190003657

Computer Security
CS4203

Practical 2
15th of November 2022

The Message

University of St Andrews

Contents

Introduction

The goal of this assignment was to explore the trade off between usability and security and develop a fully-encrypted messaging system with other goodies like non-repudiability.

To achieve this, an end-to-end encrypted messaging application was developed in python - where clients connect to a server that accepts and forwards messages.

1 Design

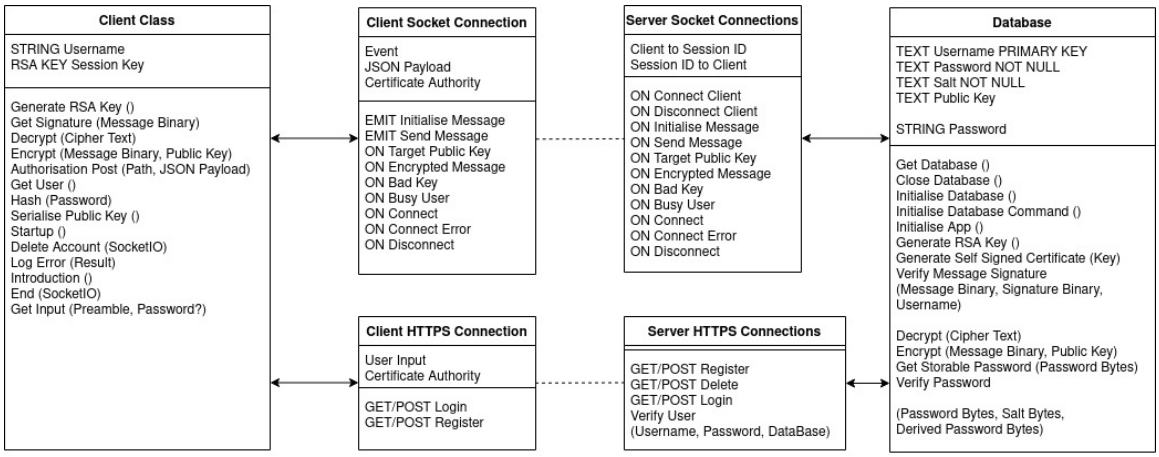

Figure 1: System Design Diagram

The messaging system can be broken down into two major components. The client and the server, which communicate with each other in order to create the messaging system. Figure 1 breaks these two components down further into variables used at various points along with the methods available to each part.

A typical server client interaction may look like the following:

1. CLIENT: Provide client your username and password for login
2. CLIENT: Create a JSON containing your username, hashed password, and a new public key for this session
3. CLIENT: Initiate POST request to server
4. CLIENT: Verify certificate of server, if valid, post data to server over HTTPS
5. SERVER: On server, validate password with derived password in the database, if valid, store the new public key
6. CLIENT: Initiate socket connection with server from client
7. CLIENT: Verify integrity of server certificate against certificate authority, if valid, create a tunnel over TLS

8. CLIENT: Send username and signature over to server
9. SERVER: If user signature matches what's on the database, add it to a mapping of active clients to a session id
10. CLIENT: User asks to chat with another user
11. CLIENT: Emit from the socket to the initialised request with username, target user, and signature
12. SERVER: Verify both users active and signature, if valid, forward target user public key to client
13. CLIENT: Enter a message, create a symmetric key and encrypt it, encrypt the message using it, encrypt the symmetric key using the forwarded public key, sign, and transfer
14. SERVER: Verify again, forward to target user
15. CLIENT: Receive message, decrypt symmetric key, then decrypt message

So how does this all work, and why was it designed this way?

1.1 Server - Client Structure

A server client structure was chosen as if you use peer to peer the other client can see your IP address. This can leave you open to being doxxed by a malicious user or have a denial of service attack used against you - so we instead communicate with a trusted server.

The server can be trusted as it is identifiable with its certificate. When we communicate with it over HTTP or a socket, we verify the certificate of the server against our certificate authority, so we can know if its identity is as it says. These connections are then run over the TLS protocol so that data on these channels are encrypted allowing us to prevent snooping.

