BASTION

SECURE MAIL FORTRESS

By Ryan Forster

Contents

Table of Figures

List of Tables

1.0 Introduction

Now that digital communication affects all parts of our daily lives, how we protect our messages has become more crucial. Though email is one of the most popular methods to connect online, the system it relies on — SMTP — has security limitations. Throughout the years, additional steps like adding TLS and PGP were meant to boost security, but many times they are not required, difficult to configure or not used widely. In other words, these communications are likely to be intercepted, altered or imitated by hackers.

This project examines a more effective way. Rather than changing outdated protocols, it offers Bastion which is designed from scratch to be safe by design. Bastion is not built to replace email for all users around the world. In fact, it is a working demo that demonstrates how contemporary encryption and user identification systems help build a light, safe and secure protocol. The name refers to the fact that Bastion acts as a secure place for confidential communication that nobody can change.

Python is the programming language used to build Bastion and its security is accomplished through many layers. The connection is encrypted with SSL/TLS as the very start. Each Diffie-Hellman exchange guarantees a secret key that is only used in that session and Fernet is used to protect the messages by symmetric encryption. A HMAC code is put onto every message for authentication and passwords are securely lodged with bcrypt hashing. In addition, Bastion has light defence against brute-force attacks which keeps it secure from most common cybersecurity problems.

A major advantage of this project is that it is very straightforward. Bastion is completely independent and does not require additional services. It serves as an independent protocol for a client and server to talk, where all its features, including saving messages, user logins, encryption and session management, are built in. So, it’s ideal to use for teaching, experimenting or as a standard to develop other secure messengers.

In order to validate the process, Bastion was examined using virtual machines in a controlled environment. The security system was tested against common challenges such as guessing passwords and stealing data and its encryption was confirmed by reviewing the content of captured traffic as messages continued to be hidden. The results showed that Bastion reliably supports the principles of confidentiality, integrity and authentication.

We are, in essence, focused on designing and testing a communication system that is safe from the outset. This subject brings together practical programming with cryptography, showing that you don’t need complicated technology to achieve secure communication. The upcoming sections explain Bastion’s design, implementation, testing and analysis, giving a picture of both the minor and major aspects involved in secure protocol creation.

1.1 System Requirements

The Bastion secure communication protocol was developed, deployed and tested by using the following system configurations and requirements.

1. **Hardware:**

- Processor: Dual-core 2.0 GHz or higher
- RAM: Minimum 2 GB (4 GB is recommended for VMs)
- Storage: 100 MB free for Bastion files and logs

2. **Software Requirements:**

- Operating System:
  - Server: Ubuntu 22.04 LTS
  - Client: Kali Linux 2023.1 (Or any Linux-based OS with Python support)
- Python version: Python 3.10 or above
- Python packages:
  - Cryptography
  - Bcrypt
  - Ssl (built-in with Python)
  - Socket (built-in with Python)
- Networking tools for testing (Kali Linux):
  - Hydra
  - Wireshark
  - Custom Python script
- Optional:
  - VMware or Virtual Box for isolated testing

By following these specifications, Bastion could be easily run and checked for accuracy in different situations such as secure client-server simulations, checking attack risks and monitoring the functioning of protocols.

All of the source code, documentation and necessary files for Bastion – Secure Mail Fortress can be found at the public source listed below.

https://github.com/Codeon45/Bastion

2.0 Design and Architecture

Bastion was designed with security and efficiency in mind; the two major goals were: security by design and simplicity in execution. While there are many secure messaging solutions, many of them rely on retrofitting encryption into systems that already exist like SMTP, which wasn't built with security in mind (***Mailspike Technologies, 2024***). This is where Bastion truly differs, it's built from the ground up with authentication, privacy and data integrity as its founding pillars.

2.1 Bastion Overview

Bastion offers simple and strong protection to make sure messages reach their intended recipients. Everything sent on this protocol is secured by TLS encryption from the time it leaves until it gets to its destination. In other words, protocol also involves:

- Users and servers check each other's identities first.
- Adding new users is simple because it can be done right through the server interface.
- On startup, the client asks for the server IP which helps make the program more flexible and easier to use.
- Session key exchange are done by using Diffie-Hellman (D-H)
- Fernet and HMAC to encrypt and verify the messages sent in the project.
- Bastion includes email simulations and safe storage for sent messages
- Data for all user credentials is stored as hashes by bcrypt.
- Protection for brute force attacks

Both usefulness and safety are maintained thanks to this design. This work is meant to demonstrate a better solution can be made and tested locally, not replace SMTP.

2.2 Core Features

Bastion accomplishes its main security aims by combining different known and working cryptography and network tools:

Table 1: Core Features

Component	**Techonology used**	**Purpose**
Transport Security	TLS via python module ssl	Encrypts client servercommunication
Session Encryption	Diffie-Hellman Key Exchange	Establishes a shared secret for each session
Message Security	Fernet + HMAC	Confidentiality and Integrity
User Auth	Bcrypt	Secure password hashing
Storage	JSON files	Simulates server-side mailboxes

The purpose of utilizing these libraries was to maintain tough security standards without creating extra obstacles for implementers (***Github.io, 2025***).

2.3 Architecture Diagram

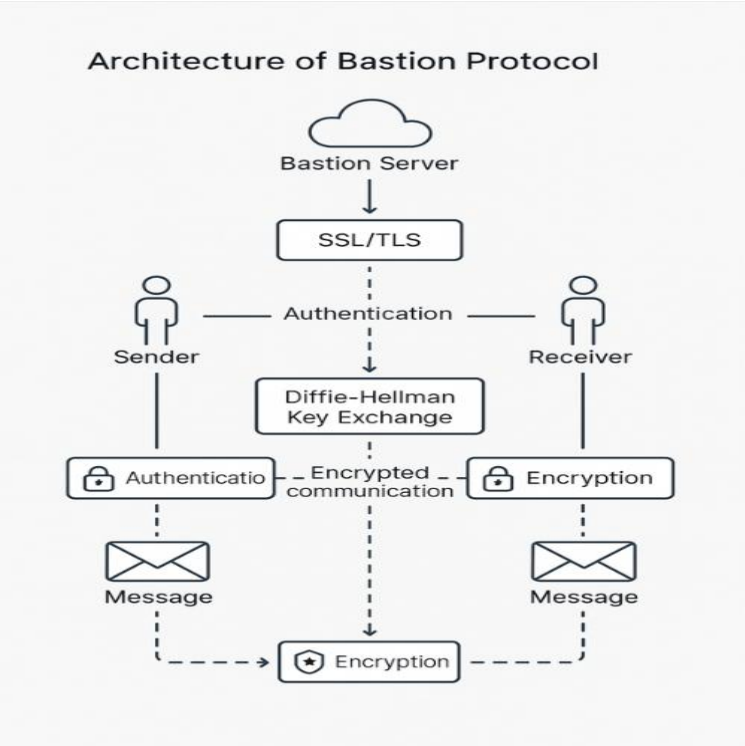


Figure 1: Architecture of Bastion Protocol

The diagram highlights how Bastion works by showing client and server interaction that occurs over a channel secured with TLS. It proves the authentication flow steps, a key exchange method and shuttling protected messages via Fernet and HMAC (***Mertens, 2025***). A secure socket is used for all communication, ensuring that data stays confidential and correct all the time.

2.4 Security Layers

Bastion is designed to defend data by using several layers of security at once. All traffic is securely encrypted using TLS 1.2 (***Banreet, 2024***). By default, the server uses a self-signed certificate which the client verifies until it's manually disabled during development. Using Diffie-Hellman Key Exchange, a new key is agreed upon for each session, so any messages sent before won't be compromised if a key is stolen (***Gillis, 2022***). The symmetric keys are used for encrypting messages, that come from the Diffie-Hellman secret and then hashed by HKDF (***Krawczyk and Pasi Eronen, 2025***). It makes certain that your conversation will be private. HMAC is added to each message to confirm it wasn't altered as it moved from one network to another (***GeeksforGeeks, 2020***). All user passwords are stored by using the bcrypt algorithm. Before storing, their passwords are both hashed and salted with bcrypt to avoid brute-force attacks (***Grigutyté, 2023***). If someone is blocked for too many unsuccessful login attempts, their account remains locked out for another 30 seconds. All these layers come together to shield data from eavesdropping, stopping people from making copies of data and help us avoid using other hacking methods.

2.5 Simplicity by Design

Bastion has excellent encryption, but its architecture is kept very simple by design. Because Bastion uses TCP, it eliminates the need for third-party server support and all of its information is saved in easy-to-use files that do not require databases. If you need email for a small group or a place where Postfix (***Postfix.org, 2025***) would be too much, Bastion makes a better choice. Because everything is transparent in the architecture, you can easily fix, test and analyse your system. It may not be ready for production, but it does have the function of a working, secure and testable mail alternative.

The decision was made not to include GUI because it would have added weight and complexity to what was designed to be a simple and clean command-line system.

3.0 Implementation

3.1 Development Environment

To make Bastion, a simple and practical toolset was built that allows for easy access, flexible testing and clear implementation. I built the main application on a **Windows PC** using **Visual Studio Code** because it is easy and has the built-in terminal feature that lets me iterate and debug as I create.

The core development of the Bastion Protocol used **Python 3.13**, mainly because it is easy to read and cryptographic libraries are widely available. Python's **ssl** module and the **cryptography** library are used by the protocol to encrypt sockets with **TLS** and to apply **Fernet** encryption, **HMACs** and **Diffie-Hellman** key exchanges. In addition, the **bcrypt** module was chosen for hashing passwords.

To simulate real world attacks and conditions, two virtual machines were set up:

- Ubuntu 22.04 LTS as the Server
- Kali Linux as the client machine and also attacker, specifically for launching brute force attacks and packet sniffing.

These virtual machines were run using VMware Workstation Pro. Both were configured to be in the same network, allowing them to communicate with each other (***Bastiaansen and Lanigan, 2019***).

Before starting development, cryptographic bcrypt packages were installed by using pip command, and additionally **OpenSSL** was used to generate a self-signed certificate and key for the server (***Openssl.org, 2025***). Command used:

```
C:\Users\ryanf>openssl req -x509 -newkey rsa:4096 -keyout bastion_key.pem -out bastion_cert.pem -days 365 -nodes
```

Figure 2: OpenSSL command

After the creation of the certificate, its used for wrapping the server in TLS and allows the client to authenticate to the server during the handshake process. During the development, certificate verification was momentarily disabled, but for testing and deployment it was turned back on again to test full SSL validation.

Below is a table containing the main tools and environments:

Table 2: Tools and Environment

Tools/Component:	**Description:**
Python 3.13	Programming Language
Visual Studio Code	IDE
Ubuntu 22.04	Server for Bastion
Kali Linux	Client and Attacker
VMware Workstation Pro	Virtual Machine Management tool
Cryptography library	Encryption, HMAC and key exchange
Bcrypt module	Password Hashing
Ssl module	TLS socket wrapping and certificate handling
OpenSSL	Generat

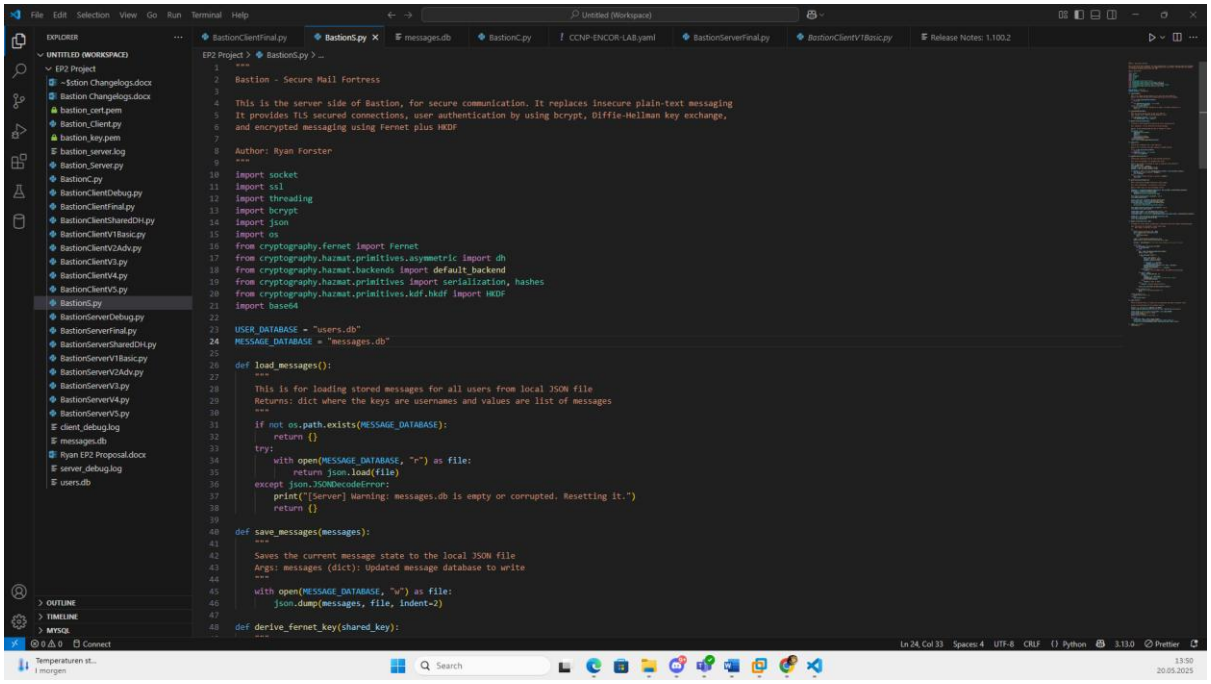


Figure 3: Visual Studio Code IDE environment

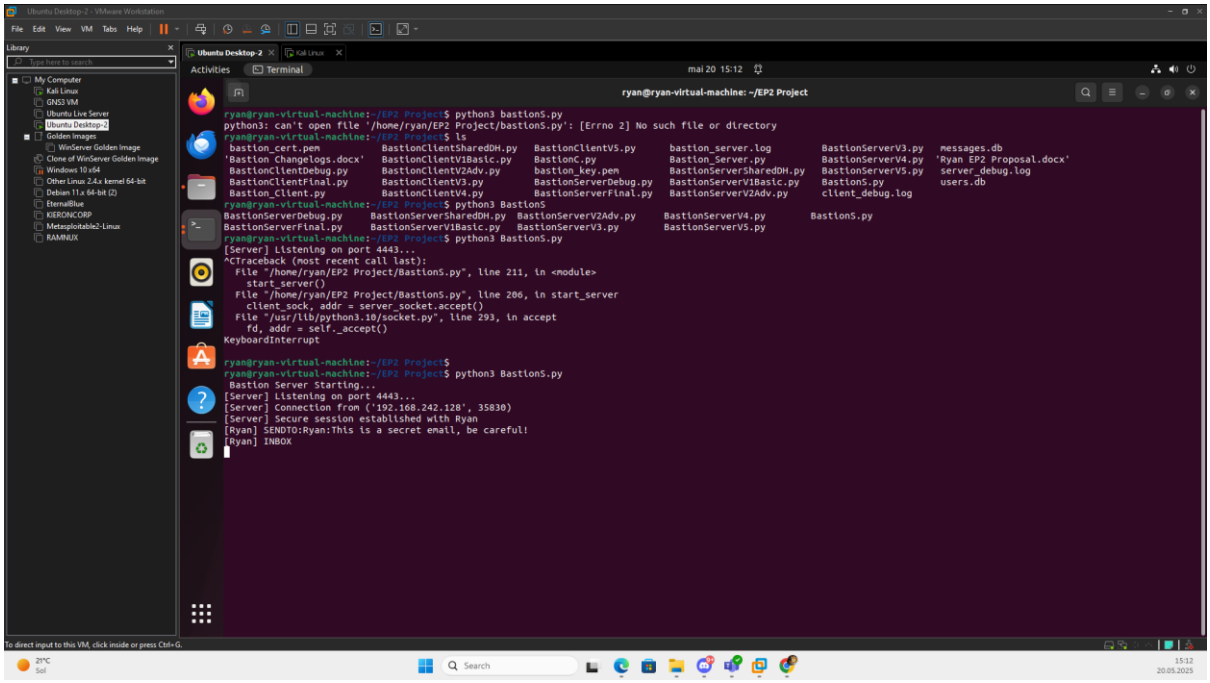


Figure 4: VMware Workstation Pro environment

3.2 File Structure Overview

Throughout the development of Bastion, a clean and minimal file structure was kept, in order to keep it simple, easy to test and clear separation of roles between client and server. Because Bastion is not built on top of an already existing mail infrastructure, there are no complex directory trees, and it doesn't rely on external databases. Everything is organized inside the project folder, like data, logic and security materials.

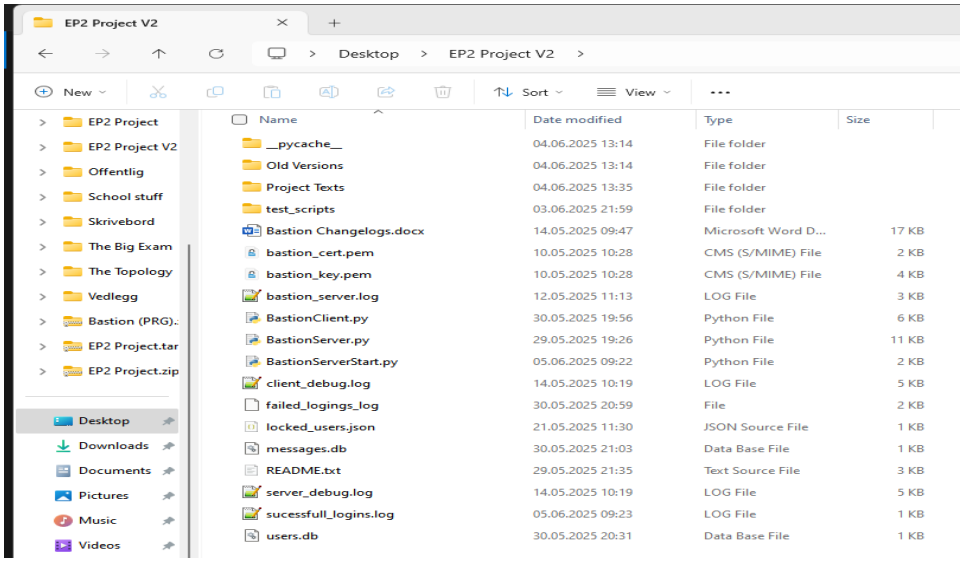


Figure 5: File Structure

Key Components:

- **BastionServer.py**
  - Here you can access all the server-side code from Bastion. The protocol includes authenticating users, setting up Transport Layer Security (TLS), changing keys with Diffie-Hellman, capturing messages and placing them into the user's inbox. It further logs and resists attempts by brute force attackers.

- **BastionServerStart.py**
  - This is a launcher to run the server or add users to the users.db file, making it easy to register new users (**BSS**)

- **BastionClient.py**
  - This is Bastion's code for the client side. When you log in, it sets up a connection, exchanges keys, displays a menu and lets you communicate through messages.

- **Bastion_cert.pem / bastion_key.pem**
  - These are the files generated with OpenSSL, they are the certificates needed for enabling TLS communication between server and client.

- **Users.db**
  - A Jason file used to simulate a database. Credentials are stored in it, passwords are hashed by bcrypt.

- **Messages.db**
  - Messages in this file are grouped with a username for each grouping. An entry is made up of a collection of received messages which provides a real-looking, message-organized inbox (BCSM).

- **Failed_logings_log**
  - Try to log in unsuccessfully and the file will note when and where you attempted to access, using time and IP information. With this, you can find brute force attacks and look into various security events.

- **Successful Logins Log**
  - This file keeps track of all successful logins, with the same functions as failed logins log, using time and IP information. Its good function to keep track of users**.**

- **Locked_users.db**
  - Users stored in this file are blocked, and the file ensures they remain blocked even if server goes down, by always loading the file at start.

Other files are README, Changelogs, debugs and folder for scripts used for testing attacks against Bastion.

- With the README file, future users can find descriptions of how to set up the protocol.
- Changelogs is a list of changes that Bastion has gone through its development
- To fix bugs, debug was implemented when the Bastion was being programmed.
- Test Scripts contains all the scripts used during testing, mostly for attacks (BCPS, BSRA)

3.3 Coding Methodology, Implementation Process and Evolution

Bastion protocol was formed using a planned and continuous process, applying both secure programming concepts and hands-on experimentation. The goal was to develop a working model that clearly explains basic ideas in secure communication without the extra complexity of heavy or old frameworks. The work was broken into different phases, each built on the last and had continuous testing along the way.

3.3.1 Development Approach

Bastion was created using a bottom-up coding method involving modules.

Phase 1 – *Secure connection*

TLS (SSL) was the first part to be implemented, with the goal in mind of encrypting the communication during transmission. In this process, a certificate was generated with OpenSSL and inserting the ssl module in the code (**BSV1, BCV1**)

Phase 2 – *User Authentication*

For the authentication part, python coding and bcrypt was used. Passwords are hashed and salted before storing them in the file users.db. During the log in phase, these credentials are checked against the hashes in that file. (**BSV5, BCV5**)

Phase 3 – *Secure Session Establishment*

Due to Diffie-Hellman Exchange, the server and client could negotiate a session key on the fly and use the keys for their communication. After that, HKDF uses the key to generate a secret which Fernet uses for symmetric encryption (**BSV6, BCV6**).

Phase 4 – *Message Handling*

After the channel was secured, messaging and receiving features were added. Messages sent over Fernet are encrypted and integrity comes from HMAC. Messages are saved to messages.db (BSM) by the server, so users can access them at any time (**BSV8, BSV9**)

Phase 5 – *Security Hardening*

To harden the security of Bastion, brute force protection was added long with account lockout, and failed log in attempts. Each failed attempt is logged with a timestamp, IP address and temporary bans for too many failed attempts repeatedly (**BSF**).

Phase 6 – *Testing and Attacking*

Bastion was put through brute force attacks, TLS sniffing and tampering tests. In this part of development, both the inbox cleansing and error handling options were improved (**BSBD, BSBD2**)

3.3.2 Programming Language and Libraries

The project was implemented entirely with Python, due to its good readability and strong ecosystem for cryptography and networking. These were the libraries used:

Library	**Purpose**
Socket	Basic TCP communication
Ssl	TLS encryption
Cryptography.fernet	Symmetric Encryption (Fernet)
Cryotography.hazmat	Diffie-Hellman and HKDF key derivation
Bcrypt	Secure password hashing
Json	Storage for credentials and messages
Threading	Handling multiple clients simultaneously
Datetime/time	Account lockouts and timeout

Because of these modules, Bastion could use a modern and secure messaging system without needing tools such as databases or SMTP servers.

3.3.3 Code Style and Documentation

Functions were structured so their names made sense, they had inline comments and were described by Python docstrings to ensure no confusion. To make code more modular, their goal was to make functions short and focused on what they needed to do. For example:

```
def derive_fernet_key(shared_key):
    """
    Derives a Fernet-compatible key using HKDF from the DH shared secret.
    """
```

As a result, the code was simple and easy to work on. In the future, it can be used with different languages or frameworks.

3.3.4 Error handling

Functions were set up to deal with difficulties due to disconnection, decryption, unusual format errors, timeout cases and altering messages (using HMAC).

As a result, users continue to have stable experiences and malicious attempts do not cause the server to crash.

3.4 Security Challenges and Mitigation

When creating Bastion, main attention was given to blocking and anticipating usual cybersecurity threats. Every stage of setting up the system presented problems with keeping data safe at rest, while transferring and during the authentication process. The next sections describe the main risks included in the attack surface and how they were handled in the Bastion protocol.

3.4.1 Brute-Force and Credential Stuffing Attacks

Challenge:

This type of attack means the system tries all kinds of usernames and passwords in a fraction of a second. If usernames are something others can guess, it becomes even more dangerous. If no defence is in place, a would-be attacker can illegally access the system or overwhelm it by making too many login attempts (***Owasp.org, 2025***).

Mitigation:

After a person tries to log in unsuccessfully three times in a row, their account is temporarily put on hold for 30 seconds. Because the server adds timestamps, the timeout cannot be avoided by setting the computer clock manually.

All attempts when a login fails are saved with the client's IP and the time the attempt was made. There is now traceability which can be checked in detail during forensic evaluation.

After a login fails, a very short delay of about two seconds is added to make attacks by robots less possible. As the users.db data would contain only bcrypt hashes and some salt, there would be only a low chance of brute-force or table attacks succeeding on them.

3.4.2 Man-in-the-middle (MitM) and Eavesdropping

Challenge:

Should an intruder get their hands on unencrypted messages being sent along the network, they may be able to read or change private data. All client-server systems are vulnerable to this type of risk (***Cisco, 2025***).

Mitigation:

All data between the client and server is kept private using TLS 1.2. For the development phase, a self-signed certificate is applied which users must verify by themselves.

All encrypted messages come with an HMAC signature. Should a single byte change while transmitting the message, the server will deny the message and add a record of it in the log.

Using the Diffie-Hellman certificate, each client creates a different key for each communication. If session keys were to change, the new session would still be protected.

3.4.3 Message Tampering and Replay Attacks

Challenge:

Challenge: In systems using messages, attackers may try to break in by changing, doubling or adding messages to change the actions of the server or pretend to be users (***Stepan Ilyin, 2025***).

Mitigation:

With Fernet Encryption + HMAC, your messages are encrypted as well as signed. Prior to processing, timestamps and HMACs found in Fernet tokens are always checked.

