Email Phishing Detection, User Credential Management, and Mitigation System

Overview

This comprehensive system is designed to enhance email security and manage user credentials with encryption. The core components include an Email Phishing Detection and Mitigation module, which identifies and mitigates phishing attempts by analyzing URLs and email metadata, and a secure credential management system utilizing MongoDB and RSA encryption. The system leverages machine learning models, domain reputation checks, and email authentication protocols to provide robust protection against phishing threats while securely storing and managing user credentials.

System Components

1. **Email Phishing Detection and Mitigation**
    - **Domain Reputation Analysis**: Evaluates domains based on age, rank, popular keywords, and TLD trustworthiness.
    - **Email Authentication**: Verifies SPF, DKIM, and DMARC records for email sender authentication.
    - **Blacklist and Whitelist Management**: Manages and updates lists of blacklisted and whitelisted domains and IPs.
    - **URL Feature Extraction**: Extracts features from URLs for classification by machine learning models.
    - **Phishing Score Calculation**: Aggregates checks to calculate a phishing score for URLs, classifying them as legitimate or phishing.
    - **Automated Phishing Mitigation**: Automatically moves phishing emails to the spam folder and notifies the user.
2. **User Credential Management with MongoDB and RSA Encryption**
    - **MongoDB Integration**: Manages user credentials using a MongoDB database.

- **RSA Encryption**: Encrypts and decrypts credentials stored in the database using RSA keys.
- **Real-Time Monitoring:** Provides real-time email monitoring and phishing detection using encrypted credentials for secure access.
- **Master Menu**: Centralized control for all functions related to credential management and phishing detection.

Code Explaination

1. blacklist_pulldown.py

```
import requests
import pandas as pd
import os

def domain_db_update():
    filename='processed_domains.pkl'
    if os.path.exists(filename):
        os.remove(filename)
        print("Deleting Previous Domain Blacklist")
```

```
  urls = [
    "https://gitlab.com/hagezi/mirror/-/raw/main/dns-blocklists/wildcard/tif-onlydomains.txt",
    "https://raw.githubusercontent.com/Spam404/lists/master/main-blacklist.txt",
    "https://malware-filter.gitlab.io/malware-filter/phishing-filter-domains.txt",
    "https://raw.githubusercontent.com/matomo-org/referrer-spam-
blacklist/master/spammers.txt",
    "https://phishing.army/download/phishing_army_blocklist_extended.txt",
    "https://big.oisd.nl/domainswild2"
  ]

  processed_domains = set()

  for url in urls:
    print(f"Fetching Domains from {url}: ")
    response = requests.get(url)

    if response.status_code == 200:
      lines = response.text.splitlines()
      for line in lines:
        if '#' in line:
          continue
        processed_domains.add(line)
    else:
      print(f"Failed to download file from {url}. HTTP Status Code: {response.status_code}")

  df = pd.DataFrame(sorted(processed_domains), columns=["domain"])
  output_filename = "processed_domains.pkl"
  df.to_pickle(output_filename)

  print(f"Processed domains saved to {output_filename}")

def ip_db_update():

  filename='processed_ip.pkl'
  if os.path.exists(filename):
    os.remove(filename)
    print("Deleting Previous IP Blacklist")

  urls = [
    "https://raw.githubusercontent.com/stamparm/ipsum/master/ipsum.txt",
    "https://sslbl.abuse.ch/blacklist/sslipblacklist_aggressive.txt",
    "https://sslbl.abuse.ch/blacklist/sslipblacklist.txt",
    "https://lists.blocklist.de/lists/all.txt",
    "http://www.talosintelligence.com/documents/ip-blacklist",
    "https://lists.blocklist.de/lists/mail.txt",
    "https://snort.org/downloads/ip-block-list"
  ]
```

```
processed_lines = []

for url in urls:
  print(f"Fetching IP's from {url}:")
  response = requests.get(url)
  if response.status_code == 200:
    lines = response.text.splitlines()
    for line in lines:
      if '#' in line:
        continue
      if url == "https://raw.githubusercontent.com/stamparm/ipsum/master/ipsum.txt":
        first_field = line.split('\t')[0]
        processed_lines.append(first_field)
      else:
        processed_lines.append(line)
  else:
    print(f"Failed to download file from {url}. HTTP Status Code: {response.status_code}")

df = pd.DataFrame(sorted(processed_lines), columns=["ip"])
output_filename = "processed_ip.pkl"
df.to_pickle(output_filename)

print(f"Processed IPs saved to {output_filename}")
```

Overview

The `blacklist_pulldown.py` script is designed to update and maintain local blacklists of domains and IP addresses that are associated with phishing, spam, or other malicious activities. The script fetches the latest data from various online sources, processes the data, and stores it in local files for later use in phishing detection or security applications.

Function: domain_db_update()

Purpose:

This function is responsible for downloading and processing blacklisted domains from several online sources. It then saves the processed list to a local file.

Steps:

1. **Delete Existing File**:

- The function first checks if the processed_domains.pkl file already exists. If it does, the file is deleted to ensure that an updated version is created.

```
filename='processed_domains.pkl' if
os.path.exists(filename):
    os.remove(filename)
    print("Deleting Previous Domain Blacklist")
```

2. URLs for Domain Blacklists:

- A list of URLs is provided, each pointing to a text file containing a list of blacklisted domains.

```
urls = [
    "https://gitlab.com/hagezi/mirror/-/raw/main/dns-
blocklists/wildcard/tif-onlydomains.txt",
    "https://raw.githubusercontent.com/Spam404/lists/master/main-
blacklist.txt",
    ...
]
```

3. **Fetch and Process Domains:**
    - The function iterates over each URL, fetching the content and processing it line by line. Comments (lines starting with #) are skipped, and each valid domain is added to a processed_domains set.

```
for url in urls:
    response = requests.get(url)
    if response.status_code == 200:
        lines = response.text.splitlines() for
        line in lines:
            if '#' in line:
                continue
            processed_domains.add(line)
    else:
        print(f"Failed to download file from {url}. HTTP Status Code:
{response.status_code}")
```

4. **Save Processed Domains:**

- The processed domains are sorted and saved as a Pandas DataFrame to a pickle file (`processed_domains.pkl`).

```
df = pd.DataFrame(sorted(processed_domains), columns=["domain"])
df.to_pickle("processed_domains.pkl")
print(f"Processed domains saved to processed_domains.pkl")
```

Function: ip_db_update()

Purpose:

This function performs a similar task as `domain_db_update()` but for blacklisted IP addresses. It fetches and processes blacklisted IPs from various online sources and saves the results to a local file.

Steps:

1. Delete Existing File:

- As with the domain update function, the existing `processed_ip.pkl` file is deleted if it exists.

```
filename='processed_ip.pkl' if
os.path.exists(filename):
    os.remove(filename)
    print("Deleting Previous IP Blacklist")
```

2. URLs for IP Blacklists:

- A list of URLs is provided, each pointing to a text file containing a list of blacklisted IP addresses.

```
urls = [ "https://raw.githubusercontent.com/stamparm/ipsum/master/ipsum.txt",
    "https://sslbl.abuse.ch/blacklist/sslipblacklist_aggressive.txt",
    ...
]
```

3. Fetch and Process IPs:

- The function iterates over each URL, fetching the content and processing it line by line. Comments (lines starting with #) are skipped. Special processing is done for the ipsum.txt file, where only the first field (IP address) is kept.

```
for url in urls:
    response = requests.get(url)
    if response.status_code == 200:
        lines = response.text.splitlines() for
        line in lines:
            if '#' in line:
                continue
            if url ==
"https://raw.githubusercontent.com/stamparm/ipsum/master/ipsum.txt":
                first_field = line.split('¥t')[0]
                processed_lines.append(first_field)
            else:
                processed_lines.append(line)
    else:
        print(f"Failed to download file from {url}. HTTP Status Code:
{response.status_code}")
```

4. Save Processed IPs:

- The processed IP addresses are sorted and saved as a Pandas DataFrame to a pickle file (processed_ip.pkl).

```
df = pd.DataFrame(sorted(processed_lines), columns=["ip"])
df.to_pickle("processed_ip.pkl")
print(f"Processed IPs saved to processed_ip.pkl")
```

Summary

This script is crucial for keeping local blacklists of domains and IPs up to date. These blacklists can then be used in phishing detection systems, email filtering, or other security applications to identify and block malicious activity. The use of online sources ensures that the blacklist is current and comprehensive, covering the latest threats.

2. mongodb_rsa.py

```
import rsa
from pymongo import MongoClient
import os
import logging

# Set up logging
logger = logging.getLogger(_name_)
logger.setLevel(logging.INFO)

# Load the public key from file
with open("public_key.pem", "rb") as f:
   public_key_pem = f.read()
public_key = rsa.PublicKey.load_pkcs1(public_key_pem)

# Load the private key from file
with open("private_key.pem", "rb") as f:
   private_key_pem = f.read()
private_key  =  rsa.PrivateKey.load_pkcs1(private_key_pem)

def store_credentials():
   print("User Registration")
   email = input("Enter the email address: ")
   app_pass = input("Enter the app-generated password: ")
   alert_server_email = input("Enter the Alert Server email: ")
   alert_server_app_pass = input("Enter the Alert Server app pass: ")

   # Connect to MongoDB
   client = MongoClient('<mongodb connection string>')
   db = client['server-rsa']
   collection = db['credentials']

   # Check if the email is already registered
   if collection.find_one({'email': email}):
      logger.warning(f"Email {email} already registered.")
      return

   # Encrypt the passwords using the RSA public key
   encrypted_app_pass = rsa.encrypt(app_pass.encode('utf-8'), public_key)
   encrypted_alert_server_app_pass = rsa.encrypt(alert_server_app_pass.encode('utf-8'),
public_key)
```

```
    credentials = {
        'email': email,
        'app_pass': encrypted_app_pass.hex(),  # Convert bytes to hex string
        'alert_server_email': alert_server_email,
        'alert_server_app_pass': encrypted_alert_server_app_pass.hex(), # Convert bytes to
hex string
        'public_key': public_key_pem.decode('utf-8')  # store the public key
    }

    result = collection.insert_one(credentials)
    logger.info(f'Registered user with id {email}')
    print("Login to your registered account")

def fetch_credentials():
    # Connect to MongoDB
    client = MongoClient('<mongodb connection string>')
    db = client['server-rsa']
    collection = db['credentials']

    # Fetch all credentials
    credentials = collection.find()

    # Create a list to store the credentials
    credential_list = []

    # Iterate over the credentials
    for credential in credentials:
        # Extract the email, app pass, alert server email, and alert server app pass
        email = credential['email']
        encrypted_app_pass = credential['app_pass']
        alert_server_email = credential['alert_server_email']
        encrypted_alert_server_app_pass = credential['alert_server_app_pass']

        # Decrypt the app pass and alert server app pass using the private key
        app_pass = rsa.decrypt(bytes.fromhex(encrypted_app_pass), private_key).decode('utf-
8')
        alert_server_app_pass =
rsa.decrypt(bytes.fromhex(encrypted_alert_server_app_pass),        private_key).decode('utf-8')

        # Add the credential to the list
        credential_list.append((email, app_pass, alert_server_email, alert_server_app_pass))

    return credential_list
```

```
def remove_user():
    email = input("Enter the email of the user to remove: ")

    # Connect to MongoDB
    client = MongoClient('<mongodb connection string>')
    db = client['server-rsa']
    collection = db['credentials']

    # Delete the user
    result = collection.delete_one({'email': email})
    if result.deleted_count > 0:
        logger.info(f"Successfully removed user: {email}")
    else:
        logger.warning(f"User {email} not found.")

def update_user():
    email = input("Enter the email of the user to update: ")

    print("Leave fields blank if you don't want to update them.")
    new_app_pass = input("Enter new app pass (optional): ")
    new_alert_server_email = input("Enter new alert server email (optional): ")
    new_alert_server_app_pass = input("Enter new alert server app pass (optional): ")

    # Connect to MongoDB
    client = MongoClient('<mongodb connection string>')
    db = client['server-rsa']
    collection = db['credentials']

    update_data = {}
    if new_app_pass:
        encrypted_new_app_pass = rsa.encrypt(new_app_pass.encode('utf-8'),
public_key).hex()
        update_data['app_pass'] = encrypted_new_app_pass
    if new_alert_server_email:
        update_data['alert_server_email'] = new_alert_server_email
    if new_alert_server_app_pass:
        encrypted_new_alert_server_app_pass =
rsa.encrypt(new_alert_server_app_pass.encode('utf-8'), public_key).hex()
        update_data['alert_server_app_pass'] = encrypted_new_alert_server_app_pass

    if update_data:
        result = collection.update_one({'email': email}, {'$set': update_data})
        if result.matched_count > 0:
            logger.info(f"Successfully updated user: {email}")
        else:
            logger.warning(f"User {email} not found.")
    else:
        logger.warning("No update data provided.")
```

Overview

This Python script is designed to securely manage user credentials (such as email addresses and passwords) by leveraging RSA encryption and storing the data in a MongoDB database. The script supports various operations, including:

1. **Storing new user credentials** securely.
2. **Fetching and decrypting stored credentials** for review.
3. **Updating existing user credentials.**
4. **Removing users** from the database.

The script uses RSA public/private key pairs for encryption and decryption, ensuring that sensitive information like passwords remains secure even when stored in the database.

Key Components

1. RSA Encryption

- **Public Key (public_key.pem)**: Used for encrypting the passwords before storing them in the database.
- **Private Key (private_key.pem)**: Used for decrypting the passwords when retrieving them from the database.

2. MongoDB Integration

- The script interacts with a MongoDB database to store and manage user credentials.
- Each user's credentials are stored as a document in a MongoDB collection.

3. Logging

- The script uses Python's `logging` module to log important operations, such as successful user registration, updates, or removals.

Functions Explained

1. store_credentials()

Purpose: Registers a new user and stores their credentials securely in the MongoDB database.

