UNIVERSITÉ LIBRE DE BRUXELLES

Computer Project Report

INFO-Y115 - Secure software design and web security

GROUP MEMBERS

Samir AZMAR
Hugo LEFEBVRE
Marcus CHRÉTIEN
Cyril HENNEN
Elric DEROEUX
Quentin LÉVÊQUE

January 19, 2025

Contents

1 Security Features **2**
- 1.1 Key derivation and Master Key . . . 2
- 1.2 Normal User Register/Sign-in Scheme . . . 2
- 1.3 Trusted User Register/Sign-in Scheme . . . 2
  - 1.3.1 Register of trusted users . . . 2
  - 1.3.2 Signing CSR . . . 3
  - 1.3.3 Login of trusted users . . . 3
- 1.4 End-to-End encryption . . . 4
- 1.5 Secure connection . . . 4
  - 1.5.1 .app top-level domain . . . 5
- 1.6 Email system . . . 5
- 1.7 NextAuth . . . 6
  - 1.7.1 NextAuth and Passkeys Integration . . . 6
  - 1.7.2 Database Schema . . . 6
- 1.8 Robust log system . . . 6

2 Checklist **8**

1 Security Features

1.1 Key derivation and Master Key

To enhance security, we use a derived key from a master password to encrypt various cryptographic keys in the application. The process for generating the master key follows the guidelines outlined in the Bitwarden white paper [1]. The master key is used for different purposes, including encrypting the AES key used for securing video data and encrypting the private keys of trusted users.

We derive the key using PBKDF2 with SHA-256 and 600,000 iterations and use the user's email as the salt. For regular users, we store a hash of this key in the database. The hash is first computed client-side (before sending to the server) using PBKDF2, and then rehashed on the server using Argon2. This double-hashing process ensures that the database stores a highly secure representation of the key. When a user accesses the app, this approach allows us to verify the validity of the key and ensures videos are encrypted with the correct key, preventing potential errors.

1.2 Normal User Register/Sign-in Scheme

Our system uses WebAuthn (Web Authentication) provided by NextAuth (see Subsection 1.7).
The private key is stored on a security device mainly the device itself but other options exist such as the use of your mobile authentication app or security key and never handled through our application. Instead, it is used through the OS's secure interface (Windows Hello, Touch/Face ID, etc.).

1.3 Trusted User Register/Sign-in Scheme

Our system uses certificate-based authentication for both trusted user registration and login to ensure robust identity verification. This is complemented by email verification and two-factor authentication (2FA) via a mobile authenticator app, adding an extra layer of security.

1.3.1 Register of trusted users

Trusted user has to provide their information and a password. On the client side (browser), we will generate 2 asymmetric key pairs:

- **Authentication Key Pair**: For verifying the user's identity.
- **Encryption Key Pair**: For securing sensitive data

We generate a master key derived from a master password (explained in section 1.1). Both private keys are encrypted with the master key. Then we send the encrypted private keys as well as the public keys and the email address to the server. The server saves them in the database.

When the client made the request to the server, it will create a `CSR` with the authentication public key and the user information. The user will have to download a `CSR` file, and submit it to their CA (more info in section 1.3.2).

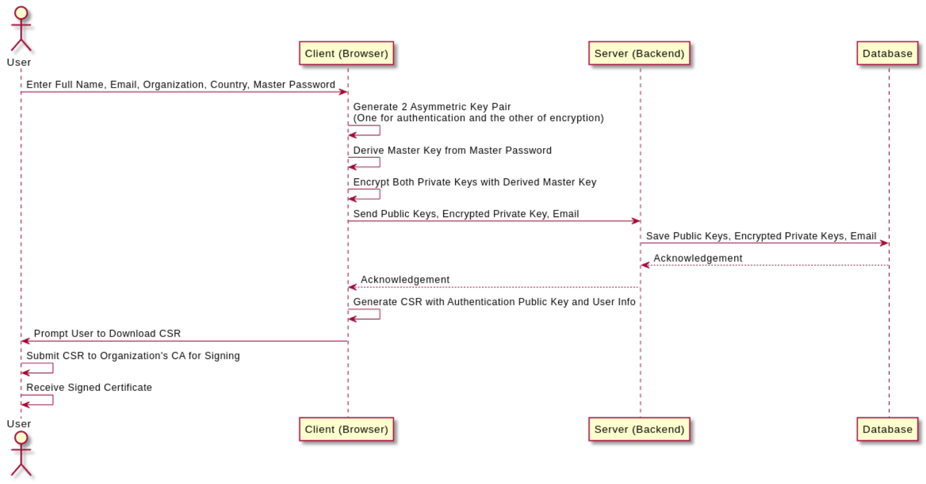
User
Client (Browser)
Server (Backend)
Database
Enter Full Name, Email, Organization, Country, Master Password
Generate 2 Asymmetric Key Pair
(One for authentication and the other of encryption)
Derive Master Key from Master Password
Encrypt Both Private Keys with Derived Master Key
Send Public Keys, Encrypted Private Key, Email
Save Public Keys, Encrypted Private Keys, Email
Acknowledgement
Acknowledgement
Generate CSR with Authentication Public Key and User Info
Prompt User to Download CSR
Submit CSR to Organization's CA for Signing
Receive Signed Certificate
User
Client (Browser)
Server (Backend)
Database