If a token is picked up by intruders, the data within can't be read or used again since the session key is not known.

After you download a message from the database on the server, it is automatically cleared out of your inbox. It stops others from copying or altering what's in the message.

3.4.4 Packet Sniffing

Challenge:

Wireshark, as well as similar tools, allows you to capture info sent over an unencrypted network connection. It is a major concern for SMTP, for example, since messages travel without being encoded (***GeeksforGeeks, 2018***).

Mitigation:

With Bastion's TLS encryption, any traffic we transmit over its network looks like a secure encrypted messages when looked at using network analysis tools.

Since Bastion relies on a distinct protocol, it becomes much harder for automatic sniffer tools to make sense of transmitted data.

Demonstration: No human-readable material was present in the captured packets, confirming that Bastion blocks any data in plain form.

3.4.5 Local Time Manipulation

Challenge:

With the lockout depending on the client's clock, someone could easily evade it by setting their time back.

Mitigation:

All the calculations for enforcing lockouts happen based on the server's time from datetime.now(). Because of this, you can't use a local clock change to unlock accounts or make a key valid again.

3.4.6 Lack of GUI and User error

Challenge:

Although only a minor concern, lack of a graphical user interface could make users more likely to do the wrong thing. Whenever users submit the wrong data or validation is missed, errors may be found.

Mitigation:

The client uses a simple plan to show each step in the process to the user.

The system efficiently handles receiving empty inputs, malformed messages and problems with unexpected disconnects.

A system could be built using libraries such as **Tkinter** or **PyQt** to enable using a GUI, instead of regularly having to type commands and format input manually (***GeeksforGeeks, 2017***).

3.5 Testing Environment and Procedure

Testing the reliability and security of Bastion was made possible by creating a test setting in virtualization. The test-run was meant to replicate actual communication situations and see if Bastion can handle well-known cyberattacks such as brute-force logins and packet checking. This section covers the organization of the lab, the equipment and tools applied and the ways the protocol is tested.

3.5.1 Virtual Lab Setup

Testing was done with two virtual machines operating within a local host setup.

- In the Ubuntu 22.04 LTS (Server), Bastion was hosted and set up to use a self-signed TLS certificate on port 4443 (BCCF, BCCT). The server ensured login, handled giving out keys and managed saving the messages.
- I used Kali Linux as a Bastion client when I wanted to connect and as an attack platform for testing with tools like Hydra, Wireshark and my own brute-force script.

All the machines were connected to a sealed virtual network so they could talk to each other using IP addresses. Both systems could call out to each other and communicate securely via sockets over the given port.

3.5.2 Tools Used

Table 3: Tools used

Tool	Purpose
Python 3.13	Running Bastion Client and Server scripts
Hydra	Brute-force attack
Wireshark	Capturing and Analysing packets
Custom Python Script	Brute-force attack script
RockYou.txt	Worldlist for brute-force attack
OpenSSL	Generate TLS certificates

3.5.3 Testing Procedure

1. **Normal Use Case Scenario**

Purpose:

Encourage secure login, establish a key exchange and make sure messages are sent. (BSBF, BCBF1)

Steps:

- TLS is used for connecting clients to the server.
- A valid username and password are required to use the username (BCWU).
- Diffie-Hellman is used to share the session key.
- A Fernet encryption algorithm and HMAC technique are used for the message.

Outcome:

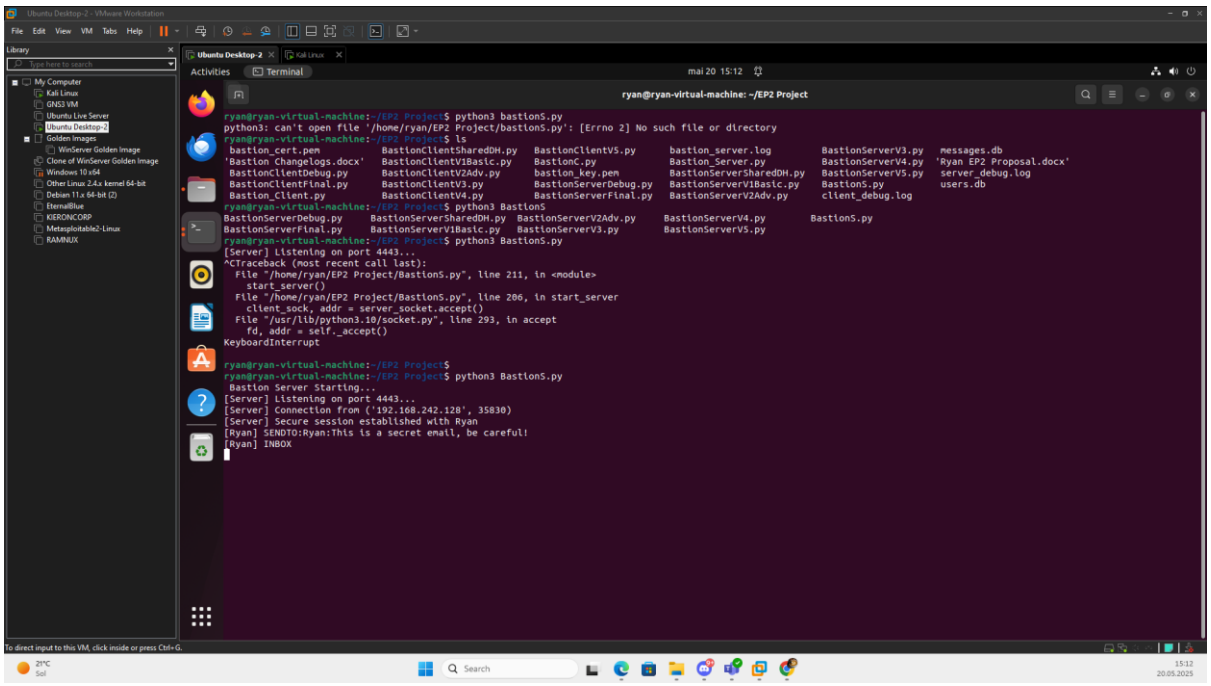


Figure 6: Basic Functionality

Secure message is saved to your account and is now accessible in your inbox, after which the message is removed once you have read it.

It passed the test — both the features and security succeeded.

2. **Hydra Brute Force Attack**

Purpose:

Investigate if automated tools such as Hydra are able to force a Bastion login without a password.

Attempted Method:

Hydra was utilized for attempting logins against Bastion, using RockYou.txt wordlist (***Lurey, 2023***).

hydra -l ryan -P /usr/share/wordlists/rockyou.txt 192.168.242.131 -s 4443 <protocol>

But instead of **HTTP POST**, **SSH**, **FTP** or any other protocol families commonly used like **HTTPS**, Hydra cannot read the protocol Bastion uses.

Outcome:

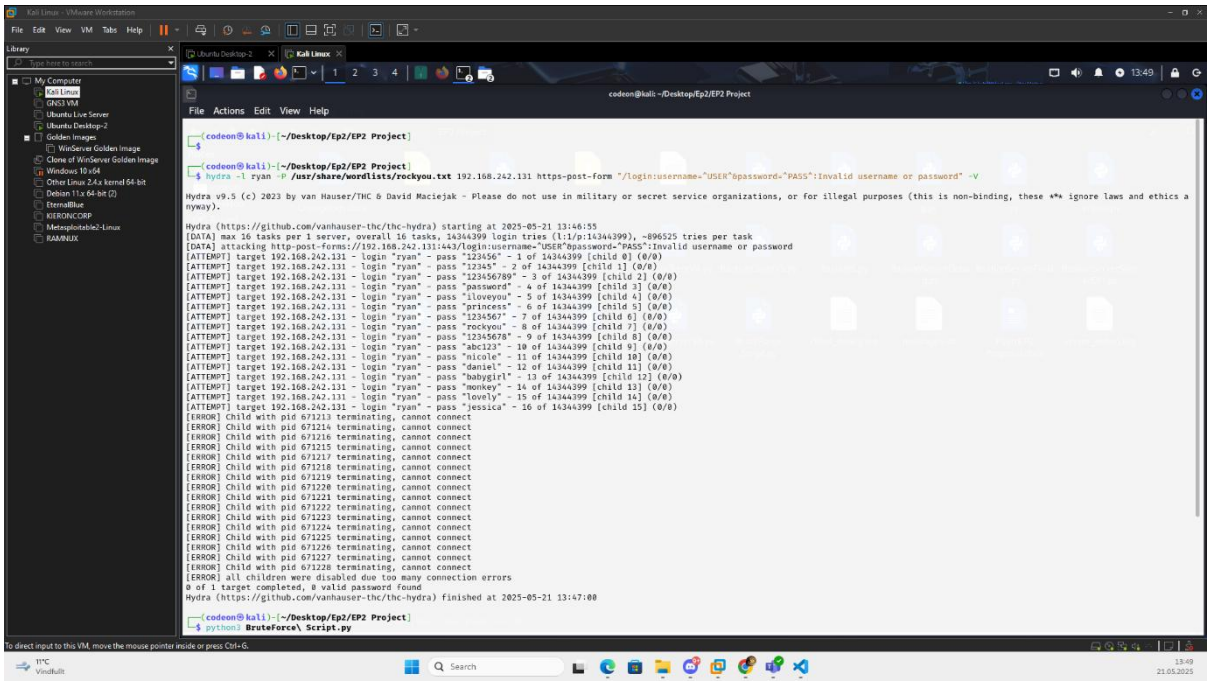


Figure 7: Hydra attack failed

All of the requests to Hydra produced errors and the connection was lost.

There were no attempts to connect with the Bastion service at any point.

Conclusion:

It is impossible to attack Bastion using Hydra because Bastion communicates with a unique and encrypted protocol. As a consequence, Hydra cannot fully make sense of how the login

system works. All brute-force efforts should be performed by making a custom Python script that reproduces the protocol. Thanks to the way protocol was designed and TLS encryption, Bastion did not fall to Hydra.

3. **Custom Script Brute Force Attack**

Purpose:

Since Hydra had no chance, now Bastion brute force defence will be tested with a custom python script (BCBF, BCBFA).

Steps:

- Python script allows you to repeat guessing passwords.
- Each time a login didn't succeed, the counter went up.

Outcome:

```
ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certif

┌──(codeon㉿kali)-[~/Desktop/Ep2/EP2 Project]
└─$ python3 BruteForce\ Script.py
[*] Trying password: 1234
[Server] Invalid username or password.
[*] Trying password: password
[Server] Invalid username or password.
[*] Trying password: admin
[Server] Too many failed attempts. User locked for 30 seconds.
[*] Trying password: noroff
[Server] Too many failed attempts. Try again later.
[*] Trying password: letmein
[Server] Too many failed attempts. Try again later.
[*] Trying password: Noroff
[Server] Too many failed attempts. Try again later.

┌──(codeon㉿kali)-[~/Desktop/Ep2/EP2 Project]
└─$
```

Figure 8: Custom Script Brute Force Attack

- Locking the account was initiated at the third consecutive failure.
- Your login request was not successful due to timing out.
- Having to delay and track on the server side slowed down the attempts to retry requests.

The outcome was Pass, thanks to server validation.

4. **Wireshark Packet Sniffing**

Purpose:

Check when sensitive data (such as login and messages) could be snatched during the attack.

Steps:

- Wireshark was used on the Kali machine at the same time the test emails were sent.
- The traffic was studied using the TLS filter.

Outcome:

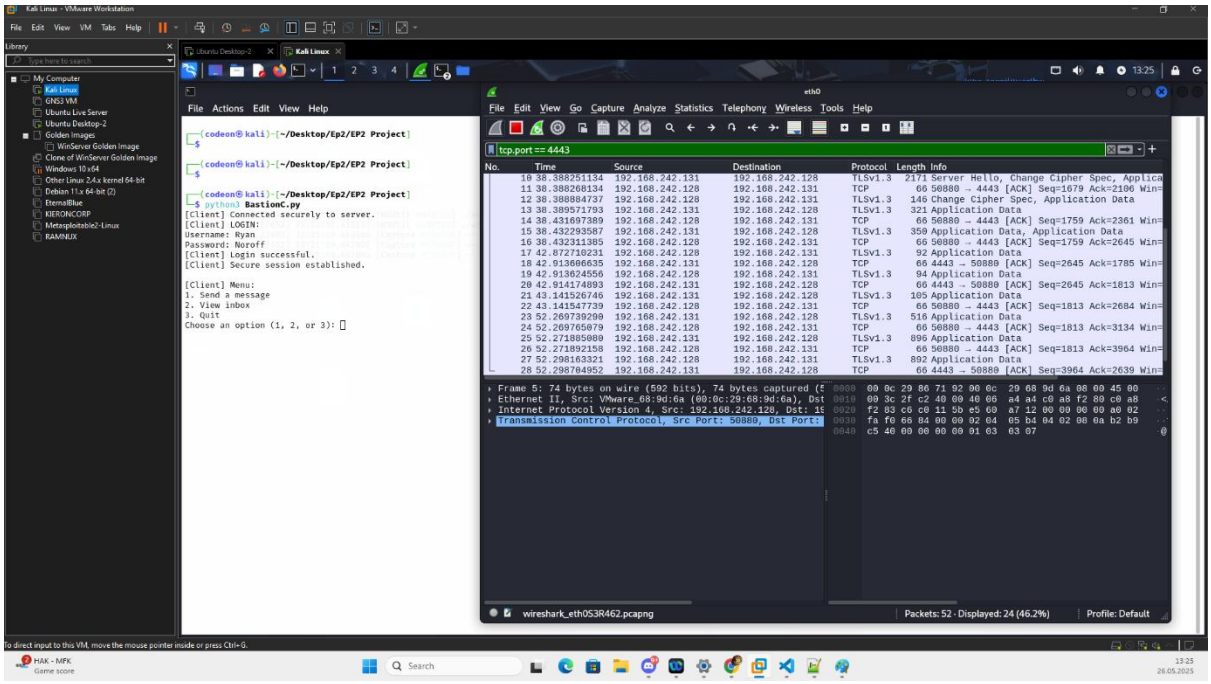


Figure 9: Wireshark Sniffing Packet

- No readable data on the packets could be seen because all traffic was encrypted. (BCWS)
- TLS authentication was performed, the data was encrypted and the certificate was transferred in an encrypted way.

The examination concluded with TLS + Fernet fully securing the traffic.

5. **Tampering and Replay attacks**

Purpose:

To find out if captured or modified packets could be utilized again.

Steps:

- Encrypted messages had to be manually recopied and sent out a second time.
- Messages that were altered were discovered to have an invalid HMAC.

Outcome:

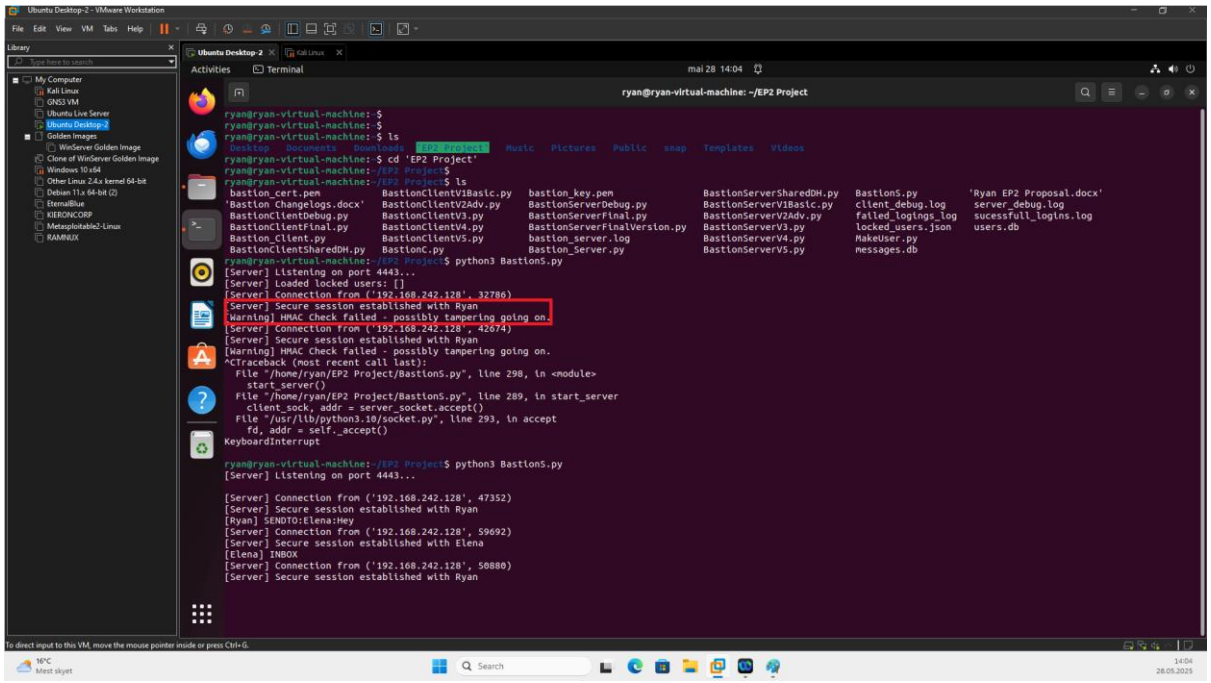


Figure 10: Messa tampering HMAC

- The server rejected any packets it said had failed in its HMAC checks.
- Trying to read the log again was not possible because of unique keys per session.

The result was clear — the integrity of the message was maintained.

3.5.4 Summary of Test Outcomes

Table 4: Summary of Test Outcomes

Test Type	Outcome	Notes
Secure Messaging	Passed	Encryption with TLS, DH, Fernet, and HMAC worked according to plan.
Brute-Force Protection	Passed	Lockouts out after 3 failed attempts.
Packet Sniffing	Passed	Nothing is readable because of encryption
Message Integrity	Passed	HMAC protected integrity
Replay Attempts	Passed	HMAC and session keys completely stopped it.

4.0 Evaluation

4.1 Protocol Strengths

Bastion shows several important features that prove it is effective for secure communication. It is designed to be very secure from the start and skips the problems found in older systems such as SMTP that do not adequately consider encryption or integrity.

4.1.1 Security by Design

While other security systems add an encryption layer onto existing protocols, Bastion puts in place secure protocols. Through TLS, communication is secure from the start and using Diffie-Hellman gives each conversation its own secret key. Knowing that one session key can be stolen, Bastion assures that previous and future sessions remain unaffected.

4.1.2 End to End Encryption and Integrity

Fernet automatically uses a symmetric key to secure and time-sensitive each message. It is also improved by HMAC verification, checking if the message has escaped tampering while sent across the network.

4.1.3 User Authentication and Secure Credentials

All user credentials are always coded or put into some form other than plain text. Therefore, passwords are not stored directly but are hashed with bcrypt, a hashing method especially built to stop brute-force attempts. By using the login process, security is protected and when users fail to log in many times, accounts are automatically locked, stopping diction and brute-force attacks.

4.1.4 Lightweight and Modular Design

Bastion is designed so that it does not complicate things. Pythone and common libraries such as ssl, bcrypt and cryptography are used for all features, making everything easy to check and keep updated. JSON files save messages in the project, standing in for inboxes and avoiding the need for a full database for use in a prototype or academic demo.

4.1.5 Compatibility with Virtualized Test Environments

Bastion was successfully installed and tested in virtual machines that ran Ubuntu (server) and Kali Linux (client). It proves that Bastion can easily be used on various systems and supports testing security systems using real world hacking tools.

4.1.6 *Resistance to common Cyber Attacks*

Attempts to guess or crash passwords with brute force were put to the test using Bastion as well as using Hydra and custom scripts. Because of rate limiting, hashing user credentials and leaving accounts locked after a timeout, all the attacks were either stopped or rendered unsuccessful — this demonstrated that there was a strong defence against most standard authentication attacks.

4.2 Limitations of Bastion

Although Bastion is a lightweight and safe method for communication, its limits should be recognized when planning to use it or expand it outside of academic research.

4.2.1 No Asynchronous Communication

Currently, Bastion functions using a synchronized client-server approach. You need to be connected actively to be able to send and receive messages. Since Bastion expects online, end-to-end sessions, the system is not practical for many common email tasks.

4.2.2 Limited User Scalability and Management

Users are authenticated using the flat JSON file named users.db. However, it isn't possible to use this process in larger settings. Users cannot sign up or be assigned roles and the system does not work with major authentication systems (like LDAP and OAuth). For enterprise-scale systems, Bastion has to be changed in a major way before it can be implemented.

4.2.3 No GUI or Web interface

Bastion is entirely controlled using command lines. Because of this, the protocol is easy to develop and test, but it may be tough for users without technical knowledge. If the interface was improved with graphs or a web version, it would be much easier for people who aren't familiar with terminal tools to use it.

While a GUI based on Tkinter was tried, it was put aside when it showed issues and the goal was to maintain the main program's basic and focused CLI nature

4.2.4 Certificate Management and Trust Issues

While being developed, Bastion relies on TLS certificates that clients consider unknown and thus not automatically trusted. While testing works best with these certificates, using them in the real world allows for security vulnerabilities if users do not question their authenticity. For production, you should only use a certificate authority (CA) that you trust.

4.2.5 Message Store Basic

New inboxes are created with the help of local JSON files. As a result, it does not include proper access control, redundancy or persistence features like today's message queues or mail servers do. There isn't a way to search, sort or manage messages other than pulling them up and deleting them.

4.2.6 No Protection Against Advanced Threats

The security system in Bastion can withstand brute force and simple replay attacks, but it does not have defence for more sophisticated threats.

- Timing attacks
- Traffic Analysis
- Denial of Service (DOS)

- Advanced MitM strategies, like certificate spoofing

4.2.7 No Multi-Factor Authentication (MFA)

At this moment, Bastion depends solely on username-password login. Standard practice in today's security systems is to require multi-factor authentication. Using token-based MFA or confirming with a mobile app would make access and security much stronger than before.

4.3 Comparing with Existing Solutions

Assessing Bastion's practical value requires a comparison with both old and current methods of secure communication systems. For better understanding, comparing these categories requires using either traditional email methods (SMTP with STARTTLS or PGP) or popular secure messaging apps like Signal or ProtonMail.

4.3.1 Bastion Vs. Traditional Email (SMTP + TLS/PGP)

Table 5: Bastion Vs. Traditional Email (SMTP + TLS/PGP)

Criteria	**Bastion**	**SMTP + TLS/PGP**
Encryption	Mandatory TLS + Fernet	Optional TLS, PGP is user-dependent
Authentication	Built-in with bcrypt hashing	External, often basic
Integrity	HMAC Verification	PGP Signature (if configured)
Easy of Setup	Easy Python-based setup	Needs mail server, DNS, PGP handling
Message Delivery	Simulated Inbox (JSON)	Asynchronous delivery via server relay
Replay Protection	Session bound keys, HMAC	Nothing, unless configured
Brute Force Defence	Account lockout, logging	Rarely implemented natively
Usability	CLI only	Widely used, GUI clients available

Interpretation:

SMTP-based email can be safely used by adding Pretty Good Privacy (PGP) or TLS, but it is the user who must set things up correctly (***Buckbee, 2020***). The design of Bastion makes security automatic, so it cannot be used insecurely by accident. Bastion does not offer the flexible systems of features found in regular email platforms.

4.3.2 Bastion Vs. Secure Messaging Apps (Signal, ProtonMail)

Table 6: Bastion Vs. Secure Messaging Apps

Criteria	**Bastion**	**Signal/ProtonMail**
Encryption	DH, Fernet, HMAC	Modern ciphers
Infrastructure	Self hosted JSON based system	Full cloud-based backend
Authentication	Username, password	MFA, biometric, device paring
Interface	CLI only	Polished GUI/mobile apps
Scalability	Limited to local setups	Designed for millions of users
Open Source	Yes	Yes (Signal), Partially (ProtonMail)
Traffic Analysis	No	Yes

Interpretation:

Current messaging apps like Signal or ProtonMail offer more advanced features and a better design; however, their private, closed setup worries some users about privacy and transparency (***Signal Messenger, 2025***) (***Proton, 2023***). The best use for Bastion is in education or with team communications, since it is more focused on control and exposure than on growing or ease of use.

Conclusion:

Bastion isn't intended to compete with large-scale business tools or simple consumer apps. It is designed to show that by using essential security principles — TLS, DH key exchange, symmetric encryption, HMAC and authentication — one can develop a new secure communication method. It has strong security features and administration, but its lack of scalability, user interface and useful options keep it behind more grown-up systems.

4.4 Summary of Evaluation

Bastion was designed as a secure communication example addressing the common issues found in email and long-established protocols. On the basis of testing and analysis, these conclusions were made:

Strengths

- With Bastion, every message you send or receive is always encrypted and validated. Unlike retrofitted approaches, security is built in and essential from the beginning.
- It was made more convenient for users to type in the server IP while connected and have accounts generated by the server.
- Use of TLS, Fernet and HMACs in message checks guarantees that stolen or tampered messages are noticed by the system.

- All passwords are protected by hashing them with bcrypt and by locking out accounts when brute-force attacks take place.
- The system is both simple to use and easy to check. By choosing simple storage methods like JSON, it is able to safeguard transactions with complex cryptographic algorithms.
- Tests show that Bastion successfully resists brute force, replay attacks and packet sniffing.

Limitations

- Since you need to type commands in a dedicated window, Bastion might not be the easiest platform for people without IT knowledge. A GUI was created briefly, but it was decided to remove it for the sake of improved testing and simpler deployment.
- Bastion is developed to fit small environments such as secure lab networks or small teams. It has not been created to work with huge volumes of traffic.
- Certificates must be set and security environments paired manually which takes some knowledge of the process.
- There is No Queue or Delivery Delay: Unlike regular postal systems, Bastion doesn't store or send messages through multiple servers. Messages are saved directly on your device.

