Securely Sharing Files

210032780

1 Overview

The development of a cryptographically-secure file server aims to address many of the privacy and control challenges associated with cloud storage. Leveraging zero-knowledge encryption, the system ensures that "useful" data remains inaccessible to the server, mitigating risks such as unauthorised access and DDoS attacks. The server facilitates secure file storage and retrieval while incorporating layered cryptographic mechanisms for individual and group-based access control, enabling users to share files, and revoking these files when necessary. This solution demonstrates one approach in enhancing data security, privacy, and autonomy in cloud-based environments.

2 Design

The design for this zero-knowledge architecture revolves around the multi-layered encryption of files being sent to, and retrieved from the server. In order develop a system whereby other users (or clients) can have access to another user's files, a single layer of encryption, done through either RSA keypairs or AES symmetric encryption, proves not as successful. Using AES to encrypt files and RSA to encrypt the AES key performs better than using either in isolation as it addresses the inefficiency and scalability issues of RSA when handling large amounts of data. RSA encryption is very computationally intensive and thus significantly slower than AES, making it impractical for encrypting large files directly. Additionally, RSA has limitations on the size of data it can encrypt, which is typically much smaller than the size of most conventional files. By using AES for the file encryption, which is optimised for large data sizes, and RSA solely for encrypting the smaller AES keys, this hybrid approach achieves both high performance and secure key distribution, making it far more efficient and practical for real-world applications.

This client-server communication is achieved over TLS connection. This is achieved using TLSv1.3, the current latest possible version of TLS as it provides faster handshake and importantly, a more robust security than its predecessors. (Patil, 2023) TLSv1.3 enforces forward secrecy, a trait not present in earlier versions, this mandates the use of Diffie-Hellman to generate one-time session keys for the ongoing network session, which are discarded after use. This proves particularly beneficial should a single session key ever be compromised, as attackers will not be able to access the rest of the server's communications, thus behaving as an effective source of resistance against brute-force, and man-in-the-middle attacks.

2.1 Assumptions

- **Useful information considerations**: One aspect of a zero knowledge architecture is what information is considered to be useful, specifically by a third-party with malicious intent. As such, the decision here was to encrypt all pieces of information server-side, extending from all the file data to the directories themselves
- **Directories emulating physical storage locations**: The entirety of this coursework security model resides within a single directory, this is under the assumption that in a real-world scenario each client would be accessing the server from a different physical device, and accessing information related to the certificate authority, as well as public keys, via the web.
- **Group file directories**: Expanding on the idea that all common-knowledge data can be accessed by both the clients and server, the choice to leave group names (and thus their members) unencrypted and accessible without making a request to the server was made. The reason for this choice stems from the idea that if a client knows who they wish to share a file with, they will also know the members of the group assigned by the server
- **Username and password storage**: All client username and password pairs have been hardcoded into client code for testing purposes. It will remain like this (and same for the server passcode) as it does not concern the architecture of the TLS connection. It is also assumed that all clients have no knowledge of each other's passwords

