E2E Chat
Design Documentation

Team us

Till Behrendt (018799298)

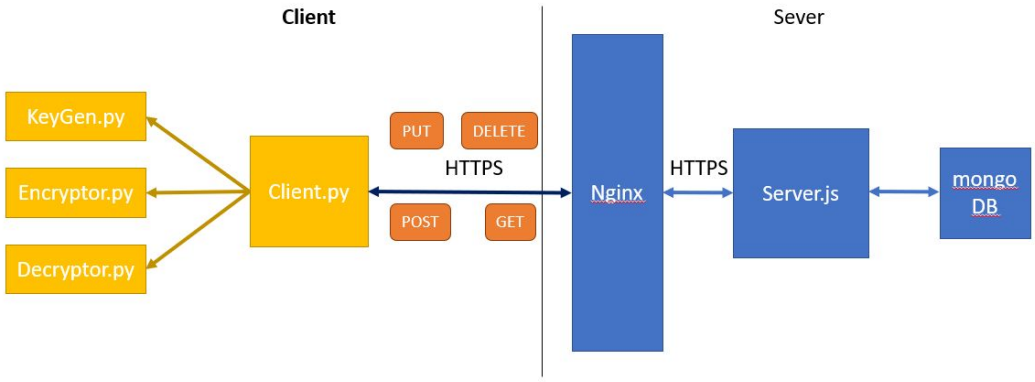
Jaime Park

This document provides an overview of the design of E2E Chat and an analysis of its security.

Design

Overview

E2E Chat consists of clients and a server. Users run client applications that connect with a server to register an account, authenticate, and send and receive messages.



The client consists of 4 python programs. The client.py is the main client program.
The server uses nginx as reverse proxy, node.js and mongoDB.
The communication between server and client relies on HTTPS requests.

Client

E2E Chat client is comprised of 4 python programs: `client.py`, `KeyGen.py`, `Encryptor.py`, and `Decryptor.py`. Users run `client.py first,` which in turn imports the other modules. The dependencies required by E2E Chat client are python libraries: `cryptography.hazmat`, `json`, `os`, `requests`, `sys`, `threading`, and `time`.

`client.py`

Graphical user interface consists of frames and buttons, while sending POST requests to the server when necessary.

`CreateNewUser(name, pw, pw2)`
When a user wants to make an account, this function asks the user to enter a `name` and a password (`pw`) , with `pw2` being the password repeated for user clarity. If `pw` and `pw2` match, then `client.py` makes a POST request to https://teamus.me/users/signup with `name` and `pw` as the payload. If the server replies with response code 200, then the account was successfully created, which can now be used to sign in.

`Login(name, pw)`
Send a POST request to https://teamus.me/users/signin with `name` and `pw` as the payload. If successful (response code 200), get their userID and JSON Web Token (JWT).

`GenerateKeyPair(filenamePrivK, filenamePK)`
Calls `KeyGen.GenerateKeyPair()` with `filenamePrivK` and `filenamePK` as names for the private and public key respectively.

`DeleteUser()`
Send a delete request to https://teamus.me/users/ passing the user's JWT for authentication.

`getMessages()`
Every second, send a POST request to https://teamus.me/messages/getmessage looking for messages from chat partner to user. JWT is passed along with the request. If there is a message, call `Decryptor.MyJSONDecrypt(data, myPrivk)` on it and insert the output into the chat box. `data` is the message received from the server and `myPrivk` is the user's private key.

`sendMessage(event)`
Get the contents of the `msgField` and call `Encryptor.MyJSONEncrypt(inputmsg, cpPK)` on it where `inputmsg` is the message to send and `cpPK` is the chat partner's public key. Then, send a post request to https://teamus.me/messages/sendmessage with the names of the user and who the message is destined to included in the payload. The user's JWT is passed along with the request.

`KeyGen.py`

Calls `cryptography.hazmat.primitives.asymmetric.rsa` to generate a private and public RSA key using e = 65537 and size of 2048. Keys are stored in the local directory and are named based on user input.

`Encryptor.py`

Encrypts and tags messages to be sent using AES, HMAC, and RSA.

`MyencryptMAC(message)`

1. Generate 2 random 32 bit numbers using os.urandom(32) to be used as encryption key (EncKey) and HMAC key (HMACKey). An additional 16 bit urandom number is also generated as the iv.
2. Get the message in bytes and pad it using PKCS7 as 128 bit blocks.
3. Encrypt the message using AES with CBC mode using `EncKey`.
4. Generate a tag of the ciphertext using `HMACKey` and SHA256.
5. Return the ciphertext, iv, tag, AES encryption key, and HMAC key.