Verdict

Bastion meets its aim by providing a secure email-like communication method prototype. It gives a clear understanding of how several cryptographic tools can be worked into a secure system using only their internal features. Bastion works efficiently and usefully in teaching, research and controlled scenarios.

5.0 Conclusion

Bastion, a secure messaging protocol, was developed in this project to fix basic security vulnerabilities found in standard email platforms. Unlike SMTP which is not as secure since it was not developed with these main principles from the start, Bastion was built with confidentiality, integrity and authentication at the center.

Bastion relies on modern techniques such as TLS for safe data delivery, Diffie-Hellman for selecting session keys, Fernet for locked data, HMAC to verify messages and bcrypt to protect user passwords. They were chosen for being secure as well as for matching suitably with streamlined, easy to maintain code structures. In addition, functions were added to keep track of login failures and turn off accounts temporarily, simulating reliable behavior for users.

Testing and implementing Bastion was done by Python in an Ubuntu server and a Kali Linux client under controlled conditions. Testing revealed that the protocol can tolerate brute-force threats, that messages cannot be read by packet sniffers and that key agreement and encryption functions are flawless.

Bastion wasn’t created to replace modern email platforms, but it shows that secure protocols can be designed without needing additional security features afterward. The project could still be developed by adding options for asynchronous delivery, building a graphical interface and ensuring the servers will support the project’s growth.

All things considered; Bastion succeeded in creating a safe prototype for communication. It gives us practical insight into cryptography and serves as a learning guide for people studying secure systems. Performing and testing actions helped show how protocols are built, systems are secured and challenges in dealing with real cyber threats occur.

6.0 Recommendations

Bastion – Secure Mail Fortress has proven to be a strong and secure messaging system, but there are still several areas where making improvements can improve its efficiency, how easy it is to use and how quickly it can grow.

6.1 Better Usability with GUI

Work on adding a graphical user interface (GUI) was attempted, but it turned out to be too difficult, so a command-line interface was kept for stability and ease of use. To be used more widely, especially by people who do not code, providing a solid GUI with frameworks such as Tkinter, PyQt or Electron (via Python bridge) would enhance user comfort and make things simpler.

6.2 Transition to a Database-Backed Backend

At this moment, JSON files are used by Bastion to save the details of users and their exchanged messages. It is appropriate for creating a prototype, but with a growing number of users, it might not work as well. You can achieve higher performance, reliable indexing and stronger data security by moving to SQLite when testing locally and using PostgreSQL when hosting remotely (**Abiodun Eesuola, 2024**).

6.3 Advanced Brute Force Mitigation

While Bastion uses strong security by limiting failed logins and locking accounts for a time, the security can still be improved with the following add-ons:

- IP Based rate limiting
- CAPTCHA Support (if GUI included)
- Permanent lock outs if failed attempts keep happening

6.4 Certificate Authority

Now, Bastion relies on self-signed TLS certificates, so it works for testing but not for larger, real deployments. Using a recognized Certificate Authority such as Let's Encrypt would create more trust in the system and eliminate TLS validation issues (**Letsencrypt.org, 2024**)

6.5 More Encryption Options

The current use of encryption in Bastion is Fernet (AES-CBC behind the scenes). In future versions, it would be useful to add some of the following:

- Using ChaCha20 allows for speeding up encryption on phone devices (**Crawford, 2023**).
- Regularly changing the keys for forward secrecy

6.6 Secure Message Deletion

Messages get deleted after opening them in the inbox, even so, it may still not be fully erased securely. In addition, secure methods or temporary saving of messages which allow users to control, would improve privacy.

6.7 Test on more platforms

Bastion was so far run within a virtual machine (Ubuntu Server and Kali Linux). Having more tests on Windows, macOS and various network configurations would help Bastion be available and safe for more people.

7.0 References

Abiodun Eesuola (2024). *SQLite vs PostgreSQL: A Detailed Comparison*. [online] Datacamp.com. Available at: https://www.datacamp.com/blog/sqlite-vs-postgresql-detailed-comparison [Accessed 5 Jun. 2025].

Banreet (2024). *Enable Transport Layer Security (TLS) 1.2 overview - Configuration Manager*. [online] Microsoft.com. Available at: https://learn.microsoft.com/en-us/intune/configmgr/core/plan-design/security/enable-tls-1-2 [Accessed 27 May 2025].

Bastiaansen, R. and Lanigan, R. (2019). *VMware*. [online] Search VMware. Available at: https://www.techtarget.com/searchvmware/definition/VMware [Accessed 29 May 2025].

Buckbee, M. (2020). *What is PGP Encryption and How Does It Work?* [online] Varonis.com. Available at: https://www.varonis.com/blog/pgp-encryption [Accessed 29 May 2025].

CISA (2025). *Secure by Design | CISA*. [online] Cybersecurity and Infrastructure Security Agency CISA. Available at: https://www.cisa.gov/securebydesign [Accessed 27 May 2025].

Cisco. (2025). *All Roads Lead to Ransomware: Inside Cisco Talos Threat Hunters*. [online] Available at: https://www.cisco.com/site/us/en/learn/topics/security/what-is-a-cyberattack.html#~types-of-attacks [Accessed 27 May 2025].

Crawford, D. (2023). *What is ChaCha20? | Proton VPN*. [online] Proton VPN. Available at:https://protonvpn.com/blog/chacha20?srsltid=AfmBOoolCZPeGRFEWZXdgDEQNHUh8ZflKzJwgRBrsC3S6pniDbwX8f3S [Accessed 5 Jun. 2025].

Cryptography.io. (2025). *Welcome to pyca/cryptography — Cryptography 46.0.0.dev1 documentation*. [online] Available at: https://cryptography.io/en/latest/ [Accessed 27 May 2025].

GeeksforGeeks (2017). *Python Tkinter*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/python-gui-tkinter/ [Accessed 27 May 2025].

GeeksforGeeks (2018). *What is Packet Sniffing?* [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/what-is-packet-sniffing/ [Accessed 27 May 2025].

GeeksforGeeks (2020). *What is HMAC (Hash based Message Authentication Code)?* [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/what-is-hmachash-based-message-authentication-code/ [Accessed 27 May 2025].

Github.io. (2025). *Libraries & Pandas*. [online] Available at: https://broadinstitute.github.io/2024-09-27-python-intro-lesson/libraries.html [Accessed 29 May 2025].

Gillis, A.S. (2022). *Diffie-Hellman key exchange (exponential key exchange)*. [online] Search Security. Available at: https://www.techtarget.com/searchsecurity/definition/Diffie-Hellman-key-exchange [Accessed 27 May 2025].

Grigutyté, M. (2023). *What is Bcrypt and how it works?* [online] NordVPN. Available at: https://nordvpn.com/blog/what-is-bcrypt/ [Accessed 27 May 2025].

Letsencrypt.org. (2024). *Let's Encrypt*. [online] Available at: https://letsencrypt.org/ [Accessed 5 Jun. 2025].

Lurey, C. (2023). *Understanding RockYou.txt: A Tool for Security and a Weapon for Hackers*. [online] Keeper Security Blog. Available at: https://www.keepersecurity.com/blog/2023/08/04/understanding-rockyou-txt-a-tool-for-security-and-a-weapon-for-hackers/ [Accessed 29 May 2025].

Mailspike Technologies (2024). *Understanding the Issues with SMTP Protocol*. [online] Anubisnetworks.com. Available at: https://www.anubisnetworks.com/blog/understanding-the-issues-with-smtp-protocol [Accessed 27 May 2025].

Mertens, X. (2025). *Have You Ever Heard of the Fernet Encryption Algorithm?* - SANS Internet Storm Center. [online] Available at: https://isc.sans.edu/diary/30146 [Accessed 29 May 2025].

Openssl.org. (2025). *openssl-req - OpenSSL Documentation*. [online] Available at: https://docs.openssl.org/master/man1/openssl-req/ [Accessed 27 May 2025].

Owasp.org. (2025). *Brute Force Attack | OWASP Foundation*. [online] Available at: https://owasp.org/www-community/attacks/Brute_force_attack [Accessed 27 May 2025].

Postfix.org. (2025). *The Postfix Home Page*. [online] Available at: https://www.postfix.org/ [Accessed 27 May 2025].

Proton. (2023). *Proton: Privacy by default*. [online] Available at: https://proton.me/ [Accessed 29 May 2025].

Python documentation. (2018). *ssl — TLS/SSL wrapper for socket objects*. [online] Available at: https://docs.python.org/3/library/ssl.html [Accessed 27 May 2025].

Signal Messenger. (2025). *Signal Messenger: Speak Freely*. [online] Available at: https://signal.org/ [Accessed 29 May 2025].

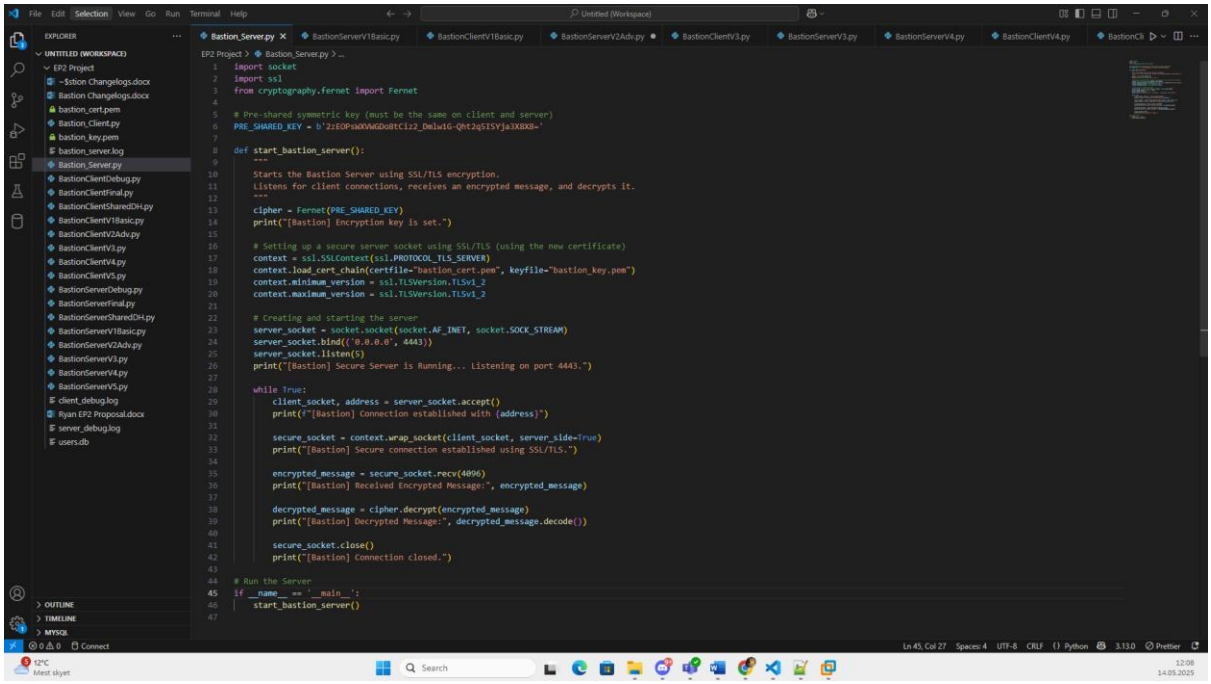
Stepan Ilyin (2025). *Replay Attacks*. [online] Wallarm.com. Available at: https://www.wallarm.com/what/replay-attacks [Accessed 27 May 2025].

W3schools.com. (2025). *W3Schools.com*. [online] Available at: https://www.w3schools.com/python/ [Accessed 29 May 2025].

8.0 Appendix

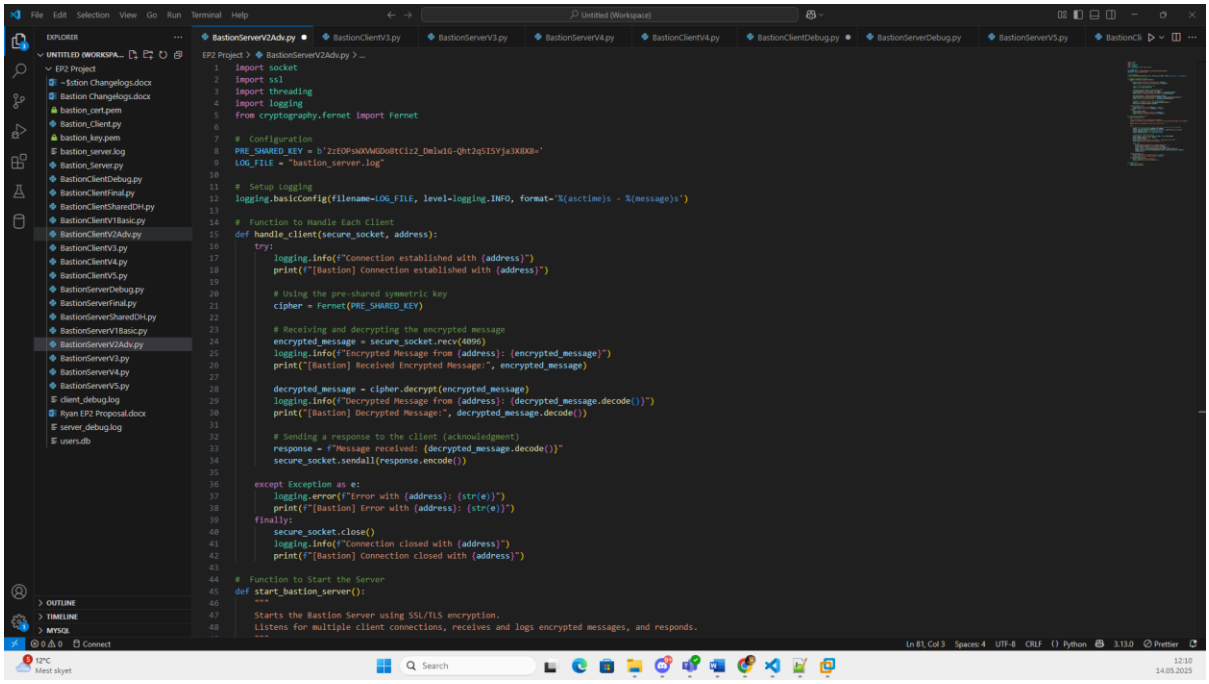
8.1 Bastion Server Code Iterations

Bastion Server V1 (BSV1)

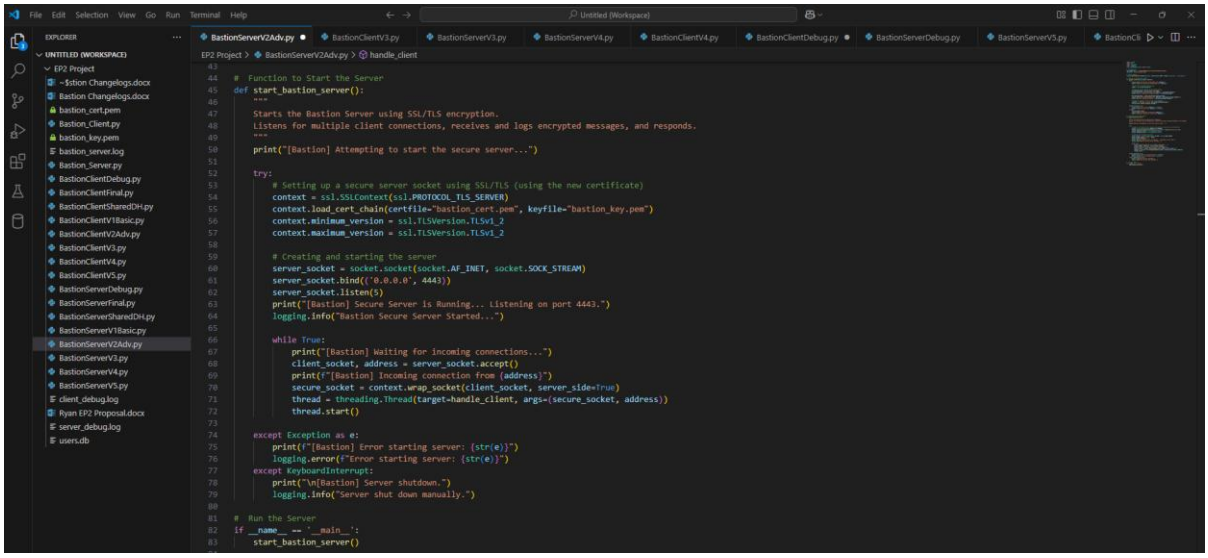


```python
import socket
import ssl
from cryptography.fernet import Fernet

# Pre-shared symmetric key (must be the same on client and server)
PRE_SHARED_KEY = b'2zEOPsWXVWGDo8tCiz2_Dmlw1G-Qht2q5ISYja3X8X8='

def start_bastion_server():
    """
    Starts the Bastion Server using SSL/TLS encryption.
    Listens for client connections, receives an encrypted message, and decrypts it.
    """
    cipher = Fernet(PRE_SHARED_KEY)
    print("[Bastion] Encryption key is set.")

    # Setting up a secure server socket using SSL/TLS (using the new certificate)
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile="bastion_cert.pem", keyfile="bastion_key.pem")
    context.minimum_version = ssl.TLSVersion.TLSv1_2
    context.maximum_version = ssl.TLSVersion.TLSv1_2

    # Creating and starting the server
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('0.0.0.0', 4443))
    server_socket.listen(5)
    print("[Bastion] Secure Server is Running... Listening on port 4443.")

    while True:
        client_socket, address = server_socket.accept()
        print(f"[Bastion] Connection established with {address}")

        secure_socket = context.wrap_socket(client_socket, server_side=True)
        print("[Bastion] Secure connection established using SSL/TLS.")

        encrypted_message = secure_socket.recv(4096)
        print("[Bastion] Received Encrypted Message:", encrypted_message)

        decrypted_message = cipher.decrypt(encrypted_message)
        print("[Bastion] Decrypted Message:", decrypted_message.decode())

        secure_socket.close()
        print("[Bastion] Connection closed.")

# Run the Server
if __name__ == '__main__':
    start_bastion_server()
```

Bastion Server V2 (BSV2)

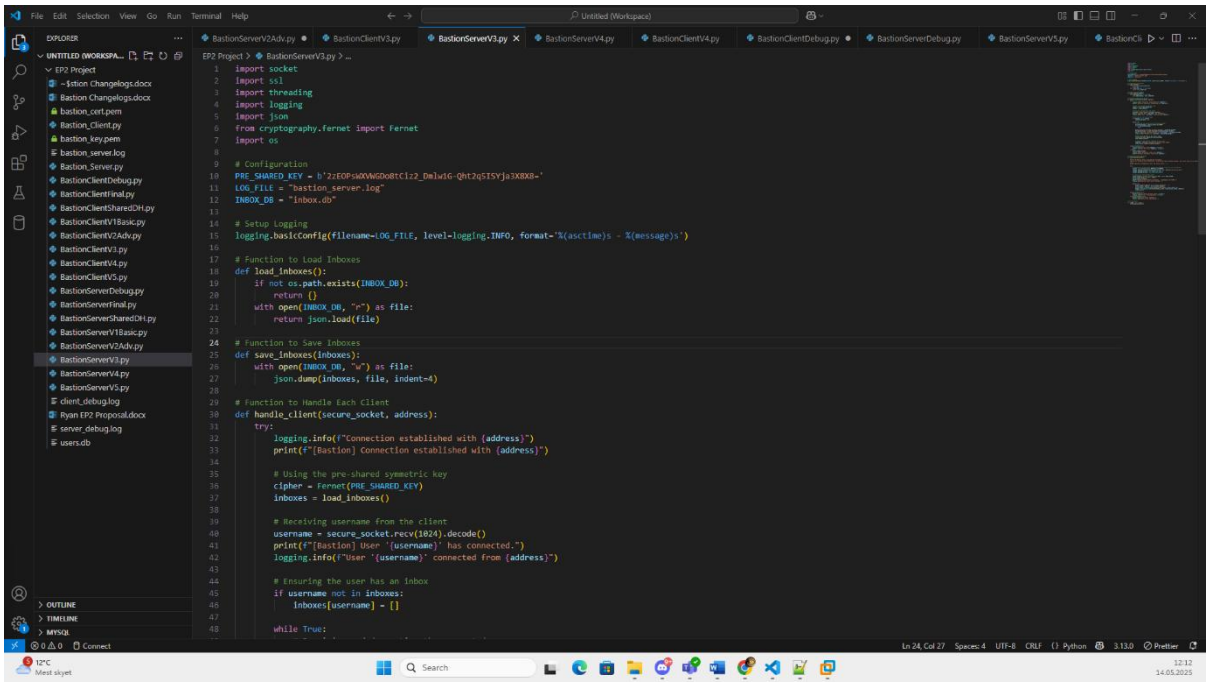


```python
import socket
import ssl
import threading
import logging
from cryptography.fernet import Fernet

#  Configuration
PRE_SHARED_KEY = b'2zEOPsWXVWGDo8tCiz2_Dmlw1G-Qht2q5ISYja3X8X8='
LOG_FILE = "bastion_server.log"

#  Setup Logging
logging.basicConfig(filename=LOG_FILE, level=logging.INFO, format='%(asctime)s - %(message)s')

#  Function to Handle Each Client
def handle_client(secure_socket, address):
    try:
        logging.info(f"Connection established with {address}")
        print(f"[Bastion] Connection established with {address}")

        # Using the pre-shared symmetric key
        cipher = Fernet(PRE_SHARED_KEY)

        # Receiving and decrypting the encrypted message
        encrypted_message = secure_socket.recv(4096)
        logging.info(f"Encrypted Message from {address}: {encrypted_message}")
        print("[Bastion] Received Encrypted Message:", encrypted_message)

        decrypted_message = cipher.decrypt(encrypted_message)
        logging.info(f"Decrypted Message from {address}: {decrypted_message.decode()}")
        print("[Bastion] Decrypted Message:", decrypted_message.decode())

        # Sending a response to the client (acknowledgment)
        response = f"Message received: {decrypted_message.decode()}"
        secure_socket.sendall(response.encode())

    except Exception as e:
        logging.error(f"Error with {address}: {str(e)}")
        print(f"[Bastion] Error with {address}: {str(e)}")
    finally:
        secure_socket.close()
        logging.info(f"Connection closed with {address}")
        print(f"[Bastion] Connection closed with {address}")

#  Function to Start the Server
def start_bastion_server():
    """
    Starts the Bastion Server using SSL/TLS encryption.
    Listens for multiple client connections, receives and logs encrypted messages, and responds.
    """
```

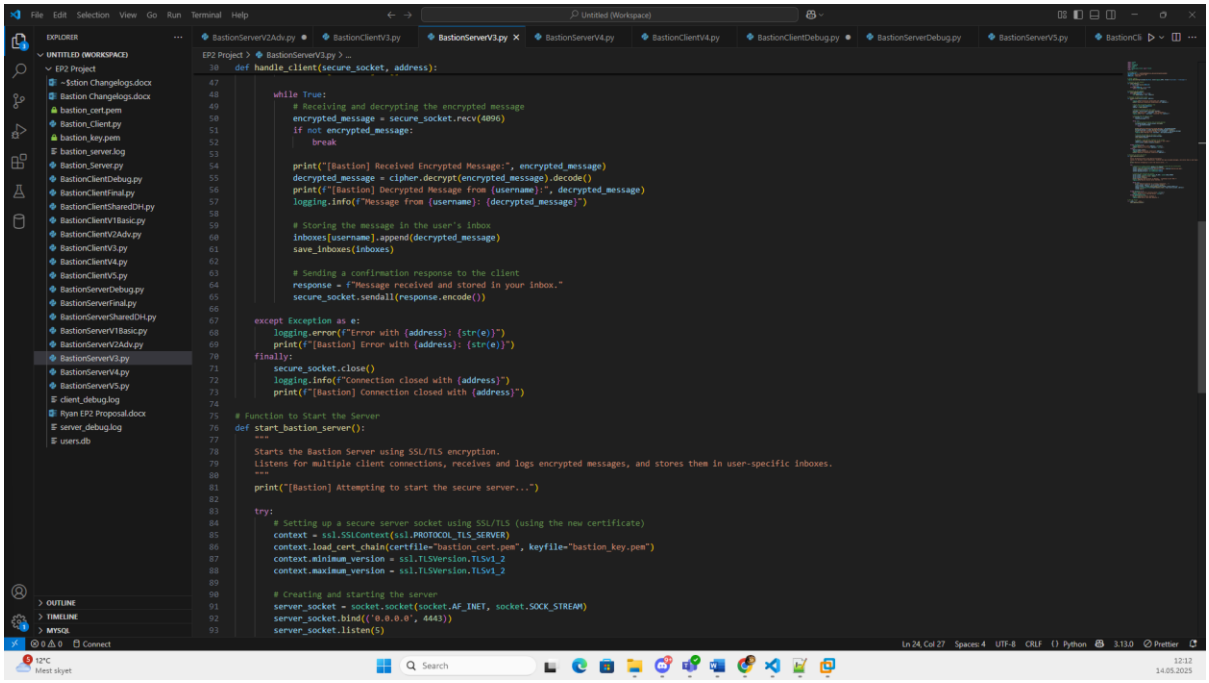


```python
#  Function to Start the Server
def start_bastion_server():
    """
    Starts the Bastion Server using SSL/TLS encryption.
    Listens for multiple client connections, receives and logs encrypted messages, and responds.
    """
    print("[Bastion] Attempting to start the secure server...")

    try:
        # Setting up a secure server socket using SSL/TLS (using the new certificate)
        context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
        context.load_cert_chain(certfile="bastion_cert.pem", keyfile="bastion_key.pem")
        context.minimum_version = ssl.TLSVersion.TLSv1_2
        context.maximum_version = ssl.TLSVersion.TLSv1_2

        # Creating and starting the server
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_socket.bind(('0.0.0.0', 4443))
        server_socket.listen(5)
        print("[Bastion] Secure Server is Running... Listening on port 4443.")
        logging.info("Bastion Secure Server Started...")

        while True:
            print("[Bastion] Waiting for incoming connections...")
            client_socket, address = server_socket.accept()
            print(f"[Bastion] Incoming connection from {address}")
            secure_socket = context.wrap_socket(client_socket, server_side=True)
            thread = threading.Thread(target=handle_client, args=(secure_socket, address))
            thread.start()

    except Exception as e:
        print(f"[Bastion] Error starting server: {str(e)}")
        logging.error(f"Error starting server: {str(e)}")
    except KeyboardInterrupt:
        print("\n[Bastion] Server shutdown.")
        logging.info("Server shut down manually.")

#  Run the Server
if __name__ == '__main__':
    start_bastion_server()
```

