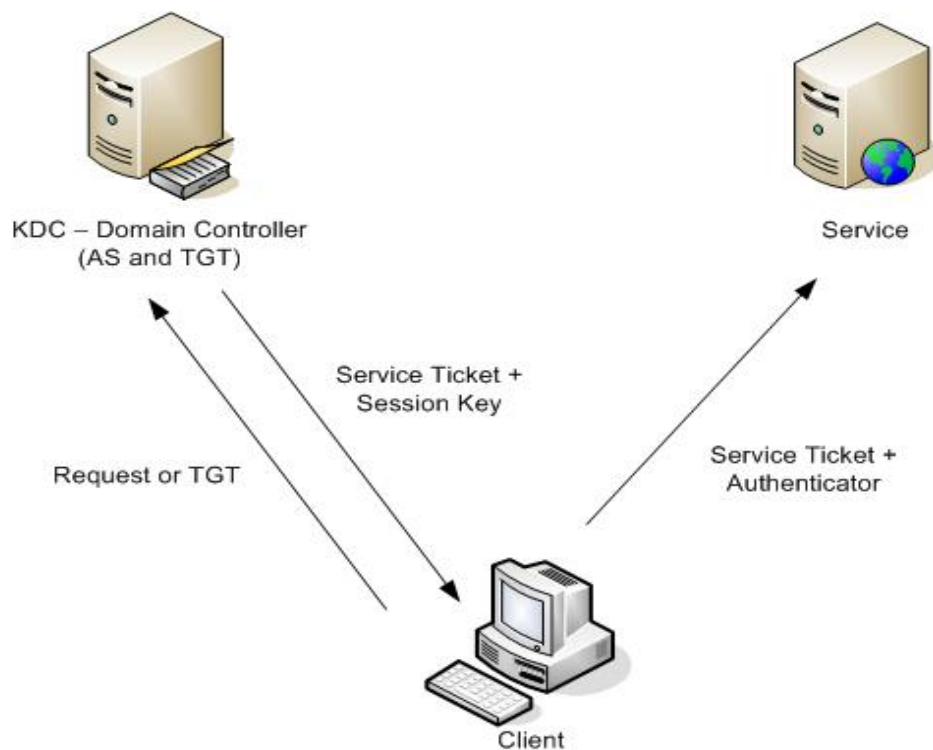


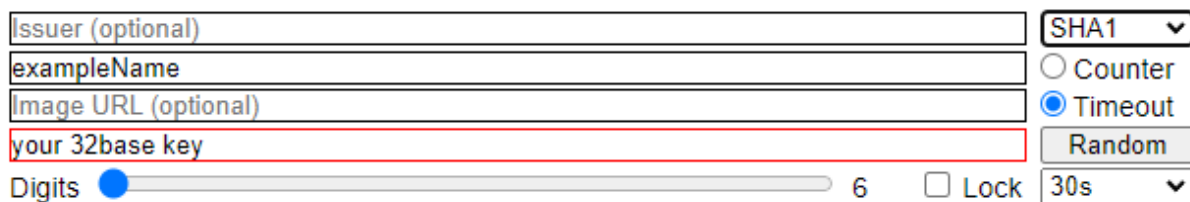
*Cloud Service with Searchable Symmetric Encryption Protocol*

In this documentation of our final project, the topic of searchable symmetric encryption is discussed. This protocol, combined with various others, provides a user with a safe and optimized form of file sharing. Because it is important to understand the differences in how protocols interact with each other, we urge any internet goer to truly understand what it means when companies brag about their security standards.

Now, before we could implement searchable symmetric encryption, we needed a secure foundation. If we were to take away anything from network security, it would be to not try and create our own protocol but to follow tried and tested security primitives. With this in mind, we decided to go with an implementation of Kerberos as that foundation. For reference this is what our Kerberos diagram looks like:



What this diagram really means in our protocol is that there are three stages. First, the client makes a connection to the Authentication Server. Here, the client can perform two actions, either create or authenticate an account. New users will be required to create an account and are given a base32 key to generate a one time password. The flow for such is as is; enter username and enter password. This results in a base32 key they can use to plug into the FreeOTP [website](#). Keep in mind, the following selections have to be made in order to avoid any issues with the OTP:



The image shows a web form for configuring a TOTP application. It includes input fields for 'Issuer (optional)', 'exampleName', and 'Image URL (optional)'. A red box highlights the 'your 32base key' field. To the right, there is a dropdown menu set to 'SHA1', radio buttons for 'Counter' and 'Timeout' (with 'Timeout' selected), a 'Random' button, and a dropdown menu set to '30s'. At the bottom, there is a 'Digits' slider set to 6, a 'Lock' checkbox, and a '30s' dropdown menu.

Now, in the case of authentication, 2FA as hinted, will be used. This means that the user enters something they know and something they have. In this case, it is the client's password and the OTP. The password and OTP are verified by checking the server's passwd.json file which contains a TOTP key, hashed password, salt, and associated user. To perform the verification, the server must calculate the OTP on its own, as well as the password hash, and check that they match what it has received. That concludes the first part of Kerberos.

