Ministry of Higher Education

And Scientific Research

ACADEMIC YEAR

2024/2025

EPI

Private International Higher School of Polytechnic

END OF YEAR PROJECT

Field of study : Cyber Security

Entitled

Detection and Response to Server Abuses in a Cloud Environment

Internship place	Author	Supervisor
Epi digital school	Hamza Ghebrich	Mr. Rouatbi Adnen

End of Year Project Report

Detection and Response to Server Abuses in a Cloud Environment

Prepared by:
Hamza Ghebrich

Date:
12 Mai 2025

Abstract

This project aims to design and develop a solution capable of detecting and automatically responding to server abuses in a cloud environment. By leveraging system and network logs generated by cloud services such as AWS, this solution will identify suspicious behaviors, such as unauthorized login attempts or excessive resource usage. Once anomalies are detected, automated actions will be executed to mitigate threats, such as blocking malicious IP addresses or isolating affected instances.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Mr. Adnen Rouatbi, for his continuous support, insightful guidance, and valuable feedback throughout the duration of this project. His expertise and encouragement greatly contributed to the successful completion of this work.
I also wish to thank all the professors and staff members of my academic institution for their dedication, knowledge, and support during my studies.
Finally, I am sincerely grateful to my family and friends for their constant encouragement, patience, and unwavering support.

General Introduction

In today's digital age, cloud computing has become the backbone of modern IT infrastructures, offering scalability, flexibility, and cost-efficiency. Organizations increasingly rely on cloud platforms such as Amazon Web Services (AWS) to host applications, store data, and run critical workloads. However, this growing dependence on cloud environments introduces significant cybersecurity challenges. With dynamic resource allocation, public accessibility, and the complexity of configurations, cloud systems become prime targets for cyberattacks and server abuses.

This project aims to design and implement a practical solution capable of detecting and automatically responding to server abuses in a cloud environment. These abuses may include unauthorized access attempts, excessive resource consumption, denial-of-service (DoS) attacks, or other forms of suspicious activity that threaten the integrity, availability, and confidentiality of cloud-hosted services.

The proposed solution leverages various tools and technologies to achieve its objectives. Infrastructure is provisioned and managed using Infrastructure as Code (IaC) tools like Terraform and Ansible. System and network monitoring are performed using AWS services such as CloudWatch and the ELK (Elasticsearch, Logstash, Kibana) stack. Detection mechanisms are based on the analysis of logs and performance metrics, and once anomalies are identified, automated response actions—such as blocking IP addresses or isolating compromised instances—are triggered to mitigate the impact.

This report documents the design, implementation, and evaluation of the solution. It explores the challenges of securing cloud infrastructure, the architecture of the proposed system, the configuration of monitoring and detection tools, and the effectiveness of automated response mechanisms.

By the end of this work, we demonstrate how automation and monitoring can significantly enhance the security posture of cloud environments and enable proactive responses to cyber threats in real time.

Contents

List of Figures

Chapter 1

General Context

Introduction

In today's digital era, organizations increasingly depend on cloud computing to host applications, store sensitive data, and scale operations dynamically. While cloud platforms such as Amazon Web Services (AWS) offer significant flexibility and cost-efficiency, they also introduce complex security challenges. Cloud infrastructures are exposed to a wide range of threats, including unauthorized access, brute-force attacks, and misuse of computing resources. In this context, ensuring the security and resilience of cloud environments becomes a critical concern.

This chapter presents the general context of a project that addresses the detection and automated response to server abuses in a cloud setting. It introduces the motivation behind the work, outlines the key objectives, and provides an overview of the core technologies used to build a secure and responsive cloud environment. Through the integration of infrastructure-as-code tools, monitoring systems, log analysis platforms, and automated scripts, the project aims to deliver a robust solution capable of identifying and mitigating threats in real-time.

1.1 Background and Motivation

The increasing reliance on cloud services presents new challenges in securing computing resources against abuse. With dynamic and scalable environments, cloud infrastructures are often targeted by malicious actors, necessitating robust detection and response mechanisms.

Malicious activities, such as unauthorized access, data breaches, and resource abuse, can lead to significant financial losses and reputational damage. Therefore, it is vital to implement proactive measures that not only detect these threats in real-time but also respond effectively to mitigate their impact

1.2 Project Objectives

The primary objectives of this project are to:

- Develop a solution to detect and respond to server abuses in a cloud environment, focusing on real-time threat detection.
- Utilize system and network logs to identify suspicious behaviors, employing advanced analytics to uncover patterns indicative of potential abuse.
- Implement automated response actions to mitigate identified threats, ensuring minimal disruption to services while maintaining security.
- Create a user-friendly dashboard for monitoring and visualizing detected incidents.

1.3 Technology Overview

This project integrates several key technologies and tools as part of its detection and response pipeline in a cloud environment. These include Terraform, Ansible, CloudWatch, the ELK stack, and Python scripting. Below is an explanation of each tool and its role in the project.

Terraform: Infrastructure as Code

Terraform is used to automate the provisioning and configuration of the cloud infrastructure. With Terraform, we define cloud resources such as EC2 instances, security groups, CloudWatch alarms, and SNS topics as code. This ensures reproducibility, version control, and ease of deployment. [1]

Figure 1.1: Terraform Logo

Ansible: Configuration Management

Ansible automates the installation and configuration of software on the EC2 instance. It deploys and manages the LAMP stack, security tools like Fail2Ban, monitoring agents like Prometheus and Grafana, and custom Python scripts. Ansible ensures consistent and repeatable configurations across environments.[2]

Figure 1.2: Ansible Logo

CloudWatch: Monitoring and Alerts

AWS CloudWatch provides monitoring for EC2 instances. It collects metrics like CPU and memory usage, and generates alarms based on thresholds. When an alarm is triggered, it sends notifications via Amazon SNS to alert administrators in real-time.[7]

Figure 1.3: CloudWatch Logo

ELK Stack: Log Management and Visualization

The ELK stack (Elasticsearch, Logstash, and Kibana) is used to centralize and analyze system and application logs. This allows for real-time detection of anomalies and generation of visual dashboards that aid in understanding attack patterns and system behavior.[11]

Figure 1.4: ELK Stack Logo

Python: Automation and Detection

Custom Python scripts are developed to automate the detection of abuse (e.g., failed login attempts) and execute response actions (e.g., blocking IPs via AWS API). These scripts use the Boto3 library to interact with AWS services programmatically.

Figure 1.5: Python Logo

1.4 General Architecture

The architecture designed for detecting and responding to server abuse in a cloud environment is built on Amazon Web Services (AWS) and emphasizes automation, monitoring, and scalability. It uses Terraform for provisioning infrastructure as code, and Ansible for configuring virtual machines and deploying necessary services. Monitoring and log collection are handled via CloudWatch and the ELK Stack (Elasticsearch, Logstash, Kibana), which centralizes logs and enables real-time visualization and analysis. Detection mechanisms rely on CloudWatch alarms triggered by suspicious activity, such as brute-force or DoS attacks simulated using tools like Hydra and stress-ng. Automated response actions, such as blocking IPs or sending alerts, are executed via Python scripts or AWS Lambda functions. This modular and extensible design allows for future integration of AI-based detection components and supports a robust cloud-based security posture. Figure **??** illustrates the full architecture of the system.

