CS6324 Information Security Project -1

ENCRYPTED FILE SYSTEM

Thirunavukkarasu, Vignesh
VXT200003

Table of Contents

Design and approach:

The goal of the project is to encrypt the file and perform read, write, and cut operations efficiently.

Efficiency in measured terms of number of read / write files and performance in terms of encrypt and decrypt functions. At the same time, the security should be intact for the files, so that any tampering of the files is noticed.

Security:

For file encryption – AES-256 mode is used, where blocks of 16 bytes are encrypted with incremental IV for every Block. Incremental counter ensures that even if same 16 bytes of data are encrypted repeatedly, the output will be different making it difficult for the attacker to figure out and change.

Also, as part of data integrity HMAC signing is done for every file (including the metadata file). Any change in the content will be validated with the HMAC variation.

Metadata File Design:

Design Explanation:

Metadata Design:

Metadata file is the 0th file of the folder (file that's created by the EFS Program). It holds the critical details for encryption / decryption, file length, password, and salt.

As per my design each line holds an information encoded using Base64 encoding.

Line 1	file_length		
Line 2	user_name		
Line 3	hash(SALT || Password) (0, 15)	Padding (16, 127)	
Line 4	SALT (0-15)	IV (16,31)	Padding (32, 255)
Null Padding			
Last Line	HMAC (0, 31)		

User Authentication:

User password is hashed and stored in metadata file. 16-byte salt is created and concatenated with the original password. The concatenated bytes are then hashed with SHA384 byte algorithm. This ensures that even if the password is same for two users, the hash will be different (as randomization characters are appended).

When the user provides the password, the password will be concatenated with the SALT bytes from the metadata file, and SHA384 algorithm is performed to get the hash and then compared with the existing hash. IF matches then user is allowed to perform operation, else password exception is thrown.

Encryption Design:

For file encryption – AES-256 mode (**AES/ECB/NoPadding**) is used, where blocks of 16 bytes are encrypted with incremental IV for every Block. Incremental counter ensures that even if same 16 bytes of data are encrypted repeatedly, the output will be different making it difficult for the attacker to decrypt.

Length Hiding:

We calculate an integer variable called score, which is sum of all lower case alpha numeric char values in the file name. This value is then divided by the file name length, which is the score. This score is then multiplied with the actual file_length, and a Base64 encoded length is stored in the metadata file. This ensures that the file length becomes different and makes it confusing for the attacker.

Message Authentication:

Authentication is used to ensure the messages are intact, and if any changes made to a single bit is identified. For authentication, I've used HMAC algorithm, which is based on double hashing (nested hash) technique which makes it difficult. I calculate the HMAC of each file (encrypted data) up to 992 bytes and pass it to the HMAC calculation function (refer to pseudo code section). This HMAC is then stored in the last 32 bytes of a file (from 992-1024).

For checking integrity, I again calculate the HMAC of the first 992 bytes and compare it with the last 32 bytes. If it's different, then the file has been modified.

Efficiency:

Read:

For a read operation, the entire file / files will be read, but only the necessary 16-byte blocks will be decrypted. Ex: Consider a read operation – starting position = 100 and read length = 900.

For my design 1st and 2nd file will be read (first file: from byte 100-991 and second file: 0-7 bytes). In any case, the 2 files must be read completely. But for decryption of the contents only blocks from 6-62 in first file and for second file only block 1 will be decrypted.

First file: 100 / 16 = 6.25, floor (6.25) = 6

Second file: 8 / 16 = 0.5, ceil (0.5) = 1

This allows a maximum of 30 extra bytes (starting 15 and ending 15) to be decrypted unnecessarily, which cannot be avoided as for the AES encryption minimum 16 bytes are needed.

Write:

For writing operation, we internally perform read to find which blocks to be encrypted in a file.

As per my design, only the first and the last file will be read, as the operation is a overwrite the rest of the in-between files will not be read. This is done to identify the suffix and prefix contents. And the newly added text will overwrite the in-between files. So, given to update n files, only 2 reads and n write operations are performed.

Consider a scenario:

Existing file length = 12000 (13 files). To write: starting position = 100 and contents length = 10000, which will span to 12 files (as each file can contain only 992 bytes of data).

Now as per my approach only file number 1 and 11 will be read to append the suffix and prefix respectively. The rest of the files will be overwritten anyway by the new content. So, no need to read them unnecessarily. Just encryption and writing will be fine.

Best storage efficiency:

Once change is that instead of storing all the HMAC bytes individually, we can read all files and get one HMAC whereby only 32 bytes will be stored for n files instead of n*32 bytes of additional storage.

