Remote document access with ransomware protection

Network and Computer Security

Alameda Campus

Group 41

Afonso Raposo
83805

Guilherme Areias
89451

1. Problem

The ability to send and receive files over the Internet without requiring one of the parties to be connected at the same time greatly simplifies file sharing.
However, this comes with risks. By default, the Internet isn't secure, and without the use of a proper communication channel, this data exchange could be exposed to whoever intercepts the connection. Moreover, if files are stored in a remote server for ease of access at all times, there should be a way to keep their contents secret from unauthorized parties, whether they are users of the system or someone with physical access to the server. This should be done in a way that still allows authorized users to transparently view and edit their files. Even if an attacker can't read the files, they can launch a ransomware attack and encrypt the files on the server. When done correctly, these attacks are nearly impossible to reverse unless one has the attacker's decryption key. As such, protection against this type of attack should be done before it occurs, and it should be detected promptly when it happens to avoid the loss of users' data.

1.1. Solution Requirements

R1. Require user authentication;
R2. Synchronize user documents between the personal device and the remote server;
R3. Allow document sharing between users - synchronize documents across multiple personal devices and the remote server;
R4. Selection of documents to synchronize with server;
R5. Ensure a secure channel between personal devices and remote server;
R6. Documents only accessible by authorized parties;
R7. Register modifications of documents (especially by unauthorized parties);
R8. Periodically backup the documents in a different machine, safe from attackers;
R9. Perform version control on the server;
R10. Intrusion Detection Systems;
R11. Penetration testing the network.

1.2. Trust Assumptions

Entities	Relationships
● User ● Firewall / Reverse proxy ● Auth API (AA) ● Resources API (RA) ● Auth Server (AS) ● Resources Server (RS) ● Backup Server (BS) ● Outside entities/attackers	● The user connects to the firewall machine, which restricts access to the 8443 port using TLS, and acts as a reverse proxy. ● The firewall machine connects to the Auth and RA, allowing only connections through their specific ports; ● The AA connects directly to the AS running a Redis instance; ● The RA connects to the RS running a MongoDB instance and uses SFTP to copy files to and from that machine; ● No machine can initiate a connection to the BS, but this server can initiate an SCP connection to the RS.

2. Proposed Solution

2.1. Overview

Our solution allows users to create projects and to push corresponding local files to a remote server. This is done through a client program, called bag, running in their computer, with which they interact similarly to Git, using commands such as clone, push, and pull.
When they first login with the client, an asymmetric key pair is generated, which is later used for key sharing. At the login, the client receives and stores a refresh token and an access token with a short life window. Users are required to authenticate themselves when accessing the Resources API (RA) using the access token. The RA verifies the access token with the Auth API (AA) and grants access to the user. The refresh token is used to obtain a new access token once the original one expires.
Each project contains its own symmetric key which is generated locally and used to encrypt files before uploading, ensuring confidentiality. This key is shared with collaborators when a user adds them to their project by encrypting it with their public key. To assure integrity of the files pushed and pulled from the server, each commit has its own signature which is verified before writing the data to memory.
To mitigate ransomware attacks, all data in our Resources Server (RS) is periodically backed up. In case of an attack, the backed up data can be recovered. The machines use a Btrfs filesystem, which has built-in filesystem level snapshot support. A script running on the RS takes a read-only snapshot of the data that is intended to be stored periodically using Cron jobs. The snapshot is compressed into a zip file which is transferred to the Backup Server (BS) via a SCP connection.
To ensure that the data stored in the BS is always valid (that is, none of the files have been modified unauthorizedly, such as by encryption through a ransomware attack), a script to check the integrity of the files in the RS is scheduled to run immediately before creating the snapshot. This compares the hash of each file to the corresponding hash stored in the database.

2.2. Deployment

We have 7 distinct machines:

1. Machine for the user;
2. Machine running Nginx and working as the main Firewall;
3. Machine running Auth REST API;
4. Machine running Resources REST API;
5. Machine running a Redis instance used for storing users, hashed passwords, salts, and refresh tokens - AS;
6. Machine running a MongoDB instance for storing users' public keys, managing projects metadata, and where project files are stored - RS.
7. Machine that stores the backups from the RS.

Each machine has a firewall configured using iptables to allow only the required connections.

2.3. Secure Channels Configured

The communication entities and their connections are as follows:

- The "users" establish an HTTP connection using TLS through port 8443 with the Reverse Proxy machine, from the outside, to have access to the Services.

