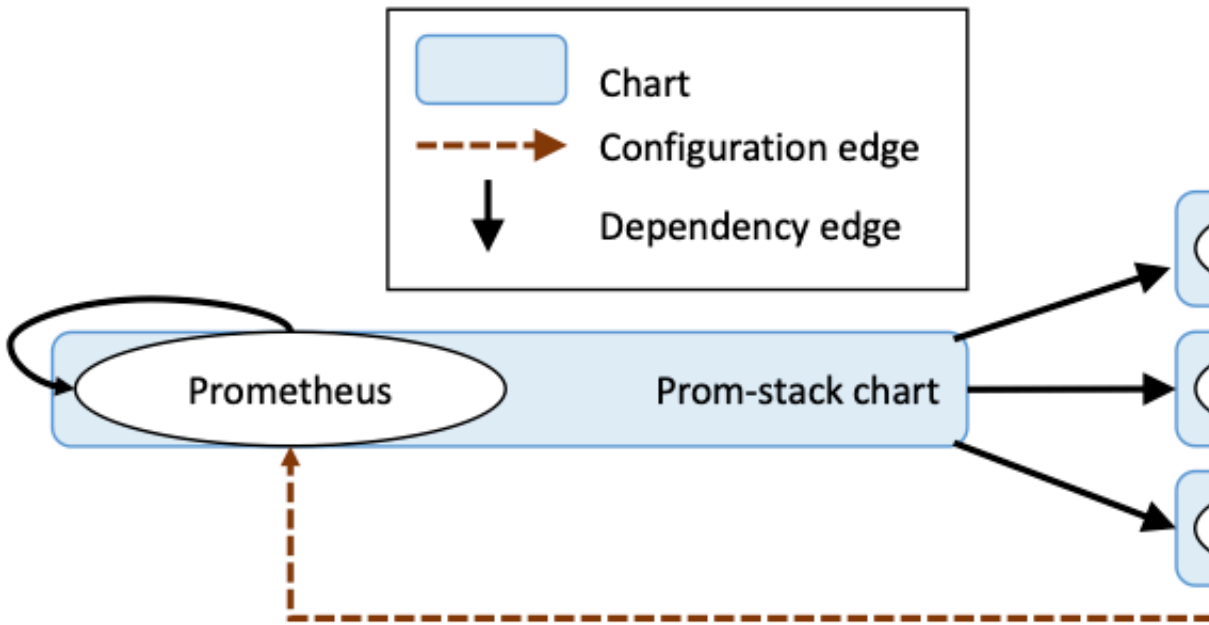
- ➜ Kubernetes manifests and Helm charts form dependencies
- ➜ Treat dependencies like software
- ➜ Use dependencies to build Network policies

```
dependencies:
  - name: kube-state-metrics
    version: "5.29.*"
- name: prometheus-node-exporter
    version: "4.43.*"
- name: grafana
    version: "8.9.*"
```

Prom-stack Chart.yaml

```
...
- name: Prometheus
      type: prometheus
      uid: prometheus
      url: http://metrics-
prometheus.default:9090/
...
```

Grafana configmap



Intent based networking

The problem of network configuration

- ➜ Policy complexity grows with microservices
- ➜ Manual network policy generation is error-prone
- ➜ Secure third-party applications is not-trivial
- ➜ Need for scalable and adaptive security models (e.g., when policies change with regulations)

Intent-Based Networking

- ➜ Specify high-level goals instead of low-level rules
- ➜ System translates intent into enforceable policies
- ➜ Continuous validation against runtime state

Intent Example

- Intent: "frontend can talk to backend"
- IBN engine: Generate egress policy from frontend to backend on port 80
- Output: NetworkPolicy, firewall rules

IBN Architecture

- Intent input (Natural language or other domain-specific languages)
- Compilation engine
- Policy deployment
- Runtime verification loop

Intent Model

- Entities: services, pods, namespaces
- Relations: allowed / denied communication
- Constraints: ports, protocols, directions (ingress/egress)

IBN Assumption

- ➜ System model is accurate
- ➜ Declarative state reflects reality

Runtime Verification Loop

- ➜ Generate policy from intent
- ➜ Observe actual traffic
- ➜ Compare intent vs runtime
- ➜ Correct discrepancies

Role of AI

- ➜ Translate natural language into intent
- ➜ Generate policies
- ➜ Validate results

Current limitations

- ➜ How to verify if proposed intent and policies are correct?
- ➜ Hallucinated outputs
- ➜ Incorrect topology assumptions

Advanced software supply chain topics

Limitations of Software Composition Analysis

SBOM interoperability

Hardened Container Images

Hardened Container Images

We have seen that most of the attack surface of container images lays in the OS dependencies:

- ➜ Default images include unnecessary components
- ➜ Large attack surface (shells, package managers)
- ➜ Frequent CVEs in unused libraries

How to solve this problem?

