Encryption Assignment

Overview

The purpose of this assignment is to understand how to properly handle encryption and decryption. This task isn't simply a certain algorithm applied to a plain text with a certain size key, but rather a step-by-step process that ensures robustness against many different exploits, including keystream reuse, padding oracles, and collisions/small sized keys that lead to a smaller work factor (although the social engineering window is still wide open).

This document will go over a statement of AI, then the approach of the program, then a step-by-step walkthrough of the output, then a conclusion section outlining overall thoughts, then finally a snippet of the conversation with generative AI

Statement of AI

This statement details how generative AI was used to complete this assignment. The LLM of use was Microsoft Copilot, specifically the desktop application. The use case of the tool was to learn about the available cryptography Python packages and how they are used, whether this be through asking for specific methods or through generating tailored examples. For example, the chatbot was asked to show how using the chosen cryptography package will allow for encrypting a file using AES. Based on the response, the corresponding methods were inserted and the parameters were changed to fit existing parameters that will come from user utility.

Generative AI was not used to complete assignment specifications directly or write any documentation or comments.

A snippet of the conversation with Microsoft Copilot with regards to this assignment can be found at the end of this document.

Approach

The language of choice for this assignment is Python not only for its user friendly syntax but also for its ease-of-access to useful packages/imports (such as cryptography). The general flow of the program is that the user will be prompted on the command line to specify if they want to encrypt or decrypt a file. From there, a series of prompts asking for choice of parameters (password, text file, encryption algorithm, hashing algorithm, and iteration count) will be given before encryption takes place. Once this is complete, the program tells the user which binary file has their text file been encrypted to.

For decryption, the user will need to run the program again and choose the decryption option. From there, they will need to enter the binary file they want to decrypt and the corresponding password for it. If there are no issues, the program will decrypt and output the plaintext.

In both cases, the encryption and decryption algorithms will first process and set the required parameters to encrypt/decrypt the given file. Once completed, the master key is generated through the password and the HMAC and Encryption keys are generated from the master key. Next, if the action is to encrypt, the file is read as binary and a HMAC tag is created from the ciphertext plus IV. If the action is to decrypt, the HMAC tag is regenerated from the ciphertext and IV and is verified before the file is decrypted. Finally, the information is saved into the corresponding files based on the action.

When it comes to error handling, input sanitization for encryption is done during the user utility section to prevent calling the encryption algorithm with faulty parameters. For decryption, however, any change to any part of the binary file (whether it be meta data, HMAC tag, or ciphertext) will result in a different HMAC tag, which will cause the program to exit.

Step-by-Step

This section will go through a step-by-step process of the program and how it runs.

First, a text file to be encrypted is created (example.txt)

example.txt

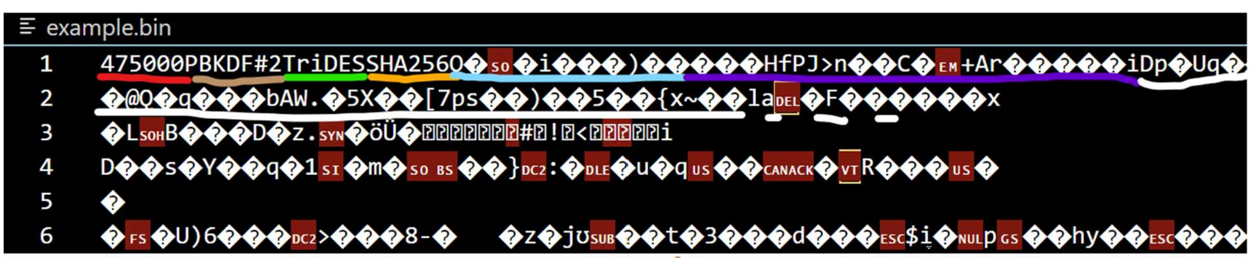
```
If you want to optimize for time, higher level
languages will be easier - C# (.NET Core is
cross-platform) has a PBKDF implementation that allows
making the hash algorithm an argument. Last time I
looked, Java only had sha1 PBKDF, but that was 2 years
ago and could be fixed by now. JavaScript and Python
have also been successfully used. As stated in class,
be warned that a string object may well be different
than a character array, and this can have implications
for encryption, especially in higher level languages.
See my padding oracle code for an example of this.
```

