CS6238 Secure Computer Systems – Fall 2020

Project 4: Secure Shared Store (3S)

Deadline: 11/24/2020 (11:59pm)

Goals & Assumptions

This project is based on the topic of distributed systems security that is covered in Modules 11 and 12. The goal of the project is to gain hands-on experience in implementing secure distributed services. You will develop a simple *Secure Shared Store (*3S) service that allows for the storage and retrieval of documents created by multiple users who access the documents at their local machines. In the implementation, the system should consist of one or more 3S client nodes and a single server that stores the documents.

Users should be able to login to the 3S server through any client by providing their private key as discussed in Module 12. Session tokens would be generated upon successful authentication of the users. They can then check-in, checkout and delete documents as allowed by access control policies defined by the owner of the document.

To implement such a distributed system, we will need to make use of certificates to secure the communication between clients and the server, and to authenticate sources of requests. You will need to make use of a Certificate Authority (CA) that generates certificates for users, client nodes and the server. All nodes trust the CA.

Project Setup

We have provided a Virtual Machine for the project. **Links to download the image (.ova file) will be posted on piazza**.
The default account on the VM is **cs6238** and the password is **cs6238**. The root password is also **cs6238**. In an ideal setting, the 3S server and the client would be on separate nodes. For simplicity, we have set up only one VM. The server and client nodes are abstracted as separate folders within the VM. For example, the *server* folder represents the server and the *client1* folder represents the client node.

The desktop contains a Project4 folder which has the skeletal implementation of the 3S service. You will be required to complete the implementation to satisfy all the functionalities which will be detailed below. The Project4 folder contains:

1. **CA** - Represents the Certificate Authority and contains the CA certificates
2. **server** - Represents the server. It contains server certificates and the 3S application code. The 3S server is implemented using Python Flask and *server.py* contains the outline of the server code which is to be fully completed.
3. **client1** - Represents one of the client nodes. *client.py* has the skeletal implementation of the client. You will be required to generate client certificates and place them in the client1/certs folder.
4. **client2** - Represents another client node and the environment should be similar to client1.
5. **Testing Tools** - Contains a sample testing script to present an idea of how your client must accept inputs.

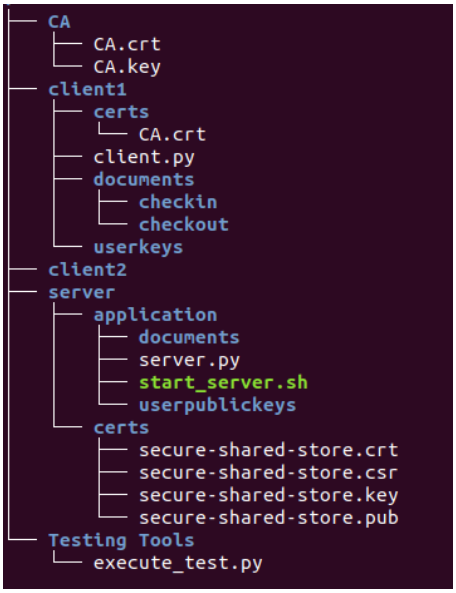

Fig: Folder structure of Project4

Certificates

As discussed above, we will need to make use of a Certificate Authority that is trusted by all nodes. This CA would be used to generate certificates for the users, client nodes and the server. One can make use of a library such as **OpenSSL** for setting up the CA and to generate certificates.

For this project, we have created a CA. This CA has been used to generate certificates for the server. You would be required to generate certificates for the client nodes and the users using this CA. The CA (certificate and key) was generated using the password **cs6238**.

Detailed instructions on generating certificates are present in Appendix A.

When the client keys and certificates are created, they should be placed in the clientX/certs folder and should be named as clientX.key and clientX.crt

3S Implementation Details

After a 3S server starts, a client node can make requests to the server. Let's assume that client nodes have a discovery service that allows them to find the hostname where 3S runs. The hostname, in this case, is **secure-shared-store**. The certificate for the server contains **secure-shared-store** as the common name of the server. Whenever the client node makes a request, mutual authentication is performed, and a secure communication channel is established between the client node and the server. Here we make use of nginx to perform mutual authentication (MTLS). Every request from the client node should include the certificate of the client node for authentication.

