CYBR372

Applications of Cryptography

Assignment 1 : Applications of Cryptography

Gayle Wootton
300578290

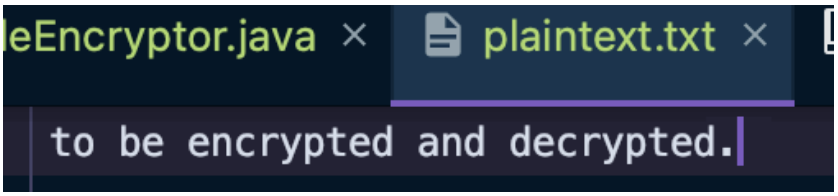
Part 1 - Symmetric File Encryption and Decryption

Performing symmetric encryption and decryption using AES- advanced encryption standard algorithm. In symmetric encryption, we use the same key when encrypting and decrypting the given files. A particular vulnerability with this method of symmetric encryption is the need to keep both keys secret. To further explain how part1 of my FileEncryptor program functions i will provide an explanation:

The program first checks if the user has provided enough arguments before the program is able to encrypt or decrypt. This was a change implemented by my program to allow command line arguments to perform either encryption or decryption. Depending on the operation mode specified by the user- either encryption or decryption symbolised by either 'enc' or 'dec' within the arguments.

- If 'enc' is specified it means it is in encryption mode. The program generates a 16 byte key and IV. This key and IV for AES encryption are printed on the console encoded in base64 for human readability (to also be re-used for decryption of that file). Arguments should contain the input file to be encrypted, saving the resultant encryption to an output file. The input file is encrypted byte by byte, writing the encrypted bytes to the specified output file.
- If 'dec' is specified, the arguments must contain the base64 key and IV in addition to the encrypted file and the output (decrypted) file. Otherwise, if either keys are not identical to the keys utilised in the encryption mode, the result of this decryption will not be successful.

Successful Encryption and Decryption Example:
First creating a plaintext.txt as the input file with the following contents:



Compile the program with "javac FileEncryptor.java":

```
gayleanto@MacBook-Pro-10 part1 % javac FileEncryptor.java
```

For **Encryption** pass "java FileEncryptor.java enc plaintext.txt ciphertext.enc". Provided arguments are appropriate, both hex and base64 keys are printed onto the console:

```
gayleanto@MacBook-Pro-10 part1 % java FileEncryptor.java enc plaintext.txt ciphertext.enc
Generated Key: EB 51 7D 0B 0A 6A 97 E5 90 34 9E DC 62 43 36 99
Generated IV: 58 7C B4 BB 05 FB 72 D7 F3 6B C3 6C 5B 04 23 CA
Base 64 Generated Key: 61F9Cwpql+WQNJ7cYkM2mQ==
Base 64 Generated IV: WHy0uwX7ctfza8NsWwQjyg==
Aug 17, 2023 12:32:08 PM part1.FileEncryptor main
INFO: Process complete, output saved at ciphertext.enc
```

Review the encrypted bytes of the contents within the plaintext.txt file to the ciphertext.enc file. You can see that encryption was successful:

```
gayleanto@MacBook-Pro-10 part1 % cat plaintext.txt
to be encrypted and decrypted.%
gayleanto@MacBook-Pro-10 part1 % cat ciphertext.enc
[illegible]
```

For **Decryption** pass in “java FileEncryptor.java dec (base64 key, in this case:) 61F9Cwpql+WQNJ7cYkM2mQ== (base64 IV, in this case:) WHy0uwX7ctfza8NsWwQjyg== ciphertext.enc plaintext.txt”:

```
gayleanto@MacBook-Pro-10 part1 % java FileEncryptor.java dec 61F9Cwpql+WQNJ7cYkM2mQ== WHy0uwX7ctfza8NsWwQjyg== ciphertext.en
c plaintext.txt
Aug 17, 2023 12:40:05 PM part1.FileEncryptor main
INFO: Process complete, output saved at plaintext.txt
```

To ensure decryption was successful, check the result of the plaintext.txt. If successful, the file should still have the same, readable contents:

```
gayleanto@MacBook-Pro-10 part1 % cat plaintext.txt
to be encrypted and decrypted.%
```

