End-to-End Chat Application

DC Security

Donato Estolano & Connor Kobel

**Security Goals**

- Assets
    - Data
        - User Information
        - Messages
        - Private/Public Keys
        - JWT
        - Database
    - Software
        - Source Code
        - Server
- Stakeholders
    - Users
    - Developers
- Adversary

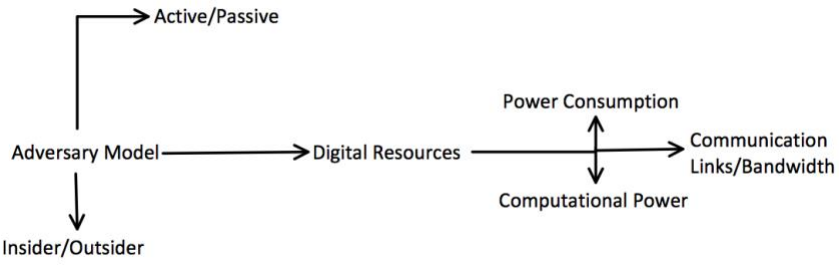


Figure 2: Adversary Model

    - Passive Adversary: someone who's tapped in observing requests being sent and results being sent back

- Active Adversary: someone who's actively trying to compromise our system
- Insider Adversary: in our case, it will be the server as it stores and knows all data being transmitted
- Outside Adversary: this could be anyone that is trying to compromise the user's credentials and data stored outside of the server
- Digital Resources: any outside component that can negatively affect our system

- Attack Surface

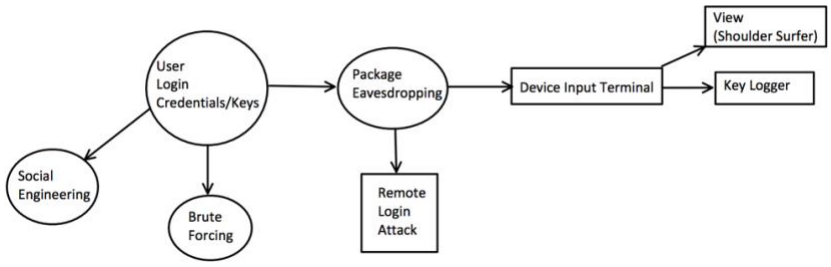


Figure 3.1: User Login Credentials/Keys Attack Surface

- Social Engineering
  - Counter: we assume that the user will not willingly give up their credentials and private/public key pair
- Brute Force
  - Counter: very costly attack that will take too long to succeed so we don't need to worry about it. Also, there's no way to really prevent this attack
- Remote Login Attack
  - Counter: we assume that the user is smart enough to never fall for tricks that would allow an adversary to gain remote access to their computer

- View/Key Logger
    - Counter: we assume that the user only uses the application at home to prevent any adversary from ever viewing their key strokes

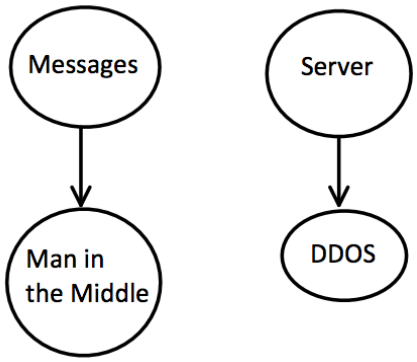


Figure 3.2: Messages/Server Attack Surface

- Man in the Middle
    - Counter: our messages are encrypted and ran through HMAC to create confidentiality and integrity making it hard for the man in the middle to even alter or view the plaintext
- DDOS

Counter: we are hoping AWS has some countermeasures regarding this type of attack

**Design and Analysis**

Our design starts at the foundation which is creating a server through AWS and have it secured with updating it to accommodate TLS 1.3 and redirecting to HTTPS. This ensures that communication from wherever to the server is secure and very unlikely to be breached. After a secured website is created, message confidentiality and integrity will be the next goal to achieve. For message confidentiality, messages will be encrypted with AES encryption with proper padding. A key will be randomly and newly generated every time to ensure that no

two messages were encrypted with the same key. To provide message integrity, messages are hashed through HMAC to generate a tag that is only unique to that message and key used. HMAC keys are also randomly and newly generated. These ensure that keys are never recycled which can potentially be exploited by the adversary. HMAC tags will be sure to prevent messages from being tampered with.

Once message confidentiality and integrity are implemented, PGP (Pretty Good Privacy) will be utilized when sending the message through the server. This means that an RSA private/public key pair will be generated to encrypt/decrypt the keys used to encrypt/decrypt the messages. Public key exchanges for the chat application will have to happen offline through physical contact or some other cloud-based storage in which the users can acquire each other's public key. These will be the foundation of secure message passing in our application. For the server and client side of things, we will be using a remote login feature for user authentication. This means that a password will never be passed through besides the registration phase which benefits the user greatly while denying an adversary at a chance to acquiring it. The server end of things will include storing the passwords after being hashed with a salt. The salt will be used with a new and randomly generated challenge for the user after every login request. The server will include four primarily used routes within a RESTful API structure.

**Code Analysis**

- Secret.py
    - Contains all code for encapsulation and decapsulation for adhering to PGP (Pretty Good Privacy)
    - Uses os, json, base64, and cryptography libraries from Python
- Chatapp.py and ChatAppTwo.py
    - Code for the GUI
    - Uses the tkinter library from Python

    - Get message requests are done through event handlers in which anytime the cursor enters or leaves most of the widgets in the window triggers that event
    - Once user has registered or logged in, ChatAppTwo is activated, exiting ChatApp
- Hmactoserver.py
    - Responsible for generating a response to the challenge sent by the server
- Server.js
    - Starting point for requests sent to the server
- Config.js
    - Holds information for database secret and connection location for the tables
- 478models.js
    - Holds the table structure for the two models used in the application: User and Message
- 478routes.js
    - Where requests are routed to the proper functions
- 478controllers.js
    - Holds all the functionalities for each request