MetArmor: Tracking Files across the operating system using eBPF

Carl El Khoury

Meta

MetArmor Problem Statement

01 - Problem statement

02 - Metarmor: Our custom solution

03 - What we support

04 - File Tracking Across the File system

05 - Challenges

Meta

The Problem

In Meta's world, where billions of people connect every day, the risk of cyber threats is very high. Without a strong Intrusion Detection and Prevention System (IDPS), Meta's network and hosts could be attacked, leading to unauthorized access, data leaks, and system breakdowns.

This is of increasing concern with new AI threats constantly involving and new risks around AI data and model theft.

The main problem is detecting and stopping advanced cyber threats in real time, considering the size and complexity of Meta's operations with low performance hit.

Meta

Our Solution: Metarmor

The overarching goal is to build a modular, config-driven platform that focuses on the following three pillars:

- **Signal Collection:** Collect security signal to retrospectively identify threats and vulnerabilities
- **Prevention:** Emit real time security alerts and/or automatically action events
- **Response:** Perform automated or manually response to ensure

With an additional pillar planned:

- **Integrity:** Ensure the server we are running on isn't compromised

Why?

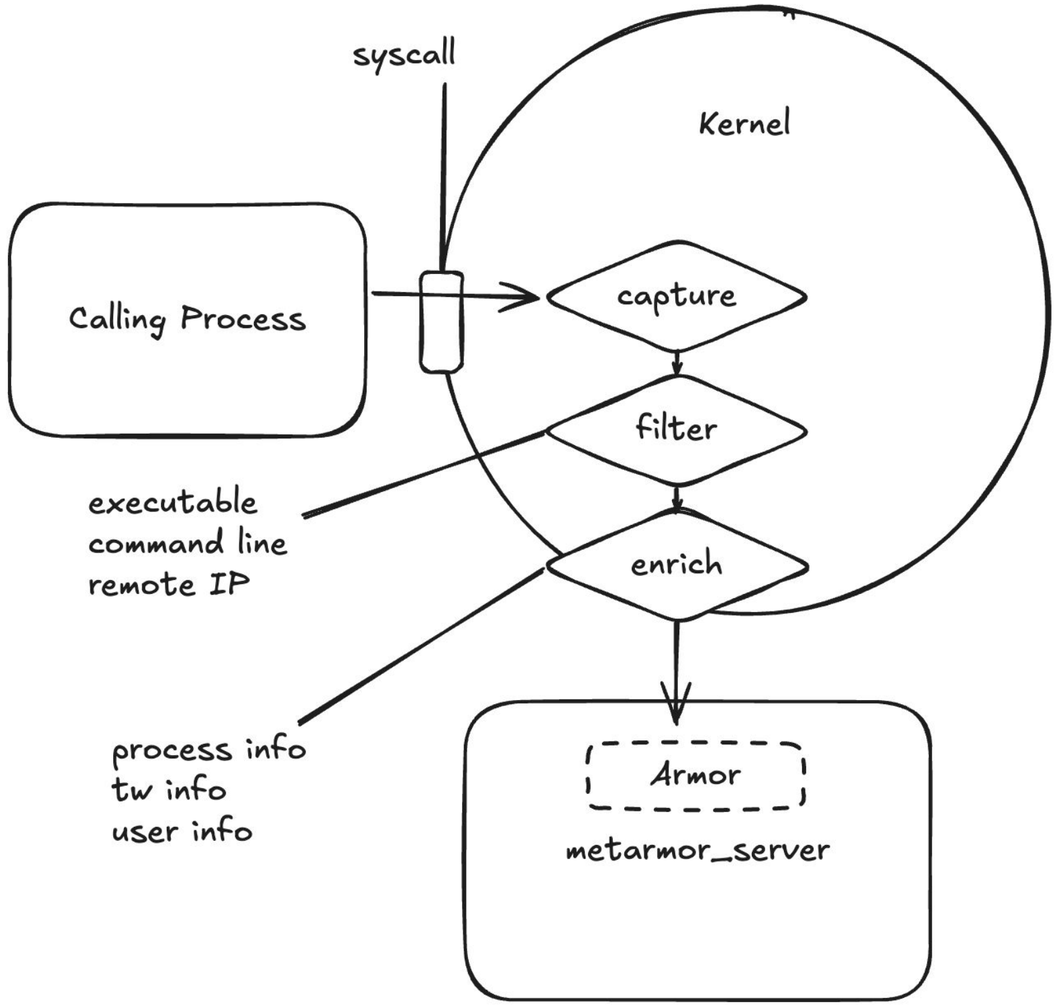
- eBPF allows fast and safe kernel monitoring and is heavily leveraged
- No reliance on scanning, all detections are real time

