Cyberster

INTERNSHIP REPORT

Week 3: Suricata IDS Integration with Wazuh

Submitted to: Cyberster Internship Program

Intern Name: Abdul Muqeet Tabraiz

Roll Number: CSI-B1-353

Table of Contents

Page Left Blank Intentionally

Cyberster Blue Team Internship — Week 3 Days 15–16

Suricata Installation and Configuration

Objective

The objective of Days 15 and 16 was to install and configure Suricata, an open-source network intrusion detection system (IDS), on the Kali Linux VM. This included identifying the correct network interface for packet capture, configuring the af-packet capture method in suricata.yaml, updating the Emerging Threats community ruleset, starting and enabling the Suricata service, and verifying detection by observing live alerts in both the fast.log compact alert log and the eve.json structured JSON event log.

Tools Used

- Kali Linux VM — primary machine on which Suricata was installed and configured
- Suricata 8.0.3 — network IDS deployed in IDS mode on interface eth1
- /etc/suricata/suricata.yaml — main configuration file edited to set network interface and rule paths
- suricata-update — tool used to download and update the Emerging Threats Open ruleset
- /var/log/suricata/fast.log — compact one-line-per-alert log used to verify detection
- /var/log/suricata/eve.json — full structured JSON event log examined to understand event schema

Procedure

Step 1: Update Kali Linux System Packages

Before installing Suricata, the Kali Linux system was updated to ensure all packages were current and no dependency conflicts would arise during installation. The apt package manager was used to run a combined update and full upgrade.

```
sudo apt update && sudo apt upgrade -y
```

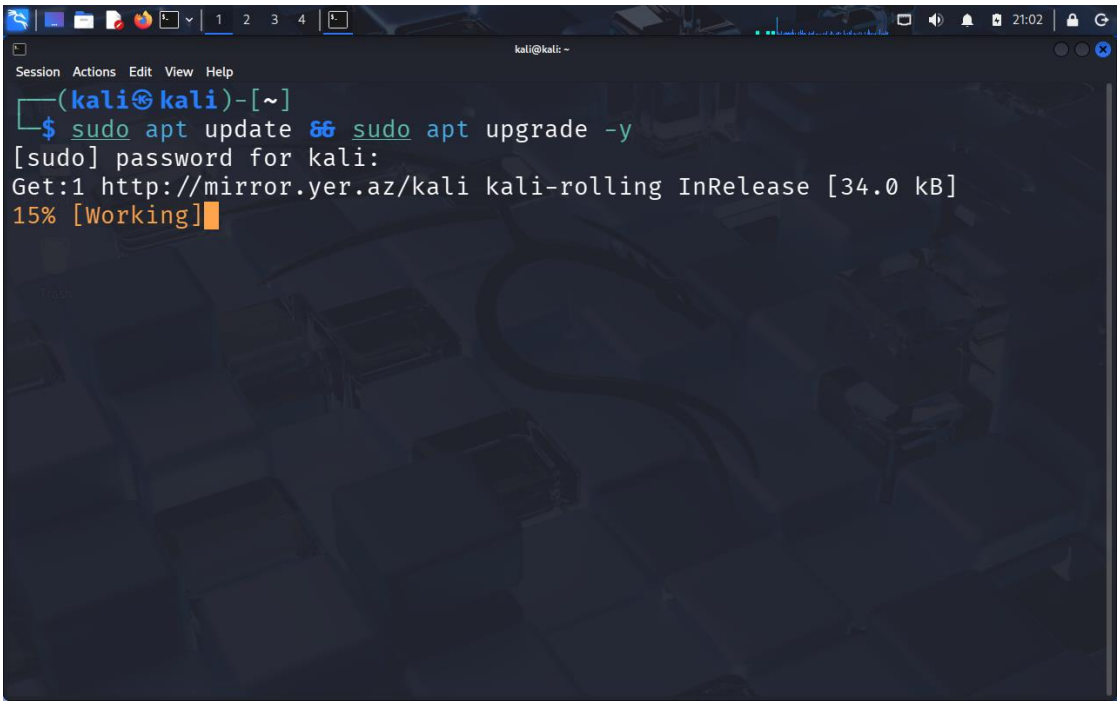

Fig 15.1 Kali Linux — sudo apt update && sudo apt upgrade -y running, fetching InRelease from kali-rolling mirror

The upgrade completed successfully, setting up kernel and desktop packages including kali-linux-default 2026.1.8 and kali-desktop-xfce 2026.1.8. The initramfs was regenerated for kernel 6.18.12+kali-amd64, confirming the system was fully up to date before proceeding.

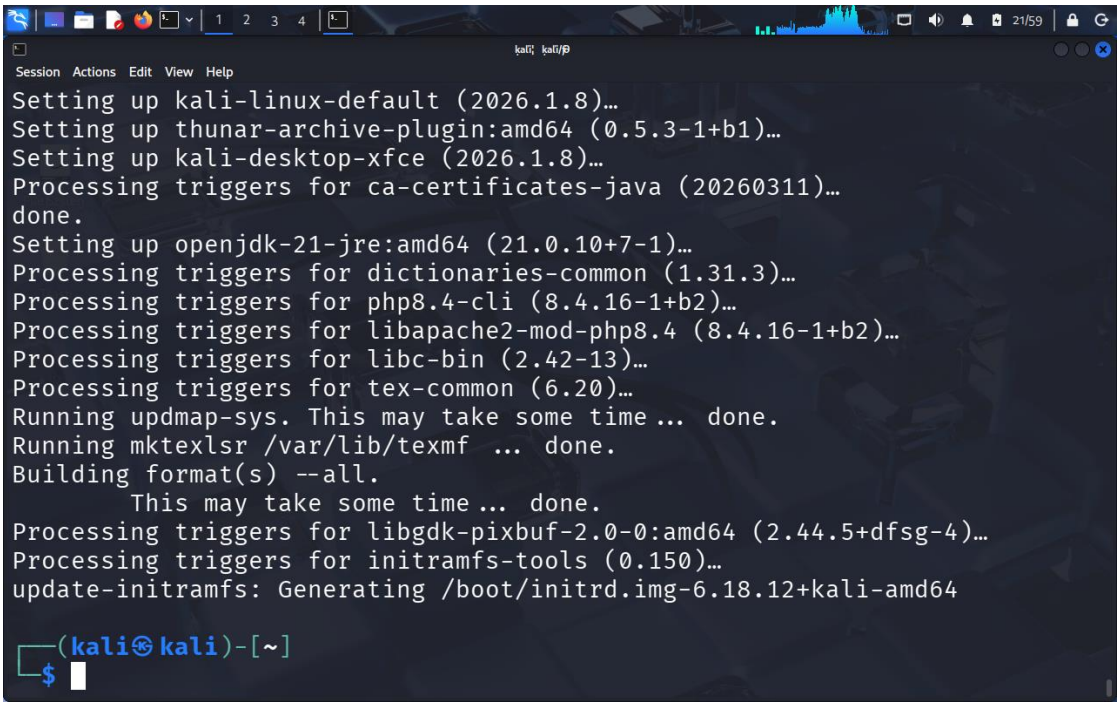

Fig 15.2 Kali Linux — apt upgrade completed, final package triggers processed and initramfs regenerated

Step 2: Install Suricata

Suricata was installed from the Kali repositories using apt. The -y flag was passed to suppress confirmation prompts and allow unattended installation.

```
sudo apt install suricata -y
```

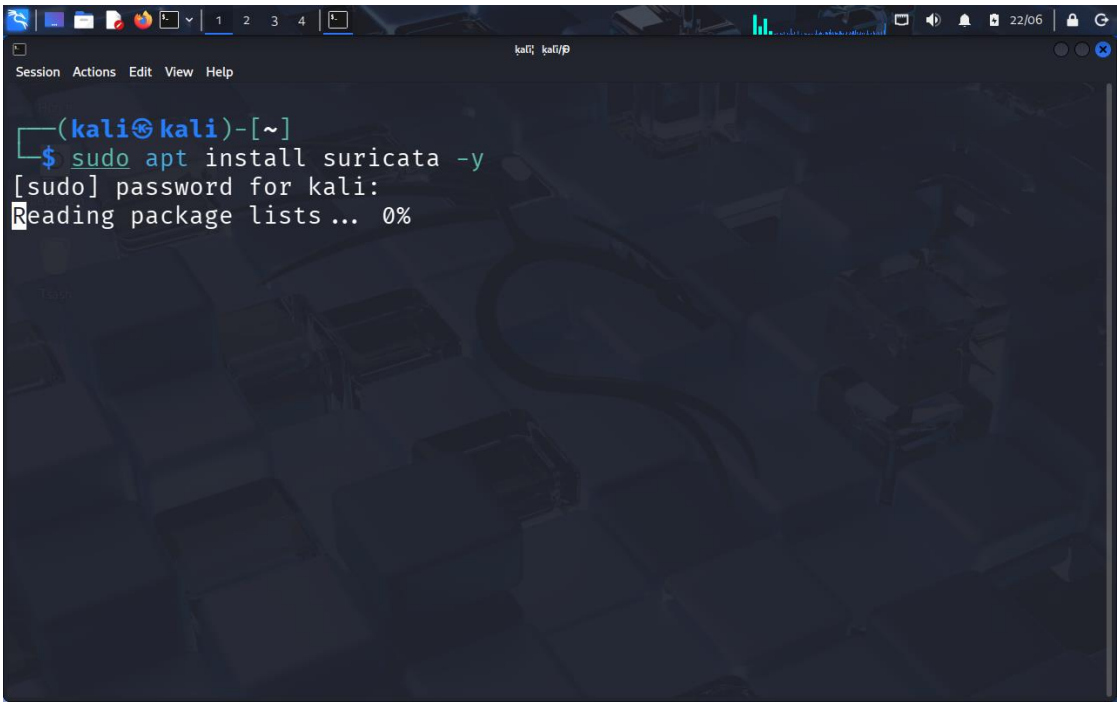

Fig 15.3 Kali Linux — sudo apt install suricata -y initiated, reading package lists

After installation, the Suricata version was confirmed by running the version flag. The output reported Suricata 8.0.3 and also displayed the full usage guide, confirming the binary was installed correctly in /usr/bin/suricata.

```
suricata --version
```

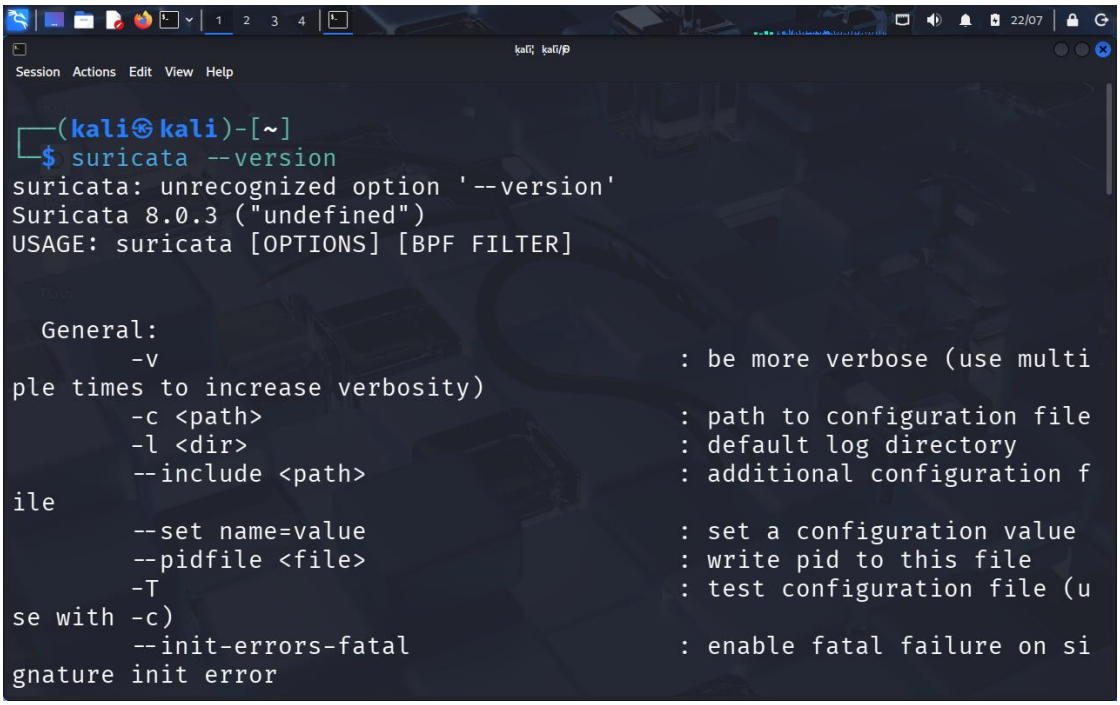

Fig 15.4 Kali Linux — suricata --version output confirming Suricata 8.0.3 ("undefined") installed successfully

Step 3: Identify the Network Interface

The correct network interface for packet capture was identified by running ip a inside the Kali VM. Two interfaces were present: eth0 (192.168.195.128/24, NAT adapter) and eth1 (192.168.14.139/24, Host-Only adapter). Since the lab environment uses the 192.168.14.0/24 Host-Only network for inter-VM communication, eth1 was selected as the monitoring interface. Configuring the wrong interface would cause Suricata to run without errors but capture no relevant lab traffic.

```
ip a
```

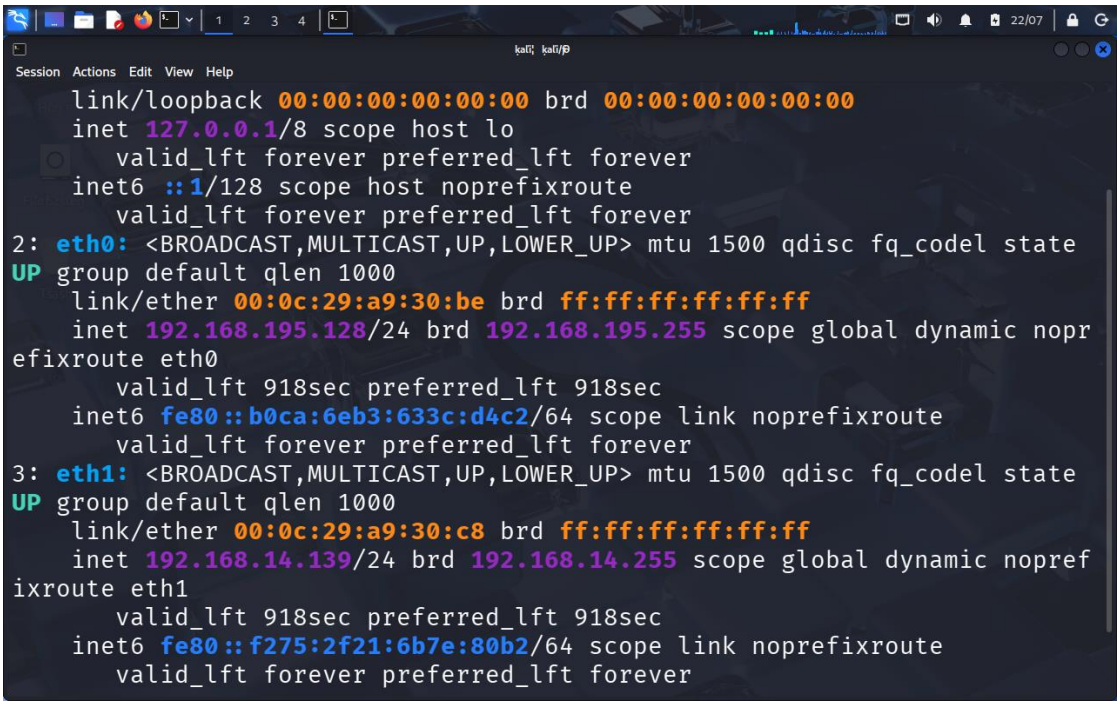

Fig 15.5 Kali Linux — ip a output showing eth0 (192.168.195.128) and eth1 (192.168.14.139) — eth1 selected for Suricata

Step 4: Configure suricata.yaml

The main Suricata configuration file was opened in nano for editing. Two sections required changes: the HOME_NET address group and the af-packet capture block.

```
sudo nano /etc/suricata/suricata.yaml
```

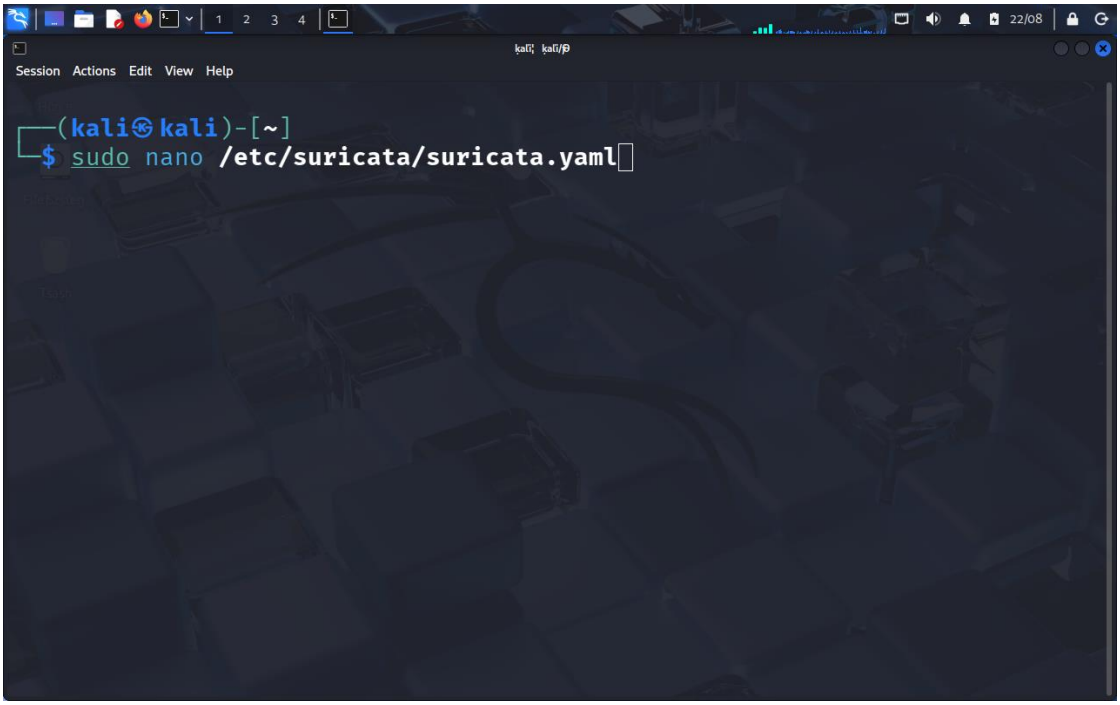

Fig 15.6 Kali Linux — opening /etc/suricata/suricata.yaml in nano for configuration

The HOME_NET variable was reviewed. The default value covering RFC 1918 private ranges [192.168.0.0/16,10.0.0.0/8,172.16.0.0/12] was retained so that alerts correctly classify traffic from within the lab network as originating from the home network. The more specific commented-out entry for 192.168.14.0/24 was noted but not activated since the default broad range encompasses the lab subnet.

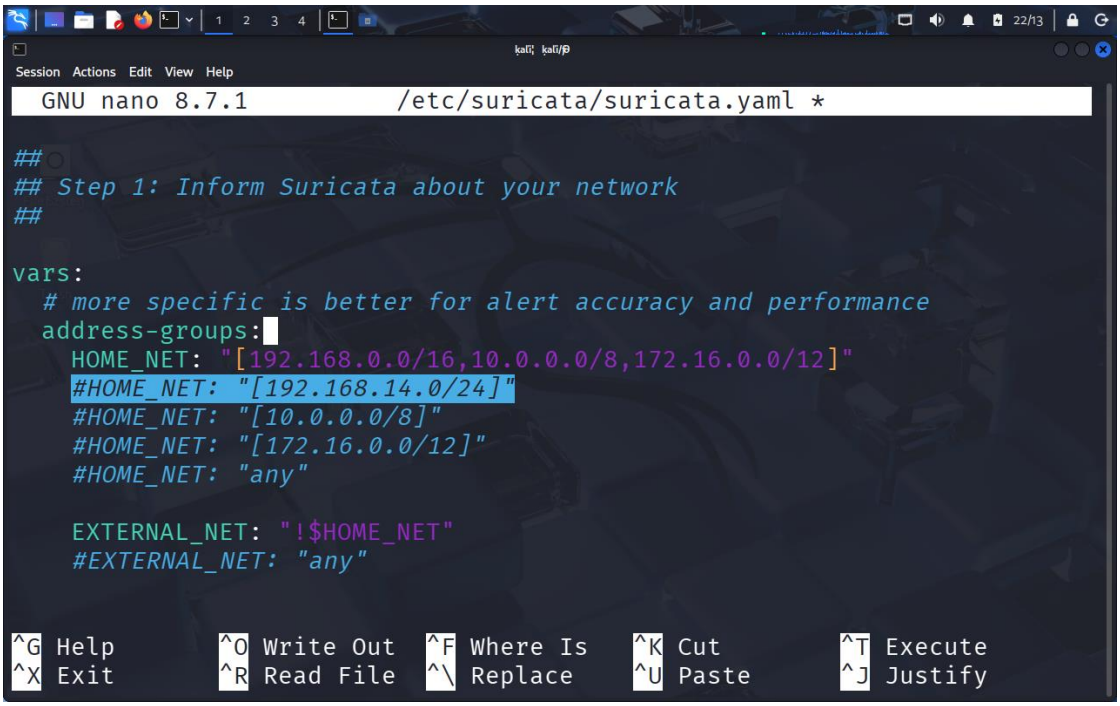

Fig 15.7 Kali Linux — suricata.yaml vars section — HOME_NET set to RFC 1918 ranges, specific lab subnet 192.168.14.0/24 visible as commented alternative

The af-packet section was located and the interface value was set to eth1, matching the Host-Only adapter identified in the previous step. The af-packet mechanism instructs Suricata to use the Linux kernel's high-performance AF_PACKET socket for zero-copy packet capture. An incorrect interface name at this point would cause Suricata to run silently without generating any alerts regardless of network activity.

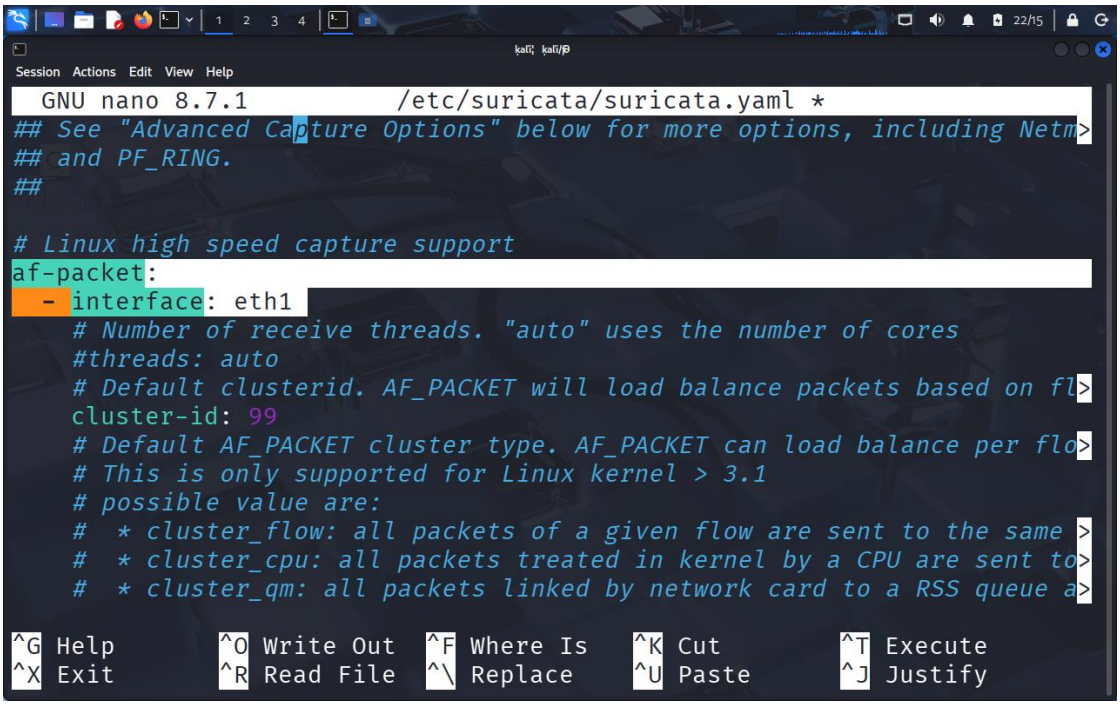

Fig 15.8 Kali Linux — suricata.yaml af-packet section — interface: eth1 set, cluster-id: 99 retained

The rule-files section was reviewed to confirm the default rule path /var/lib/suricata/rules and the suricata.rules entry were in place before saving. This would later be extended to include the custom local.rules file on Day 17.

