STEP 1: Docker Installation and Verification

What is being done: Docker Desktop is downloaded and installed on the local machine. Following installation, the user must verify the installation via a terminal or PowerShell by checking the version numbers.

Why this step matters: Docker acts as a lightweight virtual server. This allows Splunk to run in an isolated environment without polluting the host operating system, adhering to "Infrastructure as Code" principles.

Inputs required:

- Access to https://www.docker.com/products/docker-desktop/.
- Terminal or PowerShell access.

Expected output:

- Command docker --version returns a version number.

STEP 2: Project Directory Setup

What is being done: Creating the main project folder soc_pipeline.

Why this step matters: This folder acts as the root for your "Infrastructure as Code" configuration file. Note that unlike the original guide, we do **not** need to create a splunk_logs folder because your file uses a **Named Volume** managed internally by Docker.

Inputs required:

- Command: mkdir soc_pipeline.

Expected output:

- A directory named soc_pipeline is ready for use.

STEP 3: Create docker-compose.yml

What is being done: Creating a configuration file that defines three ports (8000, 8089, 1514) and a named volume for data storage.

Why this step matters: This file automates the deployment. It includes SPLUNK_GENERAL_TERMS to bypass legal prompts and uses a named volume (splunk-indexes) for high-performance indexing.

Inputs required:

- Create docker-compose.yml in the soc_pipeline folder with your specific content:

YAML

```
version: '3'
services:
  splunk:
    image: splunk/splunk:latest
    container_name: splunk
    ports:
      - 8000:8000
      - 8089:8089
      - 1514:1514
    environment:
      - SPLUNK_START_ARGS=--accept-license
      - SPLUNK_GENERAL_TERMS=--accept-sgt-current-at-splunk-com
      - SPLUNK_PASSWORD=<PASSWORD>
    volumes:
      - splunk-indexes:/opt/splunk/var/lib/splunk
volumes:
  splunk-indexes:
```

Expected output:

- A saved docker-compose.yml file.

STEP 4: Start Splunk Service

What is being done: Launching the containerized Splunk server in the background.

Why this step matters: This initializes the Splunk image and creates the splunk-indexes volume inside Docker's internal storage.

Inputs required:

- Command: docker compose up -d.

Expected output:

- Terminal shows the container "splunk" is started. docker ps shows it as "Up".

STEP 5: Service Authentication and Access

What is being done: The user accesses the Splunk Web Interface via a browser at localhost:8000 and logs in using the credentials defined in the docker-compose.yml file.

Why this step matters: Confirms the web server is active and accessible from the host machine, and verifies that the environment variables for authentication were correctly applied.

Inputs required:

- Web Browser.
- URL: http://localhost:8000.
- Username: admin.
- Password: <PASSWORD>

Expected output:

- Successful login to the Splunk Search & Reporting dashboard.

(Errors may occur here):

- "Invalid username or password" (Typo in YAML or login screen).

(Solution):

The password should meet a certain complexity requirements :
Password best practices for users | Splunk Docs

STEP 6: Index Configuration

What is being done: Within the Splunk interface, a new index named security_logs is created via the Settings menu.

Why this step matters: Creating a dedicated index isolates security data from general logs. This improves search performance, enables cleaner architecture, and allows for specific data retention policies .

Inputs required:

- Navigation: Settings > Indexes > New Index.
- Index Name: security_logs.

Expected output:

- The security-logs index appears in the Splunk index list.

STEP 7: TCP Input Configuration

What is being done: A Local TCP input is configured to listen on port 1514. The input is configured to tag incoming data with the Source Type suspicious_traffic and store it in the security_logs index.

Why this step matters: This "teaches" Splunk to listen for external traffic. It opens the pipeline, tags the data for identification, and routes it to the isolated storage created in Step 6 .

Inputs required:

- Navigation: Settings > Data Inputs > TCP > New Local TCP.
- Port: 1514.
- Source Type: suspicious_traffic.
- Index: security_logs.

Expected output:

- Splunk begins listening on TCP port 1514.