We hope to make **Metarmor open source** by 2027

Meta

eBPF Events

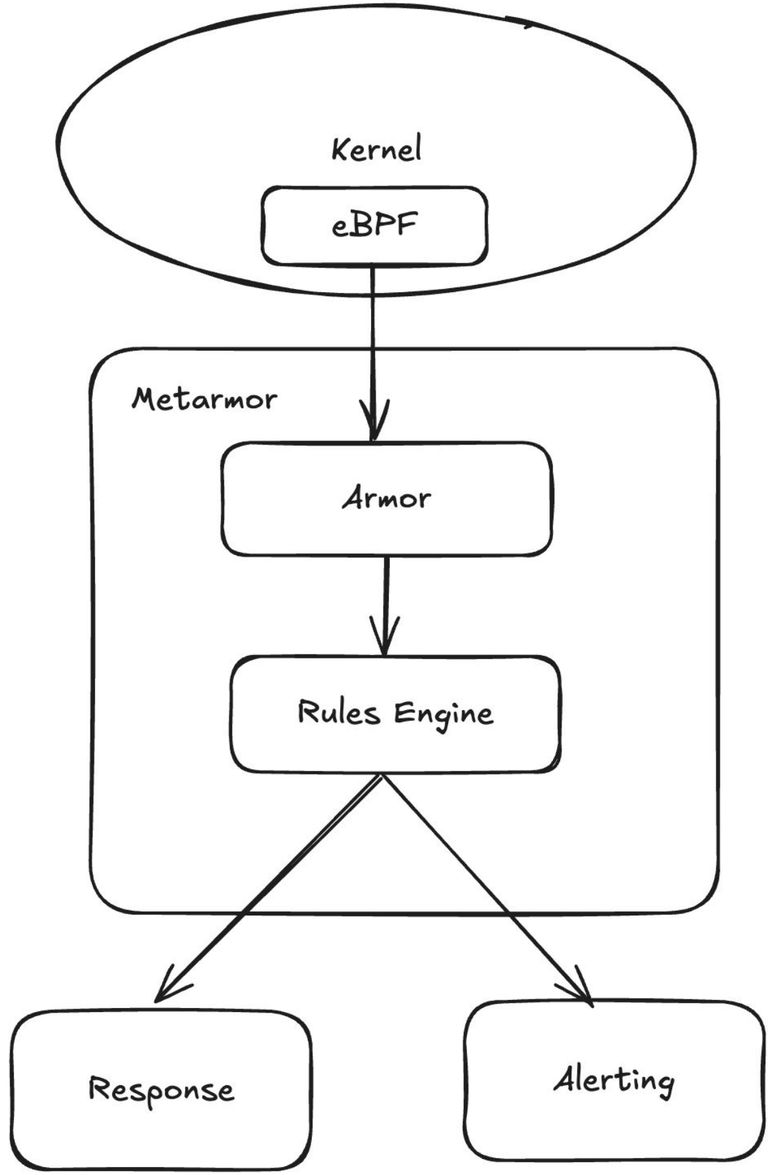
- Userspace Armor manages eBPF programs
- eBPF programs filter events based on criteria
  - IP address
  - Process executable path
  - Command line arguments
- If an event survives filtering, it is enriched
  - Process and parent process information
    - Executable
    - Command line
    - User (based on SSH as well as uid)
  - Syscall details (depending on syscall)
  - Meta specific information



Meta

Data flow graph

- Events are eBPF triggered
- Most events are filtered out based on heuristics
- Events are enriched
- Security engineers define checks that can triggered an alert or a response