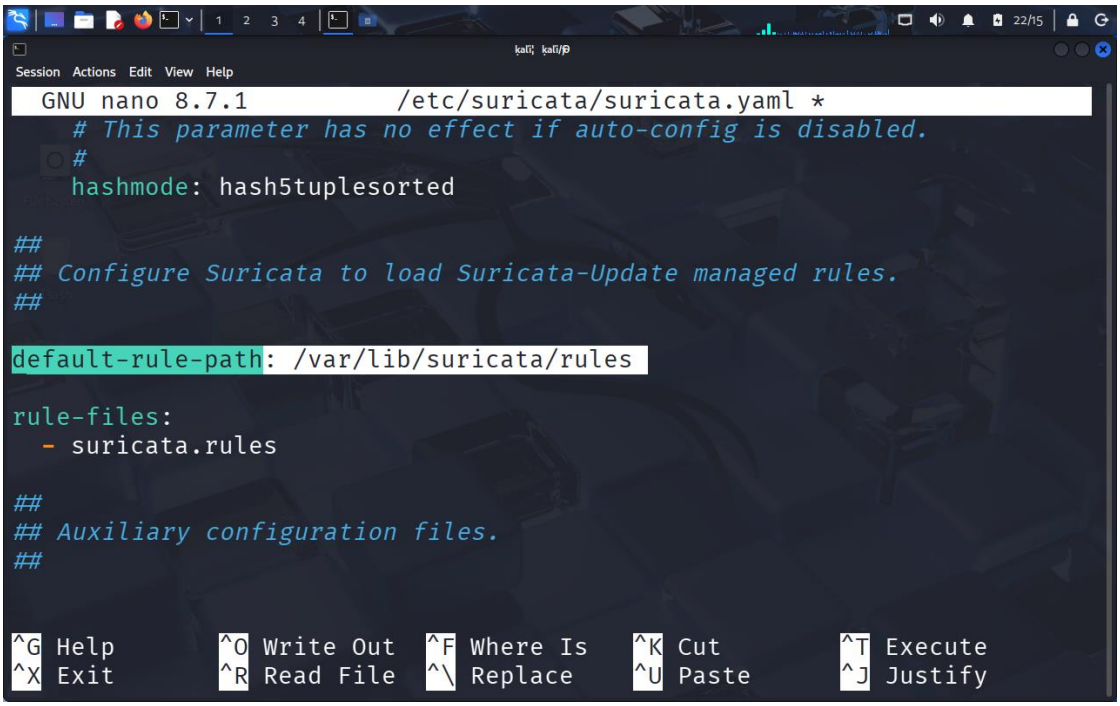

Fig 15.9 Kali Linux — suricata.yaml rule-files section — default-rule-path: /var/lib/suricata/rules and suricata.rules entry visible

Step 5: Update the Emerging Threats Ruleset

Before starting the service, the suricata-update tool was run to download the latest Emerging Threats Open ruleset. This community-maintained ruleset provides thousands of signatures covering known malware, exploits, and suspicious traffic patterns. Running the update at this stage ensures the initial detection capability is current from the moment Suricata starts.

```
sudo suricata-update
```

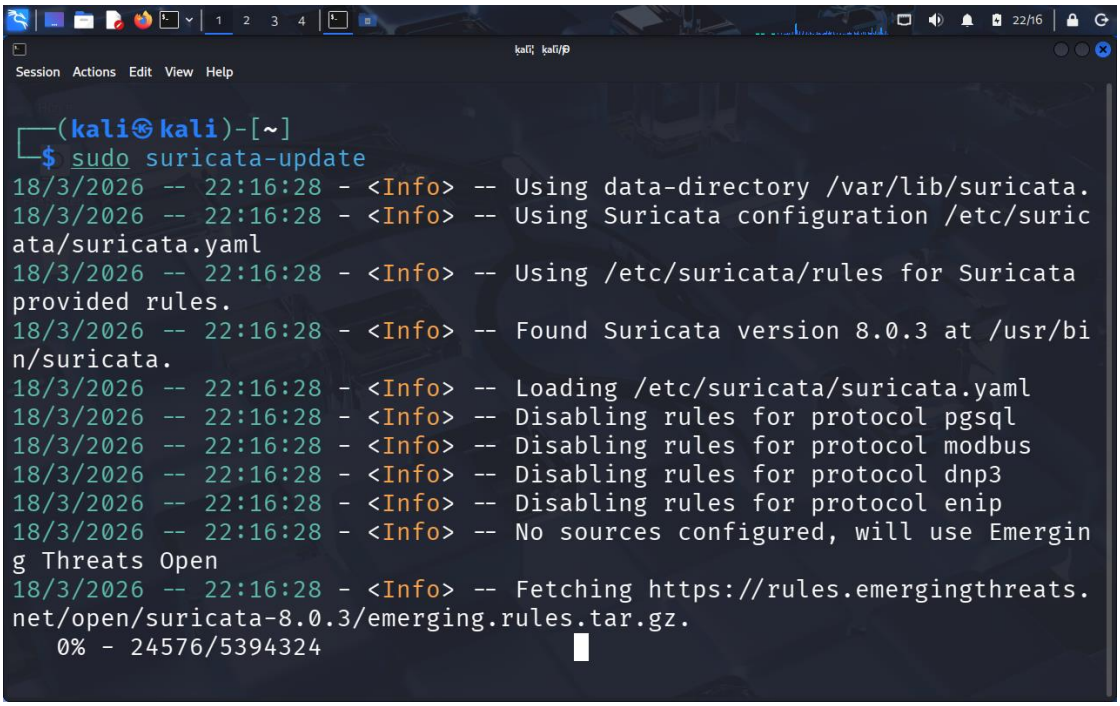

Fig 15.10 Kali Linux — suricata-update running — Suricata 8.0.3 detected, fetching emerging.rules.tar.gz from rules.emergingthreats.net

The available rule sources were also listed to understand what additional feeds could be enabled in production environments.

```
sudo suricata-update list-sources
```

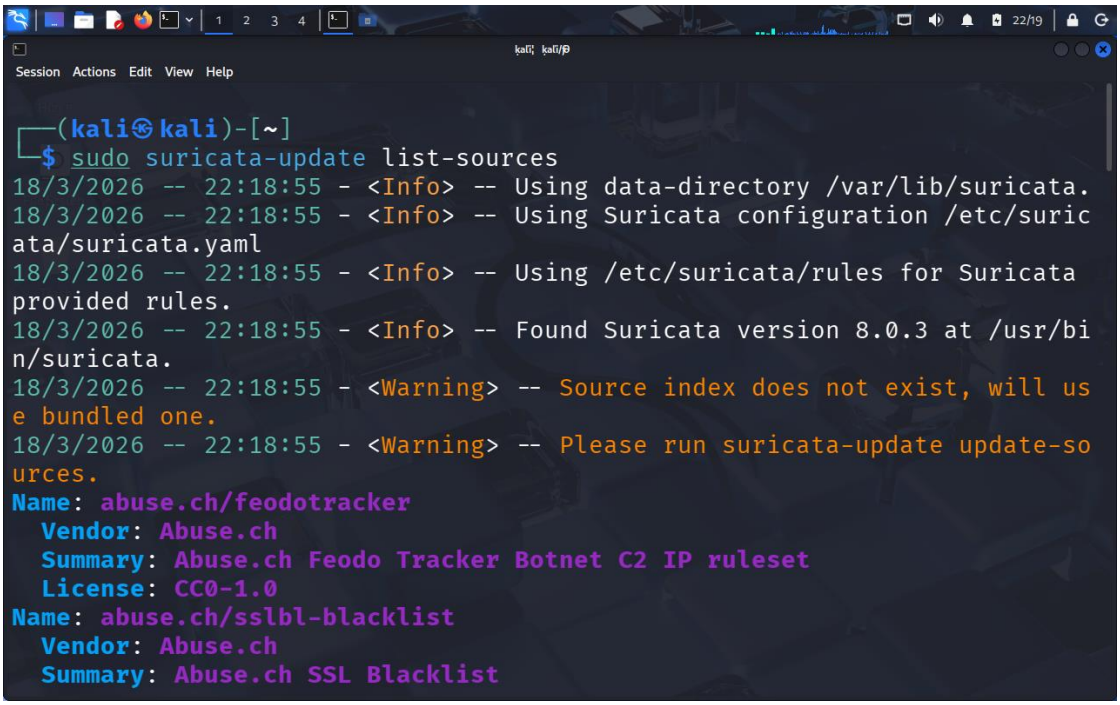

Fig 15.11 Kali Linux — suricata-update list-sources output showing available feeds including abuse.ch/feodotracker and abuse.ch/sslbl-blacklist

Step 6: Enable, Start, and Verify the Suricata Service

Suricata was enabled to start automatically at boot and then started immediately using systemctl. The service status was checked to confirm it was active and running.

```
sudo systemctl enable suricata
sudo systemctl start suricata
sudo systemctl status suricata
```

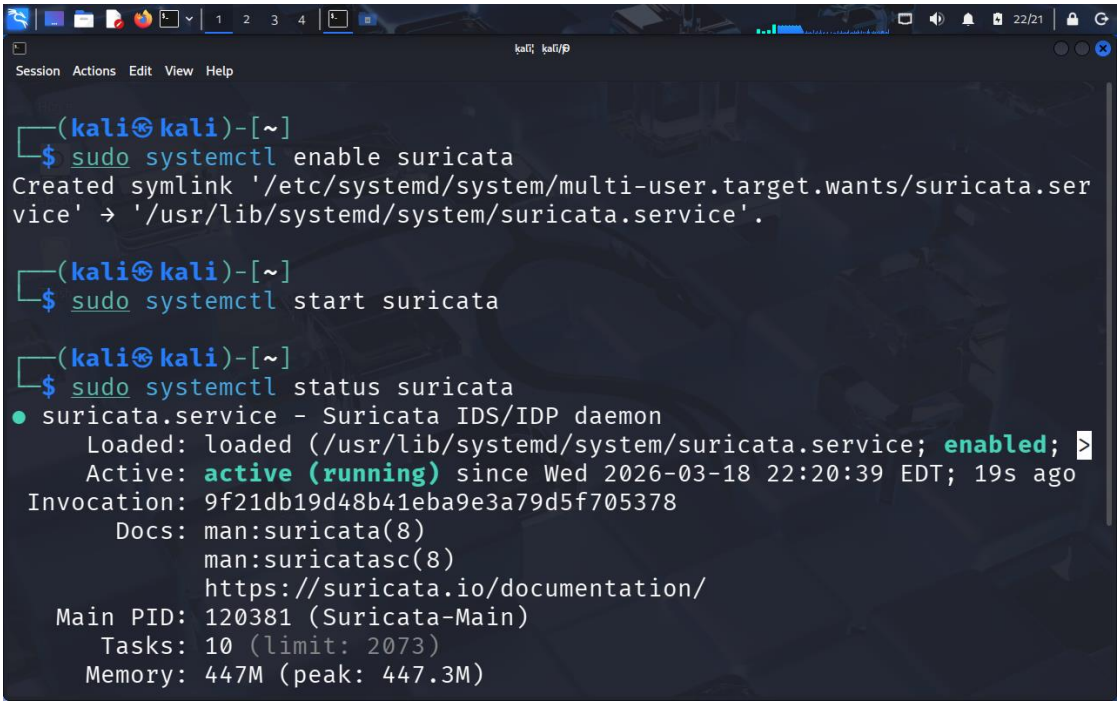

Fig 15.12 Kali Linux — systemctl enable creates symlink; systemctl status shows suricata.service active (running) since 2026-03-18 22:20:39, PID 120381

The Suricata internal log was examined to confirm all rule files were loaded without errors and that the af-packet worker threads were created on eth1.

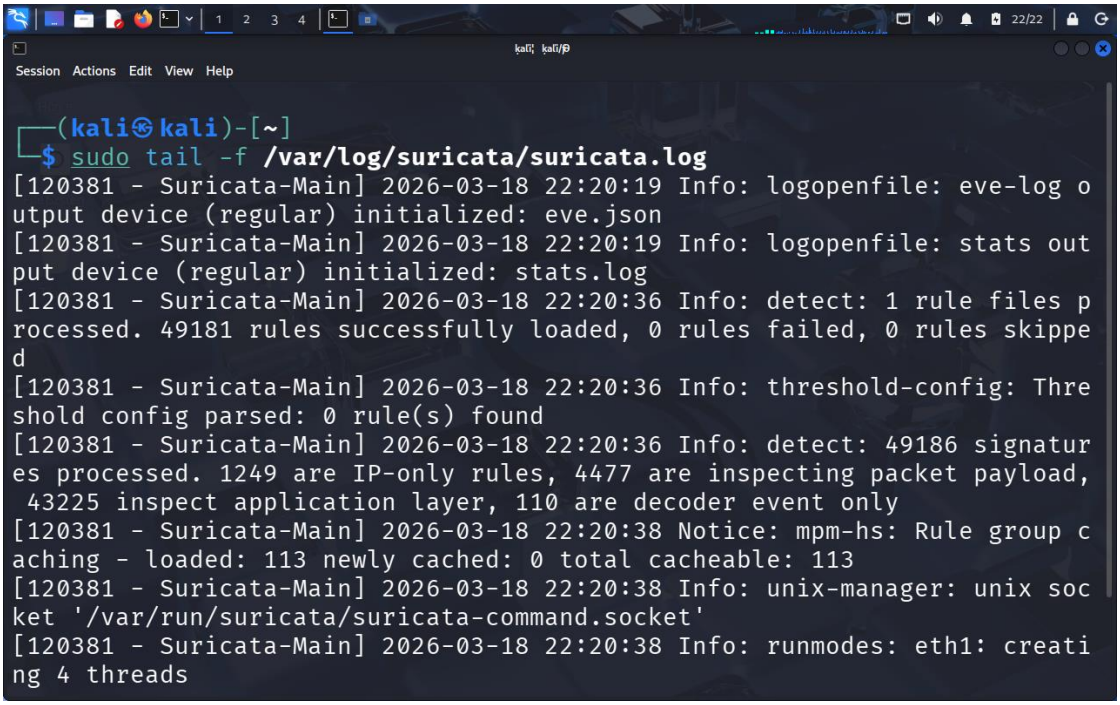

Fig 15.13 Kali Linux — suricata.log tail — 49181 rules loaded, 49186 signatures processed, eth1 threads created, engine started

Step 7: Verify Detection with fast.log and eve.json

Network activity was generated from the Kali VM to trigger default ruleset alerts. A ping to the Ubuntu Server, an nmap SYN scan, and DNS and HTTP requests were made. The fast.log file was tailed in a second terminal to observe alerts in real time.

```
sudo tail -f /var/log/suricata/fast.log
```

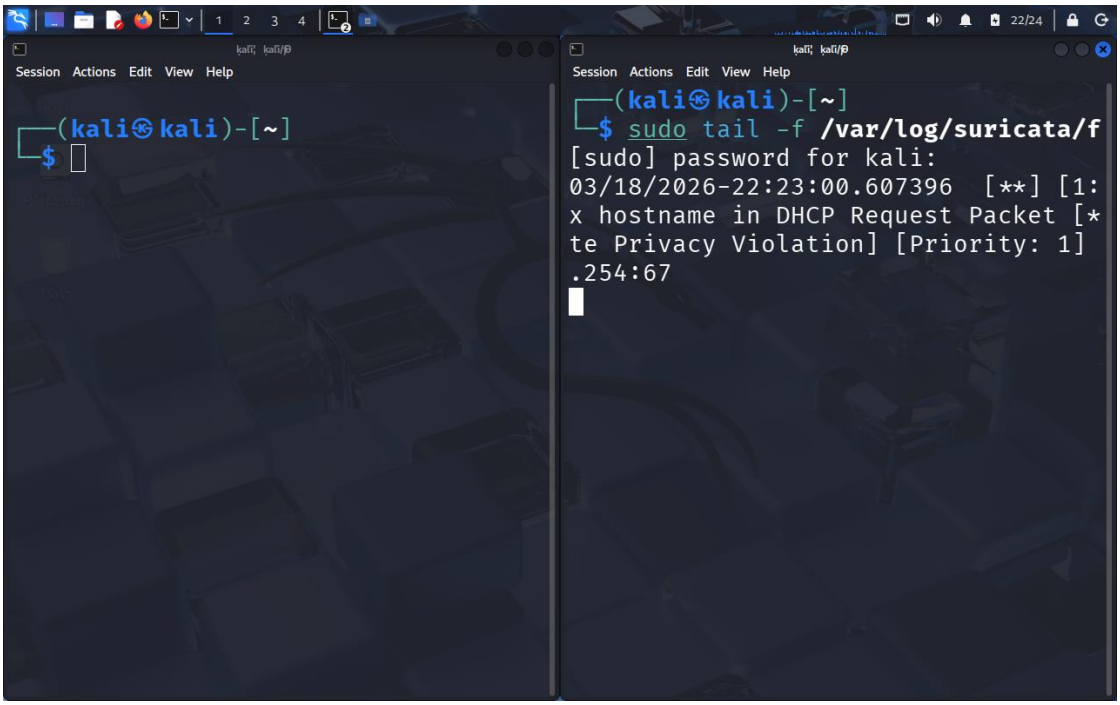

Fig 15.14 Kali Linux — fast.log first alert appearing: ET INFO Possible Kali Linux hostname in DHCP Request Packet at 22:23:00, confirming Suricata is detecting traffic on eth1

After generating ping, nmap, curl, and nslookup traffic, multiple alerts appeared in fast.log covering SURICATA STREAM events, GPL ATTACK_RESPONSE id check returned root, and repeated DHCP alerts. This confirmed the af-packet interface configuration was correct and Suricata was actively inspecting lab traffic.

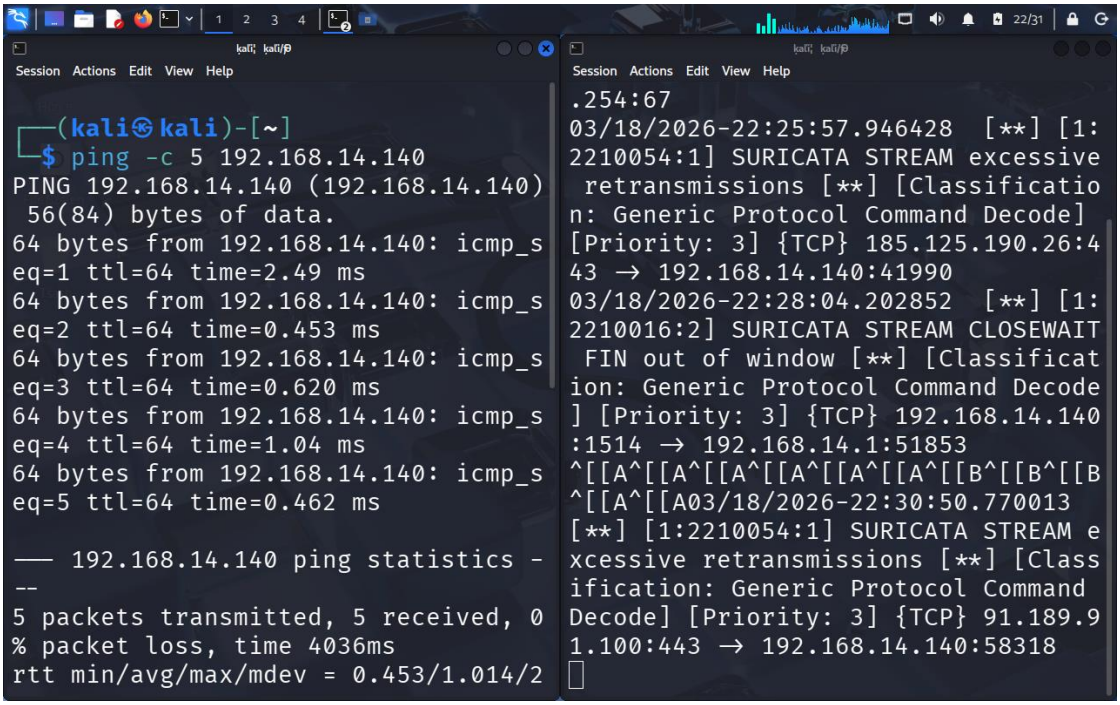

Fig 15.15 Kali Linux — ping -c 5 to Ubuntu (left) alongside fast.log (right) showing SURICATA STREAM retransmission and CLOSEWAIT FIN alerts

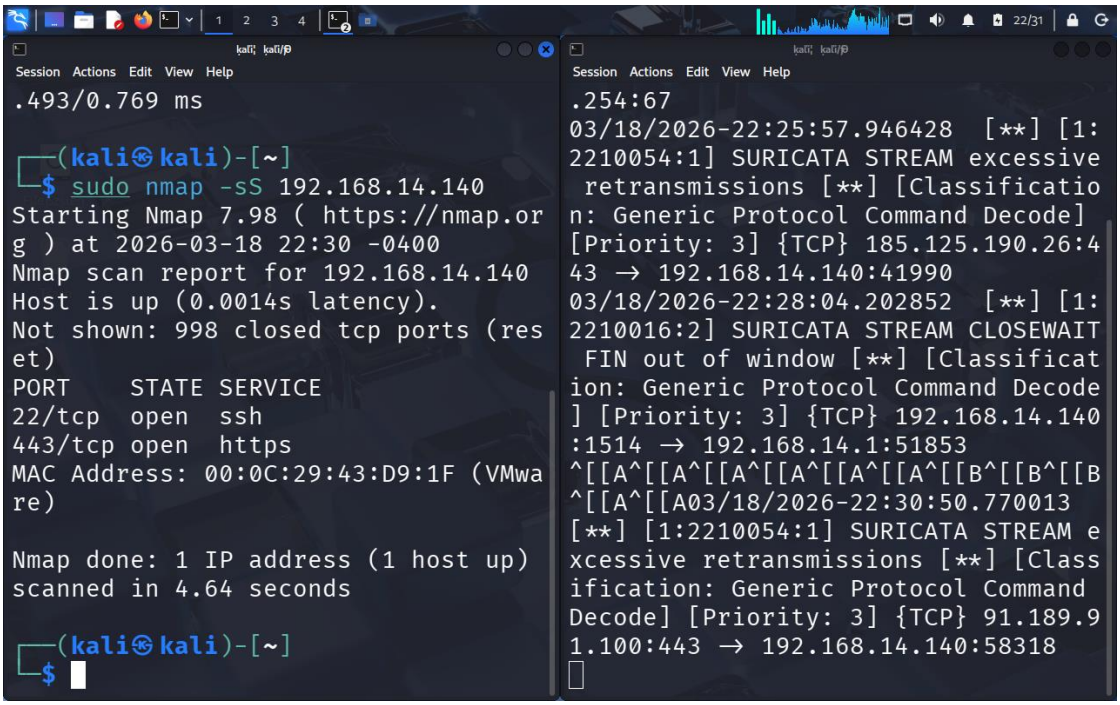

Fig 15.16 Kali Linux — nmap -sS scan of 192.168.14.140 (left) alongside fast.log showing GPL ATTACK_RESPONSE id check returned root at 22:31:53

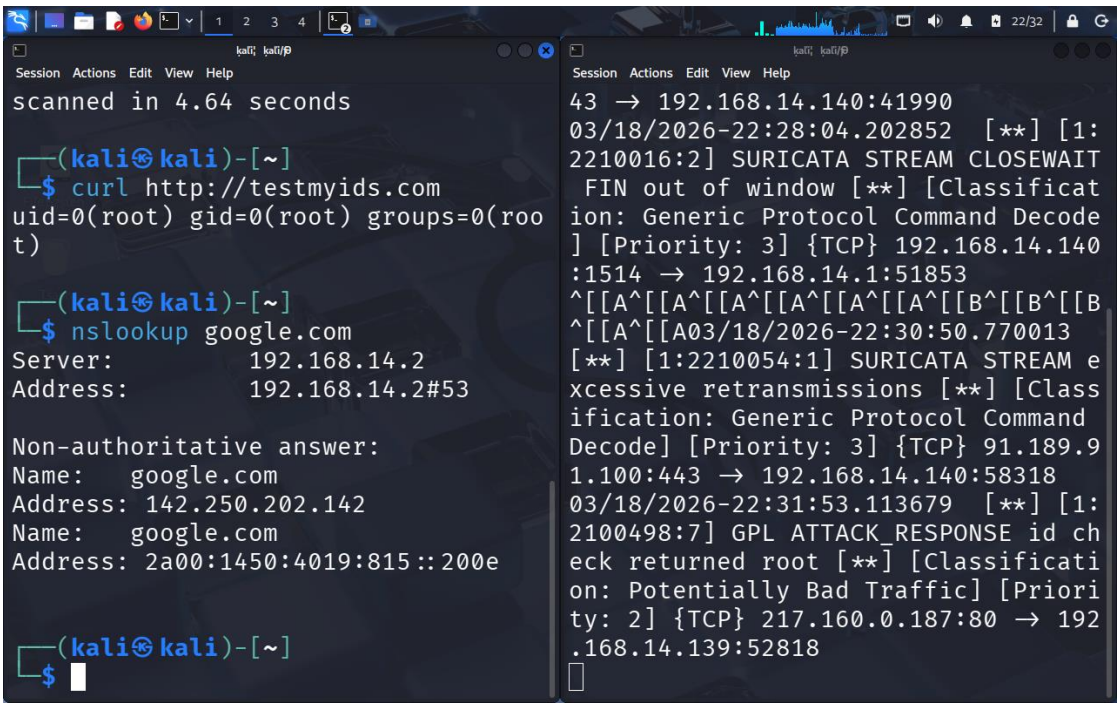

Fig 15.17 Kali Linux — curl http://testmyids.com (left) and nslookup google.com (left) alongside fast.log showing further alerts including GPL ATTACK_RESPONSE

The full fast.log was then reviewed to see all accumulated alerts from the initial testing period, confirming consistent detection across multiple event types.

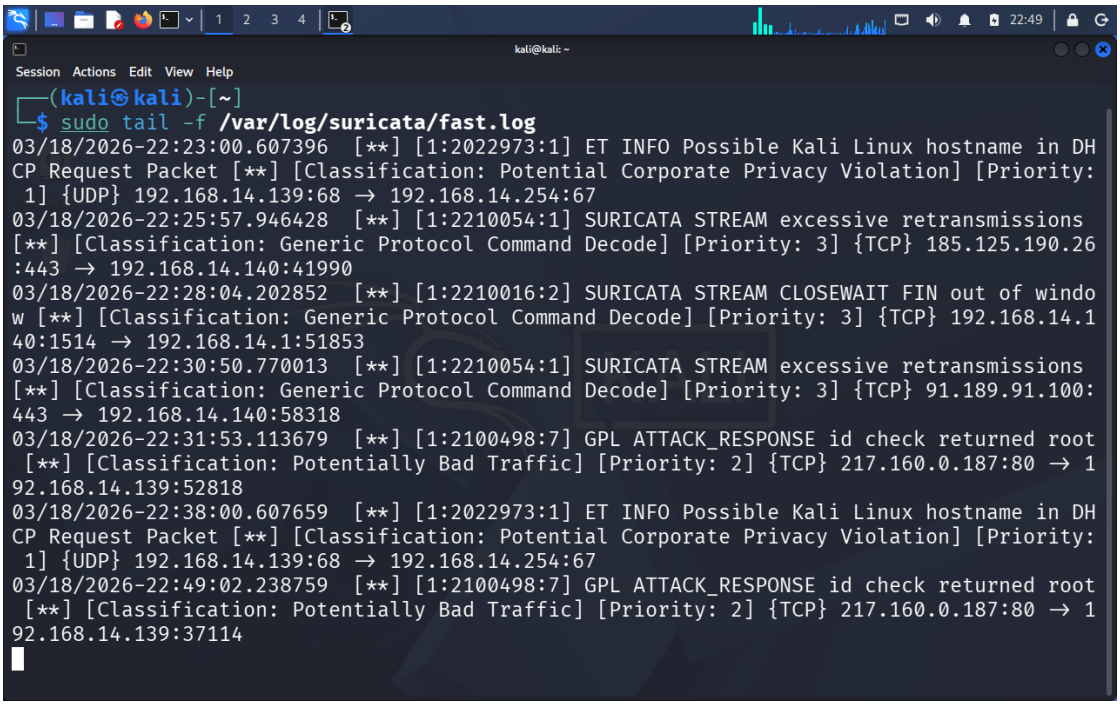

Fig 15.18 Kali Linux — full fast.log view showing all alerts from 22:23 through 22:49 including DHCP, STREAM, and GPL ATTACK_RESPONSE events

The eve.json log was examined to understand its structure. A Python one-liner was used to parse and print each alert event with its timestamp, source, destination, message, and category. This demonstrated the rich structured data available in eve.json compared to the compact fast.log format.

```
sudo cat /var/log/suricata/eve.json | python3 -c "import sys, json\nfor line in sys.stdin:\n  try:\n    event = json.loads(line)\n    if event.get('event_type') == 'alert':\n      ..."
```

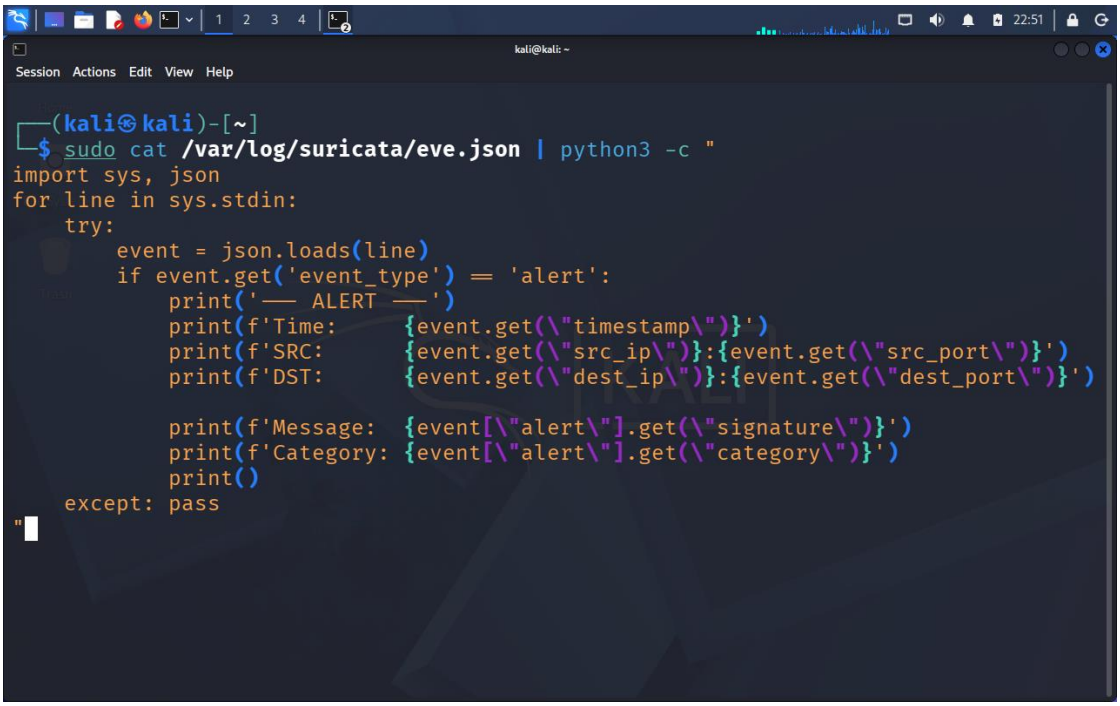

Fig 15.19 Kali Linux — Python script to filter and display alert events from eve.json by parsing event_type, timestamp, src/dst, signature, and category