Steps:

- **User Input**:
  - Prompts the user to enter their email address, app-generated password, alert server email, and alert server app password.
- **MongoDB Connection**:
  - Establishes a connection to the MongoDB database using the connection string (replace `<mongodb connection string>` with your actual MongoDB URI).
  - Selects the database (`server-rsa`) and collection (`credentials`).
- **Email Duplication Check**:
  - Checks if the email is already registered by searching the MongoDB collection.
  - If the email exists, it logs a warning and exits.
- **Password Encryption**:
  - Encrypts the provided passwords using the RSA public key.
  - Converts the encrypted passwords into hexadecimal strings for storage.
- **Data Storage**:
  - Stores the encrypted passwords, along with the email and alert server email, in the MongoDB collection.
- **Logging**:
  - Logs the successful registration of the user and informs the user to log in with their registered account.

Example Usage:

```
store_credentials()
```

This will guide the user through the registration process and securely store their credentials.

2. fetch_credentials()

Purpose: Fetches and decrypts all stored user credentials from the MongoDB database.

Steps:

- **MongoDB Connection:**
  - Connects to the MongoDB database and selects the appropriate collection.
- **Data Retrieval:**
  - Fetches all documents (user credentials) from the collection.
- **Password Decryption:**
  - Decrypts each stored password using the RSA private key.
  - Converts the hexadecimal strings back into readable passwords.
- **Return Value:**
  - Returns a list of tuples containing the email address, decrypted passwords, and alert server email for each user.

Example Usage:

```
credentials = fetch_credentials()
for cred in credentials:
    print(f"Email: {cred[0]}, App Pass: {cred[1]}, Alert Server Email:
{cred[2]}, Alert Server App Pass: {cred[3]}")
```

This will display all stored user credentials in a readable format.

3. remove_user()

Purpose: Removes a user's credentials from the MongoDB database.

Steps:

- **User Input:**
  - Prompts the user to enter the email address of the user they want to remove.
- **MongoDB Connection:**
  - Connects to the MongoDB database and selects the appropriate collection.
- **Data Deletion:**
  - Attempts to delete the document associated with the provided email.
- **Logging:**

    - Logs whether the deletion was successful or if the user was not found in the database.

Example Usage:

```
remove_user()
```

This will prompt the user to enter the email address and will remove the corresponding user from the database if found.

4. update_user()

Purpose: Updates the credentials of an existing user in the MongoDB database.

Steps:

- **User Input**:
    - Prompts the user to enter the email address of the user they want to update.
    - Optionally, allows the user to enter new values for the app-generated password, alert server email, and alert server app password.
- **MongoDB Connection**:
    - Connects to the MongoDB database and selects the appropriate collection.
- **Password Encryption**:
    - If new passwords are provided, they are encrypted using the RSA public key.
- **Data Update**:
    - Updates the MongoDB document with the new encrypted passwords or other information provided.
- **Logging**:
    - Logs whether the update was successful or if the user was not found in the database.

Example Usage:

```
update_user()
```

This will prompt the user to update the credentials for a specific user.

Usage Instructions

Setup

5. **Install Required Libraries**:
    - Install the necessary Python packages:

```
pip install rsa pymongo
```

6. **Prepare RSA Keys**:
    - Place the RSA public key (`public_key.pem`) and private key (`private_key.pem`) in the same directory as the script.
    - These keys will be used to encrypt and decrypt the passwords.
7. **MongoDB Setup**:
    - Ensure you have a MongoDB instance running.
    - Replace the placeholder `<mongodb connection string>` in the script with your actual MongoDB connection string.

Running the Script

- **Store New User Credentials**: Run `store_credentials()` to add a new user to the database.
- **Fetch and View Stored Credentials**: Run `fetch_credentials()` to retrieve and decrypt all stored user credentials.
- **Update Existing User**: Run `update_user()` to update an existing user's credentials.
- **Remove a User**: Run `remove_user()` to delete a user from the database.

Security Considerations

- **Encryption**: RSA encryption ensures that sensitive data is stored securely in the database. The public key is used for encryption, while the private key is required for decryption.
- **Logging**: Logs are generated for key operations, making it easier to monitor and audit the activities of the script.
- **Database Security**: Ensure that your MongoDB instance is secured and that access is restricted to authorized users only.

3. RSA Keys for Encryption and Decryption

RSA keys are used to encrypt and decrypt sensitive data, ensuring secure communication and storage. In this context, the RSA keys are utilized to protect user credentials stored in a MongoDB database. Below is a detailed explanation of each key file:

1. Private Key (private_key.pem)

Format: PEM (Privacy-Enhanced Mail) format, which is a Base64 encoded representation of the key enclosed between `-----BEGIN RSA PRIVATE KEY-----` and `-----END RSA PRIVATE KEY-----`.

Purpose:

- **Decryption**: The private key is used to decrypt data that was encrypted with the corresponding public key. This ensures that only the owner of the private key can access the original data.

Contents:

- **Key Header**: Indicates the start of the RSA private key.
- **Key Body**: Contains the Base64 encoded private key data.
- **Key Footer**: Indicates the end of the RSA private key.

Example:

```
-----BEGIN RSA PRIVATE KEY-----
MIIEqQIBAAKCAQEArB6JB92u1m6GA69xIE764py46qzco8Ro3TqfFMBELaam2dzk
VuVvXmQVuPGZKbPYYwkhv5LApaxyJNAPglyAaMOCn9lBDjNJ9pqwoM7NP7SZzS9S
CnBBSU5hCJ5JrT2bMdJ1R6DMWUhUOV7D/EFFKVUqOYHIQGx6XaS4IK5p7tei7HgT
gL4cO42O3QDJ22HhxtvKMl3e3wz9jwIDz8748l3GpW4lVo1eCOyTEJ+S+f6LcrPn
BRDCfReo7MCXJlPT/G/XFbQVDfJYO/2oAXncfEhP6TSa++sIrU3q2GdJL4O8+95a
E+AoieZa5OIg57223QGZHYThfT7JG3W1mMOL3wIDAQABAoIBAE6trqof1fpDj5ND
sYy5TcolJgpwdnOOJvlqAKo1sXTEx2SzqtVmD+CwE5X1KrR7d6Dd3cV6ygFxsMPP
MCfBLRm2tZFoeGqLW3YVBESeBAjo/51h0jjfU5iO8XpFERzWGolNOTqq8H6np4iK
HODCfWEFg4s9rlJOw6DiHJLTj1PfUqz/5npvTmUbSCFgAgHSi7O1VQLX2Gz3to/W
776ZlWmAfudPprVxAo3XbTWK+B3ueJ+dxQhOm+2q6EdtOSiLE+IcXLJPrvGqWzRs
s+EM8iX4fIqXLoWSdVnMMgGuikfurOTbLLKjj/ES1cSk5t1DBv6lxStT6Yc22Mtj
```

```
zkySPtECgYkA1mzE8AXzVECaJOo6AVEnKNbDiG8zGTJwlKAt1hfMkYRwxtFoI3/l
hD3jG6ZcbvrZ8g4K1iDBODovR2X5f4ingA9RNmgE8DGpLFx8R1nPV7TFtS7FtX7U
LOEsmfCSKIgD/Fn42K6pUPgLgYkeZm5XMJxtlTvFraw33S1HhQs5yHirTdFruv8f
ewJ5AM194LXcRHZSKFQcz9U+4qFrtywDp/2fPTHqeofmZMNpiFVSb6VGFtwceMXQ
EpVY7B9zFqrxWgnt24qSYQD2loa/MdoB4eAJQAiadL9vpOSA9GnSIEHLPb/Gb1Kj
n8UYY5IX6iR5Y+yUBgbXEFKCpOS+IMyowAIF7QKBiCBTVXO56GOTMCosLoJOUYr+
lYvD99gcmoxNeZuFCvoFFVtjxmdTqK4/BjXs5pJOjtbG9ytA1PcnTmldHjfzygys
QnoJRHVRhOeAyduX1oJKZmPiITwzWpmlIawzcFVb6/3nMnS4IB5sxo9ItKSMjehu
LLcKw8pi5Ckb4mRUROnMBOc8Axm1zHECeGdxiZAi/PfPLewBwQHpMS7vSXtHr+Z1
pdmd3ubQrULNukqxP/CkEtflIUr7Do/N/rdqOSv+aUP8T/s26swQVgAo+VBOiB8+
qNRr5H+GFl7ex6sJkyFrzX7fxqSEJ+aQ+3bMwRpxXUL31PzWgKn4iEd6CorcHCQo
lQKBiQCDCLDzZFH6d1qCQ3hhEmoddRcTHNuHRzex3F2g/dzEmsRmVWWO8U5ZOYOY
X5+AqfyqW2oyauLtQHmCRIKbIiT18yGViDe7h4tfsjqitbN/bP4RQSOPx4wQDZjJ
f+rGbkX5Mljo8F+l9aOgKmbfoYqDM2U68L9dS7W6sVJPaPpSDL7EJLOfcd6t
-----END RSA PRIVATE KEY-----
```

2. Public Key (public_key.pem)

Format: PEM format, similar to the private key but used for encryption.

Purpose:

- **Encryption**: The public key is used to encrypt data. This encrypted data can only be decrypted by the corresponding private key, ensuring that the data remains confidential.

Contents:

- **Key Header**: Indicates the start of the RSA public key.
- **Key Body**: Contains the Base64 encoded public key data.
- **Key Footer**: Indicates the end of the RSA public key.

Example:

```
-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEArB6JB92u1m6GA69xIE764py46qzco8Ro3TqfFMBELaam2dzkVuVv
XmQVuPGZKbPYYwkhv5LApaxyJNAPglyAaMOCn9lBDjNJ9pqwoM7NP7SZzS9SCnBB
SU5hCJ5JrT2bMdJ1R6DMWUhUOV7D/EFFKVUqOYHIQGx6XaS4IK5p7tei7HgTgL4c
O42O3QDJ22HhxtvKMl3e3wz9jwIDz8748l3GpW4lVo1eCOyTEJ+S+f6LcrPnBRDC
```

fReo7MCXJlPT/G/XFbQVDfJYO/2oAXncfEhP6TSa++sIrU3q2GdJL408+95aE+Ao

```
ieZa50Ig57223QGZHYThfT7JG3W1mMOL3wIDAQAB
-----END RSA PUBLIC KEY-----
```

How RSA Keys Work Together

Encryption and Decryption Process

1. **Encryption**:
   - When storing sensitive data (e.g., passwords), the data is encrypted using the public key. This ensures that only the corresponding private key can decrypt the data.
   - In the script, the `rsa.encrypt()` function uses the public key to encrypt the passwords before storing them in the MongoDB database.
2. **Decryption**:
   - When retrieving sensitive data, the encrypted data is decrypted using the private key. This restores the original data from its encrypted form.
   - In the script, the `rsa.decrypt()` function uses the private key to decrypt the passwords when fetching them from the MongoDB database.

Key Management

- **Key Storage**:
  - The keys are stored in files (`private_key.pem` and `public_key.pem`). These files must be securely stored and protected to prevent unauthorized access.
  - The private key should be kept confidential, while the public key can be shared as needed.
- **Access Control**:
  - Ensure that access to the key files is restricted to authorized users only.
  - Regularly review and rotate keys as part of your security practices to mitigate potential risks.

Example Usage in the Script

Encrypting Data:

```
encrypted_app_pass = rsa.encrypt(app_pass.encode('utf-8'), public_key)
```

- `app_pass` is encrypted using the `public_key`.

Decrypting Data:

```
app_pass = rsa.decrypt(bytes.fromhex(encrypted_app_pass),
private_key).decode('utf-8')
```

- `encrypted_app_pass` is decrypted using the `private_key`.

Security Considerations

1. **Confidentiality:**
   - The use of RSA encryption ensures that passwords and other sensitive data are protected while stored in the database.

2. **Integrity:**
   - RSA encryption helps in maintaining data integrity by preventing unauthorized modifications.

3. **Access Controls:**

- Ensure that the private key file is protected and access is restricted to prevent unauthorized decryption.

4. **Key Rotation:**
   - Periodically update and rotate RSA keys to enhance security.