(Errors may occur here):

- Selecting the wrong index (data goes to main or default instead of security_logs).

- Port 1514 conflict within the container (unlikely but possible).

STEP 8: Integration Testing (Manual Log Injection)

What is being done: The user manually injects a mock log entry into the system using command-line tools (Netcat)and verifies its appearance in Splunk.

Why this step matters: This acts as a "Hello World" integration test to validate the full pipeline (Laptop >TCP >Docker >Splunk >Index) without writing complex code .

Inputs required:

- "src_ip $ip=1.2.3.4$ action=blocked" | ncat localhost 1514.
- **Splunk Search:** Query index=security_logs.

Expected output:

- Splunk search returns the event containing src_ip $ip=1.2.3.4$ and action=blocked .

(Errors may occur here):

- Network tools (Netcat) missing on the host.

(Solution):

Download the tool from the official website : **Ncat - Netcat for the 21st Century**

FINAL SECTION

1. Full Process Map

- Step 1: Docker Installation and Verification
- Step 2: Project Directory Structure Setup
- Step 3: Configuration of Infrastructure as Code (docker-compose.yml)
- Step 4: Container Deployment
- Step 5: Service Authentication and Access
- Step 6: Index Configuration
- Step 7: TCP Input Configuration
- Step 8: Integration Testing (Manual Log Injection)

2. Critical Failure Points

- **YAML Formatting (Step 3):** Docker Compose is extremely sensitive to indentation. A single space error will prevent the container from starting.
- **Port Mapping (Step 3/7):** If port 1514 is not correctly mapped in the YAML or configured in Splunk, data ingestion will silently fail.
- **Index Mismatch (Step 7/8):** If the input is configured for security-logs but the search is done on main (or vice versa), the data will appear lost.

3. Optimization Opportunities

- **Security:** The password <PASSWORD> is hardcoded in the YAML file. In a production environment, this should be managed via .env files or Docker secrets.
- **Data Retention:** The guide creates a persistent volume but does not specify retention policies (size/age) for the security_logs index, which could eventually consume all disk space.

4. Validation Checklist

- [ ] docker ps shows the container is up and healthy.
- [ ] Browser can access localhost:8000.
- [ ] index=security_logs in Splunk shows 0 events initially, and 1+ events after the Netcat/PowerShell test.

The Attack Script (ssh_brute_force.py)

What is being done: A Python script named ssh_brute_force.py is created to mimic an SSH brute-force attack. This script utilizes the standard socket library to generate raw TCP streams.

Why this step matters: Random logs do not resemble an actual attack. By forcing the script to loop failures quickly from a single IP, we create a specific statistical anomaly (high-velocity failures) required to test behavioral detection logic later. Using raw sockets instead of APIs ensures the simulation mimics actual infrastructure, like a Linux server's Syslog daemon.

Inputs required:

- Create a file named attacksimulation.py in the project folder.
- Paste the following Python code :

Python

```
import socket

import time

import random

import datetime


# CONFIGURATION

SPLUNK_HOST = 'localhost'

SPLUNK_PORT = 1514


# SIMULATION PARAMETERS

ATTACKER_IPS = ['192.168.1.105', '10.0.0.15', '172.16.23.88', '203.0.113.42']

TARGET_USERS = ['admin', 'root', 'service_account', 'jdoe']

SUCCESS_RATE = 0.1
```

```
def send_log(message):
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((SPLUNK_HOST, SPLUNK_PORT))
        sock.sendall(message.encode('utf-8'))
        sock.close()
        print(f"[+] Sent: {message.strip()}")
    except Exception as e:
        print(f"[-] Connection failed: {e}")

def generate_brute_force():
    attacker_ip = random.choice(ATTACKER_IPS)
    target_user = random.choice(TARGET_USERS)
    num_failures = random.randint(5, 20)

    print(f"[*] Starting attack from {attacker_ip} targeting user '{target_user}'...")

    # 1. GENERATE FAILURES
    for _ in range(num_failures):
        timestamp = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        log_entry = f"{timestamp} event_type=ssh_failed src_ip={attacker_ip}
user={target_user} reason='password check failed'\n"
        send_log(log_entry)
        time.sleep(random.uniform(0.1, 0.5))
```

```
    # 2. DECIDE IF SUCCESSFUL
    if random.random() < SUCCESS_RATE:
        timestamp = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        log_entry = f"{timestamp} event_type=ssh_success src_ip={attacker_ip}
user={target_user} reason='password accepted'\n"
        send_log(log_entry)
        print(f"[!] BREAKTHROUGH: {attacker_ip} successfully logged in as {target_user}!")


if __name__ == "__main__":
    print("--- Starting Traffic Generator ---")
    try:
        while True:
            generate_brute_force()
            time.sleep(random.randint(5, 10))
    except KeyboardInterrupt:
        print("\nStopping generator.")
```

