National College of Ireland

Configuration Manual

MSc Research Project
MSc. In Cybersecurity

Arbaz Adib Dalwai
Student ID: x23161795

School of Computing
National College of Ireland

Supervisor: Dr. Rohit Verma

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Arbaz Adib Dalwai
Student ID:	x23161795
Programme:	MSc. In Cybersecurity **Year:** 2024-25
Module:	MSc Practicum Part-2
Lecturer:	Dr. Rohit Verma
Submission Due Date:	29.01.2025
Project Title:	BOLSTERING CLOUD SECURITY WITH REAL-TIME SIEM USING HYBRID-RULE BASED AND ML INSIGHTS.
Word Count:	8436 **Page Count:** 49

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Arbaz Adib Dalwai

Date: 28.01.2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Arbaz Adib Dalwai
x23161795

1 Setting up the Instances with necessary permissions on Amazon Web Services (AWS)

Total 6 instances were created in this project and each instance had a separate role to serve as per the requirements. All these instances were placed in the same virtual private cloud (VPC) to avoid any connectivity issues. The process for creating the instances can be explained in the following steps

- Go to the elastic cloud compute (EC2) service on AWS and launch the instances.
- Select the Application and operating system (OS) images (as per the requirements of the different components).
- Select the instance type (as per the computational strengths needed to perform the functionalities).
- Allocate the storage space.
- Set up a unique key pair (.pem file) for logging in to instance.
- Allow the necessary network settings and create unique security groups (for maintaining the network ports & connections)-

Elastic IP (internet protocol) allocation steps-

- Go to the Elastic IP section on the EC2 service in AWS. Shown in figure 1 below-

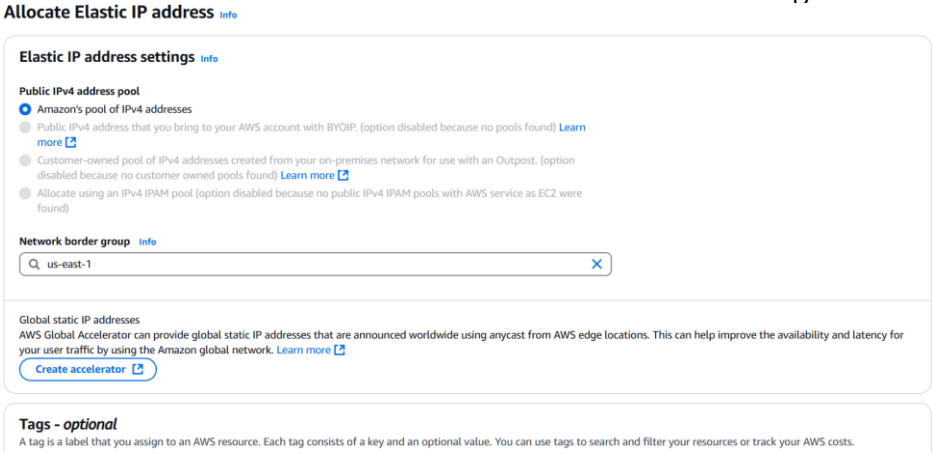


Figure 1

- Through the "Allocate IP address" button on top right getting the IPs from Amazon's pool of IPv4 addresses. A total of 5 elastic IPs could be allocated to my AWS account (as per the permissions granted through college's AWS team).
- After getting the elastic IP associating the IP address to the instance ID as shown below in figure 2.

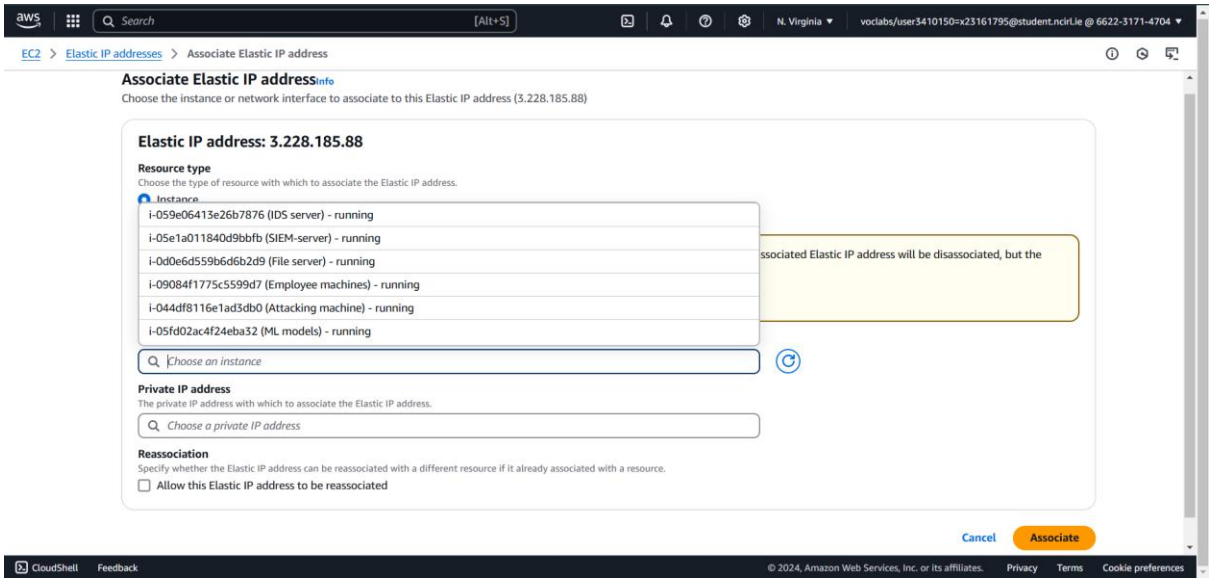


Figure 2

- Allocated IP addresses as depicted in figure 3 below-

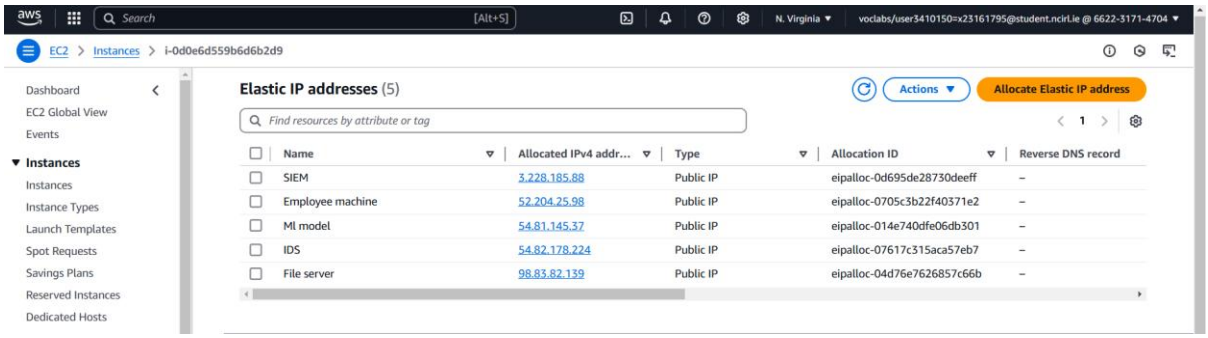


Figure 3

All the instances created along with their configurations including the security groups are listed below-

1. File server- Below is the figure 4 showing the same.

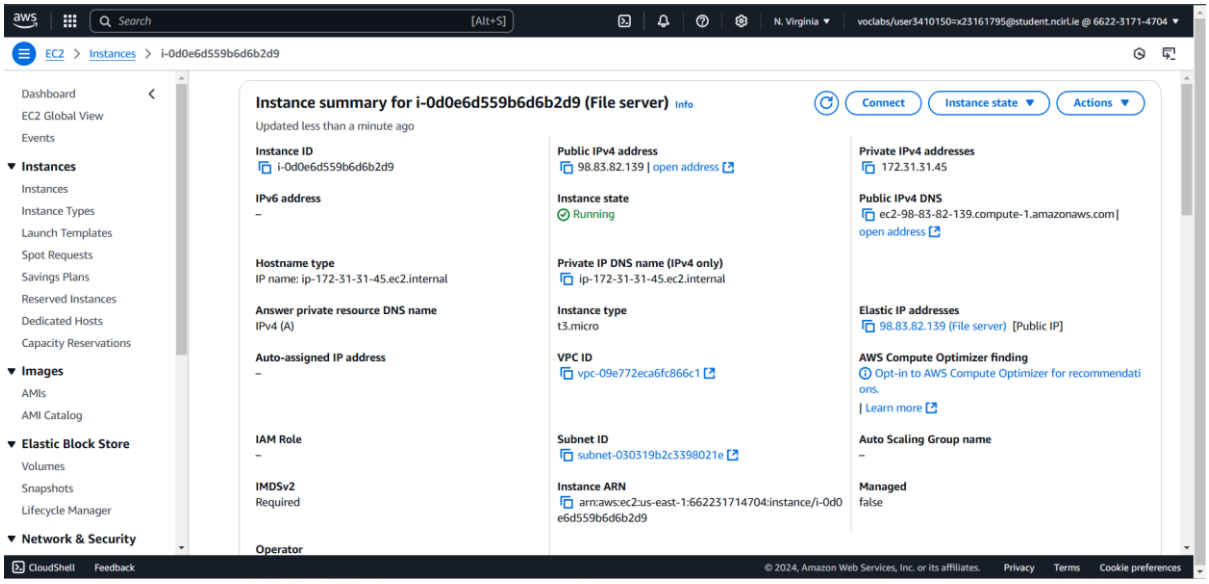


Figure 4

Instance Type: t3.micro
Application OS: Amazon Linux; VERSION="2023"; ID="amzn".
Storage: 8 GiB
Key pair: fileserver.pem
Elastic IP allocated: Public IP- 98.83.82.139, Private Ip- 172.31.31.45
Open ports & Security group configuration:

- Port 22 is open for secure socket shell (ssh) in order to get an administrative access to the instance to configure and manage it.
- Port 443 is enabling secure web traffic to the file server using the HTTPs
- Port 80 allows access to the web application hosted on the file server
- Custom ICMP- IPv4 rule in the outbound rules is facilitating the connection with the intrusion detection system (IDS) server so that all the traffic generated at file server is monitored and analysed by the IDS.

The below figures 5 & 6 depict the inbound and outbound rules resp.

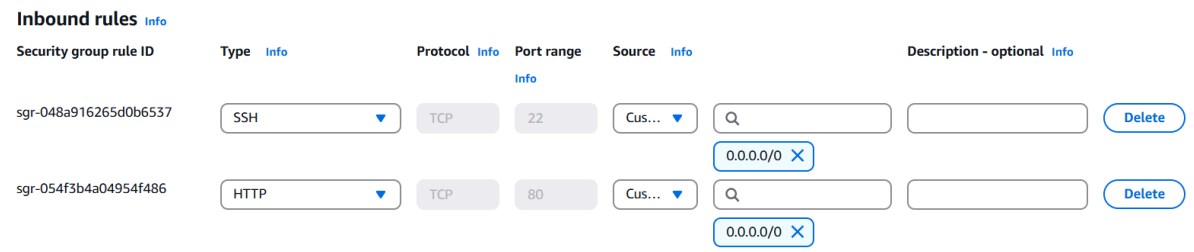


Figure 5

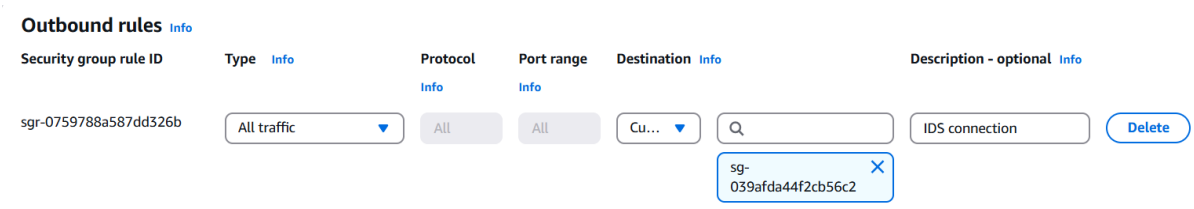


Figure 6

Testing the access connection-

The SSH connection with the instance (file server) is established using the command- **ssh -i "F:\fileserver.pem" ec2-user@98.83.82.139** and is visible through the below figure 7

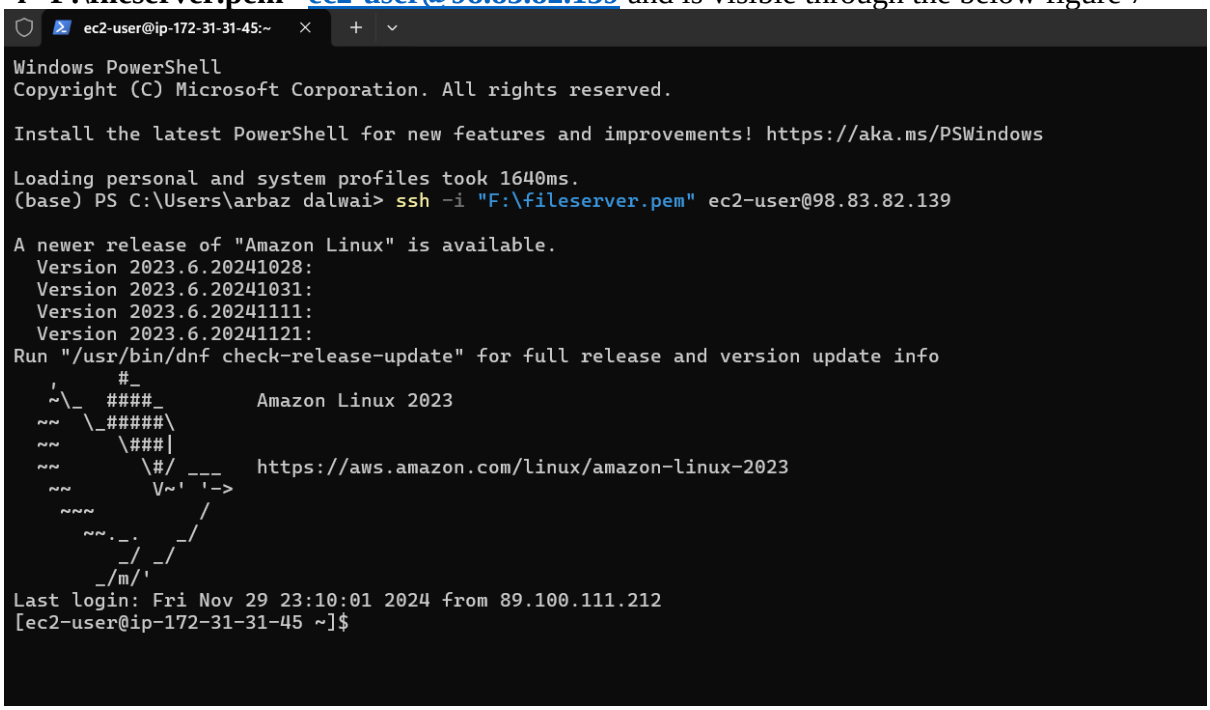


Figure 7

2. Employee machine- Below is the figure 8 depicting the same

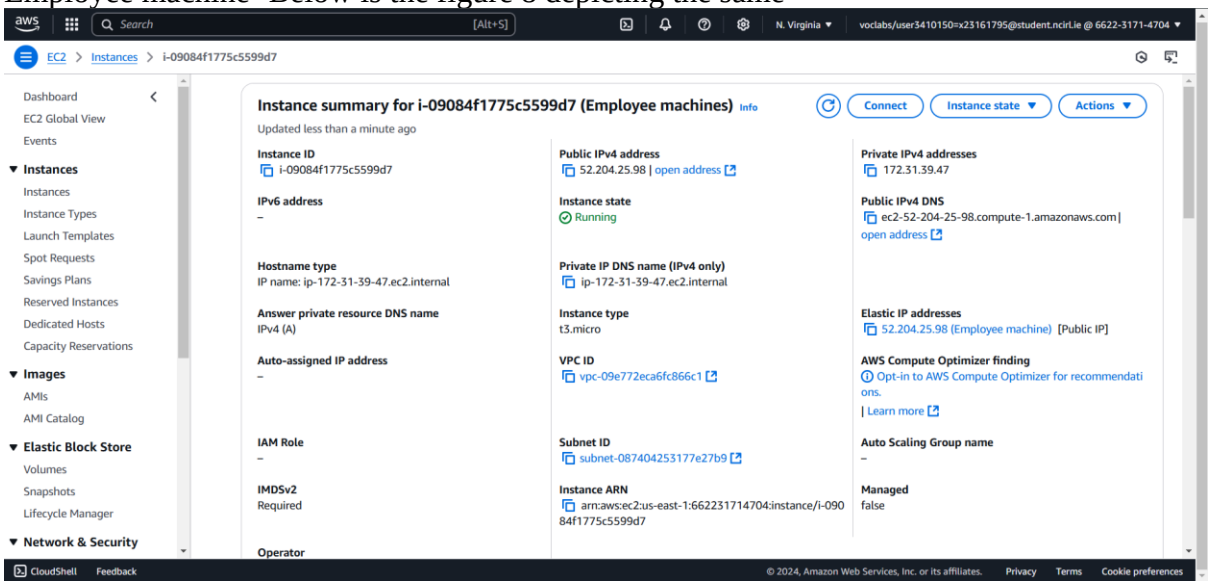


Figure 8

Instance Type: t3.micro
Application OS: Amazon Linux; VERSION="2023"; ID="amzn"
Storage: 8 GiB
Key pair: employee machine.pem
Elastic IP allocated: Public IP- 52.204.25.98, Private Ip- 172.31.39.47
Open ports & Security group configuration:

- Port 22 is open for ssh in order to get an administrative access to the instance to configure and manage it.

- Port 80 allows the employee machine to access web applications which also includes the phishing page hosted by the attacking machine.
- Custom ICMP- IPv4 rule in the outbound rules is necessary for the connection with the IDS server so that all the traffic generated at employee machine is monitored and analysed by the IDS.

The below figures 9 & 10 depict the inbound and outbound rules resp.

