VRIJE
UNIVERSITEIT
BRUSSEL

SECURITY IN COMPUTING

Chat platform with end-to-end encryption

Bram Silue, Lars Willems

May 31, 2022

Science and Bio-Engineering Sciences

Contents

Abstract

This document is a report on the development of project for the course "Security in Computing" taught at the Vrije Universiteit Brussel.

1 Introduction

The purpose of this project is to create a secure chat platform similar to Slack, which allows users to communicate messages in channels or as private messages in a secure way. We were provided a simple web-based Slack clone that contains the main functionality from Slack, although without any regard to security, both at client-side and server-side. Our work aims to protect the application from common exploits and to limit what data can be read by different parties (such as other users, or the server itself).

More concretely, some of the goals in the security improvement are to prevent XSS attacks, CSRF attacks, clickjacking attacks, SQL injections, and sniffing. Additionally, we want to provide end-to-end encryption to make sure nobody other than the sender and the receiver can access the message data.

Our implementation provides two versions of the chat app. One version provides a smooth user experience but without persistent storage of data, and can be found in the folder called *chat-app (no-db)*. The other version does provide persistent storage of data using MongoDB, but comes with bugs on the client-side that make for some issues with regards to displaying messages and adding or removing channels/users in some scenarios. These bugs make for a severe lack of usability, though this version demonstrates how we would implement secure and hashed persistent storage that provides protection against SQL injections.

1.1 How to run

For instructions on how to run the chat application, manually head to the README.md file or simply read it from our Github repository, which is public and can be found by clicking here.

1.2 Used packages

Here is a list of all of our extra node modules with their version:

- bcrypt: 5.0.1
- cookie-parser: 1.4.6
- crypto: 1.0.1
- crypto-js: 4.1.1"
- ejs: 3.1.8
- express: 4.17.1
- helmet: 5.1.0
- jsonwebtoken: 8.5.1
- mongodb: 4.6.0
- mongoose: 6.3.4
- socket.io: 4.5.1

Socket.io and express were both already used in the original implementation. We used crypto and crypto-js for encryption and hashing on both client and server-side. We utilized the ejs module to use ejs files instead of HTML files. The cookie-parser module gets used on the client-side to

validate the JWT token, hence the use jsonwebtoken on both client and server-side. The mongodb and mongoose modules are used as database modules.

2 Login / Authentication

The login using aa prompt needed a re-do. New webpages were implemented on which it is possible to both register and login. We have used inspiration from the internet when it comes to implementing the whole login-register system from this web page (source).

When attempting to register, an error will occur whenever the username is less than 1 character or more than 30 characters long, or, whenever the password is less than 8 characters long. This is checked on both client and server-side. To implement this, we have used REST requests using JSON. Whenever a user registers for the first time, their password is put in our database (MongoDB) after being hashed using the pbkdf2Sync function. The function makes use of a random salt, such that even if an attacker were to find one password, they cannot find the other passwords. This salt is stored in the database together with the password and the username. Usernames are not hashed. After a register/login attempt, the user will only be redirected to the main page whenever the server sends a 201 response. Of course, this web page needs to know your username. First, we tried implementing a solution in which we would give the username as parameter, but then anybody could simply replace this username with another username and would be able to see all of that person's messages. Hence, that is not a secure solution. We then came up with an alternative. We pass the username of the user as a cookie, together with the hashed version of it. When hashing this username, we use the same seed the one used for hashing the password. When the browser gets redirected to the main page, the first thing that happens is a comparison applied to the username-cookie to the hashed username-cookie. Our code hashed the password with the same seed and whenever it matches the hashed username-cookie, the user is able to see the main page. Otherwise the user sees an error on screen. Both the username-cookie and the hashed username-cookie are setup as 'sameSite=stric' and 'secure', and the hashed username-cookie is also httpOnly.

In addition, we implemented the JSON Web Token. This is an extra way of authenticating users. Whenever registering/logging-in, a token is sent back which the users need to use whenever making a request. The max age of this token has been set to 3 hours. This means that after 3 hours, the user will not be able to perform any requests anymore. Hence, the user will need to login again. This is an enforcement of re-authentication, which is an important security feature. The token is made using its sign function which has as input the user's id, name and a predefined secret. Whenever this token gets leaked, users still have no access as the username still needs to be the same as the hashed version in the cookies.

3 Persistence

We have used MongoDB as our database. Every time a new user registers, the user is automatically saved in the database. Next to this, we also save rooms, messages within that room and as well all members who are present in the rooms. Whenever a user logs in, all of the rooms will be shown if he is present in any. The messages will be loaded when the user clicks on any room but after a room is retrieved from the database, we have the bug that new messages do not appear on screen. You just need to refresh and the messages will appear.

4 End-to-end encryption

End-to-End Encryption is a method of secure communication that prevents anyone besides the sender and the intended receiver from accessing the data during transfer. More concretely, the sender sends an encrypted message that is decrypted only when it arrives at the receiver. Nobody

in the middle, not the chat provider nor other entities has the ability to decrypt the message.