Expected output:

- A saved ssh_brute_force.py file ready for execution.

STEP 2: Execute the Attack

What is being done: The user executes the Python script in the terminal to begin generating traffic.

Why this step matters: This action generates a "controlled noisy" dataset. The script introduces "jitter" (random sleep variance), ensuring logs do not arrive with identical timestamps, which prevents the SIEM from de-duplicating them or ingesting them out of order.

Inputs required:

- **Terminal Command:** python3 ssh_brute_force.py.
- **Action:** Let the script run for 2-3 minutes.

Expected output:

- Terminal output displaying [+] Sent: ... for each log entry.
- Roughly 50-100 logs generated.

(Errors may occur here):

- Connection failed. **(Solution):**
- Ensure the Docker container from Phase 1 is running and port 1514 is mapped correctly.

STEP 3: Verify in Splunk

What is being done: The user navigates to the Splunk Search & Reporting interface to validate data ingestion.

Why this step matters: We must confirm that Splunk is correctly parsing the key=value pairs (e.g., src_ip=1.2.3.4). Proper formatting allows for automatic field extraction without complex Regex, aligning with CIM (Common Information Model) best practices.

Inputs required:

- **Splunk Interface:** Search & Reporting.
- **Search Query:** index=security_logs sourcetype=suspicious_traffic.

Expected output:

- A list of events appears in the search results.
- **Interesting Fields:** src_ip and event_type appear as blue clickable links in the sidebar, confirming automatic parsing.

(Errors may occur here):

- Fields are not clickable/extracted.

  (Solution):

- Verify the Python script is outputting clean key=value strings with spaces between pairs.

Phase 3 — Detection Engineering

Goal: Create a high-fidelity alert that triggers only when a specific threshold of malicious behavior is met (behavioural detection → webhook → SOAR).

Executive Summary — What we're building

We will convert the brute-force pattern we generated in Phase 2 into a behavioral detection in Splunk. The detection aggregates events (not single atomic failures), applies a threshold to reduce noise, and triggers a Webhook that forwards an incident JSON to a local SOAR listener for automated enrichment and response.

STEP 3.1: The Discovery Search (Hunt phase)

What is being done:
Run an aggregation search (pivot) to find IPs with repeated failures and understand the attack pattern.

Task (Run in Splunk Search):

index=security_logs sourcetype=suspicious_traffic

| stats count by src_ip, user, event_type

Why this step matters:
stats collapses thousands of raw events into a simple table, letting the analyst spot high-count IPs and associated users. Senior engineers never build alerts on raw one-off events — we build alerts on aggregated behaviors.

Inputs required:

- Splunk Web → Search & Reporting.
- Index: security_logs.
- Source type: suspicious_traffic (configured in Phase 1).

step 1 soar

Expected output:
A table that shows src_ip, user, and event_type with counts. Attacker IPs from Phase 2 should appear with high ssh_failed counts (e.g., 15–20).

(Errors may occur here):

- *No results returned*.
  Solution: Confirm security_logs index exists and the TCP input is configured on port 1514 (Phase 1). Verify the attack script produced event_type=ssh_failed key=value pairs (Phase 2).

STEP 3.2: Apply the "Brute Force" Logic (Thresholding / Noise reduction)

What is being done:
Refine the hunt query to compute the number of failures per source IP, list attempted usernames, apply a numeric threshold, and add meta fields used by downstream automation.

