ELK SIEM Project – Brute Force Response Automation Documentation

Overview

This project documents the process of updating an ELK (Elasticsearch, Logstash, Kibana) stack SIEM using python and Kibana querying to automate a response to Brute Force SSH attempts.

Tools & Stack

- **OS: Ubuntu**
- **UFW** (Ubuntu firewall)
- **Elasticsearch** (Search/Analytics Engine)
- **Logstash** (Used to parse and forward logs before they hit Elasticsearch)
- **Kibana** (Dashboard/Visualization)
- **Filebeat** (Linux log shipper)
- **Python**

All ELK services were deployed on a Linux (Ubuntu) virtual machine named arzsec-cyber, which also served as the Linux (Filebeat) endpoint. My host machine served as the Windows (winlogbeat) endpoint.

---

1. Updating Log Ingestion

Because Filebeat was already installed on our system before, there's no need to install it now. However, we will need to update the filebeat.yml file for simplicity and performance.

1.1. Edit filebeat.yml file

Previously, Filebeat was outputting our log files into logstash which parsed and filtered our data. Now, we will utilize Elasticsearch's ingest pipelines which will do the parsing and enrichment inside Elasticsearch itself. This will allow us to bypass logstash and utilize Elastic Common Schema (ECS) for ease of querying in Kibana later.

We're first going to update our filebeats inputs in filebeat.yml and utilize filebeat system module.

sudo apt mousepad /etc/filebeat/filebeat.yml

Previous:

```
filebeat.inputs:
- type: filestream
  id: my-filestream-id
  enabled: true
```

```
  paths:
    - /var/log/*.log
    - /var/log/*/*.log
    - /var/log/messages
    - /var/log/secure
    - /var/log/audit/audit.log
    - /var/log/nginx/*.log
output.logstash:
  hosts: ["http://<vm-ip-address>:5044"]
```

Now:

```
filebeat.inputs:
- type: filestream
  id: my-filestream-id
  enabled: true
  paths:
    - /var/log/messages
    - /var/log/secure
    - /var/log/audit/audit.log
    - /var/log/nginx/*.log
output.elasticsearch:
  hosts: ["http://<vm-ip-address>:9200"]
# output.logstash:
  # hosts: ["http://<vm-ip-address>:5044"]
```

Notice that we removed the following log input paths:

```
    - /var/log/*.log
    - /var/log/*/*.log
```

And commented out the Logstash outputs:

```
# output.logstash:
  # hosts: ["http://<vm-ip-address>:5044"]
```

To compensate for the removal of these log input paths, we will be using ‘filebeat system module’. This will allow us to gather logs from the same sources that we had before with fine-tuned parsers for auth logs, syslog, etc. Let’s enable filebeat system module now:

```
sudo filebeat modules enable system
```

```
sudo systemctl restart filebeat
```

1.1. Verify that filebeat modules system is operating

To verify that our update worked, we're going to simulate a few failed ssh attempts that would generate auth logs.

Kibana should be up and running since we set it up along with our SIEM last time. When we visit Analytics > Discover and set our index pattern to filebeat-* we should be able to search for failed SSH attempts using the following KQL query:

event.dataset: "system.auth" and system.auth.ssh.event: "Failed"

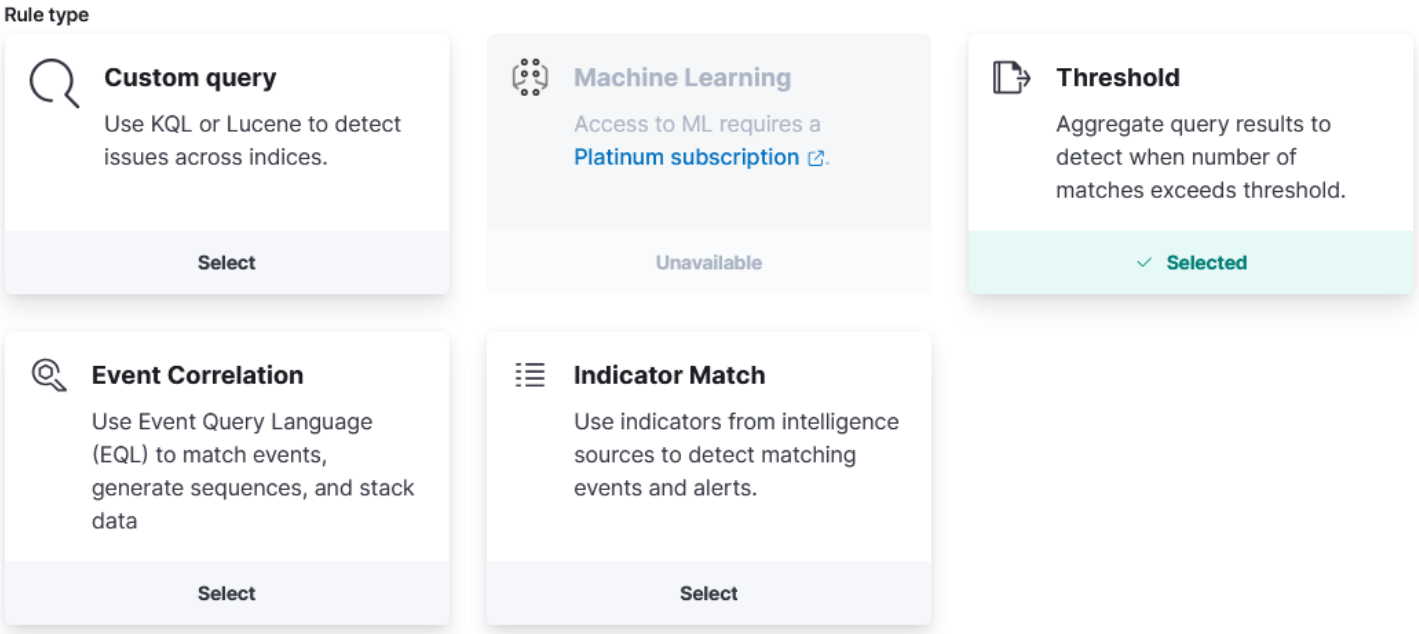
We should see an event generated for each failed SSH attempt.

