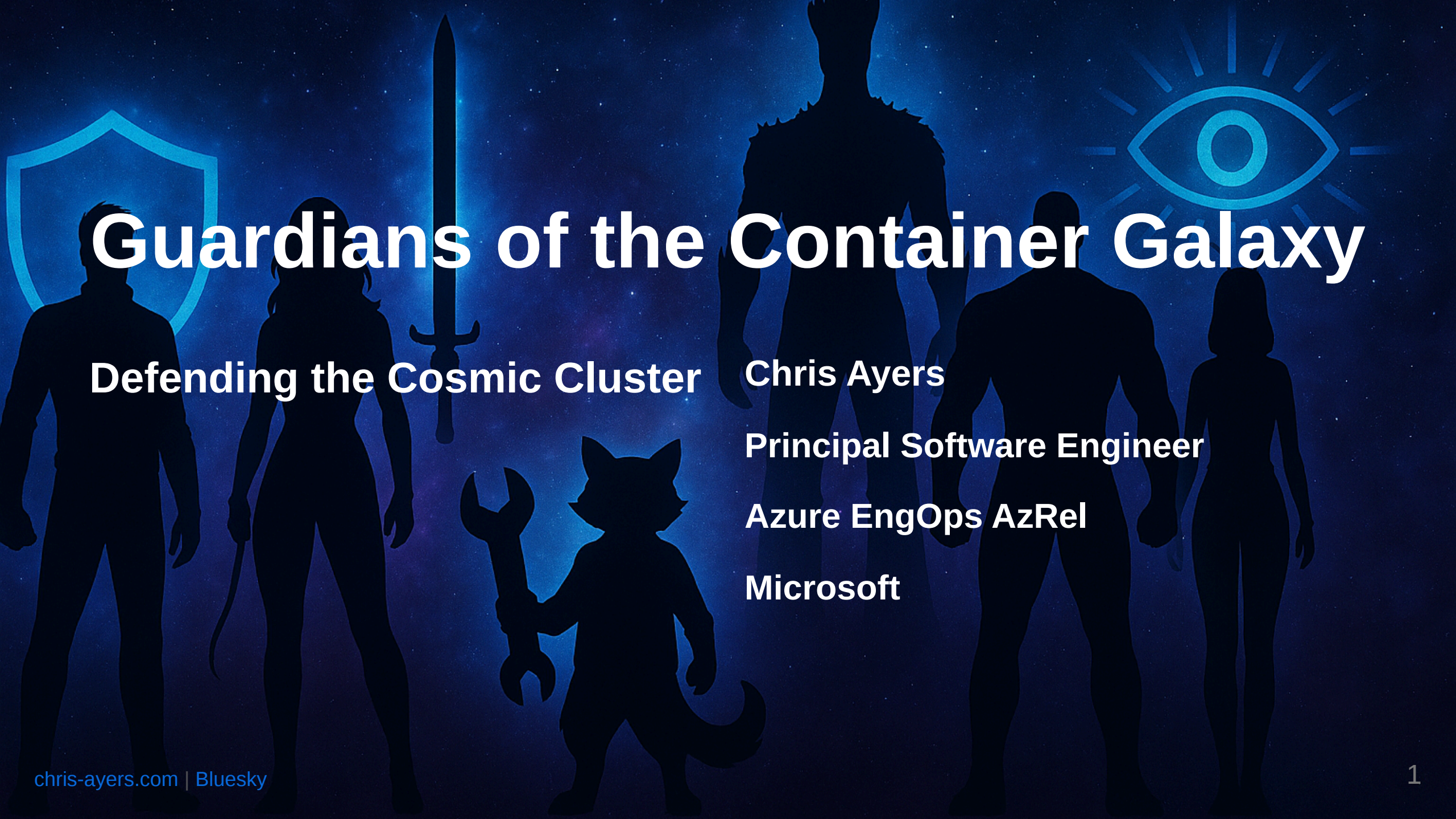


The background features a dark blue space-themed gradient with a starry pattern. Silhouettes of several figures are arranged across the scene: a man on the left with a shield behind him, a woman next to him, a large figure in the center background with a sword behind him, a small cat-like figure in the foreground holding a wrench, a muscular man on the right, and a woman on the far right. A large eye icon with radiating lines is positioned in the upper right corner.