Task (Run in Splunk Search):

index=security_logs event_type=ssh_failed earliest=-15m

| stats count as failure_count values(user) as attempted_users by src_ip

| where failure_count > 10

| eval severity="high", attack_type="brute_force"

What this does (line by line):

- earliest=-15m scopes the search to the last 15 minutes (adjustable).
- stats count as failure_count … by src_ip aggregates failures per source IP.
- values(user) returns the set of usernames tried by that IP.
- where failure_count > 10 enforces the detection threshold (filters out fat-finger mistakes).
- eval creates standard fields (severity, attack_type) to simplify SOAR ingestion.

Why this step matters (Senior perspective):
This is behavioural detection. It reduces analyst noise by ignoring single failures and focusing on statistically suspicious behaviour. Setting the threshold to > 10 is a tuning decision—start conservative and iterate based on observed false positives.

Inputs required:

- Access to Splunk Search & Reporting.

- The same index and source type as previous steps.
- A sensible time window (e.g., -15m, -30m, or -1h depending on environment).

Expected output:
A list of src_ip values that meet the brute force threshold, each with failure_count, attempted_users, severity, and attack_type.

(Errors may occur here):

- *High false positives (many legitimate IPs triggered).*
  Solution: Increase threshold, narrow time window, add allowlist (internal IPs), or require multiple correlated signals (e.g., many distinct target users **and** newdst port scanning).
- *No IPs triggered even though attack script ran.*
  Solution: Verify time window (earliest), confirm event_type field is ssh_failed, and check field extraction. See Phase 2 parsing guidance.

step2soar

STEP 3.3: Save the Alert & Configure the Webhook (Automation hookup)

What is being done:
Save the query above as a scheduled alert that runs every 5 minutes and forwards results to the SOAR listener using a Webhook.

Task (Splunk UI steps):

1. In Splunk Search (with the query above) click **Save As → Alert**.
2. **Title:** Brute Force Detected - Forward to SOAR
3. **Alert Type:** Scheduled (Run every 5 minutes).
4. **Trigger Condition:** Number of Results > 0
    - This triggers when the search returns ≥1 result set (each row = an IP that passed threshold).
5. **Add Actions → Webhook**
    - **URL:** http://host.docker.internal:5000/alerts

        - Rationale: host.docker.internal allows the Splunk container to reach the host machine where the Python SOAR listener runs.
    - Optionally configure HTTP Method = POST and select fields to include (or use the default Splunk event JSON).
6. Save and enable the alert.

Why a Webhook (Senior "Why"):

- Webhooks are machine-friendly (JSON over HTTP) enabling sub-second automated workflows. Emails are human-oriented and slow; Webhooks enable orchestration (enrichment, IP blocking, case creation).

Expected result:
When an IP exceeds the threshold, Splunk posts JSON to http://host.docker.internal:5000/alerts and your SOAR listener (Python process on host port 5000) receives a payload and begins automation.

Phase 4—Alerting & Webhook Orchestration

Goal: Bridge the gap between **Detection** (Splunk) and **Orchestration** (Python) by creating a real-time, event-driven "push" notification system.

Executive Summary

What we're building:

In this phase, we move from passive logging to active communication. We are constructing a **REST API listener** using the Flask framework. This listener acts as the "ears" of our SOC. Instead of the SOAR platform constantly polling Splunk for updates—which is resource-intensive and slow—Splunk will proactively "push" a JSON payload to our Python script the moment a threat is validated.

STEP 4.1: Establishing the SOAR Listener (The Python "Ear")

What is being done:

Setting up a Python virtual environment and developing a Flask-based web server (soar_orchestrator.py) that remains in a listening state to receive and parse incoming HTTP POST requests from Splunk.

