− How mutual authentication is achieved in the current implementation of 3S

In this project, the code achieves the mutual authentication by using set of cryptographic signatures that makes the use of certificates and keys. There are 3 users created and each of them have their own public-private key pairs which are further used for authentication purposes. The program requires the user to first put the username and his/her private key, that needs to be issued by the CA and also it has their own (CA) public-private key pair stored in the server. Now, a statement is being made that links the userid with the client being used as client id. Further, the statement is hashed using SHA256. Next, the hash is signed using the user's private key and shared to the server with the userid and statement. As we have already placed the user public key in the server under server > application > userpublickeys directory, hence the server uses the public key of the user to further verify the signature and validate the user's identity.

In this way, the Mutual authentication is thus accomplished when the CA and the signatures in this implementation, which are trusted mechanisms, are used by the client and the server to verify each other's identities. This creates a secure connection and guarantees that both parties are who they say they are.

- Details on the cryptographic libraries and functions used to handle secure file storage.

The libraries used in this implementation are:

1. cryptography.hazmat.primitives.padding.pkcs7

PKCS #7, or Cryptographic Message Syntax (CMS), is a standard for storing encrypted and signed data. PKCS #7 is used to generate and verify digital signatures and certificates. Digital signatures use strong cryptographic techniques like encryption and hashing to prove the authenticity of digital messages, files, and documents

2. cryptography.hazmat.primitives.ciphers.Cipher, algorithms, modes

Cipher: A class that combines an algorithm and a mode to create a cipher for encrypting or decrypting data.

algorithms: A module containing various encryption algorithms (like AES, DES, ChaCha20, etc.). We are using AES encryption

modes: A module containing modes of operation for block ciphers (like CBC, CTR, ECB, etc.).

3. cryptography.hazmat.primitives.asyymetric.rsa_padding: Basically, padding is a process used to prepare data for encryption or cryptographic transformation. I am using this lib to encrypt the file during checkin.

4. cryptography.hazmat.primitives.asyymetric.hashing: Hashing is a fundamental cryptographic operation that takes an input (or 'message') and returns a fixed-size string of bytes.

How the user information and metadata related to documents were stored.

The user’s public key is stored in the server that helps to verify the user identity. The files (file1, file2) provided by users were stored in the documents folder with an associated .json file which stores the metadata of the file that contains fields such as grant access, owner, doc_id, grant_expiry time, aes key and grant_users.

```
metadata = {

        'doc_id':””,

        'owner': “”,

        'security_flag': “”,

        'grant_users': “”,

        'grant_access': “”,

        'grant_exp': “”,

        'aes_key': “”

    }
```

A randomly generated AES key was used to encrypt confidentiality-based files. This key was then further encrypted using the server's public key and stored in the metadata file. Additionally, files with an integrity focus were stored and signed with the original file.

Implementation details:

Login

1. First, the filename containing the private key and the user ID are entered. The client node confirms the existence of the private key. After the private key is found, the SHA256 algorithm is used to hash a statement that links the user ID and the client node in use. Along with the statement and user ID, this hash is additionally signed at the client end using the user's private key before being sent to the server. The server confirms the user's identity using the plain statement and uses the user's public key to validate the signature. In case it works, a session token is sent by the server.

2. Check in

The user enters the document to be uploaded, selects the security flag, and sends the document and related session token to the server. The user's session token is first verified. The file is then moved to the check-in folder by the server after it has verified that it is in the check-out folder. Subsequently, the corresponding metadata file loads, aiding in verifying whether the user is the file's owner; if not, it verifies whether the user is included in the list of permitted users. If the user is included in the list of permitted users (grant_users), the server additionally checks for access and time limits. Lastly, the file is checked in based on the security flag. A randomly generated AES key is created and used to encrypt the content of files associated with confidentiality. The metadata file stores this key after it has been further encrypted with the server's private key. A signed copy of integrity-focused files is kept in storage with the original.

3. Checkout

First, using the filename entered by the user as a guide, it is verified that the file exists on the server. Next, in order to carry out a series of checks, the server loads the related metadata. If the user is not the file's owner, it then determines whether they are on the list of authorized users. Checks for access type and duration are also made in case the server is not the owner. And when all these conditions are met, then the user can successfully checkout.

4. Grant

The filename, grant access type (checkin, checkout, or both), grant_user (to indicate which user is permitted to carry out such intended actions), and the duration of the file to which grant should be assigned are required for this operation. The file owner is the only one who can validate this operation. In order to perform this operation, the grant attributes grant_users, grant_expiry_time, and grant access type must be updated in the metadata file.

5. Delete

Validating the session token and confirming that only the owner is authorized to carry out this action are part of this operation. Once the owner of the file is verified and he/she is performing this action then all the files related to fileX .txt such as fileX.txt, fileX.txt.json, and fileX.txt.Sgn are deleted permanently from the server.

6. Logout

This action involves revoking the user's session token from the session token dictionary. But before that the server verifies if there are any newly modified files in the checkout directory using hashing and a checkout dictionary. If present the server first checks in this file and terminates the session.

-List the assumptions made if any.

1. The 256-bit AES key, the hashing algorithm I used, and the session tokens are assumed to be ideally pseudo-random and challenging for a computationally limited entity to guess.
2. I am also assuming that the server can handle heavy data requests and allocate the memory appropriately for larger files as well.

- Results of the static code analysis and the tools used.

I have used pyflakes for code analysis. For client.py and server.py, the maximum number of warnings were resolved. For client.py all the warnings were resolved. For server.py, The issues were, I was using some variables that were not used in the code and these variables were meant for debugging purposes.

. The tool pylint was also utilized for code analysis that gave me a score of 7.5/10. The score 7.5 was because I have been using many print statements to debug the code during development. Also, there were some spaces due to which it showed a bit less score.

-Threat modelling

Assets:

The things which we need to protect from the attackers are the session key, encryption key (aes key), private keys of the users and the files (fileX.txt, fileX.txt.json and fileX.txt.sign) that are used during checkin and checkout process.

Threats:

- MITM: An attacker intercepts the communication between the user and the server to capture sensitive information such as session tokens.

- Exploitation of server side vulns: If there are existing vulnerabilities on the server side, they could be exploited by attackers to compromise the server.

- OS command injections :An attacker could potentially inject and execute malicious OS commands into the server through insecure input fields that do not sanitize user inputs..

Vulnerabilities:

- Dependency weaknesses:
  The code depends on external dependencies that might contain security weaknesses, impacting the security of your own code if these dependencies are exploited.

- Insecurely handling session tokens: The system's handling of session tokens might be insecure, allowing potential interception by an attacker during transmission.

- Lack of input validation: The system does not sanitize user inputs, which allows for OS command injection. This is a significant vulnerability as it permits execution of arbitrary commands on the server.

Mitigations:

- Using the latest and updated dependency can minimize the impact on the application.
- Implementing Virtual Private Networks (VPNs) or other secure tunneling protocols when sensitive data is transmitted over potentially insecure networks.
- Whitelisting of specific inputs or usage of domains so as to prevent server side vulnerabilities such as SSTI, SSRF and server misconfiguration etc.
- Properly sanitizing the user inputs so as to avoid various kind of Injection attacks.