As mentioned before, the 3S service should enable functions such as login, checkin, checkout, grant, delete, and logout. You will have to complete the skeleton code provided for the server and client to achieve these functionalities. Details are as follows:

1. login(User UID, UserPrivateKey): This call allows a client node to generate necessary statements to convince the 3S server that requests made by the client are for the user having UID as its user-id. The client node will take UID and UserPrivate key as two separate inputs from the user. The filename of the key is to

provided as input as opposed to the key value itself. A user's private key should only be used to generate the necessary statements and then erased. The statement should be of the form "***ClientX as UserY logs into the Server***" where X represents the client-id and Y represents the user-id. On successful login, the server should return a unique session-token for the user. The session token will have to be included in all the subsequent requests and would play the role of the statement in those requests. Also, you must ensure that each user has a unique UID. You can assume that a given client node only handles requests of a single user in one session (if a user logs in successfully from another client, the previous session will be invalidated).

2. checkin(Document DID, SecurityFlag): A document with its id (DID) is sent to the server over the secure channel that was established when the session was initiated. If the document already exists on the server, it may be overwritten along with its meta-data. If a new document is checked in, the user at the client node becomes the owner of the document. The owner does not change if the document is updated (using checkin) by an authorized user who is not the owner. You can make use of any scheme to ensure that the documents created by different users have unique DIDs. The *SecurityFlag* specifies how document data should be stored on the server. The documents that are to be checked into the server must be present in the documents/checkin folder within the client directory [It is already created within client1]. On the server, the documents that are checked in must be stored in the documents folder within the server directory.

When the Security Flag is set as *Confidentiality* (to be represented by "1"), the server generates a random AES key for the document, uses it for encryption and stores data in the encrypted form. To decrypt the data at a later time, this key is also encrypted using the server's public key and stored with document meta-data. When the Security Flag is set as *Integrity* (to be represented by "2"), the server stores the document along with a signed copy.

3. checkout(Document DID): After a session is established, a user can use this function to request a specific document based on the document identifier (DID) over the secure channel to the server. The request is granted only if the checkout request is made either by the owner of the document or if performed by a user who is authorized to perform this action. If successful, a copy of the document is sent to the client node. The server would have maintained information about documents (e.g., meta-data) during checkin that allows it to locate the requested document, decrypt it and send the document back to the requestor. Once the document is checked out, it must be stored in the documents/checkout folder within the Client directory.

When a request is made for a document stored with *Confidentiality* as the *SecurityFlag*, the server locates the encrypted document and its key, decrypts the data and sends it back over the secure channel. Similarly, when a request is made for a document stored with *Integrity* as the *SecurityFlag*, the signature of the document must be verified before sending a copy to the client.

Additionally, when a request is made to **checkin** a document that is checked out in the current active session, the client must **move** (not copy) the document from the "/documents/checkout" folder into the "/documents/checkin" folder. The client implementation must handle the transfer of these files between the folders automatically.

4. grant(Document DID, TargetUser TUID, AccessRight R, time T): Grant can only be issued by the owner of the document. This will change the defined access control policy to allow the target user (TUID) to have authorization for the specified action (R) for the specified document (DID). AccessRight can either be checkin (which must be represented by input 1), checkout (2) or both (3) for time duration T (in seconds). If the TargetUser is ALL (**TUID=0**), the authorization is granted to all the users in the system for this specific document. If there are multiple grants that have been authorized for a particular document and user, the latest grant would be the effective rule. Basically, **the latest grant for the tuple (DID, TUID) should persist**.

Here are a few clarification scenarios:

- If an initial grant for (file1, user1, 2, 100) is successful and then a successful grant request (file1, 0, 1, 50) is made, then **file1** should be accessible for **checkin only** to **all users** for **50 seconds**. User1 loses the checkout access given earlier.
- Grant (file1, 0, 3, 100) exists and then a successful grant request (file1, user2, 2, 50), then **file1** is accessible to **user2** for **checkout** for **50 seconds** and to **all other users** for **both checkin-checkout**. The access rights for other users should not be modified.