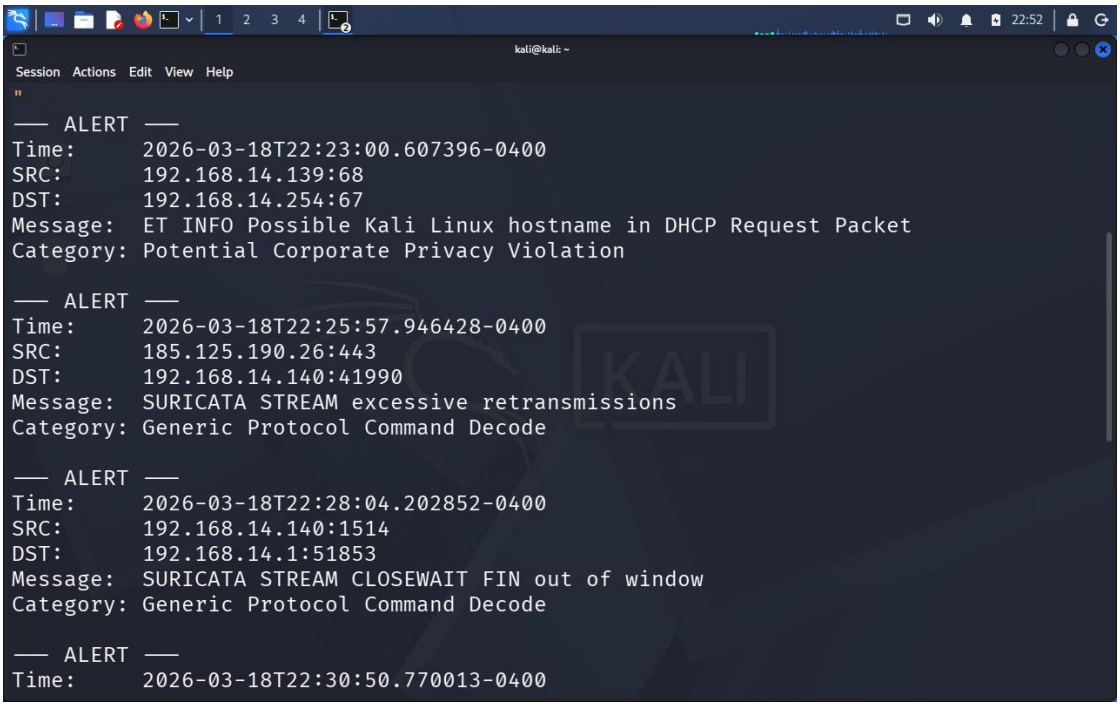

Fig 15.20 Kali Linux — eve.json alert output: ET INFO DHCP, SURICATA STREAM retransmissions, CLOSEWAIT FIN, and GPL ATTACK_RESPONSE events with full field detail

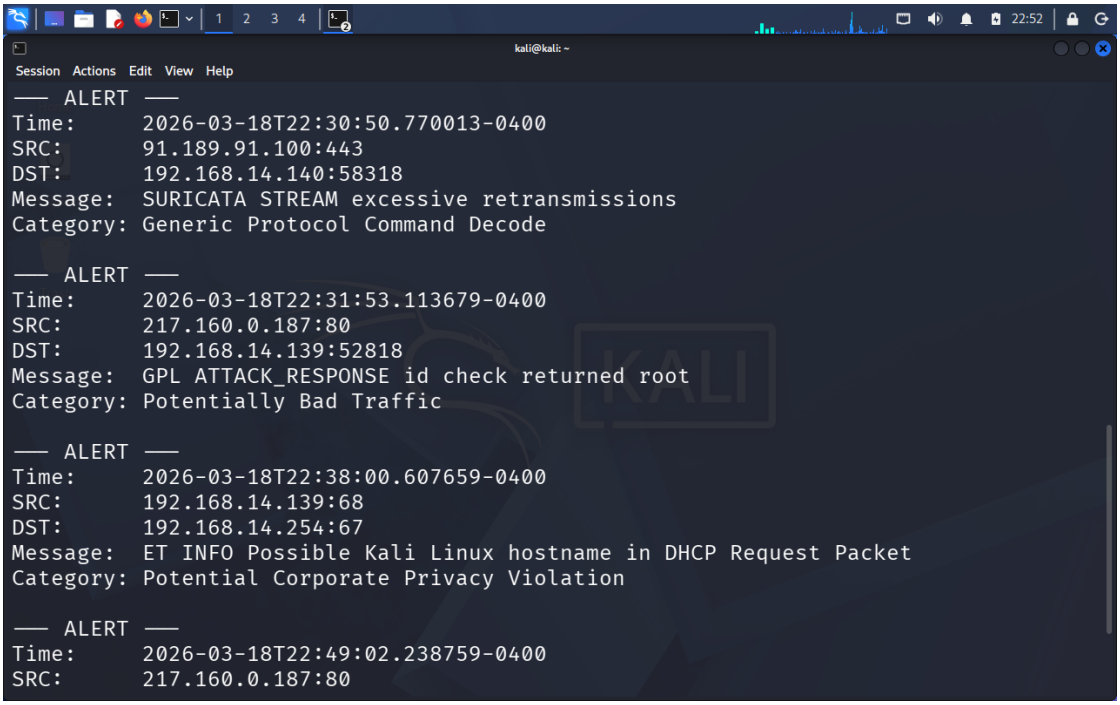

Fig 15.21 Kali Linux — eve.json alert output continued — further STREAM and ATTACK_RESPONSE events from 22:30 through 22:49

A second Python script counted all event types in eve.json to understand the full volume of telemetry being generated. The results showed 2161 flow events, 218 stats, 138 DNS, 79 mDNS, 14 TLS, 7 alerts, 7 HTTP, 5 DHCP, and 2 fileinfo events, illustrating that Suricata's eve.json log captures far more than just alerts.

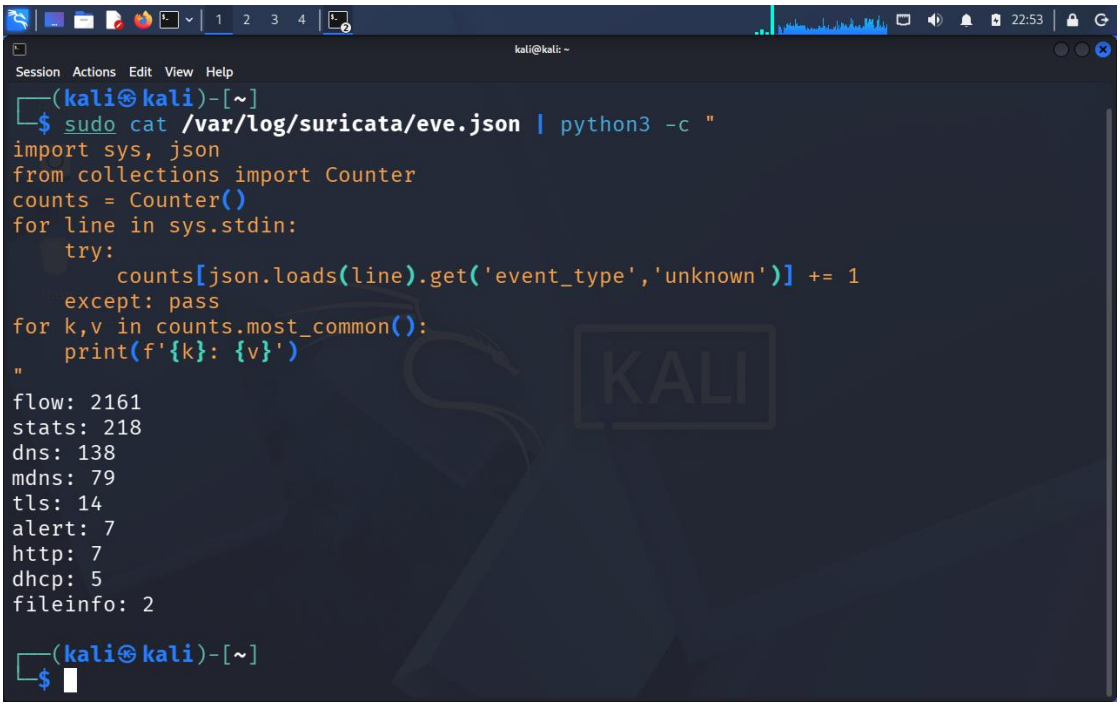

Fig 15.22 Kali Linux — eve.json event type counter: flow 2161, stats 218, dns 138, mdns 79, tls 14, alert 7, http 7, dhcp 5, fileinfo 2

Result

Suricata 8.0.3 was successfully installed and configured on the Kali Linux VM. The af-packet capture interface was correctly set to eth1, the Host-Only adapter carrying lab traffic. The Emerging Threats Open ruleset was downloaded and 49,189 signatures were loaded. The service started cleanly, confirmed active (running) in systemctl, with eth1 worker threads created. Detection was verified in both fast.log and eve.json, with default rules firing on DHCP, TCP stream anomalies, and attack-response signatures from real lab traffic. The event type breakdown from eve.json demonstrated that Suricata generates comprehensive network telemetry beyond alert events alone.

Summary of Days 15–16

Days 15 and 16 established the network monitoring layer that was missing from the Week 1 and 2 setup. Suricata 8.0.3 was installed via apt, configured with eth1 as the capture interface using af-packet, updated with 49,189 Emerging Threats signatures, and confirmed to be actively generating alerts on real lab traffic. The critical learning was the silent failure mode: if the interface name in suricata.yaml is wrong, Suricata runs without errors but sees no traffic and generates no alerts. Verifying by checking fast.log after generating traffic is the only reliable confirmation that the interface configuration is correct. The difference between fast.log (compact, one alert per line) and eve.json (full structured JSON with flow, DNS, TLS, HTTP, and alert event types) was understood and will be important in the Wazuh integration work on Days 18 and 19.

Cyberster Blue Team Internship — Week 3 Day 17

Suricata Rule Writing

Objective

The objective of Day 17 was to write three custom Suricata detection rules covering distinct threat scenarios: an Nmap SYN scan, an HTTP directory traversal exploit attempt, and an ICMP flood denial-of-service attempt. Each rule was written in /etc/suricata/rules/local.rules, referenced in suricata.yaml, and verified by triggering the detection from the Kali VM and observing the alert in fast.log.

Tools Used

- Kali Linux VM — host for Suricata and the machine generating test attack traffic
- /etc/suricata/rules/local.rules — custom rule file containing the three detection rules
- /etc/suricata/suricata.yaml — rule-files section updated to include local.rules
- suricatasc -c reload-rules — live rule reload without service restart
- nmap — used to trigger the Nmap SYN scan rule
- curl — used to craft the HTTP directory traversal request to trigger Rule 2
- ping -f — flood ping used to generate ICMP volume to trigger Rule 3
- /var/log/suricata/fast.log and eve.json — verified alert messages after each test

Procedure

Step 1: Update Ruleset Before Writing Custom Rules

suricata-update was run to ensure the base Emerging Threats ruleset was current before adding custom rules. The output confirmed Suricata 8.0.3 was detected and the rules were fetched from the Emerging Threats CDN. After the update, Suricata was restarted and the status was verified as active (running).

```
sudo suricata-update
```

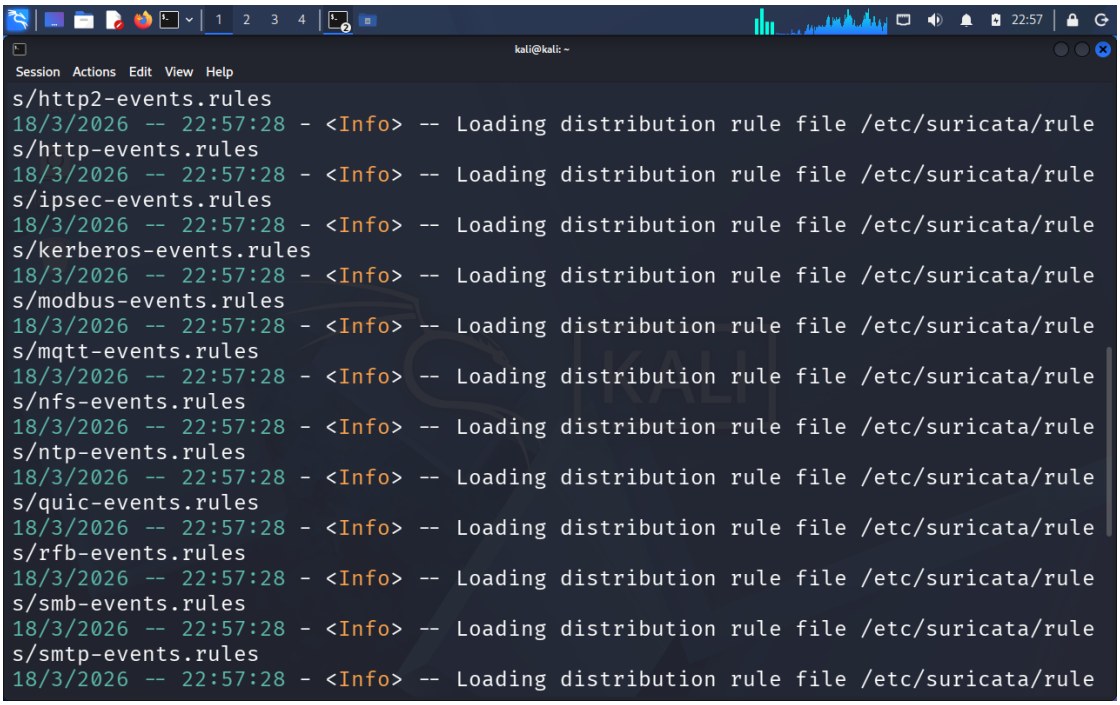

Fig 17.1 Kali Linux — suricata-update loading distribution rule files for http2, http-events, ipsec, kerberos, modbus, mqtt, nfs, ntp, quic, rfb, smb, smtp event categories

```
sudo systemctl restart suricata
sudo systemctl status suricata
```

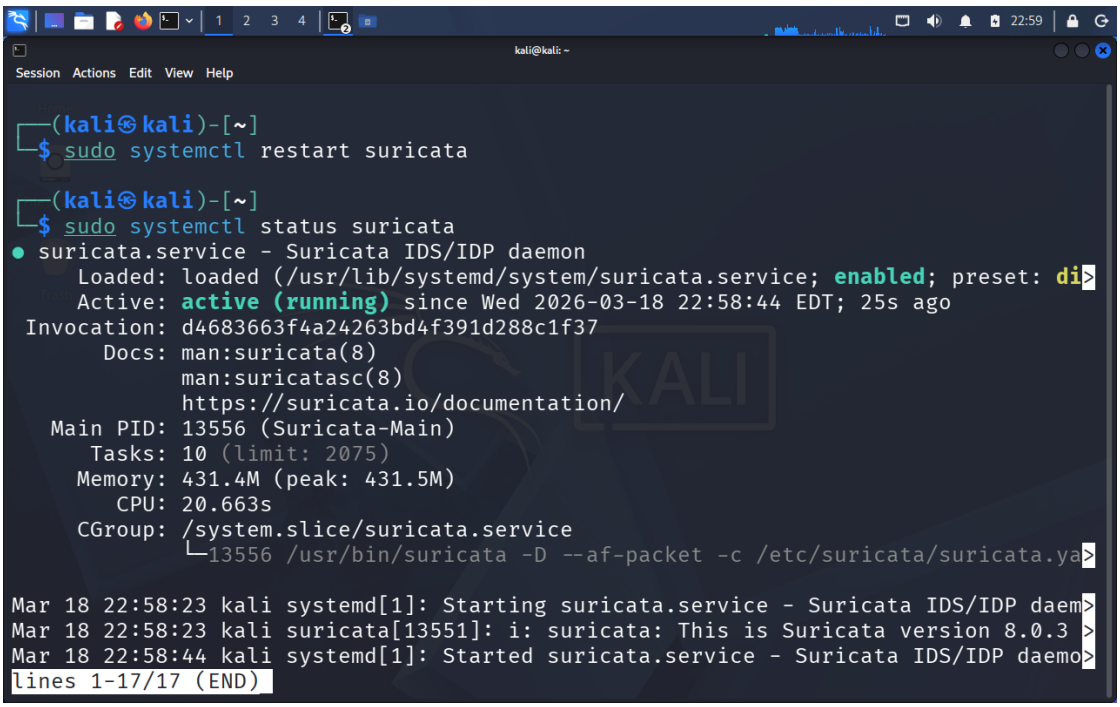

Fig 17.2 Kali Linux — suricata restarted at 22:58:44, systemctl status active (running), PID 13556, starting Suricata version 8.0.3

Step 2: Create the Custom Rules File

A new custom rules file was created at /etc/suricata/rules/local.rules. Each rule was preceded by comment lines explaining the threat scenario, the specific traffic characteristic being matched, and the reasoning behind the design choices.

```
sudo nano /etc/suricata/rules/local.rules
```

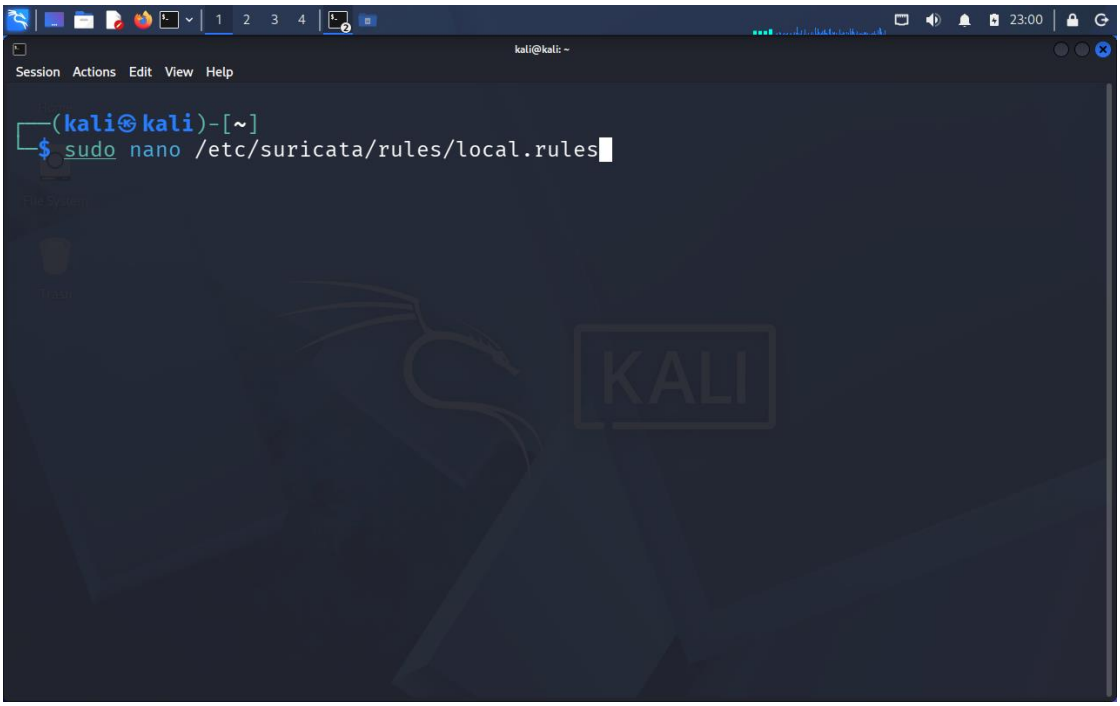

Fig 17.3 Kali Linux — nano opening /etc/suricata/rules/local.rules to create the custom detection rules file

Rule 1 detects Nmap SYN scans. An Nmap SYN scan sends TCP packets with only the SYN flag set (flags:S) without completing the three-way handshake. A threshold of 5 SYN packets within 3 seconds from the same source triggers the alert, distinguishing scan behaviour from legitimate connection attempts. The rule uses classtype:attempted-recon and SID 9000001.

Rule 2 detects HTTP directory traversal exploit attempts. Attackers append ../ sequences to URLs to escape the web root. The rule looks for the string /../ in the HTTP URI with flow:established,to_server and http_uri context, classifying it as a web-application-attack with SID 9000002.

Rule 3 detects ICMP floods. A single ICMP echo request (itype:8) is normal, but a high-rate flood from one source indicates a denial-of-service attempt. The threshold keyword with type both, track by_src, count 10, seconds 2 limits the alert to fire at most once per 2-second window per source IP regardless of packet volume. SID 9000003 and classtype:attempted-dos were used.

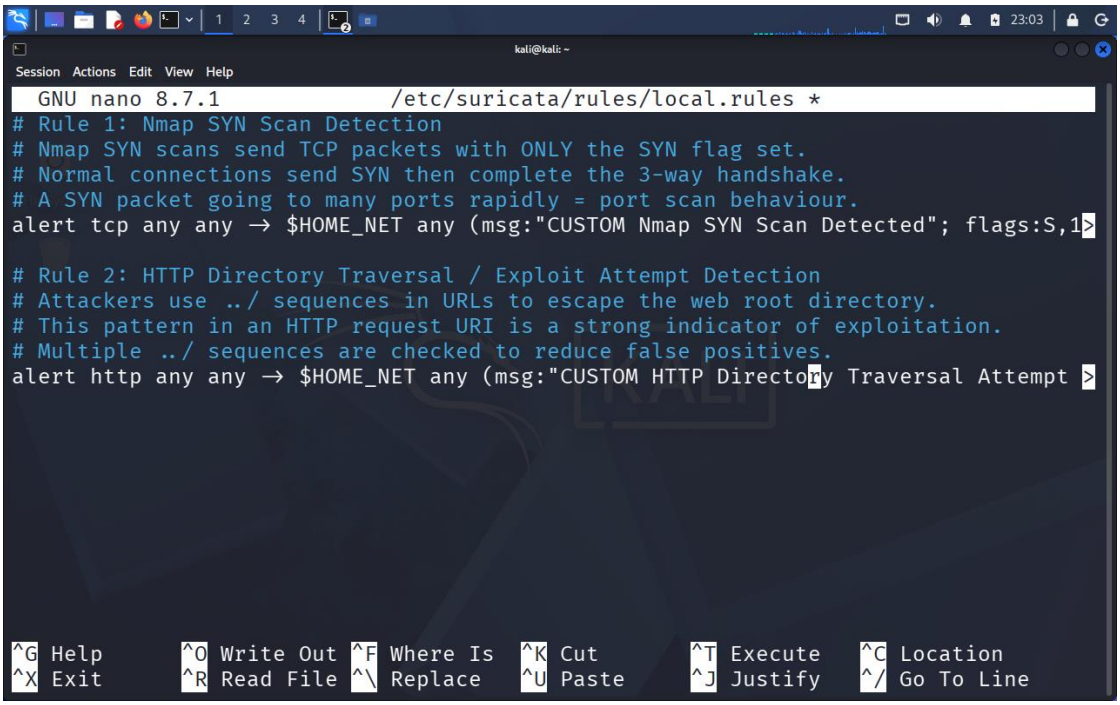

Fig 17.4 Kali Linux — local.rules Rule 1 (Nmap SYN Scan) with comment block and Rule 2 (HTTP Directory Traversal) beginning, both with explained logic

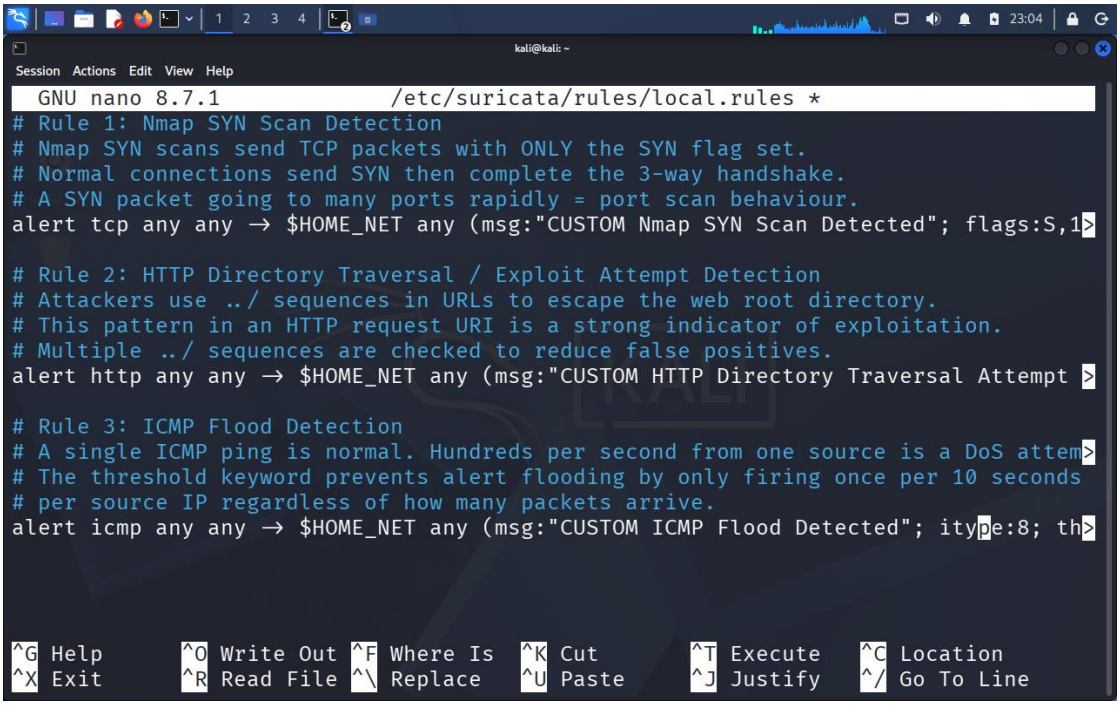

Fig 17.5 Kali Linux — local.rules all three rules: Rule 1 Nmap SYN (flags:S,12; threshold count 5 in 3s), Rule 2 HTTP traversal (content:"/../"; http_uri), Rule 3 ICMP Flood (itype:8; threshold count 10 in 2s)

Step 3: Add local.rules to suricata.yaml

The rule-files section of suricata.yaml was updated to include local.rules alongside the default suricata.rules entry. Without this reference, Suricata would not load the custom rules even if the file exists and is syntactically correct.