---

2. Detection Rule in Kibana

If our KQL query results in the expected result at the end of step 1.1, then we can proceed with setting it as an alert rule in Kibana > Security. To do this, we'll go to Kibana > Security > Rules > Create new rule.

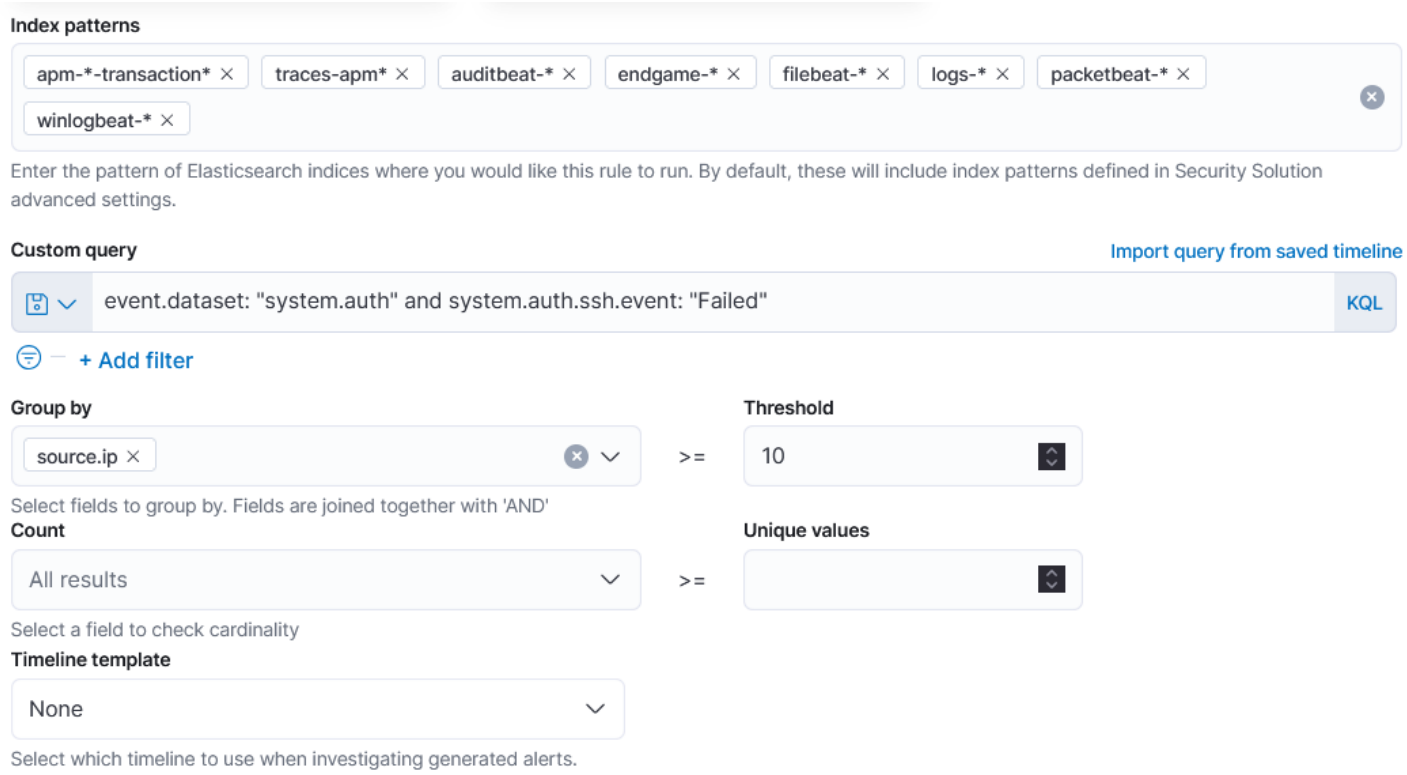
We're going to define our rule type as 'Threshold' because we want to make sure the rule only triggers after multiple failed SSH attempts.



We'll use the same custom query that we used to test our filebeat systems module operationality.

event.dataset: "system.auth" and system.auth.ssh.event: "Failed"

We'll also set the alert to group by 'source.ip' and set the threshold at >= 10. This will trigger an alert if more than 10 failed attempts come from the same IP within 5 minutes



Then we can add additional information such as a name for the alert, risk rating, MITRE ATT&CK associations, investigation guides, etc.

We can also schedule the rule to run periodically and set a 'look back time' which will tell the rule to check the trailing period of whichever length we set it at.

Finally, we can set 'Rule actions' that tell Kibana what to do if a rule is triggered. We will instruct Kibana to send a webhook to a python server which we'll set up shortly.