Figure 1: Sequence diagram: trusted user register

1.3.2 Signing CSR

To facilitate the trusted user registration process for the project, we introduced a dev interface at `<url>/ca`. This interface mimics the behaviour of a Certificate Authority (CA) for testing and demonstration purposes only. It is important to emphasize that this interface is not part of the final application and is strictly for development. In the real world, users would submit their Certificate Signing Request (CSR) to their organization's CA, which would verify the user and their details before deciding to sign the CSR. The dev interface provides four choices: three simulate certificates signed by trusted CAs, and one represents a malicious or self-signed certificate, allowing us to test how the system handles valid and invalid certificates. While this interface simplifies testing, it bypasses critical verification steps and must not be included in production, where legitimate CAs perform rigorous checks to ensure certificate trustworthiness.

1.3.3 Login of trusted users

When a trusted user logs in, the process involves two factors for authentication: their signed certificate (something they have) and their master password (something they know). The login process is carried out in two phases:

1. The user sends their signed certificate to the server. The server verifies the certificate's validity using the CA root certificate it possesses and checks if the user has an existing account. Once validated, the server generates a unique challenge and returns it to the client along with the encrypted private key.

2. The client decrypts the encrypted private key using the master key, which is derived from the user's master password. The decrypted private key is then used to sign the challenge. Finally, the client sends the signed challenge along with the certificate back to the server for verification.

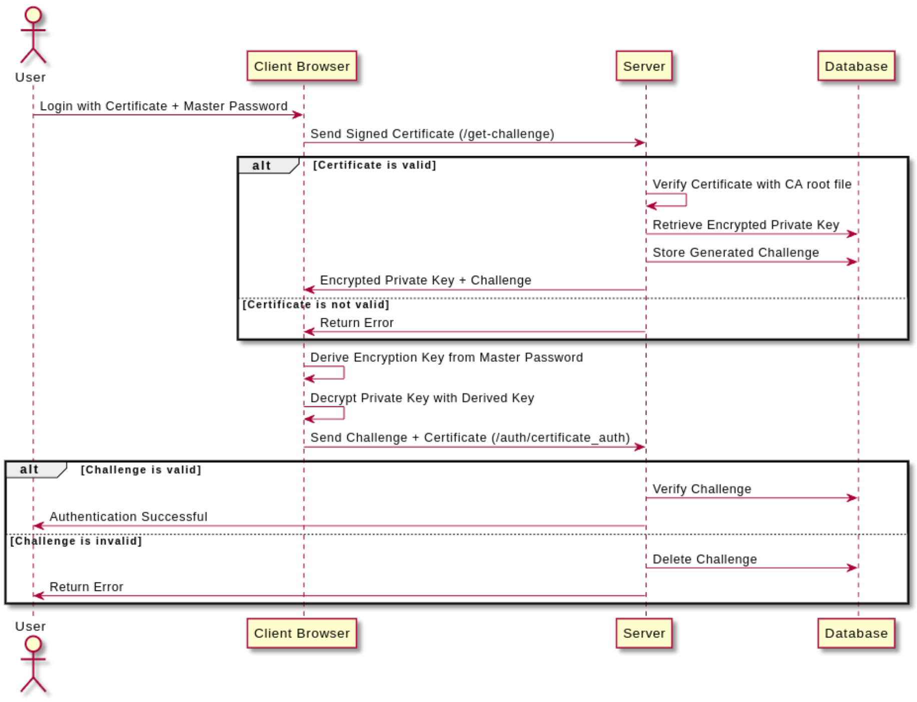
User
Client Browser
Server
Database
Login with Certificate + Master Password
Send Signed Certificate (/get-challenge)
alt [Certificate is valid]
Verify Certificate with CA root file
Retrieve Encrypted Private Key
Store Generated Challenge
Encrypted Private Key + Challenge
[Certificate is not valid]
Return Error
Derive Encryption Key from Master Password
Decrypt Private Key with Derived Key
Send Challenge + Certificate (/auth/certificate_auth)
alt [Challenge is valid]
Verify Challenge
Authentication Successful
[Challenge is invalid]
Delete Challenge
Return Error

