Auditing MCP Tool Calls at the Kernel Level

eBPF as a Trust Boundary Enforcer

By: **Harini Anand**

$whoami

- **SDE at IBM** under the **Data & AI** Division
- **Software Engineering Research** at **UIUC**
- **Computational Cognition Research** at **Georgia Institute of Technology**
- **Biomedical XAI Research** at **Dartmouth College**
- **Google KaggleX Mentee, AWS Scholar, Harvard WE Tech Fellow, and Cohere, University of Oxford & MIT** Summer School alumna
- **Stanford** High Impact WiDS Ambassador

The Question Nobody Is Asking

"Your LLM just called a tool. What did it actually do?"

- You have the MCP request ID
- You have the tool response
- You **do not** have what happened between them

The gap is not a logging problem. It's a trust problem.

You think you know	Reality
`read_file` read the file you expected	It also opened `/etc/passwd`
`call_api` hit your internal service	It also connected to `198.51.100.42:443`
`exec_shell` ran your command	It also spawned a `curl` subprocess

Why Application Logs Are Not Enough

LLM Client → [HTTP] → MCP Server → Tool Handler → ??? → System

↑

audit gap lives here

App-layer logs fail because they:

- **Only record what the code intended to record**: bugs and injections skip logging branches
- **Can be bypassed** by the exact vulnerability being exploited
- **Cannot see child processes**: subprocess spawns are invisible to the parent's logger
- **Can be prompt-injected away**: LLM-crafted inputs can alter execution paths before logging

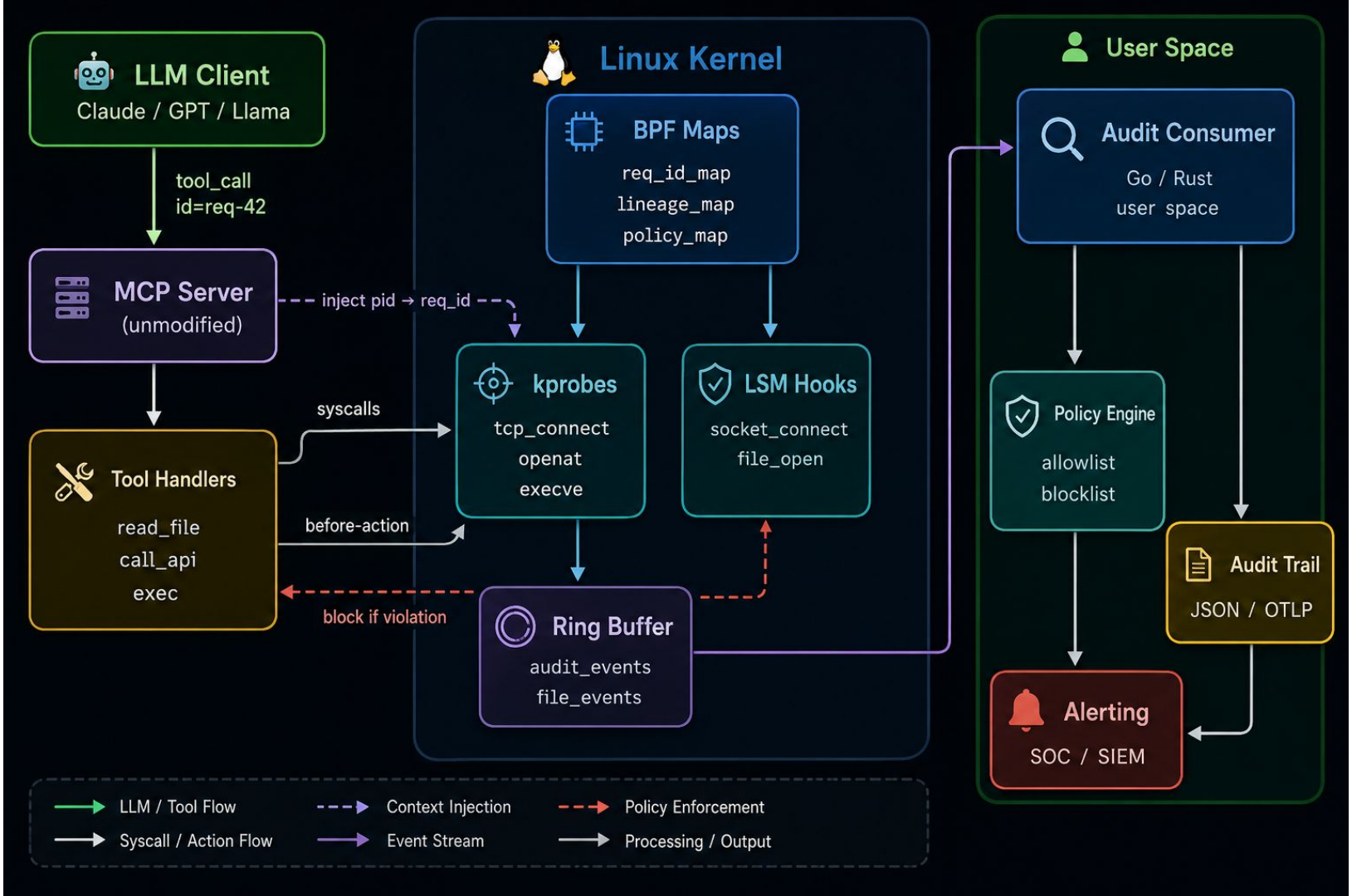
Only the kernel sees everything. Always.