The client can be trusted as each time a user starts a new client instance, a brand new assymetric key is generated. When we send data over to the server, we also send along a signature from it generated using the private key for our current session. The server can then verify this signature (as it has the public key from the login) and if valid, the client identity is established.

1.2 Encryption Techniques

Passwords are important pieces of information used to identify a user. Therefore they need to be securely stored so that we can prove a users identity. We start off by hashing the password before it arrives on the server. Even though the password goes through an encrypted tunnel, we wouldn't want for it to be in plaintext on the server where some admin might accidentally see it.

Next, a key derivation function [1] is used to derive a secret key to be stored in the database. This is where the password is combined with a salt, then processed by a function in order to produce a key which will be difficult to crack, as the output uses a salt only known by the server. This key though can still be compared with our original hashed password to verify if they match. With this, we can store the key on the database with assurance that a bad actor would have trouble using it even if the database were exposed. Our salt can even be stored as plaintext alongside our key as the rest of password would be unknown.

For our messages, hybrid encryption is used [2]. This is where both a symmetric key and asymmetric key are used to secure some data. In our case, with our message we can encrypt it using a symmetric key, as it allows arbitrary sized encryption whereas symmetric has a cap, then encrypt the symmetric key with the public key of our target user, and finally send it off to them. With this, only the target user can access the contents of the message as only they have the private key, which can be used to unlock the symmetric key, which yields the original message.

1.3 Logging

Logging was used so that the actions of a client and a server can be seen both during and after their run time. This allows you to visually go in and look for any anomalies if an attack were to occur against you, as you can go back and check in the past at what time may it have occurred, in which function, due to what actions. Although data such as what messages pass through a server and what input a client types in are recorded, only the hashed or encrypted versions of them are. In actual circumstances important information would never be recorded in the logs, but for this practical they are provided for clarity of what occurs during the run time of a program. Examples of these logs are given in the logs directory.

2 Implementation

2.1 Data at Rest

On the server, we store the username, derived password key, its salt, and the current public key of the user as part of the database instance. The password here is the core security concern, however this has been addressed previously. In addition, there is the private key of the server along with its certificate. Due to the core functionality they provide in our security it should be that even if they are ever exposed, time would be needed to break them. Leading to the use of a passphrase in the key necessary to use the certificate.

Although the public key of a client is stored during the active use of the server, once a client disconnects the public key for that user is set to NULL. This ensures that even if someone where to get the private key of a user it wouldn't be of much use as that user can no longer be identified using it.

Messages and the data that follows them are only ever forwarded on server. Information such as mapping of active clients to their sessions are only stored in volatile memory, disappearing after the closure of the server instance.

For the sake of this practical, the path to the key and certificate files are hard coded for ease of development. In addition, the passphrase of the key file is also hard coded. In practice, this should never occur as it poses a major security vulnerability if someone were to get access to the source code for the server. This was only provided so that for testing the user would not have to type in the passphrase too many times. Only once for the startup of a server instance.

On the client, no data is stored at rest, only existing in volatile memory for the duration of the client instance. This is due to the fact that the end user is typically the weakest link, so reducing the information they can expose as much as possible is paramount for security. To add to this, passwords when entered are hidden from view as well, like how you may enter a password in the shell, preventing the chance of displaying their password on their screen for a bad actor to see.

2.2 Authentication

The server uses a self signed certificate. This could be used in practice as well as if the certificate authority that issues the certificate that you own were to be exposed, it would leave you exposed as well due to certificate chaining. So having a self signed certificate means that you would be safe from this consequence.

On the client, RSA keys [3] are used for our asymmetric case as they provide a public and private key which can be used for signing, allowing non-repudiation, and encryption via its public key, securing our data in transit.

AES [4] is used for our symmetric encryption through the Fernet Class of the python cryptography library [5]. It is used as it allows us to encrypt messages of an arbitrary size compared to RSA which has a size cap based on how the key was defined.