4. realtime_server.py

```
import datetime
import json
import os
import imaplib
import email
from email.header import decode_header
from email.policy import default
import hashlib
import re
from bs4 import BeautifulSoup
import pandas as pd
from urllib.parse import urlparse, parse_qs
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.base import MIMEBase
import dns.resolver
import requests
import joblib
import whois
from cryptography import x509
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.x509.oid import NameOID
from datetime import datetime, timezone
import base64
import OpenSSL
import traceback
from concurrent.futures import ThreadPoolExecutor, as_completed
from email.parser import BytesParser
import magic
from mimetypes import guess_extension
import numpy as np
import lightgbm
import time
from tld import get_tld
import subprocess
import multiprocessing


import blacklist_pulldown
import mongodb_rsa as mongodb


global server_email, server_pass, tempvar,filename
data = None

#____________________________(AUTH FUNCTIONS)_____________________________
def get_mx_record(domain):
  try:
    # Get the MX record for the domain
    mx_records = dns.resolver.resolve(domain, 'MX')
```

```
        mx_record = sorted(mx_records, key=lambda r: r.preference)[0]
        return mx_record.exchange.to_text().strip()  # Remove any extra spaces and periods
    except Exception as e:
        print(f"An error occurred while fetching MX record: {e}")
        return None

def server_fetch(email_address, password):
    # Dictionary of common email providers and their IMAP servers
    IMAP_SERVERS = {
        'gmail.com': 'imap.gmail.com',
        'yahoo.com': 'imap.mail.yahoo.com',
        'outlook.com': 'imap-mail.outlook.com',
        'hotmail.com': 'imap-mail.outlook.com',
        'live.com': 'imap-mail.outlook.com',
        'aol.com': 'imap.aol.com',
        'icloud.com': 'imap.mail.me.com',
        'google.com': 'imap.gmail.com',
        'aspmx.l.google.com': 'imap.gmail.com' # Specific handling for MX server
    }
    # Split the email to get the domain
    domain = email_address.split('@')[1]
    if not domain:
        print("Invalid email address.")
        return None

    # Check if the domain is in the dictionary
    email_server = IMAP_SERVERS.get(domain)
    if not email_server:
        # If the domain is not in the dictionary, get the MX record
        mx_server = get_mx_record(domain)
        if not mx_server:
            print(f"Failed to retrieve MX record for domain: {domain}")
            return None

        # Remove trailing period if present
        mx_server = mx_server.rstrip('.')

        email_server = IMAP_SERVERS.get(mx_server)
        if email_server:
            return email_address, password, email_server
        else:
            # Extract the base domain using regex
            match = re.search(r'([a-zA-Z0-9-]+\.[a-zA-Z]+)$', mx_server)
            if match:
```

```
            base_domain = match.group(0)
            email_server = IMAP_SERVERS.get(base_domain, f'imap.{base_domain}')
        else:
            print(f"Failed to parse base domain from MX server: {mx_server}")
            return None

    return email_server


#__________________AI MODEL FUCNTIONS HERE______________
def get_tld_plus_one(domain):
    parts = domain.split('.')

    # If the domain has more than two parts, join the last two parts to get the TLD + 1
    if len(parts) > 2:
        tld_plus_one = ".".join(parts[-2:])
    else:
        tld_plus_one = domain  # If the domain is already in the TLD + 1 format

    return tld_plus_one

# Function to get domain age using python-whois
def get_domain_age(domain: str) -> int:
    try:
        whois_info = whois.whois(domain)
        creation_date = whois_info.creation_date
        if isinstance(creation_date, list):
            creation_date = creation_date[0]
        elif creation_date is None:
            print(f"Creation date not found for {domain}")
            return None
        age = (datetime.now() - creation_date).days
        return age
    except Exception as e:
        print(f"Error getting domain age for {domain}: {e}")
        return None

# Function to get domain rank using Open PageRank API
def get_domain_rank(domain):
    try:
        url =
f"https://openpagerank.com/api/v1.0/getPageRank?domains%5B0%5D={domain}"
        headers = {
            'API-OPR': 'ow08ogc0sgs84swk4g8g8o0kgogsgkcw8cs4g48k'  # Replace with your
Open PageRank API key
        }
        response = requests.get(url, headers=headers)
        data = response.json()
        if data and 'response' in data and data['response']:
            rank = data['response'][0]['page_rank_integer']
            return int(rank) # Ensure the rank is an integer
        return None
    except Exception as e:
        print(f"Error getting rank for {domain}: {e}")
        return None

# Function to check for popular keywords in domain
def check_popular_keywords(domain):
    popular_keywords = ["shop", "bank", "secure", "account", "login"]
    return any(keyword in domain for keyword in popular_keywords)

# Function to check if TLD is trustworthy
def check_tld(domain):
    trustworthy_tlds = [".com", ".org", ".net", ".edu", ".gov"]
    tld = '.' + domain.split('.')[-1]
    return tld in trustworthy_tlds

# Function to check domain reputation
def check_domain_reputation(domain):
    if domain is None:
        return 0.0, ["Domain is None."]

    score = 0
    reasons = []

    # Extract base domain (strip protocol and subdomains)
    domain_name = domain.split('//')[-1].split('/')[0]

    # Check domain age
    age = get_domain_age(domain_name)
    if age is not None:
        if age > 365:
            score += 0.2
        else:
            reasons.append("Domain age is less than 1 year.")
    else:
        reasons.append("Unable to determine domain age.")
    # Check domain rank
    rank = get_domain_rank(domain_name)
    if rank is not None:
        if rank >= 2:  # Adjust threshold as per Open PageRank scale
            score += 0.2
        else:
            reasons.append("Domain rank (Open PageRank) is less than 2.")
    else:
        reasons.append("Unable to determine domain rank.")

    # Check popular keywords
    if check_popular_keywords(domain_name):
        score += 0.2
    else:
        reasons.append("Domain does not contain popular keywords.")

    # Check TLD
    if check_tld(domain_name):
        score += 0.2
    else:
        reasons.append("TLD is not among the most trustworthy ones.")

    # Final adjustment
    if score >= 0.6:
        score = 1.0
    elif score >= 0.3:
        score = 0.5
    else:
        score = 0.0

    if score == 1.0:
        reasons.append("Domain has high reputation based on heuristics.")
    elif score == 0.5:
        reasons.append("Domain has medium reputation based on heuristics.")
    else:
        reasons.append("Domain has low reputation based on heuristics.")

    return score, reasons



# Function to extract features from URLs
def extract_features(url):
    parsed_url = urlparse(url)
```

```
  features = {
    'url_length': len(url),
    'hostname_length': len(parsed_url.netloc),
    'use_of_ip': int(bool(re.search(
      r'(([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-
4]\d|25[0-5])\.'
      r'([01]?\d\d?|2[0-4]\d|25[0-5])\/)|'
      r'((0x[0-9a-fA-F]{1,2})\.(0x[0-9a-fA-F]{1,2})\.(0x[0-9a-fA-F]{1,2})\.(0x[0-9a-fA-F]{1,2})\/)'
      r'(?:[a-fA-F0-9]{1,4}:){7}[a-fA-F0-9]{1,4}', parsed_url.hostname))) if
parsed_url.hostname else 0,
    'abnormal_url': int(bool(re.search(str(parsed_url.hostname), url))),
    'count.': url.count('.'),
    'count-www': url.count('www'),
    'count@': url.count('@'),
    'count_dir': parsed_url.path.count('/'),
    'count_embed_domian': parsed_url.path.count('//'),
    'short_url':
int(bool(re.search(r'bit\.ly|goo\.gl|shorte\.st|go2l\.ink|x\.co|ow\.ly|t\.co|tinyurl|tr\.im|is\.gd|
cli\.gs|'

r'yfrog\.com|migre\.me|ff\.im|tiny\.cc|url4\.eu|twit\.ac|su\.pr|twurl\.nl|snipurl\.com|'

r'short\.to|BudURL\.com|ping\.fm|post\.ly|Just\.as|bkite\.com|snipr\.com|fic\.kr|loopt\.us|
'

r'doiop\.com|short\.ie|kl\.am|wp\.me|rubyurl\.com|om\.ly|to\.ly|bit\.do|t\.co|lnkd\.in|'

r'db\.tt|qr\.ae|adf\.ly|goo\.gl|bitly\.com|cur\.lv|tinyurl\.com|ow\.ly|bit\.ly|ity\.im|'

r'q\.gs|is\.gd|po\.st|bc\.vc|twitthis\.com|u\.to|j\.mp|buzurl\.com|cutt\.us|u\.bb|yourls\.org
|'

r'x\.co|prettylinkpro\.com|scrnch\.me|filoops\.info|vzturl\.com|qr\.net|1url\.com|tweez\.m
e|v\.gd|'
                r'tr\.im|link\.zip\.net',  url))),
    'count-https': url.count('https'),
    'count-http': url.count('http'),
    'count%':  url.count('%'),
    'count?': url.count('?'),
    'count-': url.count('-'),
    'count=': url.count('='),
    'sus_url':
int(bool(re.search(r'PayPal|login|signin|bank|account|update|free|lucky|service|bonus|eba
yisapi|webscr', url))),
        'count-digits': sum(c.isdigit() for c in url),
        'count-letters': sum(c.isalpha() for c in url),
        'fd_length': len(parsed_url.path.split('/')[1]) if len(parsed_url.path.split('/')) > 1 else 0,
        'tld_length': len(get_tld(url, fail_silently=True)) if get_tld(url, fail_silently=True) else -1
    }
    return features

# Function to check if URL domain matches sender domain
def check_same_domain(url, sender_domain):
    parsed_url = urlparse(url)
    return int(parsed_url.hostname == sender_domain)

def validate_and_clean_urls(url_list):
    pattern = re.compile(
        r'^(?:http|ftp)s?://'  # http:// or https://
        r'(?:(?:[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?\.)+(?:[A-Z]{2,6}\.?|[A-Z0-9-]{2,}\.?)|'  #
domain...
        r'localhost|'  # localhost...
        r'\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})'  # ...or ip
        r'(?::\d+)?'  # optional port
        r'(?:/?|[/?]\S+)$', re.IGNORECASE)

    cleaned_url_list = []
    for url in url_list:
        # Remove any invalid characters
        cleaned_url = re.sub(r'[<>]', '', url)
        # Check if the cleaned URL is valid
        if re.match(pattern, cleaned_url):
            # Add the cleaned URL to the list if it's not already there
            if cleaned_url not in cleaned_url_list:
                cleaned_url_list.append(cleaned_url)

    return cleaned_url_list

# Function to extract URLs from email content
def extract_urls_from_email(email_content):
    urls = []
    for part in email_content.walk():
        if part.get_content_type() == 'text/plain':
            text_content = part.get_payload(decode=True).decode('utf-8', errors='ignore')
            urls.extend(re.findall(r'https?://\S+', text_content))
        elif part.get_content_type() == 'text/html':
            html_content = part.get_payload(decode=True).decode('utf-8', errors='ignore')
            soup = BeautifulSoup(html_content, 'html.parser')
            urls.extend([a['href'] for a in soup.find_all('a', href=True) if a['href'].startswith('http')])

    urls = validate_and_clean_urls(urls)
    return urls

# Function to extract sender's domain
def get_sender_domain(msg):
    sender = email.utils.parseaddr(msg.get('From'))[1]
    domain = sender.split('@')[-1]
    return domain

# Function to get sender IP from email headers
def get_sender_ip(headers):
    for header, value in headers:
        if header.lower() == 'received':
            ip_match = re.search(r'\[(\d+\.\d+\.\d+\.\d+)\]', value)
            if ip_match:
                return ip_match.group(1)
    return None

# Function to check DMARC
def ml_check_dmarc(domain):
    try:
        answers = dns.resolver.resolve('_dmarc.' + domain, 'TXT')
        for rdata in answers:
            for txt_string in rdata.strings:
                if 'v=DMARC1' in txt_string.decode():
                    return True
        return False
    except Exception as e:
        print(f"Error checking DMARC for domain {domain}: {e}")
        return False

# Function to get DNS information
def get_dns_info(domain):
    if domain is None:
        print("Domain is None. Cannot fetch DNS info.")
        return []

    try:
        dns_info = dns.resolver.resolve(domain, 'A')
        print("DNS A records:", [ip.address for ip in dns_info])
        return [ip.address for ip in dns_info]
    except Exception as e:
        print(f"Error getting DNS info for domain {domain}: {e}")
        return []


# Function to calculate phishing score and provide reasons
def calculate_phishing_score(reasons):
    score = 0
    if "URL and sender domain do not match." in reasons:
        score += 1
    if "Domain not whitelisted." in reasons:
        score += 1
    if "Domain blacklisted." in reasons:
        score += 1
    if "IP blacklisted." in reasons:
        score += 1
    if "Classified as phishing by model." in reasons:
        score += 1
    if "Poor domain reputation." in reasons:
        score += 1

    ratio = score / 6 # 6 is the total number of checks
    return ratio

# Function to classify URLs in emails and state the reason for phishing classification
def classify_urls_in_email(email_content, whitelisted_domains, clf):
    try:
        urls = extract_urls_from_email(email_content)
        print(urls)
        if not urls:
            print("No URLs found in the email.")
            return
        url_analysis = []
        sender_domain = get_sender_domain(email_content)
        headers = email_content.items()
        sender_ip = get_sender_ip(headers)
        for url in urls:
            domain = urlparse(url).hostname
            print(f"Domain Found: {domain}")
            if domain is None:
                print(f"URL {url} has no hostname.")
                continue

            reasons = []
        # Step 1: Check if sender domain matches the URL domain
        tld_plus_one = get_tld_plus_one(domain)
        if tld_plus_one != sender_domain:
          reasons.append("URL and sender domain do not match.")

        # Step 2: Check if domain is whitelisted
        if domain in whitelisted_domains:
          print(f"Whitelisted domain found: {domain}")
          # Reason for legitimate URL
          print(f"URL {url} is legitimate because it is from a whitelisted domain.")
          continue
        else:
          reasons.append("Domain not whitelisted.")

        # Step 3: Check if domain or IP is blacklisted
        ml_domain_check, bl_domain = check_domain_exists(domain)
        if ml_domain_check ==True:
          reasons.append("Domain blacklisted.")
        else:
          domain_ips = get_dns_info(domain)
          ml_ip_check,bl_ips = check_ips_exist(domain_ips)
          if ml_ip_check == True :
            reasons.append("IP blacklisted.")

        if not ml_check_dmarc(sender_domain):
          reasons.append("DMARC validation failed.")

        # Step 6: Model prediction
        url_features = extract_features(url)
        # Ensure all features are numeric and in correct format
        try:
          features_list = [float(url_features[key]) for key in sorted(url_features.keys())]  #
Convert to list of floats
          features_test = np.array(features_list).reshape((1, -1))  # Convert to 2D numpy array
        except ValueError as ve:
          print(f"Error converting features to floats: {ve}")
          continue

        # Assuming model outputs 'phishing' or 'legitimate'
        predicted_label = clf.predict(features_test)[0]

        if predicted_label == 'phishing':
          reasons.append("Classified as phishing by model.")
```

```
            # Step 7: Domain reputation check
            reputation_score, reputation_reasons = check_domain_reputation(domain)
            if reputation_score < 0.5:
                reasons.append("Poor domain reputation.")

            # Calculate phishing score and classify
            phishing_score = calculate_phishing_score(reasons)
            print(f"Phishing score for URL {url}: {phishing_score}")
            print("Reasons for classification:")
            for reason in reasons:
                print(f"- {reason}")
            if phishing_score > 0.5:
                reason_string = f"{url}:{reasons}"
                url_analysis.append(reason_string)

        return url_analysis

    except Exception as e:
        print(f"Error processing machine learning model: {e}")
        traceback.print_exc()

#________________CERTIFICATE VERIFICATION________________

def extract_certificate_from_eml(eml_content):
    """Extracts a certificate from email content.
    Args:
        eml_content (bytes): Raw email content.
    Returns:
        Tuple (bytes, str): Certificate data and format or None."""
    msg = email.message_from_bytes(eml_content)

    # Check for certificate in headers
    for header, value in msg.items():
        if 'CERTIFICATE' in value.upper():
            try:
                cert_data = base64.b64decode(value.strip())
                return cert_data, 'pem'
            except Exception as e:
                print(f"Error decoding certificate from header {header}: {str(e)}")

    # Check for certificate in body
    if msg.is_multipart():
        for part in msg.walk():
            content_type = part.get_content_type()
            filename = part.get_filename()
            content_disposition = str(part.get("Content-Disposition"))
            payload = part.get_payload(decode=True)

            if (content_type == 'application/x-x509-ca-cert' or
                content_type == 'application/pkcs7-mime' or
                (filename and filename.endswith(('.cer', '.der', '.pfx')))):
                return payload, 'attachment'
            elif 'attachment' not in content_disposition:
                # Check for inline certificate data
                try:
                    decoded_payload = base64.b64decode(payload.strip())
                    if b'-----BEGIN CERTIFICATE--- ' in decoded_payload:
                        return decoded_payload, 'pem'
                    elif decoded_payload.startswith(b'0\x82'):
                        return decoded_payload, 'der'
                except Exception as e:
                    continue
    else:
        payload = msg.get_payload(decode=True)
        try:
            if b'-----BEGIN CERTIFICATE-----' in payload:
                return payload, 'pem'
            elif payload.startswith(b'0\x82'):
                return payload, 'der'
        except Exception as e:
            print(f"Error decoding certificate from body: {str(e)}")

    return None, None

def extract_sender_email(eml_content):
    """Extracts the sender's email address.
        eml_content (bytes): Raw email content.
    Returns:
        str: Sender's email address."""
    msg = email.message_from_bytes(eml_content)
    sender = msg.get('From')
    return sender

def load_certificates(cert_data, cert_format):
    """Loads certificates from data.
    Args:
        cert_data (bytes): Certificate data.
        cert_format (str): Format of the certificate (pem, der, pfx).
    Returns:
        List[x509.Certificate]: List of certificates."""
    certificates = []
    if cert_format == 'pem':
        for cert in cert_data.split(b'-----END CERTIFICATE----- '):
            cert = cert.strip()
            if cert:
                cert = cert + b'-----END CERTIFICATE ---- '
                certificates.append(x509.load_pem_x509_certificate(cert, default_backend()))
    elif cert_format == 'der':
        certificates.append(x509.load_der_x509_certificate(cert_data, default_backend()))
    elif cert_format == 'pfx':
        pfx = OpenSSL.crypto.load_pkcs12(cert_data)
        certificates.append(pfx.get_certificate())
    return certificates

def verify_certificate_chain(certificates):
    """Verifies the certificate chain.
    Args:
        certificates (List[x509.Certificate]): List of certificates.
    Returns:
        Tuple (bool, str): Chain validity and message."""
    for i in range(len(certificates) - 1):
        cert = certificates[i]
        issuer_cert = certificates[i + 1]
        try:
            cert.public_key().verify(
                cert.signature,
                cert.tbs_certificate_bytes,
                cert.signature_hash_algorithm
            )
        except Exception as e:
            return False, f"Verification failed for certificate {i}: {e}"
        if not cert.issuer == issuer_cert.subject:
            return False, f"Certificate {i} issuer does not match the subject of certificate {i+1}"
    return True, None

def check_expiration(cert):
    """Checks if the certificate is expired.
    Args:
        cert (x509.Certificate): The certificate to check.
    Returns:
        Tuple (bool, str): Expiration status and message."""
    not_after = cert.not_valid_after.replace(tzinfo=timezone.utc)
```

```
  if not_after < datetime.now(timezone.utc):
    return False, "Certificate is expired."
  return True, f"Certificate is valid until: {not_after}"

def verify_email(cert, email_address):
  """Verifies if the email matches the certificate.
  Args:
    cert (x509.Certificate): The certificate.
    email_address (str): The email address to verify.
  Returns:
    Tuple (bool, str): Email match status and message."""
  try:
    common_name =
cert.subject.get_attributes_for_oid(NameOID.COMMON_NAME)[0].value
    if email_address == common_name:
      return True, "Email address matches the certificate (Common Name)."
    for ext in cert.extensions:
      if ext.oid == x509.oid.ExtensionOID.SUBJECT_ALTERNATIVE_NAME:
        san = ext.value.get_values_for_type(x509.RFC822Name)
        if email_address in san:
          return True, "Email address matches the certificate (Subject Alternative Name)."
    return False, "Email address does not match the certificate."
  except Exception as e:
    return False, f"Email verification failed: {e}"

def validate_email_certificate(eml_content):
  """Validates the certificate from an email.
  Args:
    eml_content (bytes): Raw email content."""
  cert_data, cert_location = extract_certificate_from_eml(eml_content)
  if not cert_data:
    return False

  # Determine certificate format (PEM, DER, PFX)
  cert_format = 'pem'  # Default to PEM
  if b'-----BEGIN CERTIFICATE---- ' in cert_data:
    cert_format = 'pem'
  elif cert_data.startswith(b'0\x82'):
    cert_format = 'der'
  elif b'-----BEGIN PKCS7-----' in cert_data or b'-----BEGIN CERTIFICATE -----' not in
cert_data:
    cert_format = 'pfx'

  # Load the certificate chain from the extracted data
```

```
    certificates = load_certificates(cert_data, cert_format)

    # Verify the certificate chain
    is_valid_chain, chain_message = verify_certificate_chain(certificates)
    if not is_valid_chain:
        print(f"Certificate chain is not valid ({cert_location}): {chain_message}")
        return True

    sender_email = extract_sender_email(eml_content)
    clean_sender_email = email.utils.parseaddr(sender_email)[1]

    cert = certificates[0]

    # Verify the email address in the certificate