Task (Run in Terminal/IDE):

1. **Initialize Environment:**

PowerShell

```
python -m venv venv

.\venv\Scripts\activate

pip install flask
```

2. **Develop the Listener (soar_orchestrator.py):**

Python

```
from flask import Flask, request, jsonify


app = Flask(__name__)
```

```
@app.route('/webhook', methods=['POST'])
def splunk_alert():
    data = request.json
    print("\n" + "="*50)
    print("!!! NEW ALERT RECEIVED FROM SPLUNK !!!")
    print("="*50)

    if data and 'result' in data:
        threat_actor = data['result'].get('src_ip', 'Unknown IP')
        target_user = data['result'].get('user', 'Unknown User')
        fail_count = data['result'].get('failure_count', 'N/A')

        print(f"THREAT DETECTED : Brute Force Attempt")
        print(f"ATTACKER IP     : {threat_actor}")
        print(f"TARGET USER     : {target_user}")
        print(f"FAIL COUNT      : {fail_count}")
        print("-" * 50)
        print("ACTION: Ready for Phase 5 (Enrichment)...")

    return jsonify({"status": "success"}), 200

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Why this step matters (Senior Perspective):

"Senior engineers never install libraries globally. Dependency hell is a real threat to production stability. By using a **Virtual Environment**, we ensure our SOAR logic is portable and isolated. Furthermore, we use **Flask** because it's lightweight; we don't need a massive enterprise framework just to listen for a JSON 'shout' from our SIEM."

Inputs required:

- Python 3.x installed.
- Flask library.
- The soc_pipeline project directory.

Expected output:

- Terminal displays: * Running on http://127.0.0.1:5000 (or 0.0.0.0).
- A "200 OK" status is returned whenever the endpoint is pinged.

STEP 4.2: Configuring the Automation Hook (Splunk Webhook)

What is being done:

Configuring the Splunk search results to trigger a Webhook action that targets the specific IP and port of our Flask listener.

Task (Splunk UI Steps):

1. **Run Logic:** Search index=security_logs event_type=ssh_failed | stats count as failure_count values(user) as user by src_ip | where failure_count > 5.
2. **Save As:** Alert.
3. **Title:** SOC-Automation-Brute-Force.
4. **Trigger Condition:** Number of Results > 0.
5. **Add Action:** Select **Webhook**.
6. **URL:** http://host.docker.internal:5000/webhook.

Why this step matters (Rationale):

We use host.docker.internal because Splunk lives in a "containerized bubble." If you used localhost, Splunk would look for the listener inside its own container. This DNS name

allows the data to "exit" the Docker network and reach the Python process running on your actual host machine.

STEP 4.3: End-to-End Integration Testing

What is being done:

Executing the full attack-to-orchestration pipeline to verify that logs are generated, detected, and successfully transmitted to the SOAR listener.

Task:

1. Start the listener: python soar_orchestrator.py.
2. Execute the attack: python ssh_brute_force.py.
3. Monitor the Splunk **Triggered Alerts** dashboard.

Expected output:

- The Python terminal running soar_orchestrator.py should print the formatted alert details (Attacker IP, Target User, etc.) automatically within seconds of the threshold being met.

Phase 5: The Enrichment Layer (VirusTotal)

Goal: Transform the system from a passive listener into an active investigator by integrating external Threat Intelligence.

Executive Summary

What we're building

In this phase, we transform your script from a simple "message receiver" into an "Automated Investigator." Previously, the system only told you *that* an IP was attacking. Now, the script will automatically query the **VirusTotal API** to check the IP's global reputation.

Why this matters (The "Senior" Philosophy)

Security analysts suffer from "alert fatigue." If an analyst has to manually copy-paste every attacking IP into a browser to check if it's malicious, they waste valuable time.

By automating this lookup **at the moment of detection**, we provide immediate context. We are building a **decision-support engine**, not just a log collector.

STEP 5.1: Get Your "Security Clearance" (API Key)

What is being done:

We are registering for a free account with VirusTotal to generate an API Key. This key acts as an authentication token, allowing our Python script to query VirusTotal's massive database of malware and threat intelligence.

Task:

1. Go to VirusTotal.com and create a free account.
2. Once logged in, click on your **Profile (top right) $\rightarrow$ API Key**.
3. Copy the alphanumeric string displayed there.

Why this step matters:

APIs (Application Programming Interfaces) are the glue of modern orchestration. Without a key, external services will reject your automated requests to prevent abuse.

Inputs required:

- A valid email address for VirusTotal registration.

Expected output:

- A secure API Key string (e.g., a1b2c3d4...).
- **Critical Note:** Keep this key secret. Never commit it to a public GitHub repository.

STEP 5.2: The Upgrade (Updating the Python Code)

What is being done:

We are modifying soar_orchestrator.py to include a new function, check_virustotal. This function handles the HTTP requests to the third-party service, parses the JSON response, and extracts a "Malicious Score."

Prerequisite Task (Terminal):

You must install the requests library, which allows Python to send HTTP traffic.

PowerShell