Guardians of the Container Galaxy

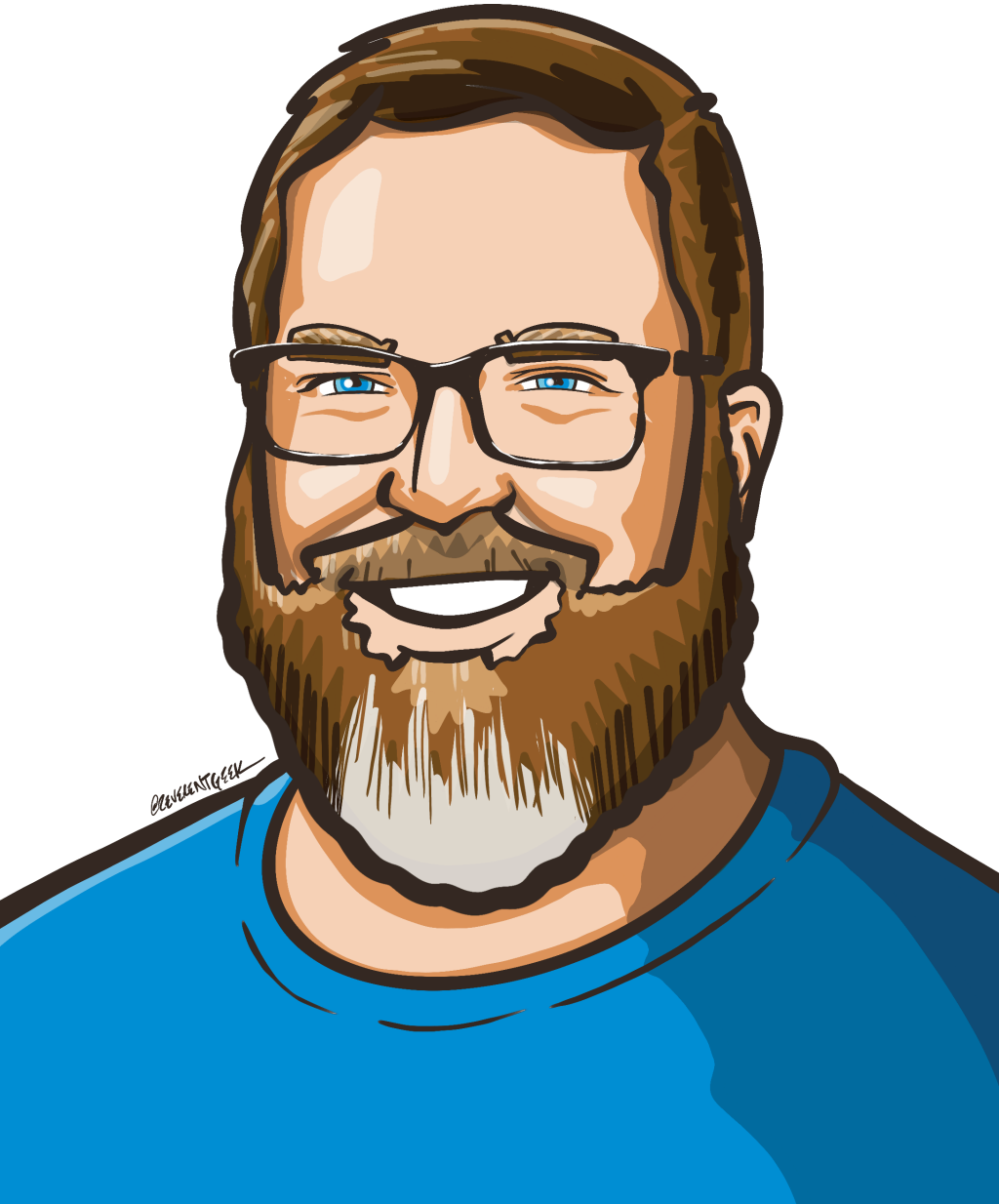
Defending the Cosmic Cluster

Chris Ayers

Principal Software Engineer

Azure EngOps AzRel

Microsoft



Chris Ayers

Principal Software Engineer
Azure EngOps AzRel
Microsoft

🦋 BlueSky: [@chris-ayers.com](https://chris-ayers.com)

in LinkedIn: - [chris-l-ayers](#)

📄 Blog: <https://chris-ayers.com/>

🐙 GitHub: [Codebytes](#)

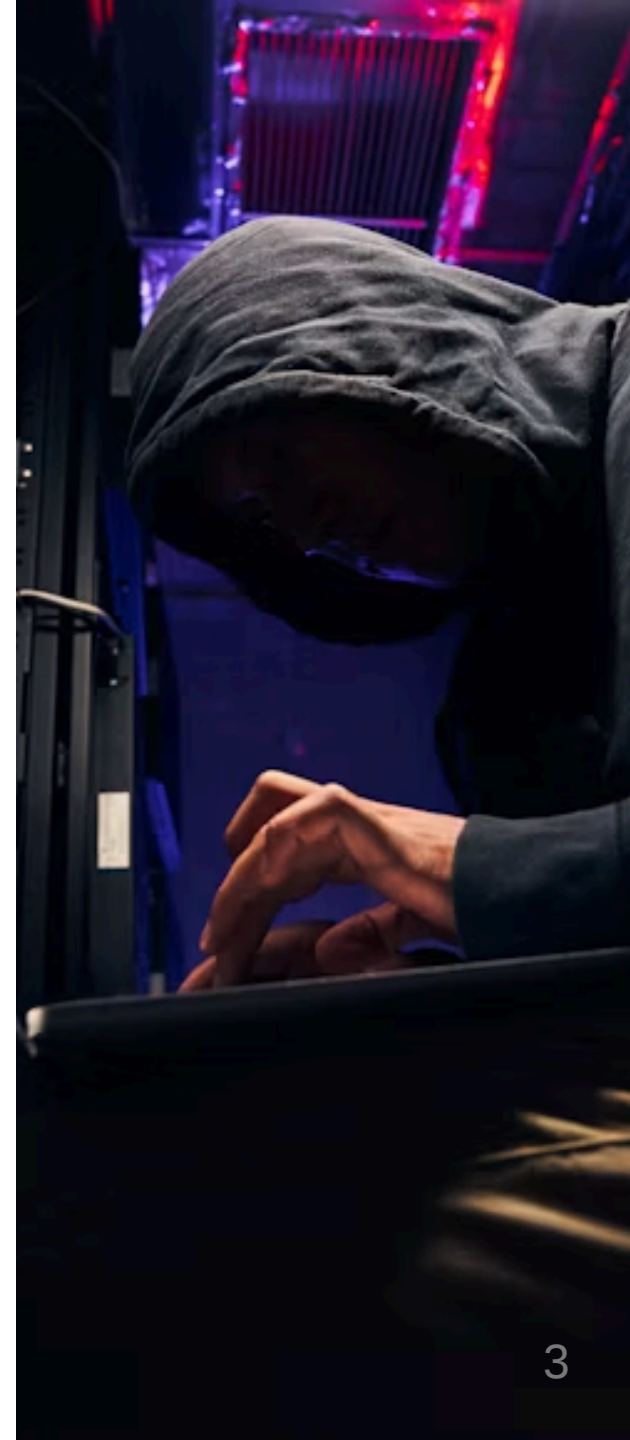
🐘 Mastodon: [@Chrisayers@hachyderm.io](https://hachyderm.io/@Chrisayers)

🐦 ~~Twitter: @Chris_L_Ayers~~

Container Security: The Challenge

Modern Container Threats:

- Supply Chain Attacks (xz-utils, LiteLLM, Axios)
- Runtime Exploits (Cryptojacking, Container Escape)
- Lateral Movement (Flat Networks)
- Visibility Gaps (Lack of Observability)