eBPF programs **run inside the Linux kernel**, attached to events that application code cannot suppress

Kernel event	What it catches
`tcp_connect` kprobe	Every outbound connection, regardless of library
`sys_openat` kprobe	Every file opened, even by child processes
sched_process_exec	Every `execve`, subprocess spawn, and shell escapes
LSM `socket_connect` hook	Called **before** the connection, can block it
LSM `file_open` hook	Called **before** the file open, can block it

Properties that matter:

- MCP server process **cannot detect** eBPF programs attached to it
- MCP server process **cannot disable** them
- Ring buffer output is **append-only** from the kernel's perspective


LLM Client
Claude / GPT / Llama
tool_call
id=req-42
MCP Server
(unmodified)
inject pid → req_id
Tool Handlers
read_file
call_api
exec
syscalls
before-action
block if violation
Linux Kernel
BPF Maps
req_id_map
lineage_map
policy_map
kprobes
tcp_connect
openat
execve
LSM Hooks
socket_connect
file_open
Ring Buffer
audit_events
file_events
User Space
Audit Consumer
Go / Rust
user space
Policy Engine
allowlist
blocklist
Audit Trail
JSON / OTLP
Alerting
SOC / SIEM
LLM / Tool Flow
Syscall / Action Flow
Context Injection
Event Stream
Policy Enforcement
Processing / Output

The Correlation Problem

Hard part: Kernel events carry PIDs. MCP requests carry IDs. How do you connect them?