Next, the program is run and the parameters are entered for encryption

```
Would you like to encrypt or decrypt? (Enter E for encrypt or D for decrypt):

E

Enter your password:

Pa$$W0rD

Enter the name of the text file to encrypt (remember to include the .txt):

example.txt

Choose your encryption algorithm
TripleDES (enter 1), AES128 (enter 2), AES256 (enter 3)

2

Choose your SHA hashing algorithm
256 (enter 1) or 512 (enter 2)

1

Choose the number of iterations (enter a whole positive number, or enter 0 for defaults)

0

File has been encrypted into the following file: example.bin
```

Here, AES-128 is chosen with a combination of SHA-256. While the user can specify their own iteration count, they can also use default numbers. For each encryption algorithm, the corresponding iteration count is provided: Triple DES: 475,000; AES-128: 550,000; AES-256: 450,000. Each one was chosen based on performance to about a second.

Once completed, example.bin holds the encrypted data and meta data.



Red: Iteration count. Brown: KDF used. Green: Encryption algorithm. Orange: Hashing algorithm. Blue: Salt (roughly). Purple: HMAC Tag. White: IV and ciphertext

To decrypt, the program is invoked again, and the decryption option is chosen. Then the binary file and the corresponding password are entered. If successful, the plaintext is saved to a corresponding file with a prefix of “decrypted”.

```
Would you like to encrypt or decrypt? (Enter E for encrypt or D for decrypt):

D

Enter the name of the binary file to decrypt (remember to include the .bin):

example.bin

Enter your password:

Pa$$W0rD

HMAC verification succeeded.
```

```
decrypted_example.txt
1    If you want to optimize for time, higher level languages will be easier - C# (.
     NET Core is cross-platform) has a PBKDF implementation that allows making the
     hash algorithm an argument. Last time I looked, Java only had sha1 PBKDF, but
     that was 2 years ago and could be fixed by now. JavaScript and Python have also
     been successfully used. As stated in class, be warned that a string object may
     well be different than a character array, and this can have implications for
     encryption, especially in higher level languages. See my padding oracle code for
     an example of this.
```

There are three cases where decryption fails: 1) The meta data is changed. 2) The incorrect password is entered. 3) The iv and ciphertext are tampered with. The following two screenshots show case 3.


```
Would you like to encrypt or decrypt? (Enter E for encrypt or D for decrypt):

D

Enter the name of the binary file to decrypt (remember to include the .bin):

example.bin

Enter your password:

Pa$$W0rd

HMAC verification failed! The data may have been tampered with or the incorrect password may have been used.
```

Conclusion

Overall, this assignment was interesting to complete as I felt that it touched on cryptography aspects that I have not explored before. For context, I have used encryption before to secure peer-to-peer mobile communication as a capstone project, but in that project, I used a combination of RSA and ECDH to achieve encryption. This project more resembles encrypting data at rest rather than communication data.

AI Conversation

ME:

can you give me an example of creating a master key with the cryptography package?

Copilot:

Certainly! Here's an example of how you can create a master key using the `cryptography` package with PBKDF2 for key derivation:

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import KeyDerivationFunction
from cryptography.hazmat.primitives.kdf.pbkdf2 import
KeyDerivationFunction_BASE
# Import other modules
import os
# Encrypt API as AES
```

Remember, you can adjust the salt and iteration count to balance security and performance. Let me know if you'd like to dive deeper into any aspect of this code or explore file encryption next!

ME:

can you give me a code example that uses KeyDerivationFunction

Copilot:

It seems there was a bit of confusion—`KeyDerivationFunction` itself is more of a conceptual placeholder rather than a directly importable or usable class in Python's `cryptography` library. However, the `cryptography.hazmat.primitives.kdf` module provides practical key derivation functions like `PBKDF2HMAC` that are frequently used.

Here's an updated example using `PBKDF2HMAC` as a key derivation function to derive a secure key:

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
from cryptography.hazmat.backends import default_backend
import os
import base64
# Define parameters
password = b"strong_password"  # Password in bytes
salt = os.urandom(16)          # Generate a random salt
iterations = 100_000           # Number of iterations
# Create PBKDF2HMAC key derivation function
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,  # Desired key length in bytes
    salt=salt,
    iterations=iterations,
    backend=default_backend()
)
# Derive a key
key = kdf.derive(password)
# Encode the key in a readable format (optional)
key_base64 = base64.b64encode(key).decode()
# Output the generated key
print(f"Derived key (Base64): {key_base64}")
```

This script derives a cryptographic key securely from a password using PBKDF2HMAC, ensuring the result is resistant to brute-force attacks.

Would you like to explore further, such as encrypting files or using alternative key derivation functions like `Scrypt`? Let me know!

ME:

What other practical key derivation functions does cryptography.hazmat.primitives.kdf.pbkdf2 have?

Copilot:

In the `cryptography` library, the key derivation functions provided in the `cryptography.hazmat.primitives.kdf` module include:

1. **PBKDF2HMAC**: Password-Based Key Derivation Function 2 with HMAC.

2. **Scrypt**: A memory-hard key derivation function resistant to hardware-based attacks.

3. **HKDF**: HMAC-based Extract-and-Expand Key Derivation Function.

4. **ConcatKDF**: Concatenation Key Derivation Function.

5. **X9.63KDF**: Defined by ANSI X9.63 for elliptic curve cryptographic protocols.

Let me know if you'd like to see examples for any of these!

ME:

with cryptography, how can i use pbkdf#2 with AES-128, AES-256, and 3DES?

Copilot:

To use PBKDF2 for key derivation and encrypt data with AES-128, AES-256, or 3DES using the `cryptography` library, you'll need to combine PBKDF2HMAC (for key derivation) with a cipher (for encryption). Below is an example implementation for each encryption method.

AES-128 Example

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import os
# Parameters
password = b"your_password"
salt = os.urandom(16)
iterations = 100_000
# Derive a 16-byte key for AES-128
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=16,
    salt=salt,
    iterations=iterations,
    backend=default_backend()
)
key = kdf.derive(password)
# Encryption using AES-128
iv = os.urandom(16)  # Initialization vector
cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default_backend())
encryptor = cipher.encryptor()
ciphertext = encryptor.update(b"Your data here") + encryptor.finalize()
print("AES-128 Encrypted Data:", ciphertext)
```

AES-256 Example

For AES-256, you only need to increase the `length` parameter in PBKDF2 to 32 bytes:

```
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,  # 32 bytes = 256 bits
    salt=salt,
    iterations=iterations,
    backend=default_backend()
)
key = kdf.derive(password)
# Rest of the AES encryption code remains the same
```

3DES Example

For 3DES, the key length should be 24 bytes:

```
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=24,  # 24 bytes for 3DES
    salt=salt,
    iterations=iterations,
    backend=default_backend()
)
key = kdf.derive(password)

# Encryption using 3DES
cipher = Cipher(algorithms.TripleDES(key), modes.CFB(iv),
backend=default_backend())
encryptor = cipher.encryptor()
ciphertext = encryptor.update(b"Your data here") + encryptor.finalize()
print("3DES Encrypted Data:", ciphertext)
```

Notes:

1. **Salt**: Always use a unique, random salt for each password. Store the salt securely alongside the ciphertext for decryption.

2. **Initialization Vector (IV)**: Ensure the IV is random and unique for every encryption session.

3. **Key Lengths**: AES-128 requires a 16-byte key, AES-256 requires a 32-byte key, and 3DES requires a 24-byte key.

Let me know if you'd like to dive deeper into decryption or have any questions!