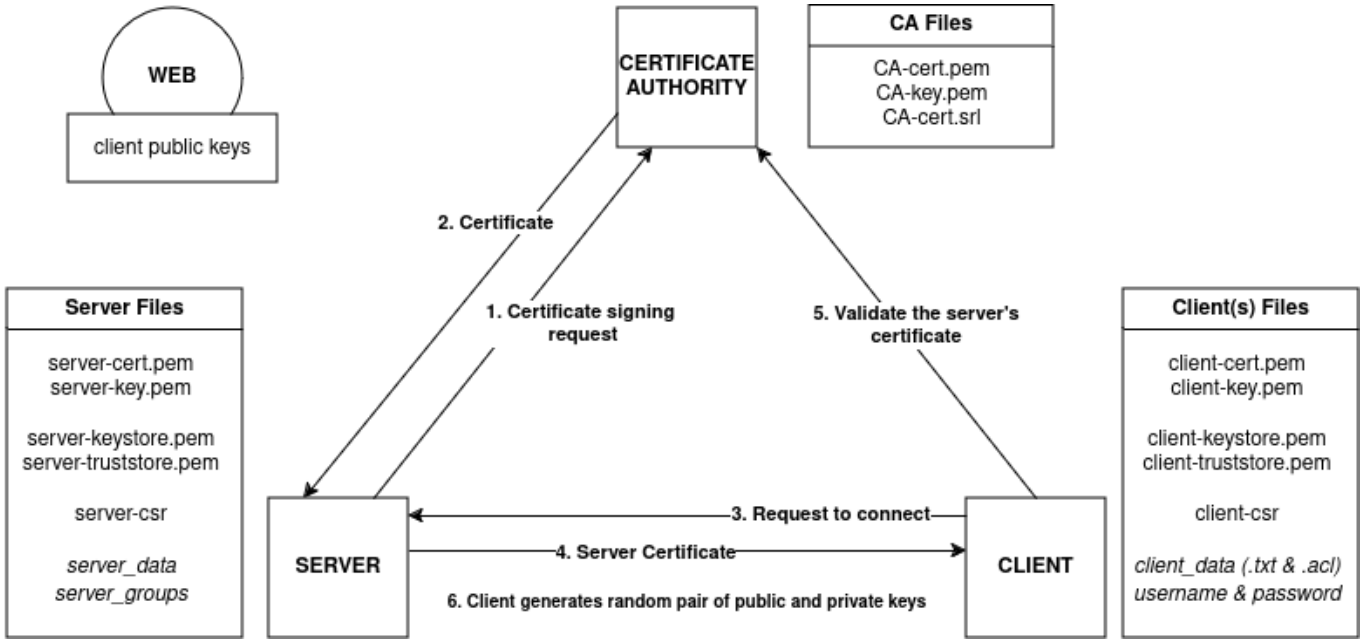

Figure 2.1: TLS Connection Setup

- **Naming conventions**: As it does not directly impact the objective of developing a secure file server, for ease of testing, it is assumed no two clients share the same username, nor can a single user have a two files of the same name
- **File awareness**: As to maintain a zero-knowledge architecture, no file names will be stored on the server, thus clients wishing to request a file will have to know its name prior to sending the request. As the hashing is not reversible, clients cannot request access to a file list as it will be fully populated with hash strings
- **Deprecated bash scripts**: As of the time of testing this [18/11/2024], all bash scripts are working as expected, however these are known to deprecate and change over time, potentially impacting certificate/key generation at a later date

2.2 TLS Handshake

This implementation begins by establishing and populating the key-stores and trust-stores on both the client and server side. Doing so makes use of an artificially created certificate authority as our single trusted third-party. Establishing a server-client begins by first running the necessary bash scripts to produce an OpenSSL X.509 certificate signing request for the CA, and in response receiving a certificate-key pair for the server. The private key of the server is then placed into a key-store, while the CA certificate is held in the trust-store of the server. The same process is done but on the client side; when a client-server connection is established, and authentication of the client is required for mutual TLS, the server is able to verify the client's certificate against the CA certificates in its trust-store to ensure the client's authenticity. The key stores, are required during the TLS handshake, for encryption and decryption with the public and private key respectively. (See Figure 2.1 For TLS communication and file storage)

In the alternative approach, self-signed SSL certificates are created, issued, and signed by the entities whose identities the certificates are meant to verify. The very nature of this means that hypothetical organisations making use of this file-server are signing off on themselves, and doing so using their own private keys. The metadata offered by self-signed certificates provide no *trust value* and thus cannot be used as a reliable source of authentication. A CA, as a trusted third party, issues certificates after verifying the identity of entities, thereby establishing a chain of trust. This verification process ensures that certificates are not only valid but also trustworthy, as they are backed by a "real" CA's reputation and adherence to certain security standards. Self-signed certificates could also be susceptible to man-in-the-middle attacks, where an attacker could create a fraudulent certificate to intercept communications, as there is no external authority to confirm their legitimacy.