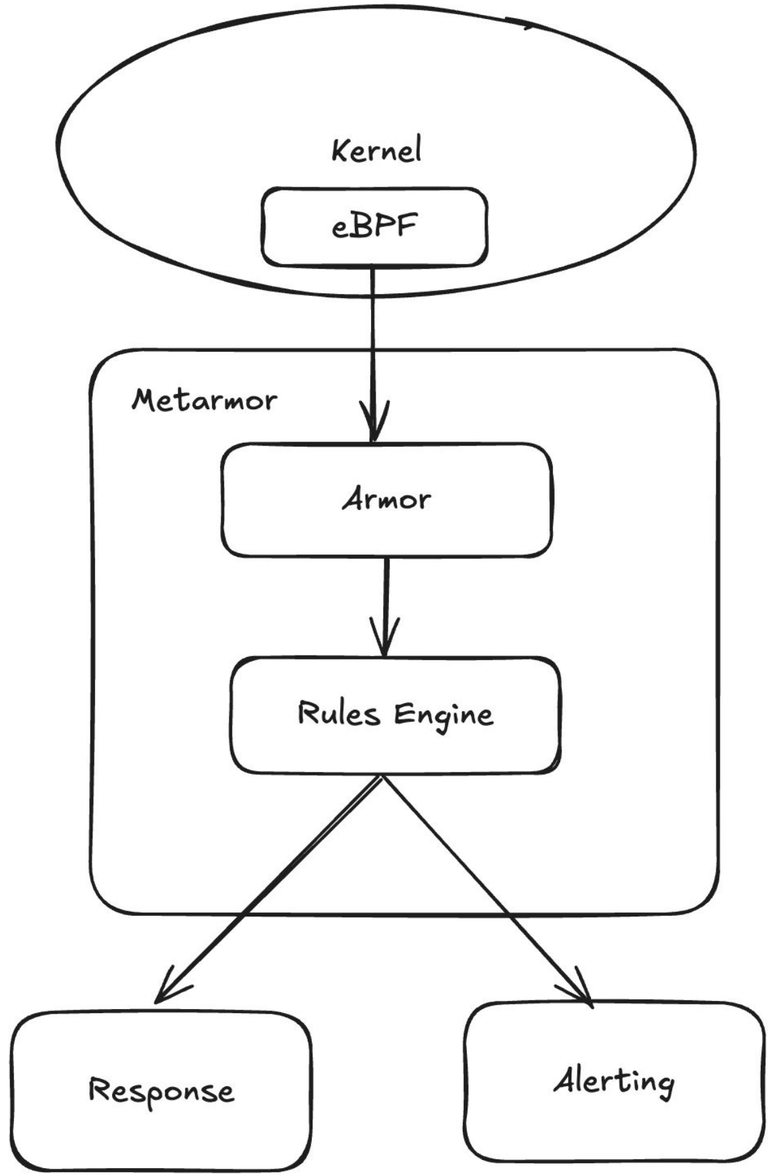


Meta

Data flow graph: Rules engine

- It’s own language that is interpreted per rule
- Contains the field name to check and a number of operations
- Logic operators to chain multiple conditions together
- Define what should be the result of that rule
- Allows us to define **rules fast without code changes**

- We actively try to move as many checks to eBPF for better performance. We would like to create a ebpf-based rule engine
- Log a subset of the data for retrospection and cross-machine correlation



Meta

Signal supported

Execution

- execve()
- bash commands
- python scripts

Networking

- bind()
- connect()
- accept()
- sendmsg()
- recvmsg()
- downloaded files
- TCP bytes transferred
- UDP bytes transferred
- DNS

Containers

- cgroup creation
- namespace creation

Filesystem

- open()
- setuid/setgid bits set
- File Lineage

Hardware

- USB
- Storage
- PCI
- Memory
- Intrusion detection

Kernel

- kernel modules
- bpf programs

Memory

- ptrace()
- sys_vm_readv()/writev()
- memfd_create()

Other

- mount()
- pivot_root()

Meta

Reponses supported

- Kill a process by pid or by binary path. A binary path can also be prevented to starting in the future using an lsm hook
- Upload a file for analysis
- Quarantine a file: Prevent a file from being opened (not only a binary)
- Information gathering: Capture more information about a process or system services
- Send an alert to request human attention
- Block Network access of a process using lsm hooks

Meta

Example: How are this signal is used

A few days ago, a major malware campaign known as Sha1-Hulud 2.0 targeted the npm ecosystem. This was a large-scale, rapidly spreading supply chain attack that compromised over 27,000 npm packages, with the number of affected repositories growing by about 1,000 every 30 minutes at its peak.

