26-05-2022

SECG4
Computer Project

Authors:

Smolinski Piotr (56212) & Delcroix Noé (55990) – D122

Introduction:

The computer project consists of an instant messaging system, made for SECG4 class. The main framework that has been used for security backend is Laravel and for the frontend React.JS. All the installation process is automated by a DockerFile (Docker installation needed) and details about running the project can be found in the readme file. The main point of this project was to implement security features to guarantee the repudiation and integrity of sent messages and especially to avoid any access for 3rd party users to the confidential data. The application only allows P2P user conversations (no group conversations), he has the possibility to add a friend, or remove him from his list. The details about the production can be found in the README.MD file.

Security:

Since the user is able to connect himself to the app and create a new account, there may be a potential risk, a vulnerability of injecting values that might crash, or return confidential data. Since we're using Laravel, some of these features are already built-in to the framework. Like handling cross site scripting, to avoid any potential scripts that may be used to steal some confidential data, these script injections are treated as plain text, so no html code is going to be executed on the website.

Also, Laravel manages the SQL injections, thanks to the prepared queries, which makes sure that only valid parameters that match the data type can be passed to prepared statements. Declared parameters are using the '?' symbol, to indicate that the given parameter (potential user input) is going to be verified before executing the query, then translated and executed.

On the other hand, the data that is stored in the SQLITE database, only the sensitive data is hashed and encrypted based on ECDH anonymous key agreement scheme [1], that consists on generating an shared key from an private key of client A and a public one from the client B, it allows to decrypt the messages for both of the clients, while only using the other person public key. Data is sent using HTTPS, secured via TLS, so since then every request is securely transmitted.

All non-confidential data is visible to every database administrator, because only messages and passwords are confidential information that should be protected, the rest of the information contained in the database can be modified by SQL queries if the administrator has permissions.

To guarantee non-repudiation, every message sent by the client is simultaneously signed using SHA, even though it is encrypted because it is one confidential piece of information that should remain so. On the other side, in order to confirm the identity, the other client must decrypt using the key as well as check the fingerprint of the signature, and compare it to the server's one.

Our chosen authentication pattern is based on tokens that are unique. Tokens are generated only in the case whenever the user's given credentials match with those that are stored, as the server validates them, the server responds with an authentication token which is also kept in the database. When the same user sends requests to access secured resources, the generated token needs to be sent to authorize the request. These tokens are persistent , since they are locally stored in the browser, it allows them to send requests without authenticating the credentials every single time. On the other hand, sort of the token replaces the credentials all over the web application. Besides, there can be only one single connection, every successful login is going to erase the previous token with a new one. The number of possible attempts is limited to 3 attempts, after these 3 attempts, the account is permanently blocked and can only be unblocked by the admin.

As for replay attacks, we have implemented a security that consists on adding an integer to the message sending request only, since this is the only inconvenience that can be disturbing in our case. This integer is passed to the server which checks in the database if the new integer -1 corresponds to the previous one, if it is the case we insert the message and we update the integer in the database. Any abuse of replay attack will not pass the verification condition at server level.

We're using token based authentication that is not persist in a cookie or header but have to be put as request parameter. So, we’re safe against CSRF attacks, because we need this token to construct the url or the post data. Thanks to the tokens, every session is unique, these tokens are securely stored and theoretically unique in the world (token are uuid), so it makes it impossible to brute-force the token in order to forge requests.

Given that, user’s manipulations are limited to sending messages, adding new friends, removing one, creating an account and registering. There’s no need to monitor and log every user manipulation, since most of them are stored in the database, also some sort of abusing certain features may end up blocking the user’s account.

Before running the web application in production, Docker takes care of installing the missing dependencies, generate the self generated TLS certificate and all the necessary frameworks to ensure that the system is up to date.

Moreover, there’s no possibility to access any page, except for signing/registering without the generated token on the server side. At every step the token is verified so there’s no possibility to bypass the security without the token, because every implemented API needs a token to verify the user and obtain its data, same goes for the broken authentication flaw, the implemented security features such as, anti-brute-force, storing the private key in the local storage, at no time token or packet contents are visible by MITM because all packets are send by HTTP over TLS.

Conclusion:

Finally, all in all, our implemented security features are most on the client side, as the client is likely to be the biggest threat to the security and integrity of the data. Each feature is designed to limit the client's ability to protect its security, as mentioned earlier, our goal was to limit the possibility of manipulation rather than monitor every movement and adapt appropriate measures, the most known and potential threats are handled.

Bibliography:

[1]: ECDH Key Exchange - Practical Cryptography for Developers (nakov.com)