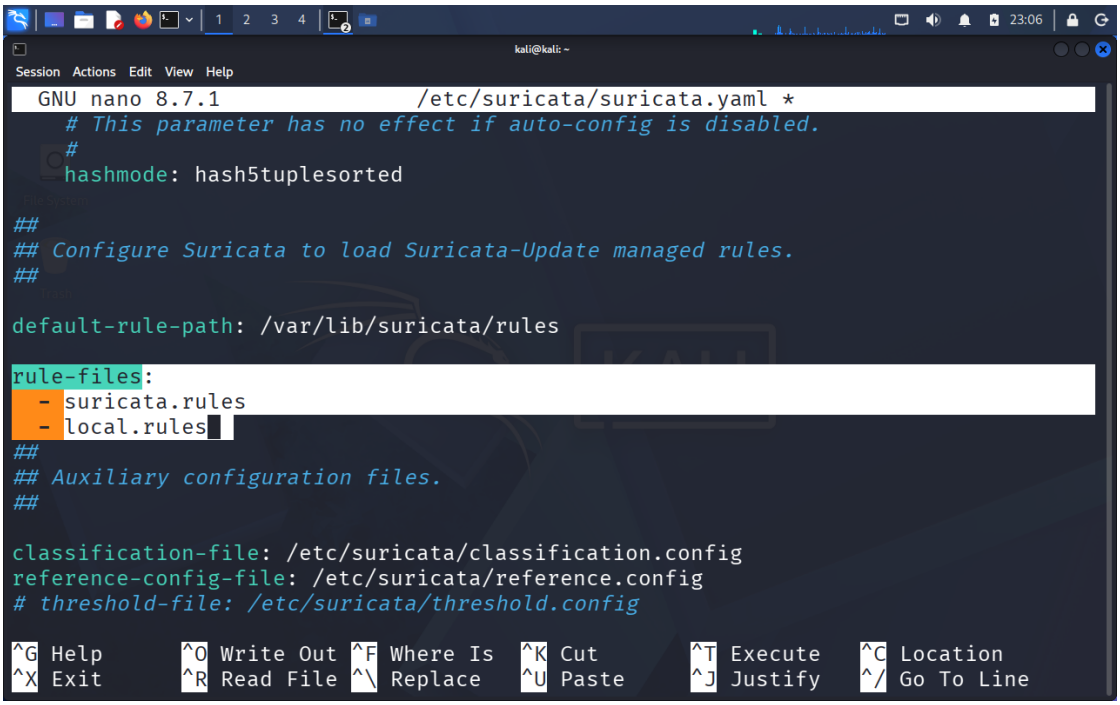

Fig 17.6 Kali Linux — suricata.yaml rule-files section updated: - suricata.rules and - local.rules both listed under default-rule-path: /var/lib/suricata/rules

Step 4: Validate Configuration and Reload Rules

Suricata's configuration test mode was used to validate the updated YAML and rule files without restarting the service. This catches syntax errors in rules or YAML before they cause a failed restart.

```
sudo suricata -T -c /etc/suricata/suricata.yaml --engine-analysis
```

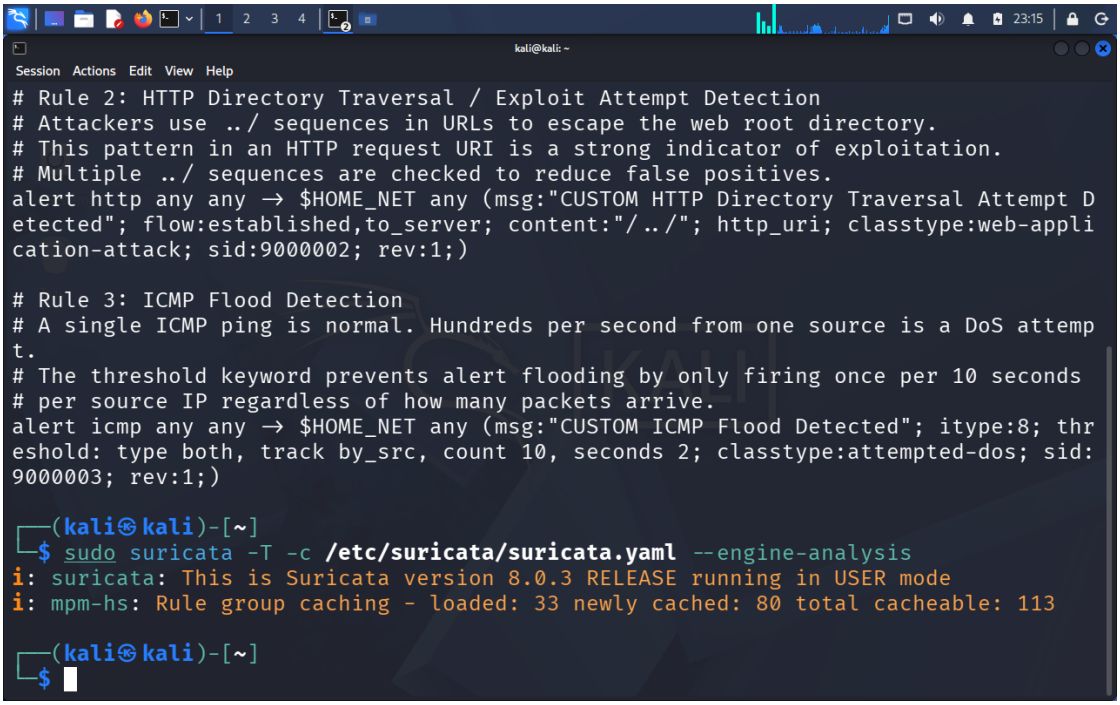

Fig 17.7 Kali Linux — suricata -T config test: Suricata 8.0.3 running in USER mode, rule group caching loaded 33 newly cached, 80 total cacheable — no errors

Suricata was then restarted to load both rule files. The log confirmed 2 rule files processed and 49,184 rules successfully loaded with 0 failed. The custom rules in local.rules were included in the 49,189 total signatures.

```
sudo systemctl restart suricata
```

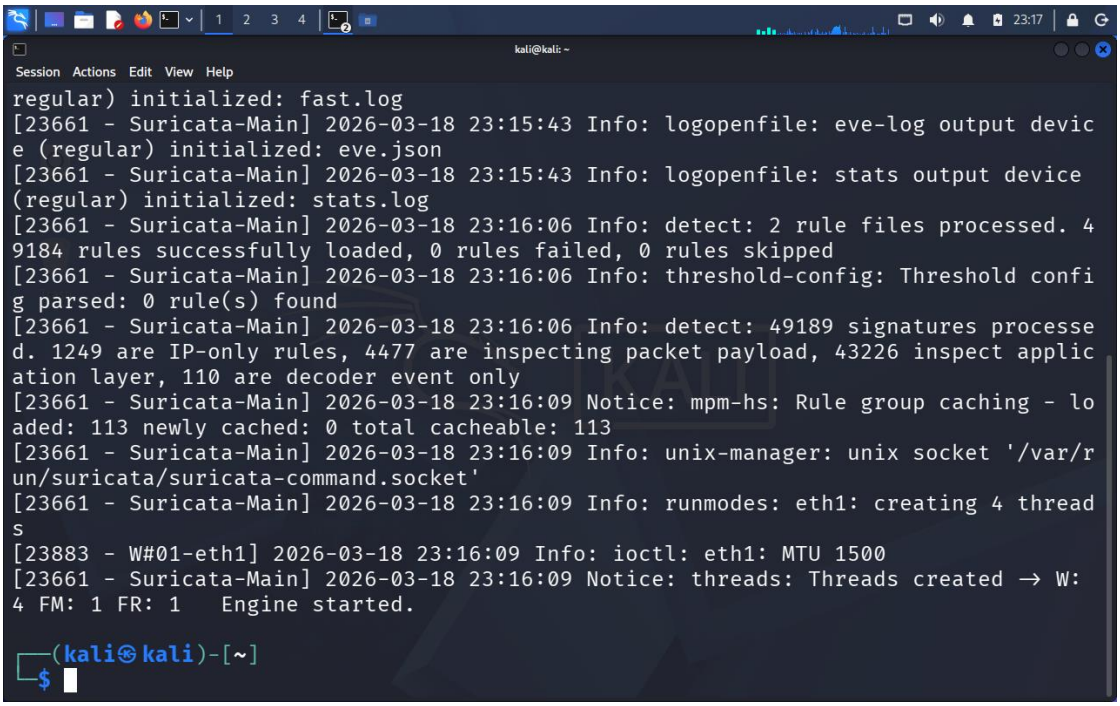

Fig 17.8 Kali Linux — suricata.log after restart: 2 rule files processed, 49184 rules loaded, 49189 signatures processed, eth1 threads created, engine started

After the rules were active in the running service, the live reload command was also tested to demonstrate that rules can be reloaded without a service restart in production environments.

```
sudo suricatasc -c reload-rules
```

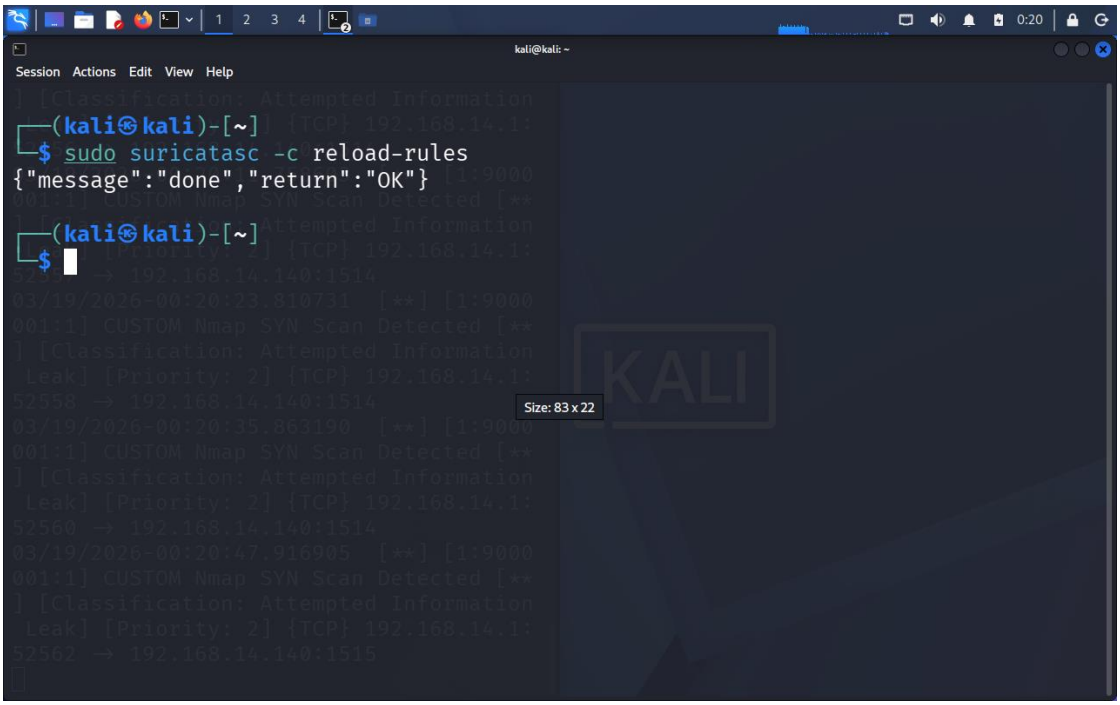

Fig 17.9 Kali Linux — suricatasc reload-rules returns {"message":"done","return":"OK"}, confirming live rule reload succeeded

The Suricata log confirmed the rule reload: rule reload starting, 2 rule files processed, 49184 rules loaded, reload complete.

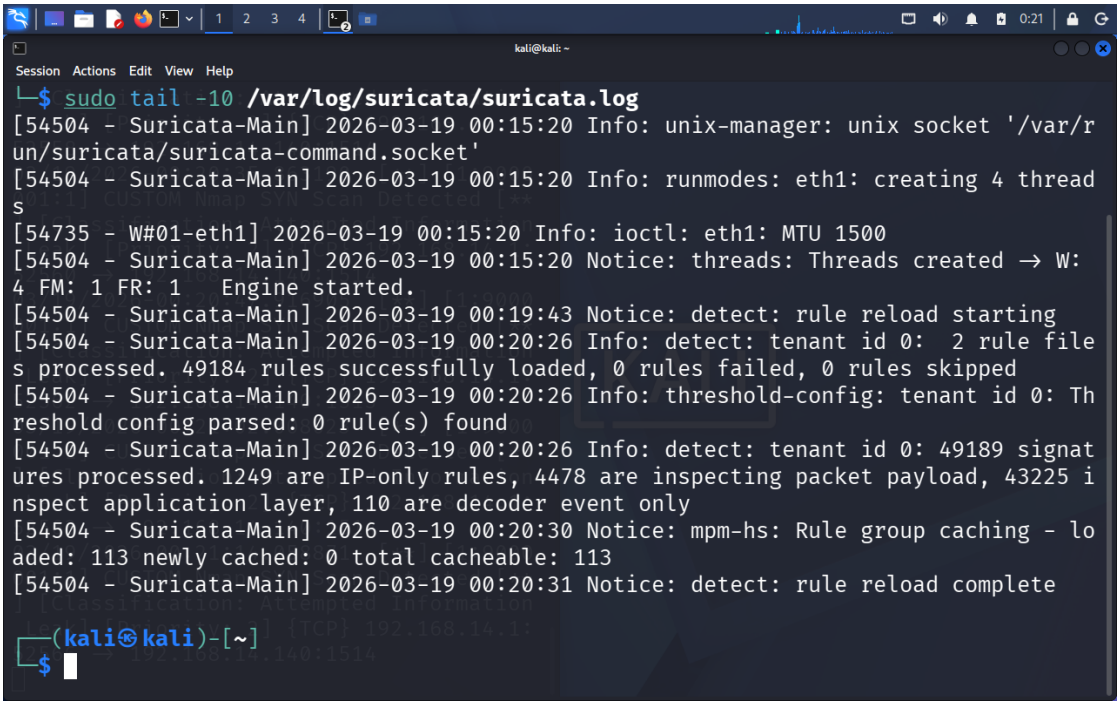

Fig 17.10 Kali Linux — suricata.log tail showing rule reload starting at 00:19:43, 49184 rules loaded, rule reload complete at 00:20:31

Step 5: Trigger Rule 1 — Nmap SYN Scan Detection

An Nmap SYN scan was launched from the Kali VM targeting the Ubuntu Server at 192.168.14.140. The fast.log was monitored in a second terminal. The custom alert fired immediately.

```
sudo nmap -sS 192.168.14.140
```

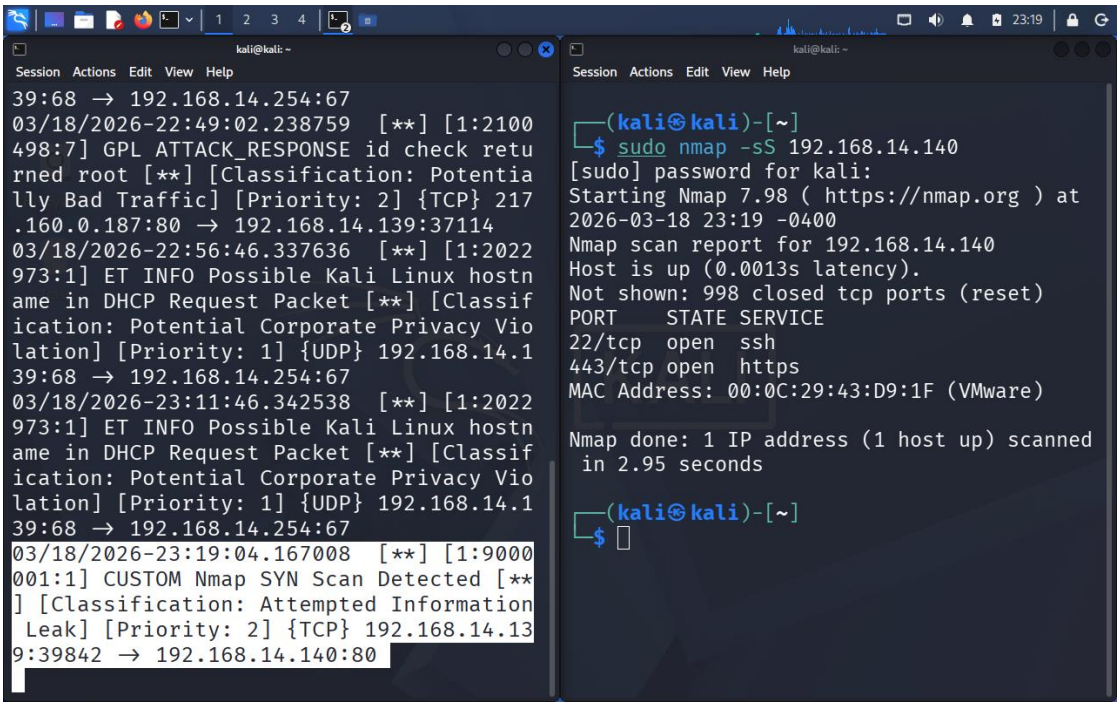

Fig 17.11 Kali Linux — nmap -sS 192.168.14.140 (right) with fast.log (left) showing [1:9000001:1] CUSTOM Nmap SYN Scan Detected at 23:19:04, Priority 2, TCP 192.168.14.139 → 192.168.14.140:80

Step 6: Trigger Rule 2 — HTTP Directory Traversal Detection

A curl request was crafted containing the /../ directory traversal sequence in the URI. The fast.log was monitored to confirm the HTTP rule fired. A Python-based SimpleHTTP server was running on the Ubuntu Server on port 80 to receive the request.

```
curl "http://192.168.14.140/test/../../etc/passwd"
curl --path-as-is "http://192.168.14.140/../etc/passwd"
```

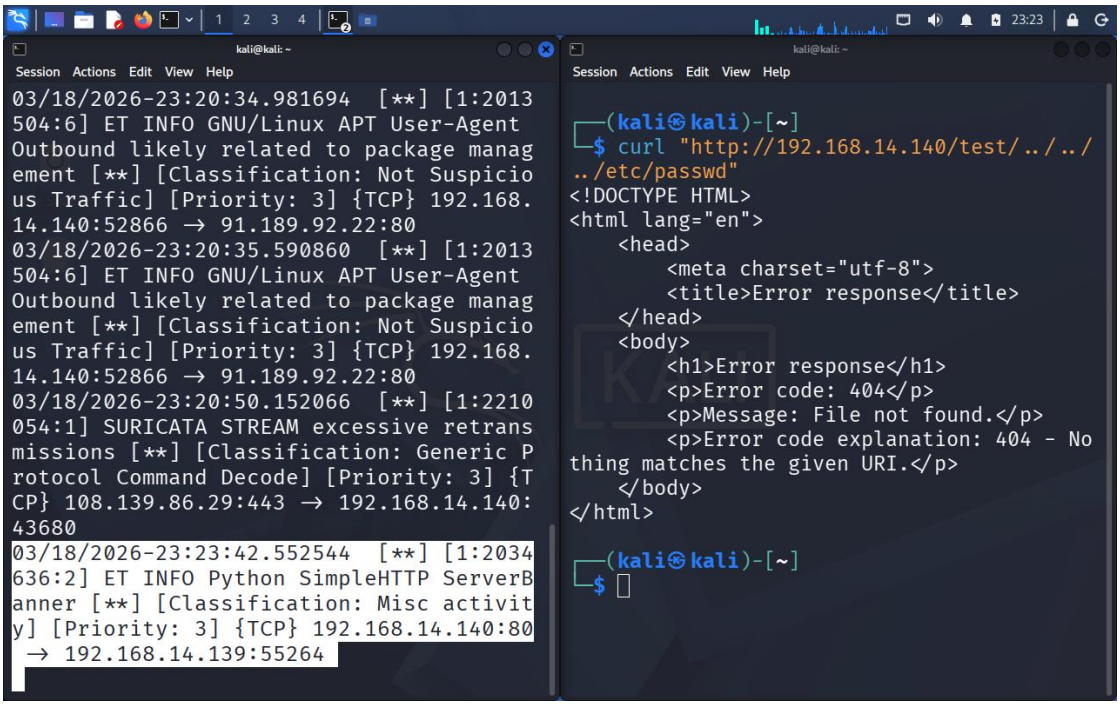

Fig 17.12 Kali Linux — curl with traversal URI (right); fast.log (left) showing ET INFO Python SimpleHTTP ServerBanner at 23:23:42, confirming HTTP traffic reached server

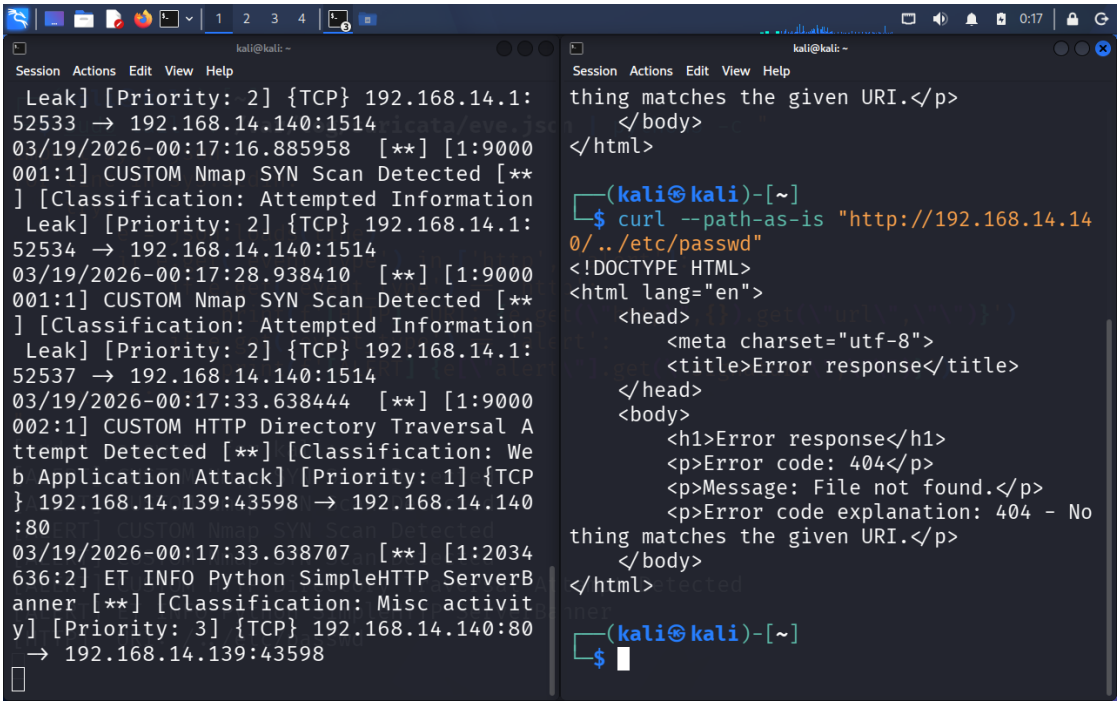

Fig 17.13 Kali Linux — fast.log (left) showing [1:9000002:1] CUSTOM HTTP Directory Traversal Attempt Detected at 00:17:33, Priority 1, Web Application Attack; curl --path-as-is traversal request (right) returning 404

Step 7: Trigger Rule 3 — ICMP Flood Detection

A flood ping was sent to the Ubuntu Server with 100 packets at maximum send rate. The threshold setting in Rule 3 meant the alert fired once per 2-second window once 10 ICMP type-8 packets from the same source were observed.

```
sudo ping -f -c 100 192.168.14.140
```

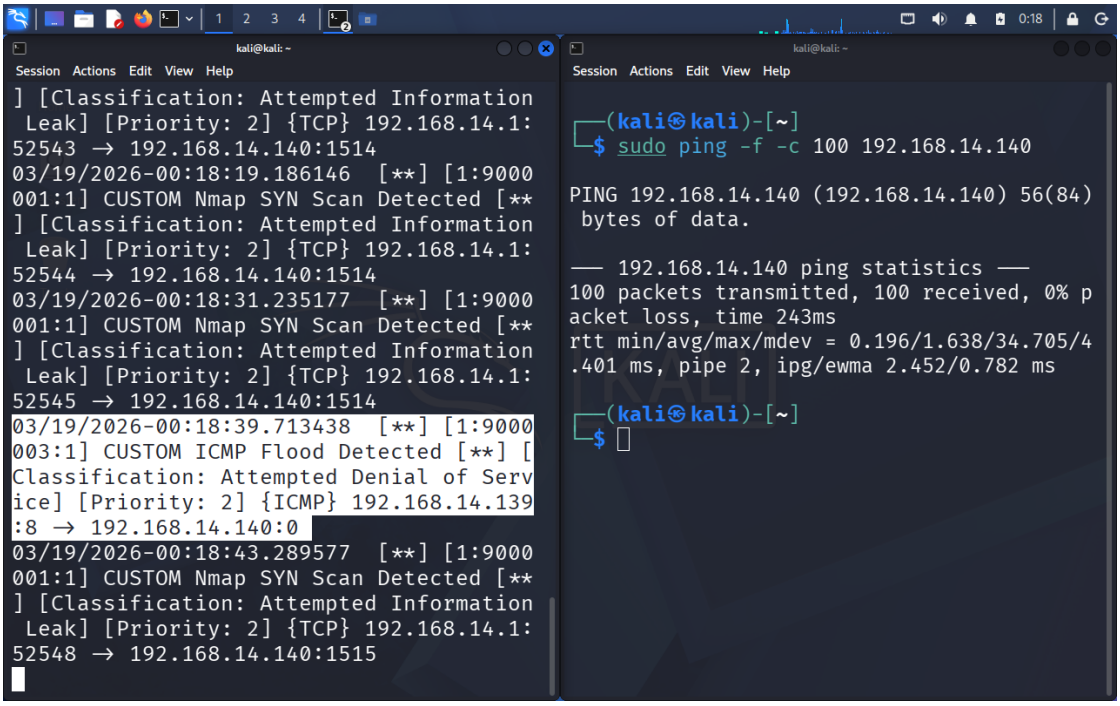

Fig 17.14 Kali Linux — ping -f -c 100 (right) transmitting 100 packets in 243ms; fast.log (left) showing [1:9000003:1] CUSTOM ICMP Flood Detected at 00:18:39, Attempted DoS, ICMP 192.168.14.139:8 → 192.168.14.140:0

Step 8: Verify All Three Rules in eve.json

A Python script was run against eve.json to extract and display all three custom alert types together, confirming each rule fired correctly with the right signature name and HTTP URI context.