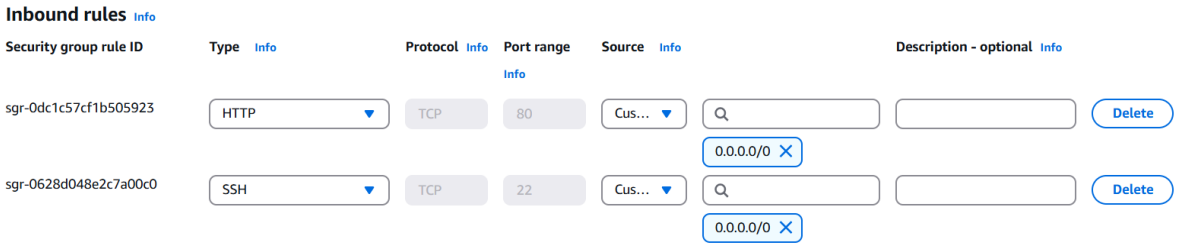


Figure 9

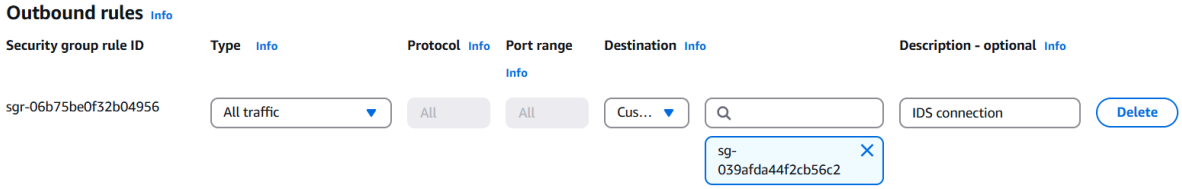


Figure 10

Testing the access connection-

The SSH connection with the instance (employee machine) is established using the command-
ssh -i "F:\employee machine.pem" **ec2-user@52.204.25.98** and is visible through the below figure 11

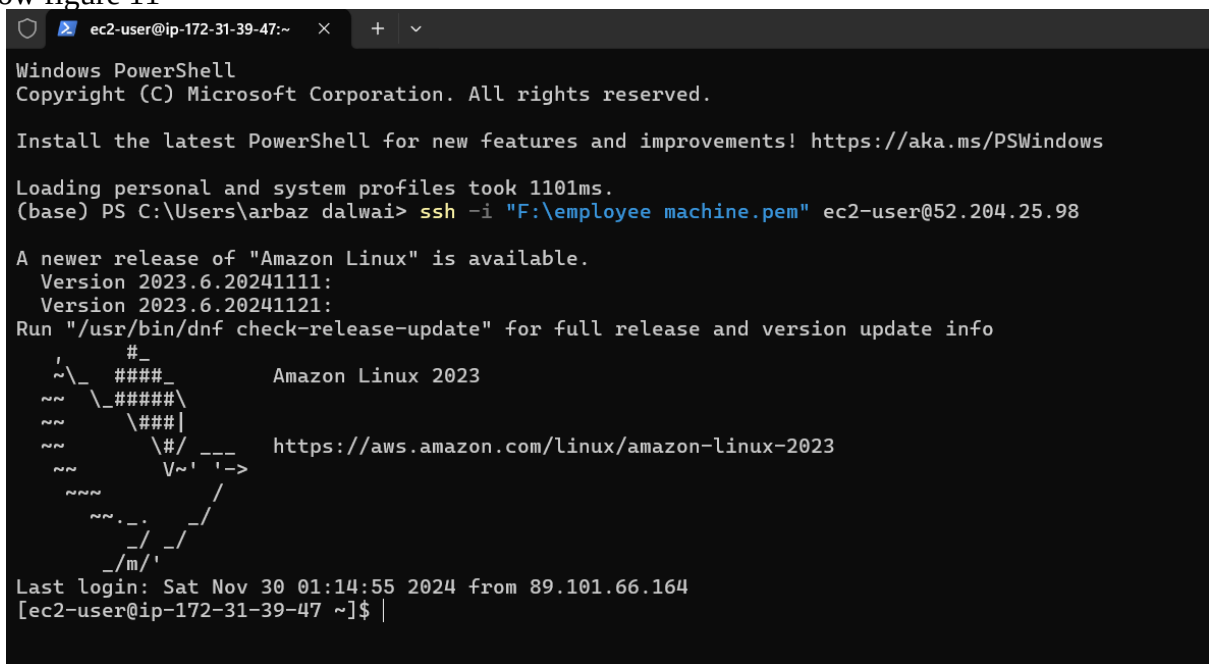


Figure 11

3. Attacking machine- Below is the figure 12 showing the same

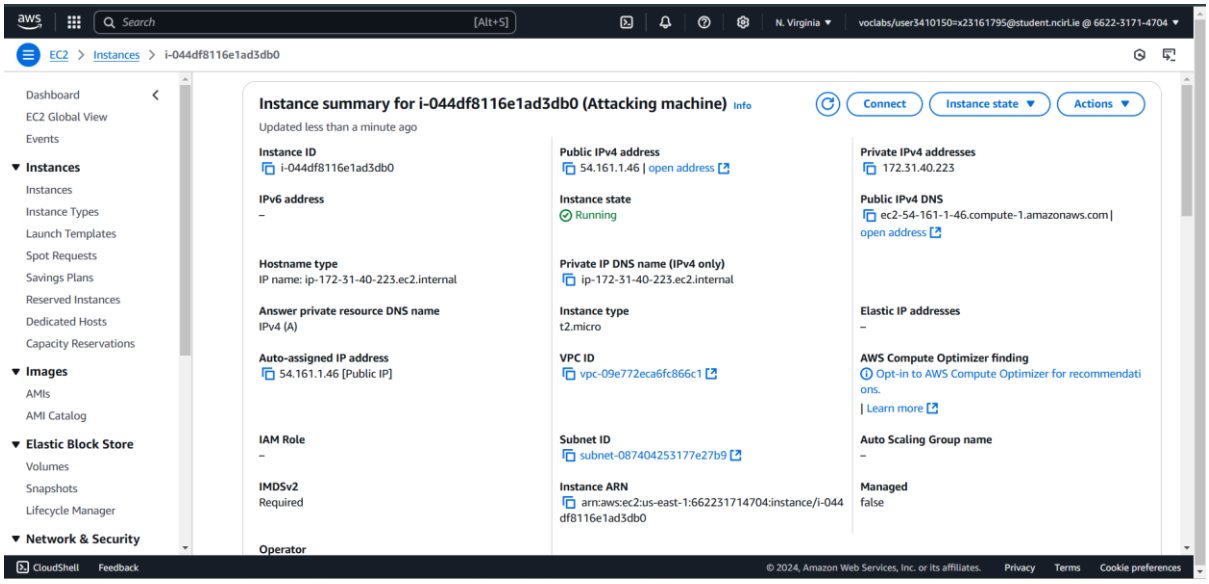


Figure 12

Instance Type: t2.micro
Application OS: Amazon Linux; VERSION="2023"; ID="amzn"
Storage: 8 GiB
Key pair: attacking machine.pem
This instance does not have an elastic IP allocated as it is the attacking machine and it should depict a realistic approach. Hence dynamic public and private IPs are allocated to it everytime the instance is restarted to increase the complexity of the attack simulation replicating an attackers mindset to avoid detection.
Open ports & Security group configuration:

- Port 22 is open for ssh in order to get an administrative access to the instance to configure and manage it.
- Port 80 allows connection to the phishing page which is to be hosted on the instance. Through this the employee machine can access the phishing simulation.
- The outbound rule for all traffic allows the instance to send the traffic including both phishing page response and distributed denial of service (DDoS) attack. This would help in inititaing the simulated attack.

The below figures 13 & 14 depict the inbound and outbound rules resp.

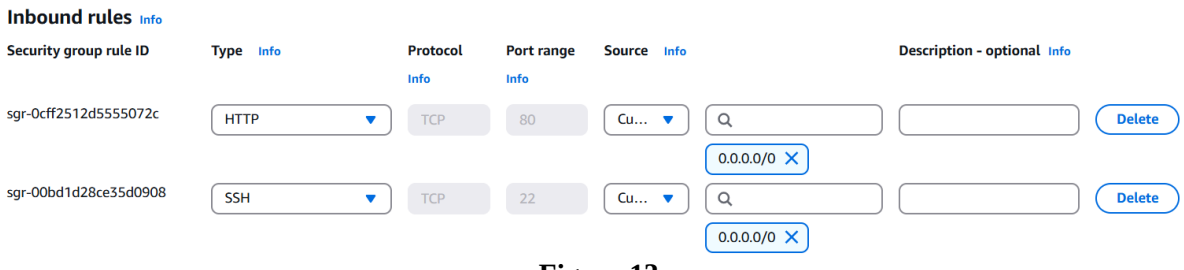


Figure 13

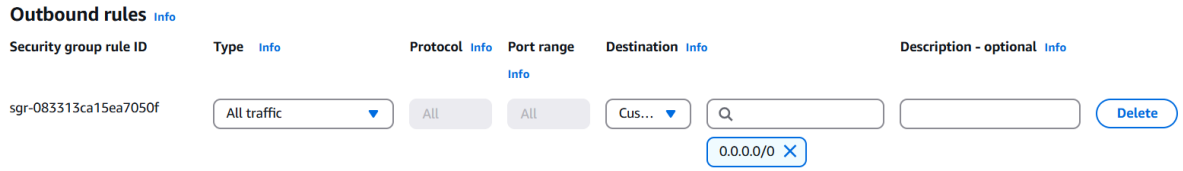


Figure 14

Testing the access connection-

The SSH connection with the instance (attacking machine) is established using the command- **ssh -i "F:\attacking machine.pem" ec2-user@54.161.1.46** and is visible through the given figure 15.

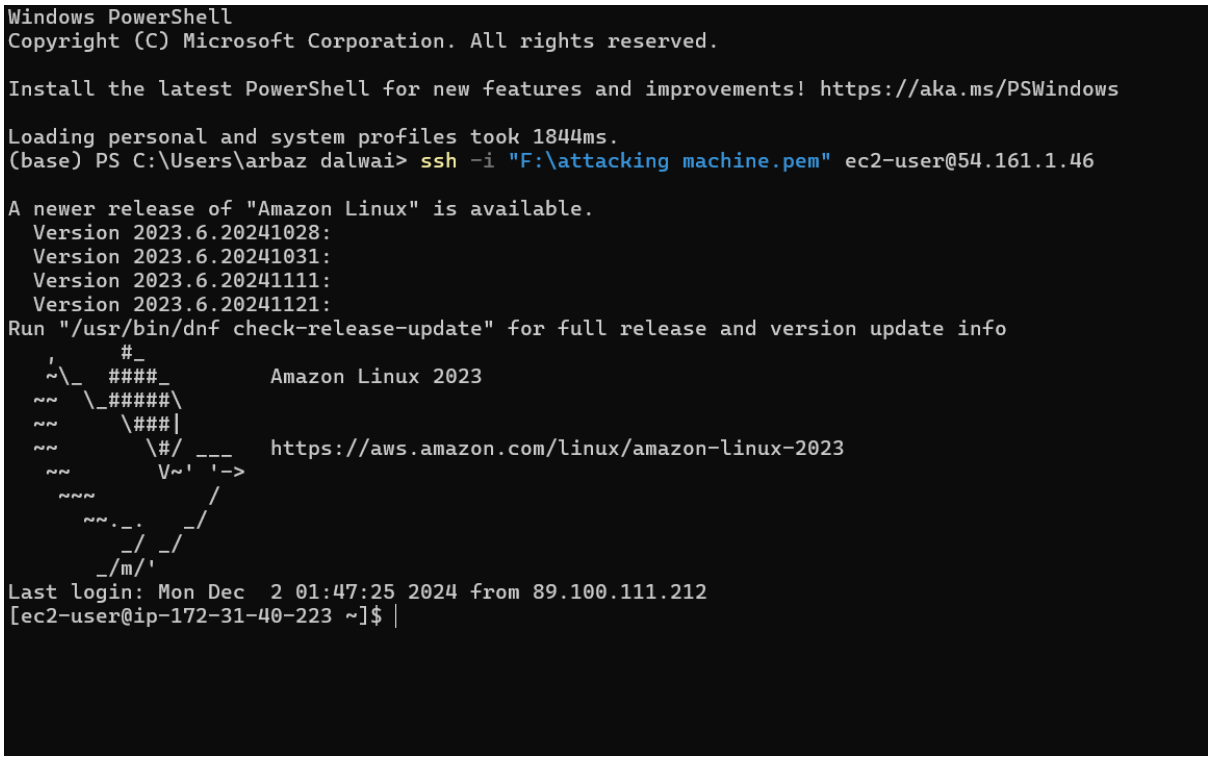


Figure 15

4. The IDS server- Below is the figure 16 showing the same.

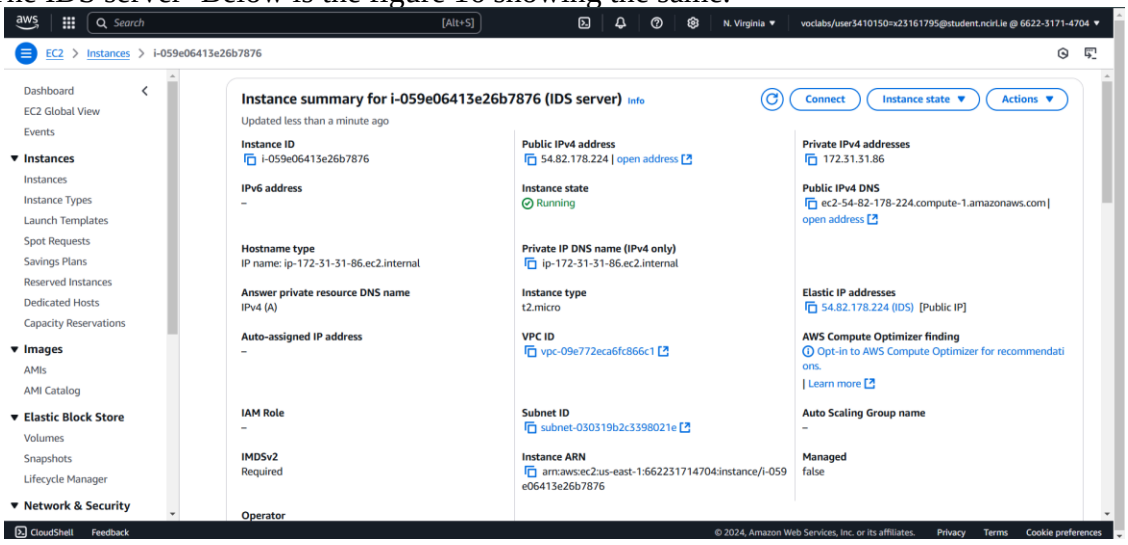


Figure 16

Instance Type: t2.micro
Application OS: Ubuntu; VERSION="24.04.1 LTS (Noble Numbat)"; ID=ubuntu.
Storage: 17 GiB
Key pair: IDS-server.pem
Elastic IP allocated: Public IP- 54.82.178.224; Private IP- 172.31.31.86
Open ports & Security group configuration:

- Port 22 is open for ssh in order to get an administrative access to the instance to configure and manage it.
- All traffic from file server and employee machine allows the IDS server to receive the mirrored traffic from the source for analysis.

The below figure 17 depict the inbound rules.

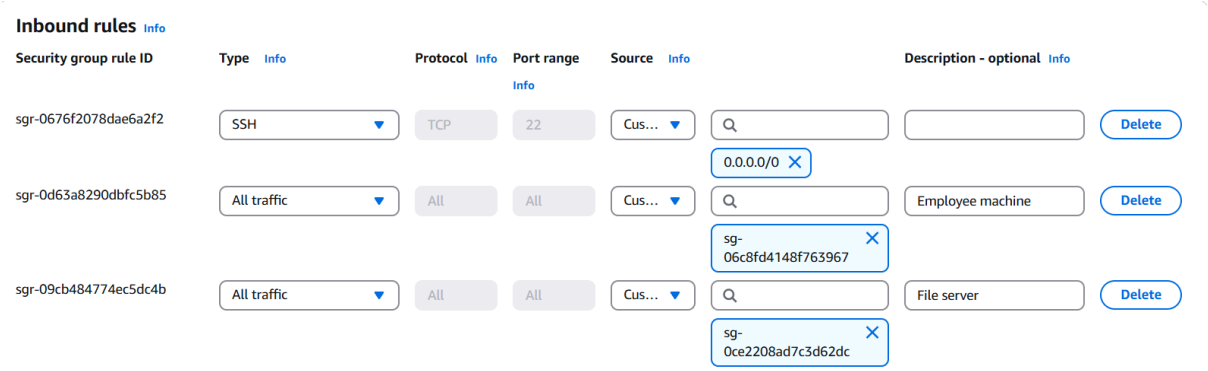


Figure 17

Testing the access connection-

The SSH connection with the instance (attacking machine) is established using the command- **ssh -i "F:\IDS-server.pem" ubuntu@54.82.178.224** and is visible in the below figure 18.

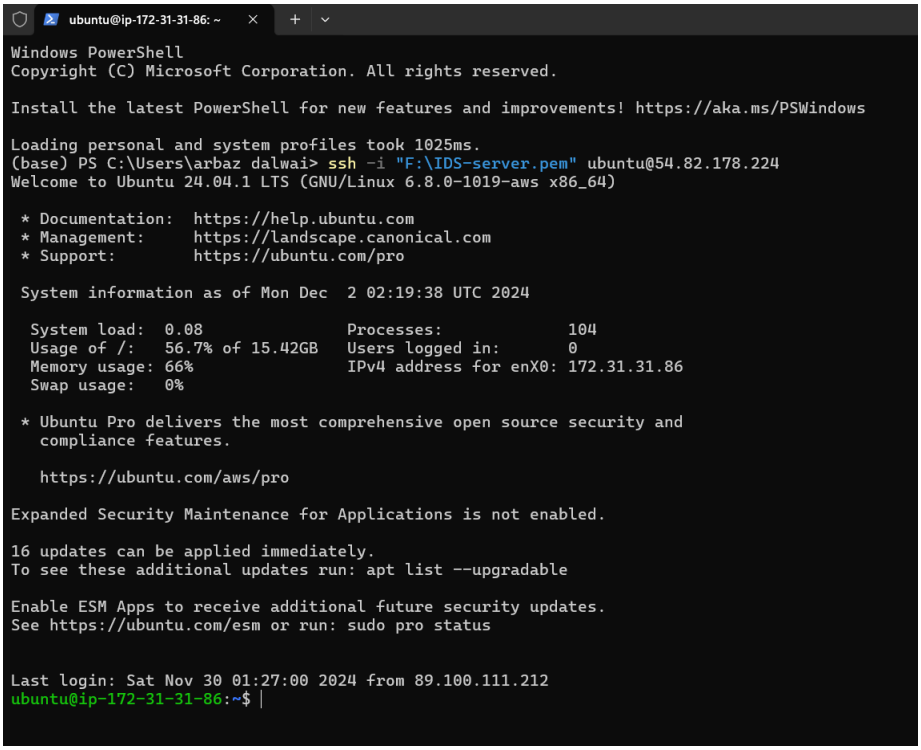


Figure 18

5. The security information and event management (SIEM) server- Below is the figure 19 showing the same.

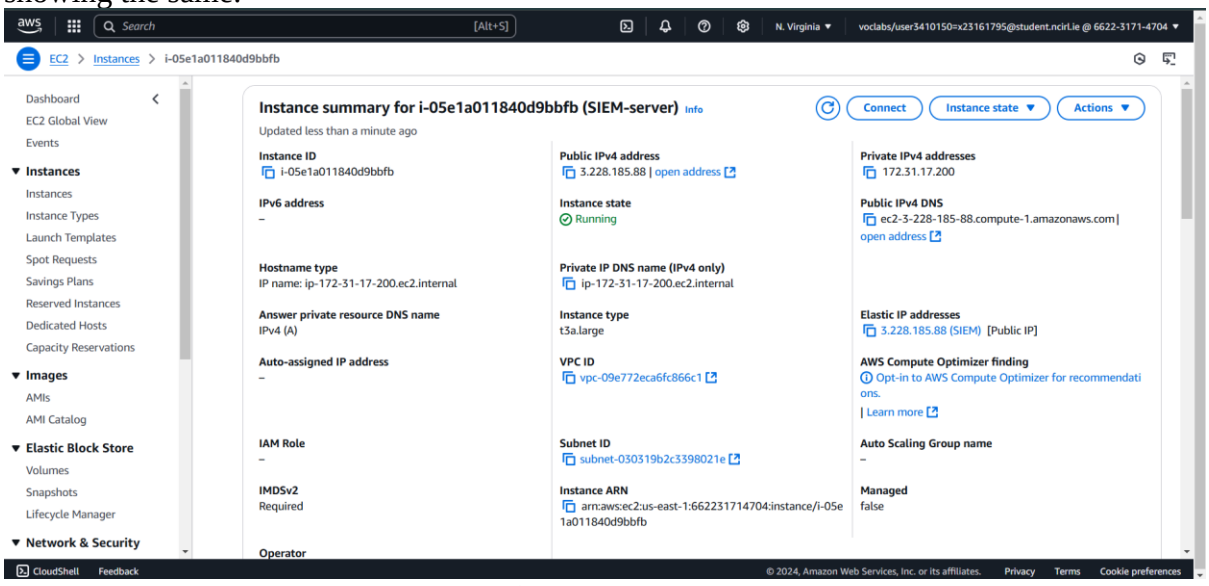


Figure 19

Instance Type: t3a.large
Application OS: Ubuntu; VERSION= "22.04.5 LTS (Jammy Jellyfish)"; ID=ubuntu.
Storage: 35 GiB
Key pair: SIEM.pem
Elastic IP allocated: Public IP- 3.228.185.88; Private IP- 172.31.17.200
Open ports & Security group configuration:

- Port 22 is open for ssh in order to get an administrative access to the instance to configure and manage it.

- Port 80 allows the web access to the elastic, logstash and kibana (ELK) server.
- Port 9200 allows the access to elasticsearch for querying and storing logs.
- Port 5044 allows the Logstash to listen to the logs shared by the filebeat from IDS server.
- Port 5601 allows the Kibana functionalities which acts as the visualization layer for the SIEM.
- Port 9200 with the source IP of machine learning (ml) models allows the connection between the two servers (ml models and SIEM), so that the ML models server is able to fetch the logs for real-time analysis.

The below figure 20 depict the inbound rules.

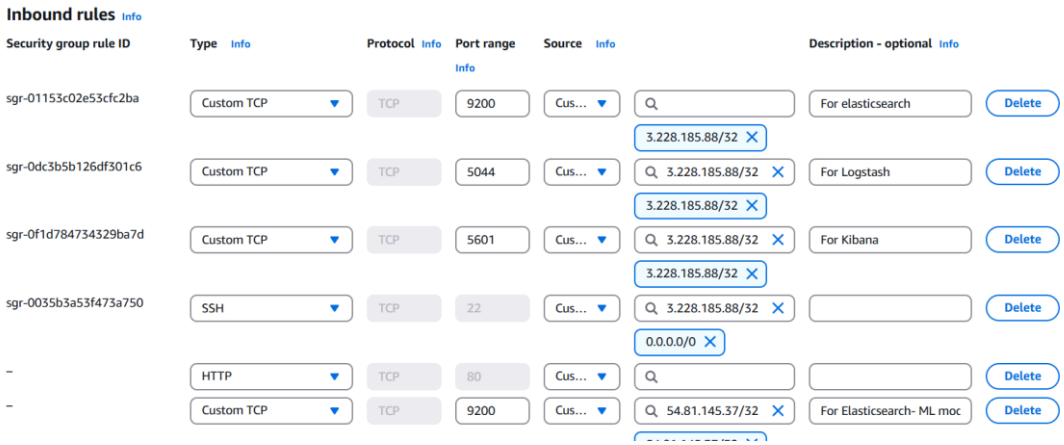


Figure 20

Testing the access connection-

The SSH connection with the instance (attacking machine) is established using the command- **ssh -i "F:\SIEM.pem" ubuntu@3.228.185.88** and is visible through the given figure 21.