5. delete(Document DID): If the user currently logged in at the requesting client is the document owner, the file is safely deleted. No one in the future should be able to access data contained in it even if the server gets compromised. The deletion of a confidential document should result in permanent removal of the key used to encrypt it.

6. logout(): Terminates the current session. If any documents received from the server were modified, their new copies must be sent to the server before session termination completes. While checking back in the modified documents, you must set *Integrity* as the *SecurityFlag*.

Since this is a security class, you should use secure coding practices. You are also expected to use static code analysis tools such as Pylint, Pyflakes, etc. and minimize the use of unsafe function calls (justify any such calls you need to make by providing inline comments). The report should list tools used to ensure that your code does not have any vulnerabilities. The report should also discuss the threat model and what threats are handled by your implementation.

Project Deliverables

1. Report. It should cover the following aspects (Each answer need not be more than a few sentences):
   - Architectural design details:
     - How mutual authentication is achieved in the current implementation of 3S.
     - Details on the cryptographic libraries and functions used to handle secure file storage.
     - How the user information and metadata related to documents were stored.
   - Implementation details:
     - Details of how the required functionalities were implemented
     - List any features that were not implemented or tested (partial points may be awarded).
     - List the assumptions made, if any.
   - Results of the static code analysis and the tools used.
   - Threat Modelling and the threats currently handled by your implementation.
   - Your report should be named as **Report.pdf**
2. Server code
   - This will be the completed version of server.py that was provided.
   - This must be named as **server.py**
3. Client node
   - This will be the completed version of client.py that was provided.
   - This must be named as **client.py**
4. Requirements
   - This should include all python modules used in your implementation.
   - You can use the command `pip freeze > requirements.txt` to generate the file

- It should be named as **requirements.txt**
- This will be used by the auto grader to replicate your environment, so ensure that the command `pip install -r requirements.txt` would install the required dependencies.

Please ensure that you do not zip the files in your submission. Also, please stick to the specified naming conventions since an auto grader would be evaluating your submissions.

IMPORTANT: Please ensure that you submit only these 4 files that are mentioned and follow the specified naming conventions. Any error in adhering to these guidelines would result in an error with the autograder and would result in a significant loss of points.

Additional Instructions

- Please go through the comments in *server.py* and *client.py* and follow the provided instructions. Ensure to complete the sections where TODOs are specified. You can add utility functions as required.
- All requests sent from the client must make use of the post_request() utility function and do not modify this function. A sample response format is given in the login() function within the server code. Feel free to make use of the same for the other server function.
- Expected response status codes for different scenarios are provided in the server.py for each function. Ensure that the completed code behaves accordingly since the auto grader would use the status for verification. (Status codes provided are custom status codes and not the standard ones. Using custom status codes in HTTP is not best practice, however, this is being used for the purpose of the auto grader)
- Failure to follow the provided instructions would result in unnecessary point loss.
- Run the script `start_server.sh` present in the server directory to start the server. It essentially invokes server.py to start the server.
- Make sure that the status of nginx is active by using the command – `systemctl status nginx`. If the status is not active, you can restart it using the command – `sudo systemctl restart nginx`
- Ensure that your implementation can be run and tested by just invoking the `start_server.sh` and *client.py*. Your code must also automatically initialize the required databases, if any and this is necessary so that the autograder can successfully be run. Additionally, ensure that your code would create any folders as necessary by your implementation since we would be using just the two python scripts for evaluation.
- No specific test cases will be provided for this project and you are free to develop a test harness that consists of a sequence of calls made to the 3S server. However, we have provided a basic testing script (execute_test.py) to give an idea of how inputs are to be provided to the client. Complete the TODO comments in the file and replace variables accordingly. Please ensure to use the testing script on your implementation to verify that your implementation would be accepted by the auto grader.
- In addition to the testing script that is present in the VM, we are also providing a sample test harness that was developed by a student last semester. However, we provide no guarantees when using this prior test harness since it was developed by a former student, and this would not cover the test cases that we would be using. This can be found at this link - https://github.gatech.edu/gist/jjames78/a72947a0194f2471c21be6022fec1863
- Make sure to test your 3S implementation using at least 2 clients and 3 users. **The autograder would be tested with users named 'user1', 'user2', 'user3' [these are the UIDs] and clients named 'client1', 'client2'. Please make sure your implementation would be able to support these. The auto grader would also expect userX.key and userX.pub as the private and public keys for those users. So, ensure to test with these three users and create any metadata (in your application and database) as required to support this mapping with the necessary files, keys or paths.**

IMPORTANT: Do not hardcode the public or private key names (eg: user1.key or user1.pub) in your code. Make sure the usernames and keys are all in lowercase only.

