TÉCNICO LISBOA

Instituto Superior Técnico

Network and Computer Security

Ransom-Aware

Ransomware-resistant remote documents

Alameda - Group A37

89434	Diogo Ravasco	
89471	João David	
91004	Daniel Gonçalves	

Problem

The challenge of this problem consists of designing a secure system where only users with permission can share, view, and edit files. This requires not only encryption on the communication channels, but also on the server's filesystem, to prevent the case where an attacker gains access to the server's machine. Moreover, the server must have some sort of authentication for the clients, which shall be stored securely, so that only then authorization to the documents is granted.

Furthermore, the system must resist ransomware attacks and illegal modification of the documents must be detected, this is a problem that concerns data integrity, for detecting unauthorized modifications to the files, but will also require the files to be stored elsewhere, saving multiple versions of the files, in case the whole filesystem is compromised, or a client is taken over by a malicious user.

Requirements

- Data shared between the communicating nodes must provide the three guarantees: authentication of the server, data integrity, and data confidentiality.
- The data must only be sent to authorized users - Authentication, and Authorization.
- Users' credentials must be saved in a secure way - Confidentiality.
- Only users who have been granted permissions can see the files - Confidentiality.
- Documents saved on the server are only valid if client's signature is valid - Integrity, Non-repudiation
- Multiple versions of the documents must be saved (and their respective signatures) - Backup.

Trust assumptions

- Trusted:
  - The server should be able to trust the backup server.
  - The server's certificate is trusted by all, and must be certified by a trusted CA.
- Partially trusted:
  - The client can partially trust the server. The server is simply a blob store and shall not have access to the files' contents. However, the client must trust the server enough for storing the files, metadata and have a valid certificate for establishing secure sessions.
- Untrusted:
  - The server should not trust any client, and so additional security measures need to be implemented, both for authentication and input validations.
  - The transport layer is not trusted and must be made trustable using protocols such as TLS and SSH.

Solution Developed

Overview

There are three main components on the designed architecture:

- Server
- Backup Server
- Clients

The clients communicate with the server through a secure channel and once authenticated, are allowed to do the operations provided by the service. The server must have a valid certificate, issued by a trusted Root CA. For simplicity, the Root CA certificate is a self-signed locally generated certificate, distributed between the clients.

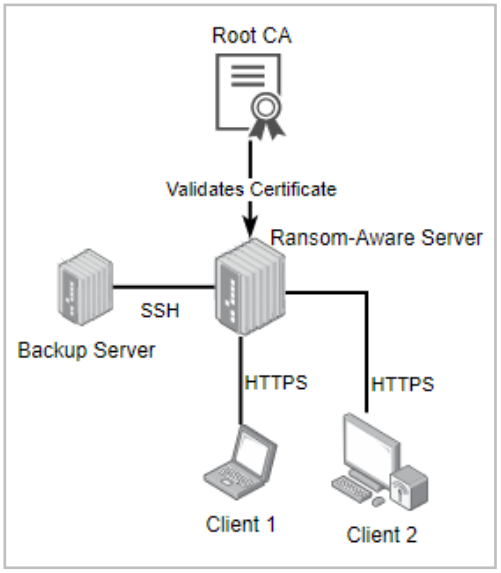

Figure 1 – Solution's Architecture

The files in the server are encrypted using a symmetric key and the data, alongside the file's metadata, is signed by each client when saving a new version, guaranteeing integrity, authenticity, and document freshness, through a signed timestamp of the edition. The key used for file encryption is unique to each file and created alongside the first submission of said file. A new encryption key is created when a user has their permission to access the file revoked. The key used for signing is unique to each client and is provided by the client at the time of registration. The certificate containing that key must also be certified by a CA.

For protecting against ransomware attacks, the server copies its documents to a backup server. The communication channel is secure, providing authentication, confidentiality, and integrity. The backup server also saves the files' metadata, so that the information can be recovered for each user in case the main server is fully compromised.

Deployment

Root CA

The Root CA is hypothetical and represents how we would normally enable the server to authenticate itself. The client hosts would import this entity's certificate, which will then enable them to trust the server. The CA will have a self-signed certificate.

Server

The server handles the clients' requests. It should accept incoming requests on the designated port and act accordingly. The server must authenticate the clients, and the

chosen method is through a username and a password. This password is stored in a database, salted, and hashed to prevent offline attacks. When a client requests a file, the server verifies if the user has access to that file and then, send the file and the encrypted symmetric key to the client.

The server is also connected to a backup server, and, when it receives a new file, it also sends it to the backup server.

The server needs the following components to work:

- A public key certificate for creating the session with the clients.[1]
- A private key for authenticating and establishing a secure session with the backup server.[2]

Backup Server

The backup server is used as a primary backup for the documents.

This component requires a public key from the main server in order to establish a secure session for file sharing, as mentioned in [2].

Client

The clients interact directly with the server. To interact with the server, the clients must establish a secure session, and so, it must import the root CA certificate. Each client gives two certificates to the server, one for encryption and one for signing. When creating a new file, the client creates a symmetric key pair to encrypt the file and guarantee confidentiality. The key is then encrypted with the public key (for encryption) of whoever has access to that file, so that only users with the corresponding private key can decrypt. The clients that receive the file verify the signature by requesting the server the author's certificate and verify if the certificate is trusted.

Secure channels configured

In order to establish a secure connection from the client to the server, we chose to use an HTTPS server. This requires the public key certificate stated above [1]. The files sent from the clients to the server must be encrypted.

To exchange the data between the server and backup server, we use rsync over SSH, using the keys previously distributed [2].

Secure protocols developed

Upon login, the server generates a session token, which is sent (securely) as a cookie to the client. The client sends that cookie on every request and the server uses that cookie for authentication.