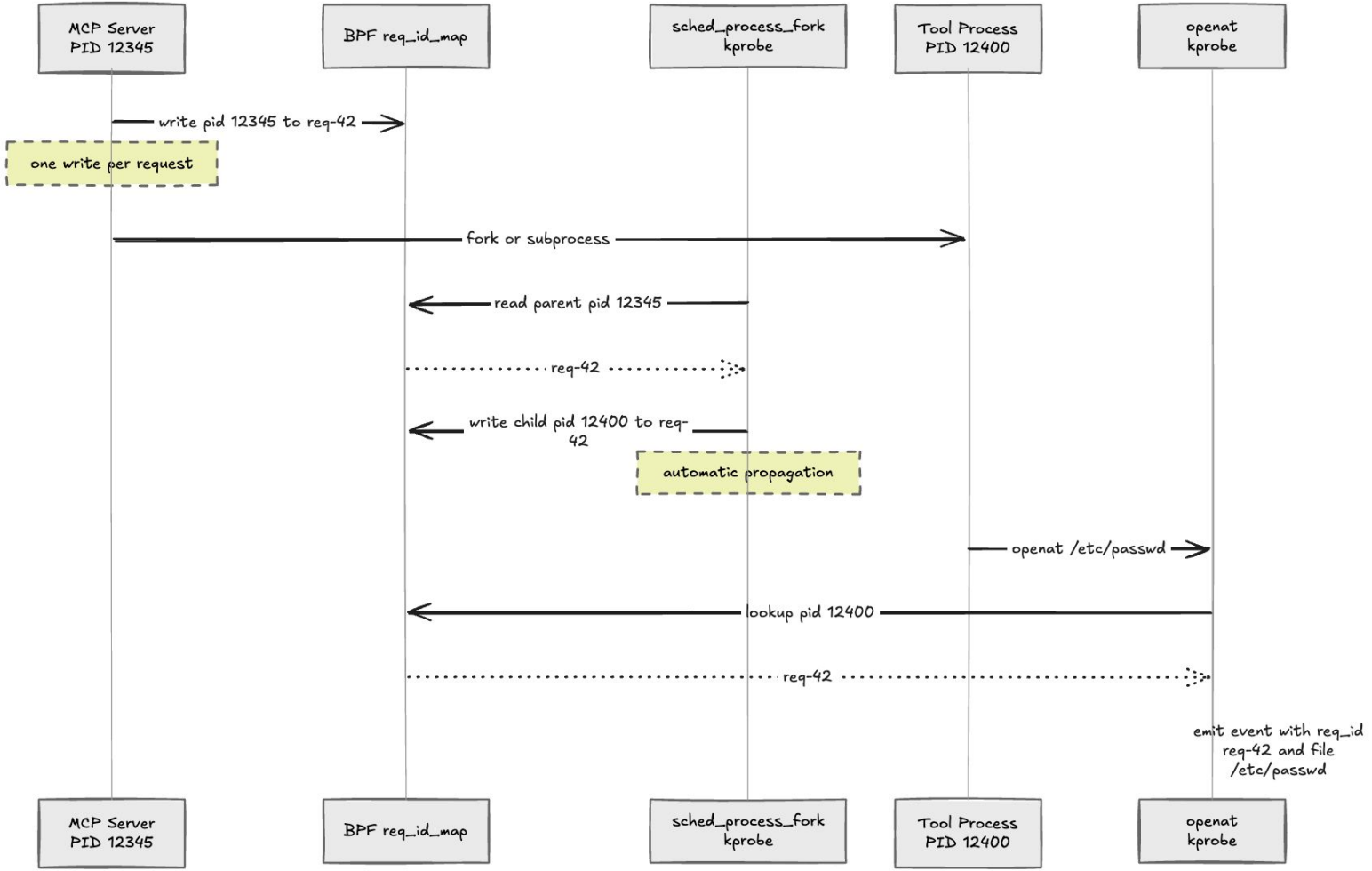
MCP Request ID: "req-42"

└── handled by: MCP Server PID 12345

└── spawned: Tool subprocess PID 12400

└── opened: /etc/passwd ← kernel sees PID 12400, not "req-42"

The naive approach fails: You can't just watch all of PID 12345's syscalls; one MCP server handles concurrent requests from hundreds of LLMs.


MCP Server PID 12345
BPF req_id_map
sched_process_fork kprobe
Tool Process PID 12400
openat kprobe
write pid 12345 to req-42
one write per request
fork or subprocess
read parent pid 12345
req-42
write child pid 12400 to req-42
automatic propagation
openat /etc/passwd
lookup pid 12400
req-42
emit event with req_id req-42 and file /etc/passwd
MCP Server PID 12345
BPF req_id_map
sched_process_fork kprobe
Tool Process PID 12400
openat kprobe