---

3. Automated Response Setup

In the last step, we allowed Kibana to send a webhook whenever an alert was triggered. We will now feed this webhook into a python server which will execute the commands we'd like.

3.1. Blocking Script

We will create a python script (block_ip.py) that:

a) Accepts an ip address as input
b) Uses ufw (Ubuntu firewall) to block traffic from that IP

We will make the script executable and place it in /usr/local/bin . Let's do this now:

sudo mousepad /usr/local/bin/block_ip.py

In the text window we will write the following python script:

```
#!/usr/bin/env python3

import sys
import subprocess
import ipaddress
import logging

# Configure logging
logging.basicConfig(filename='/var/log/block_ip.log',
                    level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')

def is_valid_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError:
        return False

def already_blocked(ip):
    # Check ufw status for the IP (returns True if present)
    proc = subprocess.run(["ufw", "status", "numbered"], capture_output=True, text=True)
    return ip in proc.stdout

def block_ip(ip):
    if not is_valid_ip(ip):
        logging.error("Invalid IP: %s", ip)
        print(f"Invalid IP: {ip}")
        return 1

    if already_blocked(ip):
        logging.info("IP already blocked: %s", ip)
        print(f"IP already blocked: {ip}")
        return 0

    try:
        # Run ufw as the current user (expecting caller to use sudo if necessary)
        subprocess.run(["ufw", "deny", "from", ip], check=True)
        logging.info("Blocked IP: %s", ip)
        print(f"Blocked IP: {ip}")
        return 0
    except subprocess.CalledProcessError as e:
        logging.exception("Failed to block IP %s: %s", ip, e)
```

```
        print(f"Failed to block IP {ip}: {e}")
        return 2

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: block_ip.py <IP>")
        sys.exit(1)
    sys.exit(block_ip(sys.argv[1]))
```

This will automate blocking attacker IPs immediately, preventing further brute force attempts without the need for manual intervention.

I'll briefly explain what this script is doing below:

How it behaves end-to-end

- You run it like this in the terminal: sudo python3 /usr/local/bin/block_ip.py <input ip>
- It validates the IP → checks if already blocked → tries to add a UFW deny rule
- It logs everything to /var/log/block_ip.log and prints a short message
- Exit codes:
    - 0 = success (blocked or already blocked)
    - 1 = bad input (invalid IP)
    - 2 = command failed (e.g., UFW error/permission issue)

NOTE: Because the script calls ufw without sudo inside, you should invoke the whole script with sudo so ufw has the privileges it needs.

Now that the script has been made, we can set ownership and permissions:

sudo chown root:root /usr/local/bin/block_ip.py

sudo chmod 750 /usr/local/bin/block_ip.py

Next, we can create the log file for this script and set permissions for that:

sudo touch /var/log/block_ip.log

sudo chown root:root /var/log/block_ip.log

sudo chmod 640 /var/log/block_ip.log

Finally, we can test the script by running the following commands:

sudo python3 /usr/local/bin/block_ip.py <input ip>

sudo ufw status numbered

or

sudo ufw status | grep <input ip>

```
cybersecurityanalyst@arzsec-cyber:~$ sudo python3 /usr/local/bin/block_ip.py 203.0.113.45
Rules updated
Blocked IP: 203.0.113.45
```

3.2. Webhook Receiver

Our next goal is to write a short python script that will receive an ip from Kibana and feed it into 'block_ip.py'. Receiving information over the internet will open us up to new security risks, and we should therefore consider how we go about deploying our server. Let's introduce these risks and mitigations step by step as we work through this.

What we want to do is build a small Flask web server (webhook_server.py) that:

a) listens on a port for POST requests from Kibana
b) extracts the attacker IP from incoming webhook payloads (the POST request)
c) calls the blocking script to add the IP to the firewall deny rules

Flask is a web framework for python that has a set of tools and rules that make it easier and faster to build web applications and APIs. This will allow us to write a script that can listen of incoming HTTP/HTTPS requests on a specific port, define specific routes that respond to these requests, parse incoming data (like the JSON payload from Kibana), and send back structured responses to the sender.

Some of the potential risks we will face are:

a) data in transit (from Kibana to our server) may be visible to someone on our network that's eavesdropping on the communication
b) our server could receive unauthorized requests from sources other than Kibana
c) vulnerabilities in the Flask application itself may allow a malicious actor to use Flask tools to escalate privileges and access a root shell through direct sudo calls
d) a malicious actor may attempt to deny service to ufw via garbage input (i.e. anything that isn't an ip)

To prevent abuse, we will incorporate the following features into our server:

a) HTTPS with a self-signed certificate
    - HTTPS will encrypt all communication between Kibana and our webhook server, preventing prying eyes from seeing traffic
b) API key authentication (shared secret token/bearer token)
    - Each request from Kibana will include an API key signature generated with a shared secret token
    - Our server will compare the received API key with the securely stored secret, and only process requests if they match
c) Dedicated, unprivileged user + running the script as a system service

- We'll create a new system service that will manage our script, reduce downtime, and utilize the permissions and identity of dedicated, unprivileged user that will be exist solely to the task of running this script
- If an attacker manages to exploit a vulnerability in the Flask application itself, their control would be limited to the very restricted user, preventing them from easily gaining root access or affecting other system components
- Additionally, system services under systemd have sandboxing directives that create a more isolated environment for the service, restricting its ability to write to sensitive system directories, access user home directories, or use shared temporary directories, even if partially compromised

d) Double Input validation
- The IP addresses from the webhook payload will be validated, first, by the Flask application, and then again, by the block_ip.py script to make sure only valid ip addresses are passed to the ufw command

e) Logging
- The Flask application will log all of its activities, including received requests, successful blocks, and errors, to a specific, controlled log file