    is_valid_email, email_message = verify_email(cert, clean_sender_email)
    if not is_valid_email:
        print(f"Certificate validation failed for email {clean_sender_email}: {email_message}
({cert_location})")
        return True

    # Check certificate expiration
    is_not_expired, expiration_message = check_expiration(cert)
    if not is_not_expired:
        print(f"Certificate validation failed for email {clean_sender_email}:
{expiration_message} ({cert_location})")
        return True

    # If all checks pass, we do not print anything as per the requirement
    return False

#_______________SPF verification____________________
def spf_verification(message):
    from_header = message['From']
    email_address_match = re.search(r'[\w\.-]+@[\w\.-]+', from_header)
    if email_address_match:
        email_address = email_address_match.group(0)
        sender_domain = email_address.split('@')[-1]
    else:
        print(f"Could not extract sender domain from {from_header}")
        return False

    # Exclude trusted domains from SPF verification
    trusted_domains = [
        'gmail.com',
        'yahoo.com',
        'outlook.com',
        'office365.com',
        'amazonaws.com',  # for Amazon SES
        'sendgrid.net'
    ]

    if sender_domain in trusted_domains:
        return trusted_spf_verification(message)  # trusted domains

    received_headers = message.get_all('Received')
    ip_addresses = []
    for header in received_headers:
        ip_address_match = re.findall(r'\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b', header)
        if ip_address_match:
            ip_addresses.extend(ip_address_match)

    if not ip_addresses:
        print(f"No IP addresses found in Received headers")
        return False

    spf_result = None
    for ip_address in ip_addresses:
        try:
            spf_record = dns.resolver.resolve(sender_domain, 'TXT')
            for txt in spf_record:
                spf_policy = str(txt)
                if 'v=spf1' in spf_policy:
                    for mechanism in spf_policy.split():
                        if mechanism.startswith('ip4:'):
                            allowed_ip = mechanism.split(':')[1]
                            if ip_address == allowed_ip:
                                spf_result = True # SPF pass
                                break
                        elif mechanism.startswith('ip6:'):
                            allowed_ip = mechanism.split(':')[1]
                            if ip_address == allowed_ip:
                                spf_result = True # SPF pass
                                break
                        elif mechanism == '-all':
                            spf_result = False # SPF fail
                            break
                    if spf_result is not None:
                        break
```

```
      except dns.resolver.NoAnswer:
        print(f"No SPF policy published for {sender_domain}")
        spf_result = False  # SPF fail
      except dns.resolver.NXDOMAIN:
        print(f"Domain {sender_domain} does not exist")
        spf_result = False  # SPF fail
      except dns.exception.DNSException as e:
        print(f"DNS error: {e}")
        spf_result = False # SPF fail

    if spf_result is None:
      print(f"No SPF policy found for {sender_domain}")
      return False  # SPF fail
    else:
      return spf_result

 def trusted_spf_verification(msg_data):
     for header, value in msg_data.items():
      if header == 'Received-SPF':
        spf_value = value
        break
    else:
      return False # SPF header not found

    if 'pass' in spf_value:
     spf_result = 'Pass'
    elif 'fail' in spf_value:
     spf_result = 'Fail'
    elif 'neutral' in spf_value:
      spf_result = 'Neutral'
    else:
      return True  # Unknown SPF result

    if spf_result == 'Fail':
      return False
    else:
      return True
#_______________________JSON OBJECT HANDLING___________________________

def convert_bytes_to_str(data):
  if isinstance(data, bytes):
    return data.decode('utf-8', errors='ignore')
  if isinstance(data, dict):
    return {key: convert_bytes_to_str(value) for key, value in data.items()}
    if isinstance(data, list):
      return [convert_bytes_to_str(item) for item in data]
    return data

def update_json_value(key, value, check_object):
    if key in check_object:
      check_object[key] = value
    else:
      raise KeyError(f"Key {key} does not exist in the dictionary")

def evaluate_email_check(check_object):

    email_body = ""
    spam_check = None

    # Check if the email is flagged as phishing
    if (check_object["ip_blacklist_check"] or
      check_object["domain_blacklist_check"] or
      check_object["malware_check"] or
      check_object["file_extension_check"] or
      (check_object["dmarc_check"] and check_object["spf_verification_check"])):
      spam_check = True
      email_body += "Email flagged as phishing.\n"
      update_json_value("email_flag","Phishing", check_object)
    # Check if the email is flagged as suspicious
    elif (check_object["dmarc_check"] or
       check_object["spf_verification_check"] or
       check_object["certificate_check"] or

       check_object["dkim_check"] or
       check_object["url_analysis_result"]):
      spam_check = False
      email_body += "Email flagged as suspicious.\n"
      update_json_value("email_flag","Suspicious", check_object)

    # Add failed checks to the email body
    if check_object["ip_blacklist_check"]:
      email_body += "Failed IP blacklist check. Blacklisted IPs: {}\n".format(",
".join(check_object["blacklisted_ips"]))
    if check_object["domain_blacklist_check"]:
      email_body += "Failed domain blacklist check. Blacklisted domain:
{}\n".format(check_object["blacklisted_domain"])
    if check_object["malware_check"]:
        email_body += "Failed malware check. Malware files: {}\n".format(",
".join(check_object["malware_files"]))
    if check_object["dmarc_check"]:
        email_body += "Failed DMARC check.\n"
    if check_object["spf_verification_check"]:
        email_body += "Failed SPF verification check.\n"
    if check_object["certificate_check"]:
        email_body += "Failed certificate check.\n"
    if check_object["file_extension_check"]:
        email_body += "Failed file extension check. Blacklisted file names: {}\n".format(",
".join(check_object["blacklisted_file_names"]))
    if check_object["dkim_check"]:
        email_body += "Failed DKIM check.\n"
    if check_object["url_analysis_result"]:
        email_body += "Failed URL analysis check.\n"
        for result in check_object["url_analysis_result"]:
            email_body += "{}\n".format(result)

    # If no checks failed, set email_body and spam_check to None
    if not email_body:
        email_body = None
        spam_check = None

    return email_body, spam_check, check_object

#______________________(SPAM AND ALERT)____________________________
def notify_user(user, server_mail, server_pass, body, spam_check):
    try:
        smtp_server = "smtp.gmail.com"
        smtp_port = 587
        if spam_check == True:
            subject = "Phishing Email Found"
        else:
            subject = "Suspicious Email Found"

        msg = MIMEMultipart()
        msg['From'] = server_mail
        msg['To'] = user
        msg['Subject'] = subject

        # Attach the email body
        msg.attach(MIMEText(body, 'plain'))

        # Connect to the SMTP server
        server = smtplib.SMTP(smtp_server, smtp_port)
        server.starttls()  # Secure the connection
        server.login(server_mail, server_pass)

        # Send the email
        server.send_message(msg)
        server.quit()

        print("Email sent successfully.")

    except smtplib.SMTPAuthenticationError as auth_error:
        print(f"SMTP Authentication Error: {auth_error}")
        print("Check if the email and password are correct. If you are using an app-specific
password, ensure it is generated correctly.")

def move_to_spam(mail, num, mail_server):
    try:
        if mail_server == 'imap.gmail.com':
            mail.store(num, '+X-GM-LABELS', '\\Spam')
        elif mail_server == 'imap.yahoo.com':
            spam_folder = "Bulk"
            # Copy the email to the spam/junk folder
            copy_status = mail.copy(num, spam_folder)
            if copy_status[0] != "OK":
                print(f"Error copying email ID {num} to {spam_folder}: {copy_status[1]}")

            # Mark the email for deletion in the original folder
            delete_status = mail.store(num, '+FLAGS', '\\Deleted')
            if delete_status[0] != "OK":
                print(f"Error marking email ID {num} for deletion: {delete_status[1]}")
            mail.expunge()

        elif mail_server == 'imap-mail.outlook.com':
            # Copy the email to the Junk/Spam folder
            mail.copy(num, "Junk")

            # Mark the email for deletion in the original folder
            mail.store(num, '+FLAGS', '\\Deleted')
        else:
            mail.store(num, '+X-GM-LABELS', '\\Spam')

        print("Email moved to Spam folder.")
    except Exception as e:
        print(f"Error moving message to Spam folder: {e}")
```

```
#____________________(DMARC and DKIM)____________________________________

def check_dmarc(message):
  from_header = message['From']
  email_address_match = re.search(r'[\w\.-]+@[\w\.-]+', from_header)
  if email_address_match:
    email_address = email_address_match.group(0)
    sender_domain = email_address.split('@')[-1]
  else:
    print(f"Could not extract sender domain from {from_header}")
    return False

  try:
    dmarc_record = dns.resolver.resolve('_dmarc.' + sender_domain, 'TXT')
    for txt in dmarc_record:
      dmarc_policy = str(txt)
      if 'v=DMARC1;' in dmarc_policy:
        for part in dmarc_policy.split(';'):
          if part.startswith('p='):
            policy = part.split('=')[1]
            if policy in ['reject', 'quarantine']:
              return True  # Phishing sign
      else:
        print(f"Invalid DMARC policy for {sender_domain}: {dmarc_policy}")
        return False
  except dns.resolver.NoAnswer:
    print(f"No DMARC policy published for {sender_domain}")
    return False
  except dns.resolver.NXDOMAIN:
    print(f"Domain {sender_domain} does not exist")
    return False
  except dns.exception.DNSException as e:
    print(f"DNS error: {e}")
    return False

# File Extension Check
def checkext_and_hash(filename, response_part):
  # Check if attachment is blacklisted
  blacklist = ['.pyx','.pyz','.exe', '.bat', '.cmd', '.js', '.adp', '.app', '.asp', '.bas', '.bat', '.cer',
'.chm', '.cmd', '.cnt', '.com', '.cpl', '.crt', '.csh', '.der', '.exe', '.fxp', '.gadget', '.hlp', '.hpj',
'.hta', '.inf', '.ins', '.isp', '.its', '.js', '.jse', '.ksh', '.lnk', '.mad', '.maf', '.mag', '.mar', '.mam',
'.maq', '.mas', '.mat', '.mau', '.mav', '.mda', '.mdb', '.mde', '.mdt', '.mdw', '.mdz', '.msc',
'.msh', '.msh1', '.msh2', '.mshxml', '.msh1xml', '.msh2xml', '.msi', '.msp', '.mst', '.ops',
'.pcd', '.pif', '.plg', '.prf', '.prg', '.pst', '.reg', '.scf', '.scr', '.sct', '.shb', '.shs', '.ps1', '.ps1xml',
'.ps2', '.ps2xml', '.psc1', '.psc2', '.tmp', '.url', '.vb', '.vbe', '.vbp', '.vbs', '.vsmacros', '.vsw',
'.ws', '.wsc', '.wsf', '.wsh', '.xnk', '.ade', '.cla', '.class', '.grp', '.jar', '.mcf', '.ocx', '.pl', '.xbap']
  if any(filename.lower().endswith(ext) for ext in blacklist):
    return True
  #Checking File hash Against Malware Database
  attachment_content=response_part.get_payload(decode=True)
  return False


def calculate_memory_hash(content):
  # Calculate SHA256 hash of the content
  sha256_hash = hashlib.sha256()
  sha256_hash.update(content)
  return sha256_hash.hexdigest()

#Blaclist_check

def check_ips_exist(ips_to_check):
  # Load the DataFrame from the pkl file
  filename = "processed_ip.pkl"
  check = None
  list = []
  df = pd.read_pickle(filename)

  # Check if any of the IPs in ips_to_check exist in the blacklist
  blacklist = df['ip'].values
  for ip in ips_to_check:
    if ip in blacklist:
      print(f"Alert: {ip} is Blacklisted !!!!")
      check = True
      list.append(ip)
  if check == True :
    return True , list
  return False , None


def check_domain_exists(domain_to_check):
  # Load the DataFrame from the pkl file
  filename = "processed_domains.pkl"
  df = pd.read_pickle(filename)

  exists = domain_to_check in df['domain'].values

  if exists:
        print(f"Alert : {domain_to_check} is Blacklisted !!!!")
        return True , domain_to_check
    return False , None

#Real_Time_analysis

def process_emails(email_id, msg_data,fetch_response, user,mail, server_email,
server_pass, email_server, clf, whitelisted_domains):
    try:
        #____________________AI MODEL INITIATION CODE_________________________
        # Load the dataset and update whitelist
        #data = pd.read_csv(r"C:\Users\moksh\swiftsafe\malicious_phish_cleaned.csv")

        blacklist = ['.pyx','.pyz','.exe', '.bat', '.cmd', '.js', '.adp', '.app', '.asp', '.bas', '.bat', '.cer',
'.chm', '.cmd', '.cnt', '.com', '.cpl', '.crt', '.csh', '.der', '.exe', '.fxp', '.gadget', '.hlp', '.hpj',
'.hta', '.inf', '.ins', '.isp', '.its', '.js', '.jse', '.ksh', '.lnk', '.mad', '.maf', '.mag', '.mar', '.mam',
'.maq', '.mas', '.mat', '.mau', '.mav', '.mda', '.mdb', '.mde', '.mdt', '.mdw', '.mdz', '.msc',
'.msh', '.msh1', '.msh2', '.mshxml', '.msh1xml', '.msh2xml', '.msi', '.msp', '.mst', '.ops',
'.pcd', '.pif', '.plg', '.prf', '.prg', '.pst', '.reg', '.scf', '.scr', '.sct', '.shb', '.shs', '.ps1', '.ps1xml',
'.ps2', '.ps2xml', '.psc1', '.psc2', '.tmp', '.url', '.vb', '.vbe', '.vbp', '.vbs', '.vsmacros', '.vsw',
'.ws', '.wsc', '.wsf', '.wsh', '.xnk', '.ade', '.cla', '.class', '.grp', '.jar', '.mcf', '.ocx', '.pl', '.xbap']
        check_object = {
            "email": user,
            "email_id": False,
            "email_flag": None,
            "date-time": None,
            "dmarc_check": False,
            "dkim_check": False,
            "domain_blacklist_check": False,
            "blacklisted_domain": None,
            "ip_blacklist_check": False,
            "blacklisted_ips": [],
            "spf_verification_check": False,
            "certificate_check": False,
            "file_extension_check": False,
            "blacklisted_file_names": [],
            "malware_check" : False,
            "malware_files" : [],
            "url_analysis_result" : []
        }
        update_json_value("email_id", email_id, check_object)
        update_json_value("date-time", datetime.now().isoformat(), check_object)

        if not msg_data or len(msg_data) < 1:
```

```
    print(f"Email data is empty for {email_id}")
    return

tempvar = None
# _______________________(DATA PARSING)___________________________
raw_email = fetch_response[0][1]
email_message = email.message_from_bytes(msg_data[1], policy=default)
from_header = email_message['From']
print(f"From: {from_header}")
email_address_match = re.search(r'[\w\.-]+@[\w\.-]+', from_header)
if email_address_match:
    email_address = email_address_match.group(0)
    sender_domain = email_address.split('@')[-1]
    print(f"Sender Domain: {sender_domain}")
else:
    print(f"Could not extract sender domain from {from_header}")
    return

received_headers = email_message.get_all('Received')
ip_addresses = []
if received_headers:
    for header in received_headers:
        ip_address_match = re.findall(r'\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b', header)
        if ip_address_match:
            ip_addresses.extend(ip_address_match)
    print(f"IP Addresses: {ip_addresses}")
#_______________________(SPF VERIFICATION)______________
print("--Running SPF Verification")
spf_check=trusted_spf_verification(email_message)
if spf_check == False:
    print("---SPF verification Failed")
    update_json_value("spf_verification_check", True, check_object)

#______________________(BLACKLIST CHECK)________________________
print("--Running DNS Blacklist check")
domainbl_check, bl_domain = check_domain_exists(sender_domain)
if domainbl_check == True:
    print(f"---Domain Blacklist check Failed")
    update_json_value("domain_blacklist_check", True, check_object)
    update_json_value("blacklisted_domain", bl_domain, check_object)

ipbl_check , bl_ips = check_ips_exist(ip_addresses)
if ipbl_check == True:
    print(f"---IP Blacklist check Failed")
```

```
        update_json_value("ip_blacklist_check", True, check_object)
        update_json_value("blacklisted_ips", bl_ips, check_object)
    #______________________(CERT VERIFICATION)______________________
    print("--Running Certiifcate Verification")
    cert_check = validate_email_certificate(raw_email)
    if cert_check == True:
        update_json_value("certificate_check",True,check_object)
    #______________________(DMARC AND DKIM CHECK)____________________
    print("--Running DMARC Check")
    dmarc_check = check_dmarc(email_message)
    if dmarc_check == True:
        print("---DMARC Check Failed !")
        update_json_value("dmarc_check", True , check_object)
    #____________________________(FILE EXTENSION AND MALWARE
CHECK)____________________
    print("--Running File extension and Malware analysis checks")
    filenames = []
    check_flag = None
    for response_part in email_message.walk():
        filename = response_part.get_filename()
        if filename:
            fileext_check = checkext_and_hash(filename, response_part)
            if fileext_check == False :
                file_data = response_part.get_payload(decode=True)
                file_type = magic.from_buffer(file_data, mime=True)
                file_extension = guess_extension(file_type)
                if not file_extension:
                    file_extension = 'unknown'  # If no extension can be determined
                    print("---Unknown File extension found")
                else:
                    print(f"---File extension Detected as :{file_extension}")
                    if file_extension in blacklist:
                        check_flag = True
                        print(f"---Blacklisted Content Found in {filename} of type {file_extension}")
                        filenames.append(filename)
                        continue
            #malware_check = malware check fucntion
            if fileext_check == True:
                print(f"---{filename} Failed file extension check")
                check_flag = True
                filenames.append(filename)
                continue
    if check_flag == True:
        update_json_value("file_extension_check",True, check_object)
        update_json_value("blacklisted_file_names",filenames,  check_object)
      #______________________(MACHINE LEARNING CHECK)___________________
      print("--Running URL Analysis")
      url_analysis_check = classify_urls_in_email(email_message, whitelisted_domains, clf)
      if url_analysis_check:
        print("Suspicous URLS Found")
        update_json_value("url_analysis_result",url_analysis_check, check_object)
      #______________________(EVALUATE AND NOTIFY)_________________
      body,spam_check,check_object = evaluate_email_check(check_object)
      if spam_check == True:
        notify_user(user, server_email, server_pass, body, spam_check)
        move_to_spam(mail, email_id, email_server)
      elif spam_check == False:
        notify_user(user, server_email, server_pass, body, spam_check)
      #___________________(SAVE AND DESTROY JSON OBJECT)_____________
      converted_check_object = convert_bytes_to_str(check_object)
      if not os.path.exists("RTM_log.txt"):
        open("RTM_log.txt", 'w').close()
      with open("RTM_log.txt", 'a') as file:
        json_string = json.dumps(converted_check_object)
        file.write(json_string + '\n')
      check_object = {} #Destroy JSON Object

    except Exception as e:
      print(f"An error occurred during email processing: {str(e)}")

def email_fetchncheck(user, password, email_server, clf, whitelisted_domains,
server_email , server_pass):
    try:
      #___________________(EMAIL AUTH AND FETCH)___________________________
      mail = imaplib.IMAP4_SSL(email_server)
      mail.login(user, password)
      mail.select('inbox')

      # Fetch all unseen email IDs at once
      status, messages = mail.search(None, "UNSEEN")
      email_ids = messages[0].split()

      if not email_ids:
        print(f"No new emails to process - {user}")
        return

      # Fetch all email data in a single call
```

```
        fetch_status, fetch_response = mail.fetch(",".join(email_id.decode() for email_id in 
email_ids), '(RFC822)')

        if fetch_status!= 'OK':
          print("Error fetching emails.")
          return


        # Process emails concurrently using ThreadPoolExecutor
        with ThreadPoolExecutor(max_workers = 5) as executor:
          futures = []
           for response_part in fetch_response:
             if isinstance(response_part, tuple):
               num = response_part[0].split()[0]
               futures.append(executor.submit(process_emails, num, 
response_part,fetch_response, user, mail, server_email, server_pass, email_server, 
clf,whitelisted_domains))
          for future in as_completed(futures):
             future.result()  # To raise exceptions if any occurred during email processing

        print("Real Time Monitoring Module Executed")

    except Exception as e:
        print(f"An error occurred during email processing: {str(e)}")

def monitoring_instance(email, app_pass, mail_server, clf, whitelisted_domains, 
alert_server_email, alert_server_app_pass):
    while True:
        email_fetchncheck(email, app_pass, mail_server, clf, whitelisted_domains, 
alert_server_email, alert_server_app_pass)
        time.sleep(10)

def server_service():
    whitelisted_domains = set(pd.read_csv(r"whitelisted_domains.csv")["domain"])
    clf = joblib.load(r"lgb_model.pkl")
    #blacklist_pulldown.domain_db_update()
    #blacklist_pulldown.ip_db_update()

    credential_list  =  mongodb.fetch_credentials()

    # Create a list to store the subprocesses
    subprocesses = []

    # Iterate over the credentials
    for credential in credential_list:
```

```
    # Extract the email, app pass, alert server mail, and alert server app pass
    email, app_pass, alert_server_email, alert_server_app_pass = credential
    mail_server = server_fetch(email, app_pass)
    # Create a subprocess for the email_fetchncheck function
    p = multiprocessing.Process(target=monitoring_instance, args=(email,
app_pass,mail_server, clf, whitelisted_domains, alert_server_email,
alert_server_app_pass))

    # Start the subprocess
    p.start()

    # Add the subprocess to the list
    subprocesses.append(p)

  # Wait for all subprocesses to finish
  for p in subprocesses:
    p.join()
```