Unsuccessful decryption if either keys are not the same keys during encryption:

```
gayleanto@MacBook-Pro-10 part1 % java FileEncryptor.java dec 61F9Cwpql+WQNJ7cYkM2ww== WHy0uwX7ctfza8NsWwQjyg== ciphertext.en
c plaintext.txt
Aug 17, 2023 12:45:45 PM part1.FileEncryptor main
INFO: Process complete, output saved at plaintext.txt
```

This will result in the file still being encrypted:

```
gayleanto@MacBook-Pro-10 part1 % cat plaintext.txt
f[illegible]Fm[illegible]V[illegible]
[illegible]8[illegible]%
```

Part 2: **Demonstrate Chosen-plaintext Attack (CPA) Security**

The design of the FileEncryptor class, which uses symmetric block cipher AES, provides chosen-plaintext attack (CPA) security through the use of an initialisation vector (IV) and secret key.

The secret key used in the algorithm is specified as a base 64 string to ensure confidentiality. Only those who possess the secret key can successfully decrypt the files. The key is randomly chosen and is not hard-coded or stored in the code which makes it less susceptible to theft.

The Initialisation Vector (IV) adds randomness to the process and ensures that the same plaintext blocks produce different ciphertext blocks, even when the same key is used. This property thwarts CPA as it makes it impossible for an attacker to draw conclusions about the plaintext from the ciphertext. In this design, for every file encryption, a new random IV is generated. IV in this case is stored at the beginning of the encrypted file and not kept secret. It is important to mention that IV does not need to be secret for the system to be secure. Its main purpose is to provide block diversification, and IV plays the role of salt in this case. It only needs to be unpredictable and unique for each encryption operation with the same key. The CBC mode combined with

padding scheme PKCS5, is used which ensures that even if the attacker knows the IV, they cannot predict the plaintext without knowing the key. This makes it more resilient against attacks. Using a randomly generated secret key and IV for each encryption, combined with CBC mode and an appropriate padding scheme, makes the system secure, as it prevents patterns in the plaintext from appearing in the ciphertext, thereby securing the design against chosen-plaintext attacks.

The program, similarly to part1, checks to see if the correct arguments have been passed through for both encryption and decryption modes. Albeit in part2, to run the encryption, 4 arguments need to be passed- mode(enc), key (base64), input file, and output file. The key is required for both encryption and decryption in this case- though for decryption, the IV does not need to be passed as an argument. In part1 the random key and IV were generated during encryption, in this case a secure random IV is generated only for encryption and then written to the output file. For decryption, IV is read from the input file.

Successful Encryption and Decryption:
Create the plaintext.txt file in the same directory as the program.
First compile the program “javac FileEncryptor.java”:

```
gayleanto@MacBook-Pro-10 part2 % javac FileEncryptor.java
```

For **encryption,** pass in enc mode, base64 key, input file (plaintext.txt) and output file (ciphertext.enc):

```
gayleanto@MacBook-Pro-10 part2 % java FileEncryptor.java enc I1S/VnrYxUdkGawSgLAb8g== plaintext
.txt ciphertext.enc
Aug 17, 2023 1:07:25 PM part2.FileEncryptor main
INFO: Process complete, output saved at ciphertext.enc
```

The result of the plaintext and ciphertext files:

```
gayleanto@MacBook-Pro-10 part2 % cat plaintext.txt
to be encrypted and decrypted :)
gayleanto@MacBook-Pro-10 part2 % cat ciphertext.enc
ó&dŭ7-ZW :R*M f n+{1
                    6
                     L5Y&k%
```

For Decryption, very similar to encryption, pass in **dec** mode, base64 key (the same key used for encryption), input file (plaintext.txt) and output file (ciphertext.enc). If all arguments are correct, the result should be:

```
gayleanto@MacBook-Pro-10 part2 % java FileEncryptor.java dec I1S/VnrYxUdkGawSgLAb8g== ciphertex
t.enc plaintext.txt
Aug 17, 2023 1:41:01 PM part2.FileEncryptor main
INFO: Process complete, output saved at plaintext.txt
gayleanto@MacBook-Pro-10 part2 % cat plaintext.txt
to be encrypted or decrypted :)%
```