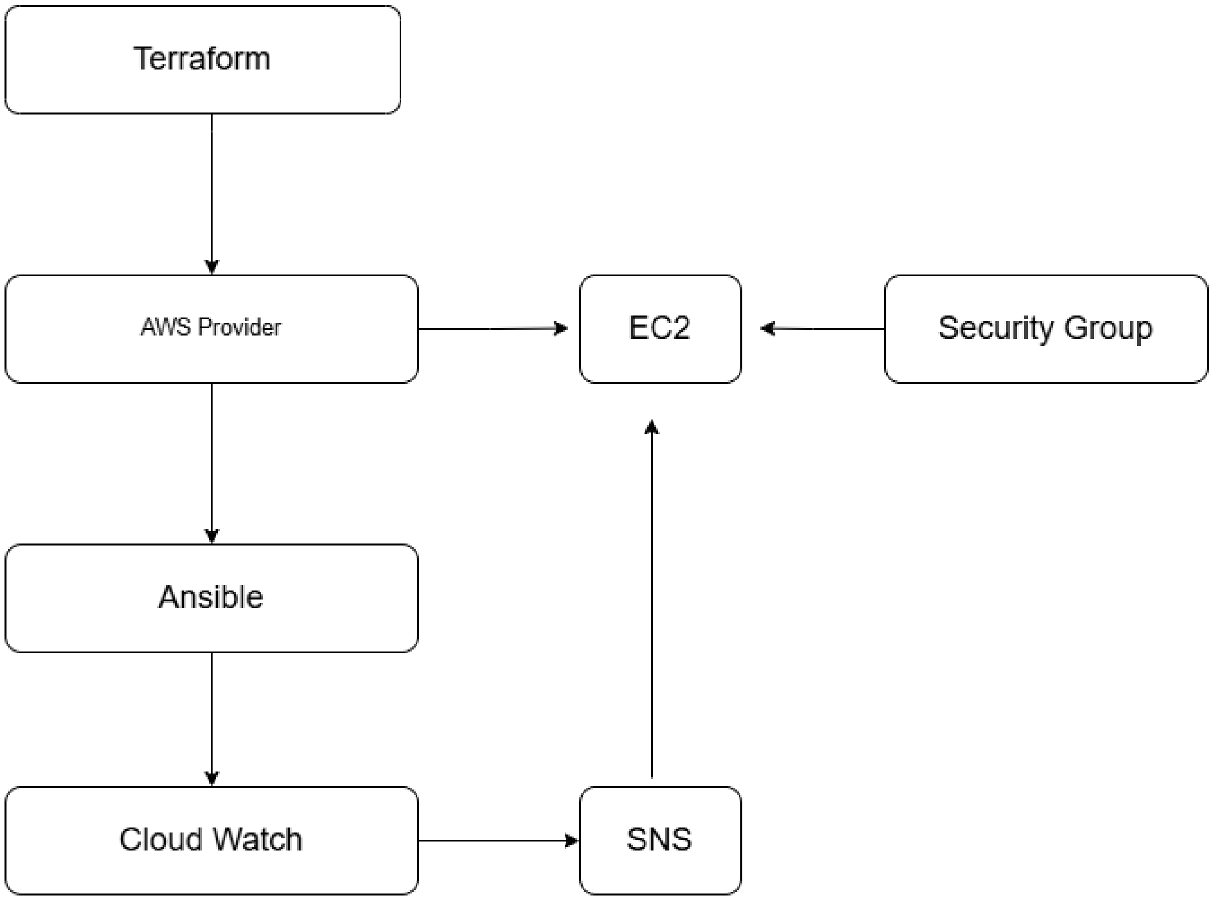


Figure 1.6: Global Project Architecture Overview

Challenge	**Description**
Infrastructure Provisioning	Managing complex infrastructure components consistently using Infrastructure as Code (IaC) with Terraform.
Automated Configuration	Ensuring reliable and secure system configuration using Ansible to deploy services and enforce system states.
Monitoring	Collecting and aggregating logs from distributed systems with CloudWatch and the ELK Stack.
Real-Time Detection	Setting up CloudWatch alarms to trigger on patterns of abuse like brute-force or DoS attacks.
Automated Response	Reacting to threats using automated scripts (Python or AWS Lambda) to block IPs, notify admins, or isolate systems.
Scalability	Designing a modular architecture that supports future growth and integration with AI-based detection models.
Data Visualization	Presenting log data meaningfully with dashboards in Kibana for real-time analysis and decision-making.

Table 1.1: Challenges and Their Descriptions in Cloud-Based Abuse Detection and Response

Project Timeline (Gantt-style Table)

The following table outlines the key milestones and activities of the project over the planned timeline. Each row represents a specific week with associated objectives and deliverables.

Timeline (Week)	**Project Activities**
February (Pre-start)	Project launch, kickoff meeting with initial objective clarification.
March - Week 1	Requirements analysis, threat modeling, and risk assessment for cloud infrastructure.
March - Week 2	Design of the cloud-based architecture and security layers (VPC, subnets, IAM).
March - Week 3	Implementation of infrastructure-as-code using Terraform: - Define VPC, subnets, EC2 instances - Use variables and outputs - Initialize and apply infrastructure
March - Week 4	Deployment and configuration automation using Ansible: - Install necessary packages - Configure services (Elastic, Filebeat, etc.) - Ensure repeatable and idempotent setup
April - Week 1	Deploy ELK stack (Elasticsearch, Logstash, Kibana) Configure log collection and alert system with Filebeat + Logstash.
April - Week 2	Launch attack simulation tools: - Dictionary attacks using Hydra - Resource abuse via stress-ng Monitor responses in real-time via logs and dashboards.
April - Week 3	Create and integrate automated response scripts: - Detect anomalies via logs - Trigger block or alert actions
April - Week 4	Integrate Grafana for advanced visualization. Start writing technical documentation, preparing result analysis.
May - Week 1	Finalize documentation, review and verify configurations, prepare final presentation and delivery.

Conclusion

This chapter has established the foundational context for the project by outlining the motivations, objectives, and technical environment required to build an intelligent detection and response system for cloud-based server abuse. The integration of tools such as Terraform, Ansible, CloudWatch, ELK Stack, and Python scripts demonstrates the project's commitment to automation, scalability, and real-time threat handling.

The architectural overview and technology stack described herein set the stage for the subsequent implementation chapters, where these components will be orchestrated to detect abnormal behaviors, trigger alerts, and enforce countermeasures automatically. By combining infrastructure automation with data-driven security insights, the project aims to contribute a practical and extensible approach to modern cloud security.

Chapter 2

State of the Art

Introduction

In the world of cloud computing and infrastructure automation, tools like Terraform and Ansible have become indispensable in creating robust, scalable, and reproducible environments. In this chapter, we will explore the role of these tools in our project, specifically in provisioning cloud resources and managing configurations, while ensuring a seamless, error-free deployment process.

Initially, the cloud infrastructure was manually configured using the AWS Management Console, but due to the increasing complexity of the setup and the need for consistency across multiple environments, we turned to Terraform for infrastructure provisioning. Once the infrastructure was set up, Ansible was employed to configure and manage the software environments, making the process more efficient and standardized. The use of both tools not only simplified the deployment but also enhanced the scalability of the system.

This chapter delves into the methodology behind using Terraform and Ansible, providing a detailed comparison of manual versus automated setups, and explaining how these tools fit together in our infrastructure lifecycle. We will also go over the practical steps taken to automate the provisioning of cloud resources and the configuration of software components, providing a seamless experience from start to finish.

2.1 Cloud Environment Setup with Terraform

2.1.1 Manual Configuration (via AWS Console)

To establish a foundational understanding of the process, the initial setup of the cloud environment was done manually through the AWS Management Console. The manual steps involved creating key pairs, setting up security groups, launching EC2 instances, configuring CloudWatch alarms, and creating SNS topics for real-time notifications. These tasks, although straightforward, are time-consuming and error-prone when performed repetitively across environments.

- Creating an SSH Key Pair manually from EC2 > Key Pairs.
- Setting up a Security Group with rules for SSH (22), HTTP (80), Grafana (3000), and Prometheus (9090).
- Launching an EC2 instance using the Debian AMI, selecting the appropriate VPC, subnet, and security group.
- Configuring CloudWatch alarms for CPU and memory thresholds.
- Creating an SNS topic and subscribing an email endpoint for notifications.

2.1.2 Automated Configuration with Terraform

Once we gained familiarity with the manual process, we transitioned to an automated approach using Terraform. Terraform was chosen for its ability to define infrastructure as code, making it easy to manage, version, and reproduce across multiple environments.

In this project, Terraform was used to automate the provisioning of resources such as EC2 instances, security groups, IAM roles, CloudWatch alarms, and SNS topics. By leveraging Terraform's declarative configuration language, we could define and deploy the entire infrastructure in one unified workflow, ensuring that it could be reliably reproduced in the future with minimal manual intervention.

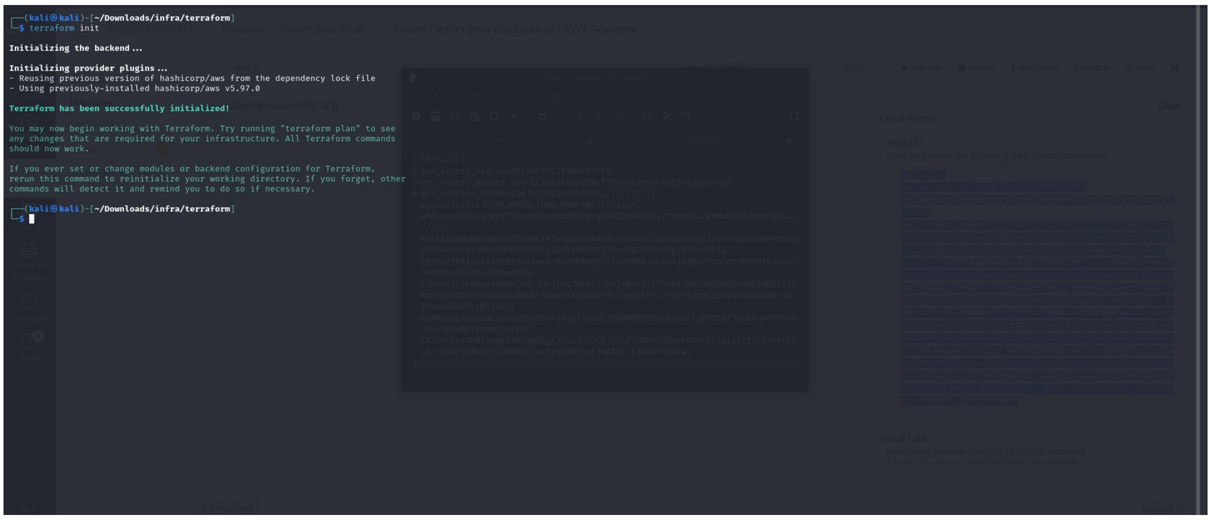
The Terraform configuration files were designed to:

- Automatically create key pairs and store them locally.
- Provision security groups with necessary open ports for SSH, HTTP, Grafana, and Prometheus.
- Launch EC2 instances with pre-defined configurations, including tags and network settings.
- Attach IAM roles to the instances for logging purposes.
- Set up CloudWatch alarms for monitoring system performance.
- Define SNS topics for alerting and subscribe email endpoints to receive notifications.
- Output instance details and generate an inventory file for use in configuration management with Ansible.

The deployment process with Terraform was highly efficient and reproducible. The Terraform plan and apply steps ensured that each infrastructure change was tracked, validated, and executed in a controlled manner.