The `real_time.py` script is quite comprehensive, covering various aspects of email analysis, phishing detection, and certificate verification. Here's a brief overview of its functionalities:

Overview

1. **Authentication Functions**:
   - `get_mx_record(domain)`: Retrieves MX records for a given domain.
   - `server_fetch(email_address, password)`: Determines the IMAP server based on the email address.
2. **AI Model Functions**:
   - `get_tld_plus_one(domain)`: Extracts the TLD+1 from a domain.
   - `get_domain_age(domain)`: Calculates the age of the domain.
   - `get_domain_rank(domain)`: Retrieves the domain rank from Open PageRank.
   - `check_popular_keywords(domain)`: Checks for popular keywords in the domain.
   - `check_tld(domain)`: Verifies if the TLD is considered trustworthy.
   - `check_domain_reputation(domain)`: Evaluates domain reputation based on various factors.

- `extract_features(url)`: Extracts features from a URL for model prediction.
- `check_same_domain(url, sender_domain)`: Checks if URL domain matches the sender domain.
- `validate_and_clean_urls(url_list)`: Cleans and validates URLs.
- `extract_urls_from_email(email_content)`: Extracts URLs from email content.
- `get_sender_domain(msg)`: Retrieves the sender’s domain from the email.
- `get_sender_ip(headers)`: Extracts the sender’s IP from email headers.
- `ml_check_dmarc(domain)`: Checks DMARC records for the domain.
- `get_dns_info(domain)`: Fetches DNS A records for the domain.
- `calculate_phishing_score(reasons)`: Calculates a phishing score based on reasons.
- `classify_urls_in_email(email_content, whitelisted_domains, clf)`: Classifies URLs in email content using the model and provides reasons for phishing classification.