Best Speed efficiency:

My design already has the best speed efficiency, one improvement is that for cut and write operation I read the entire file and decrypt it fully. But instead, I can modify the code to read the entire encrypted contents and only decrypt the necessary 16-byte block. So only 1 block will be decrypted, there by speed becomes the best.

Functionality Implementation:

Important Helper functions (custom added in EFS.java):

calculateHMAC():

Calculates the HMAC signature of the given bytes (ideally 992 bytes of file message) with the given key bytes.

Pseudo code:

- If key size > 64 bytes, HASH it, else Zero Pad it to 64 bytes.
- Generate ipad-key by XOR of key and ipad byte (0x36) repeated till key's length.
- Generate opad-key by XOR of key and opad byte (ox5c) repeated till key's length.
- Concatenate ipadKey and message to form a single byte array 1.
- Hash 1 - Perform hash_256 algorithm on the concatenated byte array 1.
- Concatenate opadKey and Hash 1 to form single byte array 2.
- Hash 2 – Perform hash_256 algorithm on the concatenated byte array 2.

- Hash 2 is the HMAC of the message.
- Return Hash 2.

GenerateMetadataFile():

Function will generate the metadata file data.

Pseudo code:

- Line 1: Initiate the Length = 0 (as initial length of the file: 0).
- Generate random 16-byte salt, 16-byte IV using the secureRandomGenerator() function provided in the Utility class.
- Line 2: username provided.
- Line 3: Concatenate the SALT and password (SALT || Password) and generate the Hash of this with hash_SHA384() Utility function.
- Line 4: Append the SALT and IV bytes generated initially.
- Line 5: Calculate the HMAC of all the lines joined with "\n" as the line separator.
- Encode each line to Base64 String and join with "\n".
- Null pad the string to 1024 bytes length, with "\0" and save the final data to 0^{th} file (metadata file).

UpdateFileLength():

Function to update the file length in metadata file.

Pseudo Code:

- Read metadata file, split with \n and save it in String array.
- 0^{th} position holds the length of the file.
- Update the 0^{th} position to the Base64 encoded of the new value provided.
- Remove the last line (which originally holds null padding \0).
- Append all the elements of the array with \n to form a String.
- Call nullpadding function with length 992 bytes, to null pad the string up to 992 bytes
- Call calculateHMAC() function to get the HMAC of the updated String (992 bytes).
- Append the HMAC with the string to form the final string of 1024 bytes.
- Save the final string back to metadata file (0^{th} file).

verifyPassword():

Pseudo Code:

- Read Salt bytes (line 4) bytes [0, 15], Hash (SALT || password) (line 3) bytes [0,31] from metadata file.
- Generate hash of incoming password hash 2 = (SALT || incoming_password).
- Compare hash 1 and hash 2, if hash 1 == hash 2, then valid password (byte comparison of the array). Else, invalid password and throw Exception.

IncrementIV():

Pseudo Code:

- Increment the LSB bit of the IV.
- If max byte value is reached, then the next LSB bit is incremented.
- Repeat till MSB

BlockEncrypt():

Encrypt the given byte array message with Incremental IV using **AES/ECB/NoPadding** Algorithm, in blocks of 16 bytes.

Pseudo Code:

- Initialize an empty byte array called outBytes.
- Pad the given byte array to the round of greatest multiple of the block size provided. (Ex: For block size = 16, and given message byte length = 44, it will be padded till length becomes 48. As the nearest multiple of 16 greater than 44 is 48).
- Loop through the padded byte array.
- For each iteration, split it into 16 bytes of length.
- Encrypt the 16 bytes with the IV and append it to the outBytes.
- Increment the IV by 1 bit using incrementIV() function.
- Repeat till all the message bytes are read.
- Return the final encrypted message.

BlockDecrypt():

Decrypt the given byte array message with Incremental IV using **AES/ECB/NoPadding** Algorithm, in blocks of 16 bytes.

Pseudo Code:

- Initialize an empty byte array called outBytes.
- Loop through the message byte array.
- For each iteration, split it into 16 bytes of length.
- Decrypt the 16 bytes with the IV and append it to the outBytes.
- Increment the IV by 1 bit using incrementIV() function.
- Repeat till all the message bytes are read.
- Trim the outBytes to the message size provided.
- Return the final encrypted message.

GetIV():

Read metadata file and split the bytes to get the IV bytes.

Pseudo Code:

- Use read_from_file() function to get the contents of metadata file.
- Split the file with “\n” new line separator to get the contents of the file in String[] format.
- Get array element 3, and split bytes from [16, 31], which is the IV bytes originally stored.

NullPad():

To make a given byte array to a given length.

Pseudo Code:

- Append “\0” to the string.
- Check if the length has exceeded the given integer length.
- If not, then keep appending “\0” till it reaches.