kprobe 1: Auditing Network Egress

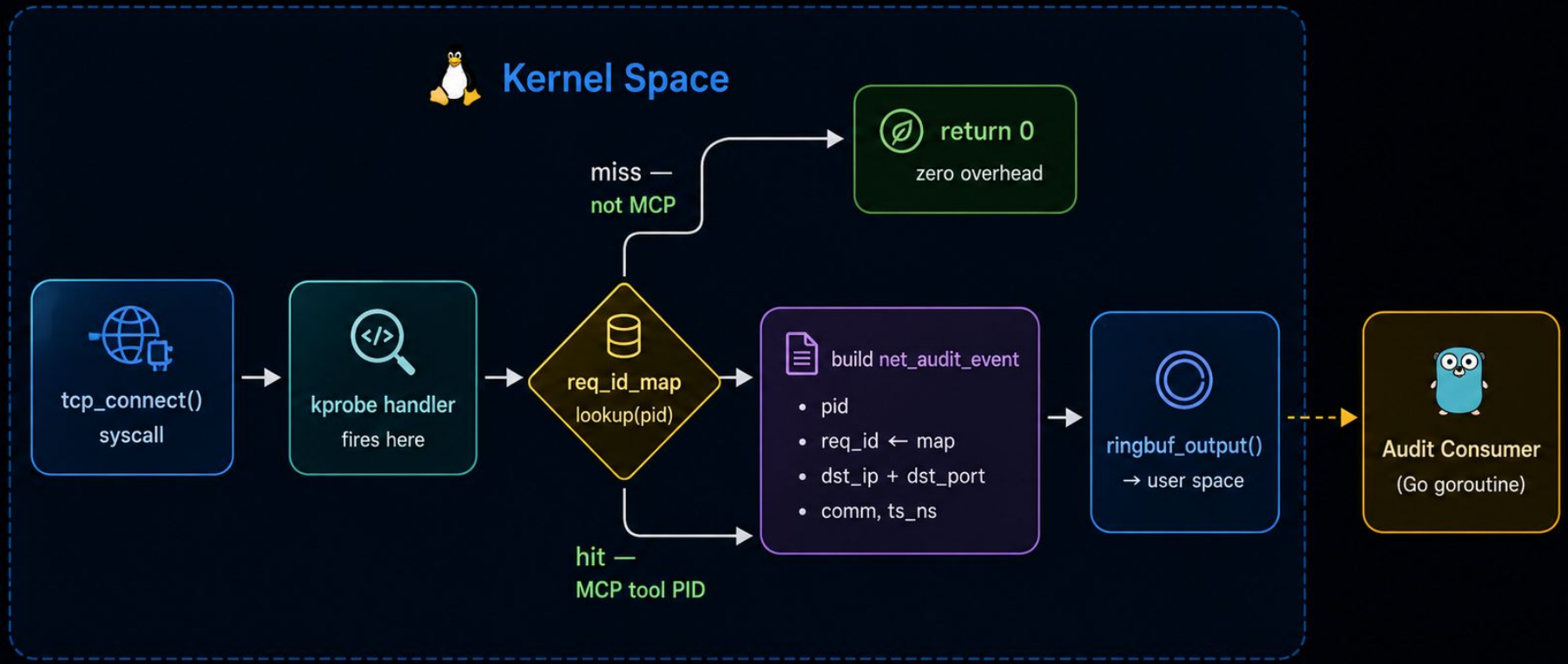
`tcp_connect`: fires on every outbound TCP connection, any library, any process

What is captured per event:

- `req_id`: which MCP request triggered this (map lookup, not app-layer log)
- `dst_ip` + `dst_port:` where the tool tried to connect to
- `comm`: process name could be a spawned subprocess, not the MCP server itself
- `ts_ns`: kernel monotonic timestamp, cannot be faked by user space

The zero-overhead guarantee

- One hash map lookup per syscall for non-MCP processes → immediate exit
- No false positives, no performance tax on the rest of the system
- Only MCP tool PIDs in req_id_map incur the full event path