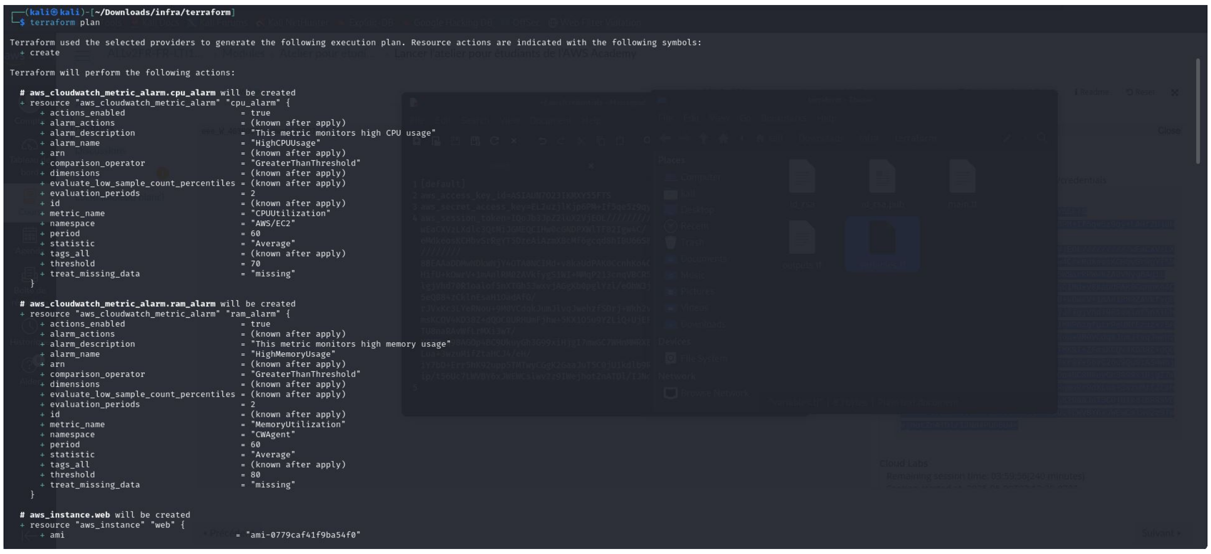
2.1.3 Validation and Execution of Terraform Configuration

To ensure that the resources were correctly provisioned, Terraform's validation and planning phases were thoroughly utilized. By executing ‘terraform init‘, ‘terraform plan‘, and ‘terraform apply‘, we validated the configuration, previewed the changes, and applied the changes in a controlled manner.



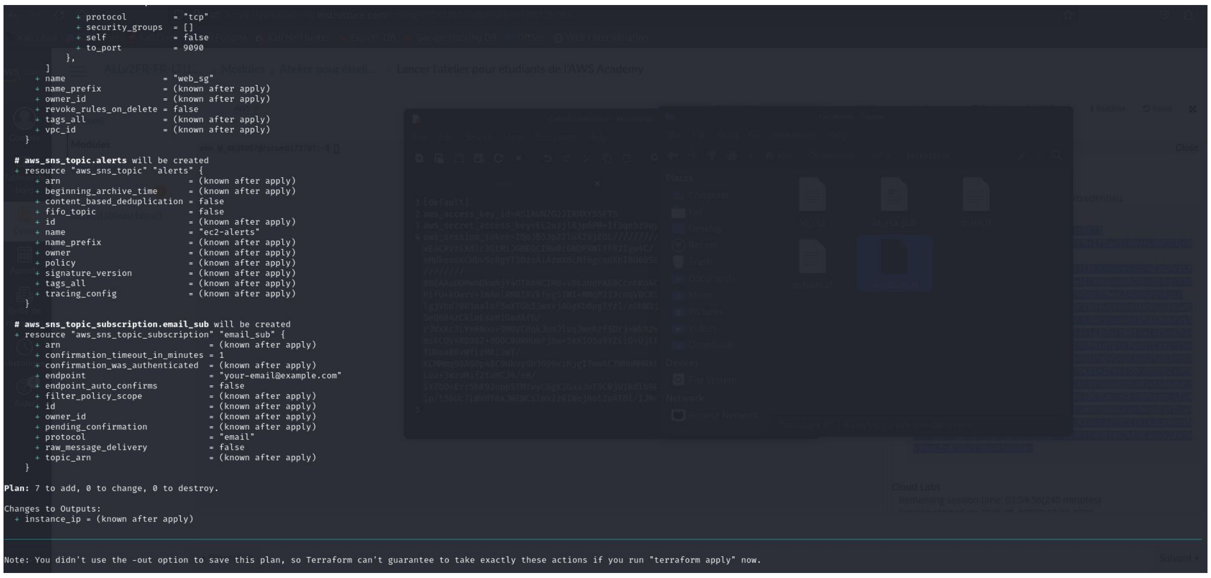
```
┌──(kali㉿kali)-[~/Downloads/infra/terraform]
└─$ terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v5.97.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.

┌──(kali㉿kali)-[~/Downloads/infra/terraform]
└─$ 
```

Figure 2.1: Execution Result of `terraform init`



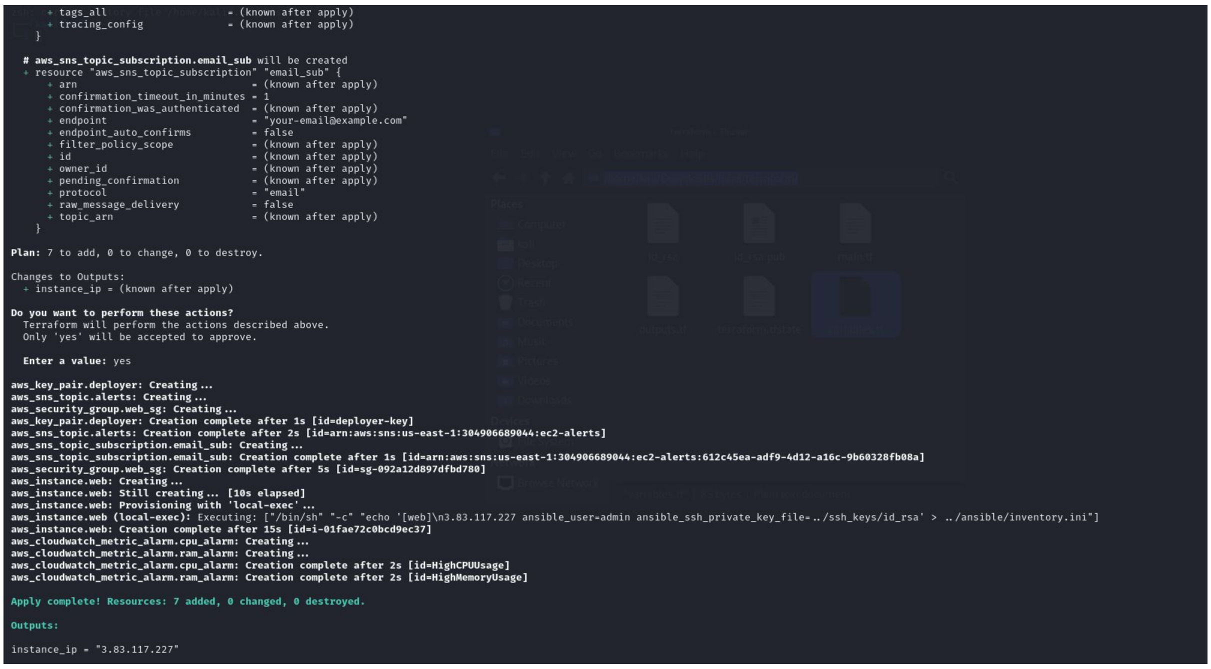
```
┌──(kali㉿kali)-[~/Downloads/infra/terraform]
└─$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_cloudwatch_metric_alarm.cpu_alarm will be created
  + resource "aws_cloudwatch_metric_alarm" "cpu_alarm" {
      + actions_enabled                       = true
      + alarm_actions                         = (known after apply)
      + alarm_description                     = "This metric monitors high CPU usage"
      + alarm_name                            = "HighCPUUsage"
      + arn                                   = (known after apply)
      + comparison_operator                   = "GreaterThanThreshold"
      + dimensions                            = (known after apply)
      + evaluate_low_sample_count_percentiles = (known after apply)
      + evaluation_periods                    = 2
      + id                                    = (known after apply)
      + metric_name                           = "CPUUtilization"
      + namespace                             = "AWS/EC2"
      + period                                = 60
      + statistic                             = "Average"
      + tags_all                              = (known after apply)
      + threshold                             = 70
      + treat_missing_data                    = "missing"
    }

  # aws_cloudwatch_metric_alarm.ram_alarm will be created
  + resource "aws_cloudwatch_metric_alarm" "ram_alarm" {
      + actions_enabled                       = true
      + alarm_actions                         = (known after apply)
      + alarm_description                     = "This metric monitors high memory usage"
      + alarm_name                            = "HighMemoryUsage"
      + arn                                   = (known after apply)
      + comparison_operator                   = "GreaterThanThreshold"
      + dimensions                            = (known after apply)
      + evaluate_low_sample_count_percentiles = (known after apply)
      + evaluation_periods                    = 2
      + id                                    = (known after apply)
      + metric_name                           = "MemoryUtilization"
      + namespace                             = "CWAgent"
      + period                                = 60
      + statistic                             = "Average"
      + tags_all                              = (known after apply)
      + threshold                             = 80
      + treat_missing_data                    = "missing"
    }

  # aws_instance.web will be created
  + resource "aws_instance" "web" {
      + ami                                  = "ami-0779caf41f9ba54f0"
```