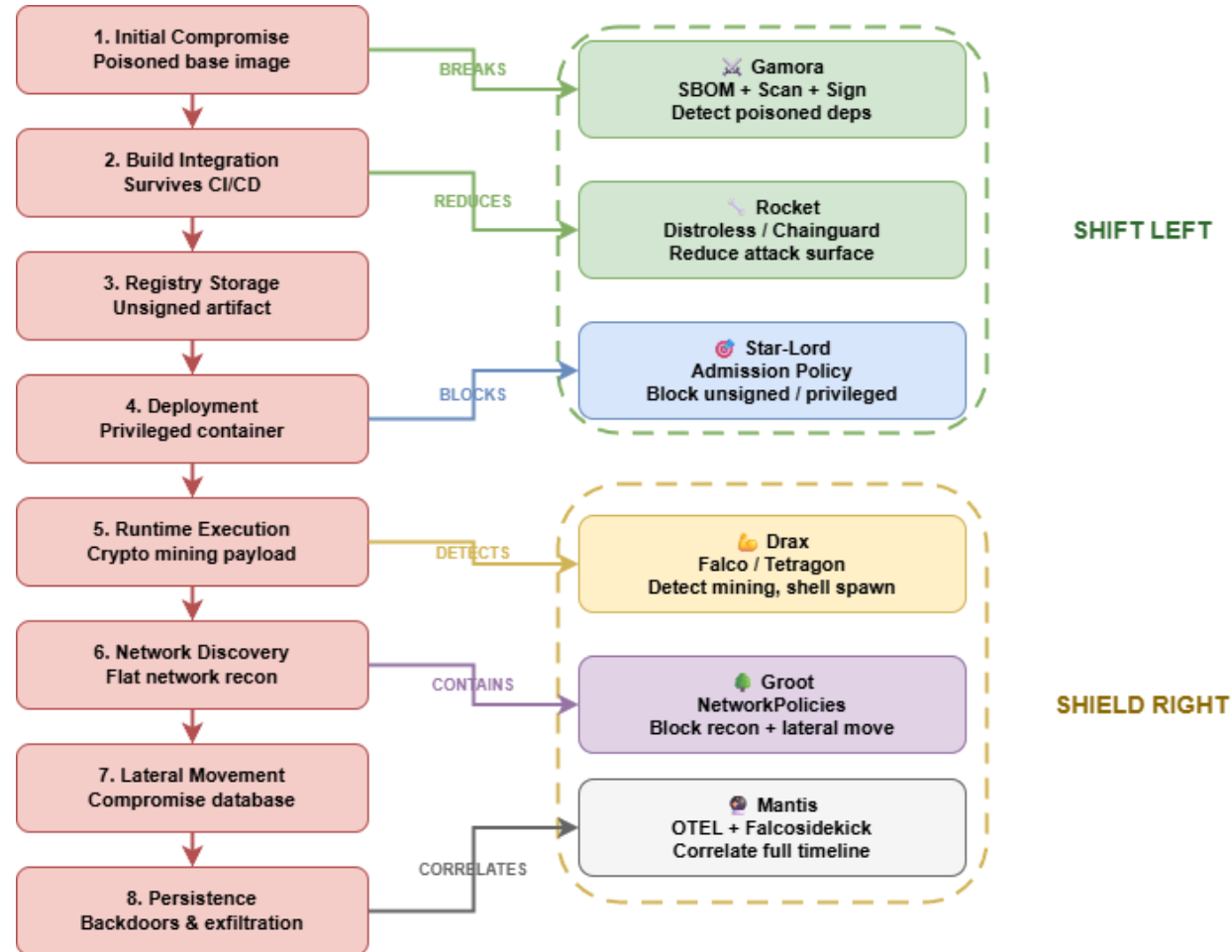


Container Security: The Impact

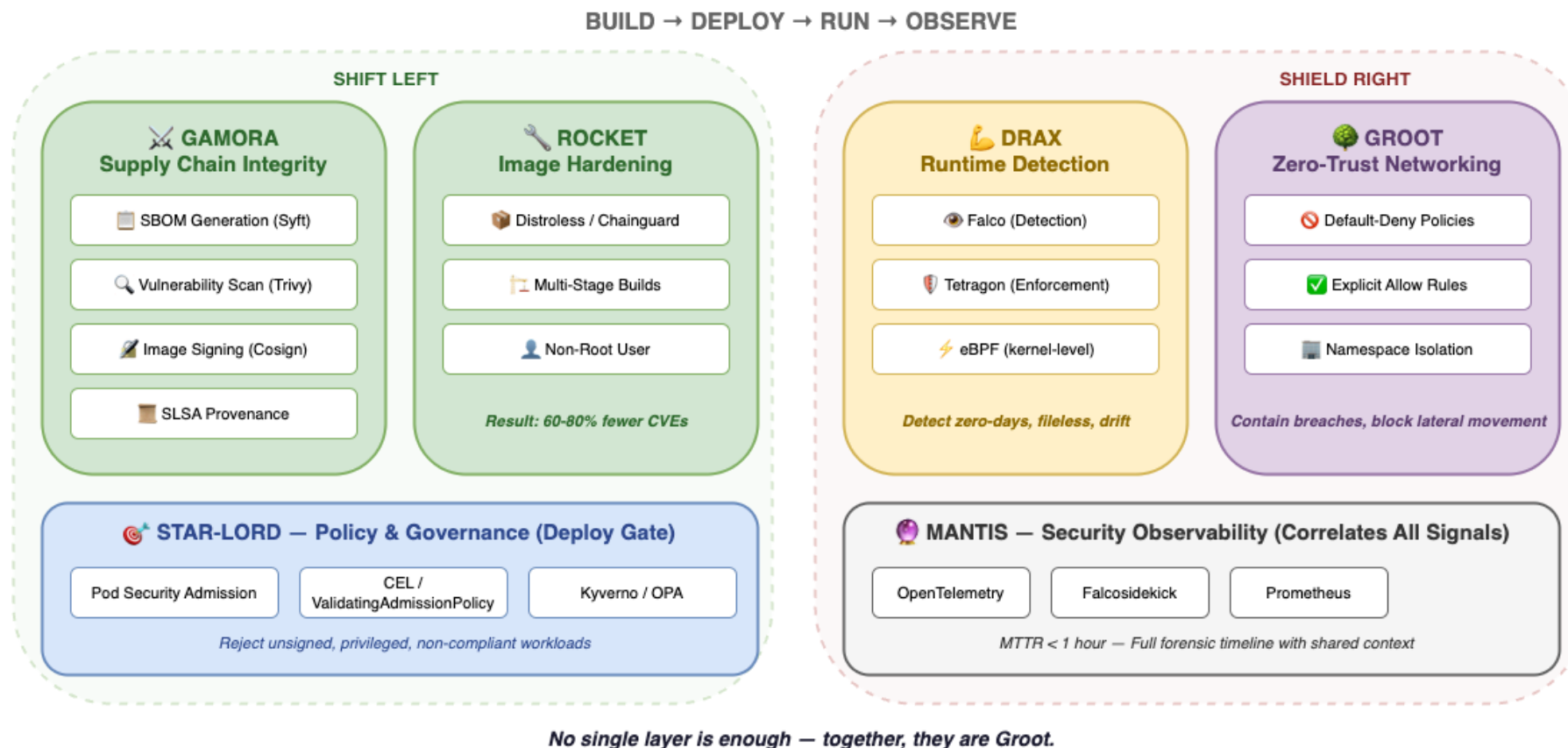
The Numbers:

- 78% of surveyed orgs fail audits due to unresolved container CVEs
- 63% of surveyed large enterprises hit by supply chain attacks (2024-2025)
- Recent M-Trends reports show dwell time still measured in days without runtime detection

The Container Attack Kill Chain



The Guardians Framework



Why Layering Matters

Security controls fail differently. Design for overlap, not perfection.

Log4Shell (CVE-2021-44228) is the pattern:

Layer	What It Contributed
Scanning	Flagged vulnerable Log4j packages in images
Runtime	Detected suspicious JNDI exploitation behavior
Network	Blocked outbound C2 traffic
Observability	Reconstructed timeline across signals



Shift Left + Shield Right

Motion	Focus	Core Controls	Outcome
Shift Left	Prevent known bad at build time	SBOM, vuln scanning, image signing, policy checks	Most issues stopped before deploy
Shield Right	Detect and contain runtime abuse	Behavioral detection, NetworkPolicy, observability	Faster detection and blast-radius reduction

Both are required: prevention reduces volume, runtime defense reduces impact.