To demonstrate what would happen if the wrong key was passed, the file would be encrypted:

```
gayleanto@MacBook-Pro-10 part2 % java FileEncryptor.java dec I1S/VnrYxUdkGawSgLAwww== ciphertex
t.enc plaintext.txt
Aug 17, 2023 1:42:43 PM part2.FileEncryptor main
INFO: Process complete, output saved at plaintext.txt
gayleanto@MacBook-Pro-10 part2 % cat plaintext.txt
◆◆◆N◆◆◆@◆iJ◆◆6)%
```

To demonstrate that the ciphertext is different each time i used hexdump to showcase this:
First, I encrypted the file and hexdump the encrypted file:

```
gayleanto@MacBook-Pro-10 part2 % java FileEncryptor.java enc I1S/VnrYxUdkGawSgLAb8g== plaintext
.txt ciphertext.enc
Aug 17, 2023 1:46:14 PM part2.FileEncryptor main
INFO: Process complete, output saved at ciphertext.enc
gayleanto@MacBook-Pro-10 part2 % hexdump -b ciphertext.enc > dump1.txt
```

I performed this a second time:

```
gayleanto@MacBook-Pro-10 part2 % java FileEncryptor.java enc I1S/VnrYxUdkGawSgLAb8g== plaintext
.txt ciphertext.enc
Aug 17, 2023 1:47:07 PM part2.FileEncryptor main
INFO: Process complete, output saved at ciphertext.enc
gayleanto@MacBook-Pro-10 part2 % hexdump -b ciphertext.enc > dump2.txt
```

By comparing both dumps (dump1 and dump2) we can see that the ciphertext is different each time:

```
gayleanto@MacBook-Pro-10 part2 % diff dump1.txt dump2.txt
1,3c1,3
< 0000000 371 114 361 061 226 175 051 150 257 224 234 152 022 024 317 041
< 0000010 257 263 151 325 344 111 301 350 104 216 237 222 065 074 171 151
< 0000020 114 312 166 210 070 363 150 307 103 022 314 076 123 377 206 165
---
> 0000000 275 347 131 326 230 025 331 224 011 233 262 264 364 251 242 346
> 0000010 206 233 224 234 224 332 027 305 175 055 211 301 333 155 127 074
> 0000020 234 165 320 134 043 345 353 153 366 326 071 245 121 245 053 361
```

Part 3 : **Generating a Secret Key from a Password**

Part 3 is an implementation of the FileEncryptor, which accepts arguments from the command line to either encrypt or decrypt a file given a specified password. If the user wants to encrypt a file, the code will generate a salt and initialisation vector randomly, use PBKDF2WithHmacSHA256 to generate a secret key based on the password provided by the user, and then encrypt the file with AES/CBC/PKCS5PADDING. The salt, keys and initialisation vector are also saved to corresponding files for later use.

to decrypt a file, you need to provide the password. The program will then retrieve the salt and initialisation vector from existing files, generate the key using this password and the existing salt using PBKDF2WithHmacSHA256, and then use AES/CBC/PKCS5PADDING to decrypt the input file. It allows a user to generate a robust encryption key from a memorable password, instead of having to remember an encryption

key themselves. The method of creating the encryption key is also designed to be resistant to excessively weak or predictable passwords, via the addition of random salt and the use of the PBKDF2WithHmacSHA256 key function.

Encryption and **Decryption** steps:
First, we again compile the program:

```
gayleanto@MacBook-Pro-10 part3 % javac FileEncryptor.java
```

For Encryption, we only need to pass in 4 arguments (like part 2) but instead of requiring a base64 key, the user can select their choice of password:

```
gayleanto@MacBook-Pro-10 part3 % java FileEncryptor.java enc mrpeanutbutter plaintext.txt cipher
text.enc
Generated Salt: m38XFEF0HnqFPctyl2yJvQ==
Generated Key: o9AwgkbjUAwK0cC8ORpNeA==
Generated IV: cTSEyib2hkqV0CtvRn2BeA==
Aug 17, 2023 5:34:03 PM part3.FileEncryptor main
INFO: Process complete, output saved at ciphertext.enc
```

For decryption, just pass dec as the mode, the password used to encrypt, the ciphertext (output file) and the plaintext (input file):

```
gayleanto@MacBook-Pro-10 part3 % java FileEncryptor.java dec mrpeanutbutter ciphertext.enc plaintext.txt
Recovered Key: o9AwgkbjUAwK0cC8ORpNeA==
Aug 17, 2023 5:41:16 PM part3.FileEncryptor main
INFO: Process complete, output saved at plaintext.txt
gayleanto@MacBook-Pro-10 part3 % cat plaintext.txt
to be encrypted and decrypted :)
```

