**Project Bird**

**Authors** : João Arthur Gaia da Rocha Almeida and Thiago José Silva Santos

**About the application**

Project Bird is an alternative to common aplications of modern world. It is known that countries invest heavily in international espionage, and some of done through backdoors guaranteed by non-transparent legal agreements between private companies and the government as Snowden exposed. Since its creation, Project Bird has guaranteed a virtual private network of communication between the group of logged-in users, ensuring through open code and constant code inspection that no backdoor is opened for eavesdropping.

**Features**

*Application differentials*

* **From scratch**: The only library used is the official GNU C/C++ library, and only the most recent and proven safe functions are used.
  + Using any foreign material is hard but offers an advantage, as we ensure that data is not intercepted by backdoors in other libraries. This also becomes an advantage when it is noted that, unlike many applications that follow pre-established interfaces by external sources, we have total mastery of what happens in our application, since everything was implemented from scratch. Some of the things our implemented-from-scrath error handler deals are, for example, system interruption, blocking threads, incomplete packages and sudden disconnection of users.
    - A starting point for understanding how we do this is through the function TransmitData(..) at \*Socket\* class.
* **Complete parallelism**: Server handles a ilimited number of connections and resolve client queries in parallel.
  + Creating different threads to receive connections, authenticate users, answer questions or do any operation that can stop the server from responding., users do not need to wait for the server to respond to another user, since for each user the server allocates a specific thread. This means that an user can search the server database while another user authenticates or sends a message to other user. There is no predefined behavior and everything is done at the same time.
    - A starting point for understanding how we do this is through the function HandleUserRequest(..) at \*ApplicationServer\* class.
* **Point-to-point encryption**: All data exchange between the clients and the server is encrypted.
  + Man-in-the-middle attacks are more common than we think and we should be concerned even with the traffic that comes out of our own computer, before reaching the network layer. For this, all data is encrypted before reaching or leaving the network layer, and the decryption only happens within our application, whether the data is on the client or on the server.
    - A starting point for understanding how we do this is through \*Cripto\* class.

**Implemented applications**

* **Organizer**: stay up to date on your tasks without having to share them on non-transparent platforms like Trello.
  + Implements full task list management over periods of months or weeks.
    - **Future plans**: implement week and month management using the date and time structures provided by the GNU C/C++ library. Also, allow the creation of development groups, along with the management of tasks that are shared by these groups.
* **Messenger**: talk to whoever you want without worrying about having your message history requested by the government.
  + Implements a simple system for exchanging messages between authenticated users.
    - **Future plans**: implement user group chats and file submissions.

**Protocols**

Protocols were developed for the exchange of messages between the user and the server. As the server always keeps in which application the client is, we guarantee that protocols with similar signatures do not imply unwanted behavior.

The table below illustrates the behavior of the organizer application. The client sends a request to the server (central column) and the server responds with an appropriate response (rightmost column). By default, we define that if errors occur, we return the response \*ERROR\* for treatment.

|  |  |  |
| --- | --- | --- |
| Description | (firstly) Client sends | (secondly) Server response |
| Add task to day | ADD <day> <title> | SUCCESS |
| Delete i-th task from day | DEL <day> <index> | SUCCESS |
| Edit i-th day from day to title | EDIT <day> <index> <title> | SUCCESS |
| Get all tasks of day | DAY <day> | formated sequence of <title> |
| Get all tasks of week | WEEK | formated sequence of <day> <title> |
| Quit organizer menu | QUIT | *no response* |

Below is the table that illustrates the behavior of the messenger application.

|  |  |  |
| --- | --- | --- |
| Description | (firstly) Client sends | (secondly) Server response |
| Set yourself available to chat | ENROLL | SUCCESS |
| Get users available to chat | USERS | formatted sequence of <login> <id> |
| Get messages from user with given id | FROM <id> | formated sequence of <message> |
| Send message to user with given id | SEND <id> <message> | SUCCESS |
| Quit messenger menu | QUIT | *no response* |

There is also the possibility of having a client subtly closing the application in such a way that there is no window for sending a `QUIT` message to the server. In this case, we use the fact that the next socket transmission/reception fail, and procced accordingly to assign a status of `DISCONNECTED` to the user, which implies removing the user of all data structures allocated by the server.