tcp_connect()
User-space initiates an outbound connection

kprobe handler
eBPF kprobe attached to tcp_connect

req_id_map lookup
Check if PID belongs to an MCP tool

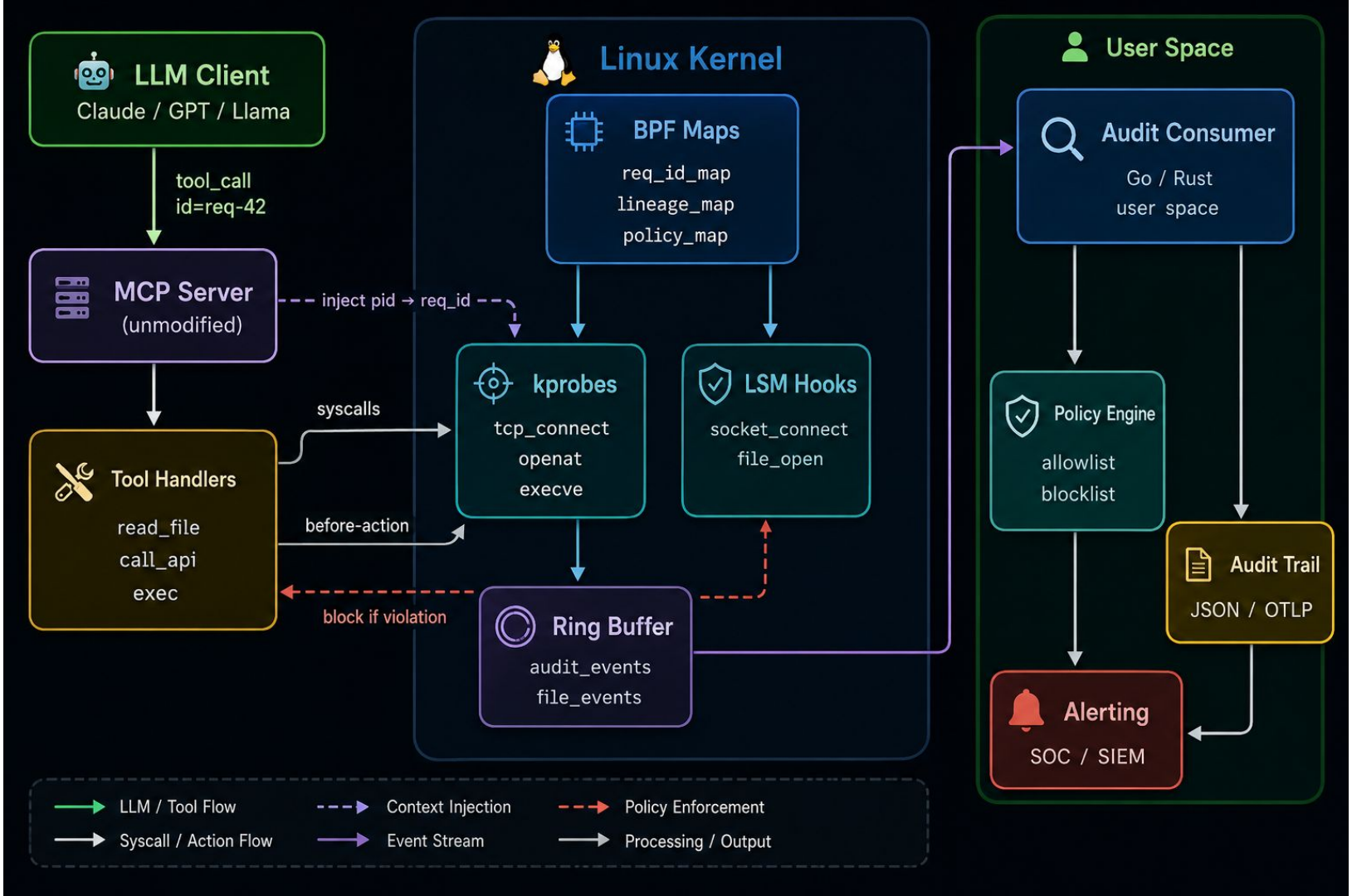
net_audit_event
Build enriched event with context for auditing

ringbuf_output()
Send event to user space via ring buffer

Audit Consumer
Userspace Go routine processes the event

kprobe 2 + 3: File Access and Fork Propagation

Two hooks working together: attribution self-assembles


LLM Client
Claude / GPT / Llama
tool_call
id=req-42
MCP Server
(unmodified)
inject pid → req_id
Tool Handlers
read_file
call_api
exec
syscalls
before-action
block if violation
Linux Kernel
BPF Maps
req_id_map
lineage_map
policy_map
kprobes
tcp_connect
openat
execve
LSM Hooks
socket_connect
file_open
Ring Buffer
audit_events
file_events
User Space
Audit Consumer
Go / Rust
user space
Policy Engine
allowlist
blocklist
Audit Trail
JSON / OTLP
Alerting
SOC / SIEM
LLM / Tool Flow
Syscall / Action Flow
Context Injection
Event Stream
Policy Enforcement
Processing / Output

Why fork propagation is the core insight

- MCP server writes **exactly one entry**: `{server_pid → req_id}` at request start
- Every `fork()`, shell escape, subprocess spawn: child inherits automatically
- The whole process tree is attributed to `req_id` with no further user-space action
- Zero modification to the tool handler code

open_flags as a risk signal:

Flag	Meaning	Risk
`O_RDONLY` **(0)**	Read only	Expected: baseline
`O_WRONLY` **(1)**	Write only	Flag if path is unexpected
`O_RDWR` **(2)**	Read + write	Higher: alert on sensitive paths
`O_CREAT` **(64)**	Create new file	Alert if outside expected dirs

LSM Hooks: From Audit to Enforcement

kprobes = observe what happened. LSM hooks = prevent it.