Principles We'll Apply Throughout

Principle	Guardian	In Practice
Policy as Code	Star-Lord	Explicit, versioned, auditable gates
Supply Chain Security	Gamora	Verify artifacts and provenance
Least Privilege	Rocket	Smaller images, fewer privileges
Runtime Awareness	Drax	Detect hostile behavior fast
Zero Trust	Groot	Enforce east-west boundaries
Security Observability	Mantis	Correlate signals end-to-end

Standards + Tooling

Standards: NIST SP 800-207 | SLSA (OpenSSF)

Tooling approach: CNCF-first, portable, community-driven, production-proven

- **Graduated:** Falco, OPA, Cilium, Prometheus, Kyverno, OpenTelemetry
- **Incubating:** Trivy, Sigstore

Guardian #1

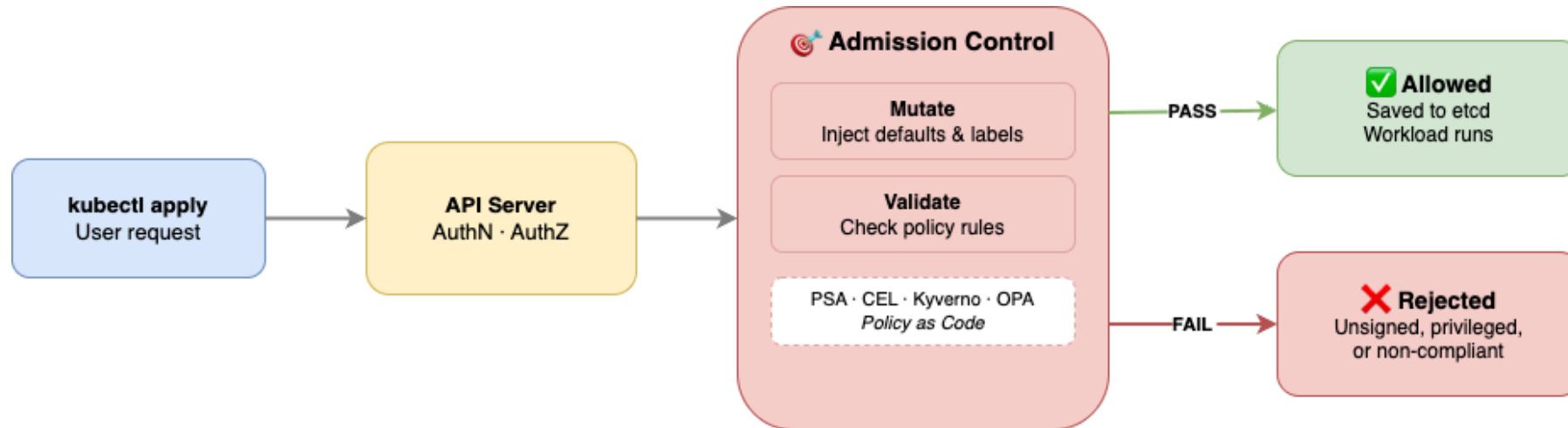
 Star-Lord

Policy Orchestration



POLICY

Star-Lord: Admission Control



Star-Lord: Policy as Code

Choose the simplest enforcement layer that solves the problem:

Layer	When to Use
Pod Security Admission (PSA)	Baseline pod hardening — fastest built-in guardrail
ValidatingAdmissionPolicy (CEL)	Simple custom validation — native, in-process (K8s v1.30+)
Kyverno / OPA Gatekeeper	Advanced policies — mutation, reporting, external data

All Policy as Code: version controlled, peer reviewed, auditable

Star-Lord: Image & Pod Policy Patterns

Image Trust:

- Require signed images (verify with Cosign)
- Block images from untrusted registries
- Deny `:latest` tag (enforce immutable tags)

Pod Hardening:

- Require non-root user
- Disallow privileged containers
- Drop all Linux capabilities by default

Star-Lord: Runtime & Governance Patterns

Runtime Boundaries:

- Block hostPath, hostNetwork, hostPID mounts
- Enforce read-only root filesystem
- Prevent privilege escalation

Operational Governance:

- Require resource limits (CPU, memory)
- Enforce required labels (team, cost-center)
- Restrict allowed namespaces / service accounts

Demo #1: Star-Lord

Policy Enforcement with Kyverno

What We'll Show:

1. Deploy Kyverno admission controller
2. Apply policy: non-root + simulated signed-image gate
3. Try insecure image tag → **✗ Blocked**
4. Try root container → **✗ Denied**
5. Deploy compliant workload → **✓ Success**

Guardian #2

✂️ Gamora

Supply Chain Integrity



**SUPPLY CHAIN
INTEGRITY**

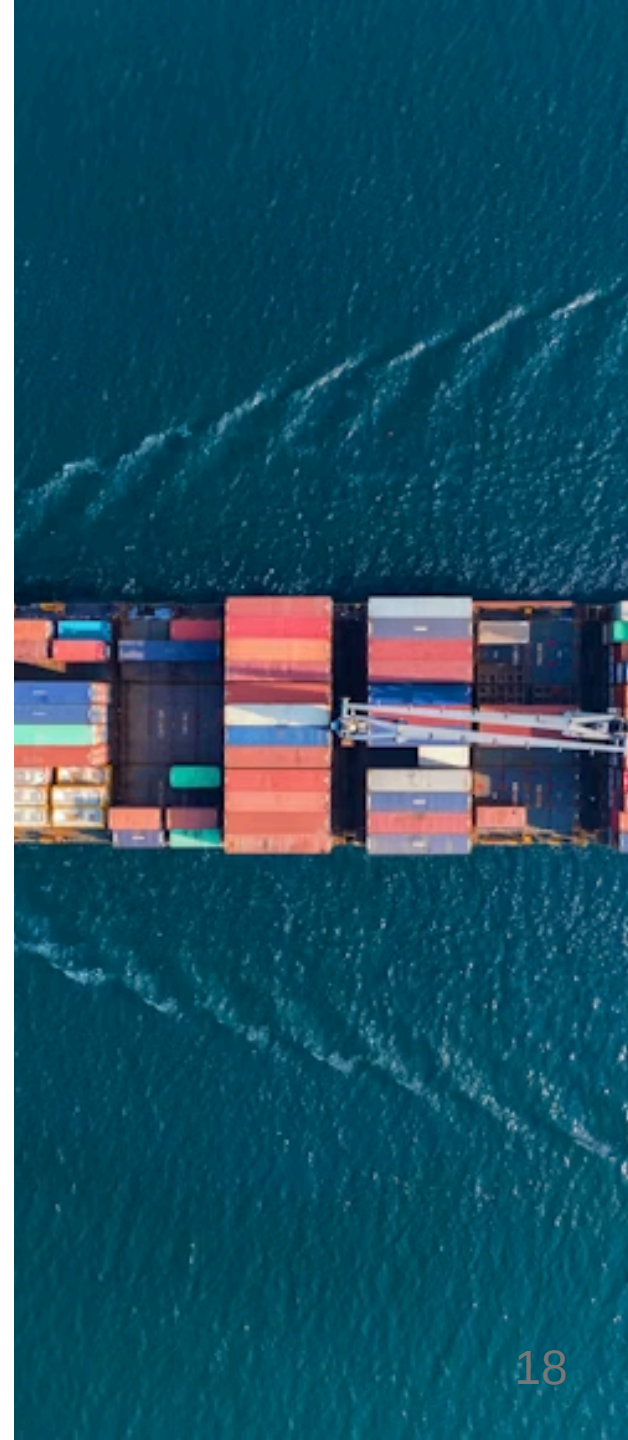
Gamora: The Supply Chain Problem

Trust is a vulnerability. You don't control:

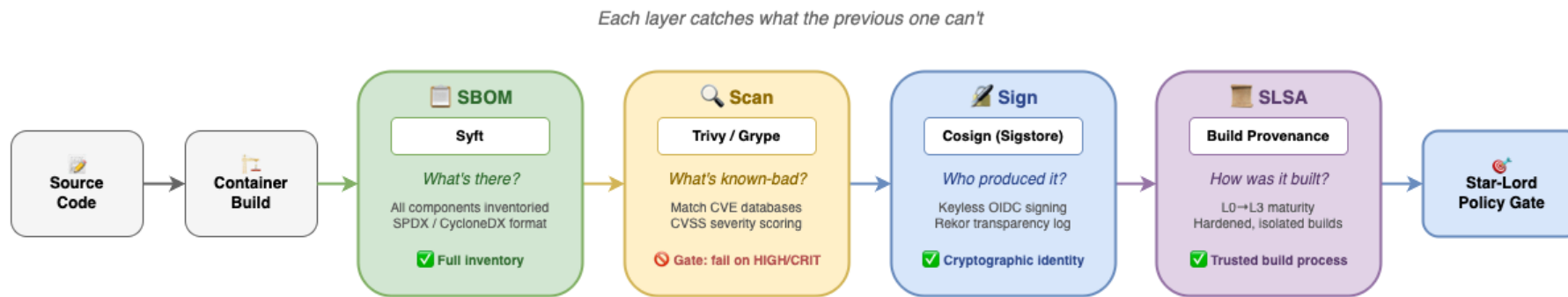
- Base images (Docker Hub, public registries)
- Transitive dependencies (your deps pull other deps)
- Build tools & CI infrastructure (can be compromised)

Real-World Proof:

- **xz-utils (2024):** Trusted maintainer planted SSH backdoor after 2 years
- **LiteLLM (2026):** Compromised security scanner → AI gateway backdoored on PyPI
- **Axios NPM (2026):** Hijacked account → RAT delivered to 70M+ weekly downloaders



🔪 Gamora: Supply Chain Defense Pipeline



Cryptographic trust from source code to production — each step adds assurance the previous can't provide

Gamora: Vulnerability Scanning & Signing

Vulnerability Scanning:

- Match packages against CVE databases (NVD, OSV)
- Severity scoring (CVSS) — **Gate:** Fail builds on HIGH/CRITICAL
- Tools: Trivy, Gype, Snyk

Cryptographic Signing:

- Keyless with OIDC is the production goal (no key management!)
- Sigstore: Cosign + Rekor + Fulcio
- **Verify:** Only signed images deploy
- *(Demo #2 uses a keyed example — committed `cosign.pub` — for offline, reproducible runs)*

Gamora: SLSA Framework

Supply chain Levels for Software Artifacts (v1.1)

- **Build L0:** No guarantees (status quo)
- **Build L1:** Build provenance exists
- **Build L2:** Hosted build platform (tamper-resistant)
- **Build L3:** Hardened build platform (isolated, auditable)

Goal: Move from L0 → L2+ for production

Standard: OpenSSF (Open Source Security Foundation)

Demo #2: Gamora

Complete Supply Chain Pipeline

What We'll Show:

1. Generate SBOM with Syft → See all packages
2. Scan image with Trivy → Find CVEs
3. Sign with Cosign → Reproducible keyed demo signature
4. Verify signature → Cryptographic proof
5. Deploy with policy → Only signed allowed

Key Takeaway: Cryptographic trust from build to deploy

Guardian #3

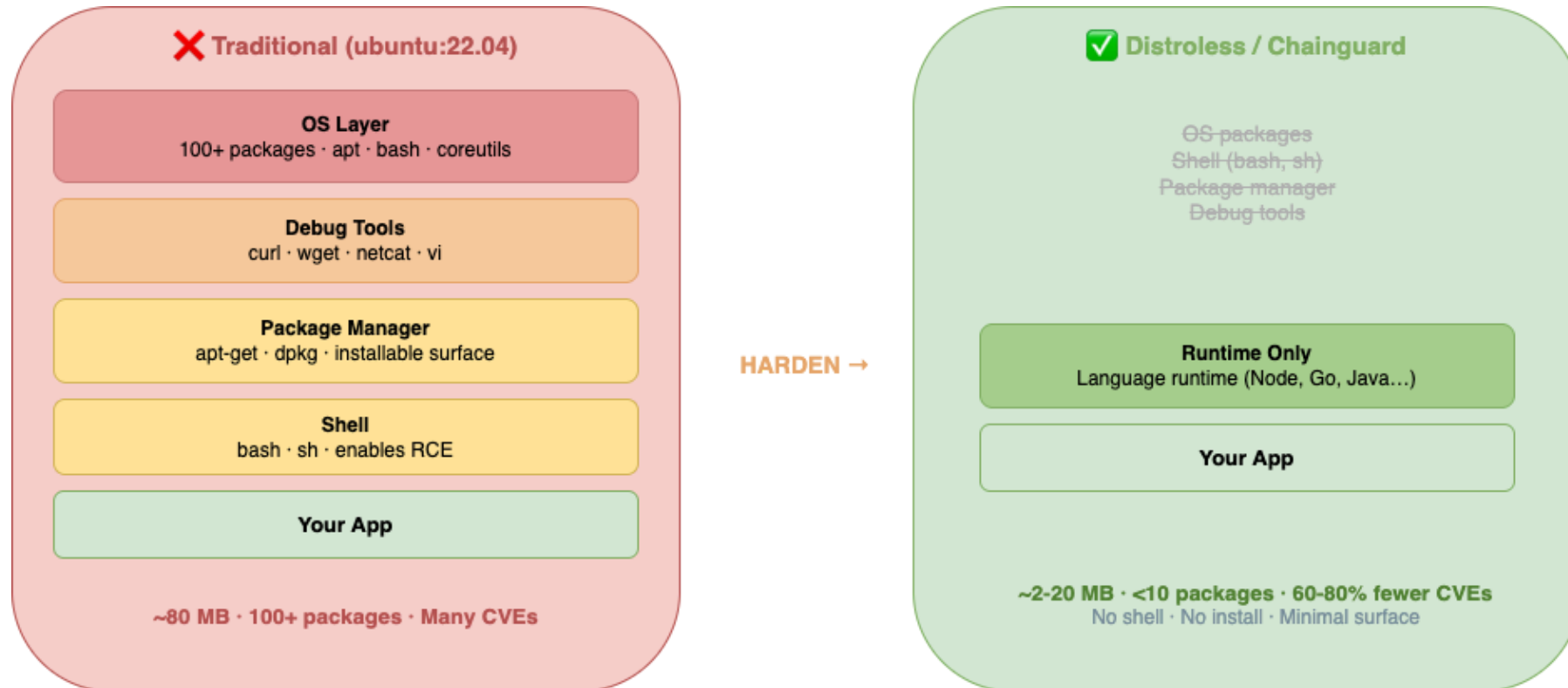
 Rocket

Image Hardening



**IMAGE
HARDENING**

Rocket: Before & After



Rocket: Distroless Philosophy

What is Distroless?