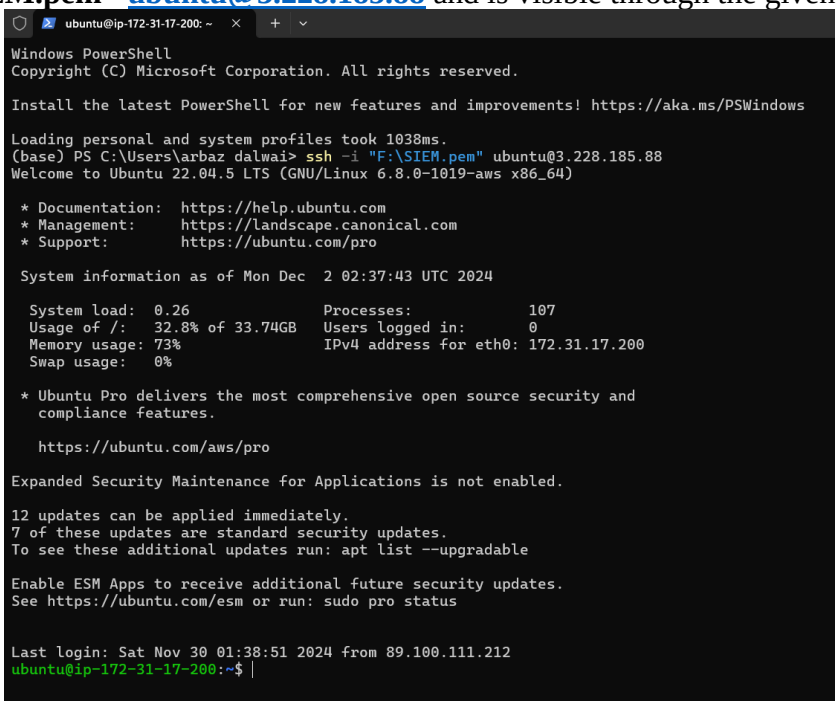


Figure 21

6. The ML models server- Below is the figure 22 showing the same.

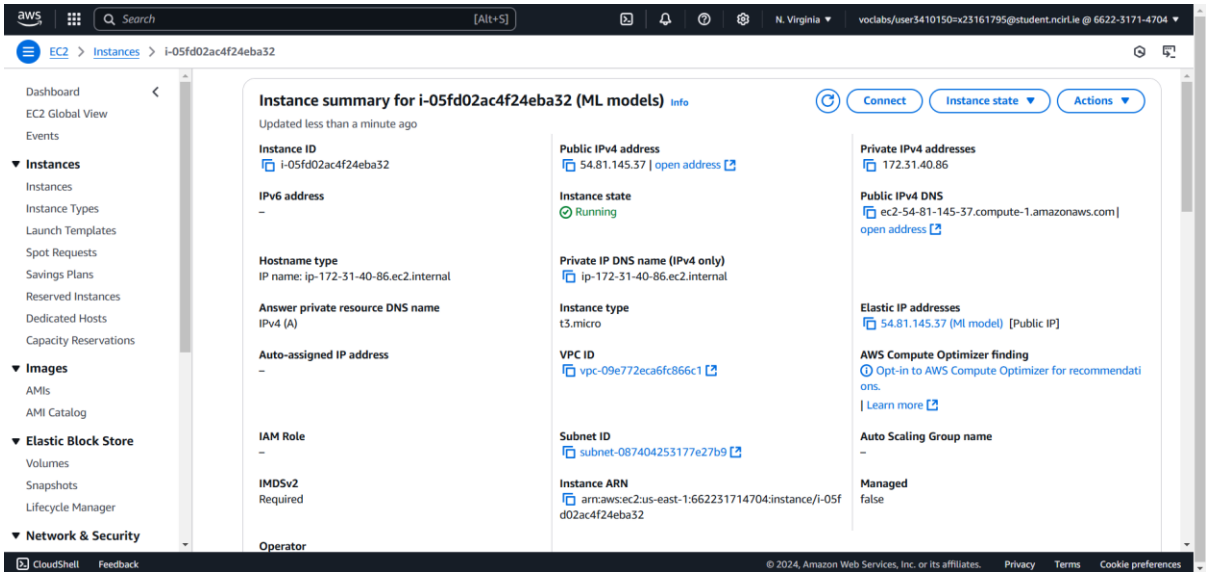


Figure 22

Instance Type: t3.micro
Application OS: Ubuntu; VERSION= "22.04.5 LTS (Jammy Jellyfish)"; ID=ubuntu.
Storage: 8 GiB
Key pair: ml models.pem
Elastic IP allocated: Public IP- 54.81.145.37; Private IP- 172.31.40.86
Open ports & Security group configuration:

- Port 22 is open for ssh in order to get an administrative access to the instance to configure and manage it.
- All traffic in the outbound rules section allows the ml model server to send the predictions and logs back to the elasticsearch (SIEM-ip as shown in the screenshot).

The below figures 23 & 24 depict the inbound and outbound rules resp.

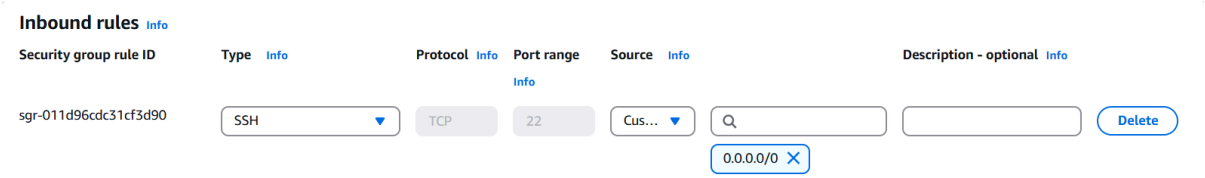


Figure 23

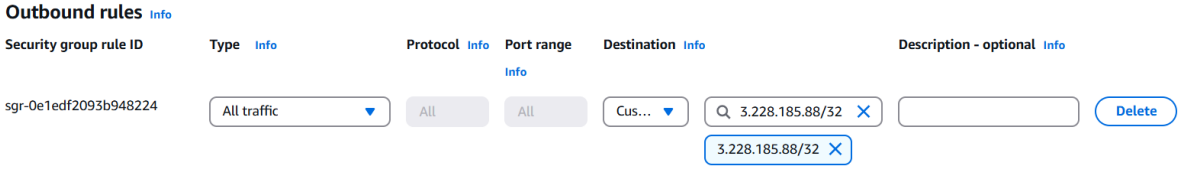


Figure 24

Testing the access connection-

The SSH connection with the instance (attacking machine) is established using the command- **ssh -i "F:\ml models.pem"** ubuntu@54.81.145.37 and is visible through the figure 25.

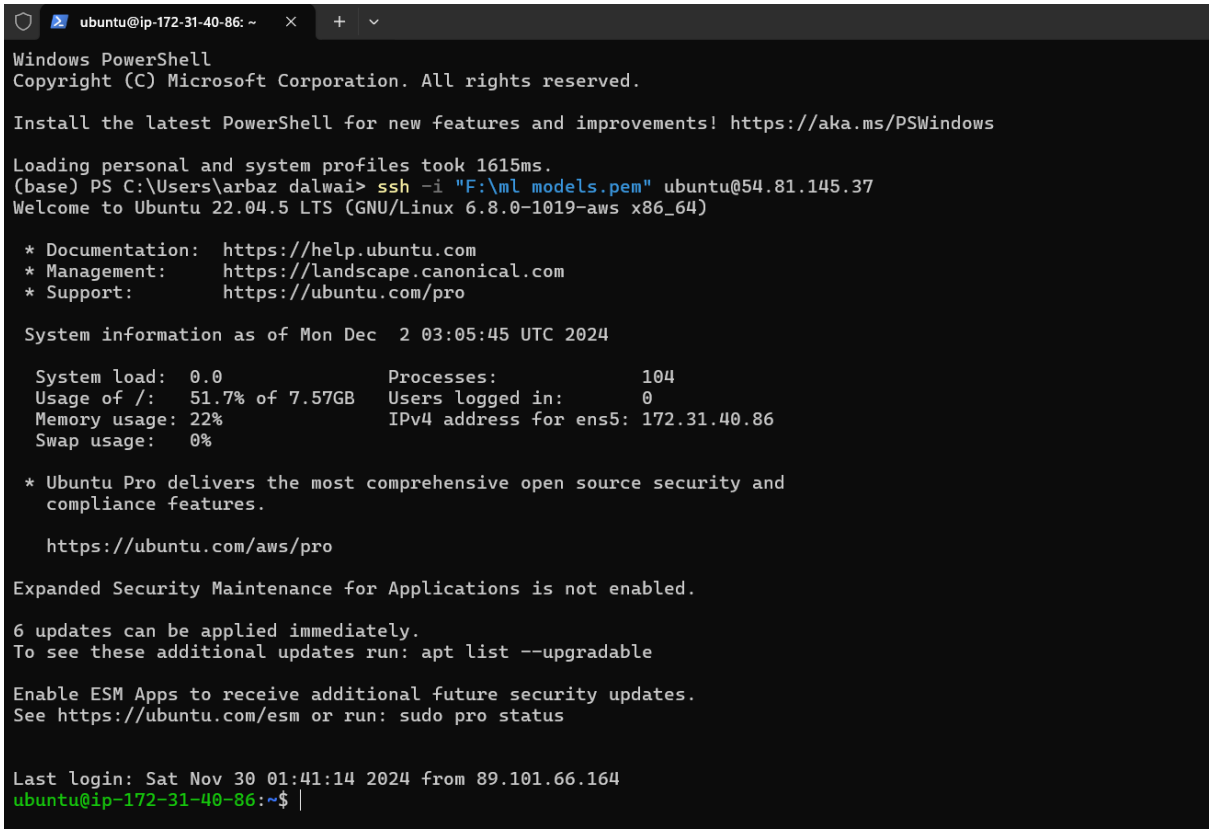


Figure 25

2 Traffic forwarding from File server & Employee machine to the IDS server

For forwarding the traffic from file server and employee server instance, traffic mirroring feature from AWS is used. This guarantees that the IDS gets the traffic from these sources and can monitor all the data/network activities that happen within the network. The steps through which the Traffic Mirroring is incorporated in the thesis are explained below-

1. Setting up the Mirror target

- Navigate to the VPC service on AWS and open the Traffic mirror targets section to create traffic mirror target.
- Set the Target name.
- Select the target type as network interface and select the target from the list of instances/interface ID available (i.e. Elastic network interface ENI) and then save the target. As the IDS had to be selected as the mirror target the following was done in my setup.

Below is the figure 26 showing the creation of the mirror target.

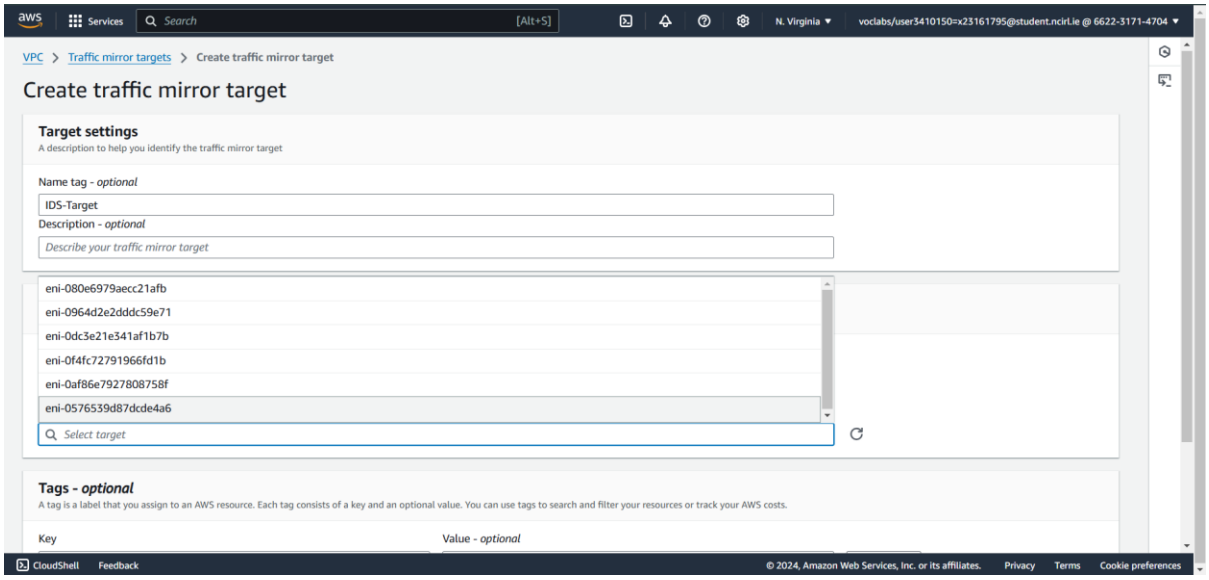


Figure 26

2. Creating Mirror filter
    - Navigate to the VPC section on AWS and open Traffic mirror filters, then create traffic mirror filter
    - Set the name and add the rule in inbound and outbound section and finally save the filter.
    - In this setup as all the traffic had to be sent to IDS the inbound and outbound traffic is set to ALL. This can be seen in the figure 27-

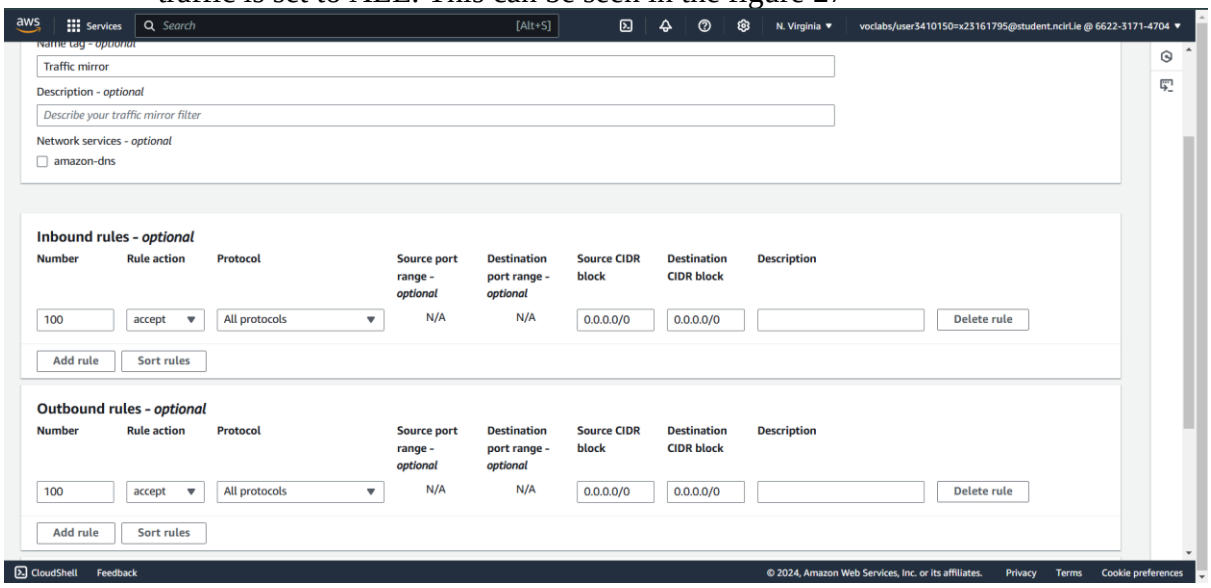


Figure 27

3. Establishing Mirror sessions
    - Navigate to the VPC, then traffic mirror sessions and create traffic mirror session.
    - Set the name for the session.
    - Select the mirror source (a list of ENI) from where the traffic will be initiated (in this case file server eni and employee machine eni)
    - Select the mirror target which is available as per requirements (in this case the IDS target which is already created through above steps).
    - Select the mirror filter (in this case the above filter is selected).
    - Set the session number (any unique number between 1 to 32766).

Figure 28 & 29 show the mirror sessions created for this project-

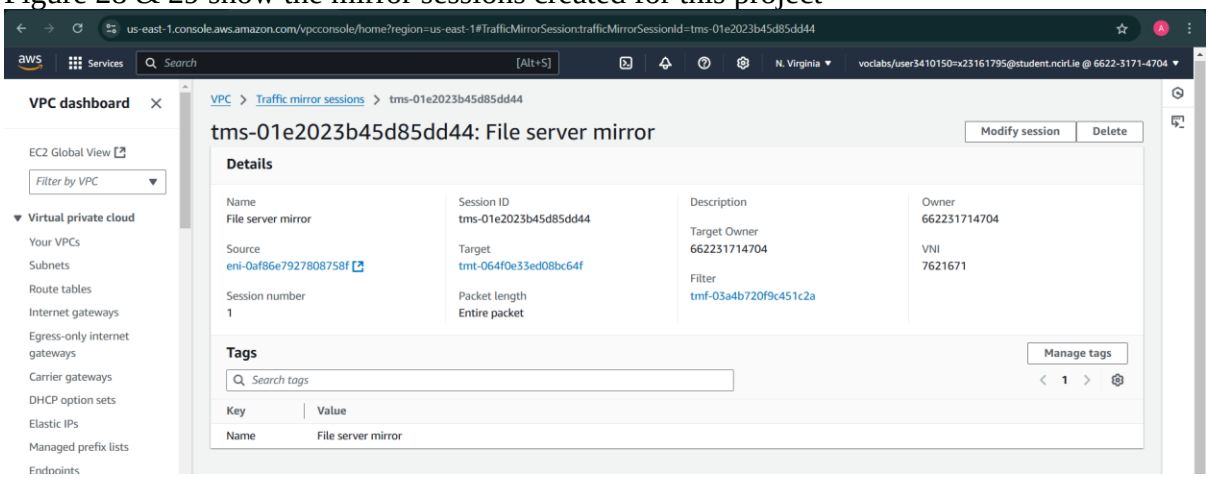


Figure 28

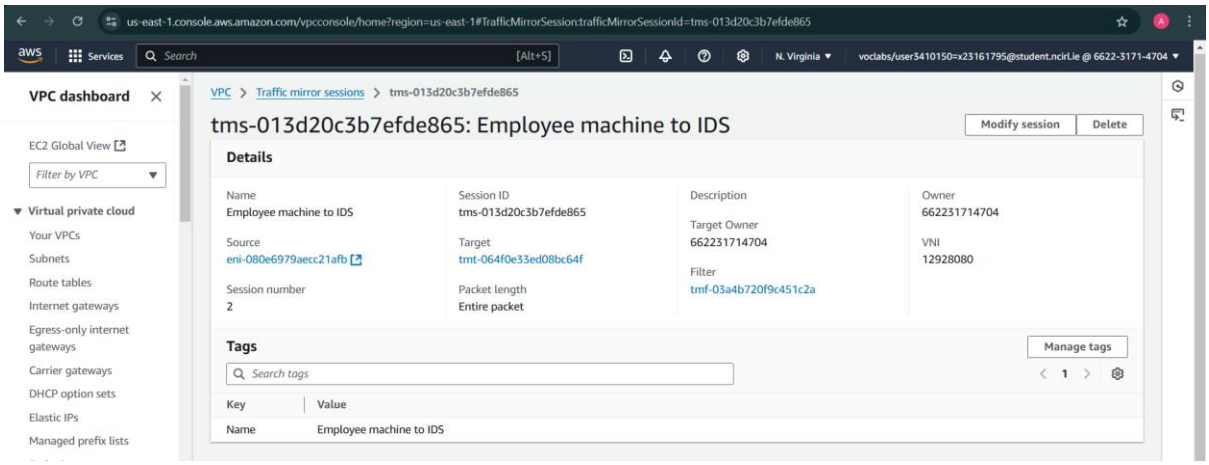


Figure 29

3 Setting up the File server

The process of setting up the File server will be explained in this section.

1. Configuring the web server Apache:
   - Updating the package Repository through-

sudo yum update -y

   - Installing Apache

sudo yum install httpd -y

   - Start and enable Apache

sudo systemctl start httpd
sudo systemctl enable httpd

Server version: Apache/2.4.62 (Amazon Linux)
Server built: Jul 23 2024 00:00:00

The figure 30 depicts the successful loading of the file server.