`MyRSAEncrypt(message, RSA_Publickey_filePath)`
With RSA_Publickey_filePath as the chat partner's public key, encrypt the AES key and HMAC key with RSA using OAEP, MGF1, and SHA256. Return RSA ciphertext, AES ciphertext, iv, and tag.

`MyJSONEncrypt(message, RSA_publickey_filePath)`
Create a json object containing RSA ciphertext, AES ciphertext, iv, and tag. Replace special characters "&" and "+" and then return.

`Decryptor.py`

Decrypts and checks tags of incoming messages. The order is RSA -> HMAC verification -> AES.

`MyJSONDecrypt(message, private_key)`
Replace manually edited out special characters ("&", "+") and pass the contents of the json to `MyRSADecrypt`.

`MyRSADecrypt(RSACipher, ct, iv, RSA_Privatekey_filePath, tag)`
Decrypt RSA ciphertext with the user's private key and pass the decrypted AES and HMAC keys with the rest of the parameters to `MydecryptMAC`.

`MydecryptMAC(ct, iv, tag, EncKey, HMACKey)`
1. Check if the AES and HMAC keys are at least 32 bits long
2. Verify the tag using `HMACkey` and ciphertext
3. AES decrypt ciphertext using `EncKey`
4. PKCS7 unpadding of message
5. Return message

Server

All of the routes use HTTP status codes to inform the client whether the action was successful or failed. The status code 200 confirms that a request was successful. If the server responses with a 403 status code, something went wrong with the authentication of the user. A 500 status code means that an internal server error occurred. If an error happens, the client should send another request.

`/users/signup`
Using this route a new user can signup for the chat. They send a POST request which contains the information “name” and “password”.
The “name” must be unique in the database, and an error will be thrown if the username already exists. If not, the server will respond with a JSON object containing the “name”, “password” which is a salted hash and “id” of the user.
Before the user is saved to the database, `.pre('save')` in the model is called. This function converts the provided plaintext password into a salted password hash using bcrypt.

/users/signin
This route is used to signin for the chat. The client sends a POST request with “name” and “password” to this route.
The server checks whether the username exists in the database. If so, the model function `comparePassword()` is called, which compares the provided password with the password hash in the database and returns a boolean value if they are equal.
After confirmation, the server signs a JSON web token for the user. The token is signed with the username, the user id and a secret value. The secret value is a static string which is hardcoded in the server application. This token has a lifetime of 24 hours. This token is then sent back to the user together with the user id.
If an error occurs, the user is not issued a token and they should retry the signin.

/users/:id
This route can only be used with a DELETE requests and is used to delete users from the database. This route contains the unique database id from the user which the client can only obtains if they sign in.
The requests to this route must contain the user's JWT, which is issued after a successful signin.
Before the server deletes a user from the database, it will confirm that the JWT using `jwtCheck()` from routes. This function verifies the token by using the secret static value on the server with which the token was signed. If the token is valid, the user is deleted. If the token is invalid, the client receives a error message from the server. If the server successfully delete the user from the database, it will send a confirmation to the client.

/messages/getmessage
This route, with a POST request containing "to", "from" and a valid JWT, a user can check whether new messages are available.
First, verify the token using `jwtCheck()`. If the token is valid, `getMessage()` function in the controller is called. This function takes the provided sender and receiver information from the request and checks whether the database has any messages for the receiver by a specific sender. If there is a message, the server will send it to the client and delete the message afterwards from the database. If there is no message for the client, the server responses with a 204 status code and that there are no messages for this receiver.
If the authentication fails or an error is thrown, the server will respond to the client with an error message and the client has to retry.