Three properties that make this production safe:

- **Allowlist by default**: anything not in policy_map gets blocked. Zero-trust posture.
- **Hot-reloadable policy**: update policy_map from Go at runtime. No eBPF reload. No restart.
- **Minimal privilege**: CAP_BPF (Linux 5.8+), not full root. Narrow attack surface.


Audit Only (kprobe)
connect() executes
kprobe fires
AFTER start
event logged
data may
already be sent
Enforcement (LSM hook)
connect() called
LSM hook fires
BEFORE execution
policy_map
lookup(ip, port)
in allowlist
not in allowlist
return 0
connection proceeds
return -EPERM
no SYN sent
no bytes leave machine

Requirements:

Kernel ≥ 5.7 + CONFIG_BPF_LSM=y + CAP_BPF capability

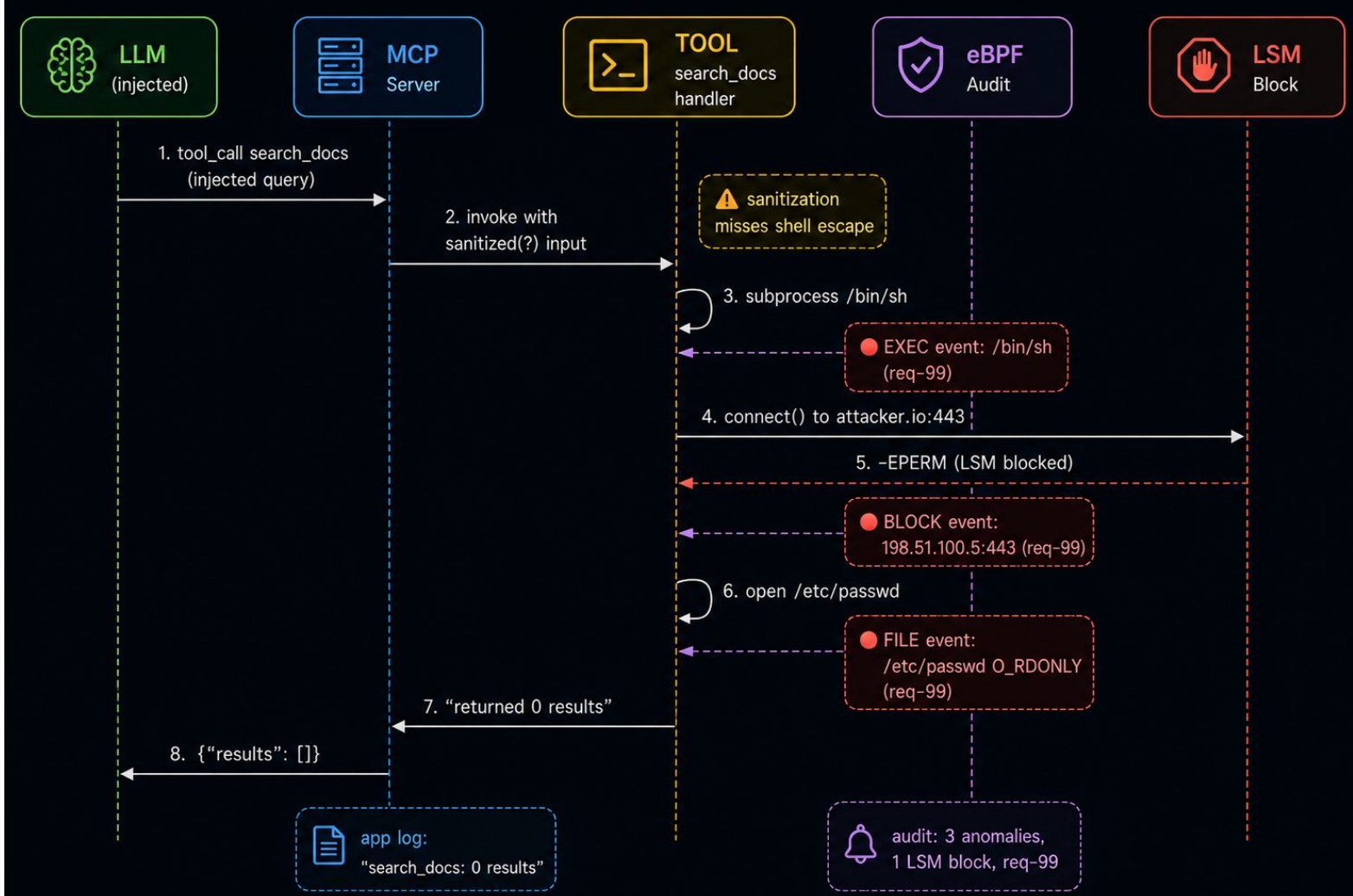
What the tool sees on block: connect(): Connection refused: a normal syscall error. The application has no visibility into the eBPF program that caused it.