Figure 30

2. Setting up the login page for File server

   - Navigate to the web root directory where Apache hosts the files: **cd /var/www/html**
   - Creating the login.html file by **sudo nano login.html**
   - The HTML code for login page can be seen in the figure 31-

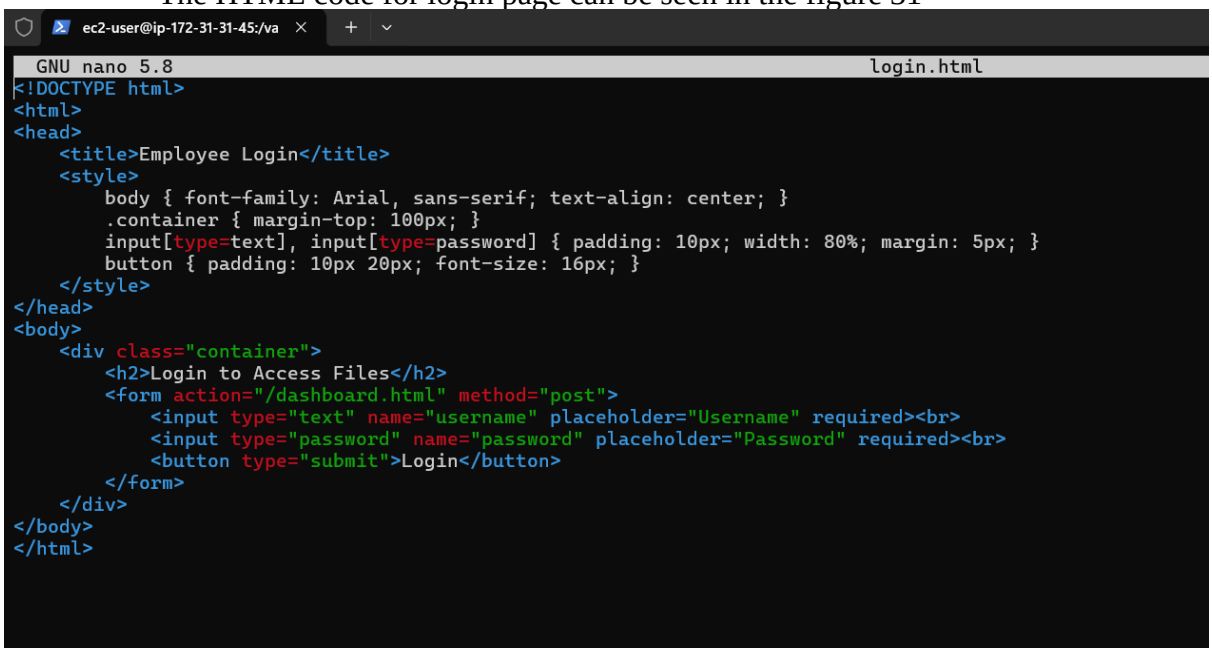


Figure 31

Login page hosted on the file server is presented in the figure 32-

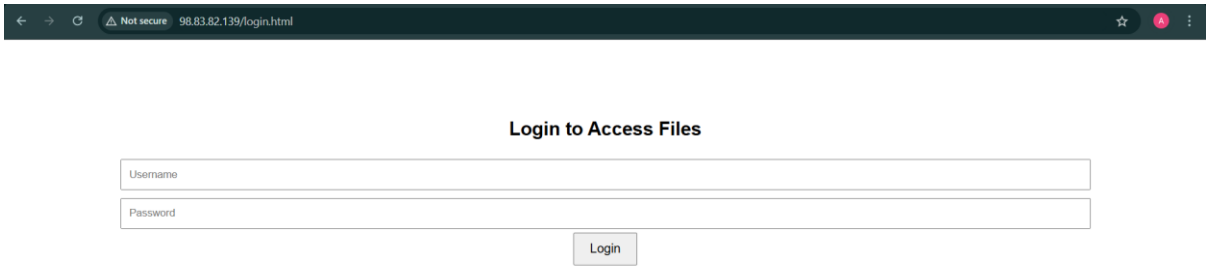


Figure 32

3. Setting up the dashboard for the file server so that it replicates as a companies' crucial asset
    - Navigate to the web root directory where Apache hosts the files: **cd /var/www/html**
    - Creating the dashboard.html file by **sudo nano dashboard.html**
    - The HTML code for employee machine can be seen in the figure 33-

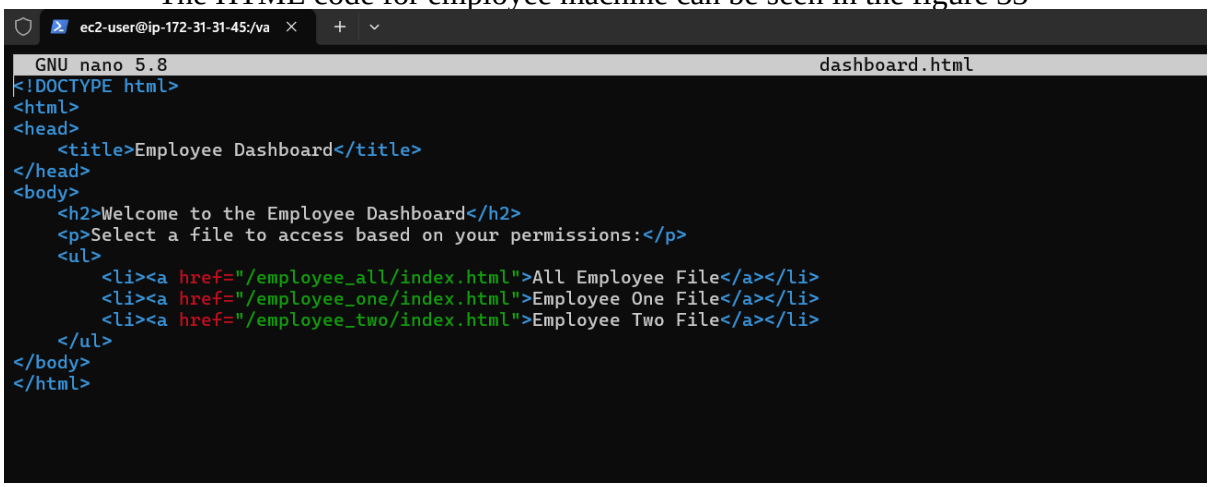


Figure 33

Dashboard page hosted on the file server is presented in the figure 34-

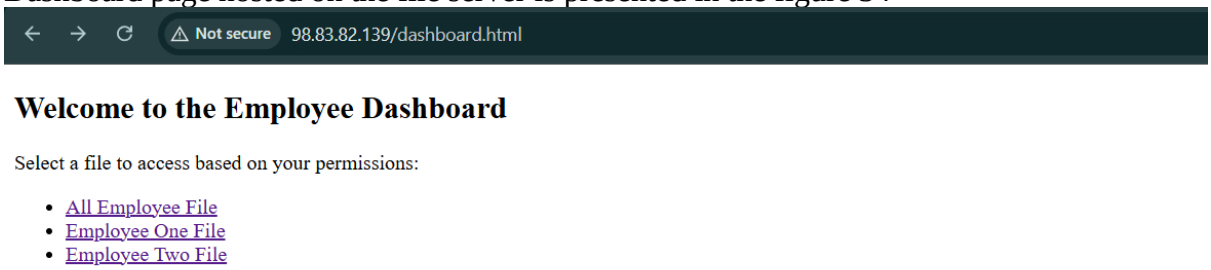


Figure 34

The three files were created in the index.html for each of the mentioned directories i.e. employee_all, employee_one, employee_two. Hence clicking on any of the above will give the foll. Results shown in the figures 35, 36 and 37.

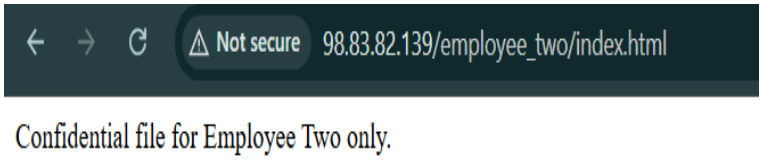


Figure 35

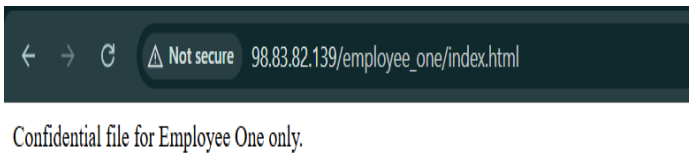


Figure 36

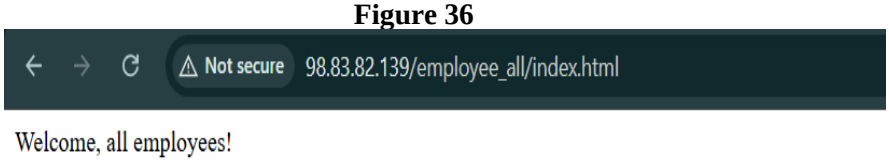


Figure 37

4 Setting up the Attacking Machine

This section would demonstrate the configuration done on the Attacking machine to setup and launch the two simulated attacks- DDoS and phishing.

1. Simulating DDoS attack
   - As this attack was launched through a docker instance first docker was installed using the command **sudo yum install docker -y**. Then it was started and enabled by **sudo systemctl start docker** & **sudo systemctl enable docker** resp.
   - Then we need to run the container, pull the image and name the container using the command- **sudo docker run -it --name attack_container ubuntu**

Below figure 38 represents the active state of the docker.

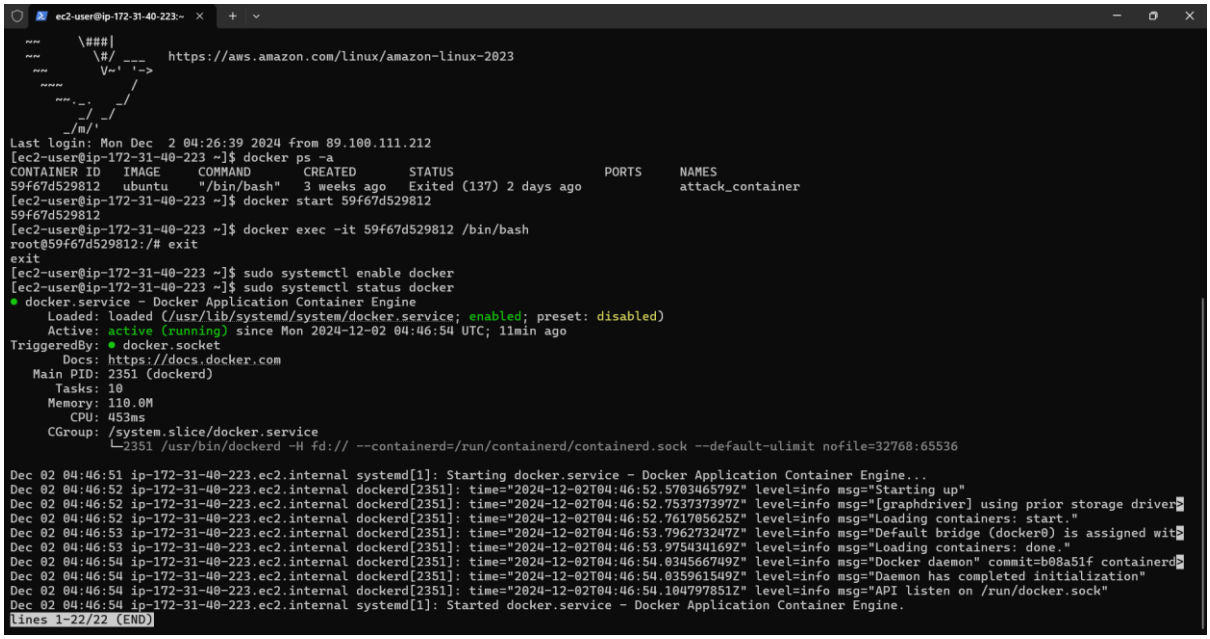


Figure 38

- Then we need to install the hping3 inside the docker container through the below commands-
  apt update
  apt install hping3 -y
- We can now start the ddos attack by attacking the file server on port 80 using the command **hping3 -S -p 80 --flood 98.83.82.139**. This will flood the network traffic of the file server eventually generating the logs we need for further analysis on Suricata IDS. Figure 39 shows the attack initiation through the command.

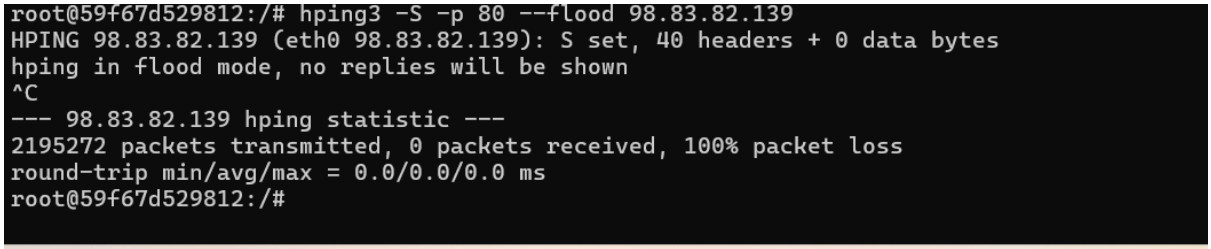


Figure 39

2. Simulating Phishing attack

This attack will be targeted to employee machine, wherein the attacking machine would host a webpage cloned through social engineering toolkit (SET). The process for the same is explained below-

- For installing SET, we will first need to update the system through: sudo yum update && sudo yum upgrade -y
- Then install the pre-requisites for SET: sudo yum install git python3 python3-pip -y
- Then we will have to clone the SET from its GitHub repository:
  git clone https://github.com/trustedsec/social-engineer-toolkit.git
- Once its cloned and available we will move to its directory: cd social-engineer-toolkit/
- Then we'll run the setup script to install SET: sudo python3 setup.py install
- Then we can finally start the SET using- **sudo setoolkit**

Figure 40 depicts the homepage of the SET.

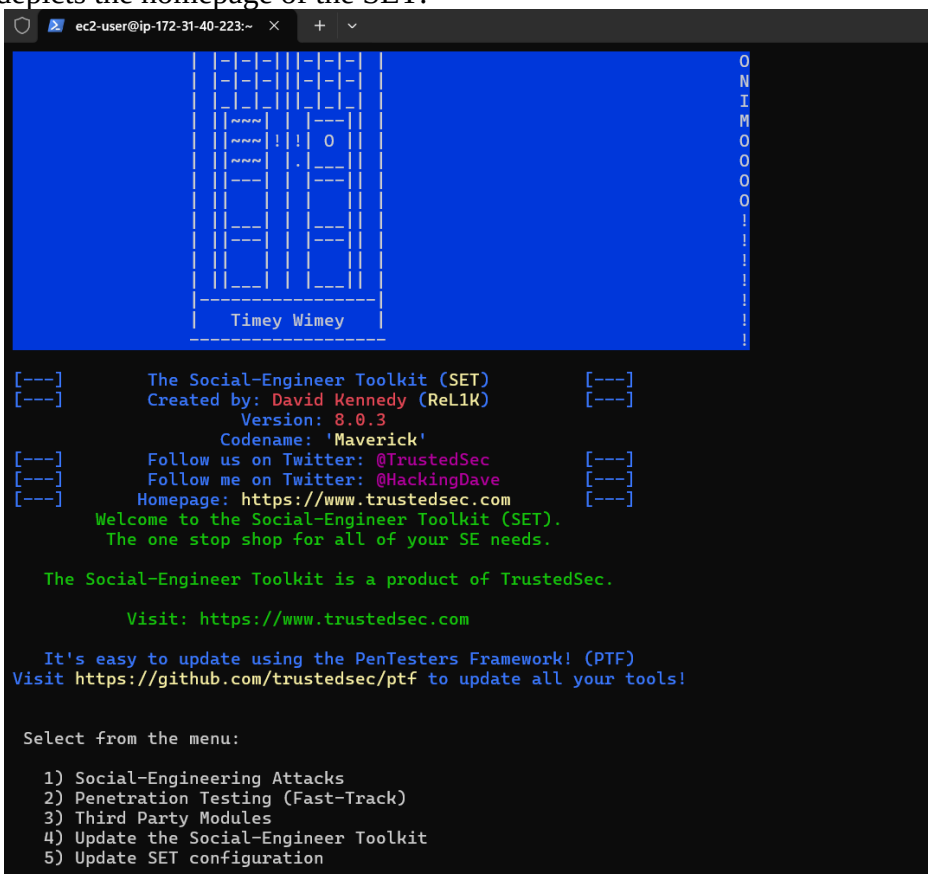


Figure 40

- Once the SET is ready, we can create a phishing page by navigating through the menu SET has to offer
- Choose the attack type by selecting the Option 1- Social Engineering attacks.
- Then choose Website Attack Vectors (Option 2).
- Then select Credential Harvester Attack Method (Option 3).
- From the menu presented now, select Site cloner (option 2).
- Now we will have to provide the POST-Back IP address which is the attacking machine's IP address. But as our attacking machine has a dynamic IP address it will be changing every time it restarts. Hence through NO-IP, I have created a hostname attackingmachine.ddns.net which will be hosting the phishing page. Figure 41 shows the NO-IP page showing the above information.

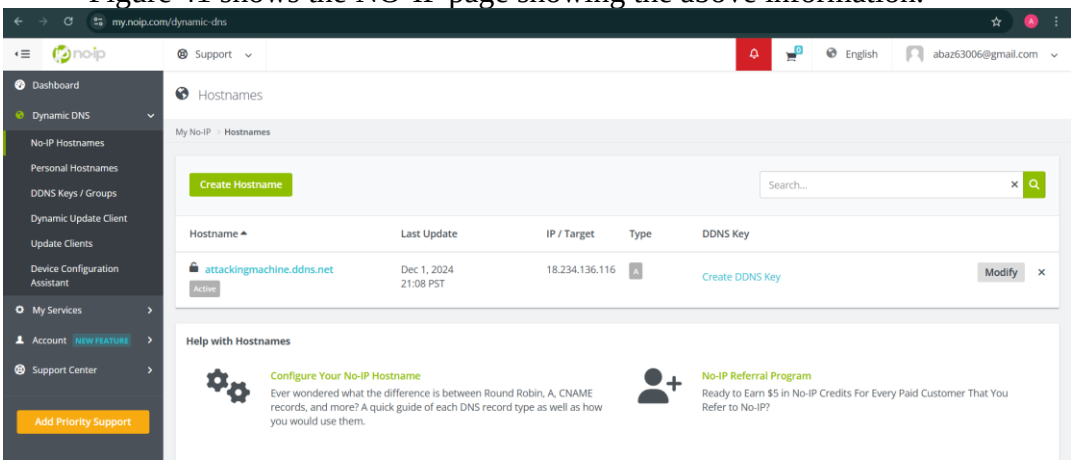


Figure 41

- Now that we have the hostname, we can enter it in the POST-back setup
- Then we will have to input the URL of a known website which is available for testing purposes. (NOTE. The URL's open for public use for research purposes are/should only be used for this cloning setup). We have used http://example.com/ as the website for cloning.
- Thus, now SET is all ready to clone the website and host it.

But this cloning of website through SET being a powerful action was not feasible for long term use as SET uses a lightweight python HTTP server which is only capable of hosting a website when SET is actively running, and once SET is terminated the hosted website also gets inaccessible. However, for phishing simulation it was very necessary to have a stable and persistent website being hosted on the web which could be accessed by the employee machine continuously during the testing phase. Therefore, for overcoming this problem and to minimize the impact of SET's limitation, Apache was selected for hosting the cloned website through which the cloned phishing page would be available as long as the hosted server i.e. EC2 instance is running.

Transitioning the cloned content to Apache-

- Once SET successfully clones the website, the cloned files are stored and available in the directory /var/www/social-engineer-toolkit
- Using the command below the files from SET directory are copied to Apache **sudo cp -r /var/www/social-engineer-toolkit/* /var/www/html**
- Hence the Apache is ready for hosting the cloned website now.
- Through the below Figure 42 it is confirmed that the cloned phishing website is active and accessible at: http://attackingmachine.ddns.net/phishing-page.html (when the attacking machine server is up and running).

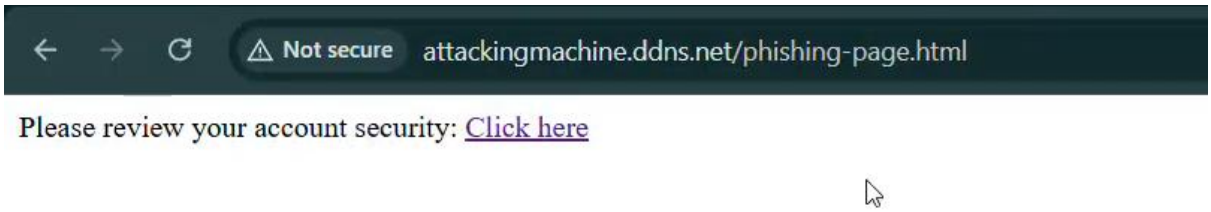


Figure 42

5 Setting up the employee machine server

This section would demonstrate the configuration of employee machine so that it can interact with the above phishing link created in the attacking machine.