Figure 2: Sequence diagram: trusted user login

1.4 End-to-End encryption

The app ensures end-to-end encryption by making it impossible for the server to decrypt videos stored on it; only the intended users have access to their data.
For each video a user records, the client generates a unique AES key specific to that video. This AES key is encrypted with the user's master key and sent to the server for storage. The video itself is uploaded to the server in encrypted chunks. If a user wishes to share a video with a trusted user, they must retrieve the encrypted AES key for the video, decrypt it locally using their master key, and re-encrypt it using the trusted user's public key before sending this newly encrypted key to the server. When the trusted user logs in, they can access the video, as they are the only ones who can decrypt it using their private key.
This approach ensures that all data remains confidential and accessible only to the intended recipients.

1.5 Secure connection

The app is deployed on a production-grade Fedora machine. The HTTP server used on the production machine is **nginx**. The **nginx** server is used to reverse proxy requests to each client app. This effectively means that no external device can communicate **directly** with our NextJS app, instead **it must pass through nginx** which handles the secure connection setup detailed in this section. Nginx also comes with some interesting features: rate limiting, ip black/whitelisting (for our two services, mail-server and log-server) and even more local logging. For instance, if someone tries to access the mailer server, either through an HTTP GET at '/' or the APIs correctly, from an external source, the request will be blocked and nginx will automatically log as follows in `/var/log/nginx/error.log`

```
$ 2025/01/19 11:17:32 [error] 4185492#4185492: *578 access forbidden by rule,
client: [IPV4_ADDR], server: mailer.insane.app, request: "GET / HTTP/2.0",
host: "mailer.insane.app"
```

For HTTPS, we use Let's Encrypt to obtain our signed certificates. This is crucial to ensure that our domains are recognized by the client. In the case we used self-signed certificates clients would not automatically trust the connection, leading to warnings about the site being insecure, the reason being that the clients **need to verify that the keys used for encryption truly belong to the server they are trying to access**. **Let's Encrypt**, a trusted Certificate Authority, verifies and authenticates our certificates when clients connect. In our setup, we generated 3 individual certificates for each service (mailer.insane.app, logs.insane.app and insane.app). To generate the certificates we use certbot which handles renewal and makes it easy to handle. The certificates enable secure TLS connections, ensuring that data between the client and server remains encrypted and protected against interception or tampering. The certificates are secure through tightly managed permission access policies set up by default (DAC) and SELinux enabled and enforced as well (MAC).

1.5.1 .app top-level domain

There's also a particularity here, and the use of a .app domain is not randomly chosen: Google Registry (who owns the .app TLD) has enforced mandatory HTTPS on its issued .app domains ever since they were introduced in 2018. This means that it is effectively impossible to use HTTP on a `.app` domain, even if the HTTP server allows it. This enforcement is achieved through the preloading of **HSTS (HTTP Strict Transport Security)** on all `.app` domains by default, ensuring that browsers automatically upgrade any HTTP requests to HTTPS and reject non-secure connections outright. More information is provided on Google Registry.

1.6 Email system

Since our whole authentication is based on FIDO by using a secure device to log users into our systems, we use Webauthn. And in Webauthn authentication it's common to define a username/email to pair with the secure device. Thus, we implemented an email verification mechanism, to verify the emails used to log into the dashcam software. For our mail verification, we used an external mail server. It was crucial to set up a separate mail-server (using our own HTTP REST API for handling emails) instead of having the mail credentials directly passed into our main NextJS app. It allows us to fine-grain access to the mail sending functionality (in case a NextJS instance is compromised, we can prevent it from sending new emails, for instance, spam mail from the official no-reply). This is done through a very simplistic api key system on the mail-server: the mail server generates its own private api keys (strings). These keys are used in the main app to send emails. The mail server has its own local logging mechanisms, separated from the log-server, it logs every mail sent and ensures no misuse is possible (it's important to repeat here the mail server sits behind nginx and is only accessible to a specific set of whitelisted IP addresses). The emails are signed **using DKIM signatures** to ensure authenticity and prevent email spoofing. So we have **the DKIM public key stored as a DNS record**, and the **mail-server has the private key** (dkim.pem). The verification process itself follows a standard flow: when a user registers with their email, a unique verification token is generated and stored securely. This token is then sent via email to the user's address through our separate mail server. To complete the verification, the user must click the link containing this token, which is then validated against our stored value. Only after successful verification can the user proceed with the setup of their master password. All verification tokens have a limited lifetime.