The Complete Audit Record

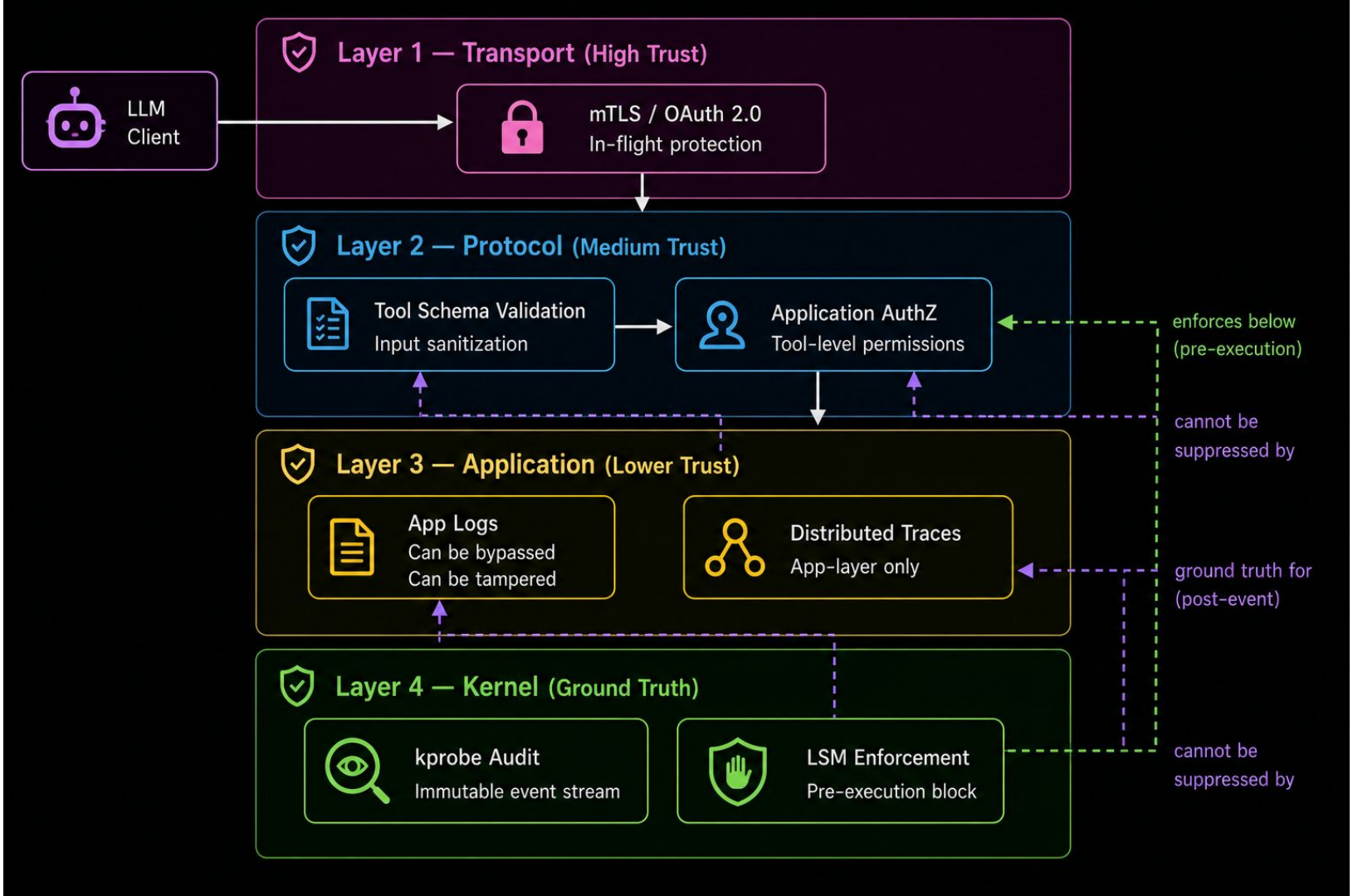
Same tool invocation: two completely different views