Bastion Server V3 (BSV3)

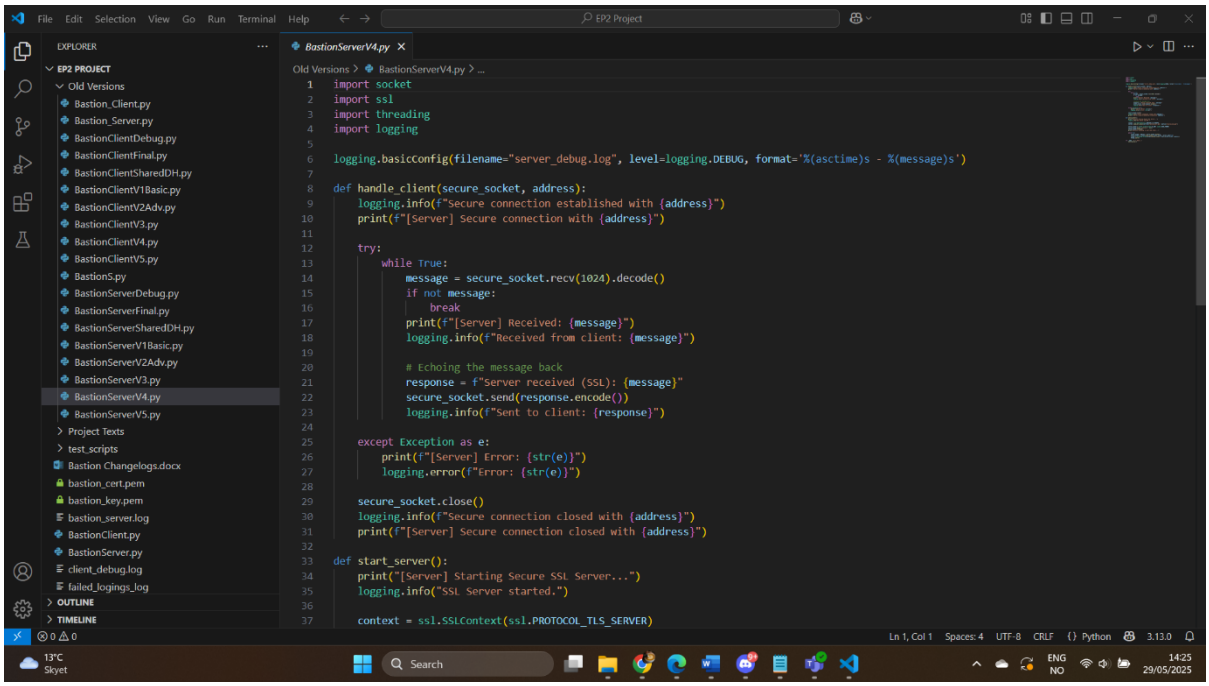


```python
import socket
import ssl
import threading
import logging
import json
from cryptography.fernet import Fernet
import os

# Configuration
PRE_SHARED_KEY = b'2zEOPsWXWGDo8tCiz2_DmlwiG-Qht2q5ISYja3X8X8='
LOG_FILE = "bastion_server.log"
INBOX_DB = "inbox.db"

# Setup Logging
logging.basicConfig(filename=LOG_FILE, level=logging.INFO, format='%(asctime)s - %(message)s')

# Function to Load Inboxes
def load_inboxes():
    if not os.path.exists(INBOX_DB):
        return {}
    with open(INBOX_DB, "r") as file:
        return json.load(file)

# Function to Save Inboxes
def save_inboxes(inboxes):
    with open(INBOX_DB, "w") as file:
        json.dump(inboxes, file, indent=4)

# Function to Handle Each Client
def handle_client(secure_socket, address):
    try:
        logging.info(f"Connection established with {address}")
        print(f"[Bastion] Connection established with {address}")

        # Using the pre-shared symmetric key
        cipher = Fernet(PRE_SHARED_KEY)
        inboxes = load_inboxes()

        # Receiving username from the client
        username = secure_socket.recv(1024).decode()
        print(f"[Bastion] User '{username}' has connected.")
        logging.info(f"User '{username}' connected from {address}")

        # Ensuring the user has an inbox
        if username not in inboxes:
            inboxes[username] = []

        while True:
```

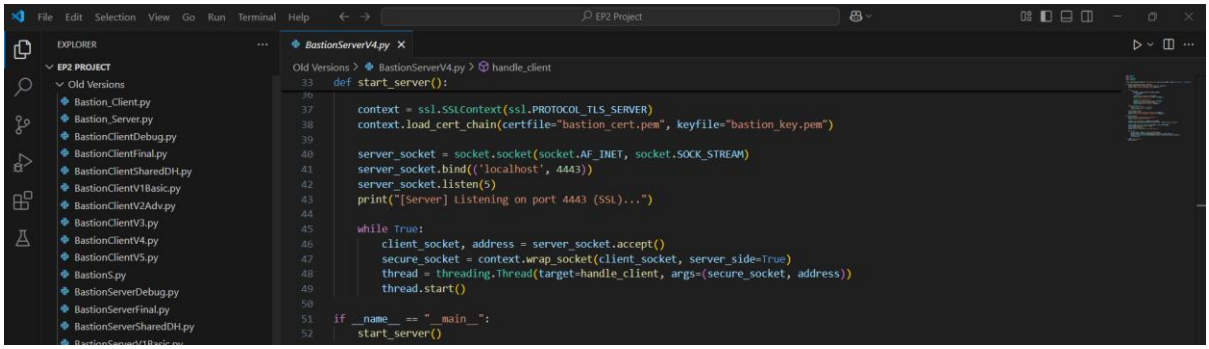


```python
    def handle_client(secure_socket, address):

        while True:
            # Receiving and decrypting the encrypted message
            encrypted_message = secure_socket.recv(4096)
            if not encrypted_message:
                break

            print("[Bastion] Received Encrypted Message:", encrypted_message)
            decrypted_message = cipher.decrypt(encrypted_message).decode()
            print(f"[Bastion] Decrypted Message from {username}:", decrypted_message)
            logging.info(f"Message from {username}: {decrypted_message}")

            # Storing the message in the user's inbox
            inboxes[username].append(decrypted_message)
            save_inboxes(inboxes)

            # Sending a confirmation response to the client
            response = f"Message received and stored in your inbox."
            secure_socket.sendall(response.encode())

    except Exception as e:
        logging.error(f"Error with {address}: {str(e)}")
        print(f"[Bastion] Error with {address}: {str(e)}")
    finally:
        secure_socket.close()
        logging.info(f"Connection closed with {address}")
        print(f"[Bastion] Connection closed with {address}")

# Function to Start the Server
def start_bastion_server():
    """
    Starts the Bastion Server using SSL/TLS encryption.
    Listens for multiple client connections, receives and logs encrypted messages, and stores them in user-specific inboxes.
    """
    print("[Bastion] Attempting to start the secure server...")

    try:
        # Setting up a secure server socket using SSL/TLS (using the new certificate)
        context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
        context.load_cert_chain(certfile="bastion_cert.pem", keyfile="bastion_key.pem")
        context.minimum_version = ssl.TLSVersion.TLSv1_2
        context.maximum_version = ssl.TLSVersion.TLSv1_2

        # Creating and starting the server
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_socket.bind(('0.0.0.0', 4443))
        server_socket.listen(5)
```

Bastion Server V4 (BSV4)

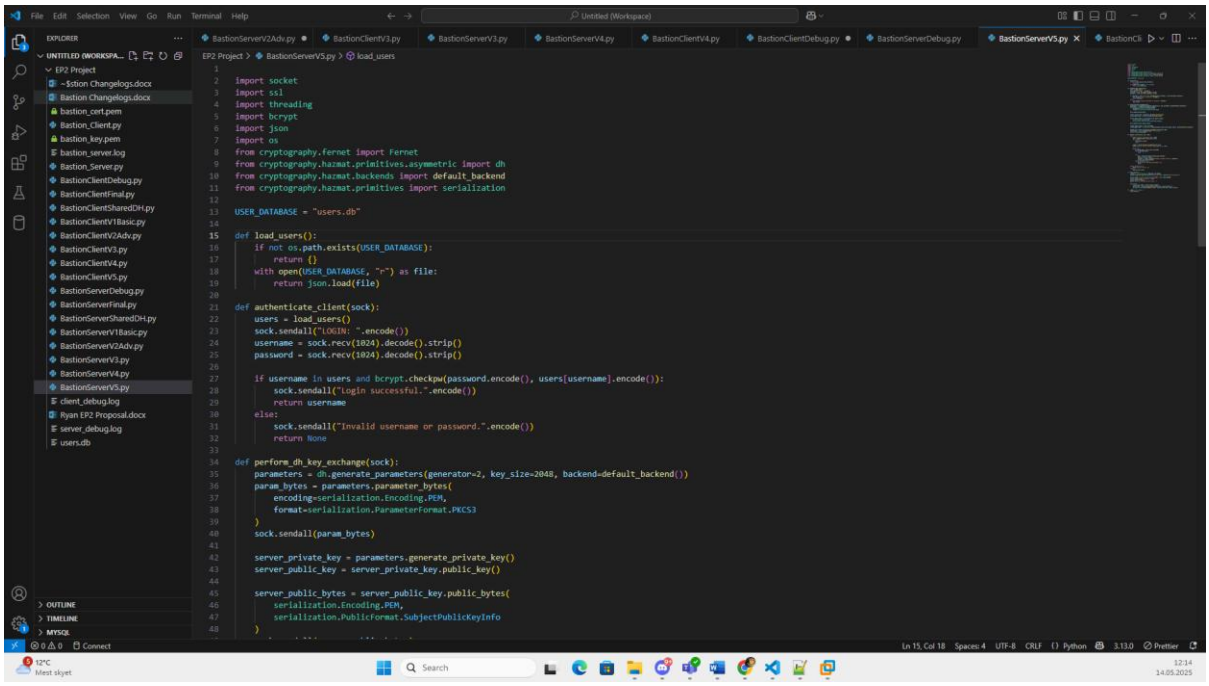


```python
import socket
import ssl
import threading
import logging

logging.basicConfig(filename="server_debug.log", level=logging.DEBUG, format='%(asctime)s - %(message)s')

def handle_client(secure_socket, address):
    logging.info(f"Secure connection established with {address}")
    print(f"[Server] Secure connection with {address}")

    try:
        while True:
            message = secure_socket.recv(1024).decode()
            if not message:
                break
            print(f"[Server] Received: {message}")
            logging.info(f"Received from client: {message}")

            # Echoing the message back
            response = f"Server received (SSL): {message}"
            secure_socket.send(response.encode())
            logging.info(f"Sent to client: {response}")

    except Exception as e:
        print(f"[Server] Error: {str(e)}")
        logging.error(f"Error: {str(e)}")

    secure_socket.close()
    logging.info(f"Secure connection closed with {address}")
    print(f"[Server] Secure connection closed with {address}")

def start_server():
    print("[Server] Starting Secure SSL Server...")
    logging.info("SSL Server started.")

    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
```

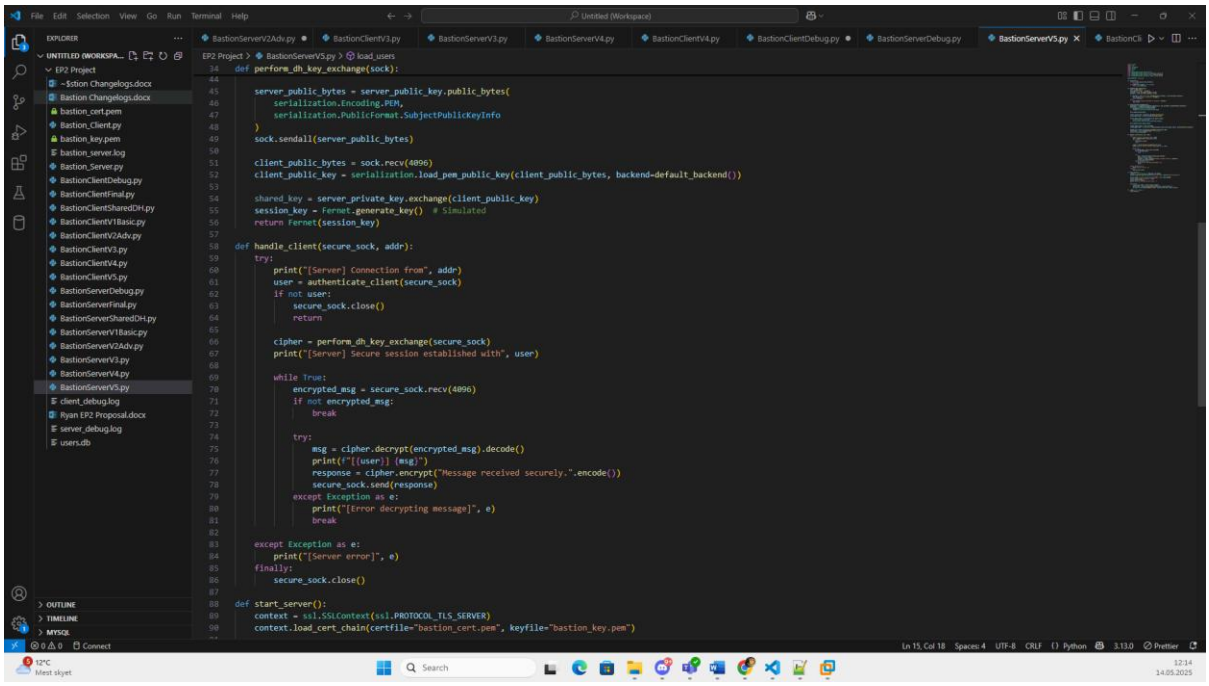


```python
def start_server():

    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile="bastion_cert.pem", keyfile="bastion_key.pem")

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 4443))
    server_socket.listen(5)
    print("[Server] Listening on port 4443 (SSL)...")

    while True:
        client_socket, address = server_socket.accept()
        secure_socket = context.wrap_socket(client_socket, server_side=True)
        thread = threading.Thread(target=handle_client, args=(secure_socket, address))
        thread.start()

if __name__ == "__main__":
    start_server()
```

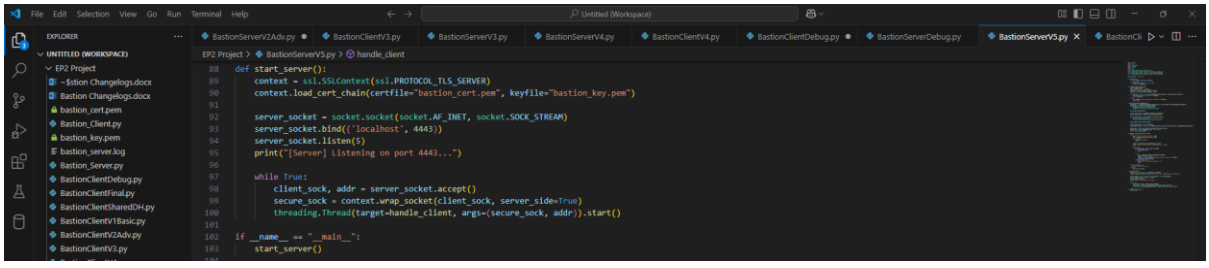
Bastion Server V5 (BSV5)



```python
import socket
import ssl
import threading
import bcrypt
import json
import os
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization

USER_DATABASE = "users.db"

def load_users():
    if not os.path.exists(USER_DATABASE):
        return {}
    with open(USER_DATABASE, "r") as file:
        return json.load(file)

def authenticate_client(sock):
    users = load_users()
    sock.sendall("LOGIN: ".encode())
    username = sock.recv(1024).decode().strip()
    password = sock.recv(1024).decode().strip()

    if username in users and bcrypt.checkpw(password.encode(), users[username].encode()):
        sock.sendall("Login successful.".encode())
        return username
    else:
        sock.sendall("Invalid username or password.".encode())
        return None

def perform_dh_key_exchange(sock):
    parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())
    param_bytes = parameters.parameter_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.ParameterFormat.PKCS3
    )
    sock.sendall(param_bytes)

    server_private_key = parameters.generate_private_key()
    server_public_key = server_private_key.public_key()

    server_public_bytes = server_public_key.public_bytes(
        serialization.Encoding.PEM,
        serialization.PublicFormat.SubjectPublicKeyInfo
    )
```

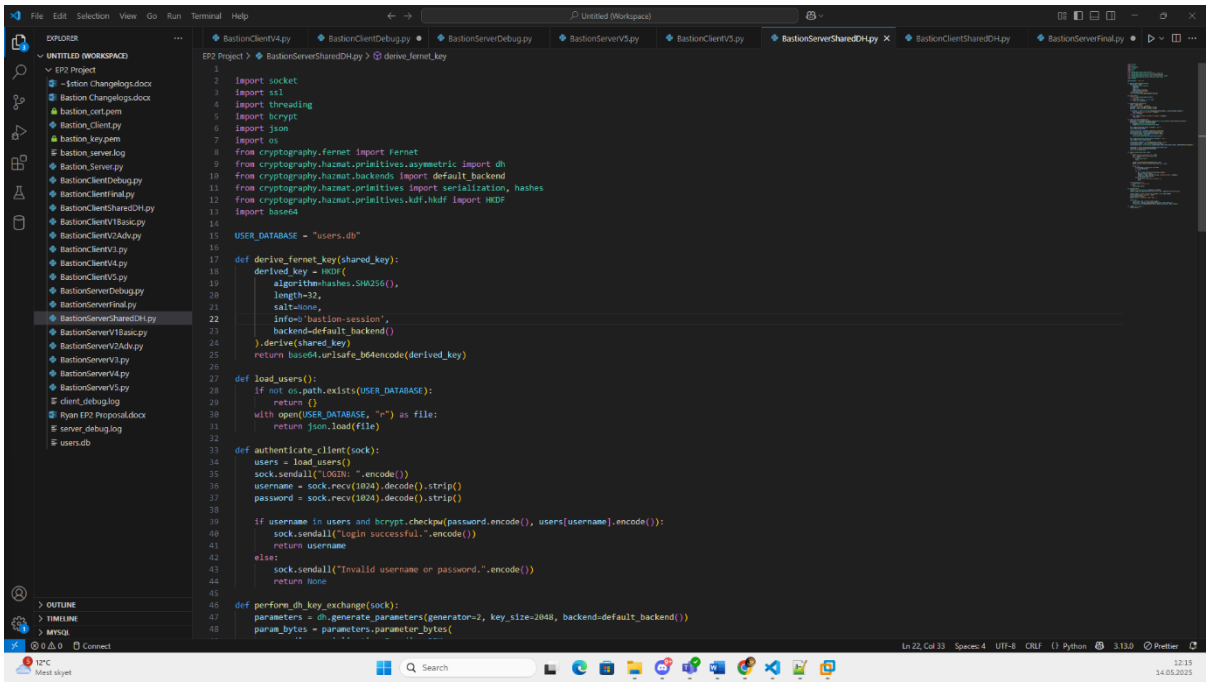


```python
def perform_dh_key_exchange(sock):

    server_public_bytes = server_public_key.public_bytes(
        serialization.Encoding.PEM,
        serialization.PublicFormat.SubjectPublicKeyInfo
    )
    sock.sendall(server_public_bytes)

    client_public_bytes = sock.recv(4096)
    client_public_key = serialization.load_pem_public_key(client_public_bytes, backend=default_backend())

    shared_key = server_private_key.exchange(client_public_key)
    session_key = Fernet.generate_key()  # Simulated
    return Fernet(session_key)

def handle_client(secure_sock, addr):
    try:
        print("[Server] Connection from", addr)
        user = authenticate_client(secure_sock)
        if not user:
            secure_sock.close()
            return

        cipher = perform_dh_key_exchange(secure_sock)
        print("[Server] Secure session established with", user)

        while True:
            encrypted_msg = secure_sock.recv(4096)
            if not encrypted_msg:
                break

            try:
                msg = cipher.decrypt(encrypted_msg).decode()
                print(f"[{user}] {msg}")
                response = cipher.encrypt("Message received securely.".encode())
                secure_sock.send(response)
            except Exception as e:
                print("[Error decrypting message]", e)
                break

    except Exception as e:
        print("[Server error]", e)
    finally:
        secure_sock.close()

def start_server():
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile="bastion_cert.pem", keyfile="bastion_key.pem")
```

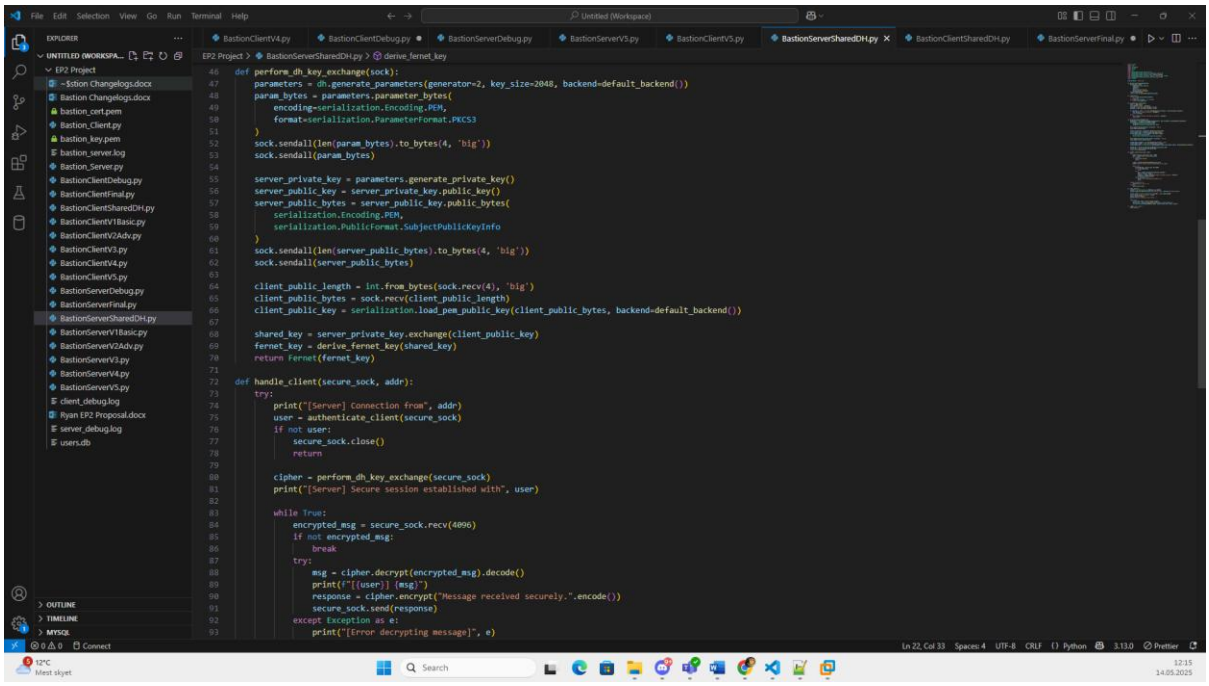


```python
def start_server():
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile="bastion_cert.pem", keyfile="bastion_key.pem")

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 4443))
    server_socket.listen(5)
    print("[Server] Listening on port 4443...")

    while True:
        client_sock, addr = server_socket.accept()
        secure_sock = context.wrap_socket(client_sock, server_side=True)
        threading.Thread(target=handle_client, args=(secure_sock, addr)).start()

if __name__ == "__main__":
    start_server()
```