As the phishing page is accessible at http://attackingmachine.ddns.net/phishing-page.html through a dynamic domain name service (DNS), it could only be accessed through a script

from the employee machine as currently there is no graphical user interface (GUI) access of the employee machine. The following procedure was used to create a script which would automate the interaction between employee machine and phishing link.

- A bash script named click_phishing_link.sh was created on the employee machine through '**nano'** command.
- The script is designed in a way that it pings the phishing link with HTTP requests every 30 seconds using the '**curl**' command. The script logic can be seen through the figure 43.

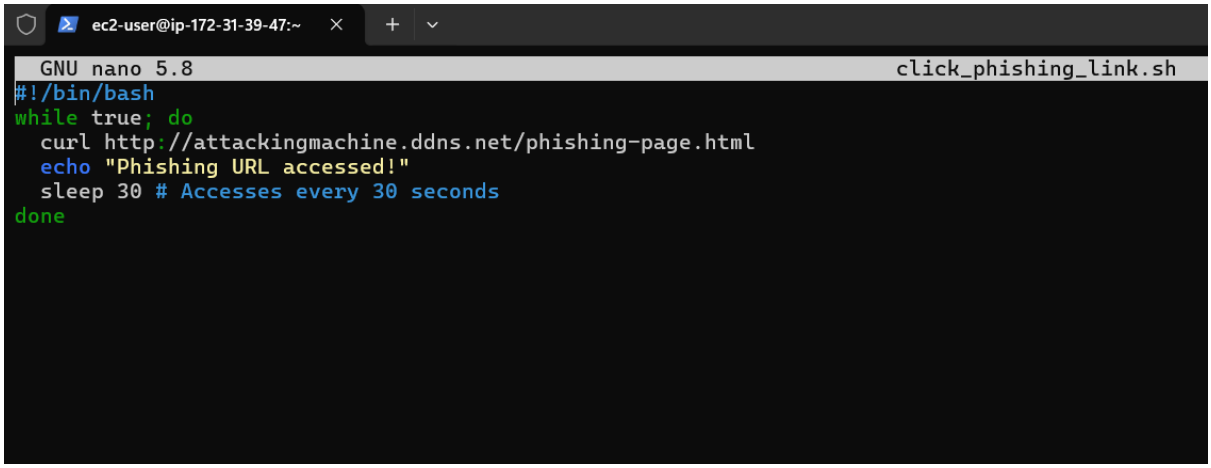


Figure 43

- Once the script file was created it was made executable using the command- **sudo chmod +x click_phishing_link.sh**

For a periodic run of the script without any interruption the '**nohup**' command was used making sure that the script runs, even if the ssh session is disconnected and terminates only when the process is killed. Figure 44 shows the detailed command for running the script.

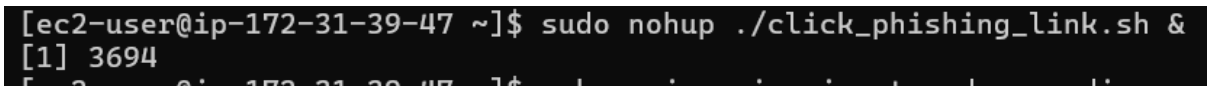


Figure 44

As the script accesses the phishing link on attacking machine, HTTP requests are generated and this traffic is received on the IDS due to the mirroring session done previously, as depicted in the figure 45-

Figure 45

Hence the IDS logs confirm the connection between the employee machine and the phishing link.

6 Setting up the IDS server

This secction will present the process of setting up the Suricata and Filebeat on the IDS server.

Setting up Suricata- IDS:

Step 1: Installing Suricata

- Before installing the suricata it is necessary to update all the required packages, for this the following commands were used-
  sudo apt update && sudo apt upgrade -y
  sudo apt install software-properties-common python3-pip -y
- Next up is adding the suricata repository, the below command adds the official suricata repository to get the latest version of the IDS.
  sudo add-apt-repository ppa:oisf/suricata-stable
  sudo apt-get update
- Once the repository is added the suricata installation can be initiated using the command **sudo apt install suricata -y**
- The successful installation can be confirmed through veiwing the installed version in the figure 46.

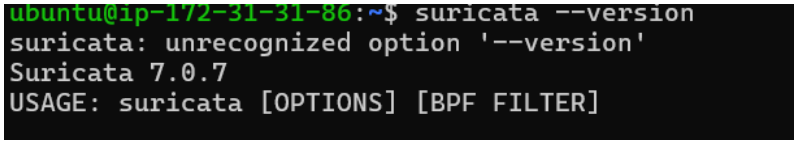


Figure 46

Step 2: Configuring Suricata

The default configuartion file for suricata is available at **/etc/suricata/suricata.yaml**

Before performing any changes in the Suricata yaml file it is necessary to download the Emerging threat (ET) ruleset and place it in the correct directory for Suricata to access. This was done using the below commands-

cd /tmp/
curl -LO https://rules.emergingthreats.net/open/suricata-7.07/emerging.rules.tar.gz
sudo tar -xvzf emerging.rules.tar.gz
sudo mv rules/*.rules /etc/suricata/rules
sudo chmod 640 /etc/suricata/rules/*.rules

Through this we make sure that the Suricata has the latest ruleset from the ET including rules for different attacks, especially DDoS and phishing which is required for this research's case. The figure 47 shows the rules have been extracted and loaded successfully.

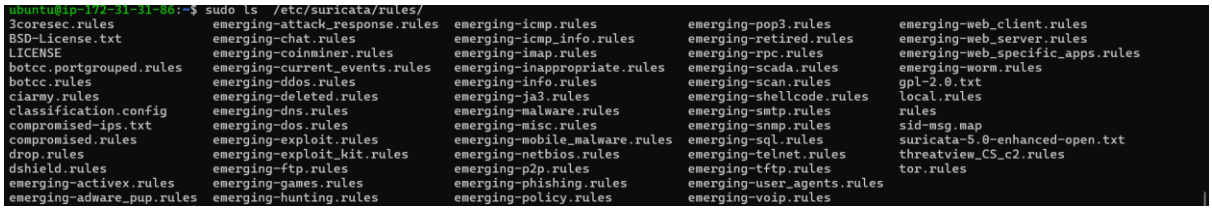


Figure 47

For further configuration, we need to modify the suricata yaml file. The following steps were inculcated for the modification in different areas.

- For accessing the file the below command was used
  sudo nano /etc/suricata/suricata.yaml
- The foremost thing to be done was changing the network interface, by specifying this Suricata would make sure to monitor that particular network. In this case the network interface is the one assigned to the server which hosts the IDS (EC2 instance) and that is enX0.
- The changes need to be made in the **af-packet** section which is shown in the figure 48-

```
# Linux high speed capture support
af-packet:
  - interface: enX0 # your interface
    # Number of receive threads. "auto" uses the number of cores
    #threads: auto
    # Default clusterid. AF_PACKET will load balance packets based on flow.
    cluster-id: 99
    # Default AF_PACKET cluster type. AF_PACKET can load balance per flow or per hash.
    # This is only supported for Linux kernel > 3.1
    # possible value are:
    #  * cluster_flow: all packets of a given flow are sent to the same socket
    #  * cluster_cpu: all packets treated in kernel by a CPU are sent to the same socket
    #  * cluster_qm: all packets linked by network card to a RSS queue are sent to the same
    #  socket. Requires at least Linux 3.14.
    #  * cluster_ebpf: eBPF file load balancing. See doc/userguide/capture-hardware/ebpf-xdp.rst for
    #  more info.
    # Recommended modes are cluster_flow on most boxes and cluster_cpu or cluster_qm on system
    # with capture card using RSS (requires cpu affinity tuning and system IRQ tuning)
    # cluster_rollover has been deprecated; if used, it'll be replaced with cluster_flow.
    cluster-type: cluster_flow
```

Figure 48

- Furthermore, we need to specify the home net(internal or trusted network which is to be monitored and protected; in this case the file server and employee machine) and externel net(any untrusted network e.g Internet) in the config file. Here the home net is assigned

with 3 IPs i.e. IDS's public IP, File server's private IP and employee machine's private IP figure 49 confirms the same. Even though the traffic mirroing is setup assigning this IPs would ensure complete monitoring of the network.

```
vars:
  # more specific is better for alert accuracy and performance
  address-groups:
    HOME_NET: "[54.82.178.224, 172.31.31.45, 172.31.39.47]"

    EXTERNAL_NET: "!$HOME_NET"
    EXTERNAL_NET: "any"
```

Figure 49

- Next up is specifying the default rule path and including all the rules which we have just downloaded using the previous steps. Below is the config screenshot (figure 50) for the same

```
default-rule-path: /etc/suricata/rules

rule-files:
  - "*.rules"
##
```

Figure 50

- Once this changes are made the file can be saved and exited and we can check if there is any error in the configuration using the below command
  sudo suricata -T -c /etc/suricata/suricata.yaml -v

Step 3: Starting Suricata

Suricata now needs to be enabled and started as a system service through the commands-

sudo systemctl enable suricata
sudo systemctl start suricata

Figure 51 shows the active state of the suricata along with the status check command-

```
ubuntu@ip-172-31-31-86:~$ sudo systemctl status suricata
● suricata.service - LSB: Next Generation IDS/IPS
     Loaded: loaded (/etc/init.d/suricata; generated)
     Active: active (running) since Mon 2024-12-02 06:21:45 UTC; 1min 19s ago
       Docs: man:systemd-sysv-generator(8)
    Process: 679 ExecStart=/etc/init.d/suricata start (code=exited, status=0/SUCCESS)
      Tasks: 7 (limit: 1130)
     Memory: 439.5M (peak: 443.7M)
        CPU: 24.793s
     CGroup: /system.slice/suricata.service
             └─734 /usr/bin/suricata -c /etc/suricata/suricata.yaml --pidfile /var/run/suricata.pid --af-packet -D -vvv

Dec 02 06:21:45 ip-172-31-31-86 systemd[1]: Starting suricata.service - LSB: Next Generation IDS/IPS...
Dec 02 06:21:45 ip-172-31-31-86 suricata[679]: Starting suricata in IDS (af-packet) mode... done.
Dec 02 06:21:45 ip-172-31-31-86 systemd[1]: Started suricata.service - LSB: Next Generation IDS/IPS.
ubuntu@ip-172-31-31-86:~$
```

Figure 51

Step 4: Veiwing logs

The generated logs in suricata are stored in the **/var/log/suricata**, figure 52 confirms the same-

```
ubuntu@ip-172-31-31-86:~$ sudo ls /var/log/suricata/
certs  core  eve.1.json  eve.2.json  eve.3.json  eve.json  fast.log  files  stats.log  suricata-start.log  suricata.log
```

Figure 52

Out of the above log files, eve.json and fast.log are the prominent ones as the former includes the detailed JavaScript object notation (JSON) formatted logs (essentially used in our setup) while the latter represent the summarized alerts of the events.

Figures 53 and 54 show the logs both in fast.log as well as eve.json through the '**tail -f**' command, confirming that the IDS is working accurately and the logs are getting generated.

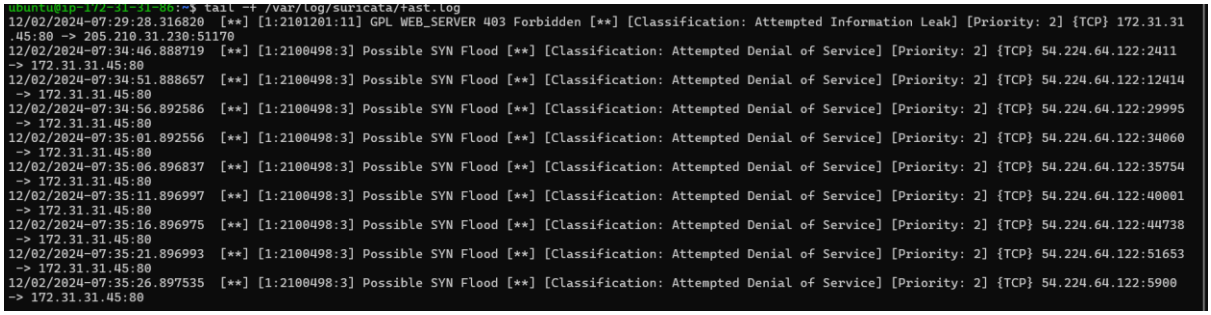


Figure 53

Figure 54

Setting up Filbeat-

Filebeat is an extremely crucial component of the entire log pipeline as it ships the log from source (Suricata) to the destination (SIEM- Logstash).

Step 1: Installing Filebeat

- Downloading and installing filebeat was done by
  sudo apt-get update
  sudo apt-get install filebeat -y

Below is the figure 55 showing the installed version of filebeat

Figure 55

Step 2: Configuring the Filebeat

There are numerous modules readily available in filebeat, Suricata is amongst them. This modules are diabled by default and should be enabled as per the requirements. Thus suricata module is enabled in our setup through the command-

sudo filebeat modules enable suricata

Once the module is enabled there are some changes which need to be done in the Suricata module configuration file which is located in the **/etc/filebeat/modules.d/suricata.yml**
The changes include setting up the input location of the logs, so that the filebeat can access the same. Here the input location as stated in the Suricata section is **/var/log/suricata/eve.json** which is updated in the var.paths as shown in the figure 56-

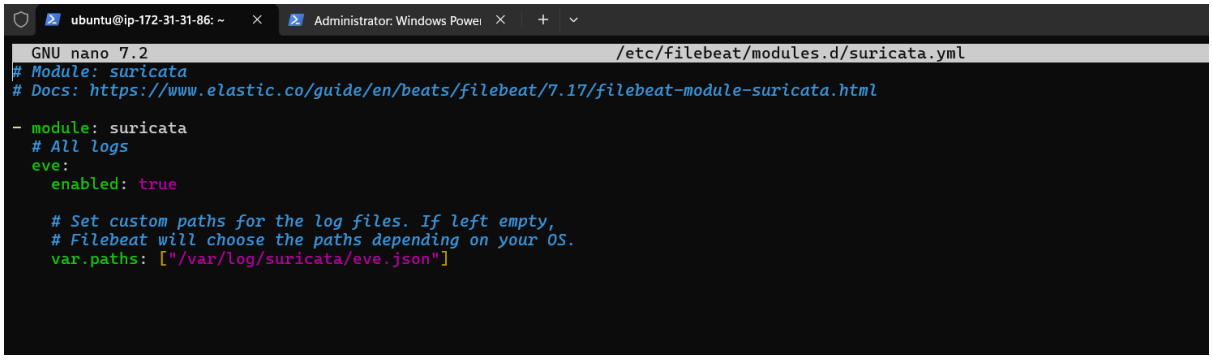


Figure 56

While the filebeat gets the input point for collecting data it is also important to specify the output for this data. For setting this, the main configuration file for filebeat needs to be edited. This file is available at **/etc/filebeat/filebeat.yml** and through the '**nano**' command we can open the same.

Setting Logstash as the output: As logstash is configured to listen on port 5044 at the SIEM end, this needs to be specified in the filebeat config file as stated in the figure 57.
Note: The IP stated here is the SIEM server's public IP.

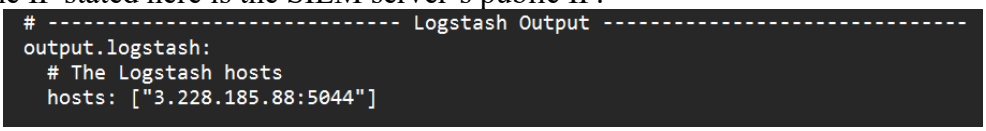


Figure 57

Step 3: Starting Filebeat

Filebeat now needs to be enabled and started as a system service through the commands-

sudo systemctl enable filebeat
sudo systemctl start filebeat

Figure 58 shows the active state of the filbeat along with the status check command-

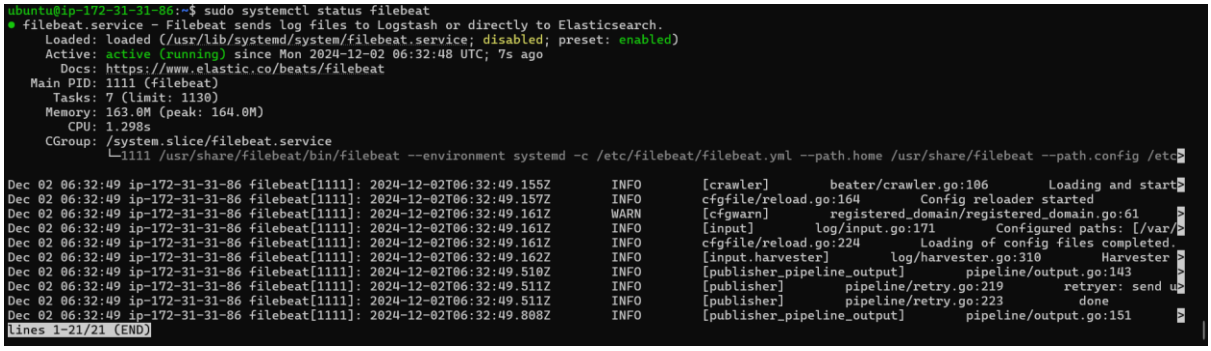


Figure 58

7 Setting up the ELK stack server

The installation and configuration of Elasticsearch, Logstash and Kibana (ELK) stack will be explained in this section.

Step 1: Installing and Configuring Elasticsearch

- For installing the elasticsearch first we need to add the elastic GPG key and repository through the below commands-

wget -q0 – https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add –

echo "deb https://artifacts.elastic.co/packages/7.x/apt stable main" | sudo tee -a /etc/apt/sources.list.d/elastic-7.x.list

sudo apt update

- Finally, the installation command for the setup is-

sudo apt install elasticsearch -y

- For enabling and starting the elasticsearch as a service the given commands are used

sudo systemctl enable elasticsearch
sudo systemctl start elasticsearch

The figure 59 shows the successful installation of the elasticsearch confirming its active state.

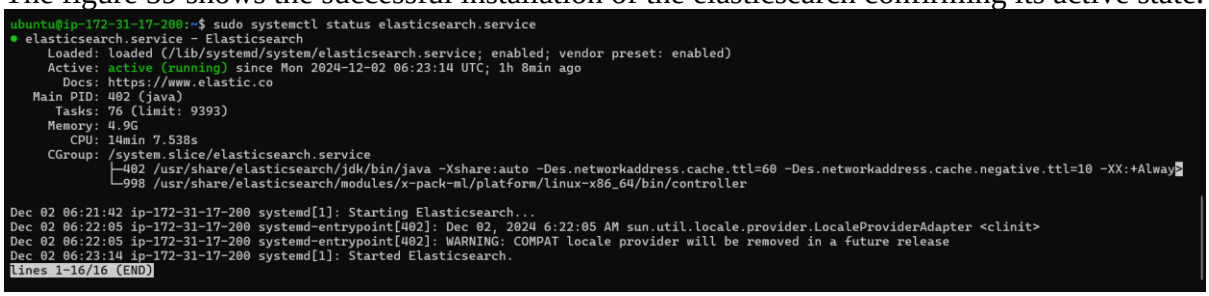


Figure 59

The elasticsearch configuration file is saved by default at /etc/elasticsearch/elasticsearch.yml

The following key changes were done explicitly in the same to suit the requirements of the setup.

- Setting the discovery node to single node to avoid multi-cluster complexity and making the elasticsearch as a standalone instance.
- Enabled xpack.security.enabled as it activates the built-in security features for the service. Enabled application programming interface (API) keys through the xpack.security.authc.api_key.enabled which is used in authenticating the requests. (This was crucial for setting up the usernames and passwords for the services in the SIEM setup). The changes made are visible in the figure 60 from the config. file

```
# ---------------------------------- Security ----------------------------------
#
#                                    *** WARNING ***
#
# Elasticsearch security features are not enabled by default.
# These features are free, but require configuration changes to enable them.
# This means that users don't have to provide credentials and can get full access
# to the cluster. Network connections are also not encrypted.
#
# To protect your data, we strongly encourage you to enable the Elasticsearch security features.
# Refer to the following documentation for instructions.
#
# https://www.elastic.co/guide/en/elasticsearch/reference/7.16/configuring-stack-security.html
discovery.type: single-node


xpack.security.enabled: true
xpack.security.authc.api_key.enabled: true
```

Figure 60

- Once these changes were implied the service was restarted using the command **sudo systemctl restart elasticsearch**
- Once the system was up and running with the changes the below command was used to set the superuser's password as well as Kibana's password.
  sudo /usr/share/elasticsearch/bin/elasticsearch-setup-passwords interactive
- Through this command an interactive prompt is launched wherein we need to input the password of one's choice.
- Through the curl command in the figure 61, the successful setup for the elasticsearch can be confirmed.