If we were to provide the wrong password, the plaintext would appear to be encrypted:

```
gayleanto@MacBook-Pro-10 part3 % java FileEncryptor.java dec passwordmaybe ciphertext.enc plaintext.txt
Recovered Key: H4nSfX6+qCMmiImvJCajoA==
Aug 17, 2023 5:43:09 PM part3.FileEncryptor main
INFO: Process complete, output saved at plaintext.txt
gayleanto@MacBook-Pro-10 part3 % cat plaintext.txt
Ev◆◆◆2◆◆K◆◆◆6`◆◆O{T`◆◆◆◆n◆V%
```

We can see that the ‘recovered key’ is not the same key produced during encryption.

Part 4: Designing for changes in recommended key length and algorithms

The part of the program is designed to encrypt and decrypt files using different algorithms (AES and Blowfish) and key lengths that change over time. The code also handles metadata associated with the file encryption and decryption processes.

Similarly to the previous part, If “enc” is the first argument, a file is encrypted using the specified algorithm and key length, where the encryption key is generated from a given password. If “dec” is inputted, the program reads

metadata from the encrypted file and utilises this to decrypt the file using the originally used key length and algorithm. Lastly, if “info” is passed, the program only prints metadata embedded associated with the file (cipher file).

The code is designed to securely encrypt and decrypt files with a user specified password.The "random" object is an instance of SecureRandom, which provides a strong random number generator. It is used to generate IVs and salt. This ensures unpredictable randomness, adding an extra layer of security to the encryption. PBKDF2 is used along with HMAC-SHA256 for the key generation from the user password. This algorithm implements key stretching to slow down brute-force attacks. One of PBKDF2's input parameters is salt. The salt in this code is randomly generated for each encryption operation making it practically impossible to use precomputed tables like rainbow tables for fast lookups of hash values.

Encryption:
The enc function begins by initialising a Cipher object. This is where built-in cryptographic functions are used to perform encryption. The initialisation vector iv and a 16 byte salt used for generating the encryption key is randomly generated. The Password-Based Key Derivation Function 2 (PBKDF2) with HmacSHA256 is employed to generate the encryption key from the password and salt input. The file is then read, encrypted, and finally written to the output file. The algorithm's name, IV, and salt are also written to the beginning of the file for decryption purposes. Metadata including key length and algorithm used are added to the file.
Encryption using AES:

```
gayleanto@MacBook-Pro-10 part4 % java FileEncryptor.java enc AES 128 mypassword plaintext.txt cipherte
xt.enc
Secret key is RKbAlk/XU0CuQ2PaxFAebQ==
Aug 18, 2023 10:57:36 PM part4.FileEncryptor enc
INFO: Encryption finished, saved at ciphertext.enc
Aug 18, 2023 10:57:36 PM part4.FileEncryptor main
INFO: Process complete
```

Encrypting using Blowfish:

```
gayleanto@MacBook-Pro-10 part4 % java FileEncryptor.java enc Blowfish 128 mypassword plaintext.txt cip
hertext.enc
Secret key is xU4mIU4xZpktfl+qaT9HSg==
Aug 18, 2023 10:58:42 PM part4.FileEncryptor enc
INFO: Encryption finished, saved at ciphertext.enc
Aug 18, 2023 10:58:42 PM part4.FileEncryptor main
INFO: Process complete
```

Decryption:
The dec function commences by extracting the embedded metadata from the input file, such as the algorithm name, IV, and salt. Using this metadata, it reconstructs the encryption key to decrypt the file, and writes out the decrypted data to the output file. The printInfo function displays the algorithm and key length used for encrypting a specific encrypted file by parsing the contents of the file..

```
gayleanto@MacBook-Pro-10 part4 % java FileEncryptor.java dec mypassword ciphertext.enc plaintext.txt
Aug 18, 2023 10:57:59 PM part4.FileEncryptor dec
INFO: Decryption complete, open plaintext.txt
Aug 18, 2023 10:57:59 PM part4.FileEncryptor main
INFO: Process complete
```