- The AA receives HTTP requests through port 8443 and accesses the Redis instance at port 6379 running in the AS.
- The RA receives HTTP requests through port 8444 and accesses the MongoDB instance running on the RS at port 27017. It also connects to the AA machine to validate access tokens using HTTP at port 8445. It connects to the RS to store and retrieve project files using SFTP through port 22.
- The BS connects to the RS to backup the current state of the project files using SCP through port 22. The BS's ssh public key is shared manually with the RS.

A self-signed TLS certificate was created and placed manually on the Firewall, AA, and RA machines. The library/tool used for generating and signing the TLS certificate was OpenSSH.
To make the connection between the AA and the AS more secure, it can be tunneled through an SSH tunnel, however, this was not implemented. The same applies to the RA and the RS.

2.4. Secure Protocols Developed

The client communicates with the AA for user authentication. The authentication follows a protocol similar to OAuth 2.0 where each user gets a refresh token which can be used to retrieve access tokens. These tokens are generated using JSON Web Tokens, which allow to encode all relevant parts of an access token into the access token itself instead of having to store them in a database. To access the AA, the user must register using username and password. The password is enhanced on the local client using a KDF. The AA stores the hashed version of the password with a salt on the Redis instance. Using a KDF and a salt helps mitigate rainbow table attacks. Every request to the RA should be authenticated using an access token, which the RA verifies with the AA.
Each user has a pair of public/private keys generated by the local client during the first login. It is assumed that if the user logs in to a new machine, the private key is transported by him. The users' public keys are stored on the RS. Every time a user creates a project, a symmetric key is generated on his machine and encrypted using his public key. The encrypted symmetric key is sent to the RS for storage, so the user can access it on a different machine. This key is used to encrypt all the project files. If the user wants to share the project with a different user, the client encrypts the project symmetric key using the other user's public key. The user's private key is used for signing every commit he sends to the server.
The BS will use an asymmetric key pair for establishing an SCP connection with the RS.

3. Used Technologies

- Dart for the REST APIs using the following libraries: chunked_stream, crypto, dart_jsonwebtoken, dartssh2, encrypt, envify, http, http_parser, mime, mongo_dart, redis_dart, shelf, shelf_router, and uuid;
- Docker for deploying each application and instance;
- HTTP with TLS for making the requests from the client to the remote server;
- Iptables for setting up firewalls;
- Java for the bag client, using the following libraries: com.google.gson, javax.crypto, java.io, java.net, java.nio.file, java.time, java.util, org.apache.commons.compress, java.security.
- MongoDB for storing public keys, projects' information, and versions;
- Nginx for reverse proxy;
- Python for the snapshot and backup script, using the following libraries: subprocess.

- Redis for storing users, hashed passwords, salts, and refresh tokens;
- SCP for backing up files from the RS to the BS.
- SFTP to write and read files to and from the RS.

4. Results

R1. Require user authentication

Satisfied. A protocol similar to OAuth 2.0 was implemented. The implemented protocol ensures that only users that know the username and password combination can obtain a refresh token and, therefore, an access token. The access token has a validity of 5 minutes, so they become useless after that time, even if they fall into the hands of an attacker. The refresh token must be kept secret and used only to refresh access tokens or logout. In both actions, the used refresh token is discarded. Once a new access token is obtained, a new refresh token is issued, maintaining the original expiry date. The tokens use JSON Web Tokens, which contain information regarding the subject and the issuer. Each token is signed using a secret key, making the AA the sole issuer of these tokens. Every interaction with the RA requires a valid access token. Since the access token contains the subject it was issued to, the access tokens cannot be shared between different users.

R2. Synchronize user documents between the personal device and the remote server.

Satisfied. The user documents are pushed to the remote server and stored on the RS. A copy of each commit is saved. The user can then pull the latest version to his personal device or checkout a specific version.

R3. Allow document sharing between users - synchronize documents across multiple personal devices and the remote server.

Satisfied. Each project has its secret key which is used for encrypting project's files. This secret is encrypted using every collaborator's public key and then stored on the MongoDB instance on the remote server. This allows sharing of documents between collaborators of a project ensuring confidentiality.

R4. Selection of documents to synchronize with the server.

Partially satisfied. The user can create a project inside a directory on his local machine. Once a project is created, he can choose when to push the files inside that directory to the remote server. However, it is not possible to select which files to commit, as it is possible with Git.

R5. Ensure a secure channel between personal devices and remote servers.

Satisfied. The interactions between the client and the remote server use TLS. A self-signed certificate was generated and is used to protect the information exchanged between the user and the remote server.