Leveraging the process execution signal, Metarmor initially enabled us to monitor that the malware hasn’t spread to our infrastructure. After we isolated the malware, an update to the rules allowed us to block this malware from ever executing on Meta’s servers by matching on the process’ spawning behavior

Pushing that new security rule to all hosts took 15 minutes.

Meta

Track file uploads: Real Time Malware Protection

- connect() hook used to catch connections to our internal proxy

  ```
  SEC("fentry/security_socket_connect")
  ```

- Files created by the process are assumed to be “downloaded” and are filtered with regex to remove unwanted files

  ```
  SEC("fentry/security_file_free")
  SEC("lsm/file_free_security")
  SEC("ksyscall/close")
  ```

- Downloaded files uploaded for scanning and possibly retention

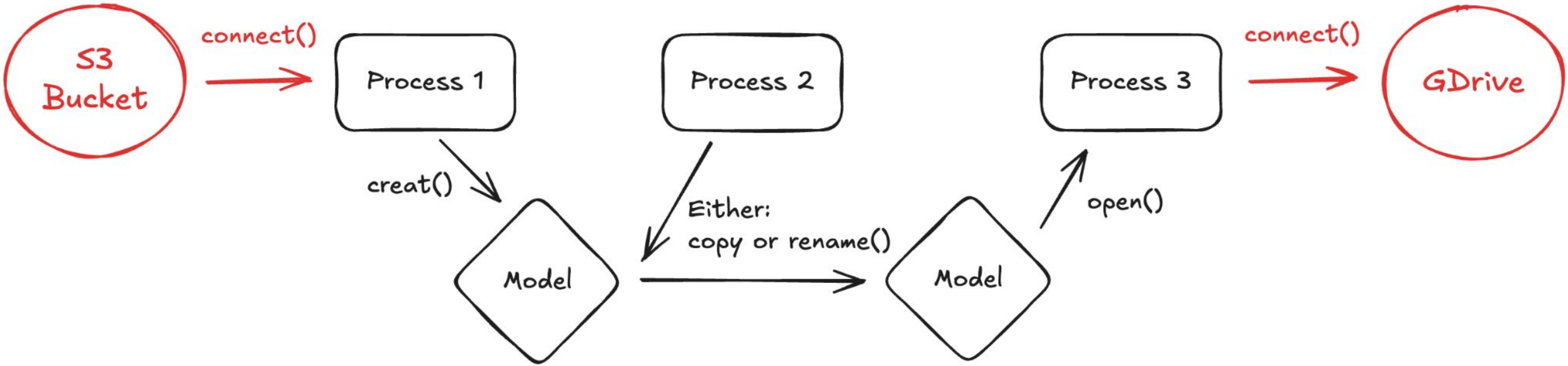
We use different hooks to handle different kernel versions and method being in-lined by the compiler