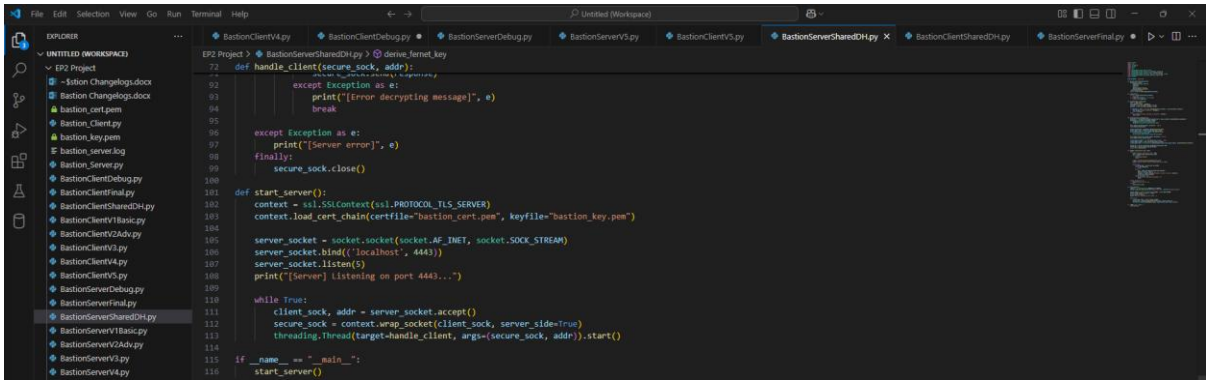
Bastion Server V6 (BSV6)



```python
import socket
import ssl
import threading
import bcrypt
import json
import os
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import base64

USER_DATABASE = "users.db"

def derive_fernet_key(shared_key):
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'bastion-session',
        backend=default_backend()
    ).derive(shared_key)
    return base64.urlsafe_b64encode(derived_key)

def load_users():
    if not os.path.exists(USER_DATABASE):
        return {}
    with open(USER_DATABASE, "r") as file:
        return json.load(file)

def authenticate_client(sock):
    users = load_users()
    sock.sendall("LOGIN: ".encode())
    username = sock.recv(1024).decode().strip()
    password = sock.recv(1024).decode().strip()

    if username in users and bcrypt.checkpw(password.encode(), users[username].encode()):
        sock.sendall("Login successful.".encode())
        return username
    else:
        sock.sendall("Invalid username or password.".encode())
        return None

def perform_dh_key_exchange(sock):
    parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())
    param_bytes = parameters.parameter_bytes(
```

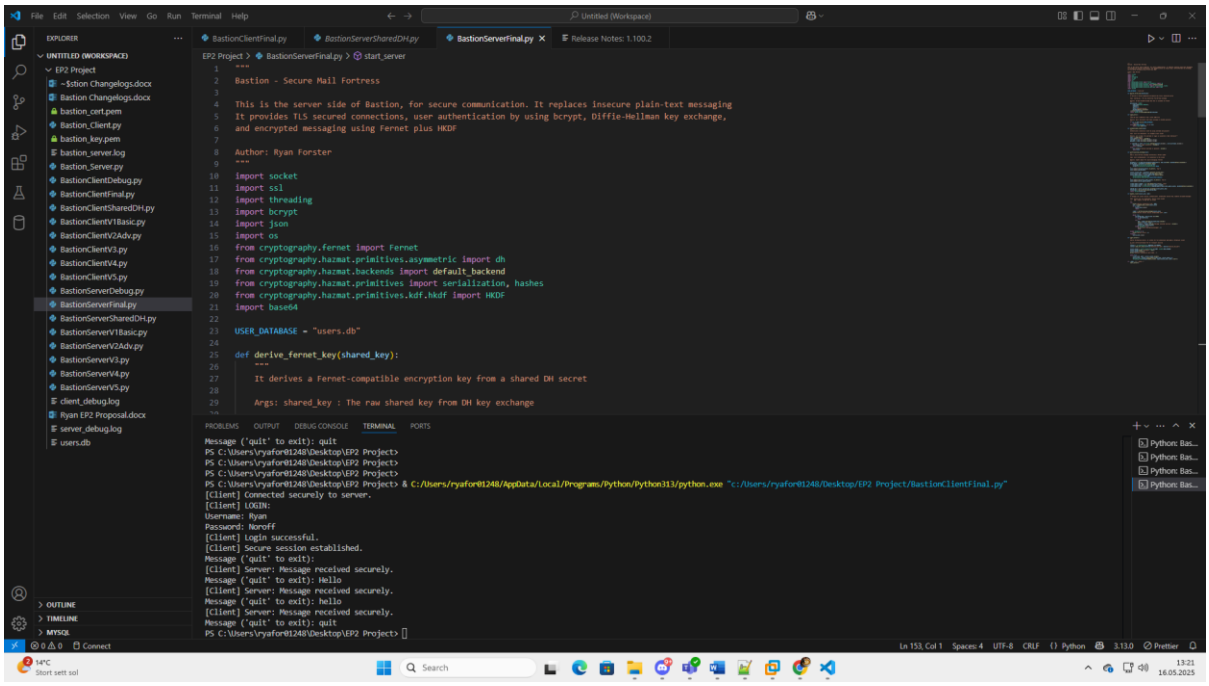


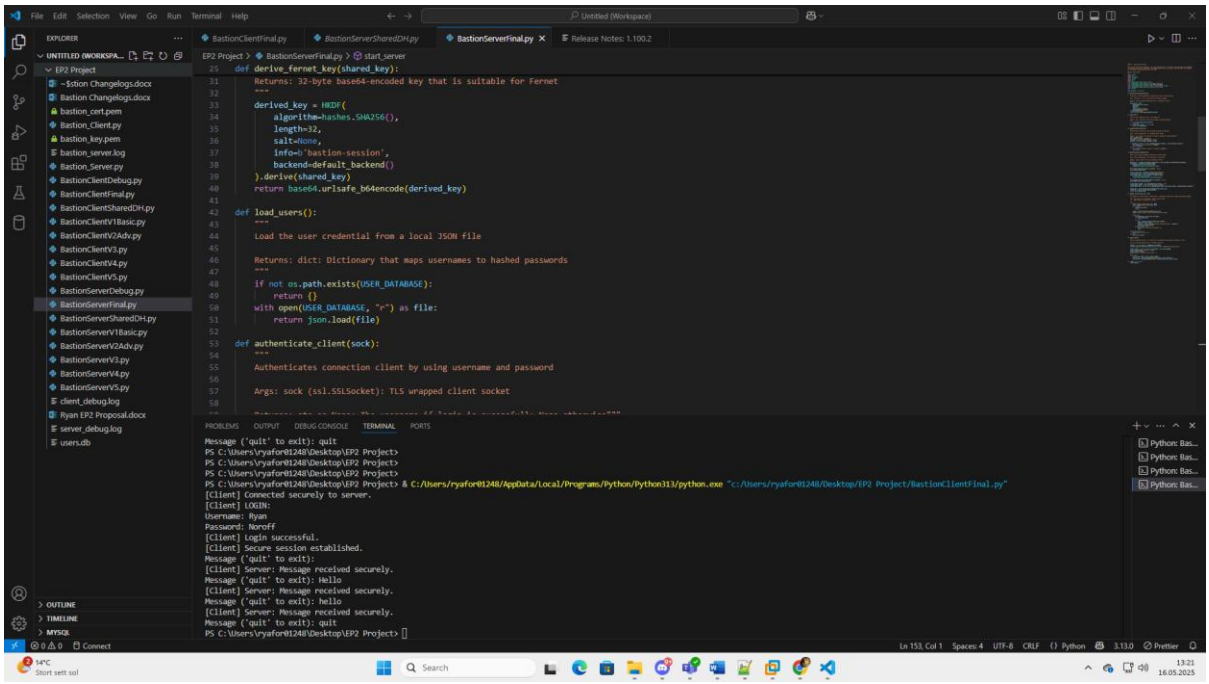
```python
def perform_dh_key_exchange(sock):
    parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())
    param_bytes = parameters.parameter_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.ParameterFormat.PKCS3
    )
    sock.sendall(len(param_bytes).to_bytes(4, 'big'))
    sock.sendall(param_bytes)

    server_private_key = parameters.generate_private_key()
    server_public_key = server_private_key.public_key()
    server_public_bytes = server_public_key.public_bytes(
        serialization.Encoding.PEM,
        serialization.PublicFormat.SubjectPublicKeyInfo
    )
    sock.sendall(len(server_public_bytes).to_bytes(4, 'big'))
    sock.sendall(server_public_bytes)

    client_public_length = int.from_bytes(sock.recv(4), 'big')
    client_public_bytes = sock.recv(client_public_length)
    client_public_key = serialization.load_pem_public_key(client_public_bytes, backend=default_backend())

    shared_key = server_private_key.exchange(client_public_key)
    fernet_key = derive_fernet_key(shared_key)
    return Fernet(fernet_key)

def handle_client(secure_sock, addr):
    try:
        print("[Server] Connection from", addr)
        user = authenticate_client(secure_sock)
        if not user:
            secure_sock.close()
            return

        cipher = perform_dh_key_exchange(secure_sock)
        print("[Server] Secure session established with", user)

        while True:
            encrypted_msg = secure_sock.recv(4096)
            if not encrypted_msg:
                break
            try:
                msg = cipher.decrypt(encrypted_msg).decode()
                print(f"[{user}] {msg}")
                response = cipher.encrypt("Message received securely.".encode())
                secure_sock.send(response)
            except Exception as e:
                print("[Error decrypting message]", e)
```

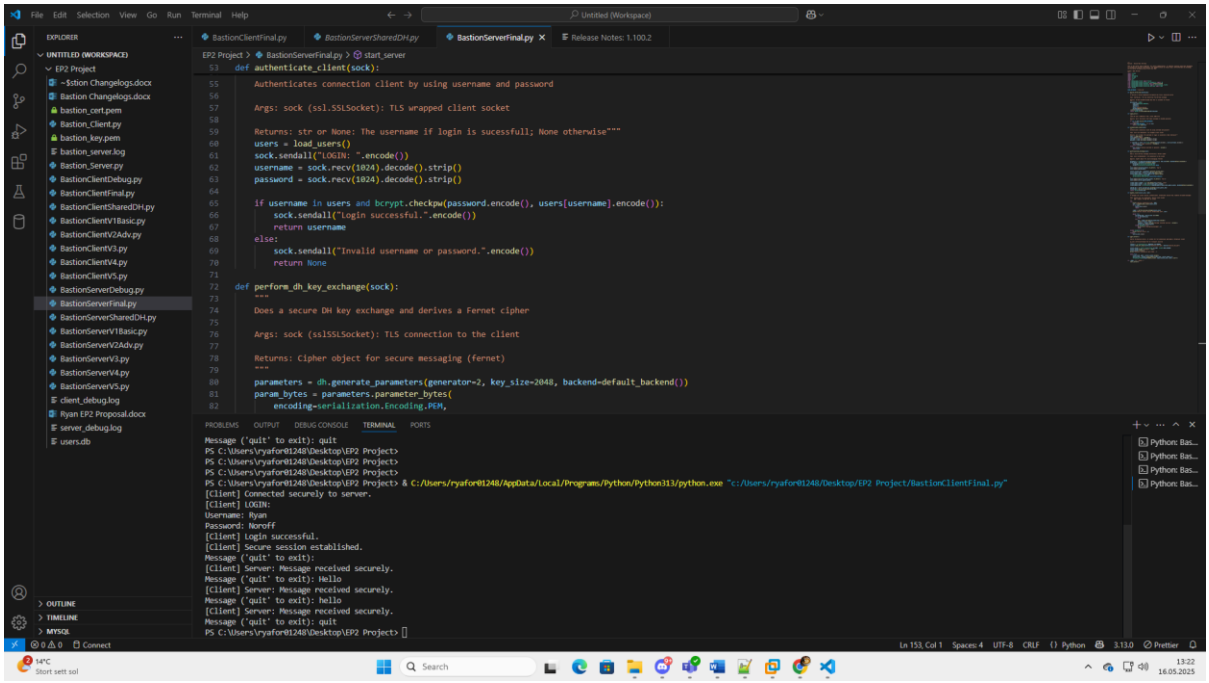


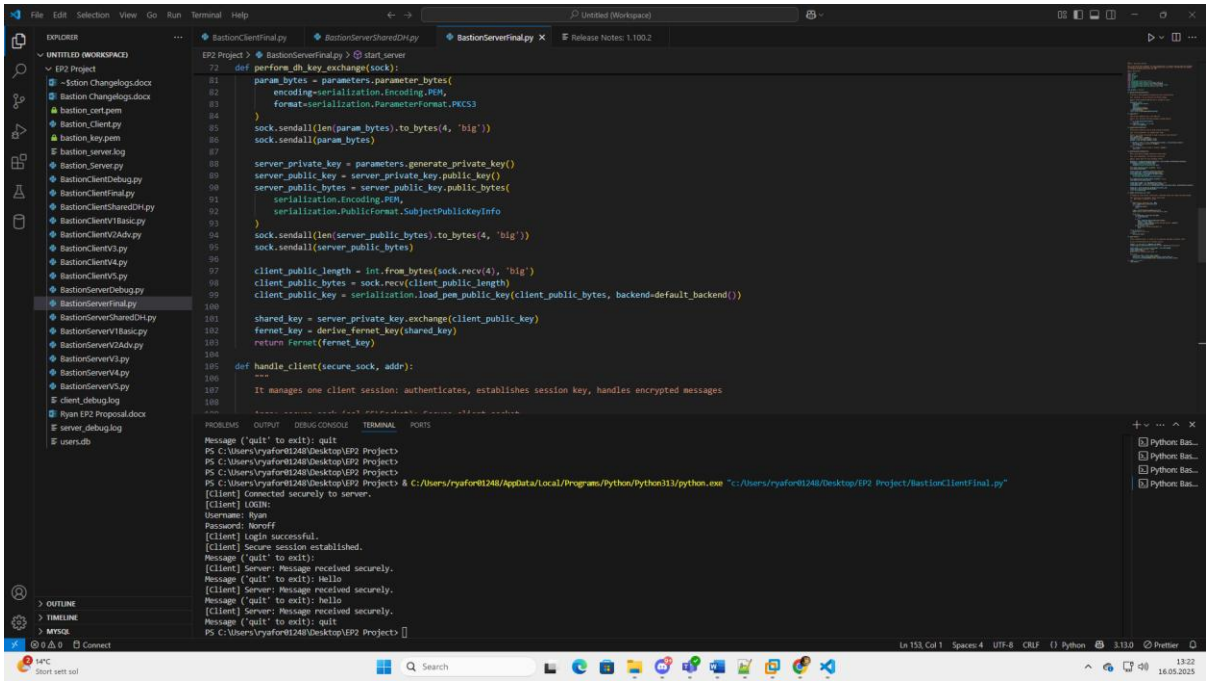
```python
def handle_client(secure_sock, addr):
                print("[Error decrypting message]", e)
                break

    except Exception as e:
        print("[Server error]", e)
    finally:
        secure_sock.close()

def start_server():
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile="bastion_cert.pem", keyfile="bastion_key.pem")

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 4443))
    server_socket.listen(5)
    print("[Server] Listening on port 4443...")

    while True:
        client_sock, addr = server_socket.accept()
        secure_sock = context.wrap_socket(client_sock, server_side=True)
        threading.Thread(target=handle_client, args=(secure_sock, addr)).start()

if __name__ == "__main__":
    start_server()
```

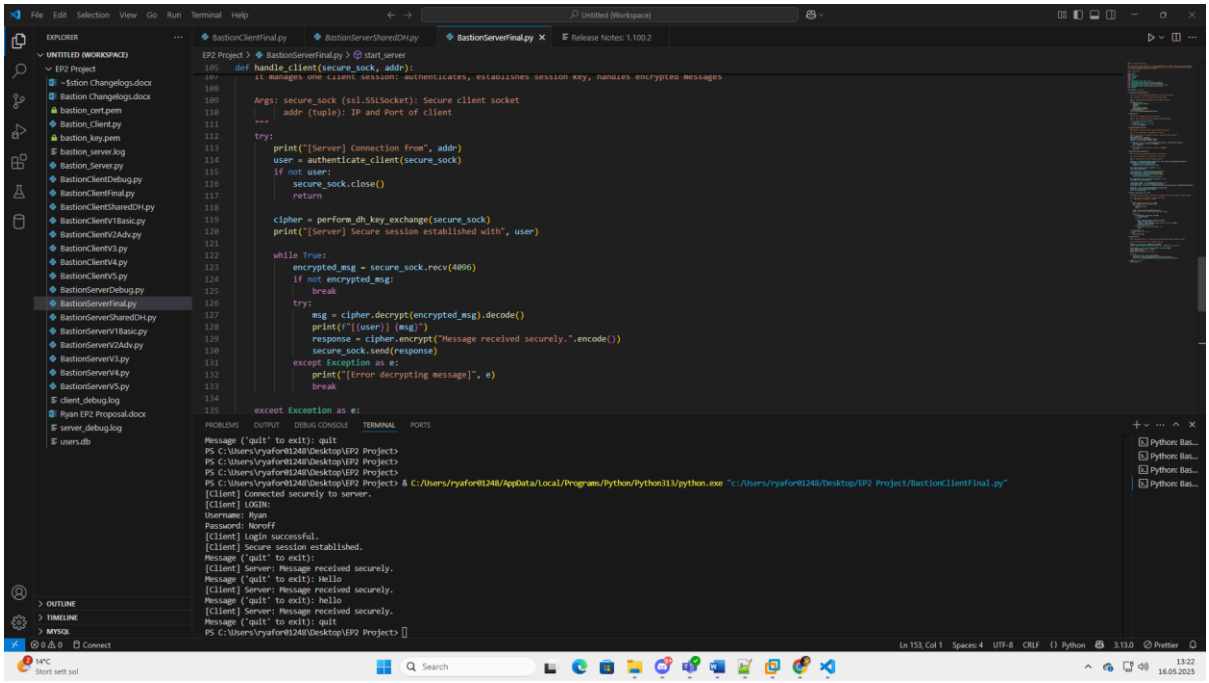
Bastion Server V7 (BSV7)

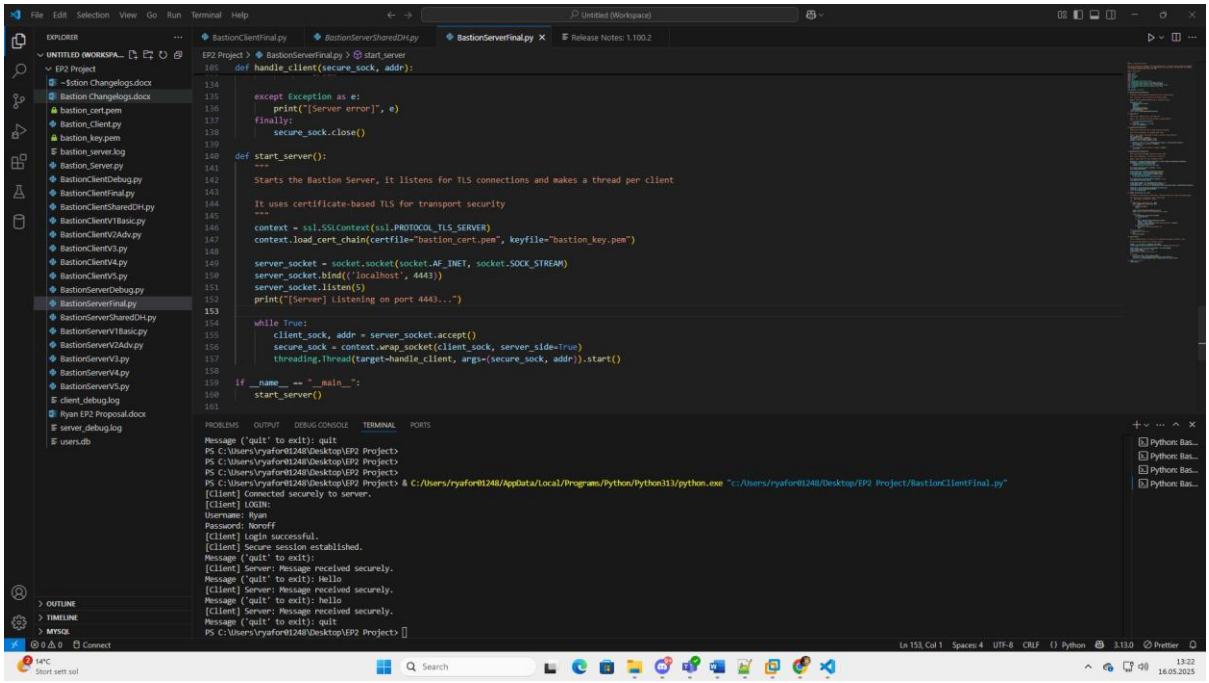




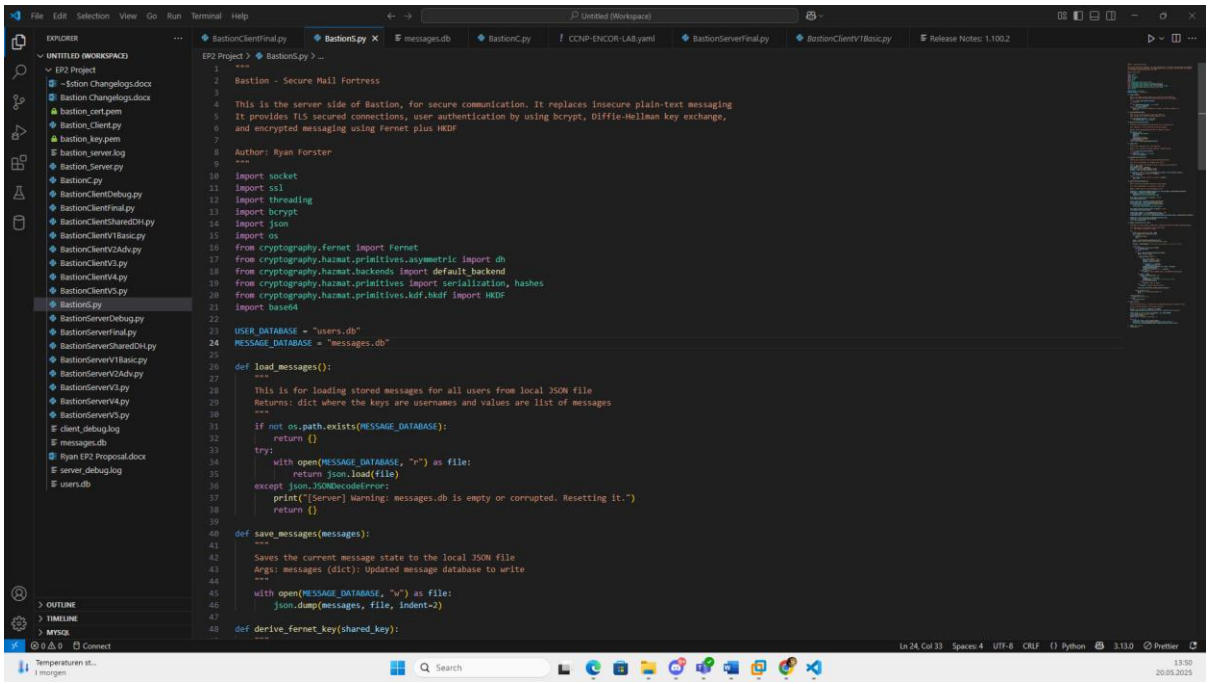

File Edit Selection View Go Run Terminal Help
Untitled (Workspace)
EXPLORER
UNTITLED (WORKSPACE)
EP2 Project
~$stion Changelogs.docx
Bastion Changelogs.docx
bastion_cert.pem
Bastion_Client.py
bastion_key.pem
bastion_server.log
Bastion_Server.py
BastionClientDebug.py
BastionClientFinal.py
BastionClientSharedDH.py
BastionClientV1Basic.py
BastionClientV2Adv.py
BastionClientV3.py
BastionClientV4.py
BastionClientV5.py
BastionServerDebug.py
BastionServerFinal.py
BastionServerSharedDH.py
BastionServerV1Basic.py
BastionServerV2Adv.py
BastionServerV3.py
BastionServerV4.py
BastionServerV5.py
client_debug.log
Ryan EP2 Proposal.docx
server_debug.log
users.db
BastionClientFinal.py
BastionServerSharedDH.py
BastionServerFinal.py
Release Notes: 1.100.2
EP2 Project > BastionServerFinal.py > start_server
def authenticate_client(sock):
Authenticates connection client by using username and password
Args: sock (ssl.SSLSocket): TLS wrapped client socket
Returns: str or None: The username if login is sucessfull; None otherwise"""
users = load_users()
sock.sendall("LOGIN: ".encode())
username = sock.recv(1024).decode().strip()
password = sock.recv(1024).decode().strip()
if username in users and bcrypt.checkpw(password.encode(), users[username].encode()):
sock.sendall("Login sucessful.".encode())
return username
else:
sock.sendall("Invalid username or password.".encode())
return None
def perform_dh_key_exchange(sock):
"""
Does a secure DH key exchange and derives a Fernet cipher
Args: sock (ssl.SSLSocket): TLS connection to the client
Returns: Cipher object for secure messaging (fernet)
"""
parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())
param_bytes = parameters.parameter_bytes(
encoding=serialization.Encoding.PEM,
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project> & C:/Users/ryafor01248/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ryafor01248/Desktop/EP2 Project/BastionClientFinal.py"
[Client] Connected securely to server.
[Client] LOGIN:
Username: Ryan
Password: Noroff
[Client] Login successful.
[Client] Secure session established.
Message ('quit' to exit):
[Client] Server: Message received securely.
Message ('quit' to exit): Hello
[Client] Server: Message received securely.
Message ('quit' to exit): hello
[Client] Server: Message received securely.
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>
Python: Bas...
OUTLINE
TIMELINE
MYSQL
Connect
Ln 153, Col 1 Spaces: 4 UTF-8 CRLF Python 3.13.0 Prettier
14°C Stort sett sol
Search
13:22 16.05.2025