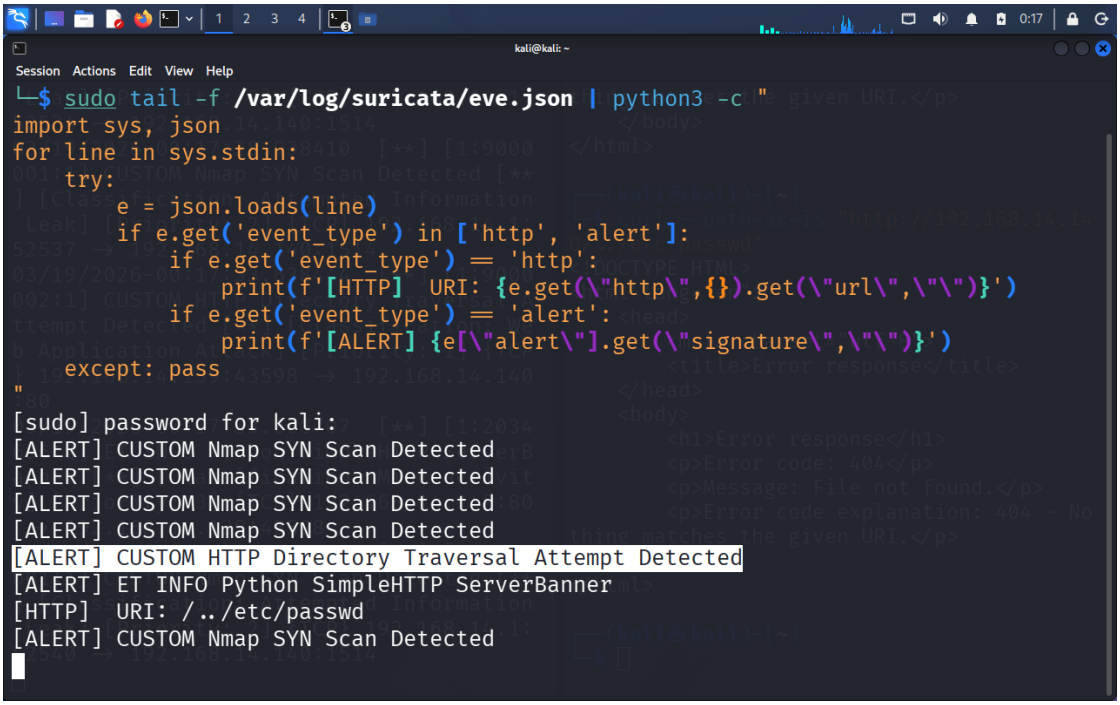

Fig 17.15 Kali Linux — eve.json parsed output showing [ALERT] CUSTOM Nmap SYN Scan Detected, [ALERT] CUSTOM HTTP Directory Traversal Attempt Detected, and [HTTP] URI: /../etc/passwd all confirmed in structured log

Step 9: Confirm Complete Rules File

The final state of the local.rules file was verified with cat to confirm all three rules were saved correctly with their complete options including threshold settings.

```
cat /etc/suricata/rules/local.rules
```

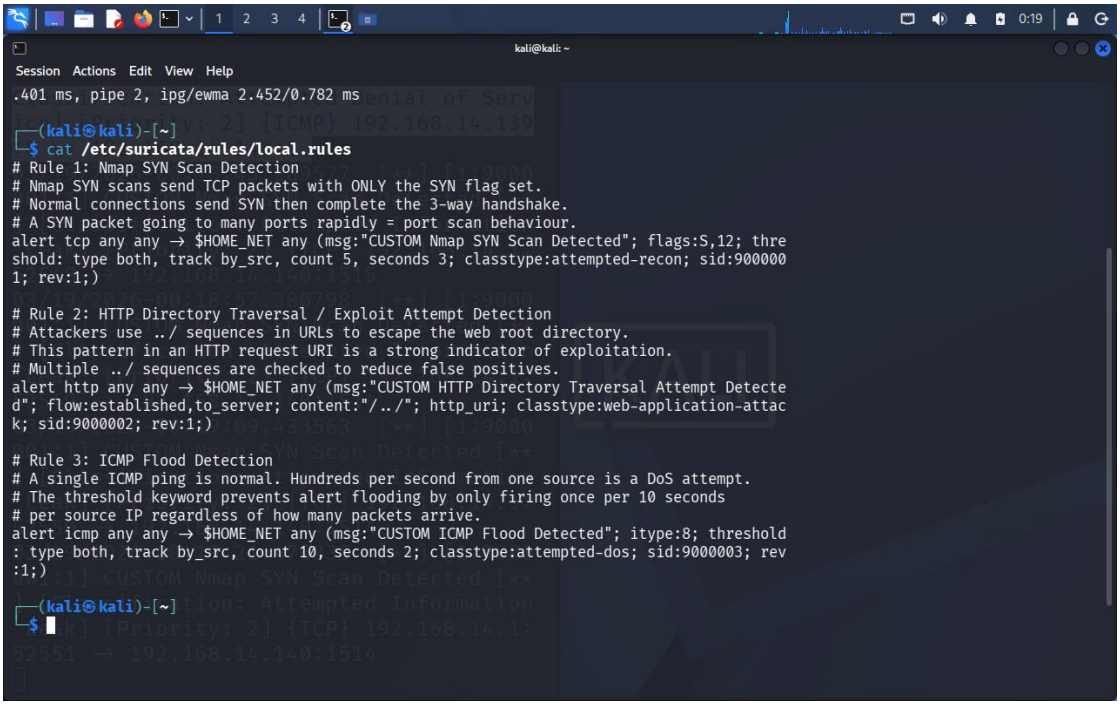

Fig 17.16 Kali Linux — cat local.rules output: Rule 1 (flags:S,12; threshold count 5, seconds 3), Rule 2 (content:"/../"; http_uri), Rule 3 (itype:8; threshold count 10, seconds 2) — all three rules complete

Result

All three custom Suricata rules were successfully written, loaded, and verified. Rule 1 (SID 9000001) fired on the Nmap SYN scan with priority 2 and classification Attempted Information Leak. Rule 2 (SID 9000002) fired on the HTTP directory traversal request with priority 1 and classification Web Application Attack, confirmed in both fast.log and eve.json with the exact URI visible. Rule 3 (SID 9000003) fired on the ICMP flood with priority 2 and classification Attempted Denial of Service. The suricatasc live reload was confirmed working, and the configuration test mode was used to validate syntax before loading. A total of 49,184 combined rules were active after the update.

Summary of Day 17

Day 17 translated threat knowledge into working detection logic. Writing effective rules required understanding what distinguishes malicious traffic from normal traffic at the packet and protocol level: a SYN packet to many ports without handshake completion indicates scanning; the string /../ in an HTTP URI indicates path traversal; hundreds of ICMP type-8 packets per second from one source indicates a flood. The threshold keyword was essential for the ICMP rule because without it, a 100-packet flood would generate 100 identical alerts per session, flooding the log with noise. The first adjustment made after initial testing was reducing the Nmap rule's threshold window from 5 seconds to 3 seconds to account for faster scan rates. Rule SIDs above 9,000,000 were used to avoid conflicts with the

community ruleset. The eve.json data confirmed each rule's signature_id and full parsed fields, validating that the alerts were correctly structured for the Wazuh integration work in Days 18 and 19.

Cyberster Blue Team Internship — Week 3 Days 18–19

Suricata and Wazuh Integration

Objective

The objective of Days 18 and 19 was to integrate Suricata with Wazuh by configuring the Wazuh agent on Kali Linux to monitor eve.json as a log source, verify that the Suricata decoder and ruleset were active on the Wazuh Manager, confirm that Suricata network alerts appeared in the Wazuh Security Events dashboard, and perform a correlation exercise combining a Suricata network alert with a Wazuh FIM (File Integrity Monitoring) alert to demonstrate the combined detection picture.

Tools Used

- Kali Linux Wazuh Agent v4.7.5 — configured to monitor /var/log/suricata/eve.json as a json-format localfile
- /var/ossec/etc/ossec.conf (Kali) — agent configuration file updated with Suricata localfile block
- Ubuntu Server Wazuh Manager — received and processed Suricata events; hosts Suricata decoder 0006-json_decoders.xml and ruleset 0475-suricata_rules.xml
- Wazuh Security Events dashboard (Windows host) — used to view and expand Suricata alerts in the UI
- nmap, curl, ping -f — traffic generation tools to trigger Suricata alerts for integration testing
- touch, echo, rm in /etc — FIM test events to generate Wazuh host-based alerts for correlation exercise

Procedure

Step 1: Verify Wazuh Agent and Suricata are Running

Before modifying the agent configuration, both the Wazuh agent and Suricata service were confirmed active on the Kali VM. The agent had been running since 22:41:54 on March 18, confirming it was operational and connected to the Manager from the previous week's configuration.

```
sudo systemctl status wazuh-agent
sudo systemctl status suricata
```

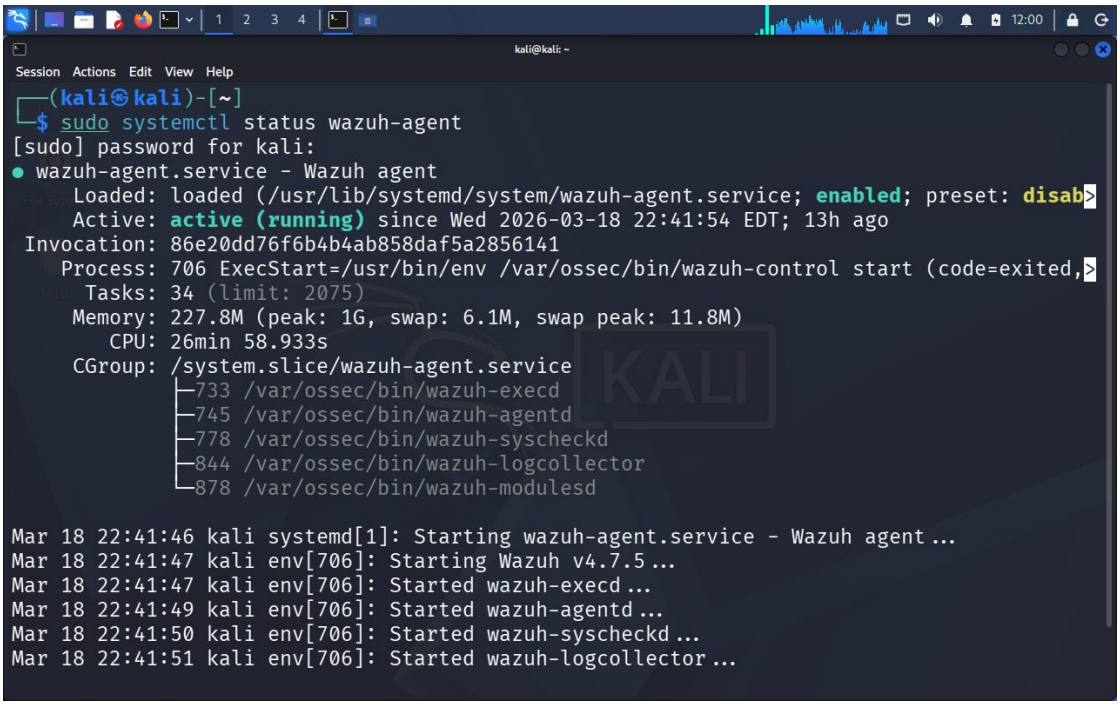

Fig 18.1 Kali Linux — wazuh-agent.service active (running) since 2026-03-18 22:41:54, all subprocesses wazuh-execd, agentd, syscheckd, logcollector, modulesd running

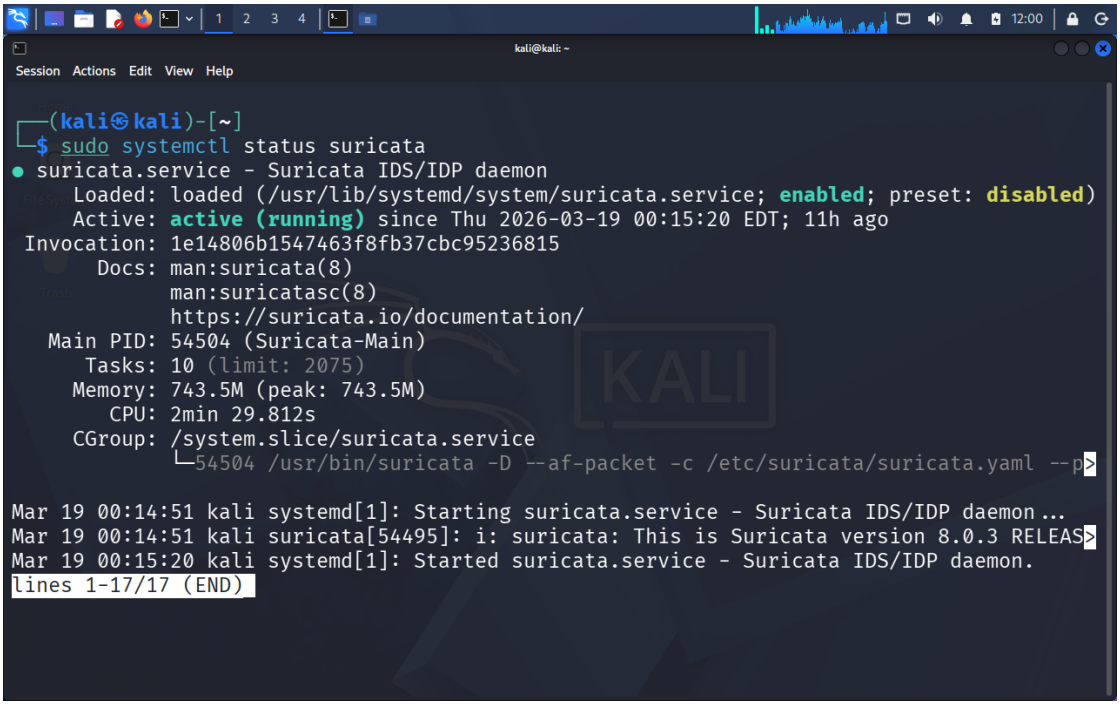

Fig 18.2 Kali Linux — suricata.service active (running) since 2026-03-19 00:15:20, PID 54504, running with --af-packet on /etc/suricata/suricata.yaml

Step 2: Add Suricata eve.json localfile Block to Agent ossec.conf

The Wazuh agent configuration file on Kali was opened and a new localfile block was added to instruct the log collector to monitor eve.json. The log_format was set to json, which is critical: this tells the agent to parse each line as a JSON object and forward the structured fields to the Manager rather than

treating the file as plain syslog text. Without json format, the Suricata decoder cannot extract fields from the events.

```
sudo nano /var/ossec/etc/ossec.conf
```

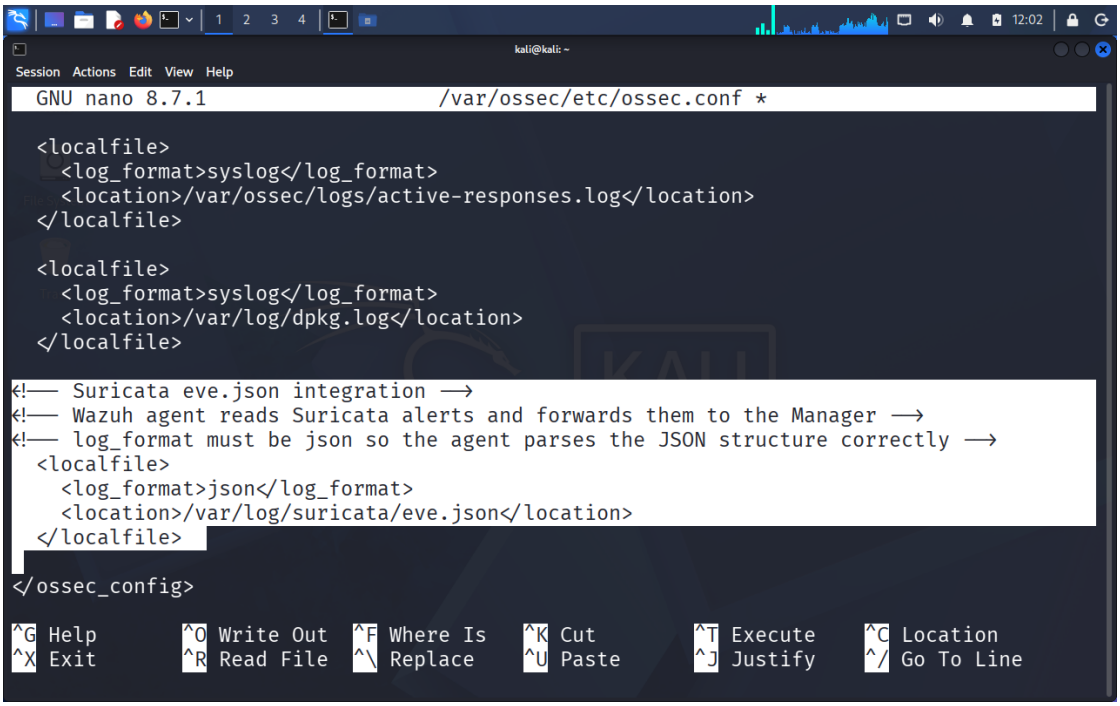

Fig 18.3 Kali Linux — ossec.conf Suricata localfile block: <log_format>json</log_format> and <location>/var/log/suricata/eve.json</location> added with explanatory comments

Step 3: Restart the Wazuh Agent

The Wazuh agent was restarted to apply the new configuration. The status was checked to confirm it came back active and all subprocesses were running including wazuh-logcollector, which is the process responsible for reading the new eve.json localfile entry.

```
sudo systemctl restart wazuh-agent
sudo systemctl status wazuh-agent
```

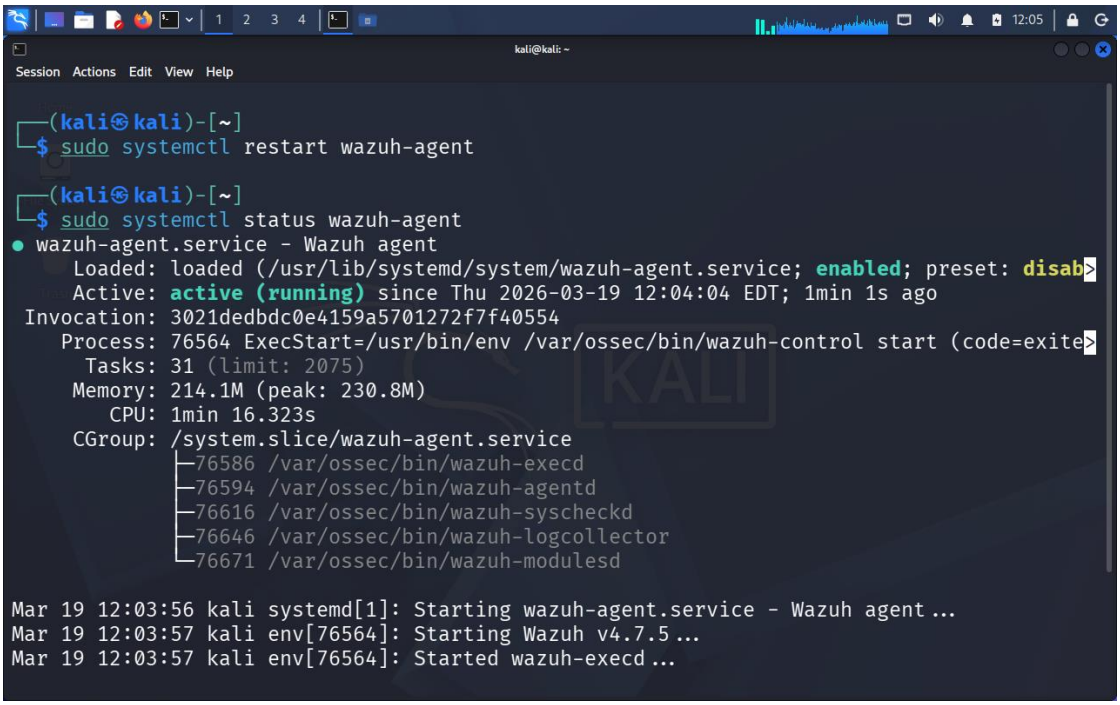

Fig 18.4 Kali Linux — wazuh-agent restarted at 12:04:04, all 5 subprocesses active: wazuh-execd, agentd, syscheckd, logcollector, modulesd

Step 4: Verify Suricata Decoder and Rules on the Wazuh Manager

On the Ubuntu Server (Wazuh Manager), the built-in Suricata decoder and rules were located and examined. The JSON decoder at /var/ossec/ruleset/decoders/0006-json_decoders.xml handles the parsing of eve.json events. This decoder uses the JSON_Decoder plugin to extract all structured fields from each event.

```
sudo cat /var/ossec/ruleset/decoders/0006-json_decoders.xml
```

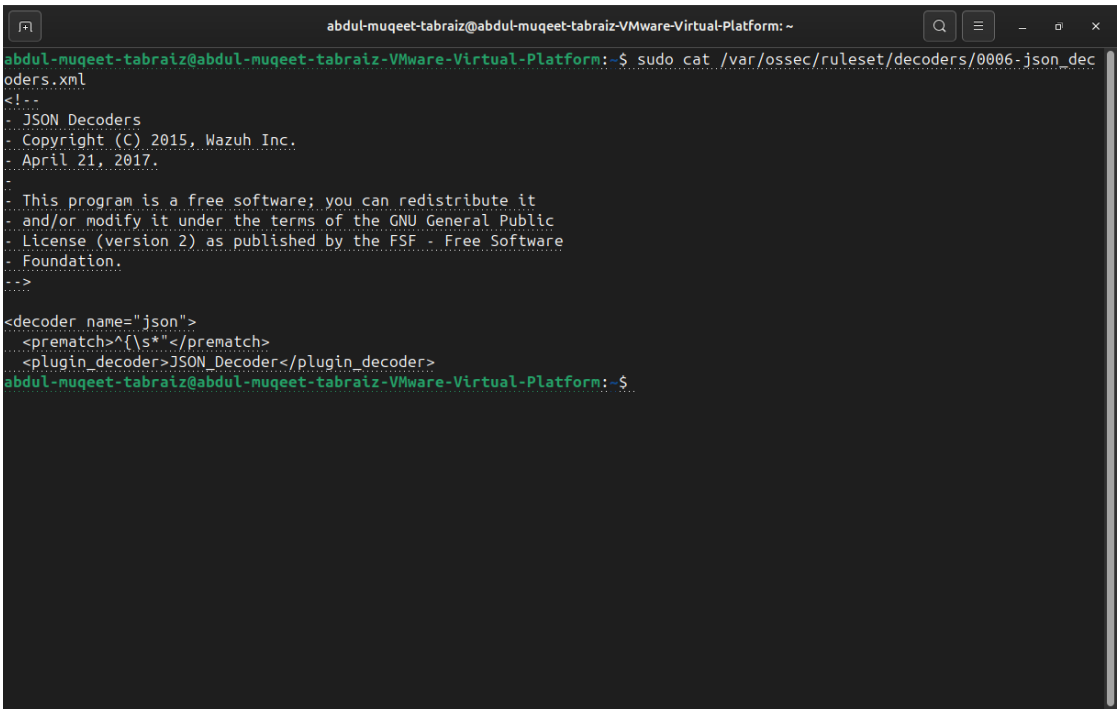

Fig 18.5 Ubuntu Server (Wazuh Manager) — 0006-json_decoders.xml: decoder name="json" with plugin_decoder JSON_Decoder and prematch ^{\s* for JSON events

The Suricata-specific ruleset at /var/ossec/ruleset/rules/0475-suricata_rules.xml was also examined. This file contains the rules that match decoded Suricata events and assign them to the ids,suricata rule group. Rule 86600 (level 0) matches any Suricata message and rule 86601 (level 3) specifically matches alert-type events with the signature description.

```
sudo cat /var/ossec/ruleset/rules/0475-suricata_rules.xml | head -40
```

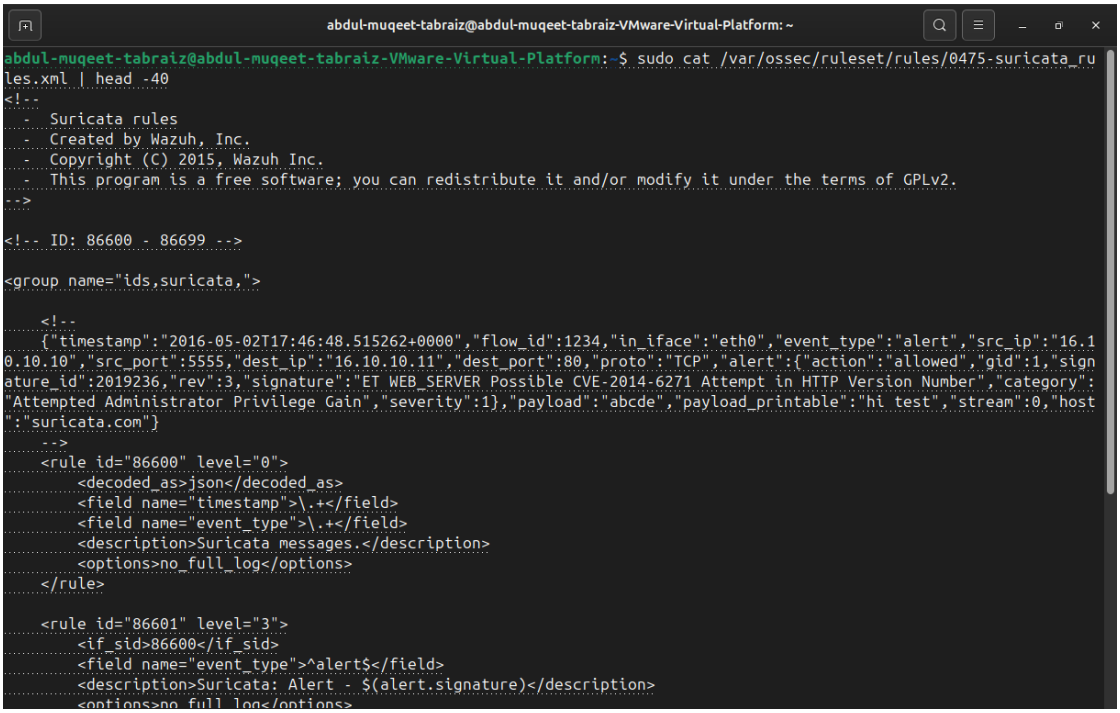

Fig 18.6 Ubuntu Server (Wazuh Manager) — 0475-suricata_rules.xml header showing rule 86600 (Suricata messages, level 0) and rule 86601 (Suricata alert — matches event_type:alert, level 3)

Step 5: Troubleshoot — Too Many Fields for JSON Decoder

When first testing the integration, the Wazuh Manager log showed repeated "Too many fields for JSON decoder" errors. This was caused by Suricata's eve.json flow events containing more JSON fields than the Wazuh JSON decoder was configured to handle. This is a known behaviour with certain Wazuh versions and is harmless for alert-type events, which the decoder processes correctly. The alert-type events continued to reach the dashboard.

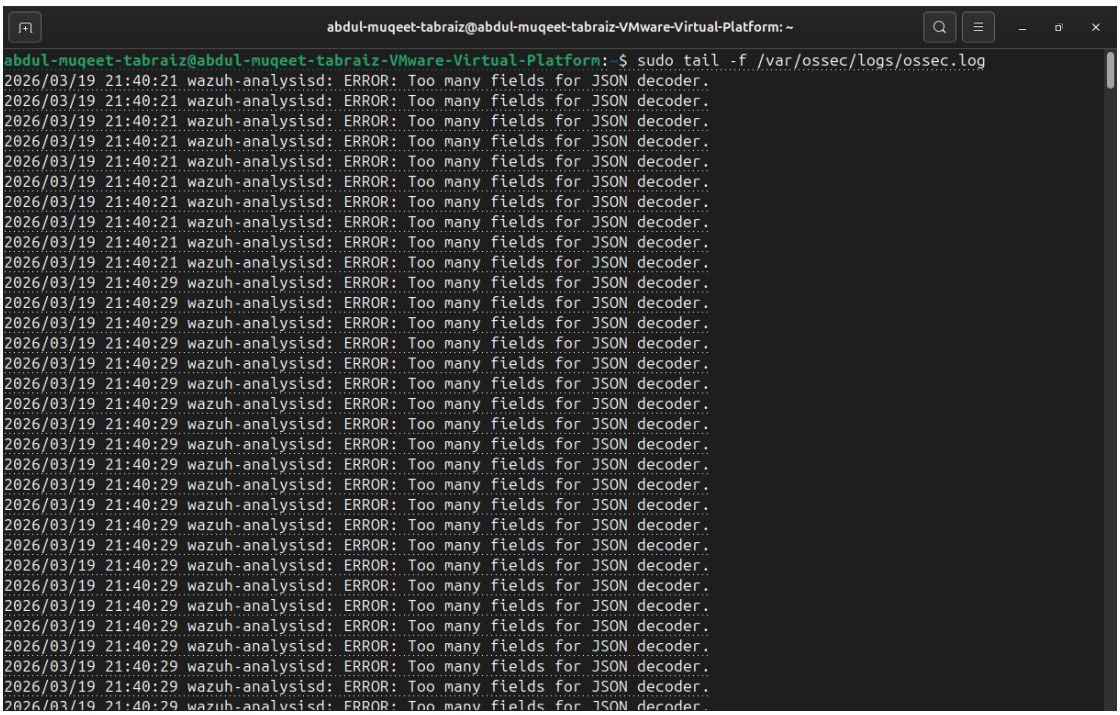

Fig 18.7 Ubuntu Server (Wazuh Manager) — ossec.log showing repeated wazuh-analysisd: ERROR: Too many fields for JSON decoder at 21:40:21 and 21:40:29 — caused by eve.json flow event volume

An attempt was made to locate a more specific Suricata rule file at /var/ossec/ruleset/rules/0850-suricata_rules.xml, but this file did not exist in this Wazuh version. The existing 0475-suricata_rules.xml was confirmed to be the active ruleset.