We were able to flag/block **1000s of potential malware** through this

Meta

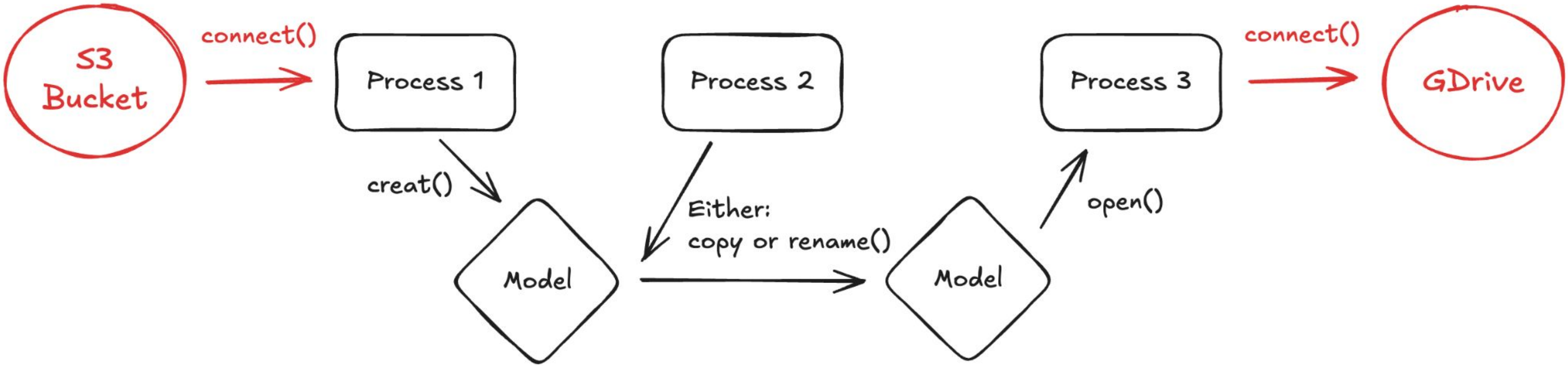
File Lineage:

- Tracking files across the whole file system
- Tag a file by path/IP of origin
- Track it’s movements (copy, move, compression)
- Tag files that could get derived from it (i.e Process x open files and writes a bunch of files)
- Track if the file gets uploaded



File Lineage: How do we do that

- Moves/Rename: Hook into the move syscall
- Copy/Compression/Derived file: Hook into file open and track all the files this process writes afterwards. Compression libraries are market with ‘Compression’ flag
- Deletion: Hook into File close to detect deletions. This allows us to decrease the memory burden we incur
- Upload: Hook into both connect and file open to allow us to detect uploading



File Lineage: How do we do that (Create)

```
FUNCTION detect_create(file, task, proc):
  IF task or parent is metarmor process: RETURN 0
  IF file is not valid or not newly created: RETURN 0
  IF file is not from allowed file systems: RETURN 0
  file_name = get file name
  IF file_name is empty or extension not allowed: RETURN 0
  event = reserve event
  IF event not available: RETURN 0
  Fill event fields (action, pid, uid/gid, file info, proc, parent pid, filename, ssh info)
  Submit event
  RETURN 0
```