1.7 NextAuth

To handle authentication in our NextJS app, we use NextAuth.js (sponsored by Vercel and Auth0). This library provides support for a plethora of authentication schemes; however, in the context of this course, we emphasized using FIDO authentication systems. Specifically, we settled on WebAuthn, a modern, passwordless authentication standard that uses asymmetric cryptography. During registration, a unique public key is generated: the private key is securely stored on the user's device, and never accessed directly, while the public key is saved on the server. Authentication is performed by the user proving possession of the private key, often secured behind device biometrics or hardware tokens (Windows Hello, ...), ensuring both security and user convenience (across multiple devices). The support of WebAuthn in NextAuth is through something called "passkeys".

1.7.1 NextAuth and Passkeys Integration

To use this feature, the `Passkey` provider must be added to the configuration. A compatible database adapter, such as Prisma, is required alongside schema updates to include the `Authenticator` model.

1.7.2 Database Schema

The `Authenticator` model stores WebAuthn credential data and is linked to the `User` model. Key fields in the schema include:

- `credentialID`: Stores the WebAuthn credential ID as a unique string.
- `credentialPublicKey`: Holds the public key for authentication.
- `counter`: Tracks the authentication counter for replay protection.
- `credentialDeviceType`: Specifies the type of device used.
- `credentialBackedUp`: Indicates whether the credential is backed up.
- `transports`: Optionally lists supported transport methods (e.g., USB, NFC).

After updating the database schema with the Authenticator model provided by the AuthJS library, Passkeys can be integrated into the code directly.
This setup enables passwordless login as expected in the context of this course. The implementation of AuthJS with passkeys in our project strictly follows the official recommendations and code samples provided by the GUI library used MUI Toolpad passkeys.

1.8 Robust log system

We have implemented a log system that logs user activity and different sorts of errors. The backend records every request that it receives. The log system has a blockchain-like structure that works as the following:

Main server side:

1. The main server computes a HMAC based on ***SequenceNumber*** + ***log(level, message, additional data, timestamp)*** + ***secret***.
2. Then it sends the ***SequenceNumber*** + ***log(level, message, additional data, timestamp)*** + ***HMAC*** to the log server.

Log server side:

1. The log server verifies the authenticity and integrity of what the main server sent using the HMAC of the main server.
2. It verifies the correct sequence number and the HMAC of the current block.
3. If the sequence number and previous log are correct then it computes the blockHMAC based on ***SequenceNumber + log + mainServerHMAC + previousLogServerHMAC + timestamp + secret(only known by the log server)***
4. Once the log server computes the blockHMAC then it saves in the database **SequenceNumber + log + mainServerHMAC + previousBlockHMAC + blockHMAC + timestamp(date of when the block was added)**

2 Checklist

1. Do I properly ensure confidentiality?

    - Are sensitive data transmitted and stored properly?

      Yes, sensitive data are encrypted on the client side using the master key only known to the user. Thus, when the client transmits sensitive data, they are encrypted and stored encrypted.

    - Are sensitive requests sent to the server transmitted securely?

      Yes, our production version uses TLS v1.3 (X25519 and AES_128_GCM) with X.509 certificates signed by Let's Encrypt. It ensures the connection is ciphered and that the cryptographic material is authentic (Let's encrypt). The client starts the DNS resolution of insane.app then establishes a TCP connection. Then comes the TLS handshake where the client lists its available ciphers and the server selects the appropriate one. Key exchange occurs using Ephemeral Diffie-Hellman (ECDHE) and both parties derive session keys securely. After the TLS handshake, the client confirms the server's certificate is valid and issued by a trusted Certificate Authority.

    - Does a system administrator have the ability to access any sensitive data?

      A system administrator does not have the ability to access any sensitive data related to the app (video recordings, private keys and secret cryptographic materials, etc.). Sensitive data are encrypted using the master key known only to the user. This key is not stored anywhere. However, administrators have access to logs (which contain IP addresses and email addresses). We divided our solution into three distinct services, it is assumed these services are distributed among different machines and operated by different providers with different administrators limiting the potential issues arising from a malicious system administrator.

2. Did I harden my authentication scheme?

    - Do I use Captcha, MFA, a zero-knowledge proof scheme?

      We use Google reCaptcha V3. For MFA, we use the passkey for normal users and after a 6 digits is asked. A trusted user needs to provide a signed certificate and password after a 6-digit code is asked.