audit_record
req_id: req-42
tool: read_customer_data
Kernel Event Timeline
file_open /var/data/customers.json O_RDONLY pid=12400
file_open /etc/passwd O_RDONLY pid=12400
→ not declared in tool schema
exec /bin/sh → curl 198.51.100.5 pid=12401
→ unexpected subprocess spawned
tcp_connect 198.51.100.5:443 pid=12401
→ LSM BLOCKED — no bytes sent
Summary
pid_lineage: 12345 → 12400 → 12401
policy_violations: 2
lsm_blocks: 1
app_log: 'completed successfully'

Layer	What was recorded
Application log	**read_customer_data**: completed successfully
Kernel audit trail	2 policy violations · 1 LSM block · 1 unexpected subprocess


LLM
(injected)
MCP
Server
TOOL
search_docs
handler
eBPF
Audit
LSM
Block
1. tool_call search_docs
(injected query)
2. invoke with
sanitized(?) input
sanitization
misses shell escape
3. subprocess /bin/sh
EXEC event: /bin/sh
(req-99)
4. connect() to attacker.io:443
5. -EPERM (LSM blocked)
BLOCK event:
198.51.100.5:443 (req-99)
6. open /etc/passwd
FILE event:
/etc/passwd O_RDONLY
(req-99)
7. “returned 0 results”
8. {“results”: []}
app log:
“search_docs: 0 results”
audit: 3 anomalies,
1 LSM block, req-99

eBPF doesn't replace Layers 1–3. It enforces a floor they cannot undermine.


LLM Client
Layer 1 — Transport (High Trust)
mTLS / OAuth 2.0
In-flight protection
Layer 2 — Protocol (Medium Trust)
Tool Schema Validation
Input sanitization
Application AuthZ
Tool-level permissions
enforces below (pre-execution)
cannot be suppressed by
Layer 3 — Application (Lower Trust)
App Logs
Can be bypassed
Can be tampered
Distributed Traces
App-layer only
ground truth for (post-event)
Layer 4 — Kernel (Ground Truth)
kprobe Audit
Immutable event stream
LSM Enforcement
Pre-execution block
cannot be suppressed by

Production Reality Check

Question	Answer
Overhead?	Ring buffers + targeted kprobes: measured <1% CPU. Cilium uses this at scale.
Kernel version?	LSM+BPF: kernel ≥ 5.7. kprobe-only audit: kernel ≥ 4.9.
Privilege needed?	`CAP_BPF` (Linux 5.8+): no full root required
Kubernetes?	DaemonSet + `CAP_BPF` securityContext. Works today.
Multi-tenant MCP?	Namespace-scoped BPF maps. Tenant isolation in policy_map.
Compliance (FIPS)?	Ring buffer = append-only from kernel. Satisfies immutability requirements.
What about eBPF exploits?	eBPF verifier, signed programs, and BTF. Narrower attack surface than privileged agents.

Key Takeaways

1. Application logs describe intent. Kernel events describe reality. eBPF is the only layer that cannot be suppressed by application bugs or prompt injections.

2. Process lineage tracking bridges PIDs to request IDs. One BPF map write per MCP request gives you full syscall attribution across the entire process tree for free.

3. LSM hooks turn audit into prevention. You can block unexpected network destinations and file paths *before they execute*, not just log them after.

4. Zero changes to your MCP server. Everything runs in kernel context, invisible to the application.

5. The threat model is clear. Prompt injection, supply-chain compromise, and logic bugs all surface in the same kernel event stream.

Resources

- cilium/ebpf (Go): github.com/cilium/ebpf
- bpftrace: bpftrace.org
- Linux LSM+BPF docs: kernel.org/doc/html/latest/bpf/prog_lsm
- MCPSpy: github.com/mcpspy/mcpspy
- ebpf-mcp server: github.com/ebpf-mcp

Thank You

- Harini Anand
- Connect: linkedin.com/in/harinianand