Meta

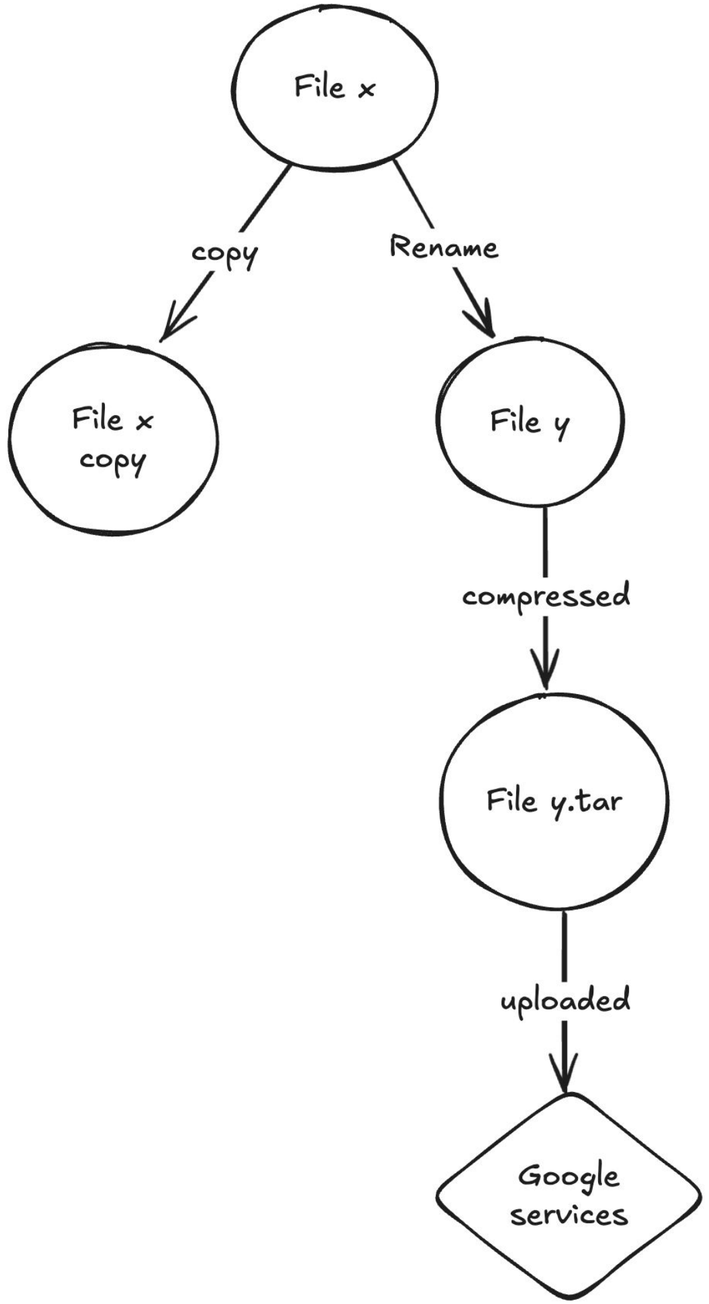
File Lineage: How do we do that (Delete)

```
FUNCTION security_path_unlink(dir, dentry):
  task = get current task
  parent = task's real parent
  ns_pid = get namespace pid for task
  ns_ppid = get namespace pid for parent
  IF task or parent is metarmor process: RETURN 0
  proc = lookup process info for task
  file_name = get file name buffer from map
  IF file_name not found: RETURN 0
  d_name = get name from dentry
  copy d_name into file_name buffer
  IF file_name is empty or extension not allowed: RETURN 0
  event = reserve event
  IF event not available: RETURN 0
  Fill event fields (action=FILE_DELETE, pid, uid/gid, file info, proc, parent pid, filename, ssh info)
  Submit event
  RETURN 0
```

Meta

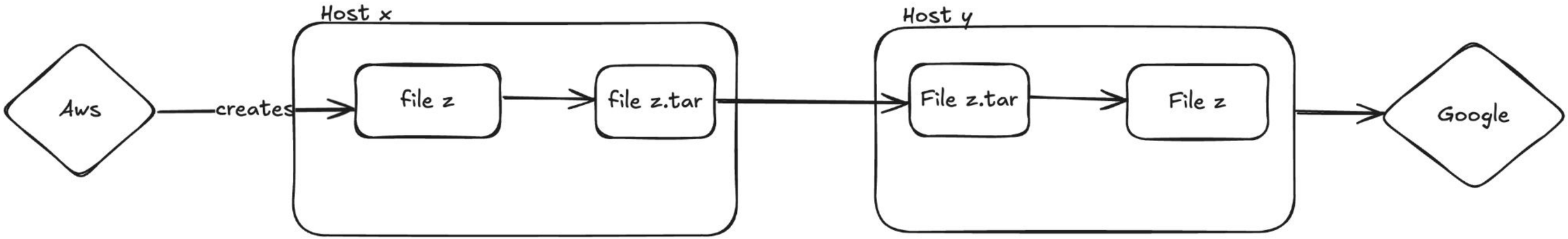
File Lineage: Event tree

- Userspace keeps track of every file movement (creation, copy, compression) and creates an event if it’s uploaded anywhere
- Use SQLite database to keep track for movements to avoid memory problems
- SQLite saves the Directed Acyclic Graph to disk and traverses it through recursive queries
- Everything is bounded by size



File Lineage: cross-host correlation

- The data we presently collect can allow us to track files across different servers
- Needs to be traced by hand currently



Meta

What we want to improve on:

- Many of our filtering for Metarmor and File Lineage rely on File path which can be finicky. We should move to inode eventually
- Memory and CPU limits what we can do. We can’t track every file
- We want to move towards hashes to detect files of interest
- Currently we lose state between restarts, something we would like to address when we gain more confidence in the system
- How to decide which file gets tagged? IP and file path is the only things, we want to add domain if the file is downloaded and hashes

Meta