- **Only runtime dependencies** (language runtime + your app)
- **No shell** (bash, sh) → Can't RCE via shell injection
- **No package manager** → Can't install malware
- **No OS utilities** → Minimal attack surface

Rocket: Distroless by the Numbers

Illustrative static-base examples:

- Ubuntu base: ~80MB, 100+ packages
- Distroless: ~2-20MB, <10 packages

Result (this demo): 1322 → 26 HIGH/CRITICAL findings (~98% fewer); 430 → 34 OS packages

Modern Options:

- **Google Distroless** (Debian-based, Bazel builds)
- **Chainguard Images / Wolfi** (2,000+ images, nightly rebuilds, built-in SBOMs, near-zero CVEs)

Rocket: Multi-Stage Builds

Separate Build and Runtime:

```
# Stage 1: Build (has compilers, tools)
FROM python:3.11-slim AS build
WORKDIR /app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Stage 2: Runtime (minimal)
FROM gcr.io/distroless/python3-debian12
COPY --from=build /usr/local/lib/python3.11/site-packages \
    /usr/local/lib/python3.11/site-packages
COPY --from=build /app /app
USER 65532
ENTRYPOINT ["python", "/app/main.py"]
```

Build tools never reach production

Demo #3: Rocket

Image Hardening Before/After

What We'll Show:

1. Scan "before" (python:3.11-bullseye) → Count CVEs
2. Scan "after" (distroless python3-debian12) → Count CVEs
3. Compare: **1322** → **26 HIGH/CRITICAL findings** (~98% fewer)
4. Compare sizes: **1.42 GB** → **125 MB** (~91% smaller)
5. Show: No shell in distroless container

Key Takeaway: Minimal base = minimal risk

Guardian #4

 **Drax**

Runtime Detection



**RUNTIME
DETECTION**

Drax: Why Runtime Detection?

Build-time scanning can't detect:

- **Zero-day exploits** → No CVE exists yet
- **Fileless attacks** → Malware in memory only
- **Living-off-the-land** → Abuse curl, bash, legitimate tools
- **Insider threats** → Authorized malicious actions
- **Configuration drift** → Runtime container changes

Remember dwell time is still measured in days? Runtime detection is how you shrink it further.

You need eyes on running containers



Drax: eBPF Technology

Extended Berkeley Packet Filter

What is eBPF?

- Kernel-level syscall monitoring
- Verified safe by kernel (can't crash system)
- Low-overhead, event-driven, JIT-compiled
- Kernel-level visibility without polling every process
- Harder for userspace malware to tamper with than app logs

Used by: Cilium, Falco, Tetragon, Pixie, Hubble

Industry consensus: eBPF is the future of observability

Drax: Detection vs. Enforcement

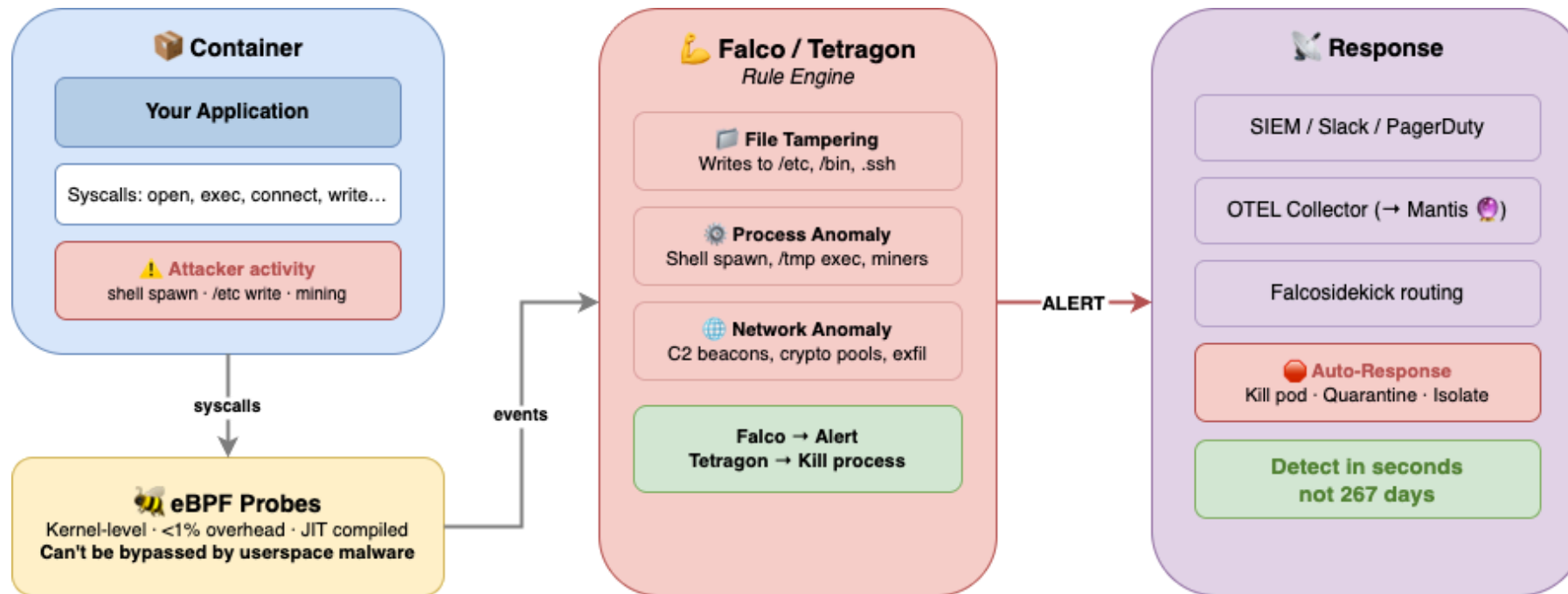
Falco (CNCF Graduated) → **Detection** (alert on suspicious behavior)

Tetragon (part of Cilium) → **Enforcement** (kill processes, block syscalls in-kernel)

Use Falco for broad behavioral monitoring + alerting

Use Tetragon when you need real-time kernel-level blocking

Drax: Detection Architecture



Demo #4: Drax

Runtime Detection with Falco

What We'll Show:

1. Deploy Falco with modern eBPF
2. Apply custom rule: Detect /etc writes
3. Monitor Falco logs real-time
4. Trigger: Pod writes /etc/shadow, /etc/passwd
5. Observe: Alerts with pod, file, user context

Key Takeaway: Detect malicious behavior instantly

Guardian #5



Zero-Trust Networking



**ZERO-TRUST
NETWORKING**

Groot: The Lateral Movement Problem

Kubernetes Default: Flat Network

- Any pod can reach any other pod
- No network boundaries between namespaces
- Attacker compromises frontend → pivots to database
- A single vulnerability can expand blast radius without segmentation

Cloud lateral-movement lesson: Capital One breach (2019)

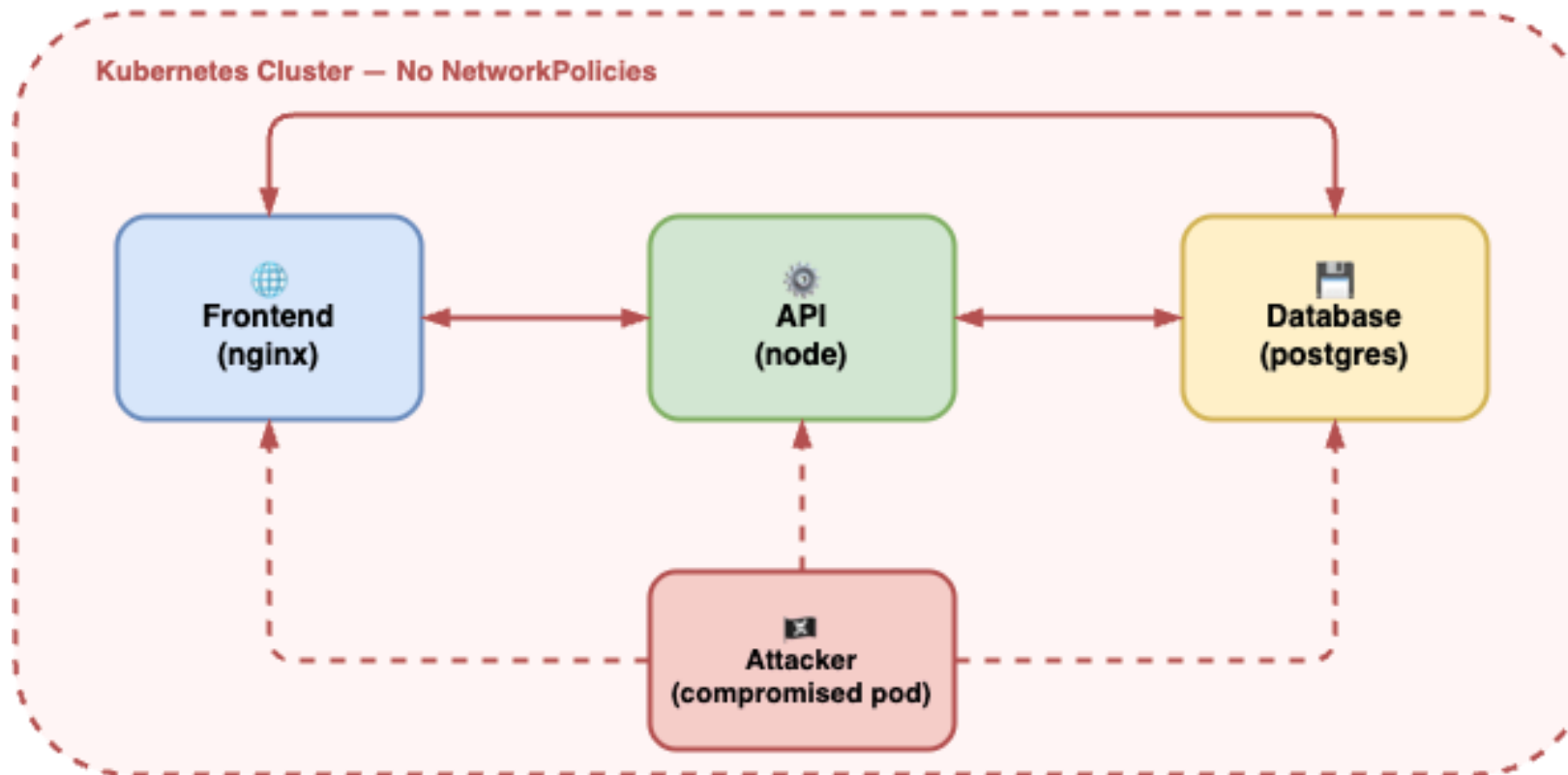
- SSRF in web app → AWS metadata service
- Over-permissioned credentials → S3 bucket access
- **Lesson:** segmentation and least privilege limit pivots



Groot: Zero Trust in Kubernetes

✗ BEFORE: Flat Network (Default Kubernetes)

Any pod can reach any other pod — single compromise = full access



Groot: Kubernetes NetworkPolicies

How They Work — by example:

- **Selector:** "This policy applies to pods labeled `app=api` "
- **Ingress:** "Only `app=frontend` can call the API"
- **Egress:** "API can only connect to `app=database` "
- **Namespace:** "Nothing in `dev` can reach `prod` "

CNI Plugin Required: Calico or Cilium; Docker Desktop's default cluster accepts NetworkPolicy objects but does not enforce them without an enforcing CNI

Beyond L3/L4: Service mesh (Istio, Linkerd) adds mTLS + L7 identity

Demo #5: Groot

Zero-Trust Network Policies

What We'll Show:

1. Deploy 3-tier app (default flat network)
2. Show baseline risk: no policy boundary yet
3. Apply default-deny → All blocked
4. Apply allow rules → Only approved paths
5. Test: Tester can't reach API/DB ✓
6. Test: Frontend → API → DB works, rest blocked ✓

Key Takeaway: Contain breaches, prevent lateral movement

Guardian #6

 Mantis

Security Observability



Mantis: The Observability Gap

Siloed Teams, Siloed Tools:

Without Correlation:

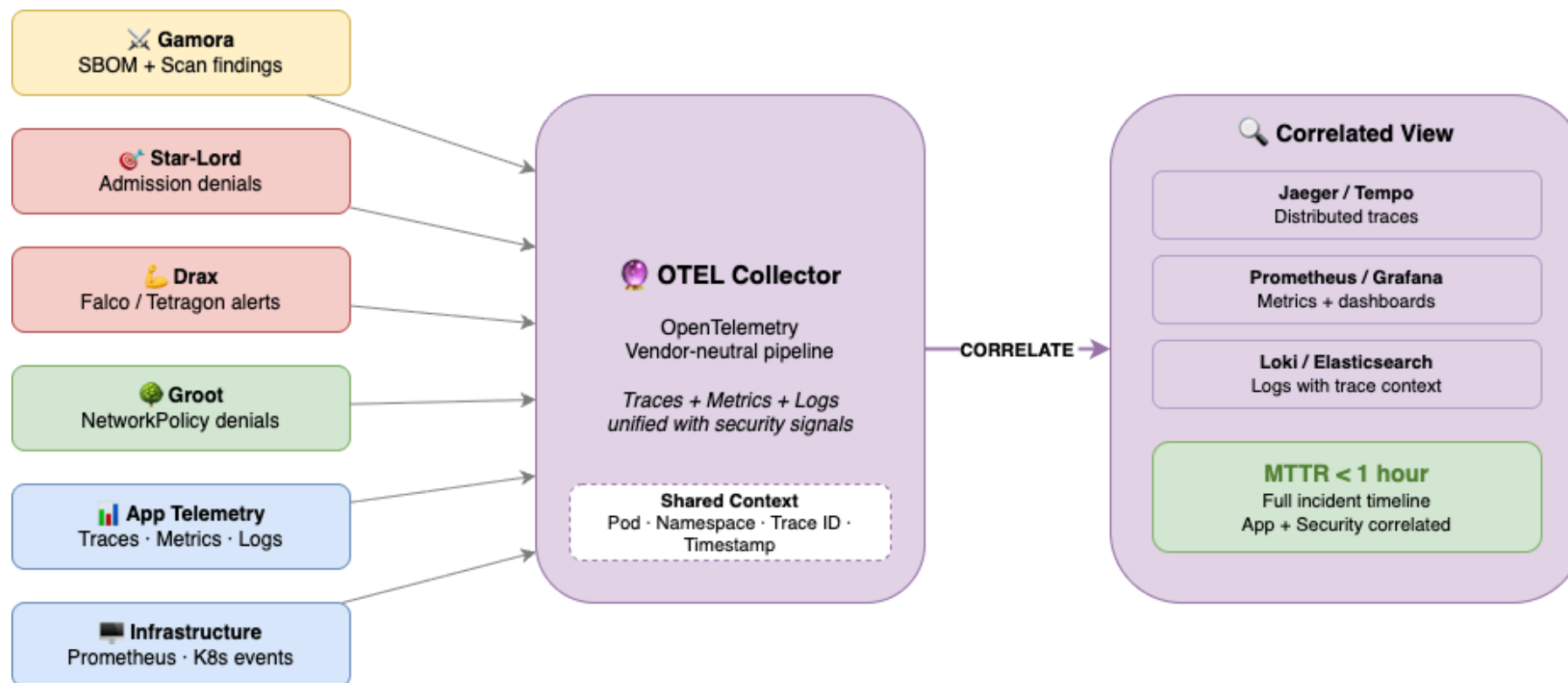
- **Ops team:** "API is slow" (looks at Grafana)
- **Security team:** "No alerts" (checks SIEM)
- **Reality:** Crypto miner running for days

Mantis: Correlation in Action

With Correlation:

- **9:00 AM:** API latency spike (APM)
- **9:02 AM:** High CPU usage (Prometheus)
- **9:02 AM:** Suspicious process (Falco alert)
- **Context:** Same pod, namespace, and timestamp window
- **Result:** Detected in minutes, not days

Mantis: Observability Correlation



Mantis: The Observability Context

Common Context:

- Pod name, namespace
- Trace ID (links requests across services)
- Timestamp (timeline reconstruction)

Internal goal: Drive Mean Time To Respond (MTTR) toward < 1 hour

Mantis: OpenTelemetry for Security

Why OTEL Matters:

Traces: Show which services were accessed during incident

Metrics: Detect resource anomalies (CPU spike = crypto miner)

Logs: Capture security-relevant events with context

Vendor-Neutral: Single instrumentation → any backend

- Jaeger, Prometheus, Grafana
- Datadog, New Relic, Splunk
- **No lock-in**

Demo #6: Mantis

Observability Correlation

What We'll Show:

1. Deploy OTEL collector + instrumented app
2. Wire Falcosidekick/adaptor for Demo #4 alerts
3. Generate traffic → See traces in logs
4. (Optional) Trigger Falco → See security events alongside app traces
5. Correlate by pod, namespace, and timestamp

Key Takeaway: Link security to business impact

Guardians Together: Prevent & Harden

Scenario: Cryptominer in compromised Node.js image

Attack Step	Guardian	Action	Result
Poisoned base image	 Gamora	Scan + SBOM detects vuln	⚠️ Known threats caught
Bloated surface	 Rocket	Distroless reduces tooling	✅ Less to exploit
Unsigned deploy	 Star-Lord	Policy rejects image	✅ Blocked at gate

Prevention catches what's known — but what gets through?

Guardians Together: Detect & Contain

The attacker bypassed build-time controls...

Attack Step	Guardian	Action	Result
Mining process spawns	 Drax	Falco detects anomaly with tuned rules	✅ Near-real-time alert
C2 network beacon	 Groot	Egress policy blocks it	✅ Contained
Full timeline needed	 Mantis	Correlates all signals	✅ MTTR trending toward target

Not every layer prevents — some reduce, some detect, some contain.

Container Security Maturity Model

Level	Actions to Reach It
0 → 1	Image scanning in CI, Pod Security Admission (audit)
1 → 2	Image signing + verification, default-deny NetworkPolicies
2 → 3	Runtime detection (Falco), observability correlation, mTLS
3 → 4	Attestations, automated response, MTTR optimization

Your First Week

Concrete Steps to Start Monday:

-  **Day 1:** Run `trivy image` on your top 5 production images
-  **Day 2:** Generate your first SBOM with `syft` → know your dependencies
-  **Day 3:** Apply `Restricted` Pod Security Standard to one namespace (audit mode)
-  **Day 4:** Apply default-deny NetworkPolicy to one namespace
-  **Day 5:** Deploy Falco in dry-run mode → see what it detects

Don't boil the ocean: pick one namespace, one app, one pipeline.

Key Takeaways

1. **Defense in Depth** — No single tool is enough
2. **Shift Left + Shield Right** — Prevention AND detection required
3. **Principles First, Tools Second** — Pick controls you can actually run consistently
4. **Start Small** — One namespace, one pipeline, prove value, expand
5. **Measure Progress** — CVEs blocked, MTTR, coverage %

"We are layered" — Security is a team sport

Questions?



Resources & Links

- **Repo:**
github.com/codebytes/container-security
- **Slides:** chris-ayers.com/container-security
- NIST SP 800-207
- SLSA Framework
- CIS Kubernetes Benchmark
- CNCF Security TAG

Chris Ayers

Principal Software Engineer

Azure EngOps AzRel

Microsoft

 BlueSky: [@chris-ayers.com](https://chris-ayers.com)

 LinkedIn: - [chris-l-ayers](https://www.linkedin.com/in/chris-l-ayers)

 Blog: <https://chris-ayers.com/>

 GitHub: [Codebytes](https://github.com/codebytes)

 Mastodon:

[@Chrisayers@hachyderm.io](https://hachyderm.io/@Chrisayers)