- Else return the string.

Core functionalities

Create:

Creates a file with the given file name, sets the username, Hash (SALT || Password), length = 0 (as it just creates the file, and no contents are added to it yet). It also initializes 16 byte IV for AES encryption (random bytes). All contents are encoded to base64 string (ease of readability) and null padded to 992 bytes. The last 32 bytes are for HMAC of the entire contents.

Pseudo Code:

- Check if the given file exists and whether the file is a directory or not.
- If it already exists, throw Exception.
- Else, create a new folder with the given file name, and a file 0 under the folder.
- Now call the generateMetadata() function with file name, username, and password. (Refer to the pseudo code for the generate metadata function above).
- Save the result of the generate metadata function to the 0^{th} file.

Length:

Returns the length of the file by reading the metadata file (0^{th} file).

Pseudo Code:

- Confirm the password supplied if it doesn't match throw Password Exception.
- Else, read the metadata file and get the first line of the file (which will have the length).
- Return the calculated length.

Read:

Read the given file from starting position to length bytes.

Pseudo Code:

- Validate the password by calling validatePassword() function.
- Get the length of the file by calling the getFileLength() function and store it in file_length.
- If starting_position + len > file_length, throw Overflow Exception. Else proceed ahead.
- Calculate ending position = starting_position + len -1; (1 is subtracted as array position starts from 0).
- Calculate startFileBlock = starting_position / 992 bytes, and endFileBlock = ending_position / 992 bytes. (Each file has at max 992 bytes of data; rest 32 bytes is used for HMAC).
- Initialize a variable IV to the IV value in metadata file (using getIV()) method.
- Increment IV by startFileBlock * 62 positions, as each file will have 992 bytes of data = 62 blocks of 16 bytes each.
- Initialize an empty string variable called encString.
- Run a loop from startFileBlock + 1 till endFileBlock + 1, and for each iteration copy the contents of the file by using read_from_file().

- Append the string contents of the file to encString.
- Calculate the starting and ending block position with respect to the encString.
- Initialize a variable startAESBlockReference = floor(starting_position) / 16.
- Initialize a variable endAESBlockReference = floor(starting_position) / 16.
- Get the substring of the encString with start and end block reference and store it in encString.
- Calculate the starting and ending position within the start and end block.
- Initialize a variable returnString, decrypt the encString and store it in returnString.
- Return the substring of encString with starting and ending position within the start and end blocks respectively.

Write:

Write the given byte contents from starting_position. The write is overwritten, which means if starting position = 10 and length of new contents = 20, then we start from 10th position and overwrite all contents till 29th position, remaining contents remain the same. So, the length of the file will be affected only if the new contents updated is greater than (starting_position + len_to_write).

- Validate the password by calling validatePassword() function.
- Get the length of the file by calling the getFileLength() function and store it in file_length.
- If starting_position + len > file_length, throw Overflow Exception. Else proceed ahead.
- Calculate ending position = starting_position + len -1; (1 is subtracted as array position starts from 0).
- Declare the variables prefixStartingPosition, prefixLength, suffixStartingPosition, suffixLength.
    - prefixStartingPosition = startingFileBlock * 992
    - prefixLength = Math.min(992, file_length - startFileBlock * 992)
    - suffixStartPosition = endFileBlock * 992
    - suffixLength = Math.min(992, file_length - endFileBlock * 992)
- Declare 2-byte arrays suffixBlockContents, prefixBlockContents to store the prefix and suffix contents.
    - Use read function to extract the prefix block conetnts with prefixStartingPosition and prefixLength.
    - Similarly use read function to extract the suffix block conetnts with suffix StartingPosition and suffixLength.
- Convert the prefixBlockContents and suffixBlockContents into prefixString and suffixString respectively.
- If the new contents to write exceeds the length of the file (w.r.t starting position), then suffixString = “” (empty string), else substring to get the suffix contents
- Declare a string variable finalString and concatenate prefixString, contents to write, suffixString.
- Null pad the final string to next greatest multiple of 992 bytes.
- Declare an int variable final_length = length of the final string.
- Initialize a variable IV to IV bytes by calling getIV() function.
- If final_length < file_length, then final_length = file_length.
- Run an iteration starting from startFileBlock +1, till endFileBlock + 1.
    - For each run, split the finalString into 992 bytes.

    - Declare a variable enc of type byte[] and initialize it to the output of blockEncrypt function with input parameters as 992-byte message and IV.
    - Calculate the HMAC of the enc message with IV.
    - Append enc message and HMAC of the file and save it, by calling save_to_file() function.
- If the finalLength > file_length, then update the length in metadata file by calling updateFileLength() function.