3.2.1. Adding a dedicated, unprivileged user

Now that we finally have a list of everything we want to do, let's get started by creating a new, dedicated, and unprivileged user that will solely exist for the purpose of running the script we write:

```
sudo useradd –system –no-create-home –shell /usr/sbin/nologin webhookd
```

This creates a new user named webhookd
- --system: Makes it a system user (usually for services, not human logins)
- --no-create-home: Prevents creating a home directory, as this user won't be logging in interactively
- --shell /usr/sbin/nologin: Ensures this user cannot log in or execute a shell, further restricting its capabilities

3.2.2. Setting up TLS Certificates for HTTPS

Next, let's set up TLS certificates so that our script can use HTTPS instead of relying on HTTP:

```
sudo mkdir -p /etc/ssl/webhook
```

```
cd /etc/ssl/webhook
```

```
sudo openssl req -x509 -newkey rsa:4096 -keyout webhook.key -out webhook.crt -days 365 -nodes
```

This creates a new directory under /etc/ssl and generates both a self-signed SSL certificate (webhook.crt) and a private key (webhook.key) which are used in the TLS/SSL process:

```
cybersecurityanalyst@arzsec-cyber:/etc/ssl/webhook$ ls
webhook.crt  webhook.key
```

Next, we'll set ownership and strict permissions for the key, so only root can write to it, and only root or group can read it.

```
sudo chown root:root webhook.*
```

```
sudo chmod 640 webhook.key
```

Finally, we'll change the group that webhook.key belongs to:

```
sudo chgrp webhookd /etc/ssl/webhook/webhook.key
```

The file now belongs to the group webhookd. This allows the webhookd user who will run Flask (and who is part of the webhooked group) to read the private key, which is necessary for Flask to establish HTTPS connections, while still preventing other users from reading it.

3.2.3. Setting up a secret API key

Moving on to the issue of authentication, we want to make sure that Kibana is the only one sending us POST requests. To do this, we'll set up an API key that we'll give to Kibana so that it can authenticate itself when speaking to our server. Our script will only process the request if it verifies that the API key is correct.

Let's create this API key now:

```
sudo bash -c 'umask 077; echo "WEBHOOK_TOKEN=$(openssl rand -hex 24)" > /etc/webhook_server.env'
```

This generates a random 24-byte hexadecimal string using openssl rand -hex 24. It saves this string as an environment variable WEBHOOK_TOKEN into a file named /etc/webhook_server.env. The reason we want to put this key into this file is because it contains sensitive information that should not be hardcoded directly into the source code of our script.

We'll set the permissions for this file to be very strict:

```
sudo chown root:root /etc/webhook_server.env
```

```
sudo chmod 600 /etc/webhook_server.env
```

After these changes, only the root user can read or write to this file.

3.2.4 Writing our script

Finally, we can write the script that will run our server. Let's install Flask now:

```
sudo apt install python3-flask
```

In /usr/local/bin we can create our application:

sudo mousepad webhook_server.py

The script for our application is as follows:

```
#!/usr/bin/env python3
import os
import logging
from ipaddress import ip_address, ip_network
from flask import Flask, request, jsonify
import subprocess

# --- Config ---
SECRET = os.getenv("WEBHOOK_TOKEN", "")
# Allowlist: adjust to your environment (VM IP, Kibana IP, LAN)
ALLOW_NETS = [
   "127.0.0.0/8",
   "::1/128",
   "192.168.0.0/16",  # your LAN (update if needed)
   "10.0.0.0/8"
]
CERT_PATH = "/etc/ssl/webhook/webhook.crt"
KEY_PATH  = "/etc/ssl/webhook/webhook.key"
LOG_PATH  = "/var/log/webhook_server.log"

# --- App & logging ---
app = Flask(__name__)
logging.basicConfig(filename=LOG_PATH, level=logging.INFO, format="%(asctime)s %(levelname)s %(message)s")

def client_allowed(remote_ip: str) -> bool:
   try:
      rip = ip_address(remote_ip)
      for cidr in ALLOW_NETS:
         if rip in ip_network(cidr, strict=False):
            return True
   except Exception:
      return False
   return False

def unauthorized(reason: str):
   logging.warning("Unauthorized: %s (from %s)", reason, request.remote_addr)
   return jsonify({"error": "unauthorized", "reason": reason}), 401

@app.route("/blockip", methods=["POST"])
def block_ip():
```

```
    # 1) IP allowlist
    if not client_allowed(request.remote_addr or ""):
        return unauthorized("source not in allowlist")

    # 2) Auth: Bearer token
    auth = request.headers.get("Authorization", "")
    if not auth.startswith("Bearer "):
        return unauthorized("missing bearer")
    token = auth.split(" ", 1)[1].strip()
    if token != SECRET:
        return unauthorized("bad token")

    # 3) Require JSON body with 'ip'
    if request.content_type is None or "application/json" not in request.content_type:
        return jsonify({"error": "content-type must be application/json"}), 400

    data = request.get_json(silent=True) or {}
    ip = data.get("ip")
    if not isinstance(ip, str) or not ip:
        return jsonify({"error": "missing or invalid 'ip'"}), 400

    # Optional: quick input sanity (block_ip.py will fully validate again)
    try:
        from ipaddress import ip_address as _ipaddr
        _ipaddr(ip)
    except Exception:
        return jsonify({"error": "invalid ip format"}), 400

    # 4) Execute least-privileged sudo command (no shell)
    try:
        res = subprocess.run(
            ["/usr/bin/sudo", "/usr/bin/python3", "/usr/local/bin/block_ip.py", ip],
            capture_output=True, text=True, timeout=10
        )
        if res.returncode == 0:
            logging.info("Blocked ip via webhook: %s", ip)
            return jsonify({"status": "ok", "ip": ip, "output": res.stdout}), 200
        else:
            logging.error("block_ip.py failed (%s): %s", res.returncode, res.stderr.strip())
            return jsonify({"status": "error", "ip": ip, "output": res.stderr, "code": res.returncode}), 500
    except Exception as e:
        logging.exception("Exception while blocking %s", ip)
        return jsonify({"error": str(e)}), 500

if __name__ == "__main__":
    # Flask HTTPS; debug OFF
```

```
app.run(host="0.0.0.0", port=5001, ssl_context=(CERT_PATH, KEY_PATH))
```

