Project 4: White-Hat Hacker Audit

Introduction: You've been hired by NeonNet Technologies, a leading financial powerhouse, to identify and evaluate potential vulnerabilities in their digital banking systems. Threats from black-hat hacker collectives loom large, and your white-hat expertise is their shield against these shadow adversaries.

Objective: You have been handed a ZIP archive containing encrypted banking files. Your mission involves several stages: exploring the file structure of the archive, decrypting the files using a wordlist, parsing the decrypted content, validating the structure, conducting a security audit of the data, and finally re-encrypting the data using a new key. Each file's original encryption key is crafted by pairing two words from the wordlist.

Project Steps:

1. Setup:

- **ZIP Archive:** You are provided with a ZIP archive named *bank_data_archive.zip*. You can find this in the course files page. This archive shelters encrypted text files possibly nestled within a multi-layered directory structure.

- **Wordlist:** A standalone text file brimming with potential password combinations. Remember, the decryption key for each banking file is a combination of two words from this list.

2. File Exploration & Extraction:

- Manually unzip the *bank_data_archive.zip* and move its contents into a dedicated directory in your workspace. Be mindful of its location, as you'll reference it in your program.

- Utilize Python's **os** module to systematically navigate the directory structure of the extracted archive. Your goal is to traverse directories and sub-directories to pinpoint the encrypted files. Generating a list of directories.

- As you comb through the directory structure, be attentive to directories or files donning unconventional naming patterns. Such entities might bear significance or hold additional information.

3. File Operations & Exception Handling:

- Load the wordlist into a Python list.

- Endeavor to load the encrypted banking data from the identified paths. Implement **try** and **except** blocks to aptly manage potential file-related hiccups.
    - FileNotFoundError: Instances where a file is missing or the path is incorrect.
    - PermissionError: Simulated situations where a file might be unreadable.
    - Custom exceptions for undecryptable files.

- Design a logging mechanism to chronicle which file or directory triggered a particular exception.

4. Decryption:

- Each character of the encrypted data can be decrypted with some two word combination of the word list. Every file might have a different key or no key.

- For each two-word combination from the wordlist, calculate a decryption key. The key is the sum of the ASCII values of every character in the combined word pair.

- For example, if the two words are "apple" and "banana", the key would be the sum of ASCII values of "applebanana".

- For every character in the encrypted content, decrypt it by subtracting the previously calculated ASCII sum (the key). If the result is negative, loop it back into the range of valid ASCII values.

- Continue this process for the entire content of the file with the given key.
- For each identified banking file, iterate over the two-word combinations from the wordlist, experimenting with each duo as a potential decryption key.

- Upon successful decryption, you should unveil dictionaries teeming with user banking details.

5. Parsing and Validation:

- Translate the decrypted content into a structured and usable format fit for subsequent examination.

- Construct a function to:
    - Verify if the content aligns with the anticipated banking data format.
    - Confirm the presence of requisite keys.
    - Authenticate the data type of the values.

- Deploy this function to evaluate the content of each decrypted file, logging any discrepancies or structural deviations.

6. Security Analysis:

- **Password Strength Probe:** Measure each password against known security yardsticks. Acceptable passwords should:
    - Be at least 8 characters long.
    - Contain both upper and lower case characters.
    - Include at least one numeric digit.
    - Incorporate at least one special character (e.g., @, #, $, etc.).
    - Examples of weak passwords: password, 12345678, abcd1234. An example of a strong password: P@ssw0rd!.

- **Data Integrity Check:** Ascertain the completeness of data fields for each user or account. Missing data might manifest as, but not limited to:

- Empty fields: e.g., "address": "".
- Null values: e.g., "phone_number": null.
- Placeholder texts: e.g., "email": "not_provided".
- Duplicate Account Detection: Highlight potential duplicate accounts based on distinct attributes such as:
- Same email address.
- Identical account numbers.
- Matching phone numbers.

7. Re-Encryption:

- Once the analysis is wrapped up, re-encrypt the curated and validated data employing a distinct encryption key into new files. Do not overwrite the original files.

8. Statistics Generation:

- For each file and total population:
- Count and calculate the percentage of passwords that do not meet the security criteria.

- Count and calculate the percentage of accounts with missing data.

- Count and calculate the percentage of potential duplicate accounts.

- Count and calculate the percentage of files that could not be decrypted.

9. Final Report:

- Compose a report that encompasses:
    - The overall number of accounts and files reviewed.
    - A thorough breakdown of the findings and statistics.
    - Challenges encountered during the decryption phase.
    - Practical recommendations rooted in the analysis.

*** Bonus Task:**

Data Cleaning: Delve deeper into the parsed data, pinpointing and rectifying inconsistencies or anomalies.

Submit on Canvas: Your python code and report

Banking Data Decryption Rubric (100 points)

File Exploration & Management (30 points)

Correct extraction and unzipping of provided archive to appropriate directory.
Accurate traversal of directories to locate encrypted banking files.
Proper use of the os library to manage files and directories.

Decryption Logic & Keyword Combinations (30 points)

Accurate calculation of ASCII sum for the two-word combinations.
Successful decryption of banking files using the provided wordlist.
Correct identification and extraction of valid decrypted banking data.

Data Validation & Analysis (20 points)

Proper identification of valid passwords based on given criteria.
Accurate identification of missing data fields.
Correct detection of duplicate accounts and data entries.
Code Organization & Readability (10 points)

Logical code structure with clear separation between decryption, data processing, and file handling functionalities.
Consistent naming conventions and proper use of whitespace and indentation.

Comments (5 points)

Essential code sections and functions are appropriately commented on.
Comments provide clarity and meaningful insights into code's functionality.

Testing (5 points)

Well-defined test cases that cover typical scenarios and edge cases.
Annotations explaining the purpose of each test case and expected outcomes.

Bonus: Efficient Data Cleaning (5 points)

Advanced methods used for tidying up the decrypted banking data.
Handling anomalies and irregularities beyond basic requirements.

Bonus: Advanced Data Analysis (5 points)

Integration of additional analysis methods beyond basic requirements, such as trend detection, account activity patterns, or other insightful statistics.