

SOC-Use-Cases mit Suricata und Wazuh – Zusammenfassung

1. Einleitung

Diese Zusammenfassung fasst meine bisherigen Arbeiten rund um das Suricata/Wazuh-Projekt zusammen. Die ursprünglichen Dokumente beschreiben mehrere technische Use Cases, detaillierte Testschritte und später auch die Probleme bei der Integration von Wazuh als SIEM. Hier geht es weniger um jedes einzelne Kommando, sondern um das Gesamtbild: Was habe ich aufgebaut, welche Use Cases habe ich getestet, wo gab es Probleme und was habe ich daraus gelernt.

2. Testumgebung und Architektur

Die Lab-Umgebung besteht aus drei wichtigen Komponenten:

- Einer **Kali-Linux-VM** mit **Suricata** als IDS/IPS-Sensor (IP 192.168.178.94).
- Einer **Wazuh-VM** mit dem Wazuh-Docker-Single-Node-Stack (Manager, Indexer, Dashboard) als zentrale SIEM-Plattform (IP 192.168.178.100).
- Einem **Windows-Client**, der sowohl als Angreifer als auch als Opfer dient (PowerShell, SSH, Browser).

Suricata überwacht den Netzwerkverkehr und schreibt seine Events im **JSON-Format** in die Datei `/var/log/suricata/eve.json`. Der Wazuh-Agent auf der Kali-VM liest diese Datei ein und sendet die Events an den Wazuh-Manager. Im Wazuh-Dashboard können die Alerts dann im Bereich **Threat Hunting** gefiltert und analysiert werden.

3. Überblick über die Use Cases

In den Use-Case-Dokumenten habe ich vier Szenarien beschrieben, die aus Sicht eines SOC relevant sind:

1. **SSH-Bruteforce-Angriff** mit Hydra.
2. **Verdächtiger PowerShell-Download** von einer internen HTTP-Quelle.
3. Download einer **Fake-Malware-Datei** (Malware/Trojan Test Download).
4. **Netzwerk-Reconnaissance** mit ICMP-Ping (ICMP Ping Detected).

Alle Use Cases folgen demselben Grundmuster: Ich simuliere einen Angriff oder eine verdächtige Aktion, lasse Suricata den Traffic inspizieren und prüfe anschließend die Alerts – zuerst direkt in der `eve.json`, später zusätzlich im Wazuh-Dashboard.

4. Use Case: SSH-Bruteforce

4.1 Idee und Ziel

Beim SSH-Bruteforce-Use-Case geht es darum, einen typischen Angriffsvektor auf Linux-Server zu simulieren: automatisiertes Durchprobieren von Passwörtern auf dem SSH-Port. Ziel war, dass Suricata diese Aktivität als **Bruteforce** erkennt und ein Alert erzeugt. In einem realen SOC ist das wichtig, um frühzeitig zu sehen, dass ein Dienst im Fokus eines Angreifers steht.

4.2 Technische Umsetzung

Auf dem Zielsystem läuft ein SSH-Dienst, der über das Netzwerk erreichbar ist. Auf der Kali-VM starte ich das Tool **Hydra** mit einer Wortliste, um wiederholt Logins gegen diesen SSH-Dienst zu versuchen. In Suricata habe ich eine eigene Regel in der Datei `local.rules` angelegt, die auffällige SSH-Verbindungen und wiederholte Versuche erkennt. Sobald der Traffic die Bedingungen erfüllt, schreibt Suricata ein Alert-Event mit einer passenden Signatur, zum Beispiel „SSH Brute Force Attempt“, in die `eve.json`.