```
pip install requests
```

Task (Update soar_orchestrator.py):

Replace your entire existing script with the code below. **Remember to paste your API Key inside the quotes.**

Python

```
import requests
from flask import Flask, request, jsonify

app = Flask(__name__)

# 1. YOUR CONFIGURATION
# <--- PASTE YOUR API KEY BELOW
VT_API_KEY = "YOUR_ACTUAL_API_KEY_GOES_HERE"

def check_virustotal(ip_address):
```

```
    """Checks the reputation of an IP address using VirusTotal API v3"""
    url = f"https://www.virustotal.com/api/v3/ip_addresses/{ip_address}"
    headers = {
        "x-apikey": VT_API_KEY
    }

    try:
        response = requests.get(url, headers=headers)
        if response.status_code == 200:
            data = response.json()
            # Extract how many security vendors flagged this IP as malicious
            stats = data['data']['attributes']['last_analysis_stats']
            malicious_count = stats.get('malicious', 0)
            return malicious_count
        else:
            print(f"[!] VT Error: {response.status_code}")
            return 0
    except Exception as e:
        print(f"[!] Connection Error: {e}")
        return 0

@app.route('/webhook', methods=['POST'])
def splunk_alert():
    data = request.json

    if data and 'result' in data:
```

```
        src_ip = data['result'].get('src_ip')
        print(f"\n[+] Analyzing IP: {src_ip}")

        # 2. THE ENRICHMENT STEP
        malicious_hits = check_virustotal(src_ip)

        print(f"--- VIRUSTOTAL REPORT ---")
        print(f"Malicious Flags: {malicious_hits}")

        # 3. THE DECISION LOGIC
        if malicious_hits > 0:
            print(f"VERDICT: CRITICAL - This IP is a known threat actor!")
            print(f"ACTION: Sending High-Priority Alert to Slack/Teams...")
        else:
            print(f"VERDICT: LOW - This IP is not currently flagged.")
            print(f"ACTION: Logging incident for review.")

    return jsonify({"status": "success"}), 200

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Why this step matters (The "Handshake"):

1. **The Trigger:** Splunk sends the alert to /webhook.
2. **The Request:** Python pauses, takes the IP, and "shows its ID card" (API Key) to VirusTotal.
3. **The Analysis:** VirusTotal scans its database.

4. **The Decision:** The script applies logic (if malicious_hits > 0) to determine if the alert is Critical or Low priority.

Expected output:

- The script saves without syntax errors.

STEP 5.3: End-to-End Integration Testing

What is being done:

We will manually trigger the webhook using PowerShell to simulate a Splunk alert. We will use a known safe IP (Google DNS) and a potentially malicious IP to test the "Verdict" logic.

Task:

1. Start your listener in one terminal:

PowerShell

python soar_orchestrator.py

2. Open a second terminal (PowerShell) and send a mock alert using Google's IP (8.8.8.8):

PowerShell

Invoke-RestMethod -Uri http://localhost:5000/webhook -Method Post -Body '{"result": {"src_ip": "8.8.8.8"}}' -ContentType "application/json"

Expected output (Terminal 1 - The Python Script):

Plaintext

[+] Analyzing IP: 8.8.8.8

--- VIRUSTOTAL REPORT ---

Malicious Flags: 0

VERDICT: LOW - This IP is not currently flagged.