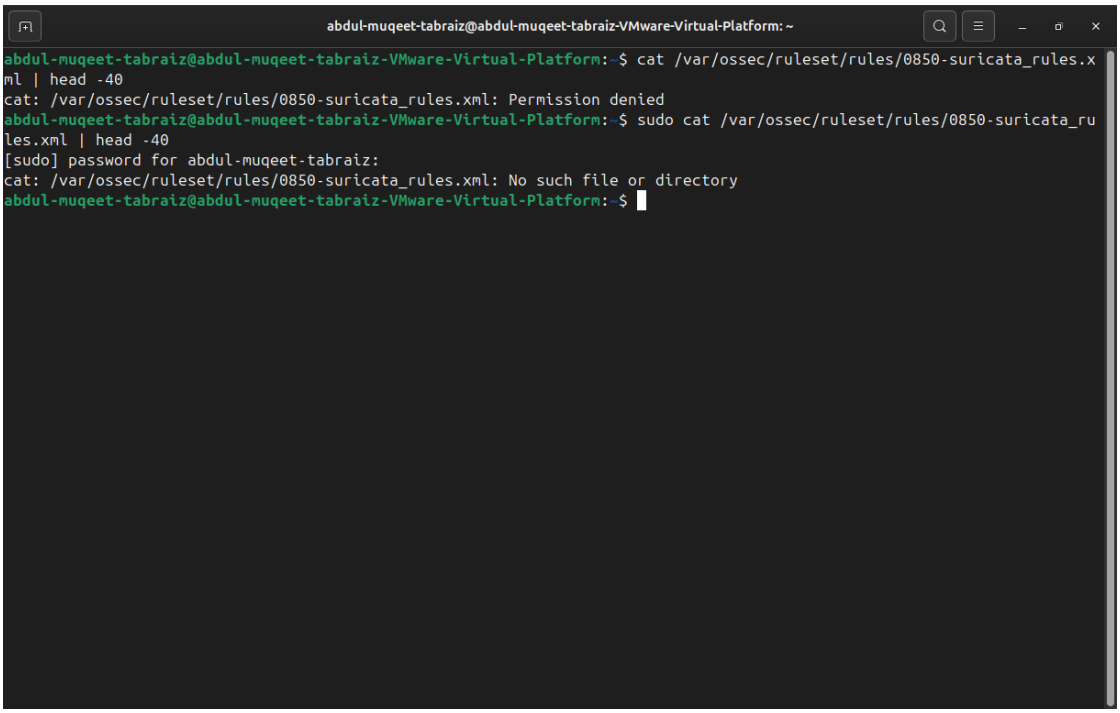

Fig 18.8 Ubuntu Server (Wazuh Manager) — 0850-suricata_rules.xml: No such file or directory; 0475-suricata_rules.xml confirmed as the active Suricata ruleset

The Manager log was grep-searched for suricata-related entries and the alerts.log was also checked, confirming that despite the decoder field errors, Suricata alert events were being received and the FIM module was processing eve.json file change events.

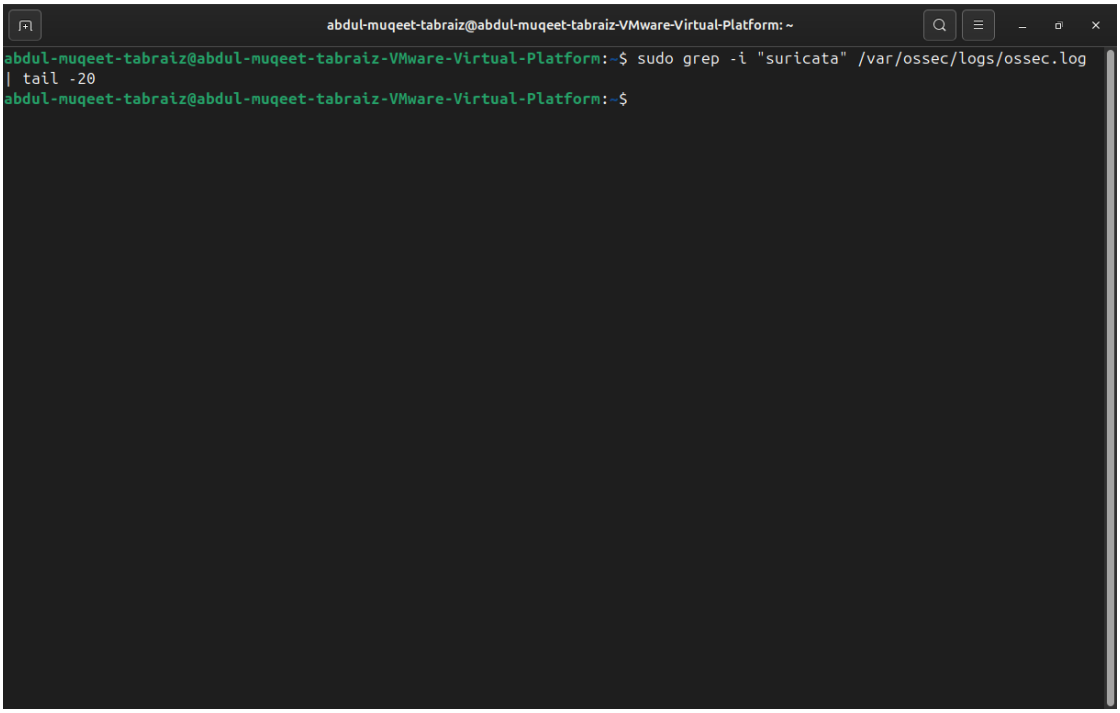

Fig 18.9 Ubuntu Server (Wazuh Manager) — grep suricata in ossec.log returns no errors; sudo grep in alerts log shows File '/var/log/suricata/eve.json' modified events confirming the agent is reading the log file

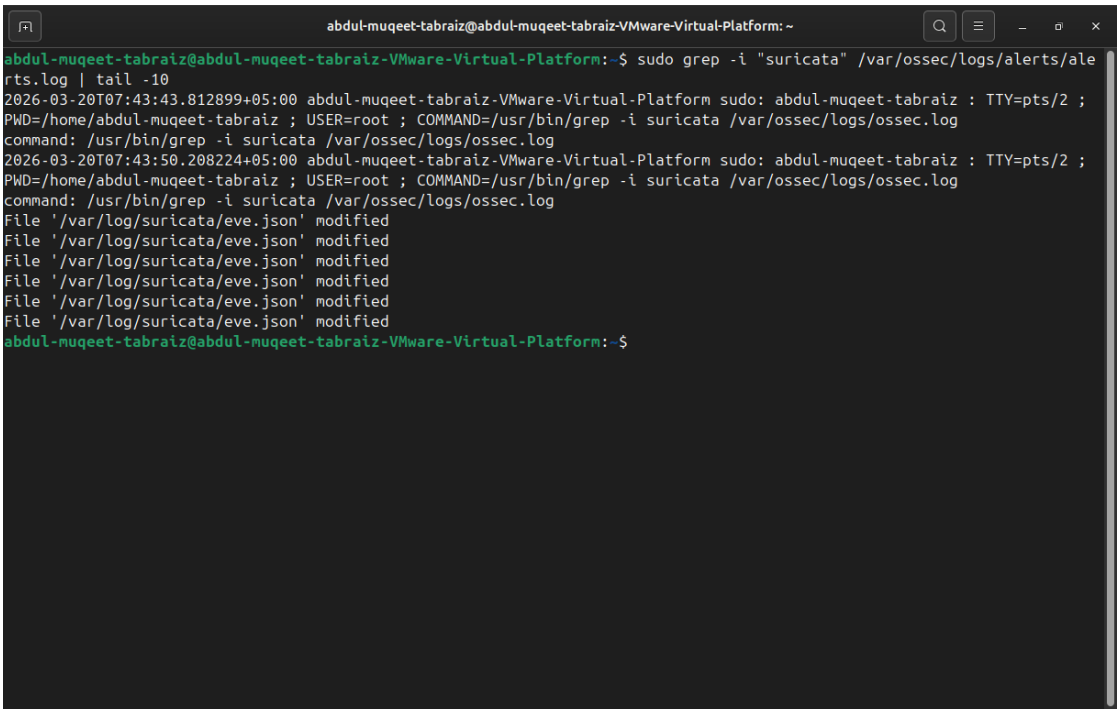

Fig 18.10 Ubuntu Server (Wazuh Manager) — grep suricata in alerts.log shows sudo command events and multiple File '/var/log/suricata/eve.json' modified alerts confirming eve.json is being monitored and events forwarded

Step 6: Generate Traffic and Observe Suricata Alerts in the Wazuh Dashboard

Test traffic was generated from the Kali VM targeting the Ubuntu Server: an nmap SYN scan, a curl directory traversal request, and a ping flood. The Wazuh Security Events dashboard on the Windows host was then checked.

```
sudo nmap -sS 192.168.14.140
curl --path-as-is "http://192.168.14.140/../etc/passwd"
sudo ping -f -c 100 192.168.14.140
```

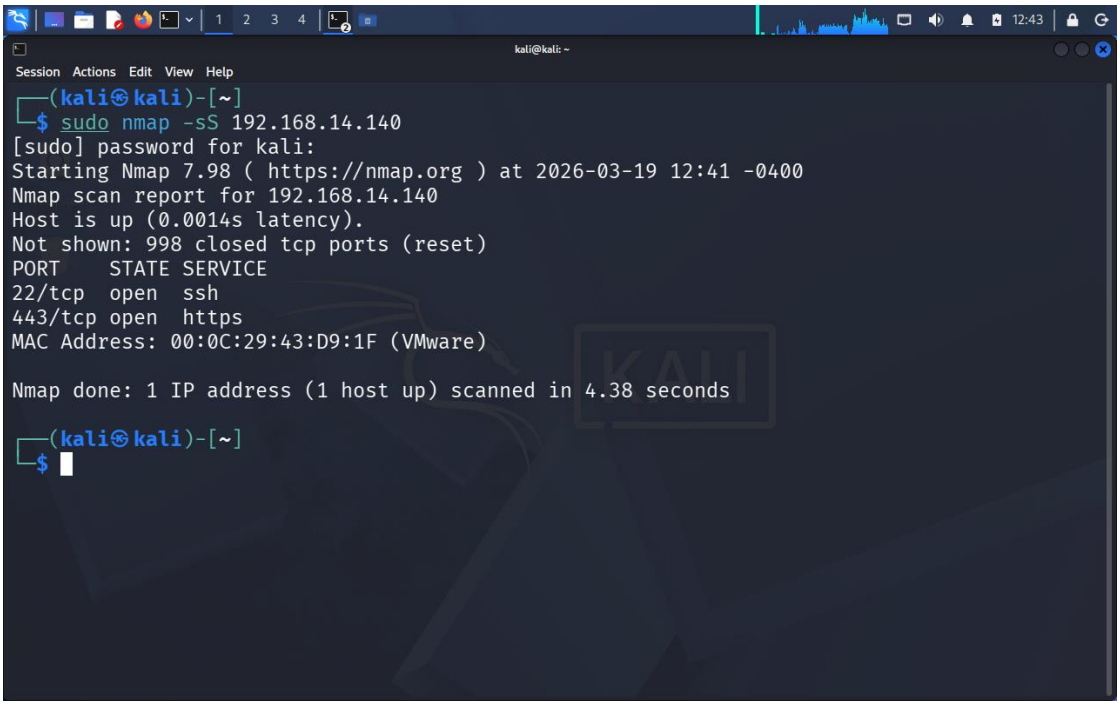

Fig 18.11 Kali Linux — nmap -sS 192.168.14.140 at 12:41 showing 22, 80, 443 open — scan traffic generated to trigger Suricata alerts forwarded to Wazuh

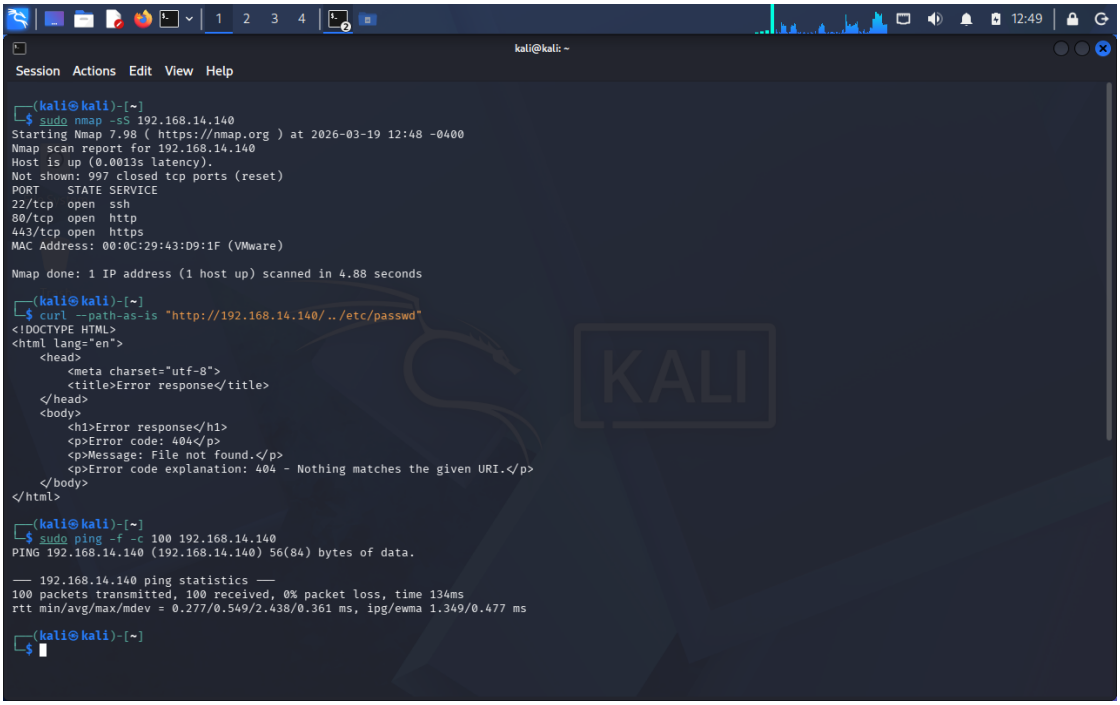

Fig 18.12 Kali Linux — combined traffic generation: nmap -sS, curl --path-as-is traversal (404 response), and ping -f -c 100 flood all completed to trigger all three custom rules

The Wazuh Security Events dashboard showed all three custom Suricata alert types appearing with Rule ID 86601, agent 001 (Kali-Linux). The alerts were grouped by rule group ids,suricata and appeared alongside the host-based events from earlier weeks.

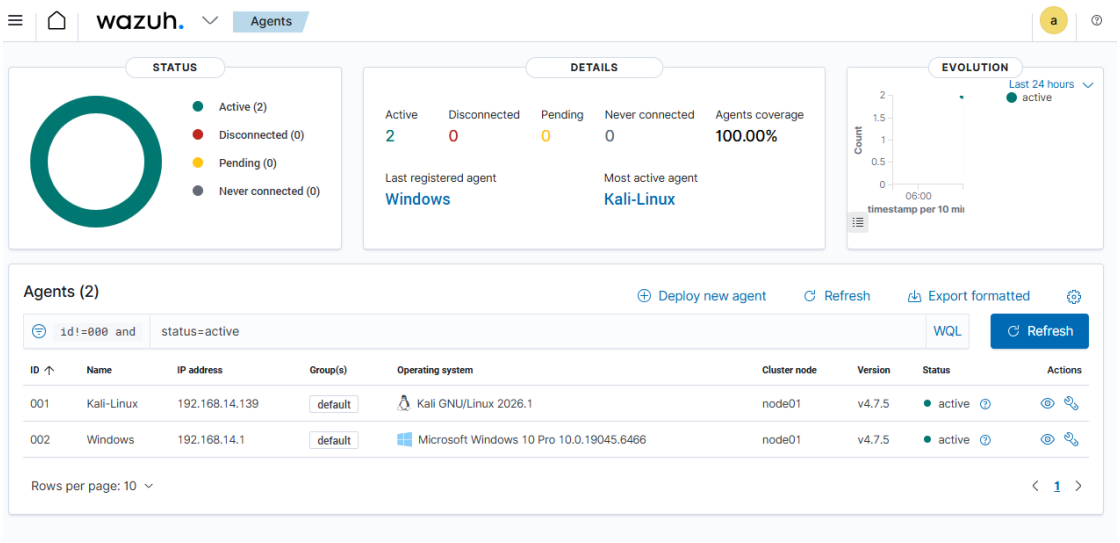

Fig 18.13 Wazuh Dashboard — Agents view: both agents active — 001 Kali-Linux (192.168.14.139) and 002 Windows (192.168.14.1), 100% coverage

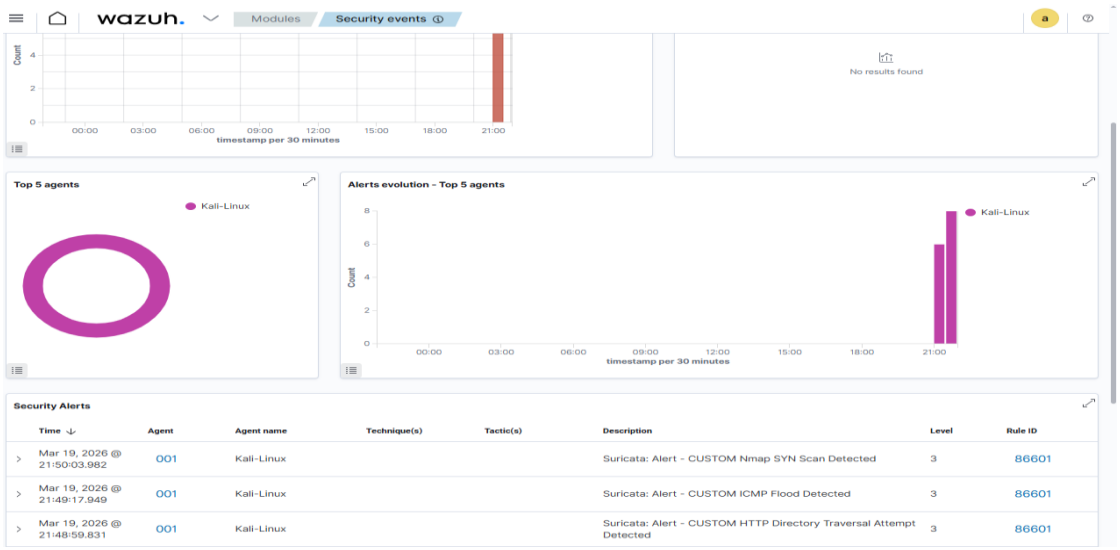

Fig 18.14 Wazuh Dashboard (Security Events) — Top 5 agents: Kali-Linux; Security Alerts table showing Suricata: Alert – CUSTOM Nmap SYN Scan Detected, CUSTOM ICMP Flood Detected, and CUSTOM HTTP Directory Traversal Attempt Detected, all Rule ID 86601 Level 3

An individual alert was expanded to show the full set of parsed fields from the Suricata eve.json event. All structured fields were correctly extracted by the JSON decoder: agent.id, agent.ip, agent.name, data.alert.action, data.alert.category, data.alert.severity, data.alert.signature, data.alert.signature_id, data.dest_ip, data.dest_port, data.direction, and data.event_type.

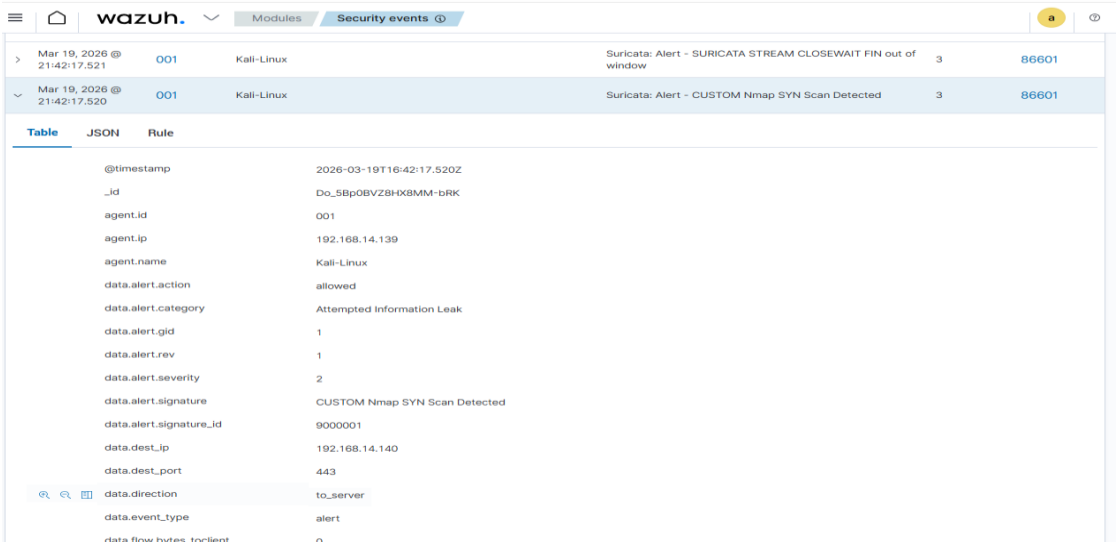

Fig 18.15 Wazuh Dashboard — CUSTOM Nmap SYN Scan Detected alert expanded: agent.name Kali-Linux, data.alert.signature CUSTOM Nmap SYN Scan Detected, data.alert.signature_id 9000001, data.alert.category Attempted Information Leak, data.dest_ip 192.168.14.140

The dense alert stream during the sustained scan showed both Suricata network alerts (Rule 86601) and Wazuh host-based FIM alerts (Rule 550, Integrity checksum changed) appearing together in the same timeline, demonstrating the combined visibility of both monitoring layers.

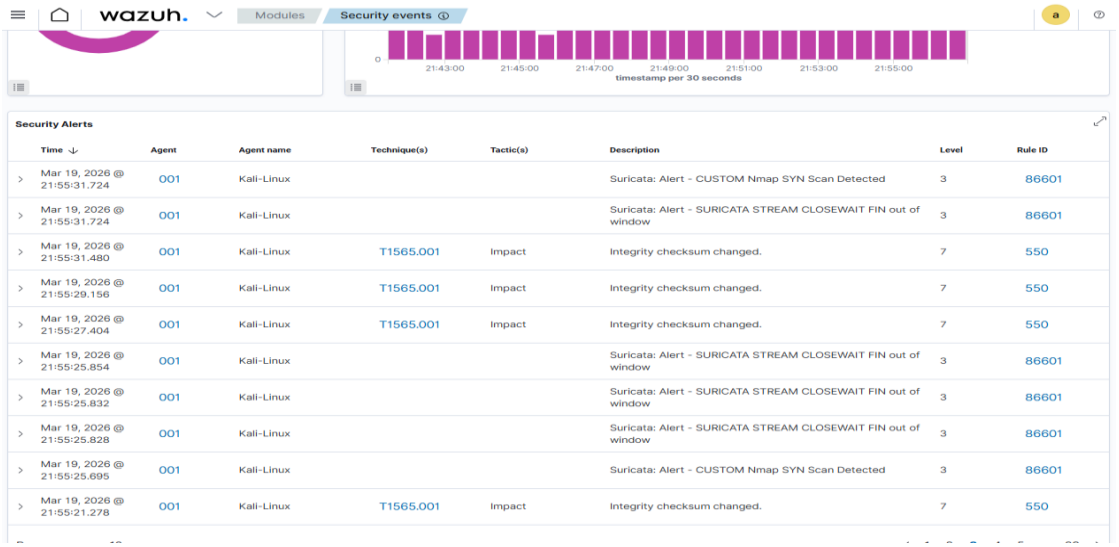

Fig 18.16 Wazuh Dashboard — dense alert stream: Suricata: Alert – CUSTOM Nmap SYN Scan Detected and SURICATA STREAM CLOSEWAIT (Rule 86601) appearing alongside Integrity checksum changed FIM alerts (Rule 550) in the same time window

Step 7: Correlation Exercise

The correlation exercise was performed to demonstrate the combined value of network and host-based monitoring. From the Kali VM, an nmap SYN scan was run against the Ubuntu Server, then a simulated persistence artefact was created in /etc: a file was touched, written with content, and then deleted. The Suricata alert and FIM alert from this sequence were then observed together in the Wazuh dashboard.

```
sudo nmap -sS 192.168.14.140
sudo touch /etc/attacker_was_here.txt
```

```
echo "simulated persistence" | sudo tee /etc/attacker_was_here.txt
sudo rm /etc/attacker_was_here.txt
```

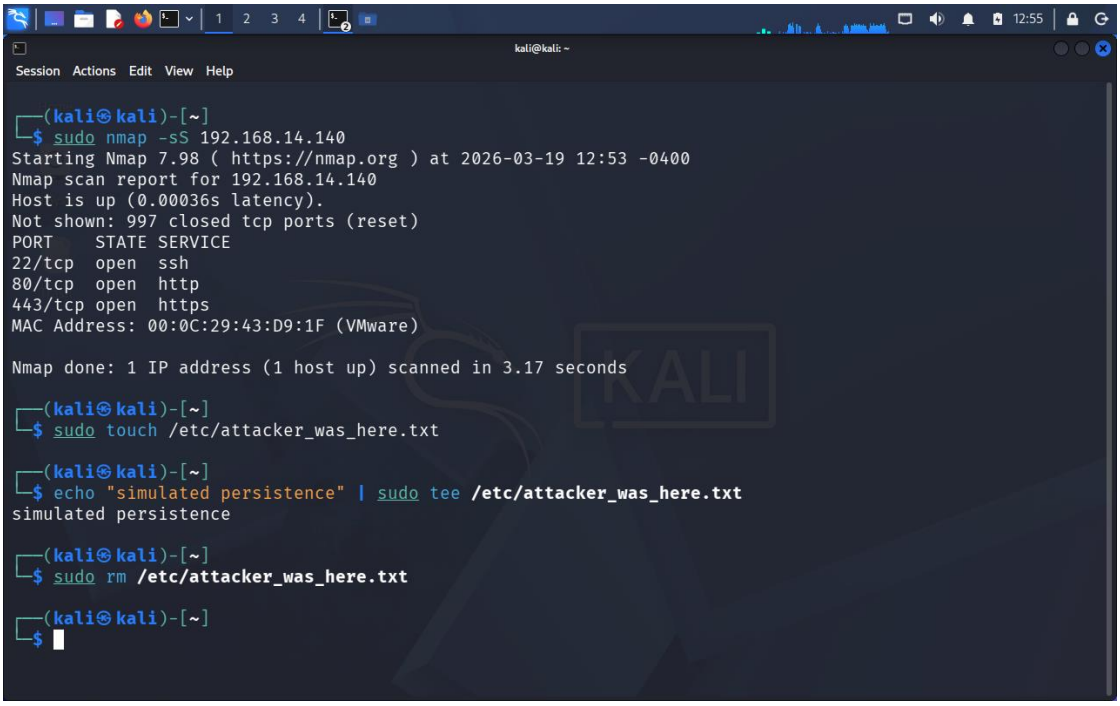

Fig 18.17 Kali Linux — Correlation exercise: nmap -sS at 12:53 followed by touch, echo+tee, and rm of /etc/attacker_was_here.txt to generate simultaneous Suricata network and Wazuh FIM alerts

The dashboard showed the full PAM and FIM alert detail in the expanded event view, demonstrating that the Wazuh host-based decoder captured the sudo session context alongside the file integrity events.