4.3 Beobachtung und Auswertung

Zunächst habe ich im Terminal die Datei `/var/log/suricata/eve.json` live verfolgt, um zu sehen, wann Suricata den Bruteforce erkennt. Später, nach der Integration mit Wazuh, habe ich denselben Angriff erneut ausgeführt und im Wazuh-Dashboard unter **Threat Hunting** nach dem Agenten „kali“ und der Suricata-Regel gefiltert. Dort tauchen die Alerts dann mit allen relevanten Feldern auf – etwa Quell-IP, Ziel-IP, Port

22 und der gewählten Regelbeschreibung. Damit kann man den Angriff nicht nur erkennen, sondern auch im Nachhinein sauber dokumentieren.

5. Use Case: Verdächtiger PowerShell-Download

5.1 Idee und Ziel

Der zweite Use Case konzentriert sich auf einen **verdächtigen PowerShell-Download**. Das Szenario: Ein Windows-Client lädt per PowerShell ein Script oder eine Datei von einem internen Webserver herunter. Solche Muster sind in echten Umgebungen sehr interessant, weil Angreifer PowerShell häufig für **Lateral Movement** und den Nachdownload von Malware verwenden.

5.2 Technische Umsetzung

Auf der Kali-VM starte ich einen einfachen HTTP-Server mit Python und stelle unter anderem eine Datei `payload.fatih` bereit. Auf dem Windows-Client führt der Benutzer (oder die Angreifer-Simulation) einen `Invoke-WebRequest`-Befehl aus, der diese Datei nach `c:\Temp` herunterlädt. In Suricata habe ich eine Regel erstellt, die HTTP-Requests auf genau diese URL beziehungsweise diesen Pfad überwacht und bei einem Treffer ein Alert mit der Signatur „Suspicious PowerShell Download“ erzeugt.

5.3 Beobachtung und Auswertung

In der `eve.json` sieht man zu diesem Use Case ein Alert-Event mit HTTP-Metadaten: Quell- und Zieladresse, Pfad, HTTP-Methode, User-Agent (z. B. PowerShell) und Status-Code 200. Über den Wazuh-Agent werden diese Ereignisse an den Manager weitergegeben und sind im **Threat-Hunting-Frontend** filterbar. Mit einem Filter wie `agent.name:"kali"` und der Regelbeschreibung kann ich mir alle verdächtigen PowerShell-Downloads anzeigen lassen. Damit wird aus einem einzelnen Testskript ein **nachvollziehbarer Detection-Use-Case** im SIEM.

6. Use Case: Malware/Trojan Test Download

6.1 Idee und Ziel

Der dritte Use Case ist eine Variante des PowerShell-Szenarios, diesmal mit Fokus auf einen generischen **Malware- oder Trojaner-Download**. Dazu nutze ich eine ungefährliche Testdatei, die ich als „malware-test.bin“ anbiete. Aus Sicht der Detection geht es darum, bestimmte Downloadpfade oder Dateinamen zu erkennen, die auf Malware-Kampagnen hinweisen.

6.2 Technische Umsetzung und Auswertung

Die technische Umsetzung ist ähnlich wie beim PowerShell-Use Case: Ein interner HTTP-Server auf Kali liefert die Datei `malware-test.bin`, ein Windows-Client lädt sie per Browser oder PowerShell herunter. In Suricata definiere ich eine Regel, die auf den Pfad beziehungsweise Dateinamen „/malware-test.bin“ reagiert und mit der Signatur „Malware/Trojan Test Download“ einen Alert erzeugt. Sowohl in der `eve.json` als auch im Wazuh-Dashboard kann ich die erzeugten Alerts eindeutig diesem Test-Download zuordnen und so zeigen, dass die **Detection funktioniert**.

7. Use Case: ICMP Ping / Reconnaissance

7.1 Idee und Ziel

Der ICMP-Use-Case ist vergleichsweise simpel, aber wichtig für das Verständnis von **Netzwerk-Reconnaissance**. Viele Angreifer testen zuerst mit Ping, welche Hosts im Netz erreichbar sind. Die Suricata-Regel „ICMP Ping Detected“ soll genau diese Aktivität sichtbar machen.

