dvenkatesh7

PROJECT 4: SECURE SHARED STORE

Architectural Design Details

1. *Mutual Authentication in the current implementation*
   mTLS is configured in nginx based on the configuration found in /etc/nginx/sites-enabled/server. The secure-shared-store.key and secure-shared-store.crt and CA details are provided in this file.

   The client constructs the body which contains the user_id, statement and the signed statement. The client signs the statement with user's private key to generate the signed statement. This is base64 encoded. This body is sent to the server.

   When server gets this request, it gets the corresponding user's public key and verifies if the signed_statement is same as the statement. If it's same, it is clear that the user is the one who he/she claims to be as the client used the user's private key to sign the statement.

   Once this check is done, a session token is generated and all the further operations and performed after verifying this token.

2. *Cryptographic libraries and functions*
   a. SHA256 is used to create hash of the statements and contents. (digest)
   b. RSA.importKey() is used to read the keys from the files where keys are stored.
   c. Crypto.Signature.PKCS1_v1_5 is used to create a verifier and verify the signed data with the actual data.
   d. Jwt.encode() is used to create the session key for a user using server's private key and RS256 algorithm.
   e. Crypto.Cipher.PKCS1_v1_5 is used to create a cipher of the server's private key.
   f. b64encode/b64decode is used to send contents to and from the server.
   g. AES is used to encrypt the strings, to do padding etc.

3. *Data Storage*
   a. Flat files are used as write ahead logs to maintain comma separated list of values. This is an append only write ahead log with the latest line (analogous to row in a RDBMS) holding the 'current' data. The advantage of this approach is simplicity of implementation, it's very fast to update data [O(1) - append]. The disadvantage is reading all the data during server startup can take longer if the number of lines in the logs increase [O(n) operation]. For the scale needed in this project, such a flat-file based database is sufficient.
   b. Files checked into the server are stored in the file system. If confidentiality is chosen, the file is encrypted, encoded and then stored in the file system. If integrity is chosen, the original file sent by the user (e.g. file1) and the file with the signed contents (file1_signed) are both store in the file system.

c. Db.log maintains the document metadata as comma separated values: did, owner, security_flag, key, is_document_deleted. The 'key' is empty for integrity option.
d. Grant.log maintains the grant data as comma separated values: did, target_uid, access_type, duration, grant_timestamp, is_grant_expired.
e. Sessions.log mainstains the session data as comma separated values: session_token, user_id, is_session_invalid.

Implementation Details

1. *Implementation of functionalities*
   We will build the grant_dict to store the grant details and doc_metadata to store the details of the documents and sessions to store session details. We read this from the respective files and store the values in our in-memory data structures.

   a. *Login function*:
   The client takes the user_id and private key file as input. The statement is generated, and this is signed using the private key of the user. This signed statement is then encoded, and all these constitute to the body of the request that client will send to the server. We are also maintaining the session details.

   Once the server gets this request, it extracts the user_id, statement, signed statement. To verify the validity of the statement, it first gets the user's public key and verifies the statement with the signed statement and if it is the same, it generates a session token for the user. To do this, we use the jwt module to encode the request data with the server's private key. This will ensure that the session token will be unique. If the session already exists for this user, it is invalidated and if not, this user id is added to the session state. If the login fails, response will be 700.

   b. *Checkin function*:
   The client specifies the document and the security flag. If the document already exists in the checkout folder, we will move it to the checkin folder. To prepare the request body, we read the contents of the file and encode it using b64encode.

   Once the server gets this request, it extracts the did, security_flag and the contents. Next from the session, we verify if the user has permission to update the document. If he doesn't have access, we return an error response. If the target UID is 0, next we check if the grant has expired or not. If both these conditions pass next, we check if the user has checkin access. I.e., the access type should be 1 or 3. If this check also passes, next we check for the security flag. If its 1, we generate a random key and encrypt it using server's public key and then we encrypt the contents using this newly generated key with AES. Then we update the doc_metadata with the new values. If it is 2, then we sign the document contents by using the secure-shared-store private key. We store the files in the 'document' folder of the server and also update our document metadata.

c. *Checkout function:*
The client specifies the action as checkout and sends a request to the server that includes the did. The server extracts the token from this to validate the session. If it is valid, the server next sees if the requested file is present. If it is present, we check If the user has permission to checkout. For this, we check if the target id is 0 or if the grant for that did is for this user requesting the checkout, and the grant timestamp expiration. If all these tests pass, we see if the user has access to checkout (options 2 or 3). Next, if this test passes, we see the security flag. If it is 1, we first decrypt the key using the server's private key and use this decrypted key to decrypt the contents and encode it. If it is 2, we check if the contents are same as the signed contents using the server public key and return the contents in the response.
The client saves the list of checked out files to be used later during logout and checking operations.

d. *Grant function*:
In the server, we first validate the session. We extract the user_id and did from the client's response. If document is not present in the metadata, we return with an error. Next, we return an error if the user_id is not the owner. If all these tests pass, we access the grant by updating our grant_dict. This is filled with the data that was passed by the client.

e. *Delete function*:
In the server, first the session is validated. Next, we check if the owner is sending the request and next if the file exists or not. If both tests pass, the file is removed from the path and the doc_metadata is updated.

f. *Logout function*:
We first check if any files were checked out from the server and were modified by verifying the md5 hashes of the file taken at the time of checkout and the current md5 hash. If yes, we checkin them and pass the security flag as 2 for integrity.