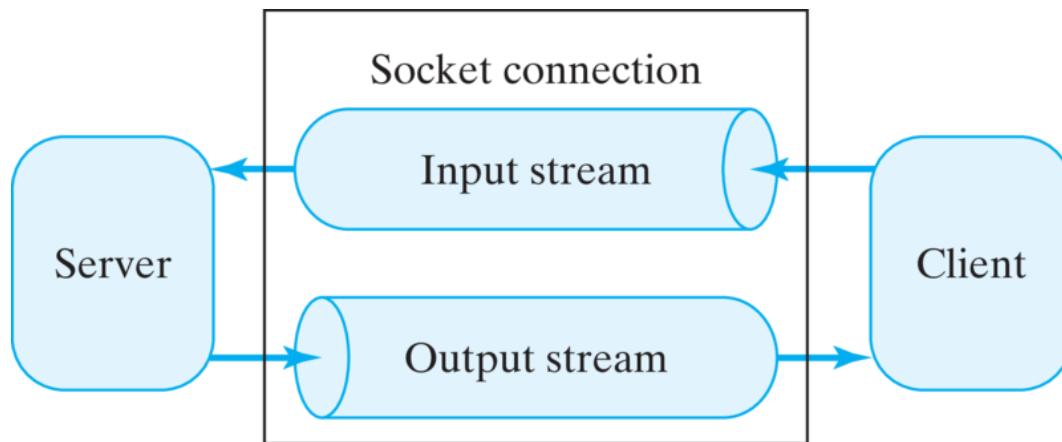
The second step is the client requesting a ticket in which it can use to interact and establish a secure connection to the file sharing service. The first step to this protocol is the client requesting which service it would like to use. The authentication server then will generate the actual ticket containing two identical session keys, one encrypted with the client's master password and the other with the service's master password. There are among a few other things such as creation time, validity time, and the iv used for encryption. Creation and validity are used to set a time limit on the amount of access said client has with the service, this ensures that a client never has unlimited access time.

Now, the third and final step goes back to the two identical session keys. The client will attempt to decrypt his end to obtain the session key as well as send off the ticket containing the other one to the service. This is the beginning of the handshake protocol, seen here:



In essence, the handshake critical provides mutual authentication; the client proves itself to the server and the server proves itself to the client. In the above figure it is done in three steps. First, the client sends the nonce and ticket as previously mentioned, here the server can obtain its session key by decrypting it. This works because the key was encrypted using the server's master password. Now, the server will encrypt the nonce it received and send it along with a fresh one. The client will receive it and verify it proving to the client that the server knows the session key, otherwise it wouldn't decrypt properly. Finally, the client encrypts its received nonce and sends it along with a fresh nonce. When the server receives it, the nonce will be decrypted and verified against the one it sent., At this point, the client has proved its identity to the server.

At this point, the user has proven themselves to the authentication server, received a ticket to its service, and completed a successful three-way handshake with the service in order to verify that both ends know the session key. This is where the client and service can move into the communication phase:

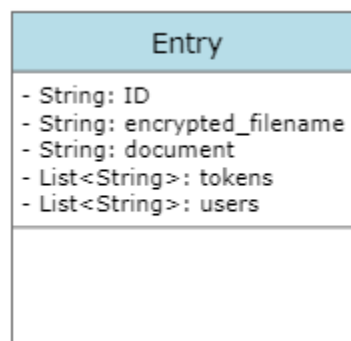


This diagram simply demonstrates that during the communication phase, the client will make an input and the server will respond with an output. In our example we are using TLS 1.3 and session key encryption.

At this point, the client and server are able to securely communicate.

## Cloud Service

Our cloud service works in a very simple way. We have a set of documents that users have uploaded to the server. We store these documents using JSON into our `./database.json` file located in the cloud server directory. Each document is represented in the database as an Entry. The following UML for the Entry model.



It is important to note that the String ID is the only attribute that is computed server-side. The tokens, users, and documents are all forwarded to the server from the client. Thus allowing the client to make any modifications, encryption, or encoding schemes of the attributes as long as they follow the Data Structure requirements for storage.

We have constructed a client-project that does modifications under the following protocol.

### **File Create**

When a user wants to create a file (add it to the cloud service) they follow the given protocol; managed by the client project.

1. Authenticate with username and password to the KDC server.
2. Give a path to the file, a list of keywords, and a list of users who are permitted access.
3. Enter a **file-password** which will be needed to gain access to the file later on and derive the key in step 4.
4. File-password is hashed using SHA-256 and a key is derived using Scrypt. Resulting in a **FP-Key** and a **FP-IV** which will be used for Symmetric Encryption.
5. Encrypt each Keyword using FP-Key and FP-IV, represented by a List<Token>, tokens.
6. Encrypt the File using FP-Key and FP-IV resulting in a **cipher\_file**.
7. Encode cipher\_file to a String using Base64 resulting in **encoded\_file**.
8. Encrypt File.name using FP-Key and FP-IV resulting in **encrypted\_filename**.
9. Upload encoded\_file & encrypted\_filename & Token's to the database with the associating keywords.