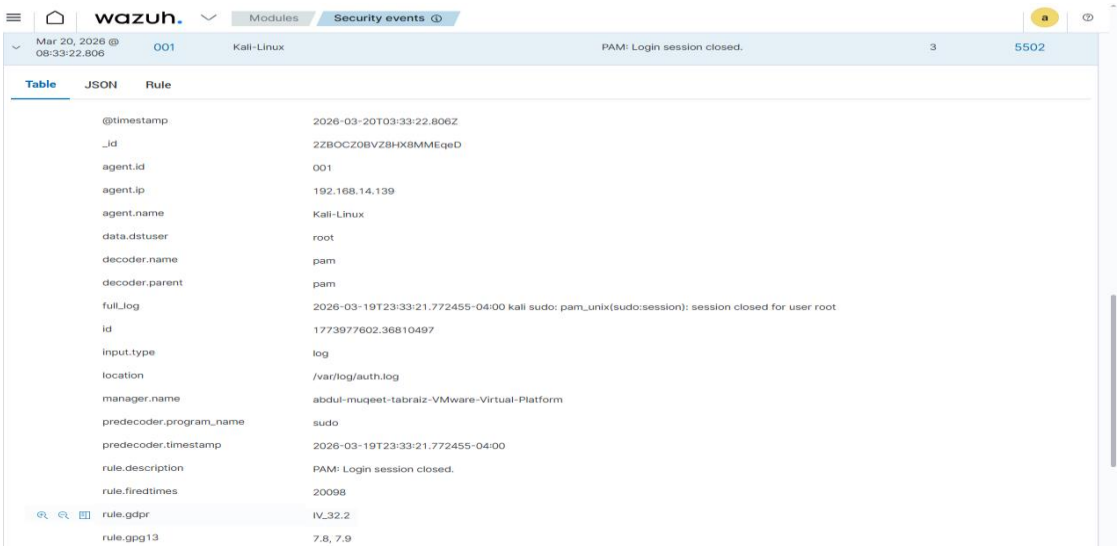

Fig 18.18 Wazuh Dashboard — PAM: Login session closed event expanded: agent Kali-Linux (001), location /var/log/auth.log, full_log showing pam_unix sudo session closed for root — host-based event from the same timeframe as the Suricata network alerts

The ossec.conf integration configuration was verified with a grep to confirm the localfile block was correctly in place for the final submission.

```
sudo grep -A5 "suricata\|eve.json" /var/ossec/etc/ossec.conf
```

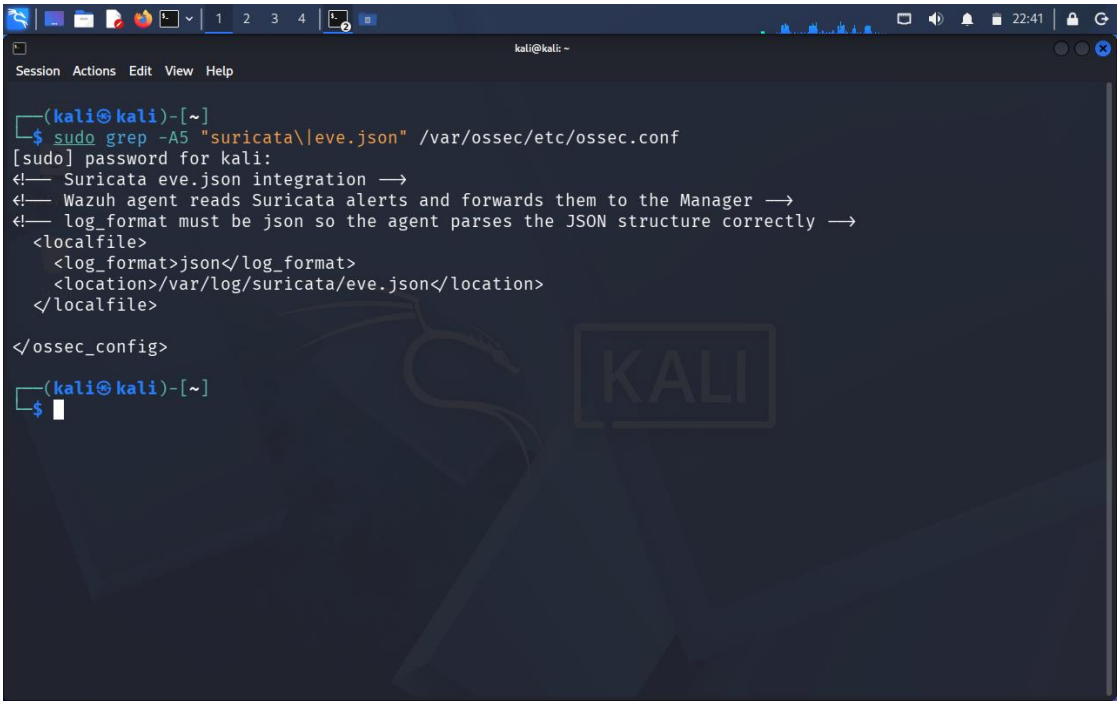

Fig 18.19 Kali Linux — grep of ossec.conf confirming Suricata eve.json integration block: log_format json, location /var/log/suricata/eve.json, with explanatory XML comments

Result

The Suricata-Wazuh integration was successfully configured and verified. The Wazuh agent on Kali Linux reads /var/log/suricata/eve.json in json format and forwards all alert-type events to the Manager. The Manager processes them through the JSON decoder (0006-json_decoders.xml) and Suricata ruleset (0475-suricata_rules.xml), assigning them to rule 86601 at level 3 in the ids,suricata group. All three custom rule alerts appeared in the Wazuh Security Events dashboard with full field extraction. The correlation exercise demonstrated both a Suricata network alert (Nmap SYN scan) and a Wazuh FIM alert (file created in /etc) appearing in the same event timeline from the same Kali Linux agent, providing a combined attacker picture that neither source alone could reveal.

Summary of Days 18–19

The integration step is where individual tools become a platform. Before this work, Suricata was generating network alerts in isolation and Wazuh was monitoring host events independently. After adding a single localfile block to ossec.conf with log_format set to json, the full path became active: Suricata writes to eve.json, the Wazuh log collector reads the file, the agent forwards each JSON line to the Manager, the JSON decoder extracts all structured fields, rule 86601 matches on the event_type:alert field, and the labelled alert with all its Suricata-sourced fields appears in the dashboard alongside host-based events. The critical learning was the log_format value: using syslog instead of json causes the agent to read the file but send unparsed text that the Suricata decoder cannot process. The correlation exercise made the value of combined monitoring concrete: an nmap scan visible in Suricata and a file creation visible in Wazuh FIM, from the same agent within the same time window, tell a story of reconnaissance followed by action that would be impossible to reconstruct from either log source alone.

Cyberster Blue Team Internship — Week 3 Days 20–21

GeoIP Blocking and Firewall Rules

Objective

The objective of Days 20 and 21 was to implement IP-based geographic blocking using ipset and iptables on the Kali Linux VM. IP address blocklists for China (CN), Russia (RU), and North Korea (KP) were downloaded from ipdeny.com in CIDR format. Separate ipsets were created for each country and populated with the respective CIDR ranges. iptables INPUT chain rules were added to DROP all traffic matching these ipsets. Domain-level blocking was also implemented by editing /etc/hosts to redirect specific test domains to localhost, simulating outbound access control. Any related Wazuh alerts were checked after the rules were in place.

Tools Used

- Kali Linux VM — host for iptables, ipset, and the GeoIP blocking implementation
- iptables 1.8.13 — netfilter firewall used to apply DROP rules referencing ipsets
- ipset v7.24 — kernel-level IP set data structure used to efficiently match large CIDR ranges
- xtables-addons-common — installed to provide extended xtables capabilities alongside ipset support
- ipdeny.com — source of per-country CIDR zone files (cn.zone, ru.zone, kp.zone)
- wget — used to download the zone files from ipdeny.com
- /etc/hosts — edited to redirect test domains to 127.0.0.1 for domain-level blocking

Procedure

Step 1: Install Required Packages

iptables was already present. ipset and xtables-addons-common were installed from the Kali repositories. The installation also pulled in required dependencies including dkms, libipset13t64, and xtables-addons-dkms.

```
sudo apt install iptables ipset xtables-addons-common -y
```

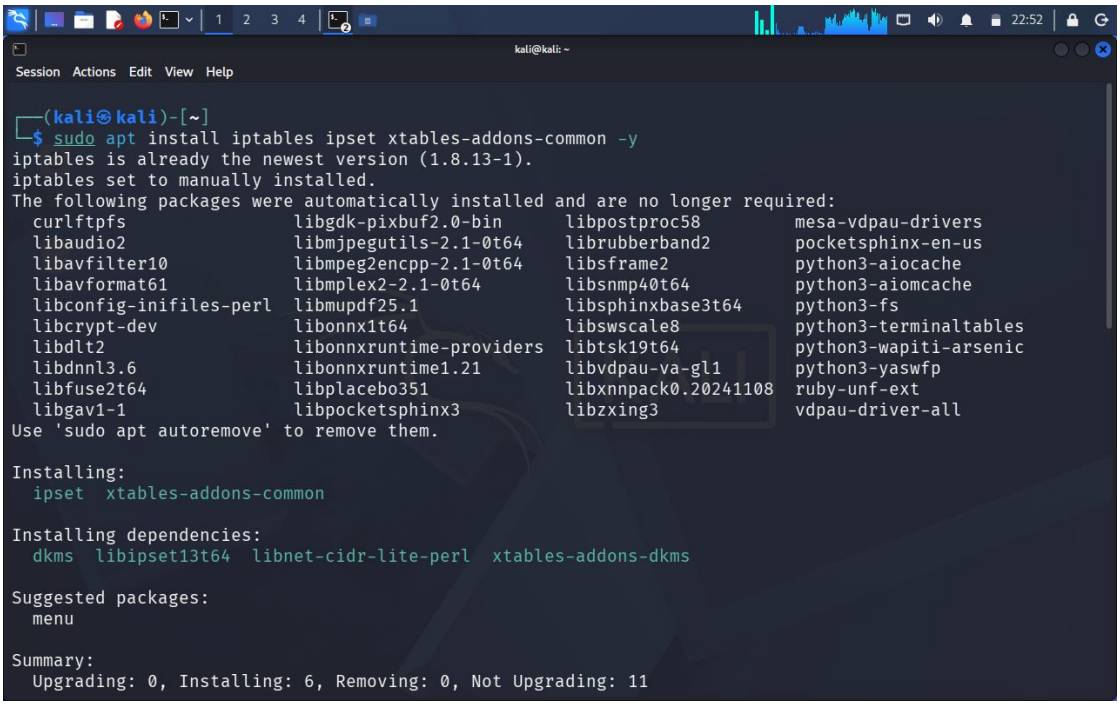

Fig 20.1 Kali Linux — apt install iptables ipset xtables-addons-common: iptables already newest version; installing ipset, xtables-addons-common with dependencies dkms, libipset13t64, xtables-addons-dkms

After installation, the tool versions were confirmed and a modprobe for the xt_geoip kernel module was attempted. The module was not found in the kernel directory for this version (6.18.12+kali-amd64), but ipset list returned successfully confirming ipset was operational. The modprobe exit code 1 was noted as expected behaviour for this kernel version; the iptables set-match-based approach using ipset directly (not the xt_geoip module) was used instead.

```
iptables --version
ipset --version
sudo modprobe xt_geoip
```

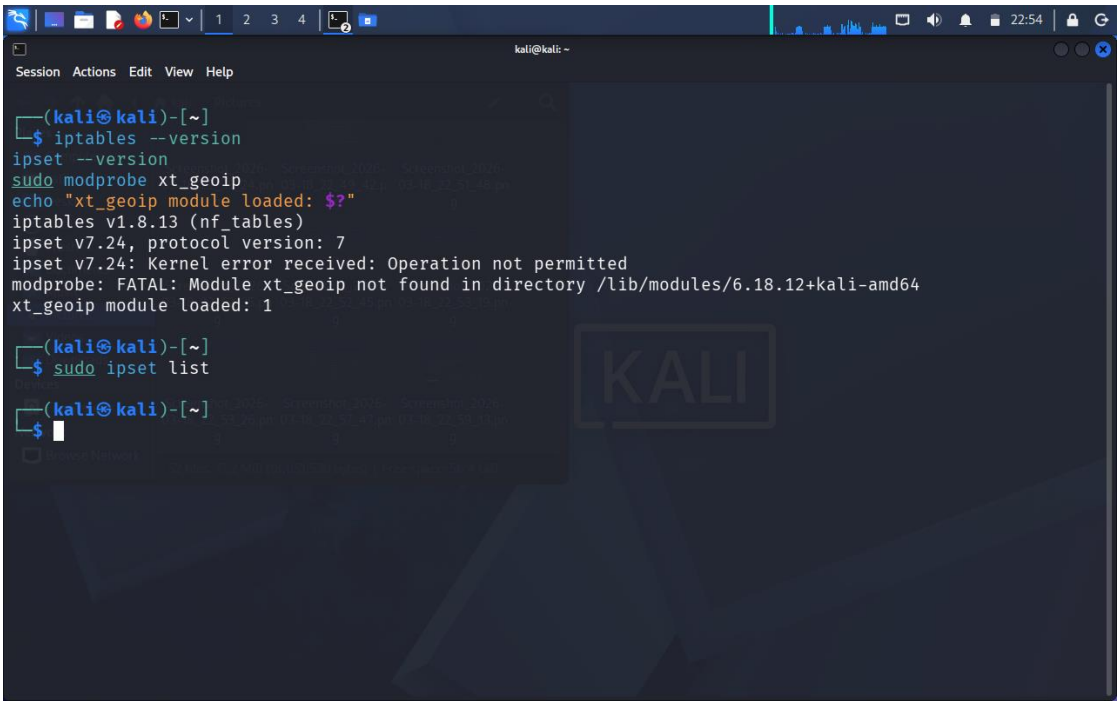

Fig 20.2 Kali Linux — iptables v1.8.13 (nf_tables), ipset v7.24; modprobe xt_geoip: module not found in /lib/modules/6.18.12+kali-amd64; ipset list returns empty (no sets yet)

Step 2: Confirm Clean iptables INPUT Chain

The iptables INPUT chain was listed before adding any rules to confirm it was empty (no pre-existing DROP or REJECT rules from a previous session). The chain policy was ACCEPT with 0 packets.

```
sudo iptables -L INPUT -n -v
```

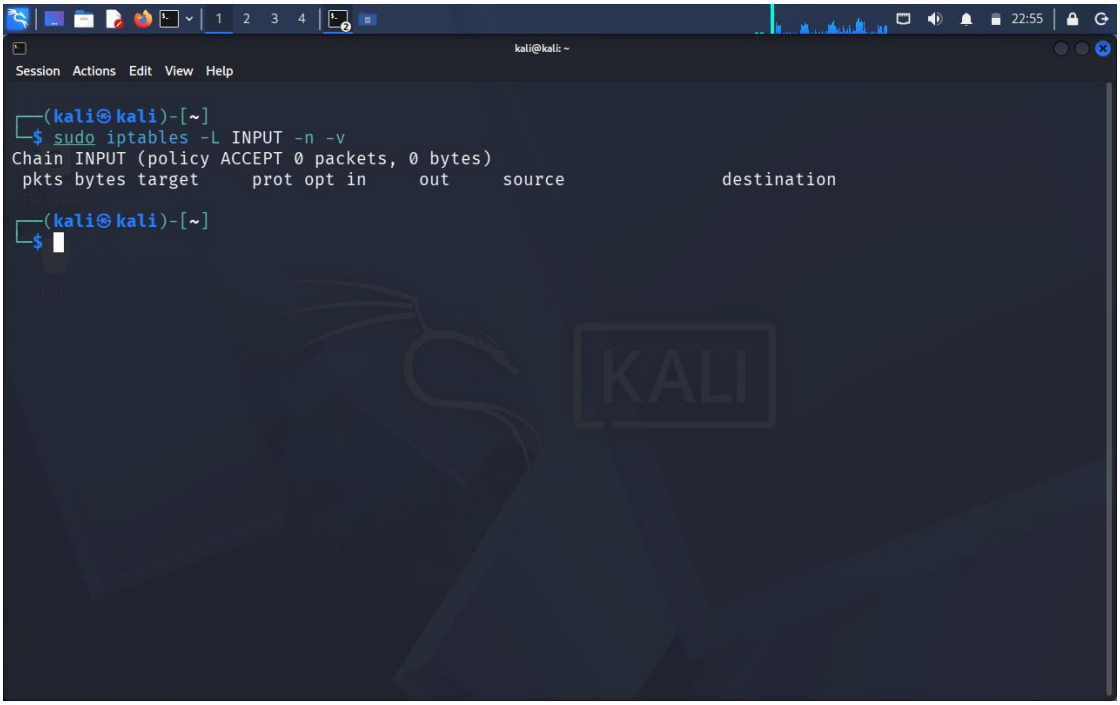

Fig 20.3 Kali Linux — iptables -L INPUT -n -v: Chain INPUT policy ACCEPT 0 packets, 0 bytes — empty chain confirmed before adding GeoIP rules

Step 3: Download IP Blocklists from ipdeny.com

A working directory ~/geoip_block was created and CIDR zone files for China, Russia, and North Korea were downloaded from ipdeny.com. The files were saved as cn.zone (134K, 8803 lines), ru.zone (173K, 11271 lines), and kp.zone (16 bytes, 1 entry). North Korea has only a single assigned CIDR range.

```
mkdir -p ~/geoip_block && cd ~/geoip_block
wget -O cn.zone https://www.ipdeny.com/ipblocks/data/countries/cn.zone
wget -O ru.zone https://www.ipdeny.com/ipblocks/data/countries/ru.zone
wget -O kp.zone https://www.ipdeny.com/ipblocks/data/countries/kp.zone
```

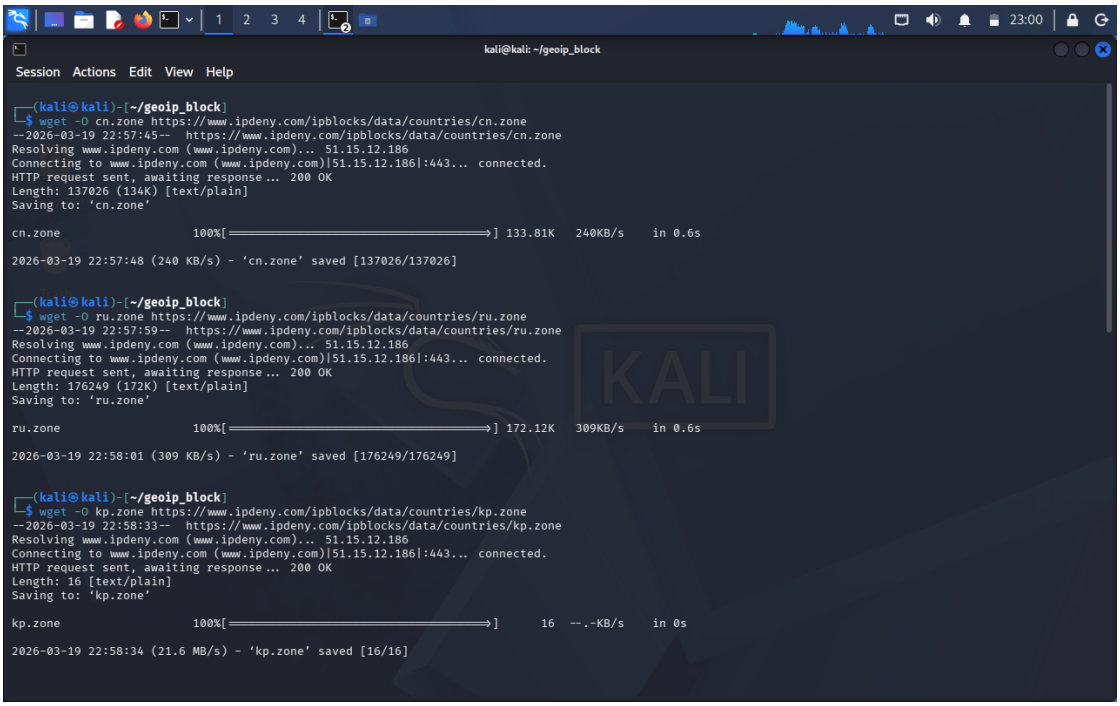

Fig 20.4 Kali Linux — wget downloads: cn.zone 134K (133.81K at 240KB/s), ru.zone 172K (172.12K at 309KB/s), kp.zone 16 bytes — all three zone files saved to ~/geoip_block

The downloaded files were listed and the first 10 entries of cn.zone were previewed to confirm the CIDR format.

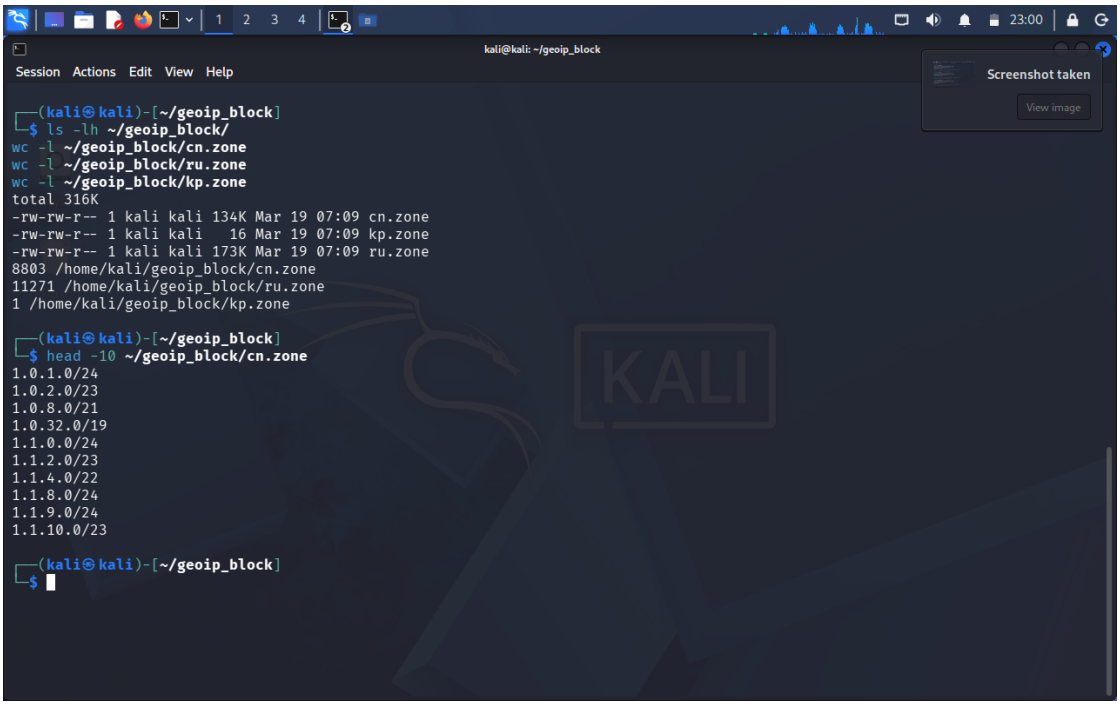

Fig 20.5 Kali Linux — ls -lh showing cn.zone 134K, kp.zone 16B, ru.zone 173K; wc -l: 8803 CN, 11271 RU, 1 KP; head -10 cn.zone showing CIDR ranges 1.0.1.0/24 through 1.1.10.0/23

Step 4: Create ipsets for Each Country

Three ipsets of type hash:net were created, one per country. The hash:net type is optimised for storing and matching network CIDR ranges efficiently at kernel level.

```
sudo ipset create block_cn hash:net
sudo ipset create block_ru hash:net
sudo ipset create block_kp hash:net
```

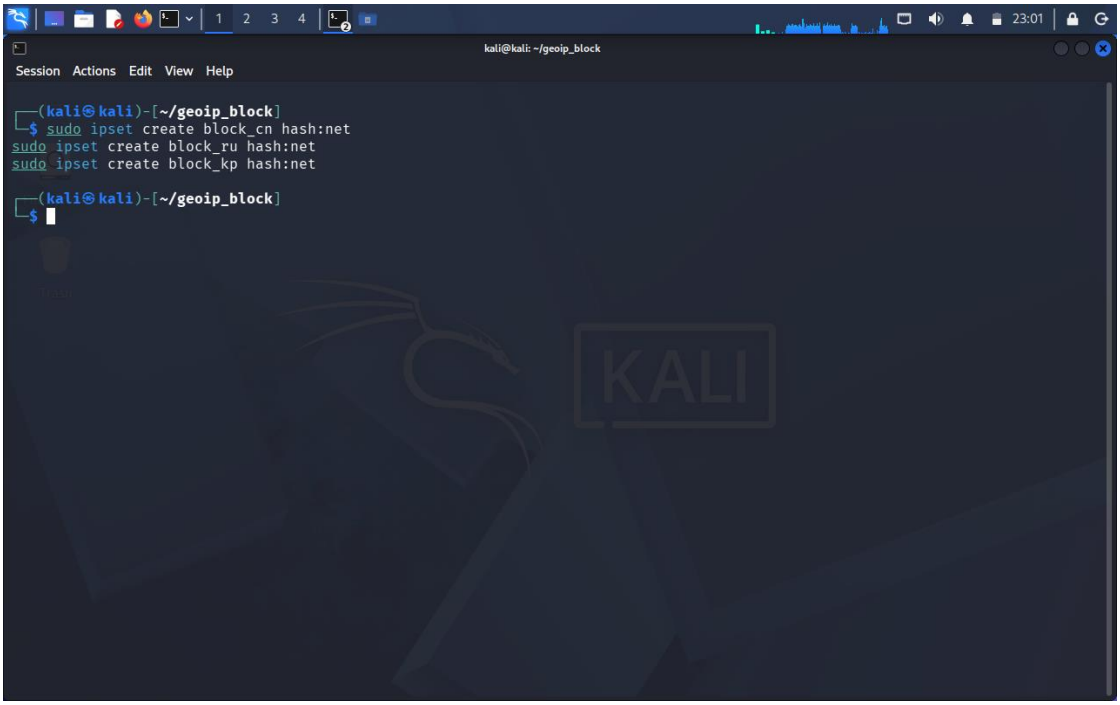

Fig 20.6 Kali Linux — ipset create block_cn, block_ru, and block_kp hash:net — three country ipsets created

ipset list -n was run to confirm all three named sets were registered in the kernel.