- When the client keys and certificates are created, they should be placed in the clientX/certs folder and should be named as clientX.key and clientX.crt **(this is an important setup step.).** These must be used in client.py when the post_request() is invoked.
- While sending requests some of you might encounter SSL errors and to avoid this issue, please have a look at the python package **certifi** for this. (You can use this link as a reference - https://incognitjoe.github.io/adding-certs-to-requests.html)
- As mentioned earlier, this project will be graded by an autograder so please ensure to follow the guidelines mentioned in this file. However, in the off chance that the autograder would fail for your submission, we require that you submit a video that walks through the implemented functionality and this video submission will be graded for partial credit. The steps required in the video are present in the **Video Requirements** section of this document. This recording should be added as a **media comment** to your submission and can be of any common video format.
- We encourage you all to discuss the project at a high-level on Piazza. Please ensure that you are not over-sharing and maintain academic honesty. Halfway through the project, if there are many common doubts, we will consolidate the clarification posts and share it as a note.

Grading Outline

Report - 30 points

- Architectural design details - 5 points
- Implementation details - 15 points
- Security Analysis of the implemented secure shared store - 5 points
- Threat Modelling - 5 points

Implementation of 3S - 70 points

Each function in the implementation will be scored as below:

1. Login - 10 points
   - Handling the private key of the user and verifying the signature of the created statements
   - Generation of a session token used for further requests
2. Checkin - 15 points
   - Secure file transfer of documents
   - Handling the security flag - Integrity and Confidentiality
   - Ownership/ Authorization check
3. Checkout - 15 points
   - Secure file transfer of documents
   - Handling the security flag - Integrity and Confidentiality
   - Ownership/ Authorization check
4. Grant - 15 points
   - Granting authorization to other users
   - Handling expiry of granted access (in seconds)
5. Delete - 10 points

- Ensuring deletion of files
- Ownership/ Authorization check

6. Logout - 5 points
    - Checking in all the modified checked out files and session termination

Video Requirements

As mentioned earlier, this project will be graded by an autograder so please follow the guidelines mentioned in this file. However, there is an alternate solution in the event that the autograder fails for your submission due to any reason. This video (a screen recording) will be required to be submitted as part of your submission and will be then graded for partial credit (only if the autograder fails). This must be added as a media comment on your submission and can be of any common video format.

The following steps will be required to be shown as a part of your video:

- Download your latest submission from Canvas
- Walk through the 6 functions that are mentioned as a part of the Implementation requirements
- Login as user1 in the first terminal and then checkin any two documents, one with the Integrity flag and the other with the Confidentiality flag.
- Login as user2 in a second terminal and attempt to checkout the first document that was checked in by user1 (This would not be allowed due to invalid permission).
- In the first terminal, user1 must grant checkout access to user2 for the second document for 20 seconds. In the second terminal, user2 then attempts to checkout the second document during these 20 seconds and should be able to checkout the document.
- In the second terminal, user2 should attempt to delete the first document created by user1 (This would not be allowed). In the first terminal, user1 attempts to delete the first document created (This should be allowed).
- Logout as users - user1 and user2 from the respective terminals.
- The entire duration of this **video must not exceed 10 minutes**.

APPENDIX A

Certificate Generation:

The resource below describes how to set up a Certificate Authority (CA) and then how it's certificate would be used to generate certificates for the nodes.

- https://deliciousbrains.com/ssl-certificate-authority-for-local-https-development/

We have already set up a CA. You can find the CA certificates in the CA folder of Project4. We have also generated the server keys and certificate (certname is **secure-shared-store**) using the CA certificate. Also, the following command was used to extract the public key from the certificate.

```
openssl x509 -pubkey -noout -in secure-shared-store.crt > secure-shared-store.pub
```

You can use the above resources to generate certificates and keys for the client nodes and users.