/messages/sendmessage

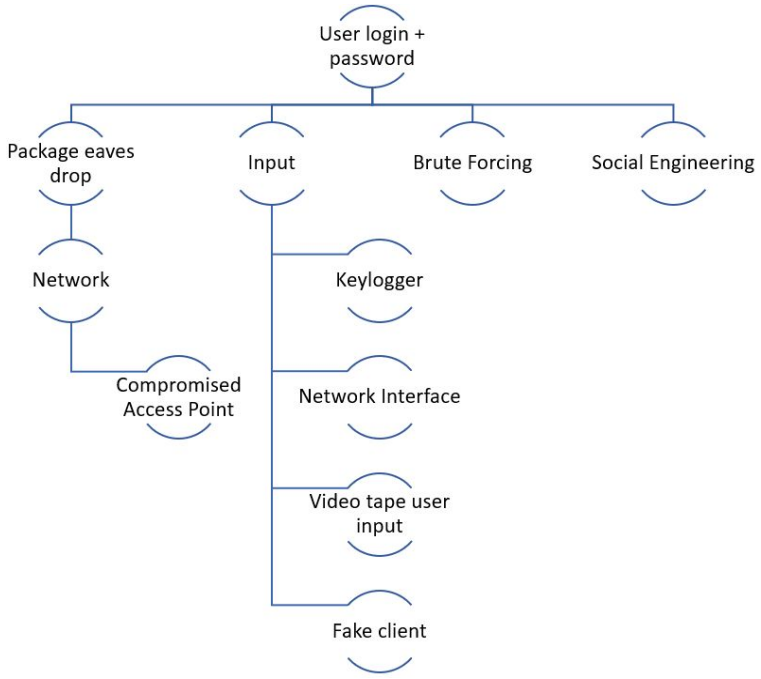
If the user wants to send a message to another user, the client will use this route with a POST request which contains information about “to”, “from”, “message” and a valid JWT.

If the server gets a request on this route, the token is verified using `jwtCheck().` If the verification is successful, `sendMessage()` in the controller is called. The server creates a new message with the provided information about sender, receiver and the message itself according to the message model. This new message is then saved in the database. At the end, the user will receive a confirmation that his message was successfully sent to the server. If an error occurs, the server will send an error message to the client and the client has to resend the message.

Security

Attack surfaces

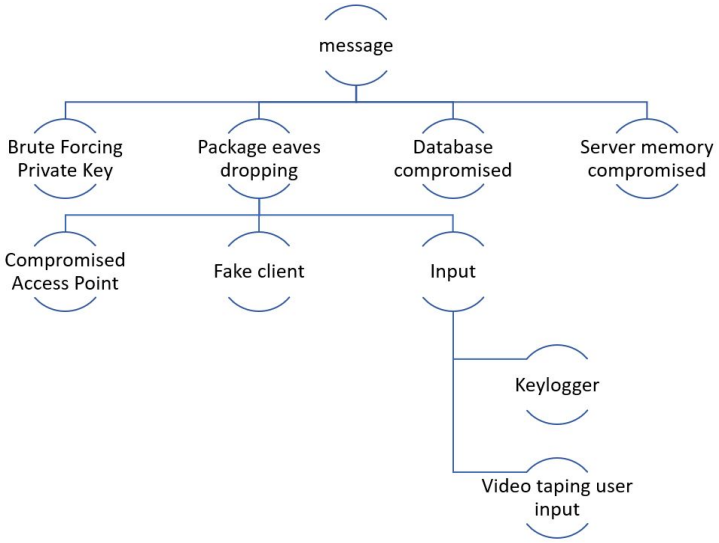
User credentials



To ensure that the user login and password are not compromised it is important, that the user is aware of possible attack surfaces. Therefore, they should ensure that they use the chat client only in a trusted network where he knows that the network itself is not compromised by any adversary. Furthermore, the user has to choose their password wisely so that a brute force attack becomes too expensive to perform. This means their password should be at least 20 characters and be a combination of lowercase letters, uppercase letters, numbers and special characters. To prevent social engineering, the user should never post their username or password online. User password are salted and hashed using bcrypt as they are stored in the server, preventing most common attacks against passwords such as rainbow tables.

Multiple possible attack surfaces involve the user input, such as keylogging. As it is important that user insures that he has the correct, unmodified client application and his computer is not compromised by an adversary in any way. E2E Chat can be obtained by building from source after checking that the repository has not been maliciously modified.