How it Behaves End-to-End

- It initializes a Flask application that listens on port 5001 for incoming HTTPS POST requests.
- Authentication & Authorization:
    - It first checks if the source IP address of the incoming request is within a pre-configured list of allowed networks (ALLOW_NETS). If not, it rejects the request immediately.
    - It then requires an Authorization: Bearer <SECRET> header. The secret token is compared against a securely loaded environment variable. Requests with missing or incorrect tokens are rejected.
- It verifies that the incoming request has a Content-Type of application/json and that the JSON payload contains a valid ip field.
- If all security and validation checks pass, it uses subprocess.run to execute your block_ip.py script, passing the extracted IP address as an argument. It prefixes the command with sudo to ensure block_ip.py has the necessary privileges.
- It captures the output and return code from block_ip.py and sends a JSON response back to the sender, indicating success or failure along with relevant output or error messages.

We'll set the following permissions for the webhook_server.py file and its associated log file:

```
sudo chown root:root /usr/local/bin/webhook_server.py

sudo chmod 755 /usr/local/bin/webhook_server.py

sudo touch /var/log/webhook_server.log

sudo chown webhookd:root /var/log/webhook_server.log

sudo chmod 640 /var/log/webhook_server.log
```

3.2.5. Restricting webhookd sudo permissions

Going back to the account we created for the purpose of running our script, we want to further limit its privileges. We're going to make sure that it can only use sudo for the purpose of executing this script and for nothing else. This way, if the account gets compromised, we can still limit its access to critical functions/files.

We're going to change our sudoers list using:

```
sudo visudo
```

At the bottom of the file, we'll write:

```
Defaults:webhookd !requiretty

Cmnd_Alias BLOCK_CMD = /usr/bin/python3 /usr/local/bin/block_ip.py *

webhookd ALL=(root) NOPASSWD: BLOCK_CMD
```

- **Defaults:webhookd !requiretty:** Prevents sudo from requiring a "tty" (terminal) for webhookd, which is necessary because the Flask app runs in the background without a terminal
- **Cmnd_Alias BLOCK_CMD = /usr/bin/python3 /usr/local/bin/block_ip.py *:** This defines a named alias (BLOCK_CMD) for the exact command that webhookd is allowed to run with sudo. The * at the end means it can accept *any* arguments after the script path
- **webhookd ALL=(root) NOPASSWD: BLOCK_CMD:** This line grants the webhookd user the ability to run BLOCK_CMD (which is your specific block_ip.py script) as the root user ((root)) **without needing a password** (NOPASSWD)

This is highly secure because it creates a very narrow keyhole for sudo access. The webhookd user can only run python3 /usr/local/bin/block_ip.py and nothing else as root.

3.2.6. Setting up a system service to run our script

With our application ready, we can set up a system service that will manage it automatically. Creating a system service allows the script to start automatically whenever we boot up our machine, restart itself if it crashes, and run reliably in the background. This way, our IP blocking system is always running and ready to receive webhooks without manual intervention.

It also allows us to run the script with the permissions and identity of the webhookd user, fully integrating everything we've done so far.

To create the system service, we'll need to make a new file:

```
sudo mousepad /etc/systemd/system/webhook-server.service
```

We'll input the following into the file:

```
[Unit]
Description=HTTPS Webhook Receiver for ELK Auto-Block
After=network-online.target
Wants=network-online.target

[Service]
User=webhookd
Group=webhookd
EnvironmentFile=/etc/webhook_server.env
```

```
ExecStart=/usr/bin/python3 /usr/local/bin/webhook_server.py
WorkingDirectory=/usr/local/bin
Restart=on-failure
RestartSec=2
ReadWritePaths=/var/log/webhook_server.log
NoNewPrivileges=false # Already fixed this
ProtectSystem=true
PrivateTmp=true
ProtectHome=true

[Install]
WantedBy=multi-user.target
```

This defines how your webhook_server.py script will run as a background service.