3. Do I properly ensure the integrity of stored data?

    No, we don't. However, sensitive data are encrypted using AES-GCM which can provide integrity of the encrypted data.

4. Do I properly ensure the integrity of sequences of items?

We use an SQL database to store our main application's data. This ensures integrity using specific columns (autoincremented IDs, dates of expiry, timestamps, etc.). The SQL database handles the integrity of the relationships between the different models (meaning some actions are inherently impossible SQL schema-wise).

- Does somebody have the ability to add or delete an item in a sequence or edit an item in a sequence, without being detected?

  As it requires to go through the backend to add, remove, or edit an item. This action is recorded and saved in the log database. Thus, we are aware of such an action.

5. Do I properly ensure non-repudiation?

   Yes. Since we have a robust log system in place that logs everything happening in the backend and saves it in a blockchain-like structure. Every service present in the solution logs every action performed.

6. Do my security features rely on secrecy, beyond cryptographic keys and access codes?

   No, we do not rely on secrecy except for cryptographic keys.

7. Am I vulnerable to injection?
   - URL Injection: URL parameters are handled securely by Next.js, and your blacklist with validator.js adds extra protection.
   - SQL Injection: Prisma uses Prepared statements which allows us to prevent SQL injection with parametrized queries, making your app safe from this threat.
   - For our frontend React escapes values to prevent XSS. On top of that, we use custom grammars to validate entered emails and passwords, and a blacklist with validator.js to sanitize all other values entered both in the frontend and through the backend endpoints.
8. Am I vulnerable to data remanence attacks?

   No, since sensitive data are stored encrypted thus recovering deleted sensitive data will lead to gibberish data.

9. Am I vulnerable to fraudulent request forgery?

   CSRF protection is enabled for routes controlled by AuthJS. All cookies and sessions are marked as HttpOnly, meaning they cannot be accessed via client-side scripts. The basic CSRF protection in use prevents cookies from being sent with cross-site requests initiated by links or iframes.

10. Am I monitoring enough user activity so that I can detect malicious intents, or analyse an attack a posteriori?

- Am I properly sanitising user input?

  We have implemented basic user sanitization for the frontend and backend.

- Did I implement some form of anomaly detection?

  We didn't implement some form of anomaly detection as it requires a large set of user data which we don't have.

- Do I use a whistleblower client?

  No, we don't.

11. Am I using components with known vulnerabilities?

    As far as we know the components used are not vulnerable.

12. Is my system updated?

    Yes, we have updated our system and used the latest version of the various languages and frameworks used.

13. Is my access control broken (cf. OWASP 10)?

    We based our access on RBAC (Role-Based Access Control). Each role (Normal User or Trusted User) is allowed to access only what they are allowed to access. For example, Trusted users aren't allowed to record, but Normal Users can.

14. Is my authentication broken (cf. OWASP 10)?

    No, it's not because:

    - We have implemented Google reCaptchaV3 to block automated behaviour.
    - We forbid weak passwords or well-known passwords.
    - 2FA is enabled for every account. (It's forced.)
    - The password is hashed on the server side using argon2.
    - A new session token is generated after each successful new login.

15. Are my general security features misconfigured (cf. OWASP 10)?

    For production, we have the following security measures in place:

- **firewalld**: All ports on production are blocked by default except HTTP 80 and HTTPS 443. We use a secure ssh port (not 22, and ssh access is secured through ECDSA keypairs) and the **firewalld** (standard on **fedora**) in default configuration: which blocks everything unless allowed.
- **Strict DAC and MAC**: our production server uses strict Discretionary Access Control (DAC): not chmoding and chowning resources that aren't supposed to be. It also runs **SELinux** as Mandatory Access Control (MAC) in enforcing mode. The `root` user is "disabled" as well.
- Errors are handled internally (logs) and visually through nginx providing our custom HTML error pages. This ensures minimal information is provided upon errors.
- **Rate-limiting**: rate-limiting is handled both at an **nginx** level and at a Google reCaptcha level. The rate limiting on **nginx** is set in place using a directive in the configuration file related to the domain name.
- We have a clear distinction between **production** mode and **development** mode
  - The production environment variables are only present on the production server. These environment variables are unique to the production environment and are not to be shared across different setups. Furthermore, these environment variables are not tracked by the source control. It makes sense but it's fairly common to stumble upon projects that forget that.
  - The production server generates cryptographic material on its own and runs appropriate setup commands that are not meant to be run in development mode.

References

[1] Bitwarden. *Bitwarden Security White Paper.* https://bitwarden.com/help/bitwarden-security-white-paper/, Accessed January 2025.