ACTION: Logging incident for review.

(Errors may occur here):

- **Error:** ModuleNotFoundError: No module named 'requests'

    - **Solution:** Run pip install requests in your terminal (ensure your virtual environment is active if you are using one).

- **Error:** [!] VT Error: 401 or 403
    - **Solution:** Your API Key is missing or incorrect. Check the VT_API_KEY variable in the script.
- **Error:** [!] VT Error: 429
    - **Solution:** Quota exceeded. The free VirusTotal API allows only 4 requests per minute. Wait a moment and try again.

Phase 6: Response & Notification (Discord Integration)

Goal: Close the feedback loop by automating the communication of critical intelligence to the security team via a centralized platform (Discord).

Executive Summary

What we're building

In Phase 5, we successfully enriched our alert data with threat intelligence from VirusTotal. However, that data currently lives in a terminal window on a server. Information is useless if it doesn't reach the analysts who need to act on it.

In this phase, we will build the **Response** mechanism. We will update our Orchestrator to format the security alert—including the IP reputation and attack details—and push it directly to a dedicated Discord channel using Webhooks.

Why this matters (Senior Perspective)

"In a modern SOC (Security Operations Center), 'Mean Time to Respond' (MTTR) is a critical KPI. If an analyst has to constantly refresh a dashboard or check log files to see an alert, we add latency to our response. By pushing alerts to a collaboration platform like Discord, we ensure the team is notified immediately, regardless of where they are looking. We are moving from a 'Pull' model (checking logs) to a 'Push' model (alerting)."

STEP 6.1: Configure the Communications Channel

What is being done:

We are creating a dedicated "webhook" within Discord. A webhook is essentially a unique URL that accepts data. When our Python script sends a JSON payload to this URL, Discord renders it as a message in the chat.

Task:

1. **Create a Server/Channel:** Open Discord and create a server (e.g., "Home Lab SOC") and a text channel (e.g., #alerts-critical).
2. **Access Integrations:** Click the "Edit Channel" (gear icon) next to your channel name $\rightarrow$ **Integrations** $\rightarrow$ **Webhooks**.
3. **Create Webhook:** Click "New Webhook".
    - **Name:** SOC Bot (or Splunk Orchestrator).

    - **Channel:** #alerts-critical.
4. **Copy URL:** Click "Copy Webhook URL". Save this string immediately.

Inputs required:

- A Discord account and permission to manage a server.

Expected output:

- A secure Webhook URL (e.g., https://discord.com/api/webhooks/12345...).

STEP 6.2: The Response Logic (Updating the Python Code)

What is being done:

We are updating soar_orchestrator.py to include a new function, send_discord_alert. This function will take the enriched data (Splunk Alert + VirusTotal Reputation) and format it into a professional Discord **Embed**.

Why this matters:

We use **Embeds** (rich text blocks with colors and fields) rather than plain text.

- **Color Coding:** We will programmatically set the color to **Red** if the IP is malicious, and **Grey/Blue** if it is safe. This allows analysts to triage severity at a glance.

Task (Update soar_orchestrator.py):

Replace your existing script with the code below.

Note: You must replace YOUR_VT_API_KEY and YOUR_DISCORD_WEBHOOK_URL with your actual keys.