Figure 2.2: Execution Result of `terraform plan (1)`



```
          + protocol        = "tcp"
          + security_groups = []
          + self            = false
          + to_port         = 9090
        },
    ]
  + name                   = "web_sg"
  + name_prefix            = (known after apply)
  + owner_id               = (known after apply)
  + revoke_rules_on_delete = false
  + tags_all               = (known after apply)
  + vpc_id                 = (known after apply)
}

# aws_sns_topic.alerts will be created
+ resource "aws_sns_topic" "alerts" {
  + arn                         = (known after apply)
  + beginning_archive_time      = (known after apply)
  + content_based_deduplication = false
  + fifo_topic                  = false
  + id                          = (known after apply)
  + name                        = "ec2-alerts"
  + name_prefix                 = (known after apply)
  + owner                       = (known after apply)
  + policy                      = (known after apply)
  + signature_version           = (known after apply)
  + tags_all                    = (known after apply)
  + tracing_config              = (known after apply)
}

# aws_sns_topic_subscription.email_sub will be created
+ resource "aws_sns_topic_subscription" "email_sub" {
  + arn                             = (known after apply)
  + confirmation_timeout_in_minutes = 1
  + confirmation_was_authenticated  = (known after apply)
  + endpoint                        = "your-email@example.com"
  + endpoint_auto_confirms          = false
  + filter_policy_scope             = (known after apply)
  + id                              = (known after apply)
  + owner_id                        = (known after apply)
  + pending_confirmation            = (known after apply)
  + protocol                        = "email"
  + raw_message_delivery            = false
  + topic_arn                       = (known after apply)
}

Plan: 7 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + instance_ip = (known after apply)

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.
```

Figure 2.3: Execution Result of `terraform plan (2)`



```
      + tags_all                    = (known after apply)
      + tracing_config              = (known after apply)
    }

  # aws_sns_topic_subscription.email_sub will be created
  + resource "aws_sns_topic_subscription" "email_sub" {
      + arn                             = (known after apply)
      + confirmation_timeout_in_minutes = 1
      + confirmation_was_authenticated  = (known after apply)
      + endpoint                        = "your-email@example.com"
      + endpoint_auto_confirms          = false
      + filter_policy_scope             = (known after apply)
      + id                              = (known after apply)
      + owner_id                        = (known after apply)
      + pending_confirmation            = (known after apply)
      + protocol                        = "email"
      + raw_message_delivery            = false
      + topic_arn                       = (known after apply)
    }

Plan: 7 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + instance_ip = (known after apply)

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_key_pair.deployer: Creating...
aws_sns_topic.alerts: Creating...
aws_security_group.web_sg: Creating...
aws_key_pair.deployer: Creation complete after 1s [id=deployer-key]
aws_sns_topic.alerts: Creation complete after 2s [id=arn:aws:sns:us-east-1:304906689044:ec2-alerts]
aws_sns_topic_subscription.email_sub: Creating...
aws_sns_topic_subscription.email_sub: Creation complete after 1s [id=arn:aws:sns:us-east-1:304906689044:ec2-alerts:612c45ea-adf9-4d12-a16c-9b60328fb08a]
aws_security_group.web_sg: Creation complete after 5s [id=sg-092a12d897dfbd780]
aws_instance.web: Creating...
aws_instance.web: Still creating... [10s elapsed]
aws_instance.web: Provisioning with 'local-exec'...
aws_instance.web (local-exec): Executing: ["/bin/sh" "-c" "echo '[web]\n3.83.117.227 ansible_user=admin ansible_ssh_private_key_file=../ssh_keys/id_rsa' > ../ansible/inventory.ini"]
aws_instance.web: Creation complete after 15s [id=i-01fae72c0bcd9ec37]
aws_cloudwatch_metric_alarm.cpu_alarm: Creating...
aws_cloudwatch_metric_alarm.ram_alarm: Creating...
aws_cloudwatch_metric_alarm.cpu_alarm: Creation complete after 2s [id=HighCPUUsage]
aws_cloudwatch_metric_alarm.ram_alarm: Creation complete after 2s [id=HighMemoryUsage]

Apply complete! Resources: 7 added, 0 changed, 0 destroyed.

Outputs:

instance_ip = "3.83.117.227"
```

Figure 2.4: Execution Result of `terraform apply`

2.2 Provisioning and Configuration with Ansible

2.2.1 Manual Configuration of Software

Following the infrastructure provisioning with Terraform, the next task was to install and configure various software components, such as Apache, MariaDB, PHP, Fail2Ban, Prometheus, and Grafana. Traditionally, these tasks would require logging into each server and executing a series of commands manually. While effective, this method is not scalable and is prone to errors, particularly when managing multiple instances.

- Updating packages with ‘sudo apt update sudo apt upgrade -y‘
- Installing Apache, MariaDB, and PHP using ‘sudo apt install‘
- Starting services with ‘sudo systemctl enable/start‘
- Installing monitoring tools such as Prometheus and Grafana by manually downloading and configuring system files.
- Setting up basic security measures like Fail2Ban and configuring log monitoring with Prometheus.

2.2.2 Automated Configuration with Ansible

To address these limitations, we turned to Ansible, a powerful automation tool designed for configuring and managing infrastructure. Ansible allows for the automation of repetitive tasks across many servers, ensuring consistency and saving valuable time.

We wrote an Ansible playbook to automate the installation and configuration of all necessary software packages. The playbook performs tasks such as:

- Updating the server packages.
- Installing Apache, MariaDB, PHP, and other dependencies.
- Starting services such as Apache and MariaDB.
- Installing monitoring tools like Prometheus and Grafana and configuring them to run as system services.
- Configuring security tools such as Fail2Ban, Rootkit Hunter, and ClamAV for enhanced server protection.

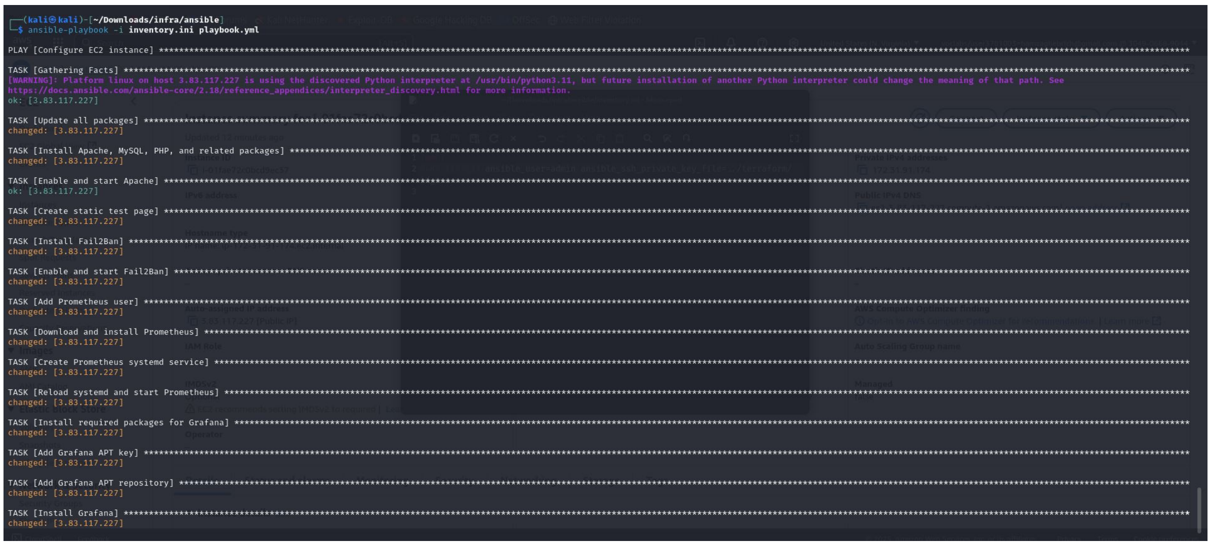
By running the playbook, all servers were configured in a uniform manner, ensuring that no manual errors occurred. Additionally, Ansible’s idempotent nature meant that running the playbook multiple times would not cause issues, providing flexibility and ensuring stability across deployments.

2.2.3 Ansible Execution and Results

Once the Ansible playbook was written, it was executed using the command:

```
ansible-playbook -i inventory.ini playbook.yml
```

The ‘inventory.ini‘ file specified the list of servers, and the ‘playbook.yml‘ file contained the configuration instructions.



```
┌──(kali㉿kali)-[~/Downloads/infra/ansible]
└─$ ansible-playbook -i inventory.ini playbook.yml

PLAY [Configure EC2 instance] ******************************************************************

TASK [Gathering Facts] *************************************************************************
[WARNING]: Platform linux on host 3.83.117.227 is using the discovered Python interpreter at /usr/bin/python3.11, but future installation of another Python interpreter could change the meaning of that path. See
https://docs.ansible.com/ansible-core/2.18/reference_appendices/interpreter_discovery.html for more information.
ok: [3.83.117.227]

TASK [Update all packages] *********************************************************************
changed: [3.83.117.227]

TASK [Install Apache, MySQL, PHP, and related packages] ****************************************
changed: [3.83.117.227]

TASK [Enable and start Apache] *****************************************************************
ok: [3.83.117.227]

TASK [Create static test page] *****************************************************************
changed: [3.83.117.227]

TASK [Install Fail2Ban] ************************************************************************
changed: [3.83.117.227]

TASK [Enable and start Fail2Ban] ***************************************************************
changed: [3.83.117.227]

TASK [Add Prometheus user] *********************************************************************
changed: [3.83.117.227]

TASK [Download and install Prometheus] *********************************************************
changed: [3.83.117.227]

TASK [Create Prometheus systemd service] *******************************************************
changed: [3.83.117.227]

TASK [Reload systemd and start Prometheus] *****************************************************
changed: [3.83.117.227]

TASK [Install required packages for Grafana] ***************************************************
changed: [3.83.117.227]

TASK [Add Grafana APT key] *********************************************************************
changed: [3.83.117.227]

TASK [Add Grafana APT repository] **************************************************************
changed: [3.83.117.227]

TASK [Install Grafana] *************************************************************************
changed: [3.83.117.227]
```

Figure 2.5: Execution of `ansible-playbook` Command

2.2.4 Ansible Architecture

The Ansible architecture consists of the following key components:

- **Control Node:** The machine from which Ansible commands are executed. It requires Ansible installed and SSH access to managed nodes.
- **Managed Nodes:** The remote systems on which Ansible operates. They do not require any agent.
- **Inventory File:** A file listing the managed nodes, which can be static or dynamic.
- **Playbooks:** YAML files that define a set of tasks to be executed on the managed nodes.