2.3 File Transmission

Following the authentication of the client, a connection is established between the two parties. The server, to maintain a zero-knowledge structure, is limited to strictly receiving files from the client and sending files to the client. As it is blindly carrying out these two operations without any means of checking the files, it has to be ensured the all data is correct and permitted, prior to its transmission. Data is only sent in an encrypted format, in order to transmit this securely and quickly, a dual layer incorporated.

- **AES Encryption**: Encryption of the file itself, as it has the potential to be exceedingly larger than any keys is accomplished using AES encryption. Traditionally, this form of encryption sacrifices security for quicker encryption, however, this downside can be restricted by implementing an additional layer of security in the form of an initialisation vector (IV) - a series of random bytes. The choice of transformation here is **AES/CBC/PKCS5Padding**, this form of AES XORs each block of plaintext against the previous, with the first against the initialisation vector. Here, the IV behaves as a salt for the plaintext, ensuring that identical blocks of plaintext produce distinct ciphertexts, thereby preventing attackers from brute-forcing pattern recognition of encrypted files. In this implementation, the size of the IV is fixed at 16 bytes for ease of transfer, however this can also be made random, adding further security to the transmission. Alongside the random IV, the AES key itself is also newly (and randomly) generated for each new file.

  Padding here is required to ensure the file can be divided up into a multiple of the cipher's block size.

- **RSA Encryption** Established by AES, two random pieces of information are required to symmetrically encrypt and decrypt the data, the secret key and the IV. These two files are transmitted to the server (alongside the encrypted file) encrypted via the client's public key using **RSA/ECB/PKCS1Padding**. Upon requesting a file from the server, the file, as well as the IV and AES key are supplied to the client; if the client has the permitted private key, they will be able to decrypt the IV and AES key, and subsequently decrypt the file

- **SHA256 Hashing**: As an additional layer of security in the zero-knowledge model, all files, file-keys, and IVs are hashed and converted to hexadecimal (as to avoid unwanted UTF-8 characters) prior to being sent to the server. The hashed files names are a composite of the client's username (or the name of the user who the file is shared with), the owner's password, and the name of the file itself. This way, multiple copies of the same file can exist on the server without causing any clashes. Specifically hashing the files with the password of the owner prevents any other clients in the network from maliciously or accidentally overwriting the owner's original file using the same combination of username and filename.

  Within the server file directory, I have made the decision to omit any subdirectories, and instead keep all files within the same folder. This choice was made under the intuition that a malicious third party knowing what to look for, based on the directory names, would make an attack much simpler to execute on. This, paired with hashed filenames makes knowing what to search for an infeasible task, and furthermore, the attacker must be able to retrieve all three correct files in order to reconstruct the plaintext, all of which would have no correlation on the server-side.

2.4 File Sharing

The process of sharing files with other clients proved to be the most difficult to execute feasibly in a zero-knowledge environment, different implementations were considered such as server-side access control lists and role based access control, however ultimately, a solution whereby all information of the ACL/RBAC could be securely stored in an encrypted manner was not found without its distinct flaws.

The process of granting access to a file, whether it be an individual or a group, involves locating all the public keys of the sharing party and encrypting the AES key and IV using RSA public key encryption. These files, alongside a copy of the owner's encrypted file are all sent to the server for storage; when another client granted access to this file wishes to then retrieve their copy of the file, they would only be able to successfully obtain it if there exists their corresponding hashed filename, and once the file is downloaded, they must then be able to decrypt it using their private key. User's granted access to the file will be able to retrieve the file (and its dependencies) based on a slightly modified hashing of the filenames. Owner's of the file have the inclusion of their hidden password as part of the hash for their files, whereas users granted access permission have their copy of the file with this detail exempt from the hash. This implementation of an ACL thus means there exist only two roles for users: Owner and Shared.