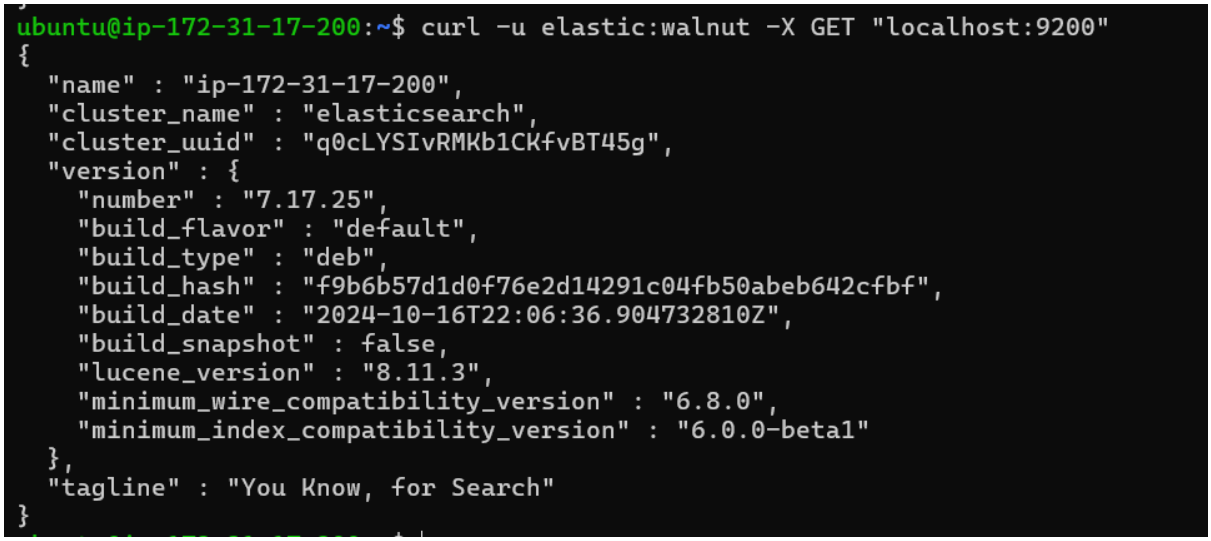


Figure 61

Step 2: Installing and Configuring Logstash

As the elastic repository and GPG key are already added, we just need to install the logstash service now using the command- **sudo apt install logstash -y**
Once logstash was installed it was enabled and started as a service using the commands

sudo systemctl enable logstash
sudo systemctl start logstash

The figure 62 shows the successful installation of the logstash confirming its active state.

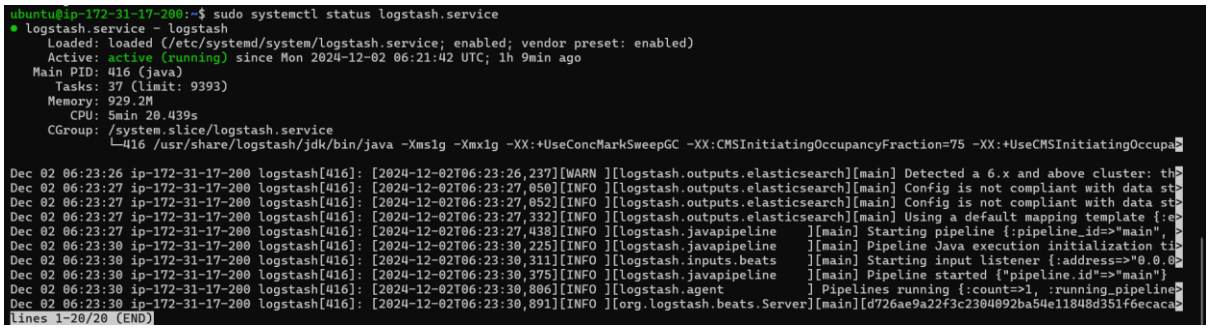


Figure 62

Logstash is put in place to listen the logs from filebeat and structure them to the elasticsearch in indices. For setting up this functionality firstly we have to make some changes in the pipelines.yml file located at **/etc/logstash/pipeline.yml**
Opening it through '**nano**' command the path.config was set to location wherein the configuration files are saved for logstash. The figure 63 represents the same

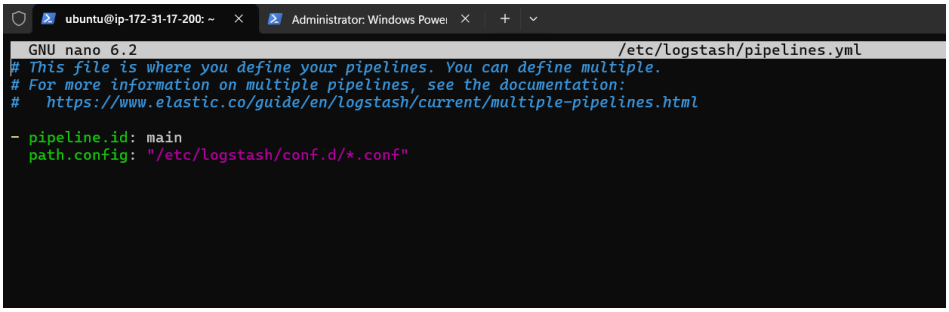


Figure 63

Then it was necessary to create a pipeline configuration file and this was created using the command- sudo nano /etc/logstash/conf.d/filebeat-suricata.conf

In this file, a detailed description of the functionalities expected from logstash are mentioned, right from listening on port 5044 for the input from filebeat up to the indexing output for elasticsearch. Filtering section is also added to ensure that only the relevant logs are processed further and these were categorised as alerts. The file configurations can be seen in the figure 64

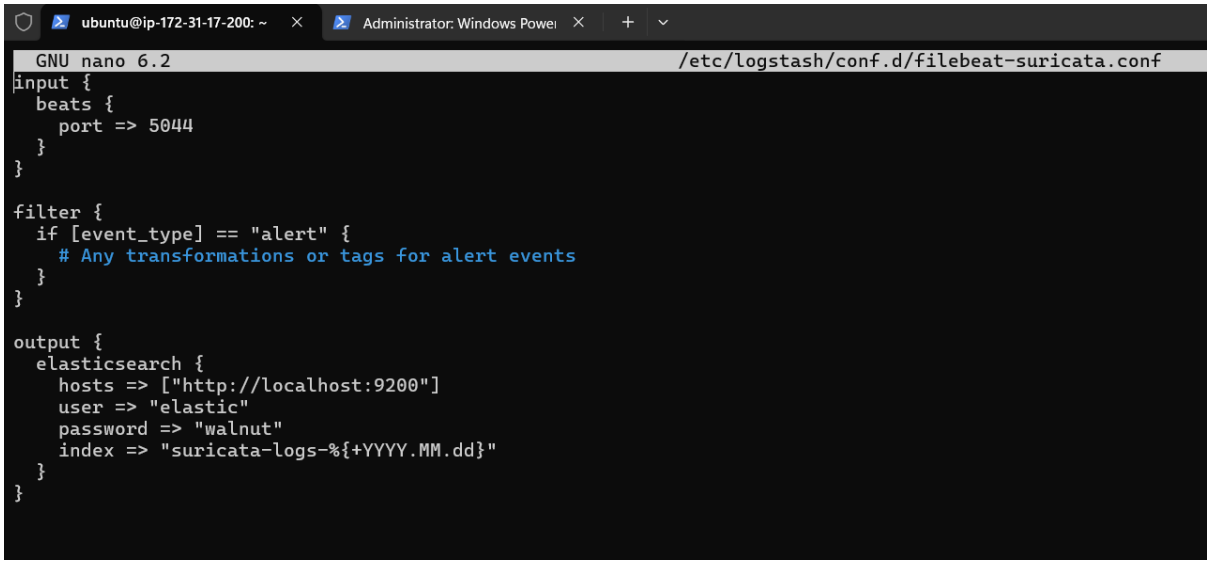


Figure 64

The following figure acts as a confirmation that the elasticsearch is receiving the indexed logs from the logstash, validating the log pipeline Suricata-> Filebeat-> Logstash-> Elasticsearch working seamlessly. The command for fetching the indices is also given in the figure 65.

Figure 65

Step 3: Installing and Configuring Kibana

Installation of kibana was done through the command- **sudo apt install kibana -y**
Once kibana was installed it was enabled and started as a service using the commands

sudo systemctl enable kibana
sudo systemctl start kibana

The figure 66 shows the successful installation of the kibana confirming its active state.

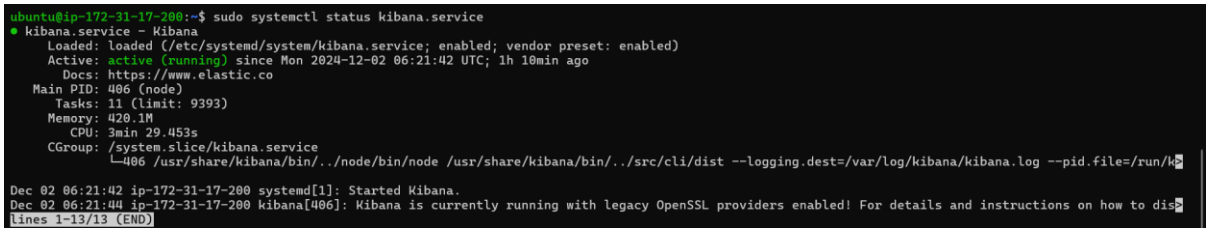


Figure 66

The configuration file for kibana is available at **/etc/kibana/kibana.yml**
Once we open the config file through '**nano**' command authentication credentials for the elasticsearch service were provided, through which kibana user can access the elasticsearch. the figure 67 shows the same.

```
# If your Elasticsearch is protected with basic authentication, these settings provide
# the username and password that the Kibana server uses to perform maintenance on the Kibana
# index at startup. Your Kibana users still need to authenticate with Elasticsearch, which
# is proxied through the Kibana server.
elasticsearch.username: "kibana_system"
elasticsearch.password: "walnut"
```

Figure 67

Moreover, the settings shown in the figure 68 were incorporated for encryption and reporting enhancements, enabling additional encryption for sensitive data as well as optimizing the report exporting settings from the Kibana's discover tab

```
xpack.encryptedSavedObjects.encryptionKey: "uydvgbhjbduhdbhvhfhvbrgfuiwrfvbf"
xpack.reporting.csv.maxSizeBytes: 104857600
xpack.reporting.csv.scroll.size: 10000
xpack.reporting.queue.timeout: 600000
```

Figure 68

The following figures 69 & 70 (login page and discover tab resp.) shows that the kibana is available and functioning as expected

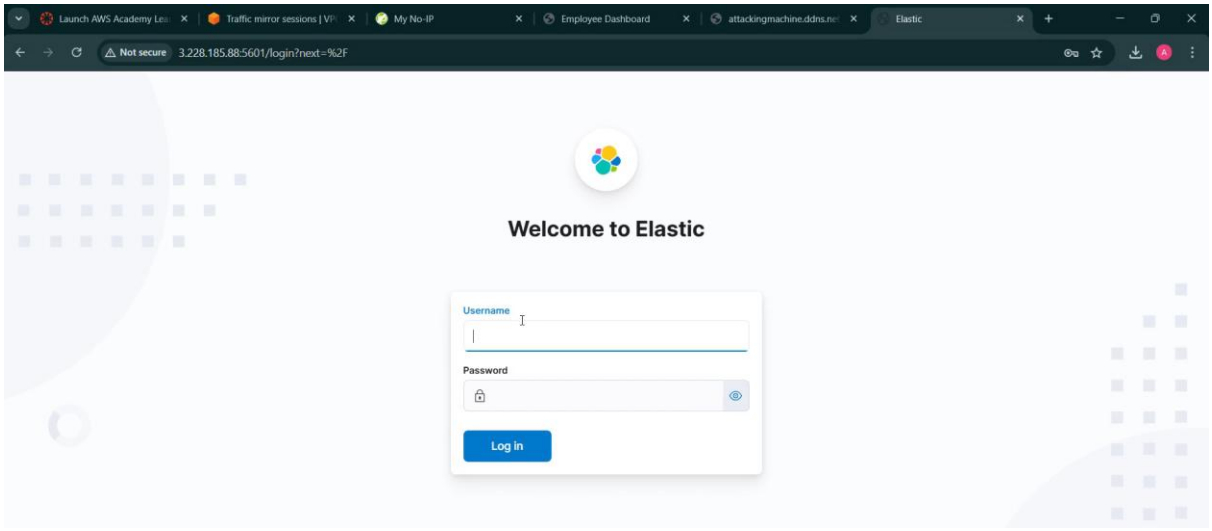


Figure 69

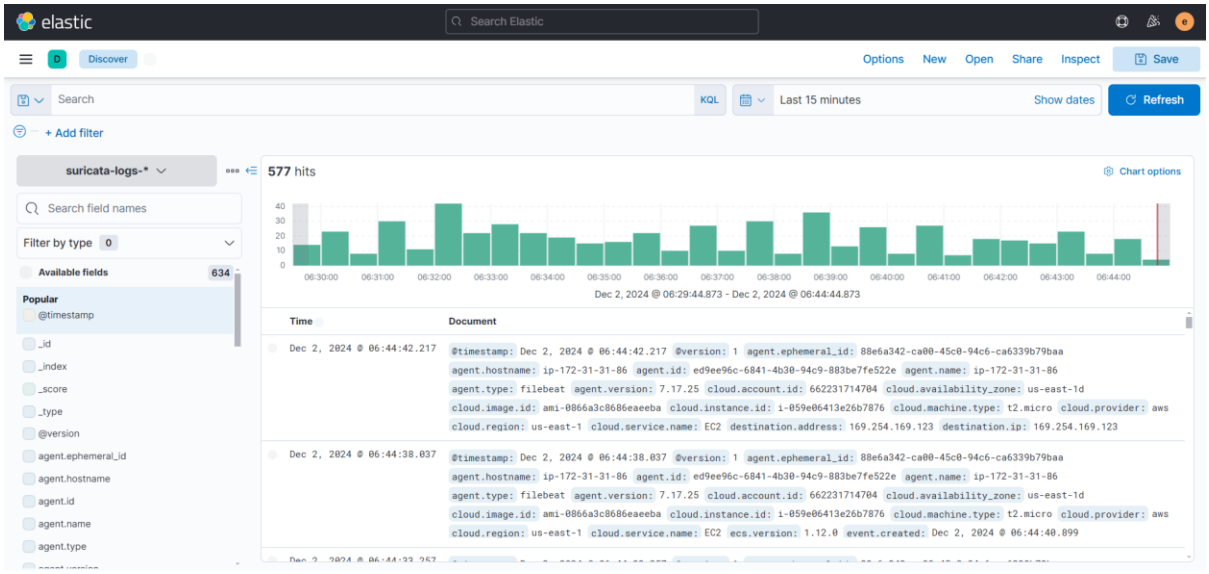


Figure 70

8 Setting up the ML Model server

In this section, the detailed steps for training the ensemble learning models (Random Forest and XGBoost) and using them for fetching the logs and indexing back to elasticsearch are explained-

1. Training the Models (Random Forest and XGBoost)

- Dataset Preparation: Firstly, it was important to collect the data on which the training and testing was to be done. This dataset was acquired from the elasticsearch itself to avoid any feature mismatch issue in the later stage of the implementation wherein the trained models will actually be making the predictions. This dataset was extracted using the '**curl**' command from elasticsearch. The logs are exported from the specific index which is suricata-logs-* as this is the index where the suricata logs are placed given their generated timeline. These logs are saved as JSON files in the local machine. The figure 71 depicts the same process of data extraction.

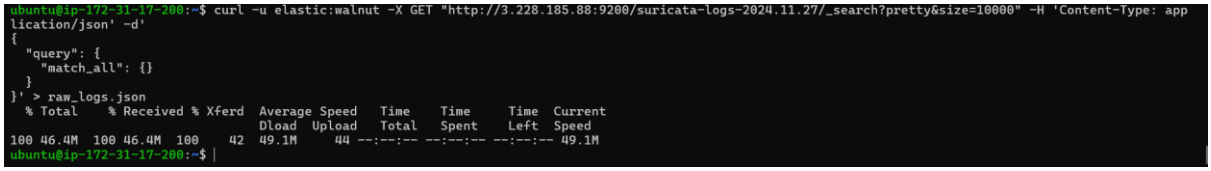


Figure 71

- Importing necessary libraries: The libraries given in the table below were used in the training process due to their mentioned reasons.

Table 1

Libraries	Working
Pandas	For data manipulation and creating data frames
NumPy	For numerical and array operations
Json	For parsing the Json files which contain the raw log data.
Sklearn	For handling the preprocessing steps, splitting the data into training and testing the ML models, standard scaling, label encoding, loading the Random Forest model
XGBoost	For implementing the XGBoost model
Joblib	For saving and loading the models

The figure 72 shows the imported libraries in the process.

```python
# Importing libraries for data handling and numerical computations
import pandas as pd
import numpy as np

# Importing libraries for data preprocessing
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Importing libraries for data splitting and model training
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier

# Importing libraries for evaluation and saving models
from sklearn.metrics import accuracy_score, classification_report
import joblib
```

Figure 72

- Loading and Parsing the extracted dataset: The raw logs extracted from the elasticsearch are loaded here. Here pandas are used to parse and load the dataset (JSON logs) into a Dataframe structure. The nested structure of JSON fields such as hits and _source are handled so as to extract only the necessary relevant fields. The figure 73 is the code snippet for the same.

```python
# Load the raw JSON logs
import json

# Load the JSON file
with open('/content/drive/MyDrive/exported_logs.json') as file:
    data = json.load(file)

# Check the root keys and inspect a sample of data to find the correct path
print("Root keys:", data.keys())

# Assuming the data you need is under 'hits' -> 'hits' -> '_source'
records = [hit['_source'] for hit in data['hits']['hits']]

# Convert to DataFrame
df = pd.DataFrame(records)

# Display first few rows to verify
print("Initial DataFrame:")
print(df.head())
```

Figure 73

Output for the above code in figure 74-

```
Root keys: dict_keys(['took', 'timed_out', '_shards', 'hits'])
Initial DataFrame:
                input                                              agent  \
0  {'type': 'log'}  {'hostname': 'ip-172-31-31-86', 'name': 'ip-17...
1  {'type': 'log'}  {'hostname': 'ip-172-31-31-86', 'name': 'ip-17...
2  {'type': 'log'}  {'hostname': 'ip-172-31-31-86', 'name': 'ip-17...
3  {'type': 'log'}  {'hostname': 'ip-172-31-31-86', 'name': 'ip-17...
4  {'type': 'log'}  {'hostname': 'ip-172-31-31-86', 'name': 'ip-17...

              @timestamp  \
0  2024-11-12T21:28:36.778Z
1  2024-11-12T21:28:36.783Z
2  2024-11-12T21:28:36.783Z
3  2024-11-12T21:28:36.783Z
4  2024-11-12T21:28:36.783Z

                                                 log  \
0  {'file': {'path': '/var/log/suricata/eve.json'...
1  {'file': {'path': '/var/log/suricata/eve.json'...
2  {'file': {'path': '/var/log/suricata/eve.json'...
3  {'file': {'path': '/var/log/suricata/eve.json'...
4  {'file': {'path': '/var/log/suricata/eve.json'...

                                                host  \
0  {'hostname': 'ip-172-31-31-86', 'os': {'kernel...
1  {'hostname': 'ip-172-31-31-86', 'os': {'kernel...
2  {'hostname': 'ip-172-31-31-86', 'os': {'kernel...
3  {'hostname': 'ip-172-31-31-86', 'os': {'kernel...
4  {'hostname': 'ip-172-31-31-86', 'os': {'kernel...

                                               event  dns
0  {'original': '{"timestamp":"2024-11-12T21:28:3...  NaN
1  {'original': '{"timestamp":"2024-11-12T21:28:3...  NaN
2  {'original': '{"timestamp":"2024-11-12T21:28:3...  NaN
3  {'original': '{"timestamp":"2024-11-12T21:28:3...  NaN
4  {'original': '{"timestamp":"2024-11-12T21:28:3...  NaN
```

Figure 74

- Data cleaning and feature extraction: The code presented in the figure below handles the preprocessing of raw logs making sure that the missing or incomplete keys are handled appropriately in the nested JSON structure. the main focus of this preprocessing step was to extract the necessary features required for training the model ahead. This included fields such as event_type, src_ip, dest_ip, src_port, dest_port, user_agent and hostname. The Python's json.loads() function is used to parse the event field that contains a nested Json structure, the extracted fields are then compiled in a dataframe structure for easier handling in the future manipulation of the data. Figure 75 shows the code snippet for the same-

```
# Extract relevant details from the 'event' field's nested JSON structure, handling missing keys
df['event_type'] = df['event'].apply(lambda x: json.loads(x['original']).get('event_type') if isinstance(x, dict) else None)
df['src_ip'] = df['event'].apply(lambda x: json.loads(x['original']).get('src_ip') if isinstance(x, dict) else None)
df['dest_ip'] = df['event'].apply(lambda x: json.loads(x['original']).get('dest_ip') if isinstance(x, dict) else None)
df['src_port'] = df['event'].apply(lambda x: json.loads(x['original']).get('src_port') if isinstance(x, dict) else None)
df['dest_port'] = df['event'].apply(lambda x: json.loads(x['original']).get('dest_port') if isinstance(x, dict) else None)
df['user_agent'] = df['event'].apply(lambda x: json.loads(x['original']).get('http_user_agent', None) if isinstance(x, dict) else None)
df['hostname'] = df['host'].apply(lambda x: x.get('hostname') if isinstance(x, dict) else None)
```

