Information Security Project 1

Encrypted File System

Name: Ashwin Sai Chandrashekhar

3.1 Design explanation

Meta-data Design:
The meta-data structure is stored in a separate file associated with each encrypted file. For example, the meta-data for a file named "abc.txt" would always be stored in a file named "0" within the directory containing "abc.txt". The precise structure of the meta-data is as follows (1024 Bytes):

Metadata structure:
Header:
Bytes 0 to 127 - user name.
Bytes 128 to 159 - password hash.
Bytes 160 to 175 - salt.

Secret Data:
Bytes 176 to 207 – hashed password
Bytes 208 to 560 - secret data (incl. file length)

HMAC is appended at the end of the Meta data content in the Metadata file.

User Authentication:
Password-related information is stored in the meta-data, including the hashed password (obtained using hash_256 algorithm in the Utility file) and the salt used for hashing. During user authentication, the stored password hash is compared with the hash_256 of the provided password after applying the same salt and hashing algorithm. If the hashes match, the user is authenticated.

Encryption Design:
Files are encrypted using AES encryption provided in the Utility file. Each file is divided into fixed-size blocks, and each block is encrypted using CTR (using salt as IV, incrementing salt by 1 for each subsequent block) individually. The encrypted block is appended with a HMAC for message authentication. This design ensures security because even if an adversary can read each stored version of the file on disk, they would need the encryption key and HMAC key to decrypt and verify the integrity of the file, which are securely stored separately. The encryption key for AES is computed using hash_512 algorithm using salted password as message.

key for encryption (AES) – hash_SHA512
key for salted hashed password - hash_SHA256
key for HMAC – hash_SHA256

Length Hiding:
The file length is hidden within the meta-data. However, it is worth noting that due to the fixed block size used for encryption, the actual file length may not be precisely hidden, but rather rounded up to the nearest block size. This design choice minimizes the overhead of hiding file length without significantly increasing the number of physical files needed.

Message Authentication:
Message authentication is achieved using HMAC. Each encrypted block is appended with a HMAC calculated using a secret HMAC key. During file read operations, the HMAC is verified to ensure the integrity of the decrypted block. Meanwhile HMAC is computed using the below formula from slides.

HMACK[M] = hash_256[(K+ XOR opad) || Hash[(K+ XOR ipad)||M)]]

where, ipad and opad are 0x36 and 0x5c, K is the key and M is the data. The key for HMAC is generated using hash_256 algorithm. Extracting the data from the positions of the salt in metadata, we use that as a key for generating HMAC for Metadata block. For others we use, the same data from the positions as of salt but from the encrypted data block for which the HMAC has to be generated and stored.

Efficiency:
The design offers a balance between storage and speed efficiency. To maximize storage efficiency, a fixed block size is used for encryption, reducing the need for additional padding. To maximize speed efficiency, AES encryption is used, which is a fast and efficient encryption algorithm. The design also minimizes redundant data storage by storing only essential meta-data information.

To achieve maximum storage efficiency, we could increase the block size, and this would reduce the number of physical files present in the disk. But it leads to risk in security as the block would contain a lot of data due to its large size.

For speed efficiency, we could encrypt the whole file with a single key, instead of generating key for each block. But again, this would lead to security issues if the attacker could get access to one block.

The design was chosen as it would compromise both storage and speed in a balanced way. Since we compute key and HMAC for every block, we compromise both storage and time but provide a comparatively secure file system.

3.2 Pseudo-code

```
function Check_Password(metadata, password):
    if not validatePassword(metadata, password):
        throw PasswordIncorrectException
```

To validate a password:

1. Extract the salt value from the metadata.
2. Retrieve the encrypted secret data from the metadata.
3. Generate the expected hash value using the provided password and the extracted salt.
4. Generate the encryption key using the password and salt.
5. Decrypt the encrypted secret data using the generated key.
6. Extract the actual hash value from the decrypted secret data.
7. Compare the actual hash value with the expected hash value.
8. Return true if they match; otherwise, return false.

read:

To read a file from a specific starting position up to a certain length:

1. Obtain the file handle and metadata associated with the file.
2. Check the validity of the provided password using the metadata.
3. Determine the total length of the file and verify that the requested start position and length are within bounds.
4. Calculate the initial and final blocks of data based on the start position and length.
5. Adjust the salt value in the metadata to reflect the initial block.
6. Extract the requested portion of the file data using the adjusted salt and other parameters.
7. Return the extracted file content.