File Edit Selection View Go Run Terminal Help
Untitled (Workspace)
EXPLORER
UNTITLED (WORKSPACE)
EP2 Project
~$stion Changelogs.docx
Bastion Changelogs.docx
bastion_cert.pem
Bastion_Client.py
bastion_key.pem
bastion_server.log
Bastion_Server.py
BastionClientDebug.py
BastionClientFinal.py
BastionClientSharedDH.py
BastionClientV1Basic.py
BastionClientV2Adv.py
BastionClientV3.py
BastionClientV4.py
BastionClientV5.py
BastionServerDebug.py
BastionServerFinal.py
BastionServerSharedDH.py
BastionServerV1Basic.py
BastionServerV2Adv.py
BastionServerV3.py
BastionServerV4.py
BastionServerV5.py
client_debug.log
Ryan EP2 Proposal.docx
server_debug.log
users.db
BastionClientFinal.py
BastionServerSharedDH.py
BastionServerFinal.py
Release Notes: 1.100.2
EP2 Project > BastionServerFinal.py > start_server
def perform_dh_key_exchange(sock):
param_bytes = parameters.parameter_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.ParameterFormat.PKCS3
)
sock.sendall(len(param_bytes).to_bytes(4, 'big'))
sock.sendall(param_bytes)
server_private_key = parameters.generate_private_key()
server_public_key = server_private_key.public_key()
server_public_bytes = server_public_key.public_bytes(
serialization.Encoding.PEM,
serialization.PublicFormat.SubjectPublicKeyInfo
)
sock.sendall(len(server_public_bytes).to_bytes(4, 'big'))
sock.sendall(server_public_bytes)
client_public_length = int.from_bytes(sock.recv(4), 'big')
client_public_bytes = sock.recv(client_public_length)
client_public_key = serialization.load_pem_public_key(client_public_bytes, backend=default_backend())
shared_key = server_private_key.exchange(client_public_key)
fernet_key = derive_fernet_key(shared_key)
return Fernet(fernet_key)
def handle_client(secure_sock, addr):
"""
It manages one client session: authenticates, establishes session key, handles encrypted messages
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project> & C:/Users/ryafor01248/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ryafor01248/Desktop/EP2 Project/BastionClientFinal.py"
[Client] Connected securely to server.
[Client] LOGIN:
Username: Ryan
Password: Noroff
[Client] Login successful.
[Client] Secure session established.
Message ('quit' to exit):
[Client] Server: Message received securely.
Message ('quit' to exit): Hello
[Client] Server: Message received securely.
Message ('quit' to exit): hello
[Client] Server: Message received securely.
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>
Python: Bas...
OUTLINE
TIMELINE
MYSQL
Connect
Ln 153, Col 1 Spaces: 4 UTF-8 CRLF Python 3.13.0 Prettier
14°C Stort sett sol
Search
13:22 16.05.2025

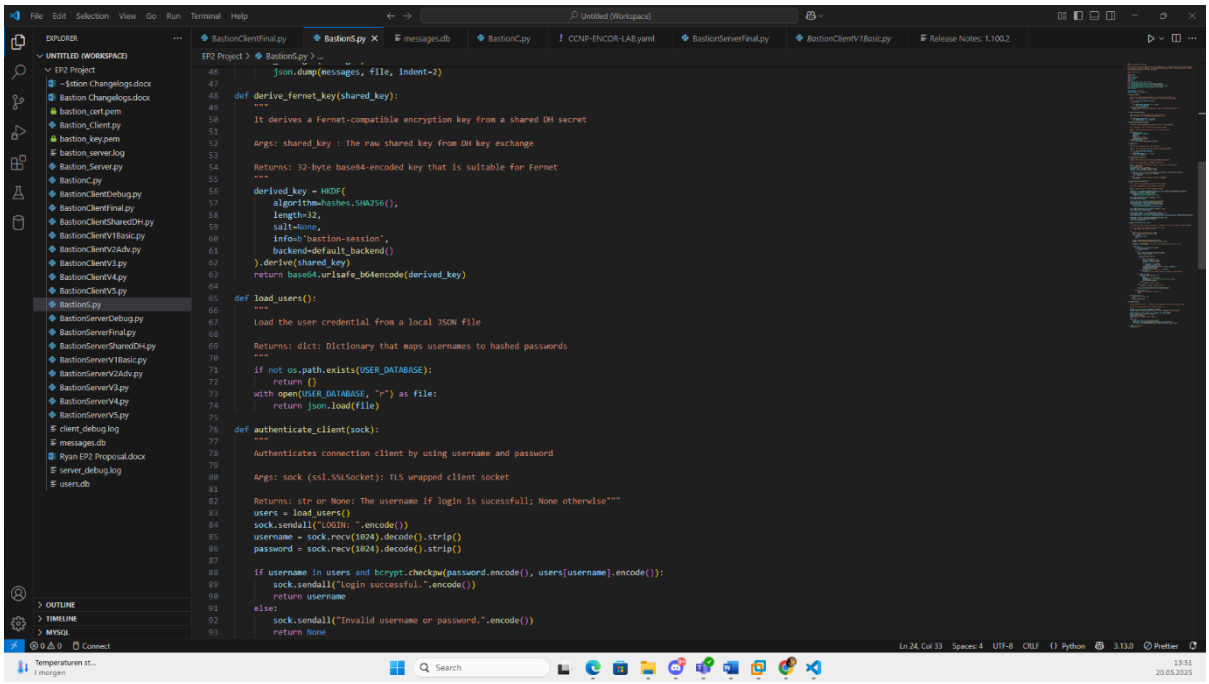

def handle_client(secure_sock, addr):
"""
Handles one client session: authenticates, establishes session key, handles encrypted messages
Args: secure_sock (ssl.SSLSocket): Secure client socket
addr (tuple): IP and Port of client
"""
try:
print("[Server] Connection from", addr)
user = authenticate_client(secure_sock)
if not user:
secure_sock.close()
return
cipher = perform_dh_key_exchange(secure_sock)
print("[Server] Secure session established with", user)
while True:
encrypted_msg = secure_sock.recv(4096)
if not encrypted_msg:
break
try:
msg = cipher.decrypt(encrypted_msg).decode()
print(f"[{user}] {msg}")
response = cipher.encrypt("Message received securely.".encode())
secure_sock.send(response)
except Exception as e:
print("[Error decrypting message]", e)
break
except Exception as e:
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project> & C:/Users/ryafor01248/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ryafor01248/Desktop/EP2 Project/BastionClientFinal.py"
[Client] Connected securely to server.
[Client] LOGIN:
Username: Ryan
Password: Noroff
[Client] Login successful.
[Client] Secure session established.
Message ('quit' to exit):
[Client] Server: Message received securely.
Message ('quit' to exit): Hello
[Client] Server: Message received securely.
Message ('quit' to exit): hello
[Client] Server: Message received securely.
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>


except Exception as e:
print("[Server error]", e)
finally:
secure_sock.close()
def start_server():
"""
Starts the Bastion Server, it listens for TLS connections and makes a thread per client
It uses certificate-based TLS for transport security
"""
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain(certfile="bastion_cert.pem", keyfile="bastion_key.pem")
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('localhost', 4443))
server_socket.listen(5)
print("[Server] Listening on port 4443...")
while True:
client_sock, addr = server_socket.accept()
secure_sock = context.wrap_socket(client_sock, server_side=True)
threading.Thread(target=handle_client, args=(secure_sock, addr)).start()
if __name__ == "__main__":
start_server()
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project> & C:/Users/ryafor01248/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ryafor01248/Desktop/EP2 Project/BastionClientFinal.py"
[Client] Connected securely to server.
[Client] LOGIN:
Username: Ryan
Password: Noroff
[Client] Login successful.
[Client] Secure session established.
Message ('quit' to exit):
[Client] Server: Message received securely.
Message ('quit' to exit): Hello
[Client] Server: Message received securely.
Message ('quit' to exit): hello
[Client] Server: Message received securely.
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>

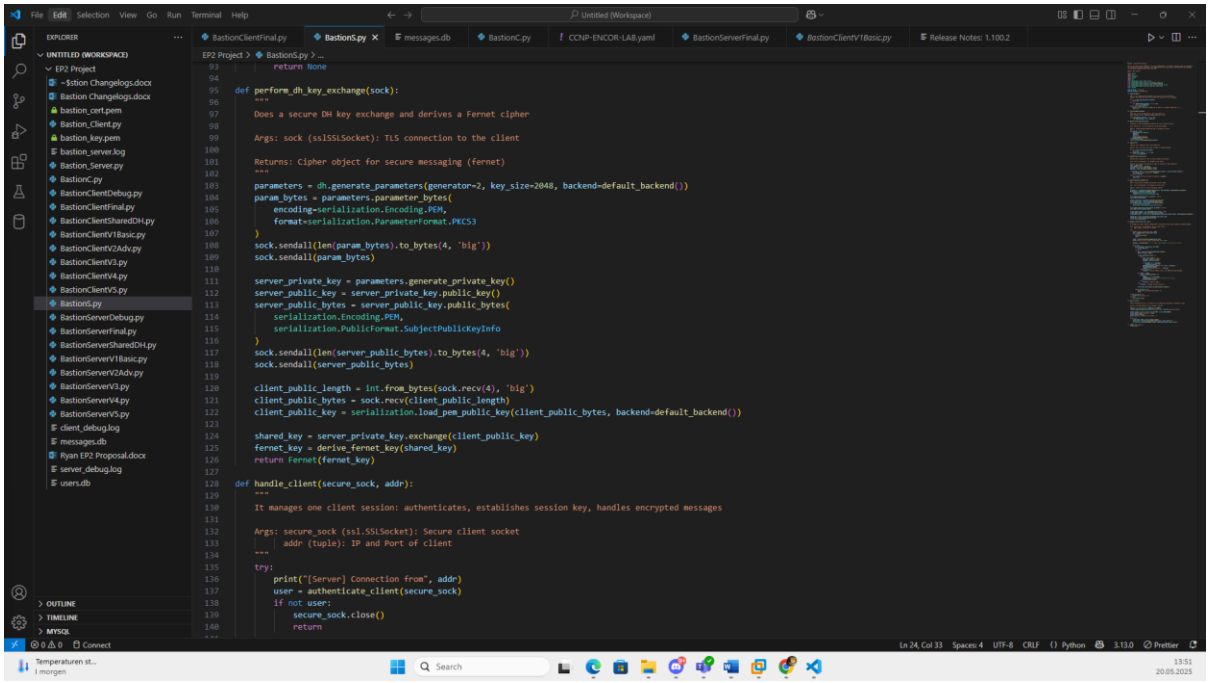
Bastion Server V8 (BSV8)



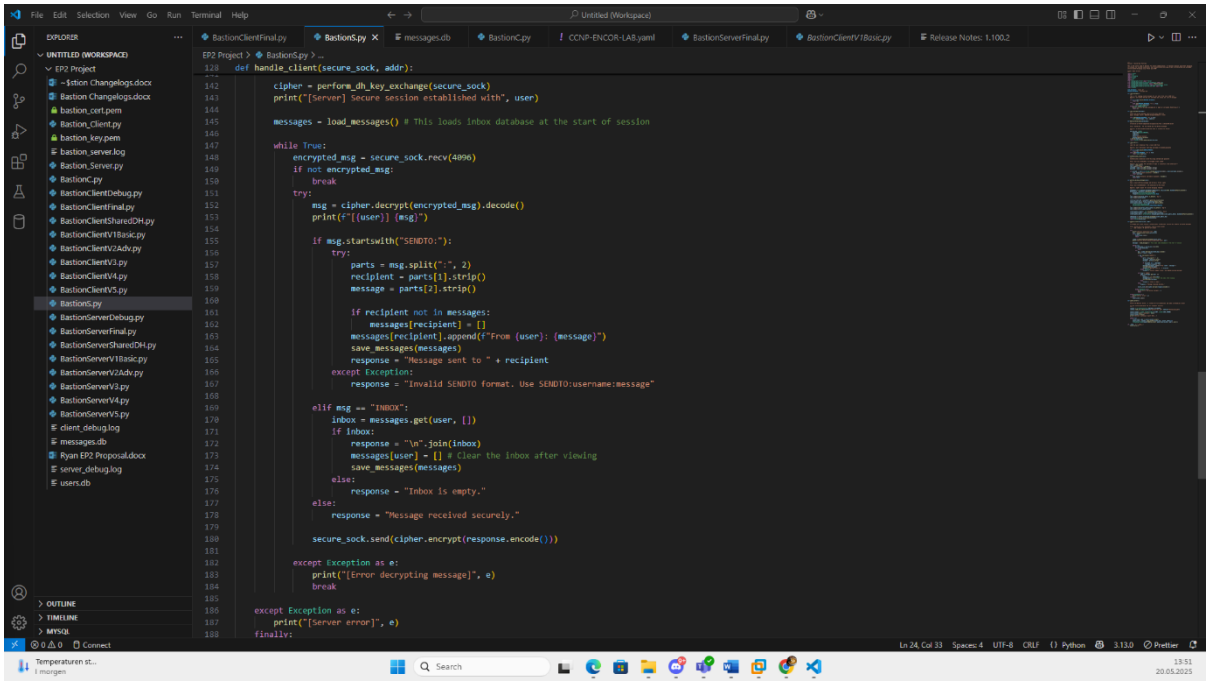
```python
"""
Bastion - Secure Mail Fortress

This is the server side of Bastion, for secure communication. It replaces insecure plain-text messaging
It provides TLS secured connections, user authentication by using bcrypt, Diffie-Hellman key exchange,
and encrypted messaging using Fernet plus HKDF

Author: Ryan Forster
"""
import socket
import ssl
import threading
import bcrypt
import json
import os
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import base64

USER_DATABASE = "users.db"
MESSAGE_DATABASE = "messages.db"

def load_messages():
    """
    This is for loading stored messages for all users from local JSON file
    Returns: dict where the keys are usernames and values are list of messages
    """
    if not os.path.exists(MESSAGE_DATABASE):
        return {}
    try:
        with open(MESSAGE_DATABASE, "r") as file:
            return json.load(file)
    except json.JSONDecodeError:
        print("[Server] Warning: messages.db is empty or corrupted. Resetting it.")
        return {}

def save_messages(messages):
    """
    Saves the current message state to the local JSON file
    Args: messages (dict): Updated message database to write
    """
    with open(MESSAGE_DATABASE, "w") as file:
        json.dump(messages, file, indent=2)

def derive_fernet_key(shared_key):
    """
```



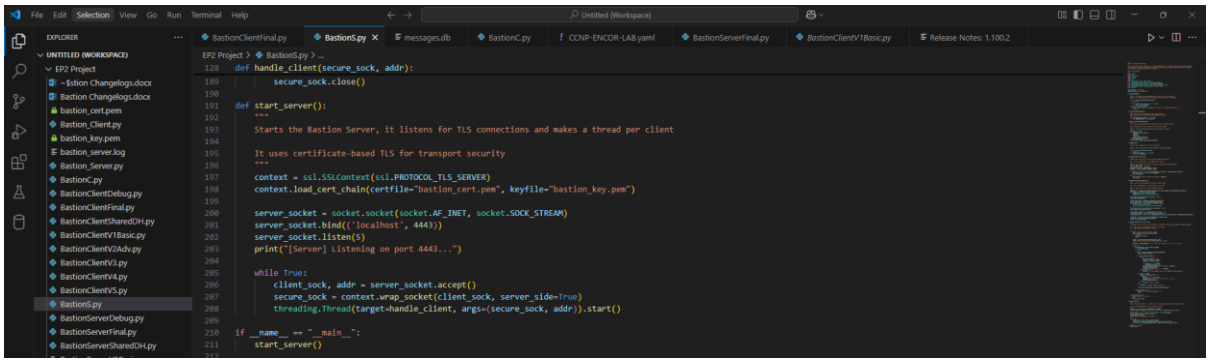
```python
        json.dump(messages, file, indent=2)

def derive_fernet_key(shared_key):
    """
    It derives a Fernet-compatible encryption key from a shared DH secret

    Args: shared_key : The raw shared key from DH key exchange

    Returns: 32-byte base64-encoded key that is suitable for Fernet
    """
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'bastion-session',
        backend=default_backend()
    ).derive(shared_key)
    return base64.urlsafe_b64encode(derived_key)

def load_users():
    """
    Load the user credential from a local JSON file

    Returns: dict: Dictionary that maps usernames to hashed passwords
    """
    if not os.path.exists(USER_DATABASE):
        return {}
    with open(USER_DATABASE, "r") as file:
        return json.load(file)

def authenticate_client(sock):
    """
    Authenticates connection client by using username and password

    Args: sock (ssl.SSLSocket): TLS wrapped client socket

    Returns: str or None: The username if login is sucessfull; None otherwise"""
    users = load_users()
    sock.sendall("LOGIN: ".encode())
    username = sock.recv(1024).decode().strip()
    password = sock.recv(1024).decode().strip()

    if username in users and bcrypt.checkpw(password.encode(), users[username].encode()):
        sock.sendall("Login successful.".encode())
        return username
    else:
        sock.sendall("Invalid username or password.".encode())
        return None
```



```python
        return None

def perform_dh_key_exchange(sock):
    """
    Does a secure DH key exchange and derives a Fernet cipher

    Args: sock (sslSSLSocket): TLS connection to the client

    Returns: Cipher object for secure messaging (fernet)
    """
    parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())
    param_bytes = parameters.parameter_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.ParameterFormat.PKCS3
    )
    sock.sendall(len(param_bytes).to_bytes(4, 'big'))
    sock.sendall(param_bytes)

    server_private_key = parameters.generate_private_key()
    server_public_key = server_private_key.public_key()
    server_public_bytes = server_public_key.public_bytes(
        serialization.Encoding.PEM,
        serialization.PublicFormat.SubjectPublicKeyInfo
    )
    sock.sendall(len(server_public_bytes).to_bytes(4, 'big'))
    sock.sendall(server_public_bytes)

    client_public_length = int.from_bytes(sock.recv(4), 'big')
    client_public_bytes = sock.recv(client_public_length)
    client_public_key = serialization.load_pem_public_key(client_public_bytes, backend=default_backend())

    shared_key = server_private_key.exchange(client_public_key)
    fernet_key = derive_fernet_key(shared_key)
    return Fernet(fernet_key)

def handle_client(secure_sock, addr):
    """
    It manages one client session: authenticates, establishes session key, handles encrypted messages

    Args: secure_sock (ssl.SSLSocket): Secure client socket
          addr (tuple): IP and Port of client
    """
    try:
        print("[Server] Connection from", addr)
        user = authenticate_client(secure_sock)
        if not user:
            secure_sock.close()
            return
```

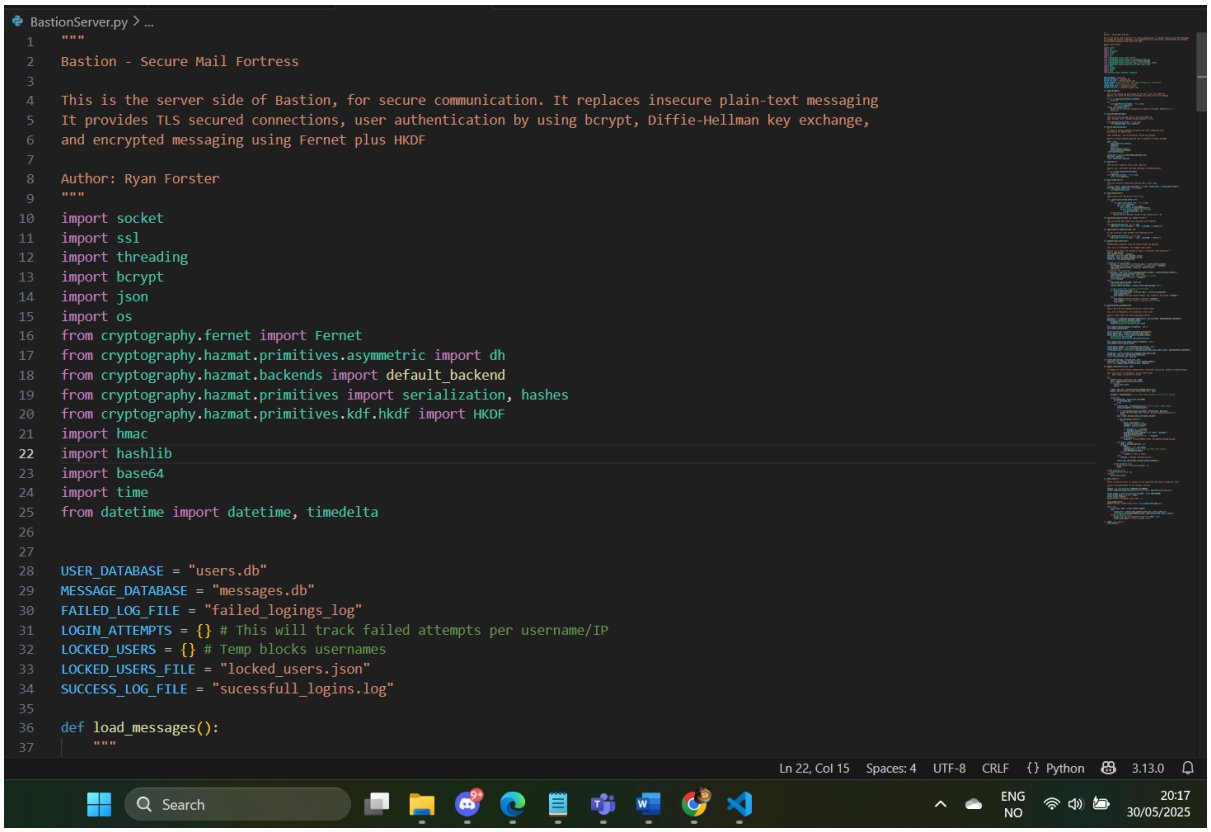


```python
def handle_client(secure_sock, addr):
        cipher = perform_dh_key_exchange(secure_sock)
        print("[Server] Secure session established with", user)

        messages = load_messages() # This loads inbox database at the start of session

        while True:
            encrypted_msg = secure_sock.recv(4096)
            if not encrypted_msg:
                break
            try:
                msg = cipher.decrypt(encrypted_msg).decode()
                print(f"[{user}] {msg}")

                if msg.startswith("SENDTO:"):
                    try:
                        parts = msg.split(":", 2)
                        recipient = parts[1].strip()
                        message = parts[2].strip()

                        if recipient not in messages:
                            messages[recipient] = []
                        messages[recipient].append(f"From {user}: {message}")
                        save_messages(messages)
                        response = "Message sent to " + recipient
                    except Exception:
                        response = "Invalid SENDTO format. Use SENDTO:username:message"

                elif msg == "INBOX":
                    inbox = messages.get(user, [])
                    if inbox:
                        response = "\n".join(inbox)
                        messages[user] = [] # Clear the inbox after viewing
                        save_messages(messages)
                    else:
                        response = "Inbox is empty."
                else:
                    response = "Message received securely."

                secure_sock.send(cipher.encrypt(response.encode()))

            except Exception as e:
                print("[Error decrypting message]", e)
                break

    except Exception as e:
        print("[Server error]", e)
    finally:
```

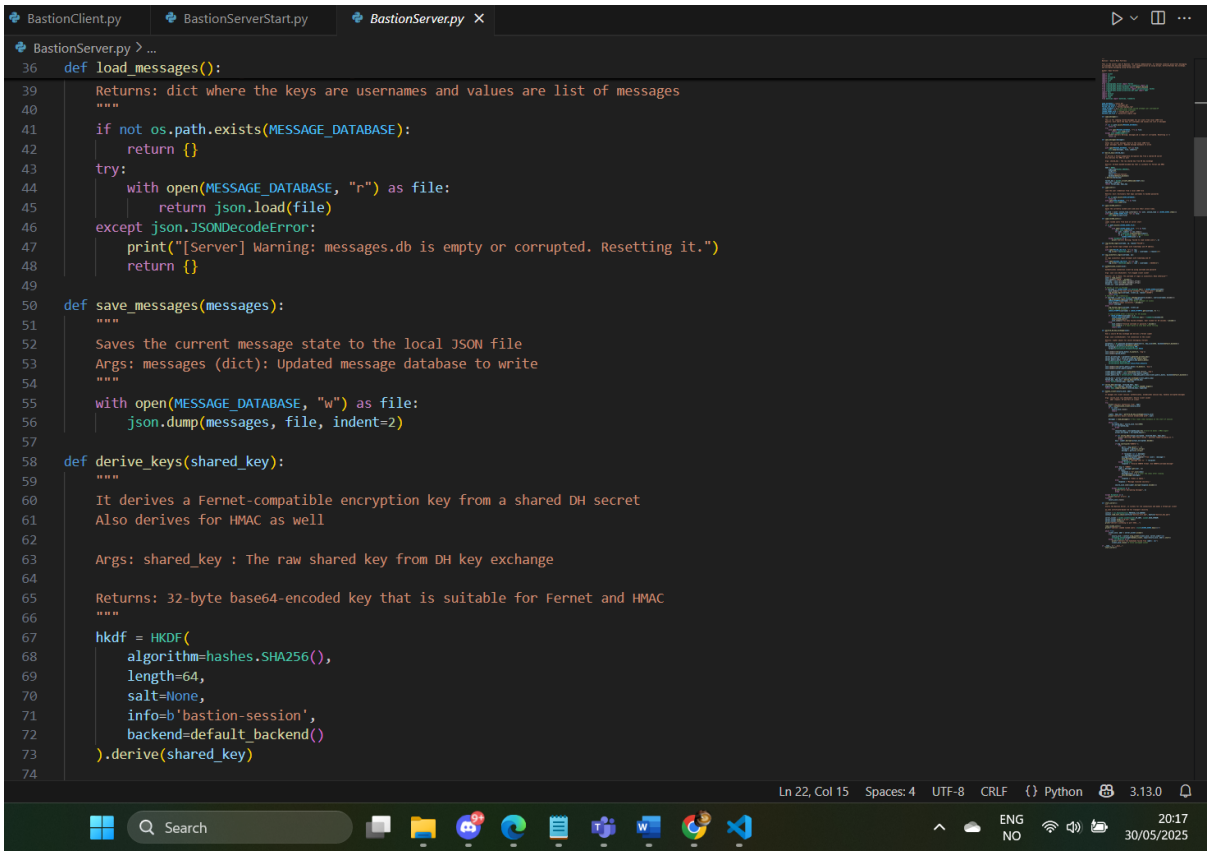


```python
        secure_sock.close()

def start_server():
    """
    Starts the Bastion Server, it listens for TLS connections and makes a thread per client

    It uses certificate-based TLS for transport security
    """
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile="bastion_cert.pem", keyfile="bastion_key.pem")

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('localhost', 4443))
    server_socket.listen(5)
    print("[Server] Listening on port 4443...")

    while True:
        client_sock, addr = server_socket.accept()
        secure_sock = context.wrap_socket(client_sock, server_side=True)
        threading.Thread(target=handle_client, args=(secure_sock, addr)).start()

if __name__ == "__main__":
    start_server()
```