SHA256 was used to hash our password before sending due to it being recommended as its peers such as SHA1 and MD5 are vulnerable to attacks [6], in addition to it being easy to implement using the provided libraries.

The salt and key derivation function chosen were to get random bytes from the operating system and the scrypt algorithm. These were based on the considerations of the python cryptography module, which references: Thomas Erin Ptacek [7], who say that it relies on the entropy of yours system, and RFC 7914 [8], which states that scrypt uses the ROMix algorithm which has been proven to be 'sequential memory-hard'.

3 Threats

As previously stated, the passphrase being stored in the source code is a major security concern and should be changed, for the same reasons as before.

In addition to this, replay attacks work fairly well against the current system. In the case that the source user is offline, this fails due to the check of active clients. However, when you replay a message that a client has send before it will be able to pass through as the server would believe the identity from the signature of the message. Leaving a major security concern as it could be possible for a denial of service attack to be carried out against a target user as the server would continue to forward replayed messages onto them.

Solutions to this problem exist. One such solution would be to rotate the session key whenever a message is sent to the server. This way, the original message would be approved as the signature would match, from which point a key that is attached to the message could be rotated with the one currently stored in the database. If a bad actor were to replay this then the signature of the replayed message would not match the public key, stopping this attack. The concern here would be the additional processing needed to generate new keys.

Another solution would be instead to have each message have an identifier, with the server keeping tracking what the next expected identifier for a given client should be. This way if a message is replayed, the next expected identifier would not match the replayed one, stopping the attack. The trade off here instead would be the additional memory required for storing these identifiers, which scales with the more users that are created.

Supply chain attacks could also occur, where during an update of libraries such as the cryptography one used malicious code is inserted into the system as well. One could solve this by instead not updating at all, but if a vulnerability with one of the methods you use is found then you would need to figure out another solution. An example could be by making your code open source so that a community can help refine it, but that leaves you with needing to ensure that there are no major concerns like with the passphrase as well.

Security through obscurity is another option, but then if an exploit is found you would never know about unless someone notifies you of it.

4 Conclusion

I believe I was able to create a fairly secure messaging system, having reasoned about why some algorithms were chosen and what security concerns the system has.

Given more time, I would like to solve the concerns like the passphrase and replay attacks, in addition to adding more features such as chat backups. This would entail another piece of data at rest as it would need to be stored in the database. The problem being how can you ensure nobody else can read that data, how could you make it so that your backup of a chat looks different to your peers - what about the privacy concerns of the peer, if they don't want a backup to occur for that chat? How would you transfer the private key used to encrypt data securely to a log in that you do on another client? There are many, many ways this practical could expand, with each one presenting their own myriad of security considerations.

5 Final Word Count: 2133

References

[1] Kaliski. *RFC 2898 Password-Based Cryptography*. 2000. URL: https://www.ietf.org/rfc/rfc2898.txt (visited on Nov. 13, 2022).

[2] Google. *Hybrid encryption*. 2022. URL: https://developers.google.com/tink/hybrid (visited on Nov. 13, 2022).

[3] IBM. *RSA private and public keys*. 2021. URL: https://www.ibm.com/docs/en/zos/2.1.0?topic=keys-rsa-private-public (visited on Nov. 13, 2022).

[4] NIST. *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*. 2001. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf (visited on Nov. 13, 2022).

[5] Individual Contributors. *Fernet (symmetric encryption)*. 2013. URL: https://cryptography.io/en/latest/fernet/ (visited on Nov. 12, 2022).

[6] IBM. *Why use SHA256 instead of SHA1?* URL: https://www.ibm.com/support/pages/why-use-sha256-instead-sha1 (visited on Nov. 13, 2022).

[7] Thomas Erin Ptacek. *How To Safely Generate A Random Number*. 2014. URL: https://sockpuppet.org/blog/2014/02/25/safely-generate-random-numbers/ (visited on Nov. 13, 2022).

[8] IETF. *The scrypt Password-Based Key Derivation Function*. 2016. URL: https://datatracker.ietf.org/doc/html/rfc7914 (visited on Nov. 13, 2022).