- **User=webhookd, Group=webhookd:** Ensures the service runs under your unprivileged webhookd user
- **EnvironmentFile=/etc/webhook_server.env:** Tells systemd to load environment variables (like your WEBHOOK_TOKEN) from this file *before* starting the script
- **ExecStart=/usr/bin/python3 /usr/local/bin/webhook_server.py:** The exact command to run your Flask application
- **Restart=on-failure, RestartSec=2:** Configures systemd to automatically restart the service if it crashes, after a 2-second delay
- **ReadWritePaths=/var/log/webhook_server.log:** Grants the webhookd user permission to write to its log file
- **NoNewPrivileges=false, PrivateTmp=true, ProtectSystem=true, ProtectHome=true:** These are **excellent security hardening options** for systemd services. They restrict what the service can do (e.g., prevent it from gaining new privileges, isolate its temporary files, prevent writing to system directories, and prevent writing to user home directories)
    - **Note:** While NoNewPrivileges is a good security control, in this specific case, it directly conflicts with the necessary function of sudo for privilege elevation. Security is still maintained by our highly specific sudoers rule which only allows the webhookd user to run that *one specific command* as root.
    - **Note:** ProtectSystem=true is less restrictive than ProtectSystem=full. While it still makes /usr and /boot read-only, it typically allows writes to certain parts of /etc that are necessary for services to operate, or it doesn't create the same kind of read-only bind over the entirety of /etc as full does. This allows UFW, running with sudo privileges within the service's context, to successfully modify its rules file.

After we enable and restart, our service and script should be running:

```
sudo systemctl daemon-reload
```

```
sudo systemctl enable --now webhook-server
```

```
sudo systemctl status webhook-server --no-pager
```

3.3 Configure Kibana Alert Webhook

Let’s finally go back to Kibana and configure the alert webhook so that it sends the appropriate data needed to interact with our system.

Go to Security > Rules > SSH Brute Force Attempt > Edit rule settings > Actions. We’ll set our actions frequency to ‘On each rule execution’ so that a POST is sent to our server as soon as an alert is generated.

Select ‘Webhook’ and configure the connector details:

- **Name:** Give your connector a descriptive name (e.g., "Auto-Block IP Webhook")
- **URL:** Enter the full URL of your webhook server, including the port and path
    - Example: https://< VM_IP>:5001/blockip
    - Replace < VM_IP> with the actual IP address of the server hosting your webhook server
- **Method:** Select **POST**
- **Headers:** Click "Add HTTP header" and add the following two headers:
    - **Header 1:**
        - **Name:** Content-Type
        - **Value:** application/json
    - **Header 2:**
        - **Name:** Authorization
        - **Value:** Bearer <PASTE_YOUR_SECRET_API_KEY_HERE>
        - **Important:** Replace <PASTE_YOUR_SECRET_API_KEY_HERE> with the exact secret token you generated earlier and saved in /etc/webhook_server.env

Configure the Body (Payload):

- In the **Body** section, you'll define the JSON payload that Kibana sends to your webhook. Your Flask server expects a JSON object with a key named ip
- Use Kibana's syntax to extract the source IP from your alert context. A common field for this is context.source.ip
- Paste the following JSON into the Body field:
    - { "ip": "{{context.source.ip}}" }
    - **Note:** If your Kibana alert rule uses a different field for the source IP (e.g., context.alerts.0.source.ip or a custom field), you'll need to adjust {{context.source.ip}} accordingly to match your alert's context

SSL/TLS Considerations:

- Since we're using a self-signed SSL certificate, Kibana might not trust it by default
- To fix this, we should import your webhook.crt file into Kibana's trust store

---

4. Conclusion

This project demonstrates how a Security Information and Event Management (SIEM) platform can be extended beyond passive monitoring into an active defense system. By building a pipeline that detects brute force SSH attempts, generates alerts in Kibana, and automatically blocks offending IPs through a secured webhook integration, I showed the full lifecycle of modern security operations:

collect → detect → respond → harden.

Beyond just making automation work, I incorporated multiple security layers into the design. TLS encryption, token-based authentication, IP allowlisting, principle of least privilege with sudoers, strict input validation, and centralized logging all work together to reduce the attack surface and mitigate risks such as command injection, unauthorized access, or rule spamming.

The result is a lightweight but realistic lab implementation of SIEM-driven automated response. While simplified compared to enterprise SOAR platforms, it captures the same philosophy: rapid detection, minimal analyst intervention, and layered defenses against abuse of automation.

This project not only highlights technical skills in ELK stack management, log ingestion, and Python scripting, but also reflects a security engineer's mindset—balancing operational functionality with risk analysis and secure design principles.

Appendix – 1

Detailed Breakdown of block_ip.py

shebang line

```
#!/usr/bin/env python3
```

- Tells the OS to run this file with the system's python3 interpreter

Libraries

```
import sys
import subprocess
import ipaddress
import logging
```

- sys: lets you read command-line arguments (sys.argv) and set exit codes
- subprocess: runs external commands (you'll call ufw through this)
- ipaddress: validates that a string is a real IPv4/IPv6 address
- logging: writes events to a log file for auditing/troubleshooting

Setting up a log file

```
logging.basicConfig(
    filename='/var/log/block_ip.log',
    level=logging.INFO,
    format='%(asctime)s %(levelname)s %(message)s'
)
```

- Sets up a log file at /var/log/block_ip.log
- level=INFO means INFO and above (INFO, WARNING, ERROR) will be recorded
- format controls each log line: timestamp, severity, then your message

Validating IP format

```
def is_valid_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError:
        return False
```

- Defining a function that tries to parse ip as an IP address
- If it parses, it's valid → return True; if not, a ValueError is thrown and you return False

Checking to see if IP is already blocked by ufw

```
def already_blocked(ip):
    # Check ufw status for the IP (returns True if present)
    proc = subprocess.run(["ufw", "status", "numbered"], capture_output=True, text=True)
```

```
return ip in proc.stdout
```

- Defines a function that will run 'ufw status numbered' and captures its output as text
- Returns True if the IP string appears anywhere in that output (meaning there's already a matching rule)
    - Note: this is a simple substring check; it should be fine for most cases but could be problematic if ufw's output structure changes i.e. adds or removes spaces, etc.