3. **Certificate Verification**:
   - `extract_certificate_from_eml(eml_content)`: Extracts certificates from email content.
   - `extract_sender_email(eml_content)`: Extracts the sender's email address.
   - `load_certificates(cert_data, cert_format)`: Loads certificates from data.
   - `verify_certificate_chain(certificates)`: Verifies the certificate chain.
   - `check_expiration(cert)`: Checks if the certificate is expired (incomplete implementation in the snippet).
   - `verify_email(cert, email_address)`: Verifies if the email address matches the certificate.
   - `validate_email_certificate(eml_content)`: Validates the certificate extracted from the email, checks expiration, and verifies the email address.

4. **SPF Verification:**

- `spf_verification(message)`: Checks the SPF record to determine if the email is coming from an authorized sender.
- `trusted_spf_verification(msg_data)`: Handles SPF verification for trusted domains.

5. **DMARC Check**:
   - `check_dmarc(message)`: Verifies if the DMARC policy of the sender domain is set to reject or quarantine.

6. **File Extension and Malware Check**:
   - `checkext_and_hash(filename, response_part)`: Checks if the file extension is blacklisted and calculates the hash for further malware analysis.

7. **Blacklist Checks**:
   - `check_ips_exist(ips_to_check)`: Checks if the IP addresses are in the blacklist.
   - `check_domain_exists(domain_to_check)`: Checks if the sender domain is in the blacklist.

8. **Processing Emails**:
   - `process_emails(email_id, msg_data, fetch_response, user, mail, server_email, server_pass, email_server, clf, whitelisted_domains)`: Coordinates the different checks and updates the email status.

5. whitelisted_domains.csv

`whitelisted_domains.csv` is a file that contains a list of domains considered safe or trusted, which should be excluded from spam or phishing detection processes. Here's a breakdown of its purpose and usage:

What It Is:

- **Content**: This CSV file lists various domain names, one per line, under the header `domain`.
- **Format**: The file is formatted with a single column of domain names.

Why It Is Used:

1. **Spam/Phishing Detection**:
   - **Exclusion from Filtering**: Domains listed in this file are trusted or whitelisted, meaning that emails from these domains should not be flagged as spam or phishing, even if they contain elements that would otherwise trigger alerts.
   - **Reducing False Positives**: By whitelisting trusted domains, you prevent legitimate emails from being incorrectly identified as threats.
2. **Email Analysis**:
   - **Domain Verification**: During email analysis, the system checks the sender's domain against this whitelist. If the domain is found in the list, the email may bypass certain security checks.
3. **Security Configuration**:
   - **Fine-Tuning**: Helps in configuring security rules more precisely, ensuring that only suspicious or malicious domains are flagged, and trusted ones are not mistakenly categorized as threats.

6. master_rsa.py

```
import mongodb_rsa as mongodb
import realtime_server as real_time

def server_menu():
  print("------Server-Menu ------ ")
  print("1. Admin Panel")
  print("2. Execute Server Monitoring")
  print("3. Exit")

def admin_panel():
  print("------Admin-Panel -------")
  print("1. Register User")
  print("2. Delete User")
  print("3. Update User")
  print("4. Exit")
```

```
def main():
  while True:
    server_menu()
    server_choice = int(input("Enter Choice : "))
    if server_choice == 1:
      while True:
        admin_panel()
        admin_choice = int(input("Enter Choice : "))
        if admin_choice == 1:
          mongodb.store_credentials()
        if admin_choice == 2:
          mongodb.remove_user()
        if admin_choice == 3:
          mongodb.update_user()
        elif admin_choice == 4:
          break
    elif server_choice == 2:
      real_time.server_service()

    elif server_choice == 3:
      print("Exiting. ")
      break

main()
```

This script provides a basic command-line interface (CLI) for managing a server, including admin functions and real-time server monitoring. It uses two modules: `mongodb_rsa` for handling MongoDB interactions related to user credentials and `realtime_server` for real-time server monitoring.

Script Breakdown

1. **Imports**:

```
import mongodb_rsa as mongodb import
realtime_server as real_time
```

- **mongodb_rsa**: Presumably handles MongoDB operations and RSA encryption for user credentials.
- **realtime_server**: Handles real-time server monitoring functionalities.

2. **server_menu() Function**:

```
def server_menu():
    print("------Server-Menu --------")
    print("1. Admin Panel")
    print("2. Execute Server Monitoring")
    print("3. Exit")
```

- Displays a menu with three options:
    i. Admin Panel: Manage user credentials.
    ii. Execute Server Monitoring: Start real-time monitoring.
    iii. Exit: Terminate the program.

3. **admin_panel() Function**:

```
def admin_panel():
    print("------Admin-Panel --------")
    print("1. Register User")
    print("2. Delete User")
    print("3. Update User")
    print("4. Exit")
```

- Provides a submenu for user management with four options:
    i. Register User: Add a new user.
    ii. Delete User: Remove an existing user.
    iii. Update User: Modify user details.
    iv. Exit: Return to the main server menu.

4. **main() Function**:

```
def main():
    while True:
        server_menu()
        server_choice = int(input("Enter Choice : ")) if
        server_choice == 1:
```

```
        while True:
            admin_panel()
            admin_choice = int(input("Enter Choice : ")) if
            admin_choice == 1:
                mongodb.store_credentials() if
            admin_choice == 2:
                mongodb.remove_user()
            if admin_choice == 3:
                mongodb.update_user()
            elif admin_choice == 4:
                break
    elif server_choice == 2: real_time.server_service()
    elif server_choice == 3:
        print("Exiting..... ")
        break
```

- **Loop**: Continuously displays the main server menu until the user chooses to exit.
- **Admin Panel**: If the user selects option 1, it enters a loop to show the admin panel menu until the user chooses to exit the admin panel.
  - **Register User**: Calls `mongodb.store_credentials()`, which presumably adds a user to the database.
  - **Delete User**: Calls `mongodb.remove_user()`, which presumably removes a user from the database.
  - **Update User**: Calls `mongodb.update_user()`, which presumably updates user details in the database.
- **Server Monitoring**: If the user selects option 2, it calls `real_time.server_service()`, which presumably starts real-time monitoring.
- **Exit**: If the user selects option 3, the program prints "Exiting... " and terminates.

5. **Execution**:

```
main()
```

    - Starts the program by calling the `main()` function.

Detailed Usage of the Imported Modules

- **mongodb_rsa**:
    - **store_credentials()**: Likely stores new user credentials in MongoDB, possibly with RSA encryption.
    - **remove_user()**: Likely deletes a user's credentials from MongoDB.
    - **update_user()**: Likely updates user credentials in MongoDB.
- **realtime_server**:
    - **server_service()**: Likely starts a real-time monitoring service, potentially to track system status, performance, or security events.

Summary

This script provides a simple CLI for managing server operations. It supports:

- **User Management**: Add, remove, or update user credentials.
- **Real-Time Monitoring**: Start a monitoring service to track server activity.
- **Exit**: Exit the application gracefully.

Detailed Explanation of Functions

1. Email Phishing Detection and Mitigation

- **Domain Reputation Analysis**
    - `get_domain_age(domain)`: Retrieves the age of a domain.
    - `get_domain_rank(domain)`: Fetches the rank of a domain via the Open PageRank API.
    - `check_popular_keywords(domain)`: Detects common phishing-related keywords in a domain.
    - `check_tld(domain)`: Verifies the trustworthiness of a domain's TLD.

    - `check_domain_reputation(domain)`: Aggregates domain age, rank, keywords, and TLD checks to determine domain reputation.
- **Blacklist and Whitelist Management**
    - `domain_db_update()`: Updates the domain blacklist repository.
    - `ip_db_update()`: Updates the IP blacklist repository.
- **URL and Email Content Processing**
    - `extract_features(url)`: Extracts features from a URL for machine learning classification.
    - `check_same_domain(url, sender_domain)`: Ensures URL domain matches the sender's domain.
    - `extract_urls_from_email(email_content)`: Extracts URLs from email content.
    - `get_sender_domain(msg)`: Extracts the sender's domain from an email.
    - `get_sender_ip(headers)`: Extracts the sender's IP address from email headers.
- **Email Authentication**
    - `check_spf(domain, sender_ip)`: Validates SPF records to ensure authorized email sending.
    - `verify_dkim(msg)`: Verifies DKIM signatures for email integrity.
    - `check_dmarc(domain)`: Checks DMARC records to validate domain policies.
    - `get_dns_info(domain)`: Retrieves DNS A records for a domain.
- **Phishing Score and Mitigation**
    - `calculate_phishing_score(reasons)`: Calculates a phishing score based on aggregated checks.
    - `classify_urls_in_email(mail, email_id)`: Analyzes URLs in emails and classifies them using machine learning.
    - `move_to_spam_and_notify(mail, email_id)`: Moves phishing emails to spam and notifies the user.
- **Inbox Processing**
    - `process_inbox()`: Processes the inbox to classify and manage emails based on phishing detection.

2. User Credential Management with MongoDB and RSA Encryption

- **MongoDB Integration**

    - `mongodb_rsa.py`: Manages interactions with the MongoDB database, including storing, verifying, and deleting user credentials.
    - **Setup**: Add the MongoDB connection string in the designated area of the script for secure database access.
- **RSA Encryption**
    - `private_key/public_key.pem`: Contains RSA keys used for encrypting and decrypting credentials.
    - **Encryption Process**: Credentials are encrypted before storage and decrypted when accessed, ensuring secure management.
- **Real-Time Monitoring**
    - `realtime_server.py`: Contains core functions for real-time email monitoring and phishing detection.
    - **Operation**: The module monitors incoming emails, classifies URLs, and takes necessary actions against phishing attempts.
- **Master Menu**
    - `master_rsa.py`: Acts as the master control interface for all functions in the application.
    - **Functionality**: Allows users to interact with different modules, including starting the phishing detection system and managing credentials.

Data Files

- **blacklisted_domains.csv**: List of blacklisted domains, regularly updated from trusted sources.
- **blacklisted_ips.csv**: List of blacklisted IPs, updated frequently.
- **cloudflare-radar-domains-top-100000-20240715-20240722.csv**: List of whitelisted domains from Cloudflare Radar.
    - **Source**: Cloudflare Radar
- **malicious_phish_cleaned.csv**: Dataset for training the machine learning model on phishing detection.
    - **Source**: Kaggle - Malicious URLs Dataset

Machine Learning Model

- **Model File**: The machine learning model is stored in a `.pkl` file, used for classifying URLs as phishing or legitimate.
    - **Download:** Model File - .pkl
    - **Integration:** Place the model file in the appropriate directory for the phishing detection system to function correctly.

API Integration

- **Open PageRank API:** Used to assess domain reputation as part of the phishing detection process.
    - **API Documentation:** Open PageRank Documentation

Setup Instructions

1. **Clone the Repository**

```
git clone <repository-url> cd
<project-directory>
```

2. **Install Dependencies**

```
pip install -r requirements.txt
```

3. **Update Configuration Constants**
    - **IMAP_SERVER**: Your IMAP server address.
    - **EMAIL:** Your email address.
    - **PASSWORD**: App-generated password for email.
    - Replace `'your_api_key'` with your Open PageRank API key.

4. **Add MongoDB Connection String**
    - Update the connection string in the `mongodb_rsa.py` file.

5. **Place the RSA Keys**
    - Store the `private_key/public_key.pem` in the designated directory.

6. **Download and Place the Machine Learning Model**
    - Ensure the `.pkl` file is in the correct directory for model loading.

7. **Ensure Data Files are Updated**
    - Place the required CSV files in the correct directories and ensure they are up to date.

Research and Development Process

The RCD process for this project involved a systematic approach to problem-solving, beginning with identifying the key challenges and progressing through the phases of research, design, development, and testing. The goal was to develop a comprehensive system that could detect phishing emails, analyze spam, and secure email credentials using encryption.

1. Problem Identification:

Phishing attacks and spam emails have become serious cybersecurity threats, with email systems being a primary target. The problem identification phase focused on understanding these vulnerabilities, the methods attackers use, and how emails can be monitored and analyzed for security threats.

- **Target Audience:** The system was designed for organizations and individual users who rely heavily on email for communication and are at risk of cyberattacks.
- **Security Requirements:** A real-time email monitoring system capable of detecting phishing emails and securing sensitive data using encryption was identified as a primary requirement.
- **Phishing and Spam Trends:** Through research, we found that phishing emails often contain deceptive URLs and malicious attachments. Spam, on the other hand, clutters inboxes and can be a medium for delivering malware.

2. Literature Review and Benchmarking:

A thorough literature review was conducted to understand the state-of-the-art solutions and tools available for email security. This step provided insights into existing models and frameworks used by email service providers.

- **Existing Solutions:** We reviewed the email security measures in platforms like Gmail, Outlook, and Yahoo, focusing on how they detect phishing emails and manage spam.
- **Machine Learning Techniques:** Machine learning algorithms such as Random Forest, Support Vector Machines, and Neural Networks were analyzed to find the most suitable model for phishing detection.
- **Encryption Protocols:** Research was conducted into cryptographic techniques like RSA to ensure the secure storage of user credentials in the database.

3. Conceptualization and Design:

Based on the problem identification and literature review, a high-level system architecture was conceptualized. The system would consist of several key modules:

- **Email Fetching Module:** Designed to retrieve emails in real-time from the user's inbox using the IMAP protocol.
- **Phishing Detection Module:** A machine learning-based system to analyze email contents, URLs, and attachments for phishing indicators.
- **Spam Management Module:** To detect and manage spam emails, storing them in a designated folder.
- **Encryption Module:** To securely store user credentials (email, app password) using RSA encryption.
- **Web Scraping Module (Future Enhancement):** A conceptualized feature that would open and analyze links from emails to check if they are safe.

The modular design ensured that new components could be easily added, and existing ones could be updated.

4. Algorithm Selection and Model Training:

For phishing detection, the **Random Forest** algorithm was chosen for its efficiency and ability to handle large datasets. Key features of URLs and email contents were identified and extracted for the training process.

- **Feature Extraction for Phishing Detection:**
    - URL length and hostname length.
    - Presence of suspicious words like "login," "verify," or "account."
    - Number of special characters such as "@" and "-".

The model was trained on a dataset that included legitimate and phishing URLs, and hyperparameter tuning was applied to maximize detection accuracy.