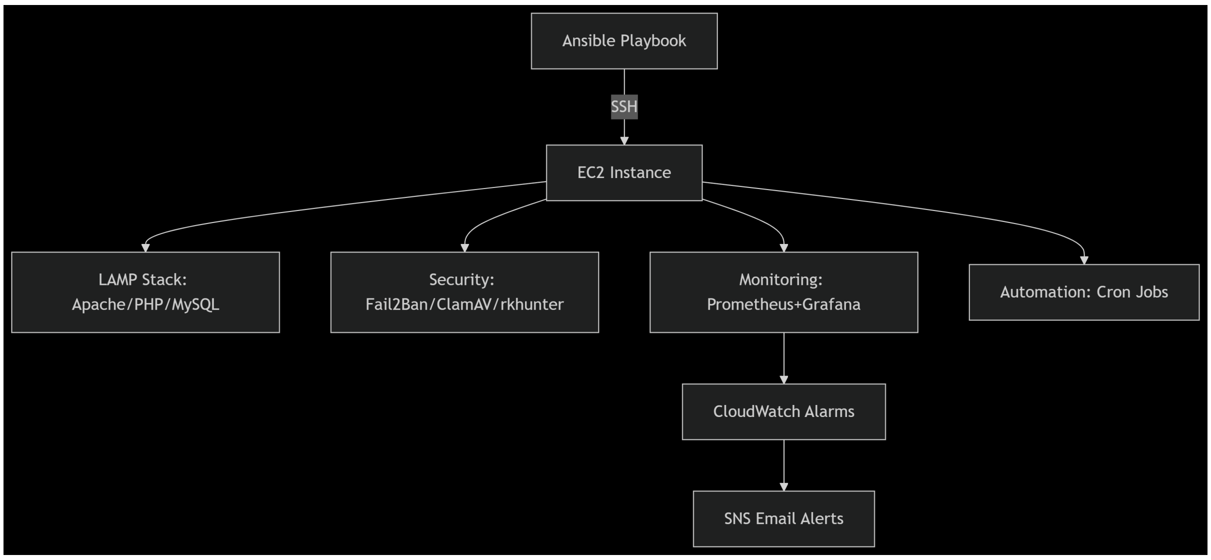

Ansible Playbook
SSH
EC2 Instance
LAMP Stack: Apache/PHP/MySQL
Security: Fail2Ban/ClamAV/rkhunter
Monitoring: Prometheus+Grafana
Automation: Cron Jobs
CloudWatch Alarms
SNS Email Alerts

Figure 2.6: Ansible Architecture Diagram

Conclusion

The combination of Terraform and Ansible enabled us to automate both the infrastructure provisioning and configuration management processes, achieving significant improvements in speed, accuracy, and scalability. Terraform served as the foundation for defining and deploying infrastructure, while Ansible streamlined the installation and configuration of critical software components. By integrating both tools, we built a reliable, consistent, and maintainable cloud environment that supports secure and scalable operations.

Chapter 3

Implementation

Introduction

This chapter presents the practical implementation of our security monitoring and response system in a cloud environment. It outlines the integration of tools and techniques for collecting, analyzing, and visualizing system logs, detecting potential threats, and responding automatically to suspicious behavior. We begin by deploying the ELK Stack (Elasticsearch, Logstash, and Kibana) for centralized log management and analysis. Next, we describe how Python scripts, in conjunction with AWS services and Boto3, are used to automate threat response actions such as blocking malicious IP addresses. Finally, we demonstrate the use of Grafana for visualizing metrics and security events, enhancing real-time system observability. Each section is supported by diagrams, workflows, and tables to clearly explain the architecture and functionality of the implemented components.

3.1 Log Collection and Analysis

To effectively monitor and analyze system activities, we utilize the ELK Stack, comprising Elasticsearch, Logstash, and Kibana. This stack facilitates the aggregation, processing, and visualization of logs from various sources.

3.1.1 ELK Stack Components

The ELK Stack consists of three main components :

- **Elasticsearch**: A distributed, RESTful search and analytics engine capable of storing and indexing large volumes of data.
- **Logstash**: A data processing pipeline that ingests data from multiple sources, transforms it, and sends it to a specified destination.
- **Kibana**: A visualization tool that works on top of Elasticsearch, providing a user interface for querying and visualizing data.

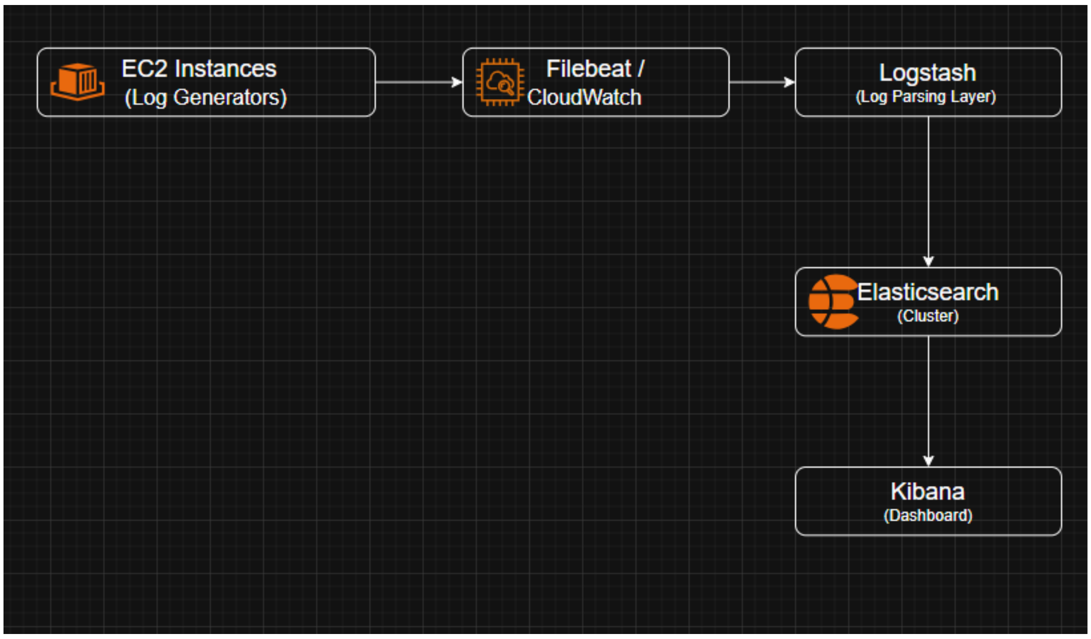


Figure 3.1: ELK Stack Architecture for Log Collection and Analysis

3.1.2 Log Ingestion Workflow

The log ingestion workflow involves four main steps :

1. **Data Collection**: Logs from various sources (e.g., application logs, system logs) are collected using Beats or other agents.

2. **Data Processing**: Logstash processes the collected logs, applying filters and transformations as needed.

3. **Data Storage**: Processed logs are stored in Elasticsearch, where they are indexed for efficient querying.

4. **Data Visualization**: Kibana accesses data from Elasticsearch, allowing users to create dashboards and visualizations.

Component	**Description**
Elasticsearch	A distributed search and analytics engine used to store, index, and search large volumes of structured and unstructured logs.
Logstash	A data pipeline tool that collects, parses, and transforms logs before forwarding them to Elasticsearch. It supports complex filtering, enrichment, and routing logic.
Kibana	A web-based interface for querying data in Elasticsearch, creating dashboards, visualizing trends, and setting alerts.
Beats (Filebeat)	Lightweight agents installed on servers to forward logs to Logstash or Elasticsearch directly.

Table 3.1: Roles and Responsibilities of ELK Stack Components

3.2 Automated Threat Response with Python and Boto3

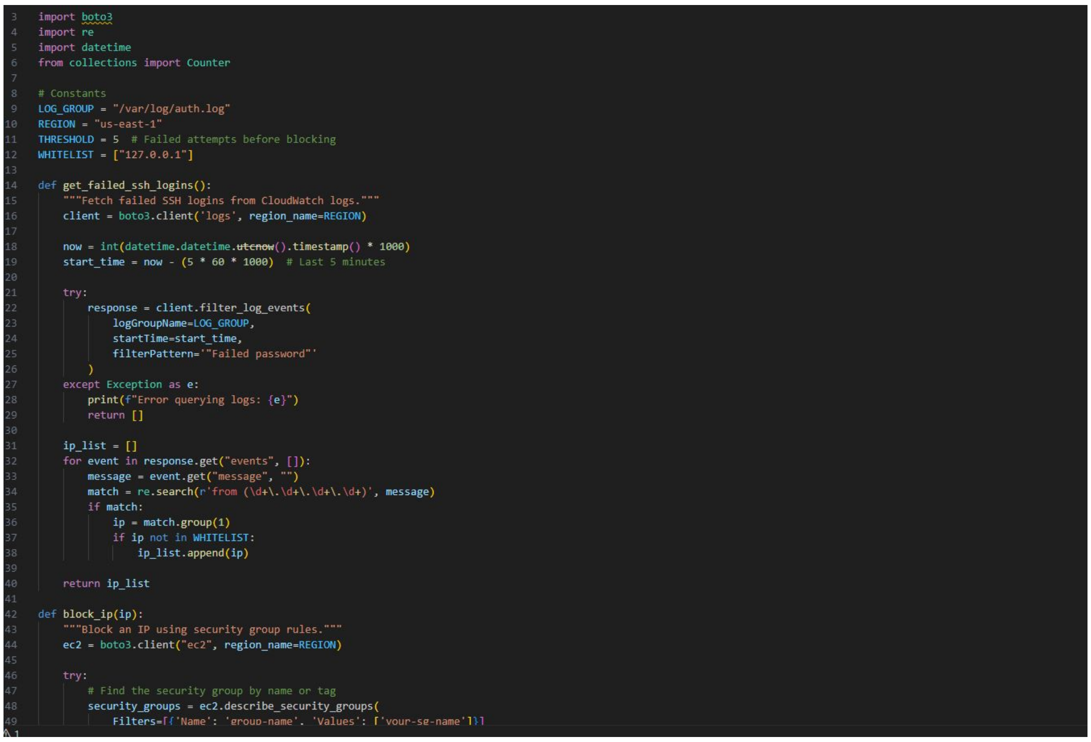
To enhance security, we implement automated threat detection and response mechanisms using Python scripts that interact with AWS services via Boto3.