- ➜ Containers with minimized attack surface
- ➜ Designed to reduce CVE counts
- ➜ Patched as soon as a CVE is discovered

What Hardened Images Solve

- ➜ Reduce attack surface
- ➜ Limit post-exploitation capabilities
  - ▶ Attackers have less tools to work with
- ➜ Improve auditability and compliance
  - ▶ Is is hard for companies to keep track of all CVEs
  - ▶ Most of the CVEs are not related to the application software

Minimalism

- ➜ Only include what is strictly required
- ➜ Remove shells and debugging tools
- ➜ Eliminate unused libraries

Building hardened images

- If you want to run a binary, only add that.
  - use the `FROM scratch` formula
  - only add the binary file after that
- Use tools like `apko` to:
  - Compile only the libraries and binaries needed

Build-Time SBOM

Build-Time SBOMs

- As of now we have seen SBOMs generated after the build
- Those types of SBOM are prone to attacks and misconfigurations
- They also miss a lot of useful information
  - From which website were dependencies installed?
  - Which files were added and removed in the same instruction?

New idea!

- Generate the SBOM during the build process
- Derived from:
  - analysis of the network traffic
  - anaysis of the system calls
- Represent the **intended composition** of the artifact

Mini-projects

Build-time SBOM

- Goal: Evaluate build provenance vs final container content
- Use real-world projects producing container images
- Compare:
  - SBOM from container scanning
  - Use the Witness tool to build runtime attestations
- Select at least 2 projects:
  - Different programming languages (e.g., Go, Python, Node.js)
  - Must produce container images
- Example:
  - Go microservice
  - Python web app

Step 1: Container Scanning

- Build container images
- Scan using:
  - Syft
  - Trivy
  - Docker Scout
- Extract SBOM and dependency list

Step 2: Witness Execution

- ➜ Identify build pipeline (Dockerfile / CI)
- ➜ Re-run build step manually with Witness
- ➜ Collect attestations:
  - Build steps
  - Inputs and outputs
  - Network traces

Step 3: Comparison

- Compare:
  - Witness outputs (build-time view)
  - Scanner SBOM (artifact view)
- Identify:
  - Missing dependencies
  - Unexpected components
  - Build/runtime mismatches
  - Do not focus on OS dependencies (they will not be in the runtime SBOM)

Expected Findings

- ➜ Differences between the SBOMs
- ➜ Hidden or dynamically included components

SCA Manipulation

- ➜ Goal: Understand limitations of SCA tools
- ➜ Focus: evasion techniques via Dockerfile manipulation

Core Idea

- ➜ SCA tools rely on recognizable files
- ➜ Removing them affects detection accuracy
- ➜ You can start from the container obfuscation benchmark

Step 1: Replicate ORCA Findings

- ➜ Reproduce known SCA evasion techniques
- ➜ Modify Dockerfiles to remove dependency files
- ➜ Observe scanner behavior

Step 2: Manual Manipulation

- ➜ Modify Dockerfiles:
  - Delete dependency manifests after install
  - Obfuscate file locations
- ➜ Rebuild images and scan again

Step 3: Real-World Search

- Search GitHub for patterns:
  - `rm requirements.txt`
  - Similar removal patterns
- Analyze the reasons

Expected Findings

- ➜ SCA blind spots
- ➜ Reduced visibility after file removal
- ➜ Potential for malicious obfuscation

Secret Scanning

- ➜ Goal: Evaluate secret detection tools
- ➜ Use a known benchmark repository

Dataset

- Use leaky-repo benchmark
- Contains:
  - API keys
  - Tokens
  - Credentials

Tool Selection

- Select at least free 3 tools e.g.,:
  - TruffleHog
  - Gitleaks
  - GitGuardian

Execution

- Run all tools on the same repository
- Collect detected secrets
- Compare results
  - True positives
  - False positives
  - False negatives

Expected Findings

- ➜ Tools differ in accuracy
- ➜ No tool provides complete coverage

Intent-Based Networking with AI

- Goal: Generate network policies from natural language using AI
- Steps:
  - Select kubernetes application (list will be given separately)
  - Deploy the app in a Kubernetes cluster
  - Ask the AI to generate network policies for the given intent
  - Verify correctness

Define Intents

- Example intents:
  - Frontend cannot access internet
  - Backend only accessible internally
  - Database isolated

Topology Discovery

- ➜ Use KubeSonde
- ➜ Extract possible connections
- ➜ Build connectivity graph

Policy Generation

To generate policies

- ➜ Use a simple LLM
- ➜ Use e.g., Claude Code
- ➜ Use kagent
- ➜ Input: intents + topology
- ➜ Output: Kubernetes NetworkPolicies

Evaluation

- ➜ Correctness of generated policies
- ➜ Over-permissiveness vs strictness
- ➜ Alignment with intent

Expected Findings

- AI may be able to generate simple policies.
- How about the more complex ones?