5. Iterative Development:

The system was developed using an iterative approach, with each module being independently tested before integration. Feedback loops allowed us to continuously improve functionality and security:

- **Phase 1:** Development of the email fetching system to connect with the IMAP server, retrieve emails, and extract attachments.
- **Phase 2:** Integration of phishing detection and URL analysis to analyze emails in real-time.
- **Phase 3:** Incorporating RSA encryption to store credentials securely in MongoDB.
- **Phase 4:** Integration of real-time monitoring to ensure that the system can run seamlessly and detect malicious activities as they occur.

C. Final Integration and Testing:

After all modules were developed, they were integrated into a unified system. Extensive testing was conducted to ensure the system performed efficiently in different scenarios:

- **Functional Testing:** Ensured that each module functioned correctly, including email retrieval, phishing detection, encryption, and spam management.
- **Performance Testing:** Checked the system’s ability to handle multiple emails simultaneously without performance degradation.
- **Security Testing:** Verified that user credentials were encrypted correctly and that the system could handle real-time threats without vulnerabilities.

Resources Utilized

1. Hardware Resources:

- **Development Machines:** Developers used standard laptops or desktops with sufficient processing power (e.g., 8GB RAM, multi-core processors) to run machine learning models and test the system.
- **Servers:** A secure Linux-based server was used to deploy the email monitoring system for testing and production.
- **Cloud Services (Optional):** AWS or Google Cloud could be used to host the database and ML models. For example, MongoDB Atlas (a cloud-based version of MongoDB) could be used to store user credentials securely.

2. Software Tools and Libraries:

- **Programming Languages:** Python was chosen for development due to its extensive library support, ease of use, and integration capabilities with machine learning and security tools.
- **Libraries and Tools:**
  - **IMAP and Email Handling:** `imaplib` and `email` libraries for fetching and parsing emails.
  - **Phishing Detection:** `Scikit-learn` for implementing the Random Forest model, along with `pandas` and `numpy` for data manipulation.
  - **Encryption:** `PyCryptodome` for RSA encryption and decryption.
  - **Web Scraping (Future):** BeautifulSoup for extracting and analyzing URLs.
- **Database:** MongoDB was selected to store user credentials and log email metadata. The MongoDB RSA encryption system was implemented to ensure secure data storage.

3. Human Resources:

- **Development Team:** Python developers with expertise in machine learning, email protocols, and cybersecurity.
- **Security Specialists:** Experts in encryption techniques, especially RSA, to ensure secure handling of sensitive data.
- **Quality Assurance and Testers:** A team responsible for testing the system’s functional and security aspects.

4. Data Resources:

- **Phishing Email Datasets:** Datasets containing labeled phishing and non-phishing emails and URLs, sourced from publicly available repositories or custom-built using phishing detection tools.
- **Genuine Email Samples:** Collecting legitimate emails for training the machine learning model and for testing the real-time detection mechanism.

Deployment Process

The deployment process ensures that the developed system is hosted in a secure, scalable environment and is accessible to users. Here’s the step-by-step deployment plan:

1. Pre-Deployment Steps:

- **Code Review and Refactoring:** The codebase underwent a final review for bugs, inefficiencies, and potential security vulnerabilities. Proper documentation was added to ensure maintainability.
- **Containerization (Optional):** The system could be containerized using Docker, ensuring that it can run in any environment without compatibility issues. This allows seamless migration between development, testing, and production environments.
- **Database Setup:** MongoDB was configured to securely store user credentials and email metadata. MongoDB Atlas (or an on-premise version) was set up with security features like IP whitelisting and SSL encryption.

2. Testing and Quality Assurance:

- **Unit Testing:** Each module (email fetching, phishing detection, encryption) was individually tested for accuracy and functionality.
- **System Testing:** An end-to-end test was conducted where emails were fetched, analyzed, and classified as phishing or legitimate, followed by encryption of user credentials.
- **Security Testing:** Testing focused on validating the RSA encryption and ensuring the security of data transmission between the application and the database.

- **Performance Testing:** Stress tests were performed to ensure that the system could handle real-time email monitoring without affecting performance, even under heavy loads.

3. Deployment to Production:

- **Server Deployment:** The system was deployed on a secure Linux server using Python scripts and necessary libraries. Either Apache or Nginx was configured to handle web traffic and manage the email monitoring system.
- **Database Hosting:** MongoDB was deployed on a secure server (or a cloud service like MongoDB Atlas) with encrypted connections between the app and the database.
- **API Integration (If Applicable):** If the system uses Gmail API for fetching emails, OAuth 2.0 credentials were set up and stored securely for authentication.

4. Monitoring and Maintenance:

- **Real-Time Monitoring:** The real-time server monitoring module was activated to monitor incoming emails for phishing and spam.
- **Error Handling and Alerts:** A logging mechanism was implemented to track errors, and notifications were configured to alert administrators in case of system failures or security threats.
- **Regular Updates:** The system was built with scalability in mind, allowing for easy updates such as improving the ML model or adding new security features.

Running the System

1. **Start the Phishing Detection and Credential Management System**

```
python master_rsa.py
```

2. **User Interaction**
    - Register and log in as a user.
    - Start the real-time email monitoring and phishing detection service.
3. **Monitor Logs**

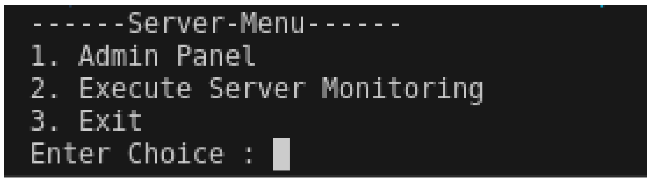
- Review logs generated for each processed email, noting any detected phishing attempts.

Testing and Evaluation

- **Functionality Testing**:
  - Test domain reputation checks, SPF/DKIM/DMARC validations, blacklist updates, and phishing score calculations.
  - Verify that phishing emails are correctly classified and moved to the spam folder.
- **Security Testing**:
  - Ensure credentials are securely encrypted using RSA before storage.
  - Validate email authentication checks and secure database communication.
- **Performance Evaluation**:
  - Measure system performance under heavy email loads.
  - Evaluate the speed and accuracy of the machine learning model in detecting phishing URLs.

Command-line interface (CLI) Images :

Main Menu (Server-Menu)



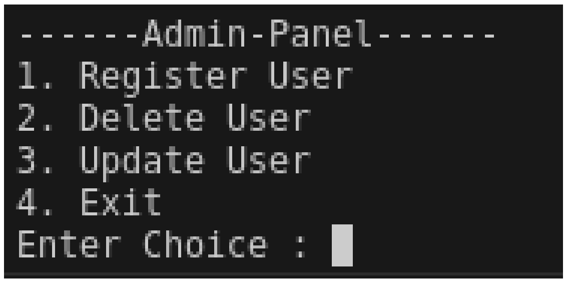
```
------Server-Menu------
1. Admin Panel
2. Execute Server Monitoring
3. Exit
Enter Choice : 
```

1. **Admin Panel**:
   - When the user selects option "1", the program will enter the **Admin Panel**. This is likely where administrative tasks related to the server are managed, such as registering, updating, or removing users. This panel may give access to critical management features and should be secured to prevent unauthorized access.
2. **Execute Server Monitoring**:
   - Option "2" triggers the **Server Monitoring** feature, which is probably a real-time monitoring service that checks the server for various operational

conditions like traffic, security, or resource usage. This could be linked to security, email monitoring, or phishing detection, depending on the system's purpose.

3. **Exit**:
   - Option "3" allows the user to exit the program safely. Once selected, the application will close the menu and terminate the running process.

Admin Panel Menu

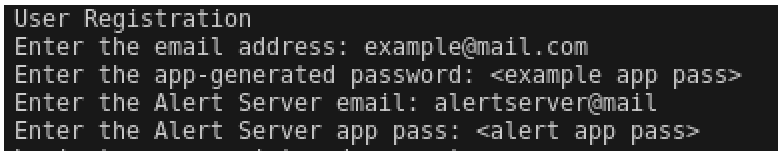


```
------Admin-Panel------
1. Register User
2. Delete User
3. Update User
4. Exit
Enter Choice :
```

1. **Register User**:
   - This option allows the administrator to add new users to the system. Typically, this would involve collecting and securely storing user credentials, such as a username, email, and password (possibly using RSA encryption as indicated in your previous code structure).
   - The function may call `mongodb.store_credentials()`, which could involve encrypting and storing the credentials in a MongoDB database.
2. **Delete User**:
   - Option "2" is used to remove users from the system. This action would likely involve finding the user in the database and permanently deleting their associated records, including any credentials.
   - In your integrated code, this would call `mongodb.remove_user()`, which ensures that all relevant details about the user are deleted.
3. **Update User**:
   - Selecting this option allows the administrator to modify a user’s details (e.g., username, email, or password). The system would fetch the existing data for the user, allow changes, and then update the MongoDB record accordingly.
   - This corresponds to `mongodb.update_user()` in your code, which enables a secure method of updating credentials.
4. **Exit**:

- Choosing this option exits the Admin Panel and returns to the **Server Menu**. It's a way to safely leave the admin functions without applying any changes.

User Registration

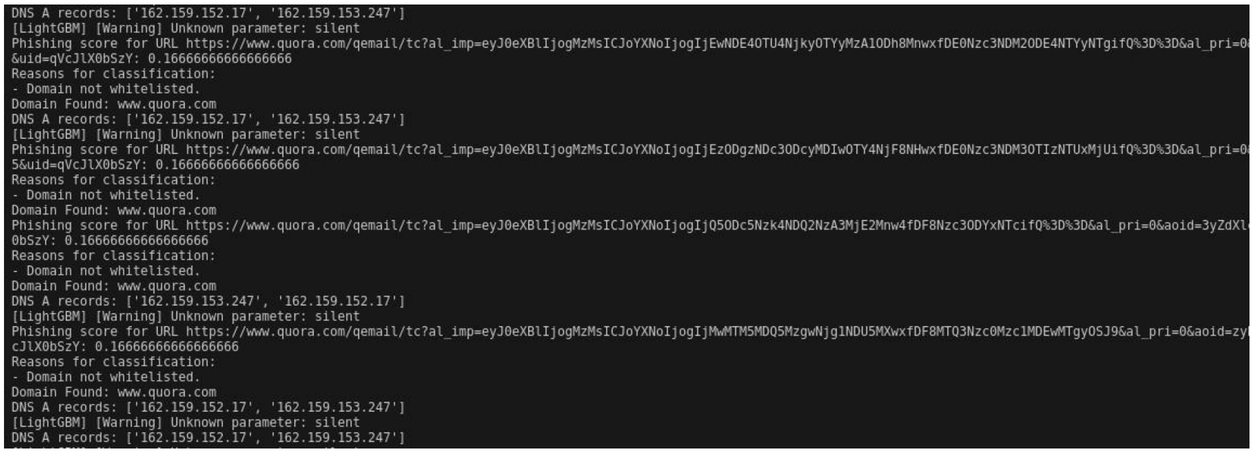


```
User Registration
Enter the email address: example@mail.com
Enter the app-generated password: <example app pass>
Enter the Alert Server email: alertserver@mail
Enter the Alert Server app pass: <alert app pass>
```

1. **Enter the Email Address:**
   - **Prompt:** `Enter the email address:`
   - The system asks the admin to input the email address of the new user being registered. This will likely serve as the primary identifier for the user in the system.
   - **Example Input:** example@mail.com
   - The system will store this information securely (e.g., in a MongoDB database) for future reference.
2. **Enter the App-Generated Password:**
   - **Prompt:** `Enter the app-generated password:`
   - The admin enters the password generated by the email service (e.g., Gmail). This password is typically created through application-specific password generation services (e.g., in Google's "App Passwords" feature), which allows the user to access their account without compromising their primary password.
   - **Example Input:** `<example app pass>`
   - **Background:** The password is stored securely, likely encrypted using RSA or another encryption method before it is stored in the MongoDB database.
3. **Enter the Alert Server Email:**
   - **Prompt:** `Enter the Alert Server email:`
   - The Alert Server email is where security notifications or system alerts (e.g., phishing attempts, suspicious activity) are sent. This allows the system to notify a security team or admin of any issues or suspicious activity detected.
   - **Example Input:** alertserver@mail.com
   - This field ensures that real-time alerts are being sent to a secure, designated email account that is monitored by the server administrators.
4. **Enter the Alert Server App Password:**

- **Prompt:** `Enter the Alert Server app pass:`
- Similar to the app-generated password for the user, this is a password specifically for the Alert Server email that is used by the system to send out email alerts.
- **Example Input:** `<alert app pass>`
- The system needs this password to authenticate and send alerts using the Alert Server email address.

Operation Breakdown:



```
DNS A records: ['162.159.152.17', '162.159.153.247']
[LightGBM] [Warning] Unknown parameter: silent
Phishing score for URL https://www.quora.com/qemail/tc?al_imp=eyJ0eXBlIjogMzMsICJoYXNoIjogIjEwNDE4OTU4NjkyOTYyMzA1ODh8MnwxfDE0Nzc3NDM2ODE4NTYyNTgifQ%3D%3D&al_pri=0
&uid=qVcJlX0bSzY: 0.16666666666666666
Reasons for classification:
- Domain not whitelisted.
Domain Found: www.quora.com
DNS A records: ['162.159.152.17', '162.159.153.247']
[LightGBM] [Warning] Unknown parameter: silent
Phishing score for URL https://www.quora.com/qemail/tc?al_imp=eyJ0eXBlIjogMzMsICJoYXNoIjogIjEzODgzNDc3ODcyMDIwOTY4NjF8NHwxfDE0Nzc3NDM3OTIzNTUxMjUifQ%3D%3D&al_pri=0
5&uid=qVcJlX0bSzY: 0.16666666666666666
Reasons for classification:
- Domain not whitelisted.
Domain Found: www.quora.com
Phishing score for URL https://www.quora.com/qemail/tc?al_imp=eyJ0eXBlIjogMzMsICJoYXNoIjogIjQ5ODc5Nzk4NDQ2NzA3MjE2Mnw4fDF8Nzc3ODYxNTcifQ%3D%3D&al_pri=0&aoid=3yZdXl
0bSzY: 0.16666666666666666
Reasons for classification:
- Domain not whitelisted.
Domain Found: www.quora.com
DNS A records: ['162.159.153.247', '162.159.152.17']
[LightGBM] [Warning] Unknown parameter: silent
Phishing score for URL https://www.quora.com/qemail/tc?al_imp=eyJ0eXBlIjogMzMsICJoYXNoIjogIjMwMTM5MDQ5MzgwNjg1NDU5MXwxfDF8MTQ3Nzc0Mzc1MDEwMTgyOSJ9&al_pri=0&aoid=zy
cJlX0bSzY: 0.16666666666666666
Reasons for classification:
- Domain not whitelisted.
Domain Found: www.quora.com
DNS A records: ['162.159.152.17', '162.159.153.247']
[LightGBM] [Warning] Unknown parameter: silent
DNS A records: ['162.159.152.17', '162.159.153.247']
```

- ***DNS A Records Resolution:***

The system retrieves the **DNS A records** for the domain associated with the URLs in the email. DNS A records map domain names to their corresponding IP addresses.
Example DNS A Records:
"162.159.152.17"
"162.159.153.247"

The system retrieves these IP addresses, possibly for additional checks, such as validating if these IPs belong to a trusted or suspicious domain.