Python

```
import requests

from flask import Flask, request, jsonify

import datetime


app = Flask(__name__)
```

```
# --- CONFIGURATION ---
# PASTE YOUR API KEYS HERE
VT_API_KEY = "YOUR_ACTUAL_VT_API_KEY"
DISCORD_WEBHOOK_URL = "YOUR_ACTUAL_DISCORD_WEBHOOK_URL"

def check_virustotal(ip_address):
    """Checks the reputation of an IP address using VirusTotal API v3"""
    url = f"https://www.virustotal.com/api/v3/ip_addresses/{ip_address}"
    headers = {"x-apikey": VT_API_KEY}

    try:
        response = requests.get(url, headers=headers)
        if response.status_code == 200:
            data = response.json()
            stats = data['data']['attributes']['last_analysis_stats']
            return stats.get('malicious', 0)
        else:
            print(f"[!] VT Error: {response.status_code}")
            return 0
    except Exception as e:
        print(f"[!] Connection Error: {e}")
        return 0

def send_discord_alert(alert_data, malicious_count):
    """Formats and sends the alert to Discord via Webhook"""
```

```
# 1. Determine Severity and Color
if malicious_count > 0:
    color = 15158332  # RED (Critical)
    severity = "CRITICAL"
    description = "**Threat Intelligence indicates this IP is malicious.**"
else:
    color = 3447003   # BLUE (Informational)
    severity = "LOW"
    description = "IP is not currently flagged in threat databases."

# 2. Extract Data
src_ip = alert_data.get('src_ip', 'Unknown')
user = alert_data.get('user', 'Unknown')
fail_count = alert_data.get('failure_count', 'N/A')
timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

# 3. Build the Discord Embed JSON
payload = {
    "embeds": [
        {
            "title": f" 🚨 SSH Brute Force Detected ({severity})",
            "description": description,
            "color": color,
            "fields": [
                {"name": "Attacker IP", "value": src_ip, "inline": True},
                {"name": "Target User", "value": user, "inline": True},
```

```
            {"name": "Failures", "value": str(fail_count), "inline": True},
            {"name": "VirusTotal Hits", "value": str(malicious_count), "inline": True},
            {"name": "Time", "value": timestamp, "inline": False}
          ],
          "footer": {"text": "SOC Orchestrator • Phase 6"}
        }
      ]
    }

    # 4. Send the Request
    try:
        response = requests.post(DISCORD_WEBHOOK_URL, json=payload)
        if response.status_code == 204:
            print("[+] Discord Alert Sent Successfully.")
        else:
            print(f"[-] Discord Error: {response.status_code}")
    except Exception as e:
        print(f"[-] Failed to send Discord alert: {e}")

@app.route('/webhook', methods=['POST'])
def splunk_alert():
    data = request.json
    if data and 'result' in data:
        # Extract Splunk Data
        result = data['result']
        src_ip = result.get('src_ip')
```

```
        print(f"\n[+] Alert Received for IP: {src_ip}")

        # Step 1: Enrich
        malicious_hits = check_virustotal(src_ip)
        print(f"    --> VT Score: {malicious_hits}")

        # Step 2: Notify (Discord)
        send_discord_alert(result, malicious_hits)

        return jsonify({"status": "success"}), 200

    return jsonify({"status": "error"}), 400

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Inputs required:

- The requests library (installed in Phase 5).
- Valid API keys for both services.

STEP 6.3: End-to-End Integration Testing

What is being done:

We will validate the entire pipeline: **Attacker $\rightarrow$ Splunk $\rightarrow$ Webhook $\rightarrow$ Python $\rightarrow$ VirusTotal $\rightarrow$ Discord.**

Task:

1. **Start the Listener:**

python soar_orchestrator.py

2. **Generate Traffic (The Attack):**

Open a second terminal and run your brute force script:

python ssh_brute_force.py

3. **Wait for Splunk:**

Wait for the scheduled alert (runs every 5 minutes) to trigger.

Expected output:

1. **Terminal:** The Python script displays [+] Discord Alert Sent Successfully.
2. **Discord:** You receive a notification in your #alerts-critical channel.
    - It should look like a formatted card.
    - It should contain the IP address, user, failure count, and VT score.

Errors may occur here:

- **Error:** requests.exceptions.MissingSchema: Invalid URL
    - *Solution:* Ensure your DISCORD_WEBHOOK_URL inside the script starts with https://.
- **Error:** Alert sends, but shows "0" VirusTotal hits for a known bad IP.
    - *Solution:* You may have hit the API rate limit (4 requests/minute) or are testing with a local IP (e.g., 192.168.x.x) which does not exist in the public VirusTotal database. To test a "Critical" alert, manually modify your ssh_brute_force.py to inject a known malicious IP (e.g., 185.156.73.100 - *Use with caution, do not connect to it, just use the string*).