Creating blocking function 1 – Log an error if an IP cannot be parsed

```
def block_ip(ip):
    if not is_valid_ip(ip):
        logging.error("Invalid IP: %s", ip)
        print(f"Invalid IP: {ip}")
        return 1
```

- If the input isn't a proper IP:
    - Log an ERROR
    - Tell the user on stdout
    - Return exit code 1 (bad input)

Creating blocking function 2 – Log an info alert if IP is already blocked

```
    if already_blocked(ip):
        logging.info("IP already blocked: %s", ip)
        print(f"IP already blocked: {ip}")
        return 0
```

- If a UFW rule already exists for this IP:
    - Log INFO
    - Inform the user
    - Return 0 (success/no change)

Creating blocking function 3 – Creating a ufw Deny rule if the ip is valid and not blocked yet

```
    try:
        # Run ufw as the current user (expecting caller to use sudo if necessary)
        subprocess.run(["ufw", "deny", "from", ip], check=True)
        logging.info("Blocked IP: %s", ip)
        print(f"Blocked IP: {ip}")
        return 0
```

- Tries to add a deny rule with ufw deny from <ip>
- check=True makes Python raise an error if ufw exits non-zero
- On success, logs INFO, prints a confirmation, and returns 0

Creating blocking function 4 – What if ufw Deny rule creation fails?

```
    except subprocess.CalledProcessError as e:
        logging.exception("Failed to block IP %s: %s", ip, e)
        print(f"Failed to block IP {ip}: {e}")
        return 2
```

- If ufw failed (e.g., not run with sufficient privileges), log the full exception (stack trace) and print an error
- Return 2 to indicate an execution failure

Executing the function

```
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: block_ip.py <IP>")
        sys.exit(1)
    sys.exit(block_ip(sys.argv[1]))
```

- Runs only when the file is executed directly (not imported)
- Requires exactly one argument (the IP). If missing, print usage and exit 1
- Otherwise call block_ip() function with the provided IP and exit with whatever code it returns (0, 1, or 2)

Appendix – 2

Detailed Breakdown of webhook_server.py

shebang line

```
#!/usr/bin/env python3
```

Imports

```
import os
import logging
from ipaddress import ip_address, ip_network
from flask import Flask, request, jsonify
import subprocess
```

- **import os:** This module provides a way to interact with the operating system, like accessing environment variables. We'll use this to securely load your WEBHOOK_TOKEN.
- **import logging:** This is for recording events and errors. It's crucial for understanding what your server is doing and for debugging.
- **from ipaddress import ip_address, ip_network:** These are Python utilities specifically for working with IP addresses and network ranges. We'll use them to validate IPs and check if an incoming request's source IP is from an allowed network.
- **from flask import Flask, request, jsonify:** These are the core components from the Flask web framework.
    - Flask: The main object that represents your web application.
    - request: An object that holds all the details about an incoming web request (like headers, body data, and the sender's IP).
    - jsonify: A helper function to create JSON responses, which is standard for web APIs.
- **import subprocess:** This module allows your Python script to run other programs on your system. This is how your Flask app executes the block_ip.py script.

Configuration

```
# --- Config ---
SECRET = os.getenv("WEBHOOK_TOKEN", "")
ALLOW_NETS = [
   "127.0.0.0/8",
   "::1/128",
   "192.168.0.0/16",
   "10.0.0.0/8"
]
CERT_PATH = "/etc/ssl/webhook/webhook.crt"
KEY_PATH  = "/etc/ssl/webhook/webhook.key"
LOG_PATH  = "/var/log/webhook_server.log"

# --- App & logging ---
```

```
app = Flask(__name__)
logging.basicConfig(filename=LOG_PATH, level=logging.INFO, format="%(asctime)s %(levelname)s %(message)s")
```

- **SECRET = os.getenv("WEBHOOK_TOKEN", ""):** This line defines your secret token for authentication. Instead of hardcoding it, it's loaded from an environment variable named WEBHOOK_TOKEN. This is a secure way to handle secrets, as systemd will inject this variable into the service's environment without exposing it directly in the script.
- **ALLOW_NETS = [...]:** This is your IP allowlist. It's a list of network ranges (in CIDR format) that are permitted to send requests to your webhook. Any request coming from an IP outside these ranges will be blocked.
- **CERT_PATH, KEY_PATH:** These variables store the file paths for your HTTPS SSL certificate and private key. Flask uses these to enable secure, encrypted communication (HTTPS).\
- **LOG_PATH:** This specifies the file path where your Flask app's logs will be written.
- **app = Flask(__name__):** This line creates your main Flask web application instance.
- **logging.basicConfig(...):** This sets up the logging system for this specific Flask application, directing its output to the LOG_PATH file with informational messages.