3.2.1 Threat Detection

Python scripts analyze logs stored in Elasticsearch to identify suspicious activities, such as multiple failed SSH login attempts from the same IP address.

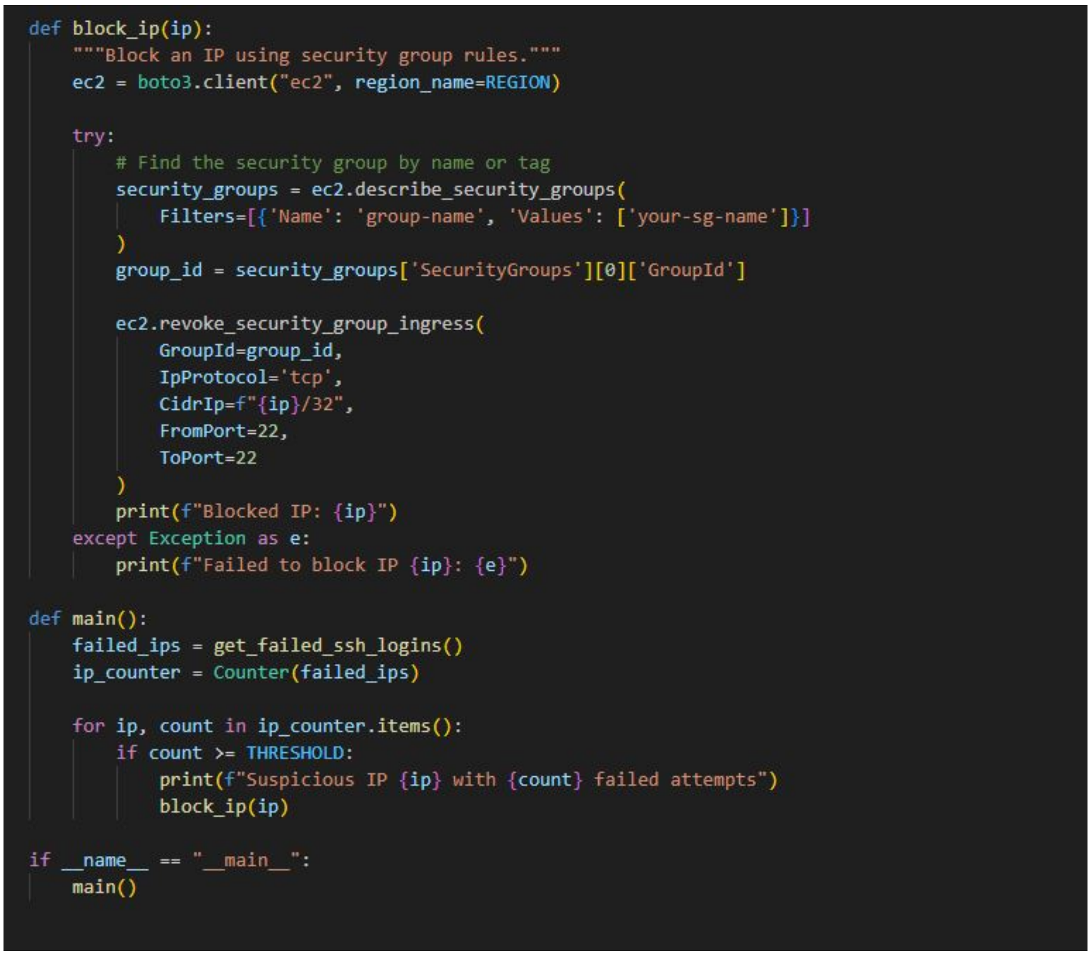
3.2.2 Automated IP Blocking

Upon detecting a potential threat, the script automatically updates AWS Security Groups to block the offending IP address.



```python
import boto3
import re
import datetime
from collections import Counter

# Constants
LOG_GROUP = "/var/log/auth.log"
REGION = "us-east-1"
THRESHOLD = 5  # Failed attempts before blocking
WHITELIST = ["127.0.0.1"]

def get_failed_ssh_logins():
    """Fetch failed SSH logins from CloudWatch logs."""
    client = boto3.client('logs', region_name=REGION)

    now = int(datetime.datetime.~~utcnow~~().timestamp() * 1000)
    start_time = now - (5 * 60 * 1000)  # Last 5 minutes

    try:
        response = client.filter_log_events(
            logGroupName=LOG_GROUP,
            startTime=start_time,
            filterPattern='"Failed password"'
        )
    except Exception as e:
        print(f"Error querying logs: {e}")
        return []

    ip_list = []
    for event in response.get("events", []):
        message = event.get("message", "")
        match = re.search(r'from (\d+\.\d+\.\d+\.\d+)', message)
        if match:
            ip = match.group(1)
            if ip not in WHITELIST:
                ip_list.append(ip)

    return ip_list

def block_ip(ip):
    """Block an IP using security group rules."""
    ec2 = boto3.client("ec2", region_name=REGION)

    try:
        # Find the security group by name or tag
        security_groups = ec2.describe_security_groups(
            Filters=[{'Name': 'group-name', 'Values': ['your-sg-name']}]
```

Figure 3.2: Automated IP Blocking Workflow Using Python and Boto3 (1)



```python
def block_ip(ip):
    """Block an IP using security group rules."""
    ec2 = boto3.client("ec2", region_name=REGION)

    try:
        # Find the security group by name or tag
        security_groups = ec2.describe_security_groups(
            Filters=[{'Name': 'group-name', 'Values': ['your-sg-name']}]
        )
        group_id = security_groups['SecurityGroups'][0]['GroupId']

        ec2.revoke_security_group_ingress(
            GroupId=group_id,
            IpProtocol='tcp',
            CidrIp=f"{ip}/32",
            FromPort=22,
            ToPort=22
        )
        print(f"Blocked IP: {ip}")
    except Exception as e:
        print(f"Failed to block IP {ip}: {e}")

def main():
    failed_ips = get_failed_ssh_logins()
    ip_counter = Counter(failed_ips)

    for ip, count in ip_counter.items():
        if count >= THRESHOLD:
            print(f"Suspicious IP {ip} with {count} failed attempts")
            block_ip(ip)

if __name__ == "__main__":
    main()
```

Figure 3.3: Automated IP Blocking Workflow Using Python and Boto3 (2)

Step	**Description**
Threat Detection	Python scripts query Elasticsearch to find suspicious patterns such as brute-force attempts or repeated failed logins.
Log Analysis	Specific fields such as IP address, login result, and timestamps are extracted and evaluated against defined rules.
Trigger Condition	If thresholds (e.g., more than 5 failed logins within 2 minutes) are met, the script flags the source as malicious.
Blocking Action	Using Boto3, the script updates the AWS Security Group to deny incoming traffic from the identified IP address.
Alerting (Optional)	Notification (via SNS or email) is sent to the admin team regarding the blocked IP and its activity.

Table 3.2: Steps in the Automated Threat Response Mechanism

3.3 Dashboard Visualization with Grafana

Grafana is employed to create dynamic dashboards that visualize system metrics and security events, providing real-time insights into system performance and potential threats.

3.3.1 Dashboard Components

The dashboard consists of several key components to provide an overview of the system's status:

- **System Metrics**: CPU usage, memory consumption and network activity.
- **Security Events**: Number of blocked IPs, failed login attempts, and other security-related metrics.
- **Alerts**: Configurable thresholds that trigger notifications when certain conditions are met.

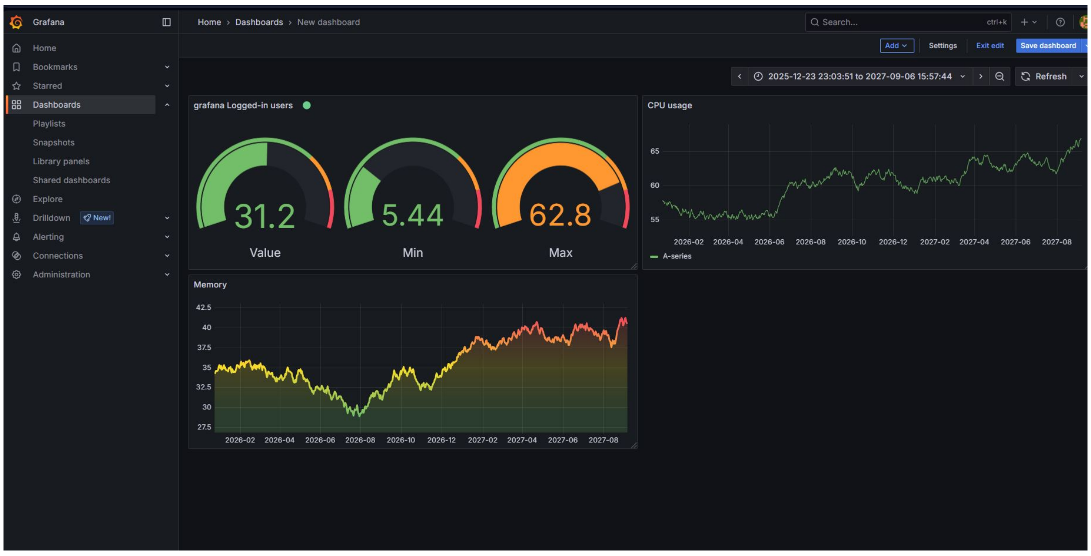


Figure 3.4: Grafana Dashboard Displaying System Metrics

3.3.2 Alerting Mechanism

Grafana's alerting feature monitors specified metrics and sends notifications via various channels (e.g., email, Slack) when thresholds are breached, enabling prompt response to potential issues.