2. *Assumptions*
   a. The appropriate certificate files are present at the expected locations.
   b. Server does not crash after writing to the log and before updating the in-memory data structure.
   c. Certifi SSL error fixing code doesn't lead to an error.

dvenkatesh7

<u>Result of static code analysis</u>:
Pylint was used to perform the analysis.

Client.py

```
cs6238@CS6238:~/Desktop/Project4/client2$ pylint client.py
No config file found, using default configuration
************* Module client
C:102, 0: Trailing whitespace (trailing-whitespace)
C:261, 0: Trailing whitespace (trailing-whitespace)
C:382, 0: Trailing whitespace (trailing-whitespace)
C: 22, 0: Constant name "session_token" doesn't conform to UPPER_CASE naming style (invalid-
name)
C: 23, 0: Constant name "checked_out_files" doesn't conform to UPPER_CASE naming style
(invalid-name)
W: 91,43: Unused variable 'err' (unused-variable)
W:138, 8: No exception type(s) specified (bare-except)
R:113, 0: Either all return statements in a function should return an expression, or none of them
should. (inconsistent-return-statements)
C:375,31: Variable name "e" doesn't conform to snake_case naming style (invalid-name)
C:  3, 0: standard import "import hashlib" should be placed before "import certifi" (wrong-
import-order)
C:  4, 0: standard import "import json" should be placed before "import certifi" (wrong-import-
order)
C:  5, 0: standard import "import os" should be placed before "import certifi" (wrong-import-
order)
C:  6, 0: standard import "import shutil" should be placed before "import certifi" (wrong-
import-order)
C:  7, 0: standard import "from base64 import b64decode, b64encode" should be placed before
"import certifi" (wrong-import-order)

-----------------------------------
Your code has been rated at 9.23/10
```

Server.py

```
cs6238@CS6238:~/Desktop/Project4/client2$ pylint ../server/application/server.py
No config file found, using default configuration
************* Module server
C:331, 0: Line too long (115/100) (line-too-long)
C: 15, 0: Constant name "secure_shared_service" doesn't conform to UPPER_CASE naming style
(invalid-name)
C: 16, 0: Constant name "api" doesn't conform to UPPER_CASE naming style (invalid-name)
```

dvenkatesh7

```
C: 18, 0: Constant name "session" doesn't conform to UPPER_CASE naming style (invalid-name)
C: 19, 0: Constant name "grant_dict" doesn't conform to UPPER_CASE naming style (invalid-name)
C: 20, 0: Constant name "doc_metadata" doesn't conform to UPPER_CASE naming style (invalid-name)
C: 26, 0: Class name "welcome" doesn't conform to PascalCase naming style (invalid-name)
R: 29, 4: Method could be a function (no-self-use)
C: 35, 0: Class name "login" doesn't conform to PascalCase naming style (invalid-name)
R: 38, 4: Too many local variables (20/15) (too-many-locals)
E: 60,18: verifier.verify is not callable (not-callable)
R: 38, 4: Method could be a function (no-self-use)
C:103, 0: Class name "checkout" doesn't conform to PascalCase naming style (invalid-name)
R:125, 4: Too many local variables (23/15) (too-many-locals)
E:236,22: verifier.verify is not callable (not-callable)
R:125, 4: Method could be a function (no-self-use)
C:257, 0: Class name "checkin" doesn't conform to PascalCase naming style (invalid-name)
R:304, 4: Too many local variables (22/15) (too-many-locals)
R:304, 4: Too many statements (57/50) (too-many-statements)
R:304, 4: Method could be a function (no-self-use)
C:469, 0: Class name "grant" doesn't conform to PascalCase naming style (invalid-name)
R:472, 4: Method could be a function (no-self-use)
C:546, 0: Class name "delete" doesn't conform to PascalCase naming style (invalid-name)
R:549, 4: Method could be a function (no-self-use)
C:637, 0: Class name "logout" doesn't conform to PascalCase naming style (invalid-name)
R:640, 4: Method could be a function (no-self-use)

----------------------------------
Your code has been rated at 8.84/10
```

Threat Analysis

1. The Man in the Middle attack will not be effective because the communication between the client and server uses certificates, private and public keys. The data that is sent between the client and servers is encoded and encrypted with keys.
2. If the user's private key is leaked, the attacker might be able to use that to sign the statement in the client and send it to the server.
3. If the operating system has a vulnerability and it is exploited after it's loaded, it can leak the user's private key. It will hold the user's private key for a short time t sign the statement.
4. If the OS is compromised, then the attacker will have access to all data present in the system and can further leak the data.
5. The very basis of a secure communication in our implementation is the certificate. If a fake certificate is provided, man in the middle attack is possible as the communication will no longer be secure.