Message



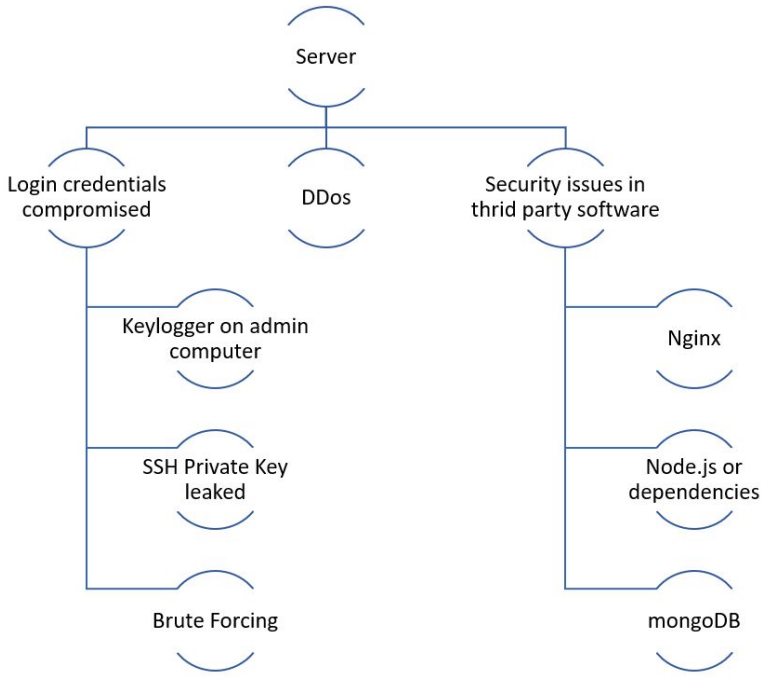
Preventing attacks on assets such as messages can be achieved by insuring that it is impractical to discover meaningful information about messages in a way outside our implementation. Initial plaintext a user composes is encrypted with AES 256 with CBC mode. A tag is then generated using HMAC and SHA-256. The keys used to encrypt and generate the tag are then encrypted with RSA 2048 with OAEP. A message contains the following information: RSA ciphertext (AES and HMAC keys), AES ciphertext (plaintext), initialization vector, and tag. These layers of protocols provide Pretty Good Privacy, confidentiality, and integrity due to the complexity of all publicly known attacks against these components to date, including brute force.

In storage, only admins are allowed access to the server and, SSH keys required to do so. This ensures that there is no unauthorized access to the server. Even if an adversary manages to get access to the server memory or

database, they are not able to read messages since they are encrypted by the clients before they are sent to the server .

In transit, https is always used for communication between client and server, which restricts the information an outside adversary can learn by observing messages in transit. Only cipher suites that support perfect forward secrecy are used. If an adversary modifies messages in transit, tag verification will discover the modifications and notify the user of the tampering. If an adversary successfully acts as a middleman between clients, they will not be able to decipher meaningful information about the messages unless they manage to break RSA 2048 with OAEP or have a user's private key.

Server



One possible attack surface against the server is distributed denial of service attacks. To prevent these from having impact on our server, we use an AWS cloud instance, which comes with AWS Shield. AWS Shield provides protection against DDos attacks.

Another attack surface is third party software vulnerabilities. We manage this by only using packages which are provided by npm, which is trusted by a large community.

An adversary might also try to compromise our login credentials to the server. It is our responsibility as administrators of the server to ensure that we securely store the SSH private key, and that no malicious software is running on our devices or development environments.

Assumptions

AES is secure, which means that if an adversary does receive a json message, they must still decrypt the RSA ciphertext in order to retrieve the message in plaintext and learn meaningful information.

2048 bit RSA with OAEP padding provides sufficient protection for messages in transit.

HMAC with SHA-256 provides sufficient integrity, enabling clients to detect whether messages have been modified in transit.

Users have securely shared RSA public keys with each other, enabling confidentiality.

Users practice good security habits, and do not disclose their RSA private key or credentials.

Users only need maximum of 4 MB per message: the maximum amount of data a JSON can hold.

Future Work

E2E Chat is currently not very user friendly. Some of the ways we would like to change this is by using alternative ways for users to share RSA public keys with each other more conveniently, such as scanning each others QR code to share public keys like how Briar does. We would like the simplify the process of creating and selecting keys for use in conversation, possibly by displaying a file manager for easy configuration.

Support multiple devices per client and multiple clients in a conversation. This can be done with the double ratchet protocol using prekeys like how Wire does, where each client regularly uploads a batch of prekeys with its public identity key. These prekeys are used to initiate a conversation with another user; a sender encrypts a message for every client and device.

Support true end-to-end communication directly between clients. No server is involved. This can be done using an implementation like how Briar does.

Send traffic over different, decentralized networks such as tor, and run the server on a hidden service. By enabling users to communicate via tor, even a global adversary will be hard-pressed to find any meaningful information through traffic analysis at all.