Conclusion

Integrating the ELK Stack for log collection, Python scripts with Boto3 for automated threat response, and Grafana for visualization creates a robust system for monitoring and securing infrastructure. This comprehensive approach ensures real-time detection, swift mitigation of threats, and insightful analysis of system performance.

Chapter 4

Validation and Evaluation

Introduction

This chapter focuses on validating the implemented security mechanisms and evaluating their effectiveness under real-world scenarios. Through a series of simulated cyberattacks, we test the robustness, responsiveness, and reliability of the monitoring and automated response system. Tools such as Hydra and stress-ng are used to emulate brute-force and resource exhaustion attacks, respectively. These controlled experiments provide insight into the system's behavior during abnormal conditions and verify the ability of our detection and mitigation strategies to handle threats efficiently. Furthermore, we analyze performance metrics, identify current limitations, and propose improvements for future enhancements, including the potential integration of machine learning for adaptive threat detection.

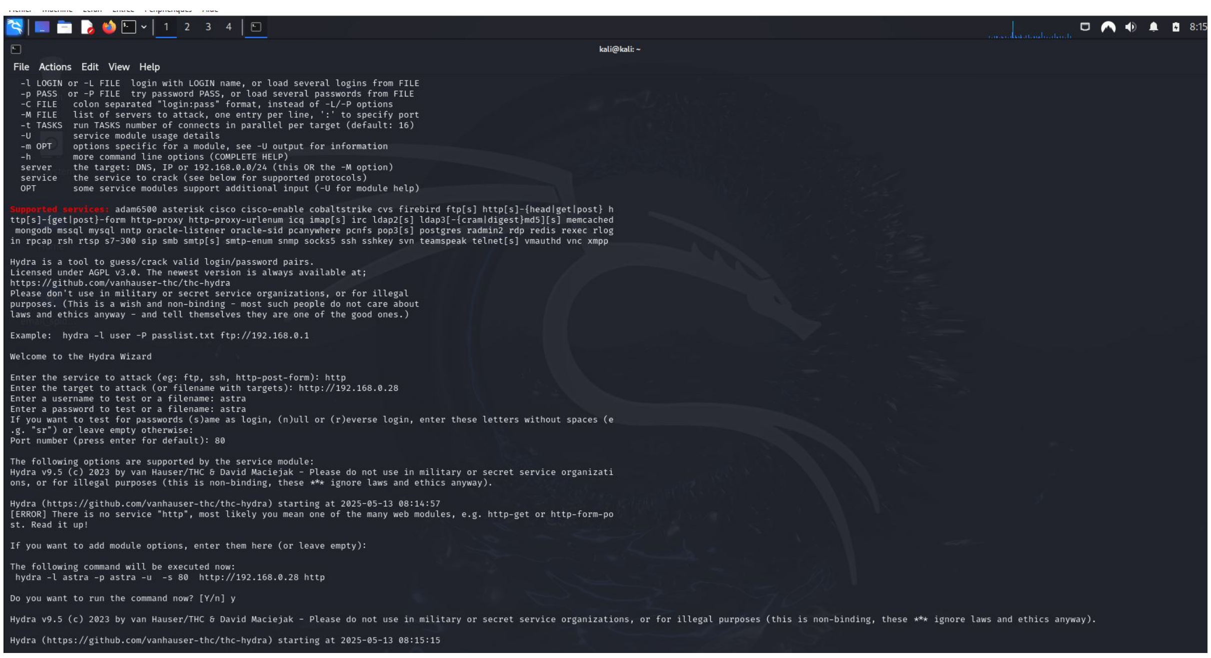
4.1 Simulated Attacks

To assess the robustness of our security infrastructure, we conducted controlled simulated attacks using industry-standard tools:

4.1.1 Brute-Force Attack Simulation with Hydra

Hydra is an open-source tool designed for performing brute-force attacks on various protocols and services to test authentication mechanisms.

- **Objective**: Evaluate the system's ability to detect and respond to unauthorized access attempts.
- **Method**: Simulate multiple failed SSH login attempts using Hydra to mimic a brute-force attack.
- **Expected Outcome**: The intrusion detection system (IDS) should identify the repeated failed attempts and trigger appropriate alerts or countermeasures.



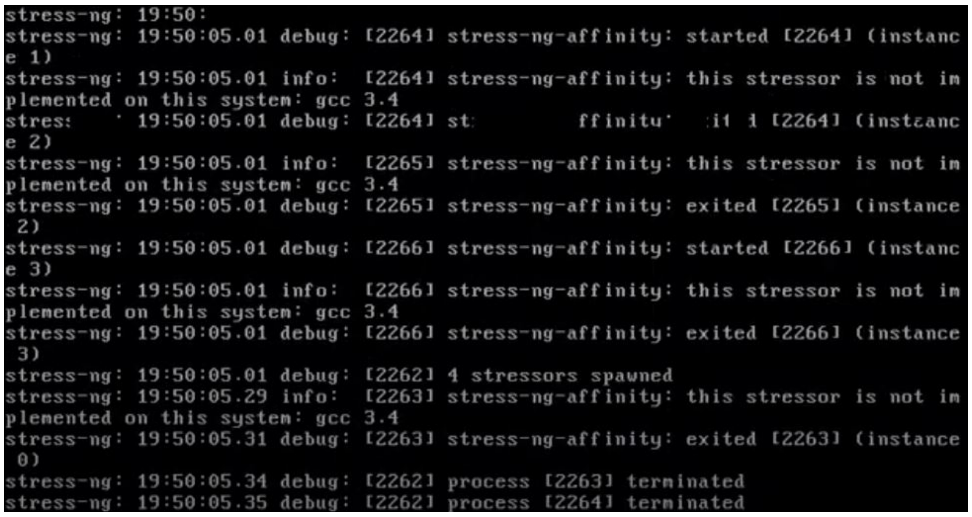
```
  -l LOGIN or -L FILE  login with LOGIN name, or load several logins from FILE
  -p PASS  or -P FILE  try password PASS, or load several passwords from FILE
  -C FILE   colon separated "login:pass" format, instead of -L/-P options
  -M FILE   list of servers to attack, one entry per line, ':' to specify port
  -t TASKS  run TASKS number of connects in parallel per target (default: 16)
  -U        service module usage details
  -m OPT    options specific for a module, see -U output for information
  -h        more command line options (COMPLETE HELP)
  server    the target: DNS, IP or 192.168.0.0/24 (this OR the -M option)
  service   the service to crack (see below for supported protocols)
  OPT       some service modules support additional input (-U for module help)

Supported services: adam6500 asterisk cisco cisco-enable cobaltstrike cvs firebird ftp[s] http[s]-{head|get|post} h
ttp[s]-{get|post}-form http-proxy http-proxy-urlenum icq imap[s] irc ldap2[s] ldap3[-{cram|digest}md5][s] memcached
 mongodb mssql mysql nntp oracle-listener oracle-sid pcanywhere pcnfs pop3[s] postgres radmin2 rdp redis rexec rlog
in rpcap rsh rtsp s7-300 sip smb smtp[s] smtp-enum snmp socks5 ssh sshkey svn teamspeak telnet[s] vmauthd vnc xmpp

Hydra is a tool to guess/crack valid login/password pairs.
Licensed under AGPL v3.0. The newest version is always available at;
https://github.com/vanhauser-thc/thc-hydra
Please don't use in military or secret service organizations, or for illegal
purposes. (This is a wish and non-binding - most such people do not care about
laws and ethics anyway - and tell themselves they are one of the good ones.)

Example:  hydra -l user -P passlist.txt ftp://192.168.0.1

Welcome to the Hydra Wizard

Enter the service to attack (eg: ftp, ssh, http-post-form): http
Enter the target to attack (or filename with targets): http://192.168.0.28
Enter a username to test or a filename: astra
Enter a password to test or a filename: astra
If you want to test for passwords (s)ame as login, (n)ull or (r)everse login, enter these letters without spaces (e
.g. "sr") or leave empty otherwise:
Port number (press enter for default): 80

The following options are supported by the service module:
Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizati
ons, or for illegal purposes (this is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-05-13 08:14:57
[ERROR] There is no service "http", most likely you mean one of the many web modules, e.g. http-get or http-form-po
st. Read it up!

If you want to add module options, enter them here (or leave empty):

The following command will be executed now:
 hydra -l astra -p astra -u  -s 80  http://192.168.0.28 http

Do you want to run the command now? [Y/n] y

Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-05-13 08:15:15
```

Figure 4.1: Hydra command-line interface used for brute-force attack simulation

4.1.2 Resource Exhaustion Test with stress-ng

stress-ng is a tool designed to stress test a computer system in various selectable ways. It was designed to exercise various physical subsystems of a computer as well as the various operating system kernel interfaces.