Cut:

Given a length L, Cut the file leaving the remaining length to L.

Pseudo Code:

- Validate the password provided.
- If valid password, read length, and IV (bytes).
- Initialize a variable file_length and get the length of the file in it (length() function).
- If file_length equals provided len, then don't perform any operation, else proceed.
- Initialize an int variable end_block = (len)/992.
- Initialize a variable IV to the IV value in metadata file (using getIV()) method.
- Read the end_block contents and store it in a variable msg of type byte[].
- Decrypt the message msg with the IV and store it in dec variable.
- Initialize a string variable str to store the split the dec variable to the appropriate length and append it with null padding for 992 bytes.
- Initialize a variable enc of byte byte[] and store the encrypted data of str, encrypt with IV in incremental mode.
- Calculate the HMAC of the enc[] with IV and store it in hmac variable of type byte[].
- Create a signedEncBytes variable of type byte[] and store the concatenated enc and hmac bytes.
- Call the save_to_file() function and pass the signedEncBytes to it to store it in the file.
- Remove the remaining files if any left that exceeds the length.
- Invoke the updateFileLength() function to update the length in metadata file.

Check_integrity:

Once confirming the user password, read every file from [0-992) bytes and calculate the HMAC of the data (with key = IV found in the metadata file). Verify the HMAC with the last 32 bytes of data of the file.

Pseudo code:

- Validate the password provided.
- If valid password, read length, and IV (bytes).
- Initialize a length variable and get the length of the file in it (length() function).
- With the length, calculate the total available data files and store it in totalFiles variable.
- Initiate a status variable to 0.
- Loop from 0 to total available files.
- For each file, read the file and extract the first 992 bytes [0,991] as data bytes and the last 32 bytes [992,1023], which is the HMAC for the file.

- Pass the 992 data bytes with IV to calculate the HMAC of the file, call the calculateHMAC() function.
- Compare the returned result of the function with the existing HMAC for the file.
- If not same increment the status variable.
- Finally compare if the status variable is 0 or not. If 0, return true (i.e., all files are intact). Else return false. One or many files could be corrupted.

Design Variations

1. Suppose that the only write operation that could occur is to append at the end of the file. How would you change your design to achieve the best efficiency (storage and speed) without affecting security?

   As per my design, there will be no change in the design (can be altered with the starting position to file length).

   If it's only appended at the end of the file, then the last file will be read and updated. If the existing contents are just till the end of the file (i.e., no padding is done). Then new contents will be added to the next subsequent file.

   So, there will be at max one file being updated along with HMAC. And all remaining will be write operations.

2. Suppose that we are concerned only with adversaries that steal the disks. That is, the adversary can read only one version of the same file. How would you change your design to achieve the best efficiency?

   There will be no change in the design. Since the encryption mode is CTR, each block will have unique IV. Even if the text is repeated 16 bytes of content, each encrypted data will be different. The advisory cannot find any patterns and decrypt it.

   The one concern will be storing of the IV in metadata file. I would rather have it stored under root permission and each user when using the application can switch to the necessary privileges which will make it more difficult for the advisory.

3. Can you use the CBC mode? If yes, how would your design change, and analyze the efficiency of the resulting design. If no, why?

   Yes, we can use it.

For any read operation, we start from the first file first block, decrypt it and then use the cypher text of the first block as the IV of the second block, and so on. This operation must be performed to any file length, even if we need to read the last file contents. So, it's exponentially worse.

For write operation, we first need to read prefix and suffix contents. We invoke the read function to get the prefix and suffix contents. Now, we then append the new contents and for encryption, we need to start encrypting from the start of the update block, till the last contents, as even after the suffix is appended, the IV gets changed so will need to re-write entirely from the start block.

In CBC each 16-byte block is encrypted and passed as IV for the next block. So, we cannot perform any parallel operations on READ / WRITE, which leads to the fact that for any read / write operation we need to perform the operation from the beginning of the file.

Also, if there are any Error in the byte in any of the blocks, it will keep propagating to all the remaining blocks, which could cause incorrect output.

4. Can you use the ECB mode? If yes, how would your design change, and analyze the efficiency of the resulting design. If no, why?

Design updates: If we remove the IV increment operation, we can use ECB mode of encryption and decryption.

But it would affect the security of the application. Because ECB uses a single key and splits the message into blocks and uses the same key to encrypt the blocks.

If a part of the text is repeated, it will lead to repetition of the cipher text as well, and the attacker could use this find the IV details.

Design discussion credits:

I discussed some of the concepts with other students.

Giftson: Discussed HMAC logic for storing in each file itself, in last 32 bytes and design variations.

Nikhilesh and Ramnath: Discussed read optimization for speed, and design variations.