R6. Documents only accessible by authorized parties.

Satisfied. The project files are encrypted using AES-256 before pushing to the remote server, therefore, assuring confidentiality. Since the project key was encrypted with each collaborators' public key, only them can decrypt it and use it to access the project's

documents. Since the utilization of certificates is not implemented, an attacker could tamper with the public keys stored on the remote server, making the sharing of a project key vulnerable. This could be fixed by having a Public Key Infrastructure that would generate certificates for users' public keys.

R7. Register modifications of documents (especially by unauthorized parties).

Satisfied. The version of each commit corresponds to the SHA-256 hash of all the commit project files, making possible the verification of each commit's integrity. When making a new commit, the author hashes the encrypted file twice using SHA-256 and then signs it using his private key, S(H(H(E))). Every commit has a corresponding signature, ensuring the integrity of the files pushed to the server (the server can hash the received file and validate the signature), as well as non-repudiation. Besides, the author also signs the single hash of the encrypted file and encrypts it using the project secret key K(S(H(E))). This guarantees that only the collaborators of the project can obtain and produce that signature. This is used to detect and discard modifications by unauthorized parties, when pulling files, the user decrypts K(S(H(E))) and validates the signature S(H(E)).

If the MongoDB instance was on a different machine from the machine storing the project files, an effective way to monitor unauthorized modifications would be to calculate the SHA-256 hash of each project's commit and periodically validate with the signature stored on the MongoDB instance. If something did not match, it would register the problem in a log file and proper measures could be taken. The script "check_files.py" is a starting point to this approach.

R8. Periodically backup the documents in a different machine, safe from attackers.

The RS uses a Btrfs filesystem, which has built-in filesystem level snapshot support. Taking a snapshot of a subvolume will save the state of the files and/or directories in that subvolume. The Btrfs filesystem supports read-only snapshots. Taking a read-only snapshot, protects that snapshot's files/directories from being modified later. The backup process is automated using Cron jobs to schedule scripts for each server. For the RS, the script used to check the presence of any tampering with the files/directories is scheduled first. The script "*snapshot-script.py*" takes a read-only snapshot of the data that is intended to be stored and compresses into a zip file. On the Backup-server, the script "*backup-script.py*" is scheduled to run minutes after the "*snapshot-script.py*" on the other server. Via a *Secure Copy Protocol* (SCP) connection, the script copies the zip file with the snapshots of the machine Resource-server. SCP uses *Secure Shell* (ssh) for data transfer and uses the same mechanism for authentication, providing authenticity and confidentiality for the data transferred. At the moment, the project only backs up the data from the RS, the processedure can be replicated to other machines, such as the AS.

R9. Perform version control on the server.

Satisfied. As stated, every commit is registered with an unique hash and the files for each commit are stored on the RS. Allowing the user to checkout older versions of the project.

R10. Intrusion Detection Systems.

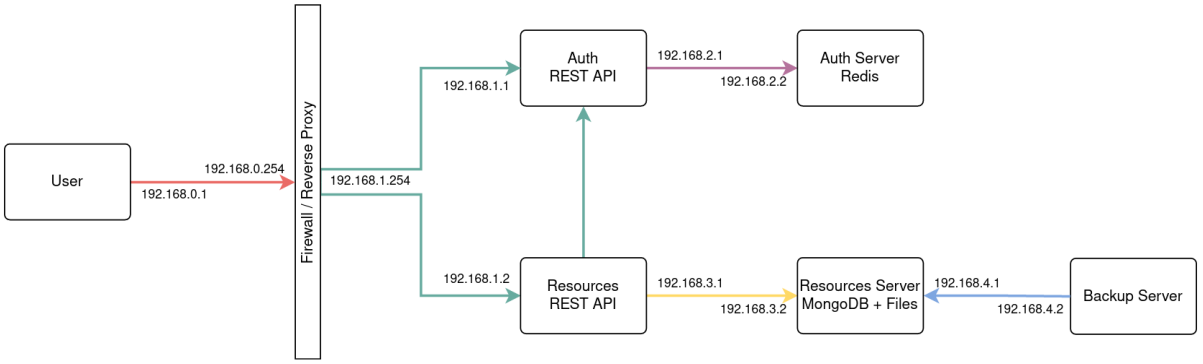
Not satisfied.

R11. Penetration testing the network.

Not satisfied.

Annexes

Machines diagram



The different colors on the diagram represent different networks. The red network constitutes the public internet, while the others illustrate internal networks.
The direction of the arrows also indicates which machine starts a connection with which.