4.1 Overview

The actions required to achieve this all happen on the client-side, while the server-side simply transmits the message through without any processing or modification. The following steps are performed:

1. The message is encrypted using *symmetric* encryption.
2. The key that was used for symmetric encryption is itself encrypted using *asymmetric* encryption. This asymmetric encryption of the key is performed with the public key of every member in the chat-room
3. Together with the name of the associated member, all encrypted keys are placed in an array. The encrypted message and the array of encrypted keys are placed together in one object.
4. The object is sent to the server which then broadcasts the object to all the members.
5. The members look for the encrypted key assigned to them, decrypt it with their personal private key and then use the decrypted key to decrypt the message.

Indeed, every single message is encrypted using its own key. This is in contrast to having one key per chatroom, which is less secure.

4.2 Implementation

The symmetric encryption is done with AES encryption in CTR mode. The choice for CTR was motivated by the fact CTR provides up to IND-CPA level security, unlike the more commonly used CBC mode. Additionally, CTR mode doesn't require any padding which makes it immune to padding oracle attacks. Both the symmetric key and the IV for each message are generated randomly.

To authenticate the encryption of each message, an HMAC is used. The HMAC makes use of a random salt of 16 bytes, and a passphrase, which is the result of a SHA256-hash of the symmetric key followed by a PBKDF2-hash. When a message is authenticated, a green dot appears next to the message. If it is not authenticated, the dot becomes red. So far, all of the encryption, decryption and hashing functions have been implemented using a library called crypto-js.

As mentioned before, the symmetric key itself must be encrypted asymmetrically. To achieve this, we used RSA encryption. The RSA algorithm used is RSA-OAEP with SHA-256, using a key size of 4096 bits. The encryption and decryption functions have again been implemented using crypto-js, which allows us to create *non-extractable* private keys. Indeed, with the "extractable" field set to "false", the private key becomes non-exportable and non-readable, making it extra secure. Using IndexedDB these keys are stored persistently, directly in the browser.

4.3 HTTPS

To make sure communication is somewhat secure with or without encryption, the application is served over a (self-signed) HTTPS configuration. Instead of configuring it ourselves, we opted to use mkcert, a specialized tool for creating locally trusted TLS certificates that are compliant with what browsers consider valid certificates. It stays updated to match requirements and best practices and is cross-platform. This ensures messages between client and server are encrypted in transit.

5 Client-side security

Running OWASP ZAP on the original implementation of the chat application revealed quite a few vulnerabilities that needed to be addressed. Some important aspects to incorporate are:

- Prevention of XSS-attacks (e.g. HTML/JS injection).
- Prevention of CSRF-attacks.
- Prevention of clickjacking attacks.
- Prevention of sniffing.

To prevent CSRF-attacks, Anti-CSRF tokens have been implemented using the double-submit-cookie pattern. Indeed, the login and register pages contain a hidden CSRF field in the submission form where the token is stored. On the server-side, this token is compared with the token stored in a particular cookie. Only matching tokens allow for a registration or a login.

To prevent XSS attacks, two measures have been taken. Firstly, all input and output variables are sanitized using the DOMPurify library. While this already provides a substantial amount of protection, a second layer of protection is implemented by defining a Content-Security-Policy. This is achieved by setting the express headers of the server such that only the specific dependencies used are whitelisted.

Preventing clickjacking and sniffing is also achieved by setting the express headers. Against sniffing, 'X-Content-Type-Options' is set to 'nosniff'. Against clickjacking, we opted to use frameguard from helmet-js. Finally, we also disabled the 'X-Powered-By' response header fields to prevent leaking server information.

5.1 A small note on CSP

A good configuration involves not using the 'unsafe-inline' option. Unfortunately, we were not able to make the chat application run without this option. This is because the chat application makes use of inline event handlers in the main.js file to add messages, channels, etc. Though the number of event handlers can be reduced, some of them cannot be removed.

This is where the next best solution would be to create hashes from the inline scripts or event handlers and include these hashes as whitelisted in the CSP. However, no matter how much we tried to implement this feature, it did not seem to work. Reluctantly, we resulted in using the 'unsafe-inline' option. Ideally, we would find a way to use the hashes such as to not use this option.

6 Dependency management

Dependency management was performed in two ways. Firstly, we made sure all dependencies used were at their latest version possible without breaking the code. Additionally, a check was performed online to make sure that for each dependency, there are currently no known vulnerabilities.

Secondly, every dependency on the client-side was set up with an integrity check using SHA384 or SHA512 hashing, to make sure they remain unchanged.

7 Extra requirements

7.1 Toggle functionality for E2E-Encryption.

A toggle functionality was implemented that allows the user to create channels that are encrypted or not, regardless of whether the channel is private or not. Though this option exists for channels, it is not for direct rooms, as direct messaging is always E2E encrypted.