As ACLs are file specific, and stored in the local directories of clients, an owner's permissions give them full control over the users in the access control list - Appending and revoking permissions. The latter of these will send a

request to the server to delete the three-tuple matching the hashed filenames of that user. This can be extended to groups as well, propagating this request to the server over all members of the group. In addition to revoking controls, an owner wishing to delete a file can also do so, eliminating all existing copies of it from the server, for all who have access to it. While this will not stop users with previous access from downloading the file, modifying it and reuploading it at a later date, it will prevent all immediate access to the file on the server.

3 Limitations

3.1 Access Control Lists

The limitations of this secure-server implementation revolve predominantly around the sharing of files between users. Here we are restricted to two forms of access control - owner and shared recipient. As it currently stands, sharing files with other groups or users provides them all with the same permissions over the file, and any changes made to the file will be reflected solely in their copy of the file and no where else. In a real-world scenario this would equate to requiring an announcement to be made should all recipients of the file require their version of the file to be updated. While this does present its own inconveniences, it allows all owners and recipients to manage their own access controls (and future permissions) as they see fit, thus offering some flexibility.

3.2 Public Key Infrastructure

As public keys are assumed to held at an accessible location online, without any verification of the keys as they are retrieved, a substitution attack could be performed during the transmission process of the public keys, whereby the attacker would place in their own. Unbeknownst to the client, these files would be encrypted using the attackers public key and sent to the server, whereby they can then be intercepted and decrypted for the attacker's purposes.

3.3 Passwords and Private Keys

Possession of a client's private key gives the attacker full access to decrypt all files pertaining to that user, and any files shared with that user, if an attacker is able to intercept outgoing files from the server, it would be possible to record the outgoing traffic until the private key is capable of decrypting files directed to that client. Equally, possession of the client's password allows an attacker to upload files to the server while impersonating the client. As the generation of private keys in the bash scripts are currently configured to use the client's password as the passkey associated with the private key, attackers would thus be able to obtain access to the incoming and outgoing files of the user; this one piece of information is capable of fully compromising the integrity of the system for any one participant.

3.4 Digital Signatures

This implementation lacks any form of tamper-mitigation or tamper-checking, without tools such as digital signatures, attackers could alter encrypted files without detection or perform replay attacks by resending old messages, thus appearing entirely secure on the client's end, while in actuality not receiving the intended data. Developing on the ACLs, should they be improved, this issue could also be combatted by storing a log of all changes made to the file in the ACL as it is transmitted alongside the file.

4 Evaluation

Reflecting on this model, it is capable of executing all the necessary tasks of a cryptographically-secure server. In regards to being a zero-trust-architecture, I believe to have covered this to the full extent; the server stores absolutely no information of value, ensuring all files, filenames, and cryptographic data is unintelligible to those without the correct access permissions. With more time, there is certainly room for improvement in regards to the flexibility of the sharing procedures, and the access controls provided to groups/individuals. Introducing new roles such as for reading and writing, whereby the former users would receive copies of the file, and the latter would receive direct access the owner's file would be the first aspect of the implementation I would consider. Developing from there, additional permissions could be added, such as an admin role, which would allow users to modify the permissions of other clients below in the hierarchy.

5 Bibliography

1. Patil, K. (2024) *Why is TLS 1.3 better and safer than TLS 1.2?*, AppViewX.
   Available at: https://www.appviewx.com/blogs/why-is-tls-1-3-better-and-safer-than-tls-1-2/ (Accessed: 18 November 2024).
2. Programming weblogic security (2008) *Using SSL Authentication in Java Clients.* Available at:
   https://docs.oracle.com/cd/E13222$_0$1/$wls/docs$103/$security/SSL_client.html$ (Accessed: 18 November 2024).