Helper functions

```
def client_allowed(remote_ip: str) -> bool:
    try:
        rip = ip_address(remote_ip)
        for cidr in ALLOW_NETS:
            if rip in ip_network(cidr, strict=False):
                return True
    except Exception:
        return False
    return False

def unauthorized(reason: str):
    logging.warning("Unauthorized: %s (from %s)", reason, request.remote_addr)
    return jsonify({"error": "unauthorized", "reason": reason}), 401
```

- **client_allowed(remote_ip):** This function is a security gate. It takes the IP address of the incoming request (remote_ip), converts it to an IP object, and then checks if that IP falls within any of the ALLOW_NETS defined in your configuration. If it's a match, it returns True; otherwise, False. It's your first line of defense.
- **unauthorized(reason):** This helper function simplifies handling unauthorized requests. If a request fails an authentication or authorization check (e.g., wrong token, not on allowlist), this function logs a warning message and returns a standard JSON error response with an HTTP 401 (Unauthorized) status code.

Webhook route (/blockip)

```
@app.route("/blockip", methods=["POST"])
def block_ip():
    # 1) IP allowlist
    if not client_allowed(request.remote_addr or ""):
```

```
    return unauthorized("source not in allowlist")

# 2) Auth: Bearer token
auth = request.headers.get("Authorization", "")
if not auth.startswith("Bearer "):
    return unauthorized("missing bearer")
token = auth.split(" ", 1)[1].strip()
if token != SECRET:
    return unauthorized("bad token")

# 3) Require JSON body with 'ip'
if request.content_type is None or "application/json" not in request.content_type:
    return jsonify({"error": "content-type must be application/json"}), 400

data = request.get_json(silent=True) or {}
ip = data.get("ip")
if not isinstance(ip, str) or not ip:
    return jsonify({"error": "missing or invalid 'ip'"}), 400

# Optional: quick input sanity (block_ip.py will fully validate again)
try:
    from ipaddress import ip_address as _ipaddr
    _ipaddr(ip)
except Exception:
    return jsonify({"error": "invalid ip format"}), 400

# 4) Execute least-privileged sudo command (no shell)
try:
    res = subprocess.run(
        ["/usr/bin/sudo", "/usr/bin/python3", "/usr/local/bin/block_ip.py", ip],
        capture_output=True, text=True, timeout=10
    )
    if res.returncode == 0:
        logging.info("Blocked ip via webhook: %s", ip)
        return jsonify({"status": "ok", "ip": ip, "output": res.stdout}), 200
    else:
        logging.error("block_ip.py failed (%s): %s", res.returncode, res.stderr.strip())
        return jsonify({"status": "error", "ip": ip, "output": res.stderr, "code": res.returncode}), 500
except Exception as e:
    logging.exception("Exception while blocking %s", ip)
    return jsonify({"error": str(e)}), 500
```

- **@app.route("/blockip", methods=["POST"]):** This is a Flask decorator that tells your Flask app: "When a POST request comes to the /blockip URL, run the block_ip() function."
- **def block_ip():** This is the main function that gets executed for each incoming webhook.
- Security Checks (Layers of Defense):

    - **IP Allowlist Check (if not client_allowed(...)):** First, it checks if the sender's IP is allowed using the client_allowed helper. If not, it immediately returns an "Unauthorized" error.
    - Bearer Token Authentication: It then inspects the Authorization header to ensure it starts with "Bearer " and that the provided token matches your SECRET. If not, it also returns "Unauthorized."
- Input Validation:
    - It checks that the request's Content-Type is application/json.
    - It tries to get the JSON data and ensures there's an ip field, and that it's a non-empty string.
    - An optional quick sanity check on the IP format is performed using ip_address to catch clearly invalid IPs early.
- Executing the Blocking Script:
    - **subprocess.run([...]):** This is where the magic happens! It executes your block_ip.py script. The key here is the sudo command.
    - Privilege Elevation (sudo): The webhook_server.py script (running as the unprivileged webhookd user) uses sudo to run block_ip.py with root privileges. This works because your /etc/sudoers file has a very specific rule allowing webhookd to execute only that particular Python script as root, and nothing else.
    - **capture_output=True, text=True, timeout=10:** These arguments ensure the output from block_ip.py is captured, treated as text, and that the command doesn't run indefinitely.
- Error Handling and Responses:
    - **if res.returncode == 0::** If block_ip.py finishes successfully (return code 0), the webhook server logs a success message and sends a JSON success response (HTTP 200 OK) back to the sender, including the output from block_ip.py.
    - **else::** If block_ip.py fails (non-zero return code), it logs an error, captures the error output from block_ip.py (res.stderr), and sends a JSON error response (HTTP 500 Internal Server Error).
    - **except Exception as e::** A general try-except block catches any other unexpected errors during the subprocess call, logs them, and returns a generic 500 error.

Main execution block

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5001, ssl_context=(CERT_PATH, KEY_PATH))
```

- **if __name__ == "__main__"::** This is a standard Python idiom. It means the code inside this block will only run when the script is executed directly (e.g., when systemd starts it), not if it's imported as a module into another script.
- **app.run(...):** This is the Flask command that starts the web server.
    - host="0.0.0.0": Tells the server to listen on all available network interfaces, making it accessible from other machines.
    - port=5001: The port on which the server will listen for incoming connections.
    - ssl_context=(CERT_PATH, KEY_PATH): Configures the server to use HTTPS, encrypting all communication using the specified SSL certificate and private key files.