Figure 75

- Tagging the attacks: Here the events are tagged as attacks for further predictions that is phishing and DDoS while the normal traffic is categorised as other. Mainly two approaches are used for the tagging logic which are condition-based tagging (tag_phishing_logs) and row-wise tagging (tag_event). This dual approach tagging helps in tagging each and every event correctly without missing out on potential attacks. Firstly, in tagging phishing logs specific conditions like event type must be an alert, destination port should be either 80 or 443, http_content_type should be indicating the HTML content whereas http_user_agent should contain either curl or wget which would mean that the automated tools were used for accessing the links and finally http method should be GET were used. Once any event is meeting these conditions it is tagged as phishing. Next up, in the row-wise tagging which basically works in a broader classification to tag each row individually. The DDoS tagging works mainly on three parameters, as the commonly targeted ports in a ddos attack are 53 (DNS), 123 (NTP), 80 (HTTP) and 443 (HTTPS), any event coming at these ports were suspicious, along with that when a event_type is flow it generally would signify multiple connections attempts resembling ddos attack, furthermore if the src_ip and dest_ip are differing indicating external traffic targetting a specific resource it would also contribute in marking the event as suspicious. When all these parameters are fulfilled by any event it will be tagged as ddos. For phishing, the dest_port and the src_ip are used for categorising the event. Moreover, any event or log entry which doesn't meet the requirements set in the above parameters is tagged as 'Other'. The approach used here for tagging the events try to replicate the real-world scenario, marking precise detections in a controlled environment. The figures 76 and 77 state both the tagging logics used in the training.

```python
def tag_phishing_logs(df):
    # Define conditions for identifying phishing logs
    phishing_conditions = (
        (df['event_type'] == 'alert') &
        ((df['dest_port'] == 80) | (df['dest_port'] == 443)) &
        (df['http_user_agent'].str.contains("curl|wget", case=False, na=False)) &
        (df['http_content_type'].str.contains("text/html", case=False, na=False)) &
        (df['http_method'] == 'GET')
    )

    # Apply the conditions to create a 'label' column for phishing
    df['label'] = 'Normal'  # Default label
    df.loc[phishing_conditions, 'label'] = 'Phishing'  # Tag phishing logs

    return df
```

Figure 76

```python
def tag_event(row):
    # Tagging logic for phishing
    if row['dest_port'] in [80, 443, 8080] and row['src_ip'].startswith("172.31."):
        print(f"Tagging as phishing - src_ip: {row['src_ip']}, dest_port: {row['dest_port']}")
        return 'phishing'

    # Tagging logic for ddos (multiple requests to the same destination IP and port)
    elif row['dest_port'] in [53, 123, 80, 443] and row['event_type'] == 'flow' and row['src_ip'] != row['dest_ip']:
        print(f"Tagging as ddos - src_ip: {row['src_ip']}, dest_ip: {row['dest_ip']}, dest_port: {row['dest_port']}")
        return 'ddos'

    # Default tag
    else:
        return 'other'

# Apply the updated tagging function
df['tag'] = df.apply(tag_event, axis=1)

# Display the updated DataFrame to verify the tags
print(df[['@timestamp', 'hostname', 'event_type', 'src_ip', 'dest_ip', 'src_port', 'dest_port', 'tag']].head(20))
```

Figure 77

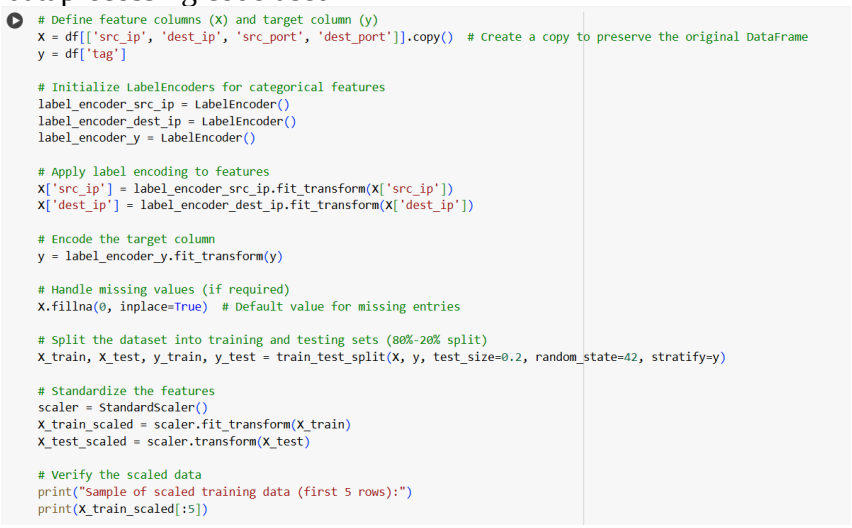
Output for the above in figure 78-

	dest_ip	src_port	dest_port	tag
0	3.228.185.88	39998.0	5044.0	other
1	169.254.169.123	53558.0	123.0	ddos
2	169.254.169.123	42588.0	123.0	ddos
3	172.31.31.86	65410.0	4789.0	other
4	162.252.172.49	55007.0	123.0	ddos
5	169.254.169.123	35290.0	123.0	ddos
6	169.254.169.123	58278.0	123.0	ddos
7	169.254.169.123	34378.0	123.0	ddos
8	169.254.169.123	50087.0	123.0	ddos
9	54.90.191.9	36828.0	123.0	ddos
10	169.254.169.123	48420.0	123.0	ddos
11	169.254.169.123	50159.0	123.0	ddos
12	169.254.169.123	35930.0	123.0	ddos
13	54.90.191.9	41027.0	123.0	ddos
14	44.201.148.133	51066.0	123.0	ddos
15	169.254.169.123	47406.0	123.0	ddos
16	169.254.169.123	40250.0	123.0	ddos
17	54.81.231.214	53790.0	80.0	phishing
18	None	NaN	NaN	other
19	172.31.31.45	46097.0	22.0	other

Figure 78

- Dataprepocessing: In this step, the dataset was cleaned for feeding the ML model. Standardization was applied to the numerical feature columns as it ensures that all the features in the dataset contribute equally in process of training the model. This was done after encoding the categorical data. So initially, features such as src_ip, dest_ip, src_port and dest_port were encoded into the numerical format using the LabelEncoder. It was used to convert the catgorical data such as the IP addresses into integers. After this, the numerical values were standardized to have a mean of 0 and a standard deviation of 1. This process standardised the feature distribution in a way that was more appropriate for the machine learning algorithms such as Random Forest and XGBoost that require features to be standardised for the model to work effectively in some cases. After all this the transformed data was split into training and testing sets. In this case, the training dataset confirms that the model is trained or learning appropriately and the testing data set

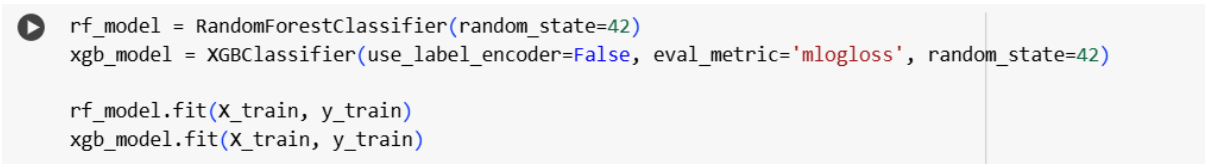
determines the ability of the model in delivering its output on new data. The figure 79 shows the dataprecossing code used-



```
# Define feature columns (X) and target column (y)
X = df[['src_ip', 'dest_ip', 'src_port', 'dest_port']].copy()  # Create a copy to preserve the original DataFrame
y = df['tag']

# Initialize LabelEncoders for categorical features
label_encoder_src_ip = LabelEncoder()
label_encoder_dest_ip = LabelEncoder()
label_encoder_y = LabelEncoder()

# Apply label encoding to features
X['src_ip'] = label_encoder_src_ip.fit_transform(X['src_ip'])
X['dest_ip'] = label_encoder_dest_ip.fit_transform(X['dest_ip'])

# Encode the target column
y = label_encoder_y.fit_transform(y)

# Handle missing values (if required)
X.fillna(0, inplace=True)  # Default value for missing entries

# Split the dataset into training and testing sets (80%-20% split)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Verify the scaled data
print("Sample of scaled training data (first 5 rows):")
print(X_train_scaled[:5])
```

Figure 79

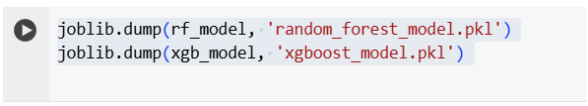
- Training the models: Followed by the preprocessing step, comes the training phase which involves two ensemble learning models- Random Forest and XGBoost. The RandomForestClassifier was initialized with a fixed random_state. This helps in making sure there is results obtained are consistent. While this process combines multiple decision trees in order to reduce overfitting and improve the overall prediction accuracy the other model which is XGBoost uses XGBClassifier was configured with parameters like use_label_encoder=False for avoiding deprecation warnings and eval_metric=’mlogloss’ for optimizng the multi-class log loss during the training. The random_state was also set for the XGBoost so that it maintains the consistency across different testcases. The entire training process was carried out on the fit method, where the preprocessed training dataset (X_train and y_train) was given as the input for the both the models. This step assisted the model to identify patterns within the data hence improving the performance of the model. The figure 80 depicts the code for the trained models.



```
rf_model = RandomForestClassifier(random_state=42)
xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='mlogloss', random_state=42)

rf_model.fit(X_train, y_train)
xgb_model.fit(X_train, y_train)
```

Figure 80

- Saving the models: The models were stored in .pkl format through the joblib.dump function. These models were downloaded in the local machine so that it could be exported to the ML model instance wherein they can be used as the detection mechanism. The figure 81 shows the joblib code



```
joblib.dump(rf_model, 'random_forest_model.pkl')
joblib.dump(xgb_model, 'xgboost_model.pkl')
```

Figure 81

2. Realtime prediction in the ML model instance

Firstly, the trained ML models which were saved in the earlier section were loaded in the instance using the command-

scp -i "F:\ml models.pem" "F:\xgboost_model.pkl" "F:\random_forest_model.pkl" ubuntu@54.81.145.37:~

Through the '**ls**' command we can verify if the trained models are imported successfully. The figure 82 is the confirmation.

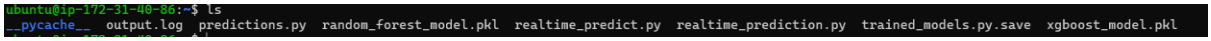


Figure 82

Before starting with the prediction's setup, it was important to install python on the instance to support the required libraries and scripts. The following procedure was used in doing the same

- Python was installed using the below commands (version 3.10.12 was used)

sudo apt install -y software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
sudo apt install -y python3.10
sudo apt install -y python3.10-venv python3.10-dev

- Installing the python package manager (pip) to handle the library dependencies through the command- **sudo apt install -y python3-pip**
- The libraries which were crucial in this setup such as Joblib, pandas, scikit-learn, xgboost, etc were installed using the pip command-
  pip install pandas scikit-learn xgboost elasticsearch Joblib numpy

Finally, after completing all the dependencies, a new file named as realtime_predict.py was generated through a '**nano**' command in the home directory. This file can also be viewed in the screenshot shared above. This script file contained the detailed steps for fetching the logs, preprocessing them, making predictions and sending them back to the elasticsearch in indexed format. The detailed explanation of the file is stated below along with the code snapshots

- Importing the necessary libraries:

Table 2

Libraries	Working
from elasticsearch import Elasticsearch	Connection with the elasticsearch
from joblib import load	Load the pre-trained models
Json	Handling JSON formatted logs
pandas	Preprocessing the log data
Numpy	Handling numerical operations and missing values
socket	Handling IP related conversions
from elasticsearch.helpers import bulk	Optimizing the bulk operations
struct	Performing binary conversions
from datetime import datetime	Comparing the timestamps between the processed logs

import time	Putting sleep time or trial attempts

The figure 83 depicts the libraries imported in the script

```
from elasticsearch import Elasticsearch
from joblib import load
import json
import pandas as pd
import numpy as np
import socket
from elasticsearch.helpers import bulk
import struct
from datetime import datetime
import time  # For real-time loop delay
```

Figure 83

- Loading the trained models: Here the trained models are loaded in to the memory which will predict the attacks. The figure 84 shows the code for the same

```
# Load models
rf_model = load('/home/ubuntu/random_forest_model.pkl')
xgb_model = load('/home/ubuntu/xgboost_model.pkl')
```

Figure 84

- Connecting to the Elasticsearch: The script establishes a connection with the elasticsearch a vital component of the SIEM to fetch the raw logs. All the necessary credentials and SIEM IP are provided for the same, this is depicted in the figure 85-

```
# Connect to Elasticsearch
es = Elasticsearch([{"host": "3.228.185.88", "port": 9200, "scheme": "http"}],
                   basic_auth=("elastic", "walnut"))
```

Figure 85

- Converting IP address into numercial format: Here the IP addresses are converted into the numerical format using the ip_to_int fucntion. If an Ip is missing or has an invalid input np.nan is returned. While socket.inet_aton() function converts an IP address into the binary format which then is converted to integer through struct.unpack("!I", …), this is depicted in the figure 86-

```
# Convert IP address to a numerical format
def ip_to_int(ip):
    if ip is None:
        return np.nan  # Return NaN if IP is None
    try:
        return struct.unpack("!I", socket.inet_aton(ip))[0]
    except socket.error:
        return np.nan  # Return NaN if IP conversion fails
```

Figure 86

- Fetching logs: Once the connection with elasticsearch is established, the ml model instance has to fetch the recent logs which the suricata is indexing in the SIEM. The fetch_new_logs function is used here which queries the elasticsearch database for logs that are recent than the last_timestamp. This minimizes the redundancy of the system and makes it more dynamic by processing only the unprocessed logs. The figure 87 depicts the code for the same.

```
# Fetch only new logs using a timestamp filter
def fetch_new_logs(index_name, last_timestamp, size=9000):
    query = {
        "query": {
            "range": {
                "@timestamp": {
                    "gt": last_timestamp  # Fetch logs newer than the last processed timestamp
                }
            }
        },
        "size": size
    }
    response = es.search(index=index_name, body=query)
    return response['hits']['hits']
```

Figure 87

- Preprocessing: In this section of the script, the raw logs are cleaned while the relevant fields are extracted for predictions. IP addresses are also converted to numerical formats here for making sure that they are compatible with the models. Relevant fields such as src_ip, dest_ip, src_port, dest_port and event_type were extracted. NaN was assigned for any missing or invalid data, while the ip_to_int function handles the conversion of IP into numerical format even through this conversion process has taken place before it is crucial to be repeated as it maintains the consistency making sure that the potential variations in the raw data are handled successfully and not resulting in the pipeline crash during the live data flow. The figure 88 is the code snippet for the same.

```
# Preprocessing function with IP conversion
def preprocess_logs(log_data):
    extracted_data = [log['_source'] for log in log_data]
    df = pd.DataFrame(extracted_data)

    def extract_fields(row):
        try:
            event = json.loads(row['event']['original'])
            row['src_ip'] = event.get('src_ip')
            row['dest_ip'] = event.get('dest_ip')
            row['src_port'] = event.get('src_port')
            row['dest_port'] = event.get('dest_port')
            row['event_type'] = event.get('event_type')
        except (KeyError, json.JSONDecodeError, TypeError):
            row['src_ip'] = np.nan
            row['dest_ip'] = np.nan
            row['src_port'] = np.nan
            row['dest_port'] = np.nan
            row['event_type'] = np.nan
        return row

    df = df.apply(extract_fields, axis=1)
    df['src_ip'] = df['src_ip'].apply(ip_to_int)
    df['dest_ip'] = df['dest_ip'].apply(ip_to_int)

    processed_logs = df.dropna(subset=["src_ip", "dest_ip", "src_port", "dest_port", "event_type"])
    return processed_logs
```

Figure 88

- Making predictions: This is an important where the features from the pre-processed functions are passed on to the models for making the predictions. The predict_proba function is used to generate the probability for each log while the combined_scores gives the classification result from both the models. If the combines score is greater than 0.7 it is termed as phishing (due to the high confidence in malicious behaviour), if the score is less than 0.3 it is tagged as DDoS (due to the low confidence mostly representing volumetric attack), while the logs with a score between them are classified as Benign. These thresholds and the entire code for the predictions is mentioned in the figure 89-

```
# Define the prediction function
def make_prediction(logs):
    features = logs[["src_ip", "dest_ip", "src_port", "dest_port"]]
    features = features.fillna(0)

    rf_predictions = rf_model.predict_proba(features)[:, 1]  # Probability score for positive class
    xgb_predictions = xgb_model.predict_proba(features)[:, 1]  # Probability score for positive class

    combined_scores = (rf_predictions + xgb_predictions) / 2

    final_classification = []
    for scores in combined_scores:
        if scores > 0.7:
            final_classification.append("Phishing")
        elif scores < 0.3:
            final_classification.append("DDoS")
        else:
            final_classification.append("Benign")

    return final_classification
```

Figure 89

- Saving the predictions: Once the models determine the scores and the predictions are made, this data is saved and sent back to the elasticsearch with a dynamic index name which separates the logs as per the dates. The predictions are attached under a prediction field in the original logs. A dynamic index name ml-predictions-YYYY.MM.DD is used for structuring the logs for each day on the SIEM. Figure 90 shows the code snippet for the same-

```
# Generate dynamic index name based on date
def get_index_name(base_name="ml-predictions"):
    current_date = datetime.now().strftime("%Y.%m.%d")  # Format: YYYY.MM.DD
    return f"{base_name}-{current_date}"

# Save predictions in JSON and send to Elasticsearch
def save_predictions_to_elasticsearch(log_data, classifications):
    output_data = []
    index_name = get_index_name()  # Generate index name dynamically
    for log, classification in zip(log_data, classifications):
        log["_source"]["prediction"] = classification
        output_data.append({
            "_index": index_name,  # Use dynamic date-based index name
            "_source": log["_source"]
        })

    bulk(es, output_data)
    print(f"Predictions saved to Elasticsearch under index '{index_name}'.")
```

Figure 90

- Real-time Execution: This script is designed to fetch the logs, process it, make the predictions and save it back continuously. Thus, it starts with the timestamp **now-1m** and processes the logs in **5-second** interval time. Even if the script faces error, it waits for **30 seconds** before reattempting. The figure 91 shows the code for the same-

```
# Main function for real-time predictions
if __name__ == "__main__":
    last_timestamp = "now-1m"  # Start with logs from the last minute
    while True:
        try:
            logs = fetch_new_logs("suricata-logs-*", last_timestamp, size=9000)
            if logs:
                print(f"Fetched {len(logs)} new logs.")
                last_timestamp = logs[-1]['_source']['@timestamp']  # Update last processed timestamp

                processed_logs = preprocess_logs(logs)
                if not processed_logs.empty:
                    classifications = make_prediction(processed_logs)
                    save_predictions_to_elasticsearch(logs, classifications)
                    print("Predictions processed and sent to Elasticsearch.")
                else:
                    print("No logs to process after filtering.")
            else:
                print("No new logs found.")

            time.sleep(5)  # Wait for 5 seconds before fetching new logs
        except Exception as e:
            print(f"Error: {e}")
            time.sleep(30)  # Wait for 30 seconds on error
```

Figure 91

So as the realtime_predict.py script is ready for functioning it is executed using the '**nohup**' command which ensures that the process for ml predictions works continuously without any interruption, until the user decides to terminate it. The figure 92 shows the command and its input-

```
ubuntu@ip-172-31-40-86:~$ nohup python3 realtime_predict.py > output.log 2>&1 &
[1] 955
```

Figure 92

9 Setting up the Kibana alerts and dashboards

This section will explain about the configurations done in the GUI end of the SIEM that is Kibana.

Step 1: Creating Index patterns

Index patterns are very crucial for Kibana to recognize the logs stored in the elasticsearch, however it already has the data from the elasticsearch, the index patterns need to be explicilty defined on kibana so that it can categorise the events. Two main index patterns which need to be defined here are suricata-logs-* and ml-predictions-*. The following steps were used for creating these indices. (The procedure remains the same for both of them except for their names).