- ***Phishing Score Calculation:***

The system is using a machine learning model, **LightGBM**, to calculate the **phishing score** for each URL.

The URLs in this case are from Quora's domain:
`https://www.quora.com/qemail/tc?al_imp=`...

The phishing score for the URL in the examples is **0.16666**, which indicates the probability that the URL could be a phishing URL (based on the model's training and threshold).

- ***Reasons for URL Classification:***

One of the main reasons the URL is flagged is that the **domain is not whitelisted**. Even though `quora.com` is a legitimate domain, the model is flagging it because the domain has not been added to the system's list of known safe domains.
Domain Found: The domain associated with the URL is `www.quora.com`.

This "not whitelisted" status may cause the URL to be treated with caution, leading to further scrutiny.

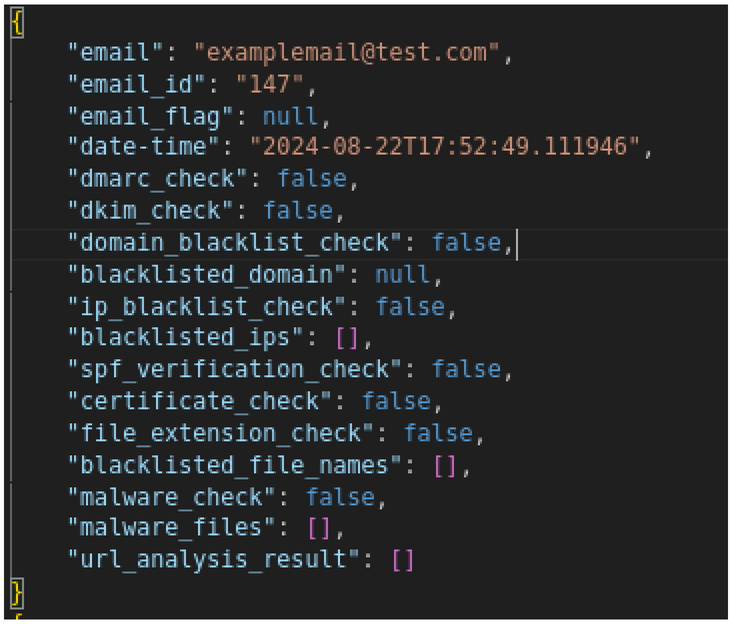
- ***Warnings from the LightGBM Model:***

You're receiving a **LightGBM warning** about an unknown parameter: `silent`.
This warning suggests that an incorrect or deprecated parameter (`silent`) is being used when calling the **LightGBM model**.
Resolution: Consider removing or replacing this parameter, as it might have been phased out in newer versions of LightGBM. The newer versions might use `verbosity` instead.

- ***Phishing Score Repeated Checks:***

Each of the URLs is being processed and checked multiple times, resulting in identical phishing scores across all instances: **0.16666666666666666**.
It's important to note that this relatively low score (0.1666) might indicate that the URL is not highly suspicious but still flagged for review due to the whitelist issue.

Log Example Breakdown:



```
{
    "email": "examplemail@test.com",
    "email_id": "147",
    "email_flag": null,
    "date-time": "2024-08-22T17:52:49.111946",
    "dmarc_check": false,
    "dkim_check": false,
    "domain_blacklist_check": false,
    "blacklisted_domain": null,
    "ip_blacklist_check": false,
    "blacklisted_ips": [],
    "spf_verification_check": false,
    "certificate_check": false,
    "file_extension_check": false,
    "blacklisted_file_names": [],
    "malware_check": false,
    "malware_files": [],
    "url_analysis_result": []
}
```

1. **Email:** examplemail@test.com
   - The email address being analyzed.
2. **Email ID:** 147
   - A unique ID assigned to this email for tracking.
3. **Date-Time:** 2024-08-22T17:52:49
   - When the email was analyzed by the system.
4. **DMARC Check:** `false`
   - The DMARC security check failed. DMARC helps confirm if the email is from a legitimate source.
5. **DKIM Check:** `false`
   - The DKIM check failed, meaning the system couldn't verify if the email is from an authorized sender.
6. **Domain Blacklist Check:** `false`
   - The email's domain is not blacklisted, so it's not marked as coming from a known bad source.
7. **IP Blacklist Check:** `false`
   - The sender's IP is not blacklisted, meaning the email isn't flagged as coming from a suspicious IP address.
8. **SPF Verification Check:** `false`

- The SPF check failed, which means the email might not be from an approved sender for the domain.

9. **Certificate Check:** `false`
   - The email does not have a valid security certificate.
10. **File Extension Check:** `false`
    - No risky or suspicious file extensions (like `.exe`) were found.
11. **Malware Check:** `false`
    - No malware was detected in the email or attachments.
12. **URL Analysis Result:** `[]`
    - No URLs in the email were flagged as dangerous.

Summary: The email has failed several security checks (DMARC, DKIM, SPF), which raises concerns about its legitimacy. However, no malware or suspicious URLs were found, and it isn't blacklisted. Further investigation might still be needed due to the failed authentication checks.

Integrated Email Analysis and Management System

Step 1: Setup and User Authentication

1. **User Input:**
   - Collect user credentials and details for fetching emails.

```
email = input("Enter your email address: ")
app_password = input("Enter your generated app password: ")
sender_email = input("Enter the sender's email address: ") date_str =
input("Enter the date to fetch emails (YYYY-MM-DD): ")
```

2. **Connecting to IMAP Server:**
   - Connect to Gmail's IMAP server to access emails.

```
import imaplib
import email
import hashlib
from bs4 import BeautifulSoup import
re
from urllib.parse import urlparse
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
```

```
import pickle
from google.oauth2.credentials import Credentials
from googleapiclient.discovery import build import
base64

imap_host = 'imap.gmail.com'
imap_port = 993
mail = imaplib.IMAP4_SSL(imap_host, imap_port)
mail.login(email, app_password)
```

Step 2: Fetch and Process Emails

1. **Fetching Emails:**
   - Search for emails from a specific sender on a particular date.

```
mail.select('inbox')
result, data = mail.search(None, f'(FROM "{sender_email}" ON "{date_str}")')
```

2. **Processing Emails:**
   - Extract attachments, compute their SHA-256 hash, and extract URLs from email content.

```
def calculate_sha256_hash(content):
    sha256_hash = hashlib.sha256()
    sha256_hash.update(content) return
    sha256_hash.hexdigest()

urls = []

for num in data[0].split():
    result, msg_data = mail.fetch(num, '(RFC822)') for
    response_part in msg_data:
        if isinstance(response_part, tuple):
            msg = email.message_from_bytes(response_part[1]) for
            part in msg.walk():
                content_disposition = part.get("Content-Disposition",
"")
```

```
                if "attachment" in content_disposition:
                    attachment_filename = part.get_filename() if
                    attachment_filename:
                        attachment_content =
part.get_payload(decode=True)
                        file_hash = calculate_sha256_hash(attachment_content)
                        print(f"The hash of the attached file
({attachment_filename}) is: {file_hash}")
                if part.get_content_type() == 'text/plain': text_content
                    =
part.get_payload(decode=True).decode('utf-8', errors='ignore')
                    urls.extend(re.findall(r'http[s]?://¥S+',
text_content))
                elif part.get_content_type() == 'text/html':
                    html_content =
part.get_payload(decode=True).decode('utf-8', errors='ignore')
                    soup = BeautifulSoup(html_content, 'html.parser')
                    urls.extend([a['href'] for a in soup.find_all('a',
href=True)])
```

Step 3: Phishing Detection

1. **Feature Extraction from URLs:**
    - Extract features from each URL to assess phishing risk.

```
def extract_features(url):
    parsed_url = urlparse(url)
    suspicious_words = ['login', 'verify', 'account', 'update', 'secure']
    return {
        'url_length': len(url),
        'hostname_length': len(parsed_url.hostname) if
parsed_url.hostname else 0,
        'num_special_chars': len(re.findall(r'[@¥-_.]', url)),
        'contains_suspicious_words': int(any(word in url.lower() for
word in suspicious_words))
    }
```

```
features_df = pd.DataFrame([extract_features(url) for url in urls])
```

2. **Phishing Detection:**
    - Classify URLs using a pre-trained Random Forest model.

```
with open('random_forest_model.pkl', 'rb') as model_file: clf =
    pickle.load(model_file)

predictions = clf.predict(features_df)
```

3. **Handling Phishing URLs:**
    - Move phishing emails to the spam folder.

```
def move_to_spam_and_notify(mail, num):
    mail.store(num, '+X-GM-LABELS', '¥¥Spam')
    print(f"Moved email {num} to spam folder.")

for num, is_phishing in zip(data[0].split(), predictions): if
    is_phishing == 'phishing':
        move_to_spam_and_notify(mail, num)
```

Step 4: Spam Management

1. **Gmail API Authentication:**
    - Authenticate with Gmail API using OAuth 2.0 credentials.

```
SCOPES = ['https://www.googleapis.com/auth/gmail.readonly']
creds = Credentials.from_authorized_user_file('token.json', SCOPES) service =
build('gmail', 'v1', credentials=creds)
```

2. **Fetching Spam Emails:**
    - Retrieve spam emails from the user's Gmail account.

```
results = service.users().messages().list(userId='me',
labelIds=['SPAM']).execute()
messages = results.get('messages', [])
```

3. **Saving Spam Emails:**
   - Save each spam email as a `.eml` file for further analysis.

```
for msg in messages:
    msg_id = msg['id']
    message = service.users().messages().get(userId='me', id=msg_id,
format='raw').execute()
    raw_data =
base64.urlsafe_b64decode(message['raw'].encode('ASCII'))
    file_path = f"spam_emails/{msg_id}.eml"
    with open(file_path, 'wb') as f:
        f.write(raw_data)
```

Summary

1. **Setup and Authentication:** Collect input and set up connections.
2. **Fetching and Processing Emails:** Retrieve and analyze emails.
3. **Phishing Detection:** Assess and handle phishing threats.
4. **Spam Management:** Manage and save spam emails.

Conclusion

This project has successfully developed a comprehensive email security system designed to detect and mitigate threats such as phishing attacks, spam, and unauthorized access. The system integrates several key functionalities, including email fetching, phishing URL detection, secure credential storage using RSA encryption, and real-time monitoring. By leveraging machine learning models and advanced email parsing techniques, the system provides a robust solution for safeguarding email communications.

Key features of the project include:

1. **Phishing URL Detection**: The system extracts URLs from emails, analyzes them using a pre-trained machine learning model, and identifies potential phishing attempts. Any detected phishing emails are automatically moved to the spam folder, protecting users from malicious links.
2. **Spam Management**: The system connects to the user's Gmail account and retrieves spam emails, saving them for further analysis. This helps in understanding and managing spam more effectively.
3. **Secure Credential Storage**: Using RSA encryption, the system securely stores user credentials in a MongoDB database, ensuring that sensitive information is protected from unauthorized access.
4. **Real-Time Monitoring**: The system continuously monitors incoming emails, processes them in real-time, and takes appropriate actions based on the analysis results. This proactive approach helps in promptly addressing security threats.

Overall, this project has successfully implemented a multi-layered approach to email security, combining real-time monitoring, machine learning, and encryption to provide comprehensive protection against various email-based threats.

Future Enhancements

While the current system provides robust email security, there are several areas where it can be further enhanced to improve its effectiveness and user experience:

1. **Improvement of the Machine Learning Model**:
   - **Feature Enhancement**: The current model can be improved by incorporating additional features that capture more subtle indicators of phishing. For example, analyzing the content of the email body, headers, and sender's reputation can provide more context for classification.
   - **Dataset Expansion**: Training the model on a larger and more diverse dataset will improve its ability to generalize across different types of phishing attempts. This will reduce the rate of false positives and increase the detection accuracy.
   - **Model Tuning**: Fine-tuning the hyperparameters of the machine learning model, such as the number of trees in the Random Forest or the depth of each tree, can optimize its performance for this specific use case.
2. **Web Scraper Integration**:

- **Automated Link Analysis**: Adding a web scraper to the system will allow it to automatically open links found in emails and analyze the content of the destination websites. This step will help verify the legitimacy of the websites and ensure that users are not redirected to malicious sites.
- **Content Verification**: The web scraper can check for common indicators of phishing websites, such as fake login pages, unusual domain names, and mismatched SSL certificates. If any suspicious elements are detected, the email can be flagged or moved to spam.
- **User Notification**: Users can be notified of the results of the link analysis, giving them the option to proceed with caution or avoid interacting with the potentially dangerous site.

3. **GUI Integration**:
    - **User-Friendly Interface**: Integrating the system with a graphical user interface (GUI) will make it more accessible to non-technical users. They will be able to manage their email security settings, view reports, and take actions without needing to use command-line tools.
    - **Dashboard Overview**: A GUI can provide a dashboard that displays real-time statistics on email security, such as the number of phishing attempts detected, spam emails processed, and user credentials secured. This overview will help users quickly assess the security status of their email accounts.
    - **Interactive Controls**: Users will be able to interact with the system more intuitively through buttons, sliders, and other controls. For example, they could easily whitelist or blacklist certain email addresses, adjust the sensitivity of the phishing detection model, or schedule regular scans.

These future enhancements will not only improve the technical capabilities of the system but also make it more user-friendly and adaptable to the evolving landscape of email security threats. By continuously refining and expanding the system, it can remain a powerful tool for protecting users from increasingly sophisticated email-based attacks.