Length:

To determine the length of a file:

1. Obtain the file directory and metadata file associated with the specified file name.
2. Fetch the metadata related to the file.
3. Validate the provided password by checking it against the metadata.
4. Ensure the integrity of the metadata file by validating its HMAC.
5. Extract the salt value from the metadata.
6. Generate a cryptographic key using the password and salt.
7. Retrieve the encrypted secret data from the metadata.
8. Decrypt the encrypted secret data using the generated key.
9. Extract the portion of decrypted data containing the file length.
10. Convert the extracted byte array representing the file length to a long value.
11. Return the file length as an integer.

Create:

To create an encrypted file:

1. Check if the provided username and password lengths are within acceptable limits.
2. Create a directory with the specified file name if it does not already exist.
3. Create a metadata file within the directory.
4. Generate a random salt value.
5. Pad the username (128) to ensure it fits a predetermined length.
6. Generate a hash of the password using salt.
7. Create a header by concatenating the padded username, password salt.
8. Initialize the file length to zero and convert it to a byte array.
9. Combine the hashed password and file length byte arrays into secret data.
10. Write the secret data and header into metadata.
11. Calculate the HMAC for the metadata.
12. Append both the metadata and HMAC into a single byte array.
13. Pad the byte array for 1024-byte size.
14. Write the padded array into file and create it.

Write:

To write and save a file:

1. Fetch the metadata of the specified file.
2. Verify the provided password against the metadata.
3. Convert the byte array content into a string.
4. Determine the length of the content.
5. Retrieve the file handle based on the file name.
6. Obtain the current length of the file.
7. Check if the starting position is valid.
8. Calculate the initial and final blocks based on the starting position and content length.
9. Extract the salt from the metadata and adjust it according to the initial block.
10. Iterate over each block from the initial block to the final block.
    a. Write the blocks of data to the file using the specified parameters.
11. Update the metadata if the new content extends the length of the file.

Check integrity:

To check the integrity of a file:

1. Retrieve the file directory based on the file name.
2. Set the initial block to 1.
3. Fetch the metadata of the specified file.
4. Obtain the length of the file using the provided password.
5. Calculate the final block by dividing the file length by the secret block size.
6. Evaluate the integrity of each block within the file directory by comparing the actual and calculated HMAC.
7. Return true if the file has not been modified, otherwise return false.

Cut:

To cut a file:

1. Retrieve the metadata of the specified file.
2. Obtain the file handle based on the file name.
3. Determine the current length of the file using the provided password.
4. Check the password for authentication.
5. If the specified length is greater than the current file length, throw an error.
6. Extract the salt value from the metadata.
7. Calculate the final block based on the specified length and the secret block size.
8. Update the salt to point to the final block.
9. Read the encrypted data from the file handle for the final block.
10. Generate the encryption key using the password and salt.
11. Decrypt the data using the generated key.
12. Convert the decrypted data to text format.
13. Truncate the text data to match the specified length.
14. Convert the truncated text data back to bytes.
15. If the length of the data is less than the secret block size, pad the data.
16. Write the cut data to the file, update the metadata, and save the changes.

3.3 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?

Solution:

We can use a block cipher mode like AES in CTR (Counter Mode) to allow independent encryption and decryption of each chunk. Additionally, we can integrate metadata directly into each chunk, eliminating the need for a separate metadata file. This approach reduces the computational overhead and storage requirements, enhancing both speed and efficiency without compromising security.

But for best security practices, the current design can be left unchanged.

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?

Solution:

To achieve the best efficiency, we could use ECB method, as we use unique key for each file we encrypt. This way we can reduce the time complexity of encryption and dependency between files. Also, the same key will be used inside the file for all the blocks. The time for generating a new key for each block in a file is also reduced. It is faster this way.

However, it is not suggested because if one block is compromised, all other blocks will also be compromised.

3. Can you use the CBC mode? If yes, how would your design change, and analyze the efficiency of the resulting design. If not, why?

Solution:

Yes, you can use it. But the drawbacks for using CBC mode is that encryption is dependent on the previous block. So, overhead of maintaining the previous block result increases. Also, checking integrity would fail for all blocks if anyone block is modified by the attacker. As for Time efficiency would almost be the same.

If not, we can proceed with the current design using CTR as it will be more secure since each file is encrypted separately and any change in the file would not affect the other files. There would be a decrease in storage efficiency as we would have to store the key for each block.

4. Can you use the ECB mode? If yes, how would your design change, and analyze the efficiency of the resulting design. If not, why?

Solution:

Yes, we can use ECB. We will be using the same key for all the blocks. This would increase in terms of time efficiency and storage since we need not store multiple keys and encryption is not dependent on previous blocks. But, if the key is out (known publicly), the attacker can easily decrypt all the files using single same key.

Generally, it is not preferred to use it because of the drawbacks and security. Security will be compromised compared to the current CTR mode.