The clients do not trust the server with the files' content, and so, the content is sent to the server already encrypted. When a user edits a file, he encrypts it with a symmetric key, and that key is sent to the server ciphered with the public keys of the users that can see the file. The client also signs the documents and metadata so that users (and the server itself) can validate the integrity of those files. Document sharing must be done in two steps at first:

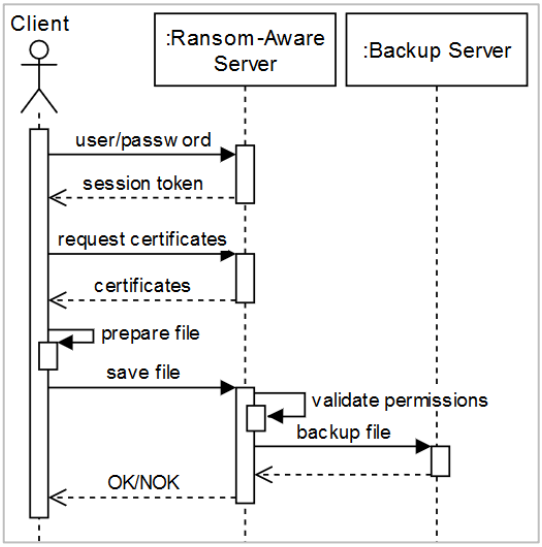

Figure 2 – Login and save file workflow.

- Ask the server for the certificates of users with access to the file, which is sent back to the client.
- The client prepares and the file to the server. The metadata contains the key of the file encrypted with each user's public key, as well as the IV used for encryption, with signature, as seen on figure 3.
- The server saves the new version of the file and the metadata. Old versions are also kept, in case a rollback is needed.

After sharing a document for the first time, the client stores the metadata its metadata. So, on further updates, the client does not need to request the certificates of the users with access to the file. This allows for a quicker interaction between user and server. Whenever the permissions are changed for a file, the client generates a new key and must request the certificates again, like the first time sharing a document. He can also opt to renew the keys for a file even without changing permissions.

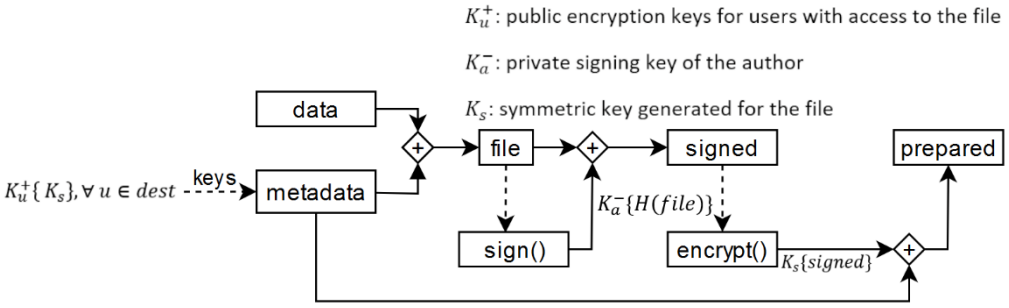

Figure 3 - Document preparation

Results

The final solution was implemented in Java, both the server and the client. We achieved all we had planned and added the ability to rollback file versions. Initially, we thought as the server would sign the files, and so it could verify the integrity by itself. In our final solution, the server is only trusted for storing the documents, and so it would not make much sense for the server to play too much with the data. If a user wants to check the file integrity, he can do it himself when the file is requested, as opposed to making the server run tasks that, in some situations, would be useless.

We achieved what we believe is a secure solution, as we have security mechanisms in almost all system's features:

The main server and the client establish a secure channel through HTTPS, guaranteeing everything that TLS guarantees. When the user registers, the credentials are stored safely, with a hash, a per-user salt and a system-wide salt. The user also provides the public key certificates for encrypting and signing. When the user logs in, he is given a cookie for a small duration of time which is exchanged during the session. We can guarantee authentication and authorization, as the server stores the permissions for each user and each file.

When sending a file, the client creates the necessary metadata, with the ciphered keys, author, and a timestamp. The file includes a signature, so that others can verify it when fetched. We followed a sign-then-encrypt approach, guaranteeing integrity and authentication.

The documents are stored ciphered in the server. A new file is created for each update, allowing the owners of the files to rollback if they find it necessary. Inside the files' metadata, a timestamp is also given, so that users can compare the files that they received with the files they had seen before, allowing to verify freshness. In figure 4, the data field, which is encrypted, contains the file's data, the metadata, and a signature.

Figure 4 - Structure of a document

When a user sends a file, the server checks if the permissions on the metadata field match the permissions previously stored on the server. Even if the deciphered metadata matches the server's, the signed metadata (which is encrypted) is validated when the file is fetched, and so the users cannot trick the system by sending wrong permissions, as clients verify the signature, which includes the metadata. When receiving certificates from the server, the clients can verify the certificate, by assuring that it belongs to the expected user and that the certificate itself was signed by a trusted CA.

Owners of a file can grant permissions to other users so that they can view and edit the files and can also revoke their access to the files. This is reflected directly on the server variables, but, even if they had unauthorized access, new versions of the uploaded file would not be readable by no longer authorized users, as the client generates a new symmetric key when changing a file's permissions.

The server also stores the files and their metadata in a remote backup server, sending it using rsync established over SSH with Public Key Authentication. The data stored on the backup server is enough to redeploy the system in case the main server is fully compromised.

References

1. Nextcloud - client-server software for creating and using file hosting services.
2. Syncthing - file synchronization tool.
3. HTTPS – Java support for HTTPS.
4. rsync4j – Java library for rsync.