- **Objective**: Assess system stability and monitoring effectiveness under high resource utilization.
- **Method**: Utilize stress-ng to apply CPU, memory, and I/O stress, observing system behavior and monitoring tool responses.
- **Expected Outcome**: Monitoring tools should accurately report resource usage spikes, and the system should maintain operational stability without critical failures.



```
stress-ng: 19:50:
stress-ng: 19:50:05.01 debug: [2264] stress-ng-affinity: started [2264] (instanc
e 1)
stress-ng: 19:50:05.01 info:  [2264] stress-ng-affinity: this stressor is not im
plemented on this system: gcc 3.4
stres:    ' 19:50:05.01 debug: [2264] st        ffinitu'   :i1 1 [2264] (instcanc
e 2)
stress-ng: 19:50:05.01 info:  [2265] stress-ng-affinity: this stressor is not im
plemented on this system: gcc 3.4
stress-ng: 19:50:05.01 debug: [2265] stress-ng-affinity: exited [2265] (instance
 2)
stress-ng: 19:50:05.01 debug: [2266] stress-ng-affinity: started [2266] (instanc
e 3)
stress-ng: 19:50:05.01 info:  [2266] stress-ng-affinity: this stressor is not im
plemented on this system: gcc 3.4
stress-ng: 19:50:05.01 debug: [2266] stress-ng-affinity: exited [2266] (instance
 3)
stress-ng: 19:50:05.01 debug: [2262] 4 stressors spawned
stress-ng: 19:50:05.29 info:  [2263] stress-ng-affinity: this stressor is not im
plemented on this system: gcc 3.4
stress-ng: 19:50:05.31 debug: [2263] stress-ng-affinity: exited [2263] (instance
 0)
stress-ng: 19:50:05.34 debug: [2262] process [2263] terminated
stress-ng: 19:50:05.35 debug: [2262] process [2264] terminated
```

Figure 4.2: stress-ng command-line interface used for resource exhaustion testing

4.2 Effectiveness Assessment

Evaluating the effectiveness of our security measures involves analyzing specific metrics:

4.3 Limitations and Improvements

While the current security setup demonstrates effectiveness, certain limitations were identified:

4.3.1 Edge Cases and Internal Traffic

Internal network traffic with unusual patterns occasionally triggered false alerts, highlighting a limitation in the current detection system. To address this, more granular traffic analysis and context-aware detection rules should be implemented to better distinguish between legitimate internal anomalies and actual threats.

4.3.2 Logging Limitations

Some transient events were not captured due to logging thresholds and storage constraints, which limits the completeness of event tracking and incident analysis. Enhancing the logging infrastructure—potentially by integrating scalable storage solutions can help ensure more comprehensive data capture and improve the system's ability to retain and analyze critical information.

4.3.3 Potential Machine Learning Integration

Incorporating machine learning can enhance anomaly detection by leveraging historical data and adapting to emerging threat patterns. To ensure effectiveness, it is important to evaluate suitable machine learning models and train them on diverse datasets to minimize biases and improve overall detection accuracy.

Conclusion

The simulated attack scenarios provided valuable insights into the system's detection and response capabilities. While the infrastructure effectively identified and mitigated threats promptly, addressing the highlighted limitations and exploring advanced solutions like machine learning integration can further bolster security posture.

General Conclusion

This project addressed the detection and automated response to server abuse within a cloud-based infrastructure. Leveraging Infrastructure as Code (IaC) principles, the solution was deployed using Terraform for environment provisioning and Ansible for configuration management. The system integrated open-source monitoring tools such as Prometheus and Grafana, log aggregation via the ELK stack or AWS CloudWatch, and automated security responses with Python scripts utilizing the Boto3 SDK.

Simulated attack scenarios using tools like Hydra and stress-ng validated the system's ability to detect brute-force and resource-exhaustion threats. The response mechanism dynamically blocked malicious IP addresses and notified administrators in near real-time. The infrastructure was evaluated on metrics such as detection time, response time, and false positive rate, demonstrating its effectiveness and operational reliability.

Throughout the project, a variety of both technical and professional skills were acquired and strengthened:

- **Infrastructure as Code (IaC):** Proficient use of Terraform for provisioning AWS resources such as EC2, SNS, and CloudWatch.
- **Automation and Configuration Management:** Hands-on experience with Ansible for server configuration and software deployment.
- **Monitoring and Logging:** Integration of Prometheus, Grafana, and ELK/CloudWatch for system visibility and performance analytics.
- **Security Automation:** Development of Python scripts for real-time log analysis, automated threat detection, and remediation via AWS APIs.
- **Cloud Platforms:** Deployment and testing in Amazon Web Services (AWS) environments, including IAM, VPC, Security Groups, and EC2.
- **Analytical Thinking:** Ability to analyze attack patterns and design appropriate automated defenses.
- **Problem Solving:** Tackling real-world security challenges using open-source tools and custom scripts.
- **Documentation and Reporting:** Clear and structured reporting of technical implementations, validation results, and system architecture.
- **Project Management:** Time management and iterative development using agile principles in a cybersecurity context.

Although the implemented solution is effective for detecting and mitigating common attack vectors, several areas can be explored to enhance the system's intelligence, adaptability, and scalability:

- **Machine Learning Integration:** Implementing anomaly detection algorithms to identify zero-day or behavioral threats beyond predefined rules. Supervised or unsupervised models could be trained on log data to distinguish between legitimate and malicious activity with higher accuracy.
- **Advanced AWS WAF Integration:** Using AWS Web Application Firewall (WAF) IP sets to automatically update deny lists in response to detected attacks, enabling centralized and scalable traffic filtering across multiple services.
- **Real-Time Visual Alerting:** Enhancing Grafana dashboards with advanced visual indicators and integrating alerting systems (e.g., Slack, email, or SMS via AWS SNS) for instant incident visibility and response coordination.
- **Scalability and Multi-Region Deployment:** Expanding the system to support multiple cloud regions or hybrid environments, ensuring resilience and scalability in large-scale deployments.
- **Compliance and Auditing:** Incorporating audit trails, access logging, and compliance checks to align with standards such as ISO 27001 or GDPR.

This project demonstrates the feasibility and effectiveness of automating cloud security tasks using open-source tools and scripting. As cloud environments grow in complexity, future solutions must evolve to combine automation with intelligence to deliver proactive, adaptive, and scalable security architectures.

Appendices

Appendix A: Full Terraform Configuration Script

The following listing presents the complete Terraform code used for infrastructure provisioning, including key pair generation, EC2 provisioning, security group configuration, and CloudWatch/SNS integration. [language=hcl, caption=Terraform Full Infrastructure Code]full-terraform-code.tf

Appendix B: Full Ansible Playbook

This section includes the Ansible playbook that automates the installation and configuration of services such as Apache, MariaDB, PHP, Fail2Ban, Prometheus, Grafana, and additional security tools. [language=yaml, caption=Ansible Full Automation Script]playbook.yml

Appendix C: Python Abuse Detection and Response Script

This Python script collects logs, analyzes SSH authentication attempts, identifies brute-force patterns, and automates the blocking of IP addresses using the AWS Boto3 SDK.

Appendix D: Monitoring Dashboard Screenshots

The following image presents the full Grafana dashboard configured to monitor CPU usage, memory utilization, brute-force detection alerts, and blocked IP metrics in real time.

Bibliography

[1] Terraform by HashiCorp. *Terraform Documentation.* Available at: `https://www.terraform.io/docs` (Accessed: 15/02/2025)

[2] Ansible by Red Hat. *Ansible Documentation.* Available at: `https://docs.ansible.com/` (Accessed: 17/02/2025)

[3] Amazon Web Services. *AWS Documentation.* Available at: `https://docs.aws.amazon.com/` (Accessed: 20/02/2025)

[4] Grafana Labs. *Grafana Documentation.* Available at: `https://grafana.com/docs/` (Accessed: 22/02/2025)

[5] Prometheus Authors. *Prometheus Monitoring System.* Available at: `https://prometheus.io/docs/` (Accessed: 25/02/2025)

[6] Boto3 Developers. *Boto3 – AWS SDK for Python.* Available at: `https://boto3.amazonaws.com/v1/documentation/api/latest/index.html` (Accessed: 28/02/2025)

[7] AWS CloudWatch. *Monitoring Amazon EC2 Instances.* Available at: `https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html` (Accessed: 02/03/2025)

[8] THC Hydra. *Hydra - Network Logon Cracker.* Available at: `https://github.com/vanhauser-thc/thc-hydra` (Accessed: 05/03/2025)

[9] Stress-ng. *Tool to Load and Stress a Computer System.* Available at: `https://manpages.ubuntu.com/manpages/focal/man1/stress-ng.1.html` (Accessed: 08/03/2025)

[10] Fail2Ban Project. *Fail2Ban – Intrusion Prevention Software.* Available at: `https://www.fail2ban.org/` (Accessed: 10/03/2025)

[11] Elastic.co. *The ELK Stack: Elasticsearch, Logstash, Kibana.* Available at: `https://www.elastic.co/what-is/elk-stack` (Accessed: 12/03/2025)

[12] Rootkit Hunter. *rkhunter - Rootkit Scanner.* Available at: `https://sourceforge.net/projects/rkhunter/` (Accessed: 15/03/2025)

[13] ClamAV. *ClamAV - Open Source Antivirus Engine.* Available at: `https://www.clamav.net/` (Accessed: 18/03/2025)

[14] Wang, C. (2021). *Cybersecurity and Cyberwar: What Everyone Needs to Know.* Oxford University Press (Accessed: 20/03/2025)