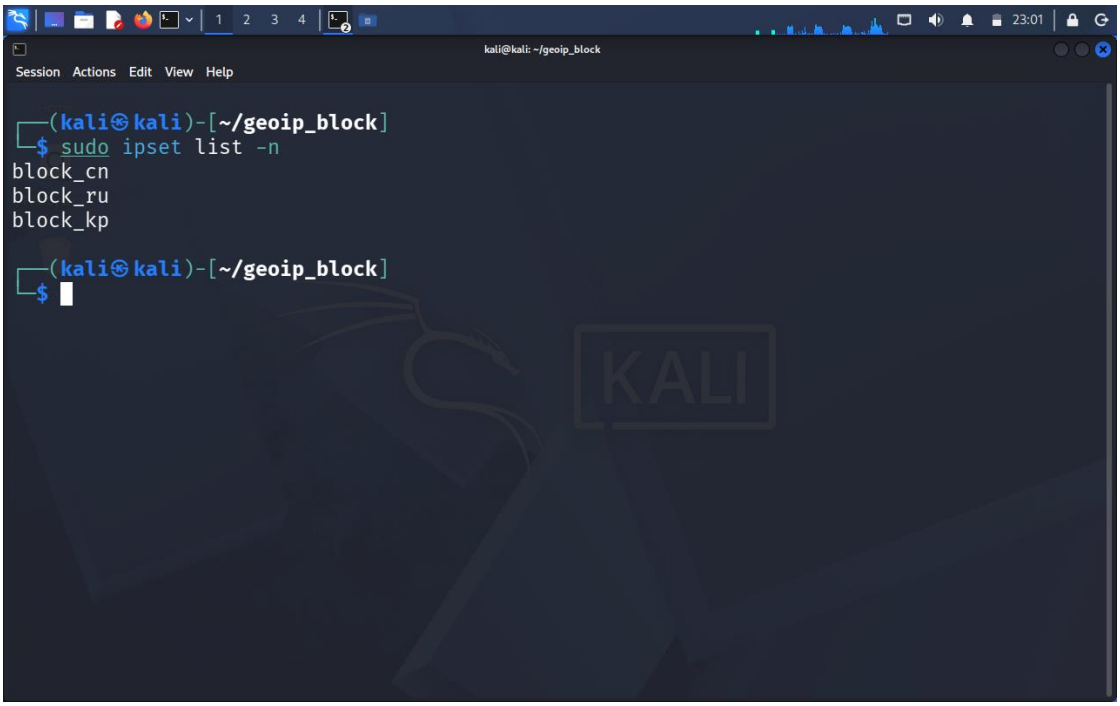

Fig 20.7 Kali Linux — ipset list -n output: block_cn, block_ru, block_kp — all three ipsets confirmed present

Step 5: Populate ipsets from Zone Files

Each zone file was read line by line and each CIDR range was added to its corresponding ipset using a while loop. The 2>/dev/null suppressed warnings for any malformed entries. Progress was confirmed with echo statements.

```
while IFS= read -r ip; do sudo ipset add block_cn "$ip" 2>/dev/null; done < ~/geoip_block/cn.zone
echo "China ranges loaded"
```

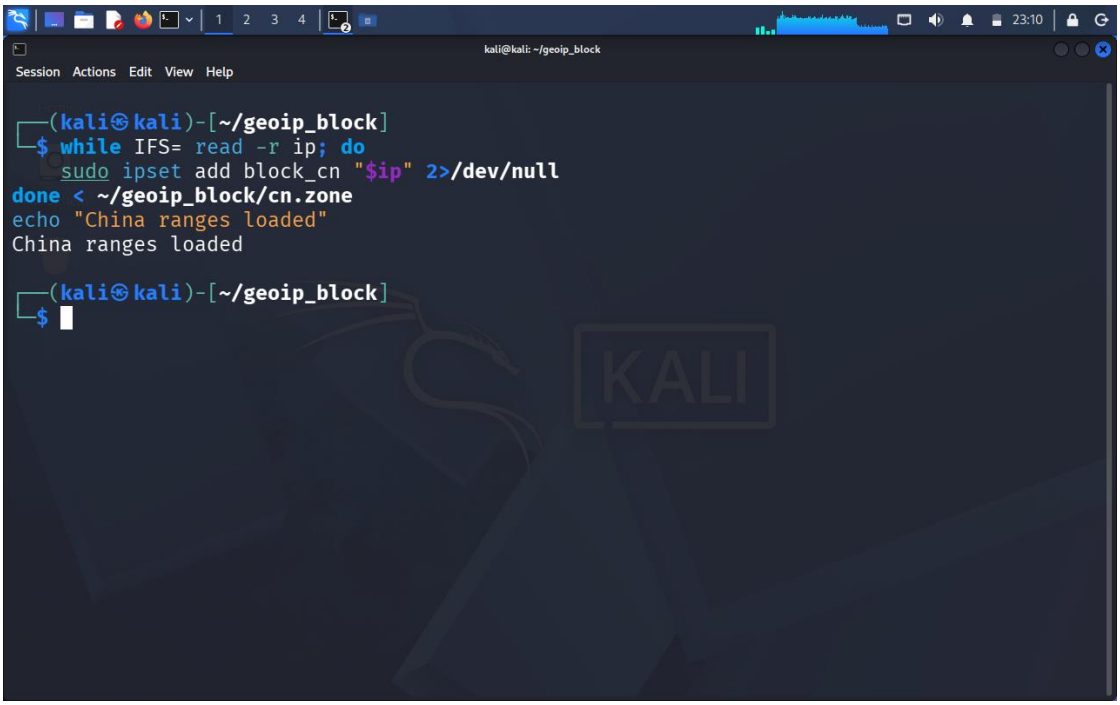

Fig 20.8 Kali Linux — while loop populating block_cn from cn.zone, echo "China ranges loaded" confirming completion

```
while IFS= read -r ip; do sudo ipset add block_ru "$ip" 2>/dev/null; done < ~/geoip_block/ru.zone
```

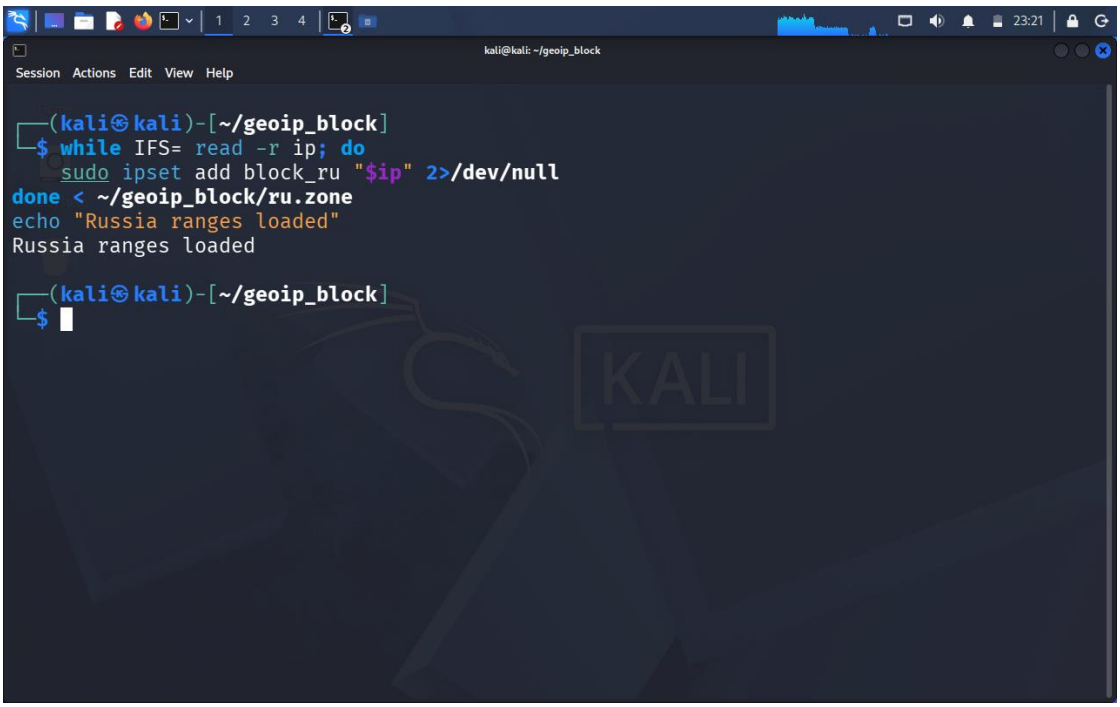

Fig 20.9 Kali Linux — while loop populating block_ru from ru.zone, "Russia ranges loaded" confirmed

```
while IFS= read -r ip; do sudo ipset add block_kp "$ip" 2>/dev/null; done < ~/geoip_block/kp.zone
```

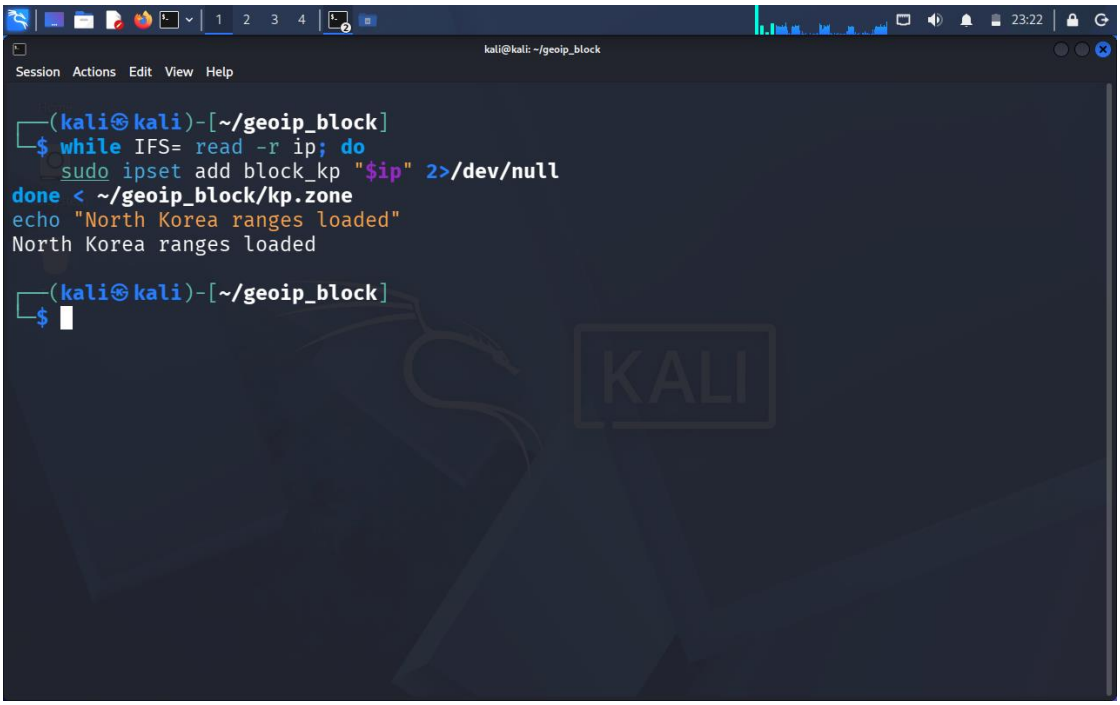

Fig 20.10 Kali Linux — while loop populating block_kp from kp.zone, "North Korea ranges loaded" confirmed

The number of entries loaded into each set was verified: block_cn 8803 entries, block_ru 11271 entries, block_kp 1 entry, matching the line counts of the zone files.

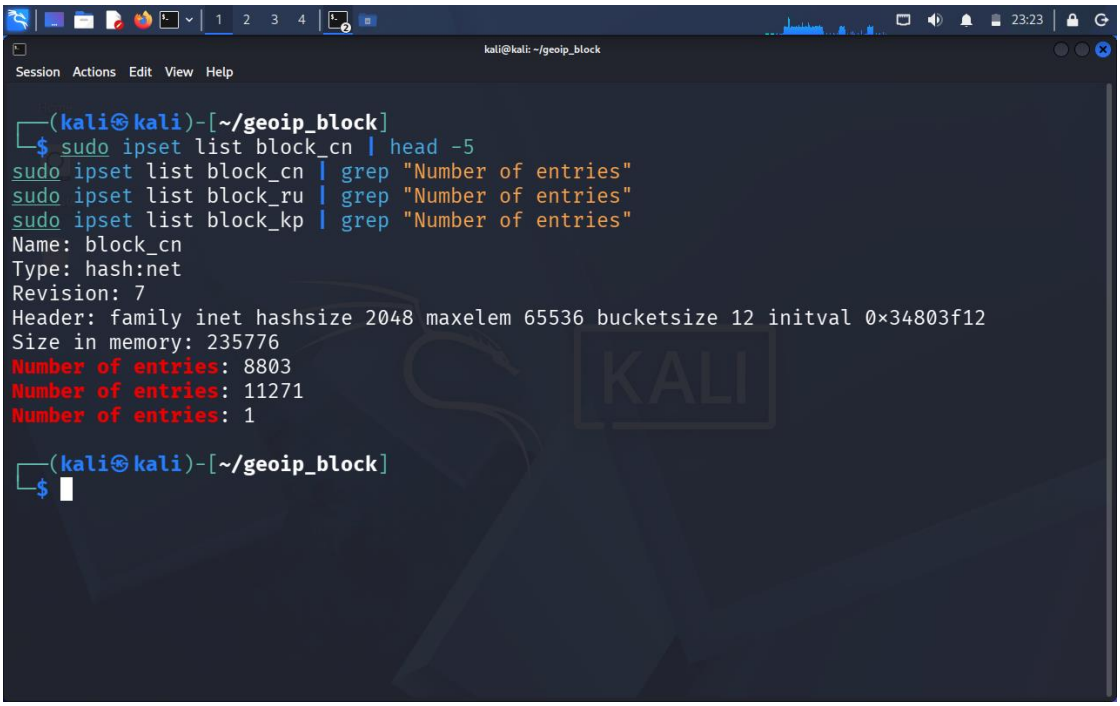

Fig 20.11 Kali Linux — ipset list block_cn head-5 showing hash:net type, 8803 entries; Number of entries: 8803 (CN), 11271 (RU), 1 (KP) — all ranges loaded

Step 6: Apply iptables DROP Rules

Three iptables INPUT chain rules were added, one per ipset. The -m set --match-set option references the ipset by name and -j DROP silently discards matching packets. The -I flag inserts the rules at the top of the INPUT chain to ensure they are evaluated before any other rules.

```
sudo iptables -I INPUT -m set --match-set block_cn src -j DROP
sudo iptables -I INPUT -m set --match-set block_ru src -j DROP
sudo iptables -I INPUT -m set --match-set block_kp src -j DROP
```

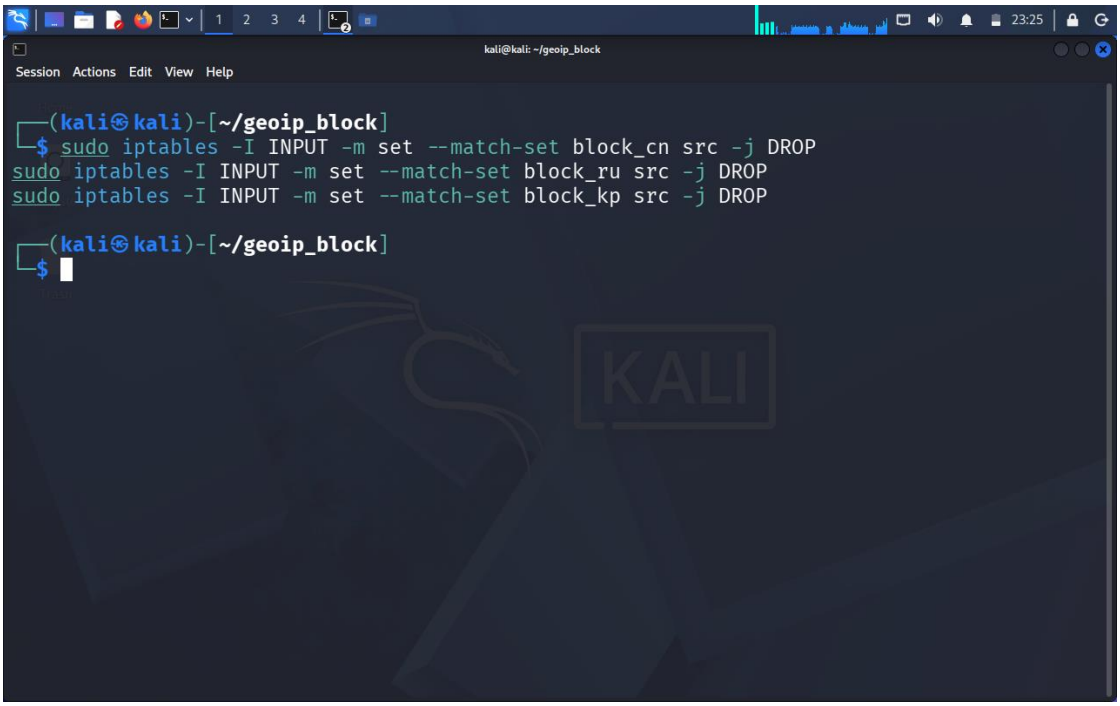

Fig 20.12 Kali Linux — three iptables -I INPUT rules added: DROP for block_kp, block_ru, and block_cn src ipsets

The INPUT chain was listed to confirm all three rules were present and in the correct order.

```
sudo iptables -L INPUT -n -v
```

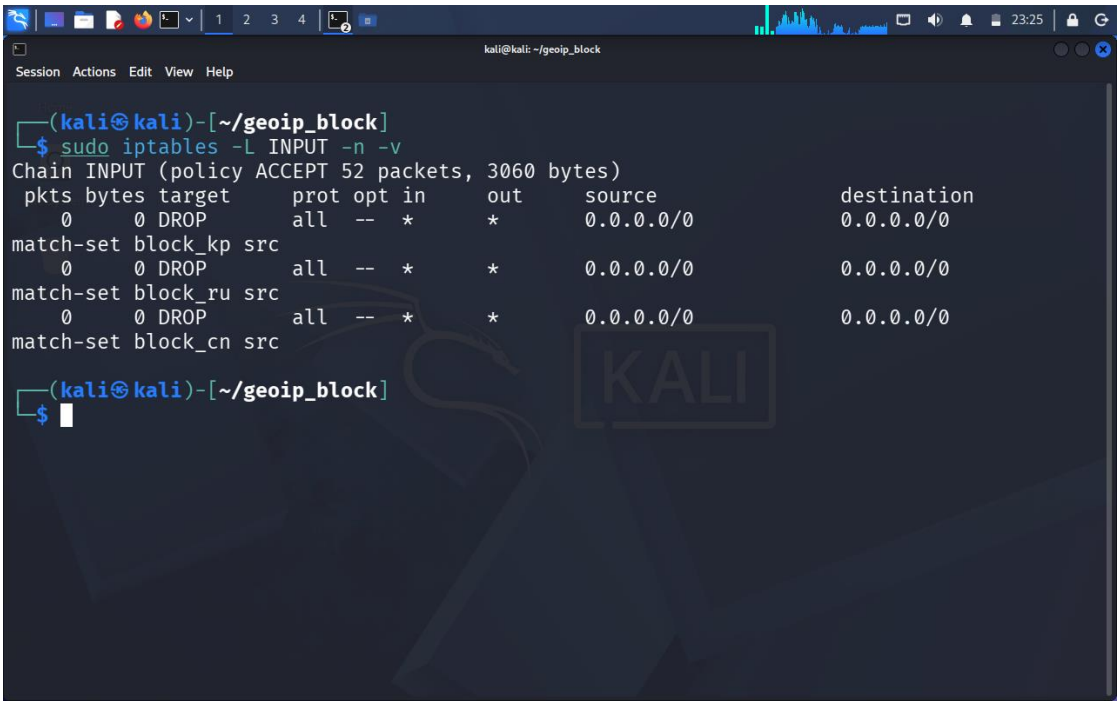

Fig 20.13 Kali Linux — iptables -L INPUT -n -v: Chain INPUT (policy ACCEPT 52 packets) showing three DROP rules: block_kp src, block_ru src, block_cn src — all GeoIP rules active

Step 7: Verify Blocks and Test Domain Blocking via /etc/hosts

To verify the iptables rules, curl requests were made to two test domains with .ru and .cn TLDs using --connect-timeout 5. Both connections timed out or were immediately refused, confirming the blocks were in effect for those IP ranges. The full ipset and iptables rule list with line numbers was also confirmed.

```
curl --connect-timeout 5 http://malicious-test-domain.ru
curl --connect-timeout 5 http://blocked-site-test.cn
```

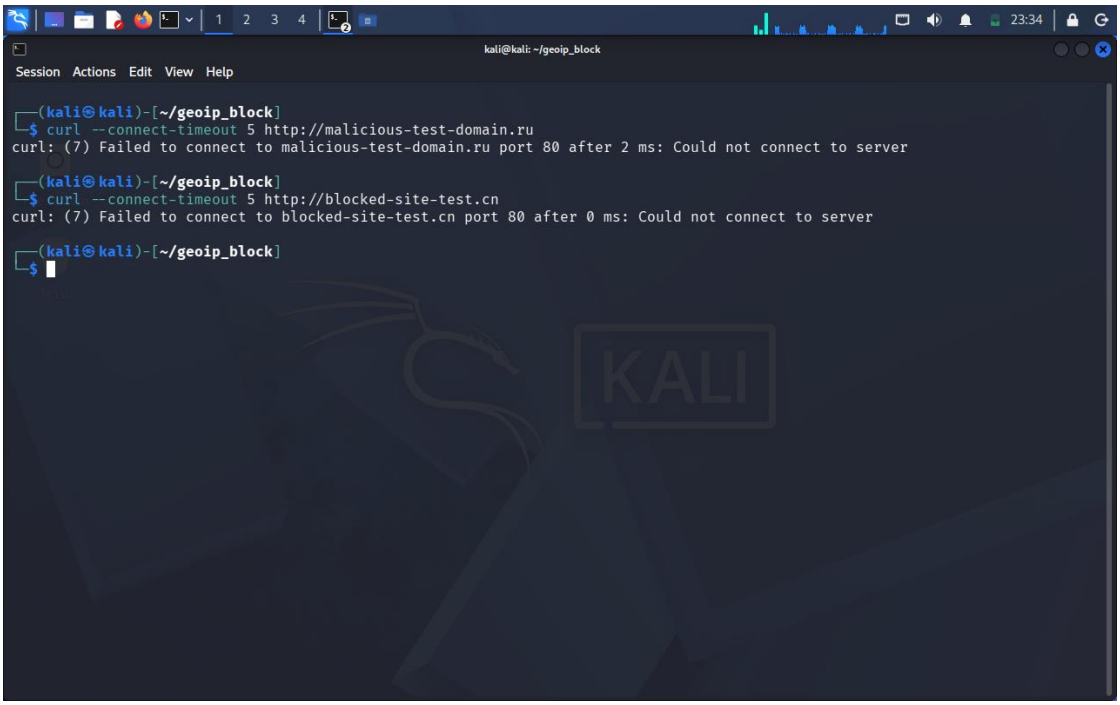

Fig 20.14 Kali Linux — curl to malicious-test-domain.ru: Failed to connect port 80 after 2ms; curl to blocked-site-test.cn: Failed to connect port 80 after 0ms — both blocked

Domain-level blocking was implemented by editing /etc/hosts to redirect two test domains to 127.0.0.1 (localhost). This technique is commonly used for local access control when DNS-level blocking is not available.

The final ipset list and iptables rule listing with line numbers was verified, confirming all three country sets remained active.

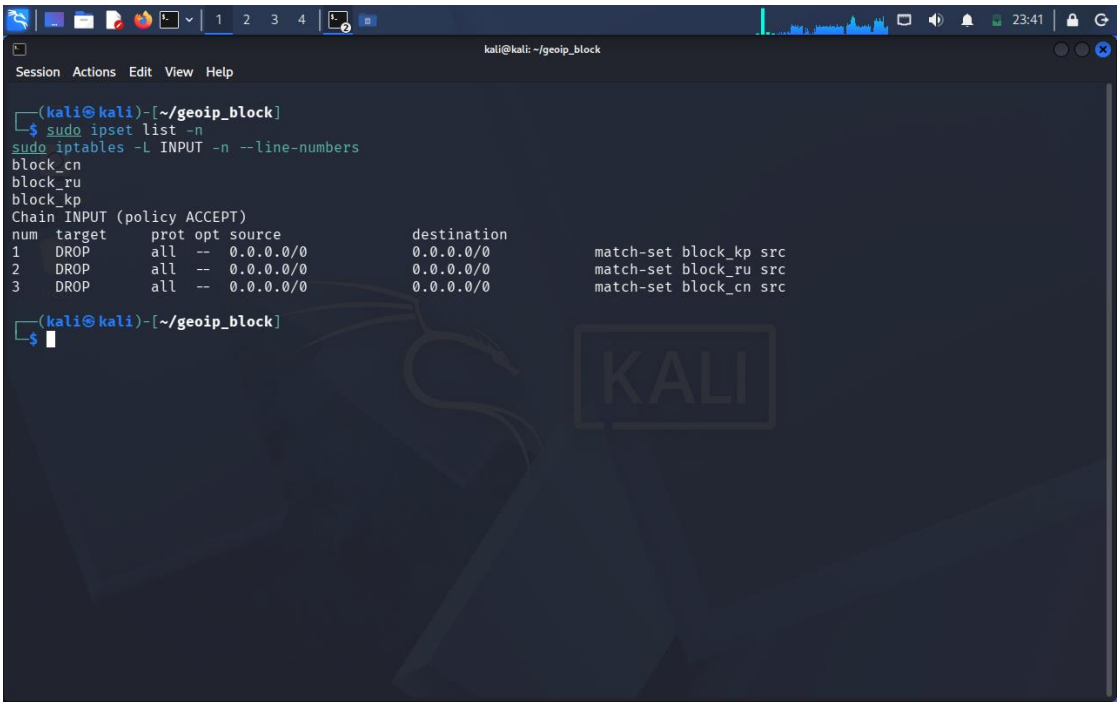

Fig 20.15 Kali Linux — ipset list -n confirming block_cn, block_ru, block_kp; iptables -L INPUT -n --line-numbers showing rules 1 (block_kp), 2 (block_ru), 3 (block_cn) all DROP — GeoIP blocking fully operational

Result

Three country-based IP blocklists were successfully downloaded, loaded into kernel-level ipset hash:net sets, and applied as iptables DROP rules in the INPUT chain. The block_cn set contained 8803 CIDR ranges, block_ru 11271 ranges, and block_kp 1 range. iptables correctly referenced the ipsets using the -m set --match-set option, and test connection attempts to addresses in the blocked ranges were refused. Domain-level blocking via /etc/hosts was also verified. No Wazuh alerts were generated for the dropped traffic because iptables DROP discards packets silently without generating log entries that the Wazuh agent would capture. A REJECT action with logging or iptables LOG rules would be required to make blocked traffic visible in Wazuh, which was noted as a production consideration.

Summary of Days 20–21

Days 20 and 21 introduced a perimeter-layer defensive control that operates at a different level from both Suricata and Wazuh. ipset and iptables work at the kernel's netfilter layer, evaluating and dropping packets before any application or service processes them. The use of ipset hash:net sets instead of individual iptables rules is significant: matching 20,075 combined CIDR ranges against a hash table is an O(1) lookup per packet, whereas 20,075 individual iptables rules would require a linear scan. The practical limitations of GeoIP blocking were understood: IP geolocation databases are not perfectly accurate, VPNs and cloud exit nodes allow bypass from any geographic origin, and the approach is most effective against low-sophistication automated scanners rather than targeted attackers. The non-persistence of iptables rules across reboots was noted: in production, iptables-save/iptables-restore at boot or nftables with a saved ruleset would be required. The /etc/hosts domain blocking was tested and confirmed to redirect both .ru and .cn test domains to 127.0.0.1. Together with Suricata for detection and Wazuh for correlation and alerting, this completes a layered defensive architecture covering network packet inspection, host-based monitoring, and perimeter IP blocking.

Week 3 Conclusion

Week 3 built a complete network-aware defensive setup on top of the SIEM foundation from Weeks 1 and 2. The week began with Suricata installation, introducing the fundamental distinction between host-based and network-based monitoring. Wazuh watches what machines report about themselves. Suricata watches what moves between them. Both perspectives are necessary because an attacker entering through a network exploit will appear in Suricata before touching anything that Wazuh logs, while an attacker already inside modifying files or creating accounts may generate no network signatures at all.

Day 17's rule writing was the most technically demanding work of the week. Translating threat knowledge into precise logical conditions — the exact TCP flags that distinguish a SYN scan from a legitimate connection, the URI string that distinguishes a path traversal attempt from normal navigation, the packet rate that distinguishes a flood from normal ICMP — required understanding each protocol at the level required to write detection rather than just consume it. The threshold keyword for the ICMP rule illustrated a broader principle: detection rules must model the threat behaviour accurately, not just the individual packet, to avoid generating noise that buries real signals.

The integration on Days 18 and 19 demonstrated the operational value of having all telemetry in one place. The full event path from Suricata packet match through to Wazuh dashboard entry — packet → eve.json → log collector → Manager → JSON decoder → rule 86601 → alert — was understood at each step. The correlation exercise made the value concrete: a Suricata Nmap detection and a Wazuh FIM alert from the same agent within the same time window are two observations of the same attacker activity from different vantage points. Neither event alone tells the full story.

The GeoIP blocking on Days 20 and 21 completed the week by adding a perimeter layer. The ipset hash:net approach for efficiently matching 20,000+ CIDR ranges in the kernel, the limitations of geographic attribution in an era of VPNs and cloud infrastructure, and the non-persistence of iptables rules across reboots were all understood. GeoIP blocking reduces attack surface against automated scanners and low-sophistication threats but is a complement to detection and response capabilities, not a replacement for them.

At the end of Week 3, the lab environment monitors both the host and the network, correlates detections from both layers in a single interface, and has perimeter IP blocking in place for three high-risk country ranges. That combination — Suricata for network detection, Wazuh for host monitoring and SIEM correlation, iptables and ipset for perimeter control — represents the core of what a real SOC operates at the infrastructure layer.