- Navigating to the Index pattern section through Stack Management -> Index Patterns.
- Clickin on "create index pattern".
- In the name bracket we need to input the same name which is given in the elasticsearch index section (in this case the names are suricata-logs-* and ml-predictions-*). This makes sure that all the with names starting from the suricata-logs- and ml-predictions- are indexed in each of them respectively.
- Selecting the timestamp field (@timestamp) from the drop down menu.
- Click on "Create index pattern" button now for the index to be created.
- The figure 93 is a sample showcasing the creation of suricata-logs-* index

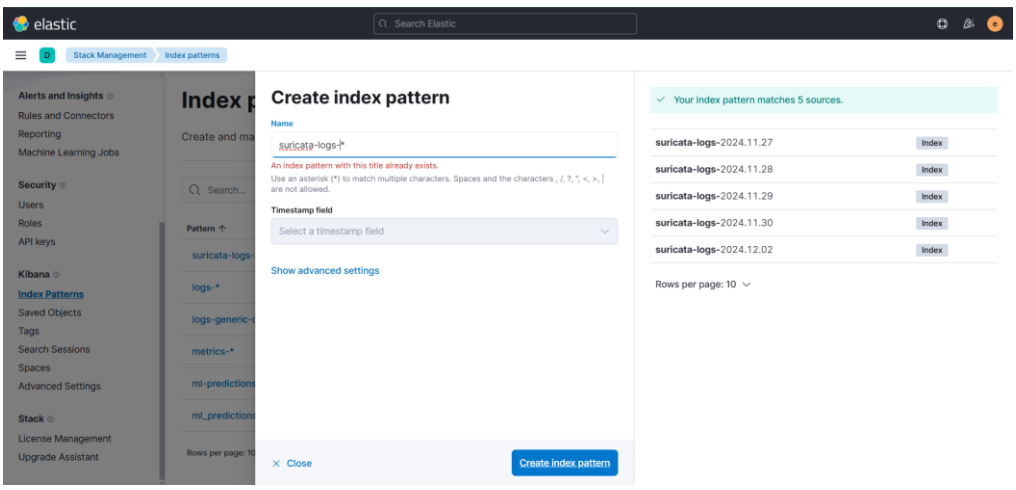


Figure 93

After the indexes are created, they can be checked in the '**Index pattern**' tab, as seen in the figure 94-

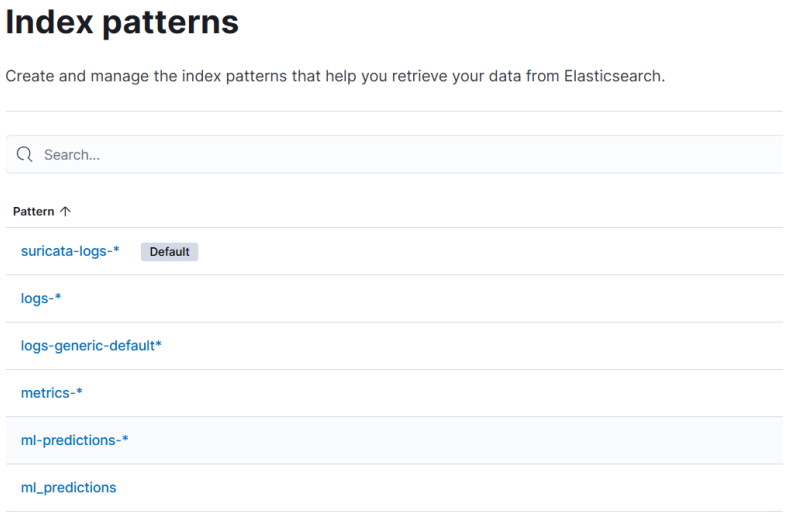


Figure 94

Step 2: Creating Rules for generating alerts

For creating the rules, it is crucial for defining the connectors first. As the connectors specify exactly where the alerts are to be sent. Connectors can be of different types which defines how the alerts will be processed whether they will be sent through email or slack or write them in a specific index. Here as we are using the open-source license for ELK we only have the access to Index connector. For this setup we need 2 customised rules mainly for DDoS and phishing, thus I have created 2 connectors for the same. The process for creating the connectors again remain the same except for their names-

- Navigate to Connectors section through Stack Management -> Rules and Connectors-> Connectors
- Click on "**Create connector**" button.
- Choose Index as the connector type
- For configuring the connector, provide a distinctive name for the connector (in this case DDoS and Phishing is given). After that specify the index name where the alerts would be stored (in this case ddos-alerts and phishing alerts name is provided).
- Toggle the Refresh index to ON so that the index is refreshed when the alerts are written.
- Finally click the "**Save & test**" button to check if the connector is properly workings. Once saved the connector will be appearing on the connectors window in Rules and Connectors section
- A sample creation of Connector can be seen through the figure 95.

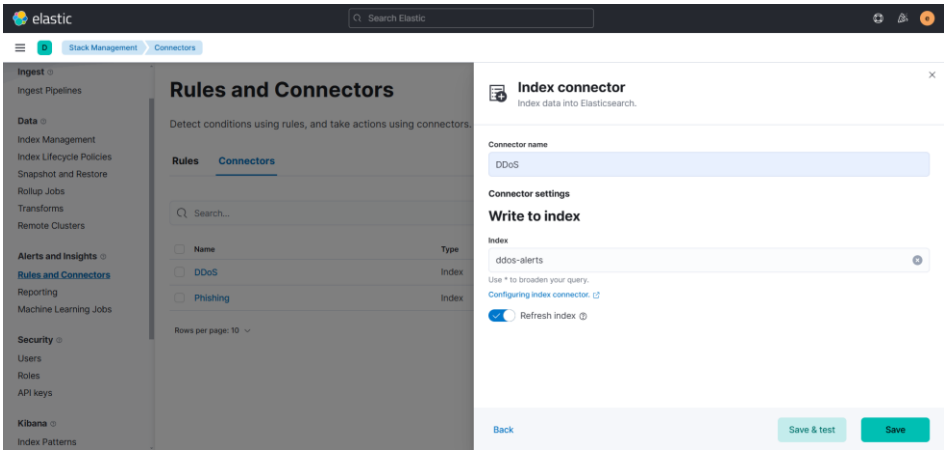


Figure 95

Now we can move on to the Rule creation part. Rules are basically the logic behind the generation of alerts, once a certain log meets the entry requirements set in a particular rule an alert is triggered against it. For this setup, I created 2 custom rules which will be detailed in the below section, but before the custom rules the rules provided by the ELK were imported in our system to enhance the rules list and criteria's majorly to understand the structure of a well-defined rule which could possibly help in setting new rules. For this, firstly navigate to **Kibana -> Security -> Alerts -> Rules** and by clicking on '**Manage Rules**' button, there is an option for '**Load Prebuilt Detection Rules**' popping up and thus the pre-built rules are integrated in the system. Now that we are already in the rules section the following process can be adapted for creation of custom rules. (both the rules created will be explained in the same setting but with different parameters as per their requirements).

- Once in the Rules section, click on '**Create Rule**'.
- Name the Rules. (in this case DDoS and Phishing).
- Select the rule type, here I have selected '**Custom Query**'. Through this we can enter a customised query which will trigger the rules. The query was extensively tested on the '**Discover**' section of the kibana to understand the logs structure and what exact fields can possibly trigger the attacks.
- After this add the index pattern '**suricata-logs-***' is the index pattern which contains all the network traffic/logs generated by IDS thus inputting this in the index pattern section makes sure that the rule is fetching the results from this section.
- After this, we need to input the custom query in a KQL (Kibana query language) format. The queries used both for Ddos and Phishing are listed below-

DDoS: **suricata.eve.event_type:"alert" AND suricata.eve.alert.signature:"Possible SYN Flood" AND suricata.eve.alert.category:"Attempted Denial of Service"**

Phishing: **suricata.eve.event_type:"alert" AND suricata.eve.http.url:"/phishing-page.html" AND suricata.eve.http.hostname:"attackingmachine.ddns.net" AND suricata.eve.alert.signature:"ET INFO DYNAMIC_DNS HTTP Request to a *.ddns .net Domain"**

The figure 96 acts an edit page while creating the DDoS Rule.

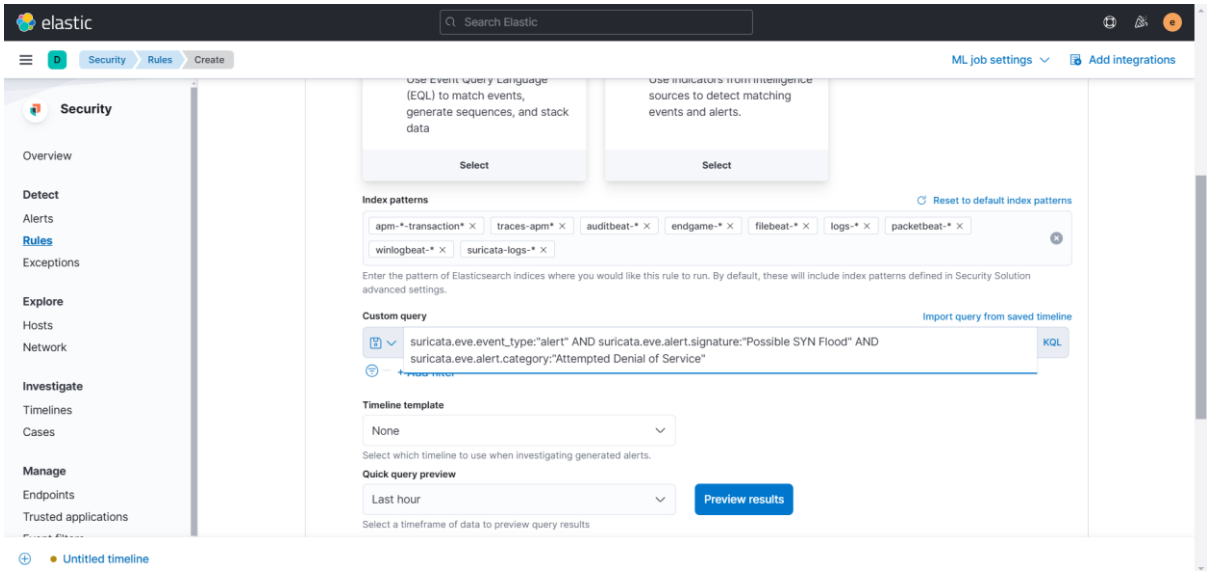


Figure 96

- After this we need to input the brief of the rule, such as setting the **Severity**, **Risk Score** and **Tags** (if any).
- After this set the alert frequency specifying how often the query should run (here I have specified it every 1m, this can be changed as per the requirements).
- Then select the index connector which we configured earlier (ddos-alerts and phishing alerts for respective rules) so that the alerts can be saved in the indices.
- Using the '**Preview Results**' button the query or the rule can be tested on the available logs whether it is fetching the results.
- Lastly save the rules and move to the '**Alerts**' tab to check if the rules are working properly. The figure 97 displays the alerts generated through the rules set.

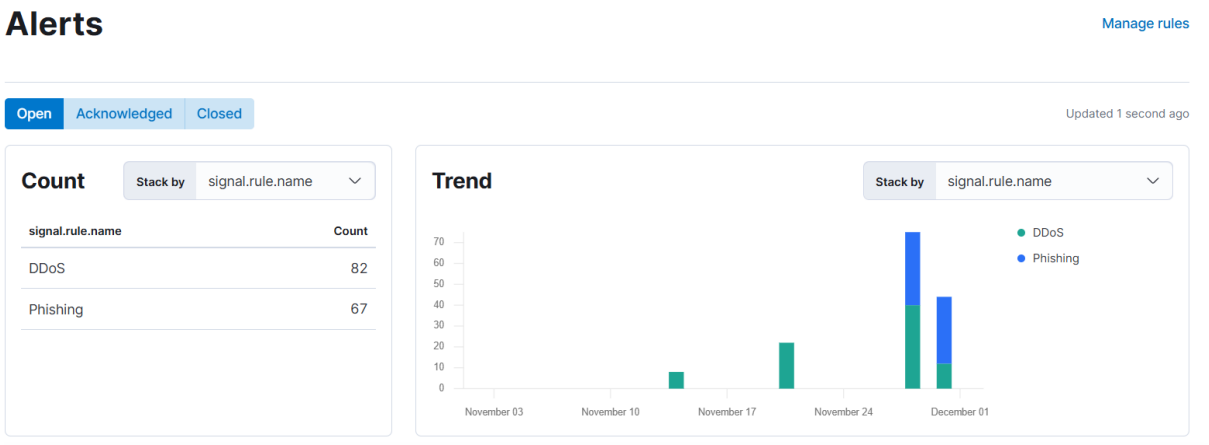


Figure 97

Step 3: Creating dashboards for visualising the data on SIEM

This is a very important step in setting up the SIEM server as it enables the end user to see the data in order to protect the IT infrastructure. We can create as many as dashboards as per the requirements with different visualisations but for this setup majorly 3 dashboards can be created depicting the ml predictions, rule-based predictions and lastly a combined graph for both the predictions. The detailed steps for these 3 dahsboards are stated below-

1. Dashboard 1: Machine Learning based Predictions

- Go to '**Dashboard**' tab on the Kibana and then click on '**Create Dashboard**'.
- Go to the 'Visualize Library' on the dashboard.
- Here I have created a pie-chart in order to show the predictions, so just select the '**Pie chart**' option after clicking the button '**Create Visualization**'.
- Selecting the accurate index pattern is important here as it would specify the service to fetch the data from there. The ML predictions are loaded in the '**ml-predictions-***'. Thus we will select this.
- Next, in the buckets section we need to split the slices by selecting the **prediction.keyword** field. This will visualise the data as per the DDoS, Phishing and Benign as classified by the ML models.
- We can check the view of the visualisation and adjust any display of the labels for more clarity.
- Finally the visualisation can be saved and named as Pie-chart for ML predictions.
- Thus the visualisation is stored in the dashboard which can also be saved once the changes are done. The figure 98 depicts the pie-chart created on kibana.

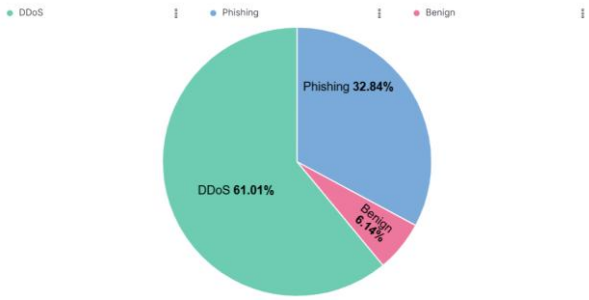


Figure 98

2. Dashboard 2: Rule-based Alerts visualisation

- Go to '**Dashboard**' tab on the Kibana and then click on '**Create Dashboard**'.
- Go to the 'Visualize Library' on the dashboard.
- Click the 'Create visualisation' button and sleect the required type of the graph. Here I have selected 'Bar vertical stacked'.
- Choose the index pattern which is '**suricata-logs-***' in this case.
- In the horizontal access section, select the field **@timestamp**, on the vertical axis section, '**Count of records**' can be selected.
- Finally to classify the data, in the '**Break down by**' section select the '**suricata.eve.alert.signature.keyword**'.

The figure 99 is the graphical representation of the steps explained above-

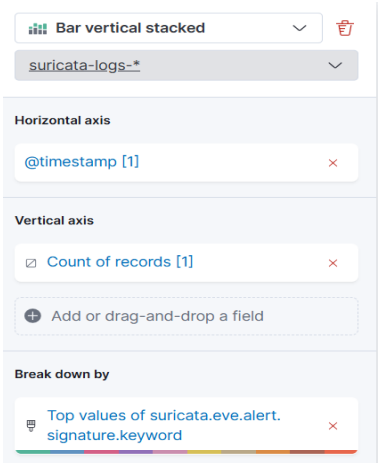


Figure 99

- Finally the graph can be saved by clicking 'Save and return' and thus the visualisation is created on the dashboard.

3. Dashboard 3: Combined ML and Rule based predictions.

In this dashboard a '**Bar vertical stacked**' graph is used to show the combination of both the detection types. By repeating the same procedures from the Dashboard 2 creation, the graph can be created which includes the rule-based detections and for adding the ML predictions in it, we need to select the 'Add layer' at the bottom right corner of the page. This can be seen through the figure 100.

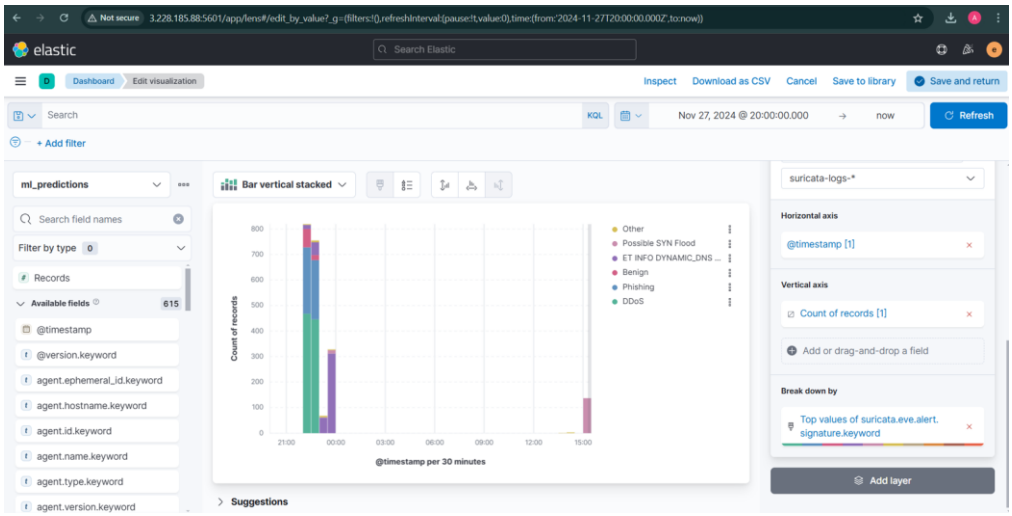


Figure 100

Once the additional layer is added, select the '**ml-predictions-***' index pattern, choose **@timestamp** field in the horizontal axis, while '**Count of records**' in the vertical axis and in the 'Break down by' section select the **prediction.keyword** field. This steps can be represented through the figure 101.

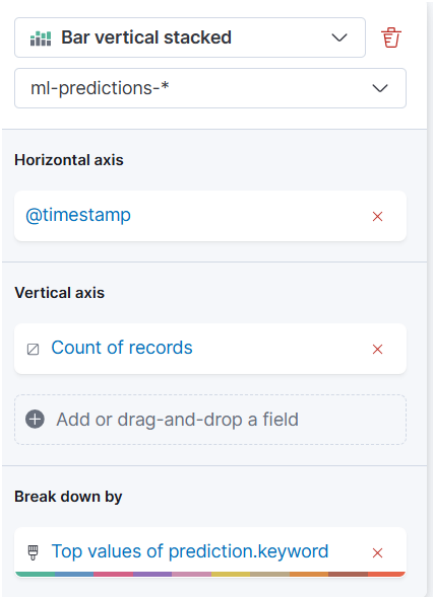


Figure 101

Thus the final graph including both the predictions can be viewed in the figure 102.

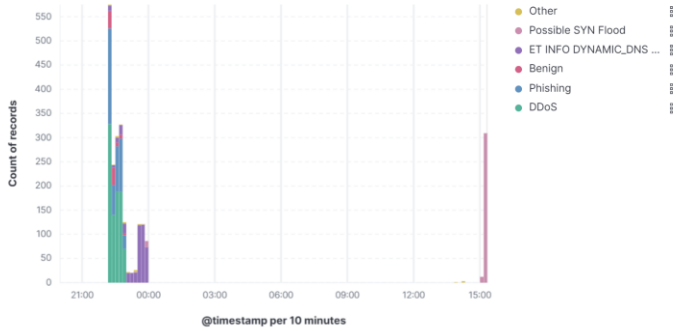


Figure 102

This was the entire configuration manual for replicating the entire real-time security detection setup.

References

Argonzo, R. (2019) *Emerging Threats PRO/OPEN Ruleset for Suricata 7.0.3 Now Available*. Available at: https://forum.suricata.io/t/emerging-threats-pro-open-ruleset-for-suricata-7-0-3-now-available/4714 [Accessed 14 October 2024].

Elastic (2024) *Elasticsearch Guide*. Available at: https://www.elastic.co/guide/en/elasticsearch/reference/7.17/index.html [Accessed 17 October 2024].

Elastic (2024) *Filebeat quick start: installation and configuration*. Available at: https://www.elastic.co/guide/en/beats/filebeat/7.17/filebeat-installation-configuration.html [Accessed 17 October 2024].

Elastic (2024) *Kibana Guide*. Available at: https://www.elastic.co/guide/en/kibana/7.17/index.html [Accessed 17 October 2024].

Elastic (2024) *Logstash Reference*. Available at: https://www.elastic.co/guide/en/logstash/7.17/index.html [Accessed 17 October 2024].

Emerging Threats (2024) *ET OPEN Ruleset Download Instructions*. Available at: https://rules.emergingthreats.net/OPEN_download_instructions.html [Accessed 14 October 2024].

GitHub (2024) Installing Suricata IDS on Ubuntu Server. Available at: https://github.com/0xrajneesh/Suricata-IDS-Home-Lab/blob/main/installing-suricata.md [Accessed 10 October 2024].