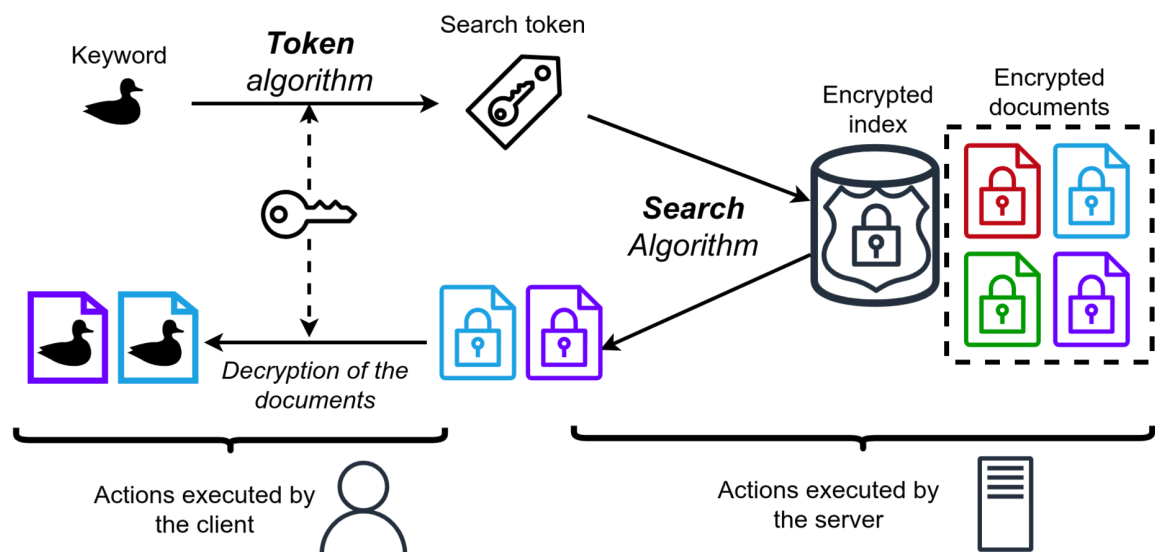
Once completed a file has been created and uploaded to the cloud-service, stored in the database. We manage a file uploaded as an Encrypted Document; modeled as followed:

EncryptedDocument
<ul style="list-style-type: none"> <li>- String: ID</li> <li>- String: encrypted_filename</li> <li>- String: encoded_file</li> <li>- List&lt;String&gt;: users</li> </ul>
equals(Object o)

## Searchable Symmetric Encryption

Our cloud service and client projects provide the ability to search for a document without revealing any information about the search query or the document being located. The cloud service trusts that any encoded\_files and keyword pairs being sent are encrypted using a File-Password (or some symmetric encryption scheme) where we represent FP-Key and FP-IV.

Searchable Symmetric Encryption (SSE) allows one to efficiently search over a set of documents without the ability to decrypt them. Allowing a client to send files to an untrusted server and allowing the server to search over the files without ever being able to read the file contents or know what the file search query entails. I have given a diagram representing our model of SSE.



We manage our EncryptedDocuments in a DocumentCollection, modeled as follows.

DocumentCollection
- HashMap<List<Token>, EncryptedDocument>: index_table
+ EncryptedDocument: Insert(List<Token> tokens, EncryptedDocument doc) + List<EncryptedDocument>: Search(Token token) + void: fromDatabase(List<Entry> entries) + List<Entry>: toEntries()

The Document Collections main purpose is to provide an index\_table which is a Key-Value Pair DataStructure holding a list of Tokens (encrypted keywords) mapped many-to-one to a single EncryptedDocument object. Multiple documents can share similar keywords. We choose the document presented to the client by the following properties: the user has access to the file, the file has the ability to be decrypted by FP-Key and FP-IV, and the encrypted keyword was found in the index\_table for the associating documents.

I claim that we provide Searchable Symmetric Encryption because both the files and keywords are encrypted, giving the cloud service no ability to decrypt the information from the documents. The search protocol is enacted by the client as follows.

## File Search

1. Authenticate with username and password to the KDC server.
2. Give a path for where the files should be returned, and a list of searching keywords.
3. Enter a **file-password** which will be needed to gain access to the file.
4. File-password is hashed using SHA-256 and a key is derived using Scrypt. Resulting in a **FP-Key** and a **FP-IV** which will be used for Symmetric Encryption.
5. Encrypt each entered Keyword using FP-Key and FP-IV, represented by a List<Token>, tokens.
6. Query the **index\_table** for any EncryptedDocuments matching any of the Tokens.
7. If the user is permitted to any of the EncryptedDocuments return the first to the client.
8. Client decodes **encoded\_file** using Base64.Decoder into **encrypted\_file**.
9. Client uses FP-Key and FP-IV to decrypt **encrypted\_file** into **plaintext\_file**.
10. Client decrypts the **encrypted\_filename** using FP-Key and FP-IV into **plaintext\_filename**.
11. Client saves the plaintext\_file, locally, to the given file path, file is named plaintext\_filename.