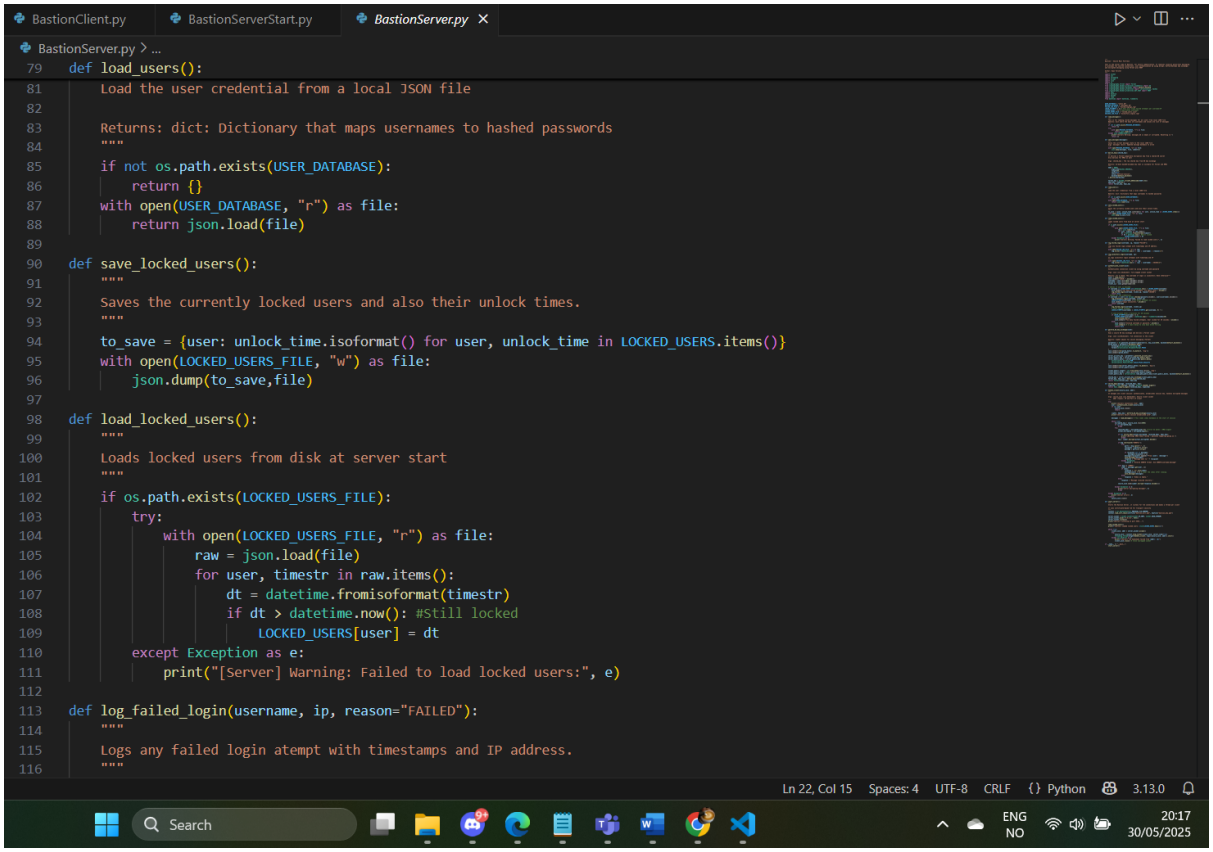
Bastion Server Final (BSF)

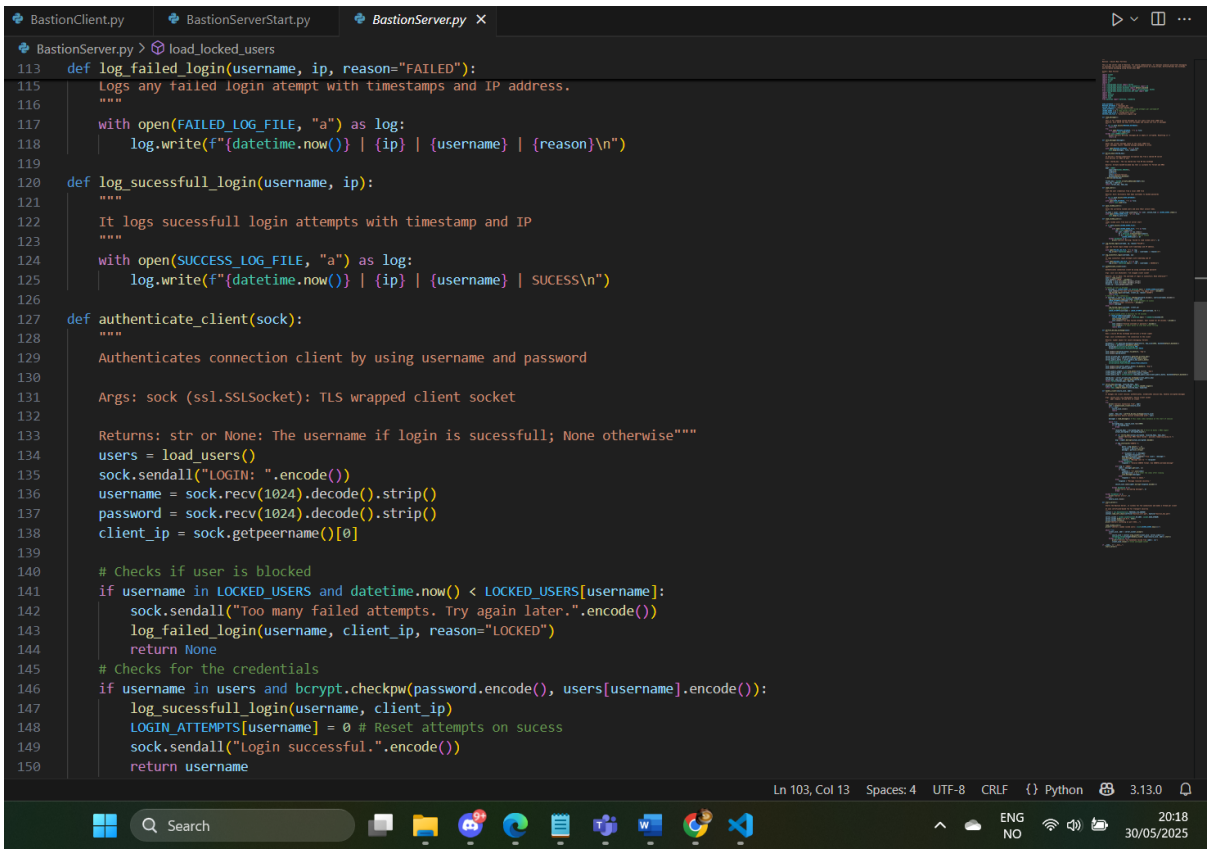


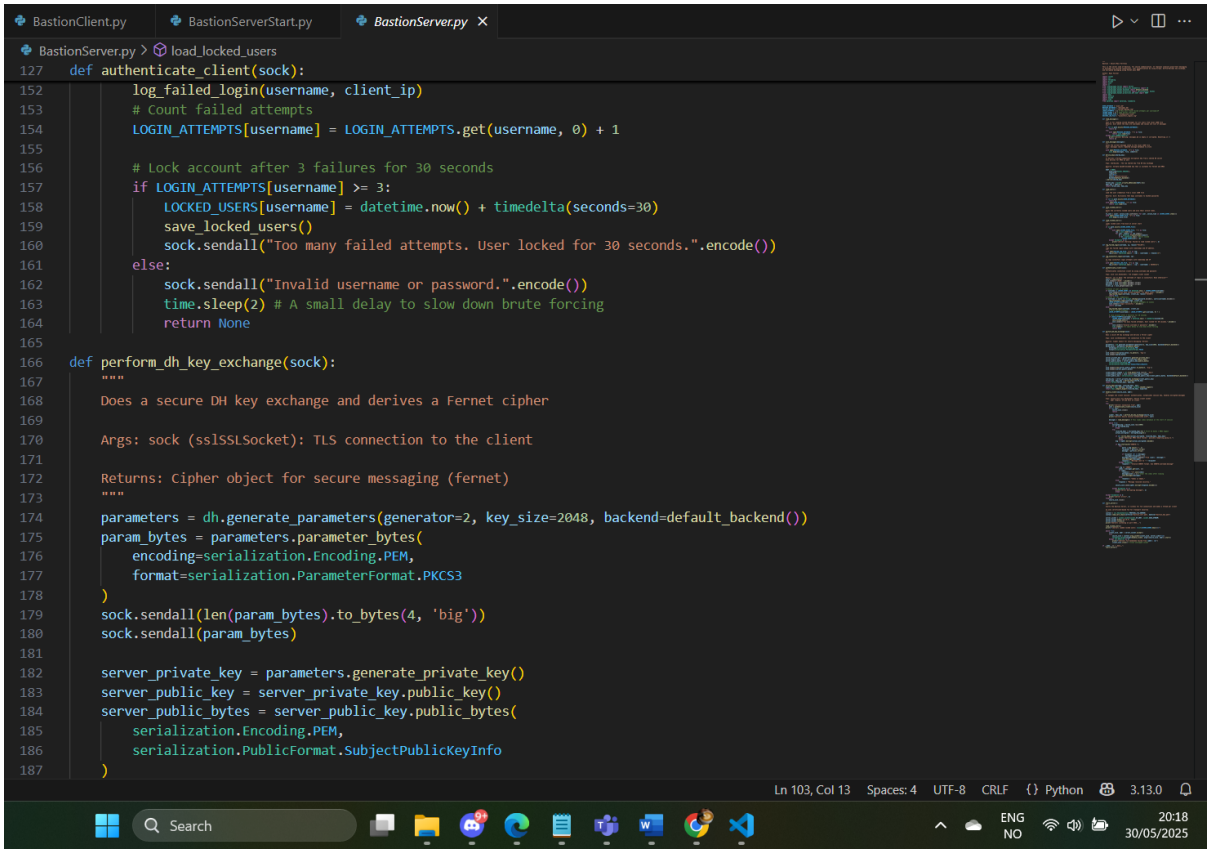
```python
"""
Bastion - Secure Mail Fortress

This is the server side of Bastion, for secure communication. It replaces insecure plain-text messaging
It provides TLS secured connections, user authentication by using bcrypt, Diffie-Hellman key exchange,
and encrypted messaging using Fernet plus HKDF

Author: Ryan Forster
"""
import socket
import ssl
import threading
import bcrypt
import json
import os
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import hmac
import hashlib
import base64
import time
from datetime import datetime, timedelta


USER_DATABASE = "users.db"
MESSAGE_DATABASE = "messages.db"
FAILED_LOG_FILE = "failed_logings_log"
LOGIN_ATTEMPTS = {} # This will track failed attempts per username/IP
LOCKED_USERS = {} # Temp blocks usernames
LOCKED_USERS_FILE = "locked_users.json"
SUCCESS_LOG_FILE = "sucessfull_logins.log"

def load_messages():
    """
```

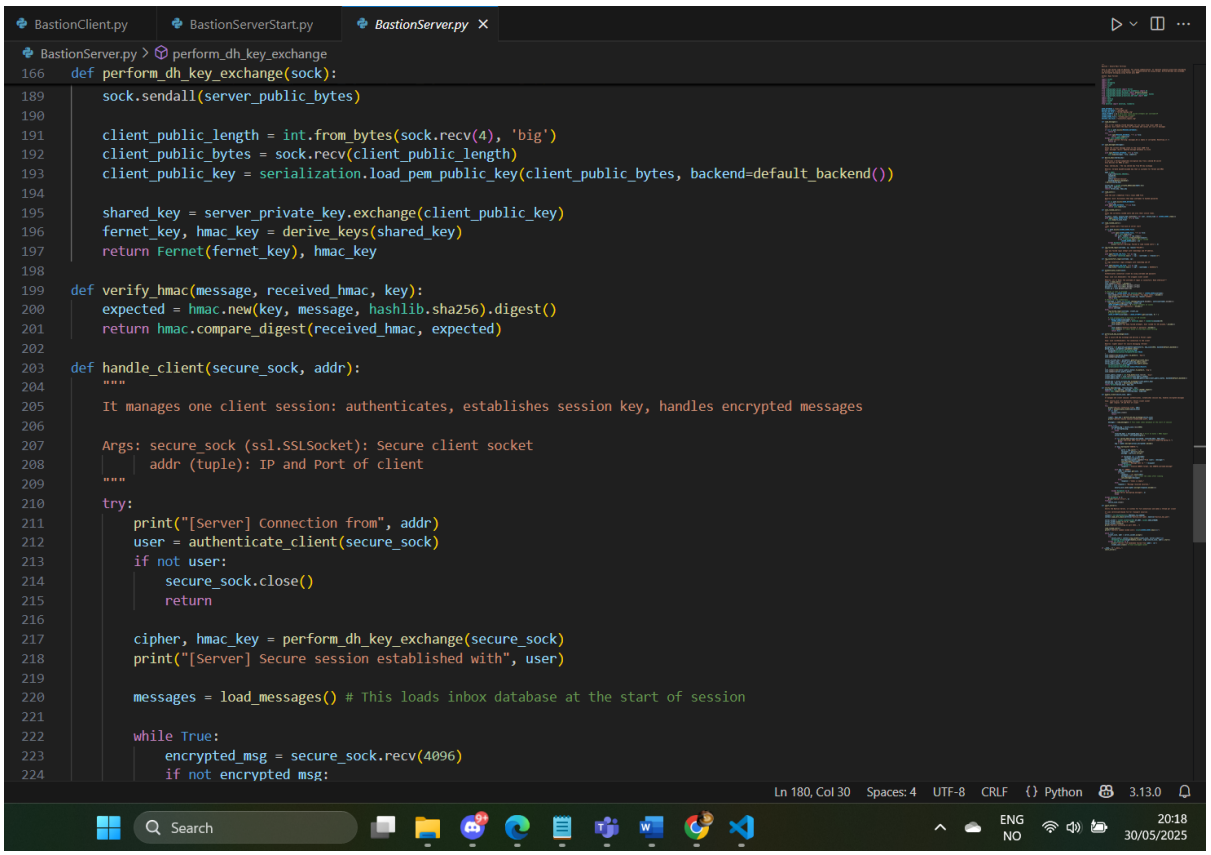


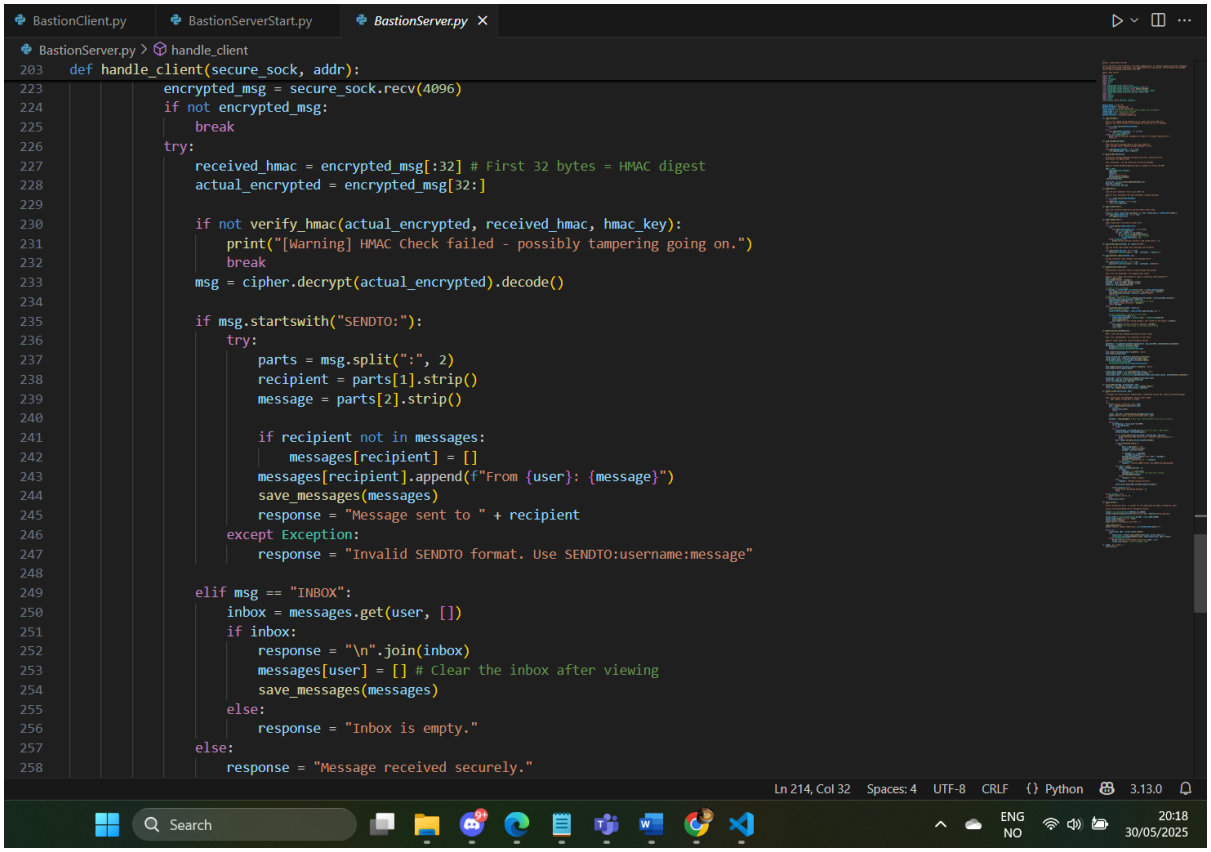
```python
def load_messages():
    Returns: dict where the keys are usernames and values are list of messages
    """
    if not os.path.exists(MESSAGE_DATABASE):
        return {}
    try:
        with open(MESSAGE_DATABASE, "r") as file:
            return json.load(file)
    except json.JSONDecodeError:
        print("[Server] Warning: messages.db is empty or corrupted. Resetting it.")
        return {}

def save_messages(messages):
    """
    Saves the current message state to the local JSON file
    Args: messages (dict): Updated message database to write
    """
    with open(MESSAGE_DATABASE, "w") as file:
        json.dump(messages, file, indent=2)

def derive_keys(shared_key):
    """
    It derives a Fernet-compatible encryption key from a shared DH secret
    Also derives for HMAC as well

    Args: shared_key : The raw shared key from DH key exchange

    Returns: 32-byte base64-encoded key that is suitable for Fernet and HMAC
    """
    hkdf = HKDF(
        algorithm=hashes.SHA256(),
        length=64,
        salt=None,
        info=b'bastion-session',
        backend=default_backend()
    ).derive(shared_key)
```

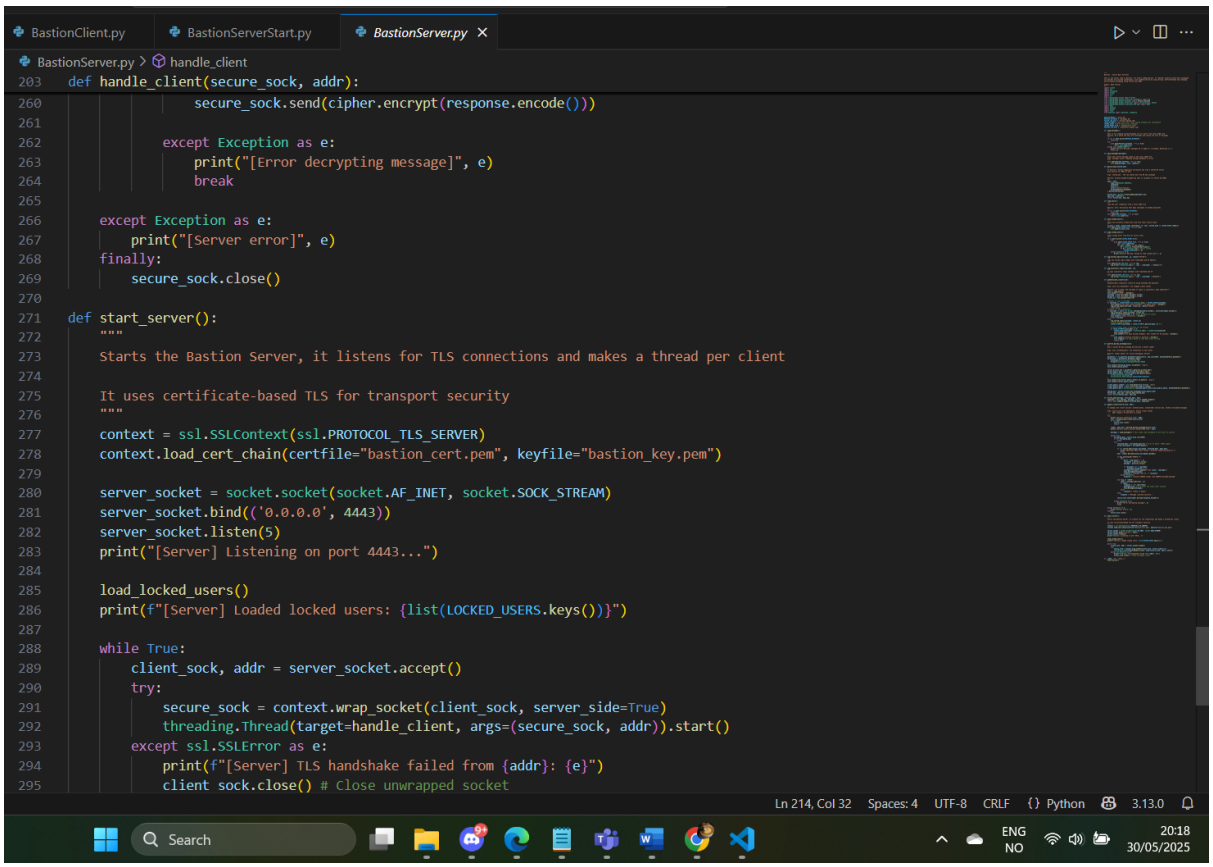

BastionClient.py
BastionServerStart.py
BastionServer.py
BastionServer.py > ...
def load_users():
Load the user credential from a local JSON file
Returns: dict: Dictionary that maps usernames to hashed passwords
if not os.path.exists(USER_DATABASE):
return {}
with open(USER_DATABASE, "r") as file:
return json.load(file)
def save_locked_users():
Saves the currently locked users and also their unlock times.
to_save = {user: unlock_time.isoformat() for user, unlock_time in LOCKED_USERS.items()}
with open(LOCKED_USERS_FILE, "w") as file:
json.dump(to_save,file)
def load_locked_users():
Loads locked users from disk at server start
if os.path.exists(LOCKED_USERS_FILE):
try:
with open(LOCKED_USERS_FILE, "r") as file:
raw = json.load(file)
for user, timestr in raw.items():
dt = datetime.fromisoformat(timestr)
if dt > datetime.now(): #Still locked
LOCKED_USERS[user] = dt
except Exception as e:
print("[Server] Warning: Failed to load locked users:", e)
def log_failed_login(username, ip, reason="FAILED"):
Logs any failed login atempt with timestamps and IP address.
Ln 22, Col 15 Spaces: 4 UTF-8 CRLF {} Python 3.13.0
Search
ENG NO
20:17 30/05/2025


BastionClient.py
BastionServerStart.py
BastionServer.py
BastionServer.py > load_locked_users
def log_failed_login(username, ip, reason="FAILED"):
Logs any failed login atempt with timestamps and IP address.
with open(FAILED_LOG_FILE, "a") as log:
log.write(f"{datetime.now()} | {ip} | {username} | {reason}\n")
def log_sucessfull_login(username, ip):
It logs sucessfull login attempts with timestamp and IP
with open(SUCCESS_LOG_FILE, "a") as log:
log.write(f"{datetime.now()} | {ip} | {username} | SUCESS\n")
def authenticate_client(sock):
Authenticates connection client by using username and password
Args: sock (ssl.SSLSocket): TLS wrapped client socket
Returns: str or None: The username if login is sucessfull; None otherwise"""
users = load_users()
sock.sendall("LOGIN: ".encode())
username = sock.recv(1024).decode().strip()
password = sock.recv(1024).decode().strip()
client_ip = sock.getpeername()[0]
# Checks if user is blocked
if username in LOCKED_USERS and datetime.now() < LOCKED_USERS[username]:
sock.sendall("Too many failed attempts. Try again later.".encode())
log_failed_login(username, client_ip, reason="LOCKED")
return None
# Checks for the credentials
if username in users and bcrypt.checkpw(password.encode(), users[username].encode()):
log_sucessfull_login(username, client_ip)
LOGIN_ATTEMPTS[username] = 0 # Reset attempts on sucess
sock.sendall("Login successful.".encode())
return username
Ln 103, Col 13 Spaces: 4 UTF-8 CRLF {} Python 3.13.0
Search
ENG NO
20:18 30/05/2025