7.2 Umsetzung und Ergebnisse

Auf einem beliebigen Host im Netz – zum Beispiel der Windows-VM – sende ich mehrere Ping-Pakete an die überwachte IP. Suricata reagiert auf ICMP Echo Requests und schreibt Alerts mit der passenden Signatur in die `eve.json`. Über Wazuh werden diese Events übernommen, so dass im **Threat-Hunting-Frontend** mehrere „Suricata:

Alert – ICMP Ping Detected “-Einträge innerhalb des Testzeitraums erscheinen. Damit ist dokumentiert, dass einfache **Reconnaissance-Versuche** im SIEM sichtbar werden.

8. Wazuh-Integration: Probleme und Lösungsweg

8.1 Typische Stolpersteine

Der Weg zur funktionierenden Suricata-Wazuh-Integration war nicht geradlinig. In der separaten Wazuh-Probleme-Dokumentation habe ich festgehalten, welche Fehler aufgetreten sind und wie ich sie gelöst habe. Die wichtigsten Punkte lassen sich so zusammenfassen:

- **Falscher Kontext:** Ich habe anfangs mehrfach Befehle im falschen Kontext ausgeführt – also z. B. Docker-Kommandos im Container statt auf dem Host oder Konfigurationsdateien auf der falschen VM gesucht.
- **JSON-Decoder-Fehler:** Suricata-Events sind sehr umfangreich. Der Wazuh-Analysis-Dienst meldete häufig „Too many fields for JSON decoder“, wenn er große JSON-Strukturen aus der `eve.json` verarbeiten sollte.
- **Fehlkonfiguration der `ossec.conf`:** Ein erster Lösungsversuch über eine zusätzliche `json`-Sektion in der `ossec.conf` hat eher neue Probleme verursacht, weil Wazuh diese Struktur nicht akzeptiert hat und Dienste mit Konfigurationsfehlern starteten.
- **Neuaufsetzen des Stacks:** Erst ein sauberer Neuaufbau des Wazuh-Docker-Stacks auf Basis der offiziellen Version und eine Konfigurationsanpassung über `local_internal_options.conf` haben das Problem nachhaltig gelöst.

8.2 Endgültige Lösung

Die eigentliche Ursache für die JSON-Probleme lag in einem internen Limit von Wazuh für die maximale Anzahl von Feldern, die der Decoder pro Event verarbeitet. Die saubere Lösung bestand darin, genau diesen Parameter über die Datei `local_internal_options.conf` im Manager-Container anzupassen. Nach dem Erhöhen des Limits und einem Neustart des Wazuh-Managers wurden die Suricata-Events ohne Fehlermeldungen verarbeitet und erschienen **zuverlässig im Dashboard**.

8.3 Lessons Learned

Aus dieser Phase ergeben sich einige wichtige Lessons Learned:

- Immer prüfen, auf welchem System und in welchem Kontext ich arbeite (Host, VM, Container).
- Konfigurationen möglichst über vorgesehene Mechanismen und Zusatzdateien anpassen, statt Kernkonfigurationen im Container direkt zu verändern.
- Komplexe Fehler systematisch mit Logs, kleinen Tests und Dokumentation untersuchen.
- Genügend Zeit für Integrationsprobleme einplanen – besonders bei Kombinationsprojekten aus IDS, Agenten und SIEM.

9. Fazit

Die zusammengefassten Use Cases und die Wazuh-Fehleranalyse zeigen, dass ich mich nicht nur theoretisch mit Security-Themen beschäftige, sondern aktiv eine **kleine SOC-ähnliche Umgebung** aufgebaut habe. Ich kann Angriffe simulieren, eigene Detection-Regeln schreiben, Logs auswerten und komplexe Probleme in einer SIEM-Integration lösen. Diese Erfahrungen bilden eine **gute Grundlage** für den Einstieg in ein SOC-Team, in dem genau solche praktischen Fähigkeiten gefragt sind.