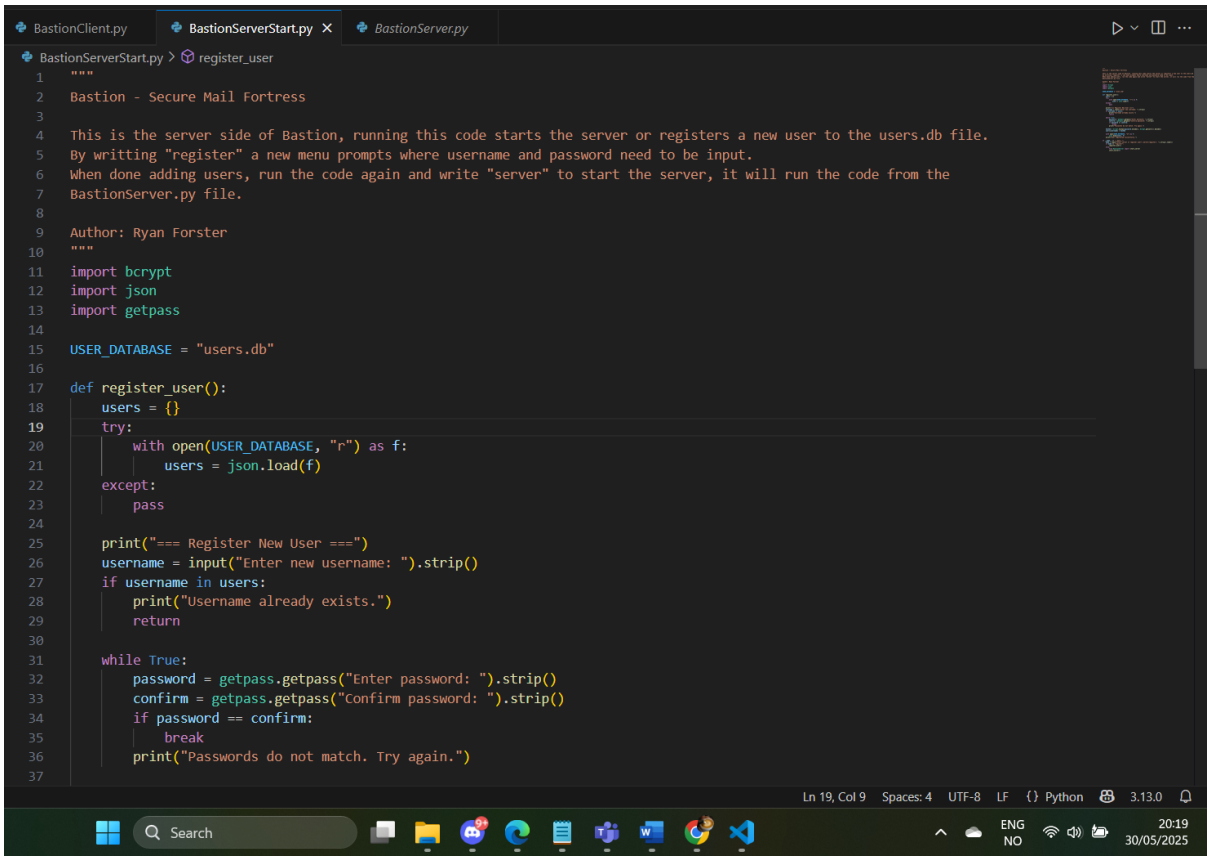

BastionClient.py
BastionServerStart.py
BastionServer.py
BastionServer.py > load_locked_users
def authenticate_client(sock):
log_failed_login(username, client_ip)
# Count failed attempts
LOGIN_ATTEMPTS[username] = LOGIN_ATTEMPTS.get(username, 0) + 1
# Lock account after 3 failures for 30 seconds
if LOGIN_ATTEMPTS[username] >= 3:
LOCKED_USERS[username] = datetime.now() + timedelta(seconds=30)
save_locked_users()
sock.sendall("Too many failed attempts. User locked for 30 seconds.".encode())
else:
sock.sendall("Invalid username or password.".encode())
time.sleep(2) # A small delay to slow down brute forcing
return None
def perform_dh_key_exchange(sock):
"""
Does a secure DH key exchange and derives a Fernet cipher
Args: sock (sslSSLSocket): TLS connection to the client
Returns: Cipher object for secure messaging (fernet)
"""
parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())
param_bytes = parameters.parameter_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.ParameterFormat.PKCS3
)
sock.sendall(len(param_bytes).to_bytes(4, 'big'))
sock.sendall(param_bytes)
server_private_key = parameters.generate_private_key()
server_public_key = server_private_key.public_key()
server_public_bytes = server_public_key.public_bytes(
serialization.Encoding.PEM,
serialization.PublicFormat.SubjectPublicKeyInfo
)
Ln 103, Col 13
Spaces: 4
UTF-8
CRLF
Python
3.13.0
Search
ENG
NO
20:18
30/05/2025


BastionClient.py
BastionServerStart.py
BastionServer.py
BastionServer.py > perform_dh_key_exchange
def perform_dh_key_exchange(sock):
sock.sendall(server_public_bytes)
client_public_length = int.from_bytes(sock.recv(4), 'big')
client_public_bytes = sock.recv(client_public_length)
client_public_key = serialization.load_pem_public_key(client_public_bytes, backend=default_backend())
shared_key = server_private_key.exchange(client_public_key)
fernet_key, hmac_key = derive_keys(shared_key)
return Fernet(fernet_key), hmac_key
def verify_hmac(message, received_hmac, key):
expected = hmac.new(key, message, hashlib.sha256).digest()
return hmac.compare_digest(received_hmac, expected)
def handle_client(secure_sock, addr):
"""
It manages one client session: authenticates, establishes session key, handles encrypted messages
Args: secure_sock (ssl.SSLSocket): Secure client socket
addr (tuple): IP and Port of client
"""
try:
print("[Server] Connection from", addr)
user = authenticate_client(secure_sock)
if not user:
secure_sock.close()
return
cipher, hmac_key = perform_dh_key_exchange(secure_sock)
print("[Server] Secure session established with", user)
messages = load_messages() # This loads inbox database at the start of session
while True:
encrypted_msg = secure_sock.recv(4096)
if not encrypted msg:
Ln 180, Col 30
Spaces: 4
UTF-8
CRLF
Python
3.13.0
Search
ENG
NO
20:18
30/05/2025

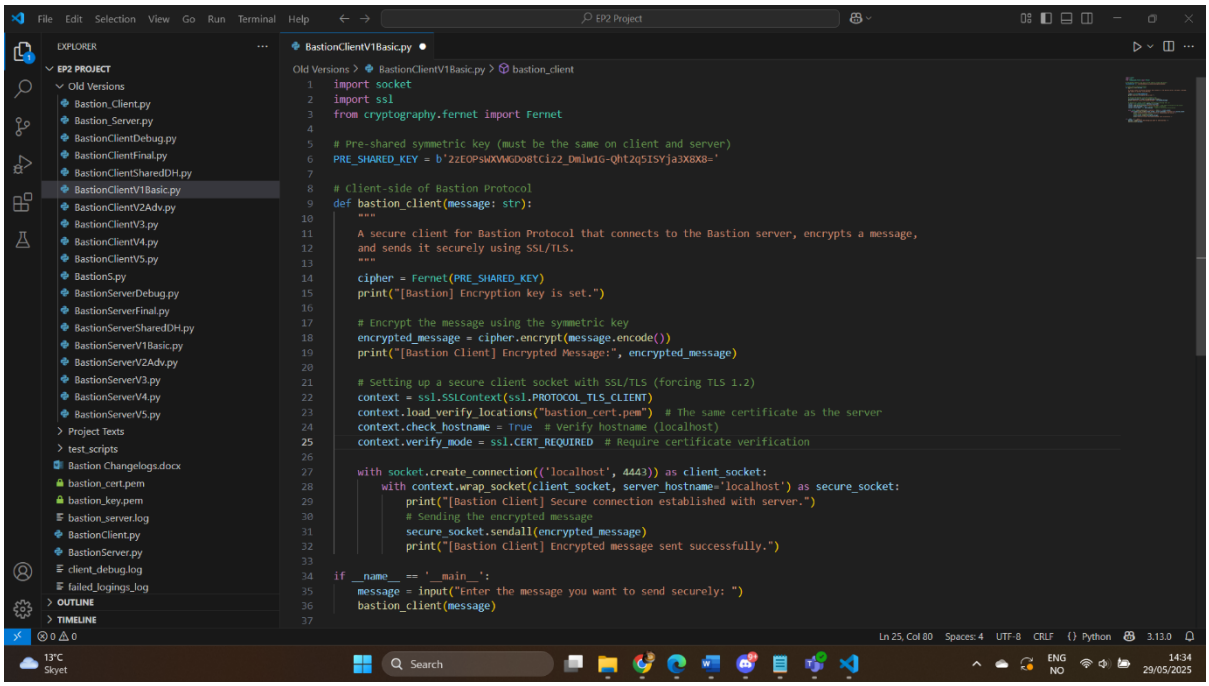

BastionClient.py
BastionServerStart.py
BastionServer.py
BastionServer.py > handle_client
def handle_client(secure_sock, addr):
encrypted_msg = secure_sock.recv(4096)
if not encrypted_msg:
break
try:
received_hmac = encrypted_msg[:32] # First 32 bytes = HMAC digest
actual_encrypted = encrypted_msg[32:]
if not verify_hmac(actual_encrypted, received_hmac, hmac_key):
print("[Warning] HMAC Check failed - possibly tampering going on.")
break
msg = cipher.decrypt(actual_encrypted).decode()
if msg.startswith("SENDTO:"):
try:
parts = msg.split(":", 2)
recipient = parts[1].strip()
message = parts[2].strip()
if recipient not in messages:
messages[recipient] = []
messages[recipient].append(f"From {user}: {message}")
save_messages(messages)
response = "Message sent to " + recipient
except Exception:
response = "Invalid SENDTO format. Use SENDTO:username:message"
elif msg == "INBOX":
inbox = messages.get(user, [])
if inbox:
response = "\n".join(inbox)
messages[user] = [] # Clear the inbox after viewing
save_messages(messages)
else:
response = "Inbox is empty."
else:
response = "Message received securely."
Ln 214, Col 32
Spaces: 4
UTF-8
CRLF
Python
3.13.0
Search
ENG
NO
20:18
30/05/2025


BastionClient.py
BastionServerStart.py
BastionServer.py
BastionServer.py > handle_client
def handle_client(secure_sock, addr):
secure_sock.send(cipher.encrypt(response.encode()))
except Exception as e:
print("[Error decrypting message]", e)
break
except Exception as e:
print("[Server error]", e)
finally:
secure_sock.close()
def start_server():
"""
Starts the Bastion Server, it listens for TLS connections and makes a thread per client
It uses certificate-based TLS for transport security
"""
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain(certfile="bastion_cert.pem", keyfile="bastion_key.pem")
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('0.0.0.0', 4443))
server_socket.listen(5)
print("[Server] Listening on port 4443...")
load_locked_users()
print(f"[Server] Loaded locked users: {list(LOCKED_USERS.keys())}")
while True:
client_sock, addr = server_socket.accept()
try:
secure_sock = context.wrap_socket(client_sock, server_side=True)
threading.Thread(target=handle_client, args=(secure_sock, addr)).start()
except ssl.SSLError as e:
print(f"[Server] TLS handshake failed from {addr}: {e}")
client sock.close() # Close unwrapped socket
Ln 214, Col 32
Spaces: 4
UTF-8
CRLF
Python
3.13.0
Search
ENG
NO
20:18
30/05/2025

BastionServerStart (BSS)

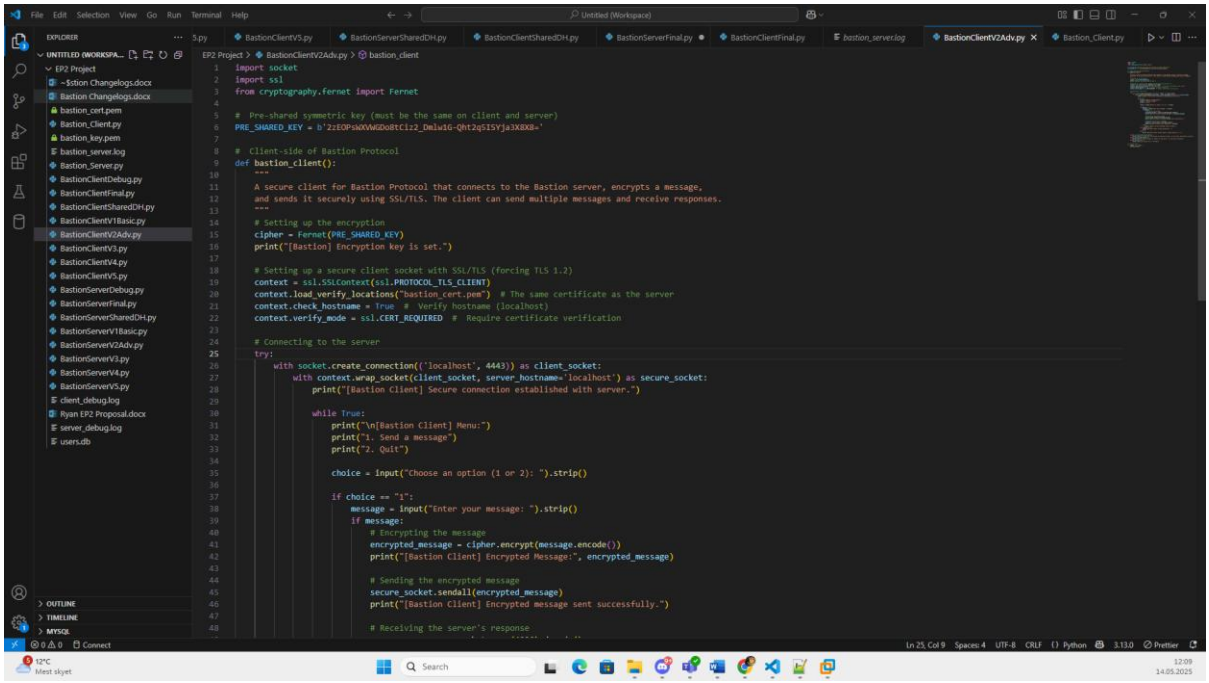


Bastion Client V1 (**BCV1**)

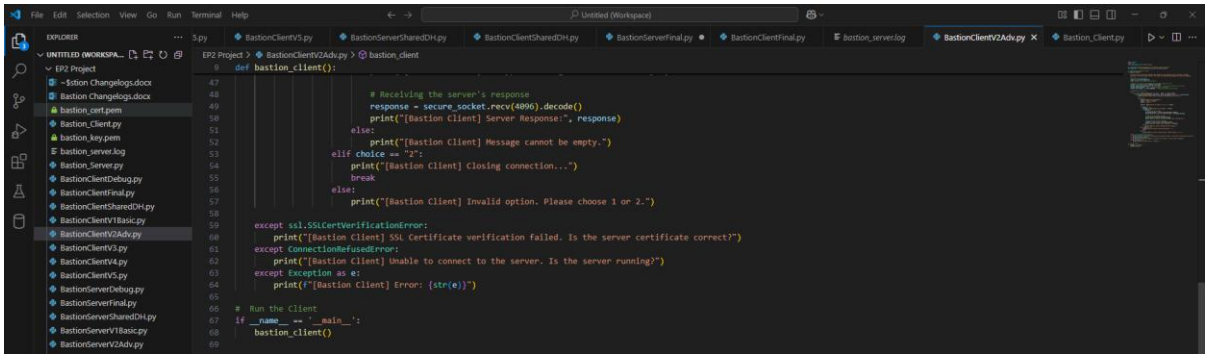


```python
import socket
import ssl
from cryptography.fernet import Fernet

# Pre-shared symmetric key (must be the same on client and server)
PRE_SHARED_KEY = b'2zEOPsWXVWGDo8tCiz2_Dmlw1G-Qht2q5ISYja3X8X8='

# Client-side of Bastion Protocol
def bastion_client(message: str):
    """
    A secure client for Bastion Protocol that connects to the Bastion server, encrypts a message,
    and sends it securely using SSL/TLS.
    """
    cipher = Fernet(PRE_SHARED_KEY)
    print("[Bastion] Encryption key is set.")

    # Encrypt the message using the symmetric key
    encrypted_message = cipher.encrypt(message.encode())
    print("[Bastion Client] Encrypted Message:", encrypted_message)

    # Setting up a secure client socket with SSL/TLS (forcing TLS 1.2)
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.load_verify_locations("bastion_cert.pem")  # The same certificate as the server
    context.check_hostname = True  # Verify hostname (localhost)
    context.verify_mode = ssl.CERT_REQUIRED  # Require certificate verification

    with socket.create_connection(('localhost', 4443)) as client_socket:
        with context.wrap_socket(client_socket, server_hostname='localhost') as secure_socket:
            print("[Bastion Client] Secure connection established with server.")
            # Sending the encrypted message
            secure_socket.sendall(encrypted_message)
            print("[Bastion client] Encrypted message sent successfully.")

if __name__ == '__main__':
    message = input("Enter the message you want to send securely: ")
    bastion_client(message)
```

Bastion Client V2 (**BCV2**)

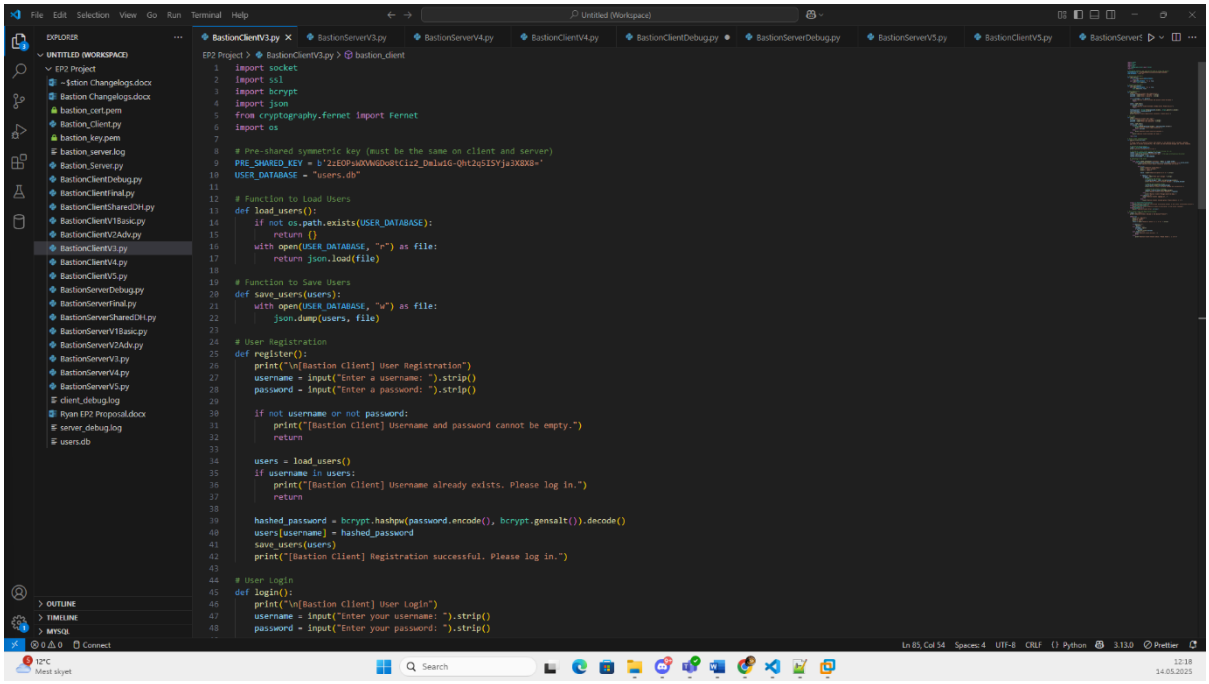


```python
import socket
import ssl
from cryptography.fernet import Fernet

#  Pre-shared symmetric key (must be the same on client and server)
PRE_SHARED_KEY = b'2zEOPsWXVWGDo8tCiz2_Dmlw1G-Qht2q5ISYja3X8X8='

#  Client-side of Bastion Protocol
def bastion_client():
    """
    A secure client for Bastion Protocol that connects to the Bastion server, encrypts a message,
    and sends it securely using SSL/TLS. The client can send multiple messages and receive responses.
    """
    # Setting up the encryption
    cipher = Fernet(PRE_SHARED_KEY)
    print("[Bastion] Encryption key is set.")

    # Setting up a secure client socket with SSL/TLS (forcing TLS 1.2)
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.load_verify_locations("bastion_cert.pem")  # The same certificate as the server
    context.check_hostname = True  #  Verify hostname (localhost)
    context.verify_mode = ssl.CERT_REQUIRED  #  Require certificate verification

    # Connecting to the server
    try:
        with socket.create_connection(('localhost', 4443)) as client_socket:
            with context.wrap_socket(client_socket, server_hostname='localhost') as secure_socket:
                print("[Bastion Client] Secure connection established with server.")

                while True:
                    print("\n[Bastion Client] Menu:")
                    print("1. Send a message")
                    print("2. Quit")

                    choice = input("Choose an option (1 or 2): ").strip()

                    if choice == "1":
                        message = input("Enter your message: ").strip()
                        if message:
                            # Encrypting the message
                            encrypted_message = cipher.encrypt(message.encode())
                            print("[Bastion Client] Encrypted Message:", encrypted_message)

                            # Sending the encrypted message
                            secure_socket.sendall(encrypted_message)
                            print("[Bastion Client] Encrypted message sent successfully.")

                            # Receiving the server's response
```

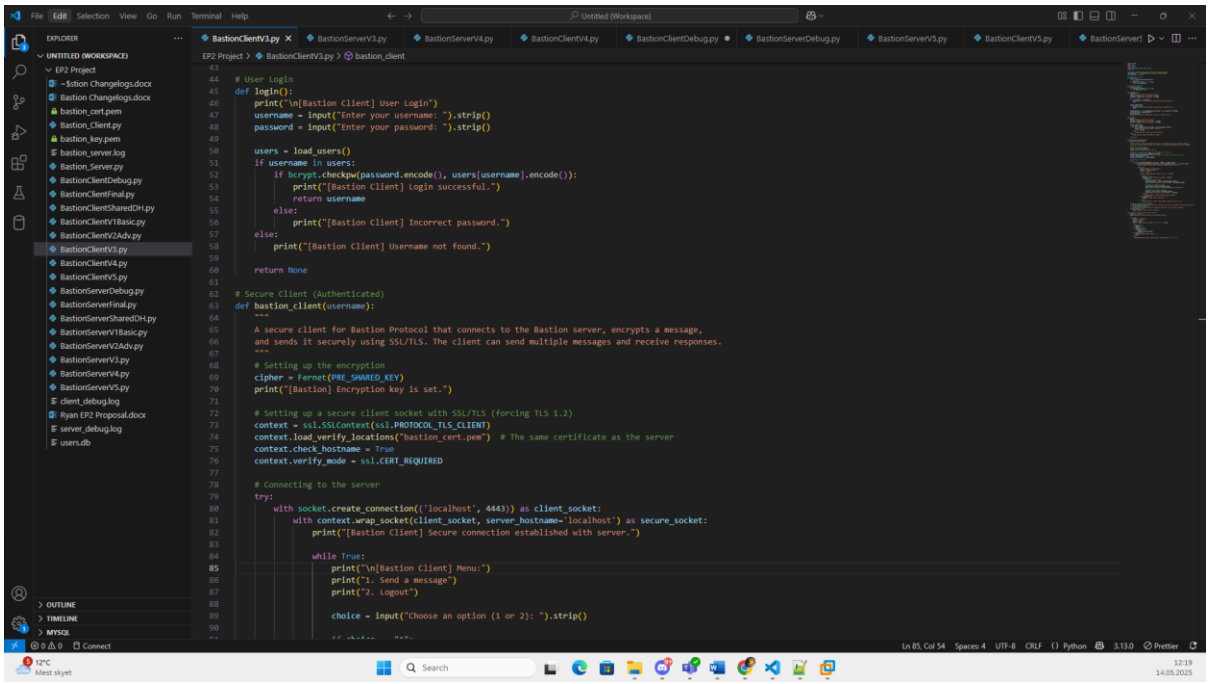


```
def bastion_client():
                        # Receiving the server's response
                        response = secure_socket.recv(4096).decode()
                        print("[Bastion Client] Server Response:", response)
                    else:
                        print("[Bastion Client] Message cannot be empty.")
                elif choice == "2":
                    print("[Bastion Client] Closing connection...")
                    break
                else:
                    print("[Bastion Client] Invalid option. Please choose 1 or 2.")

    except ssl.SSLCertVerificationError:
        print("[Bastion Client] SSL Certificate verification failed. Is the server certificate correct?")
    except ConnectionRefusedError:
        print("[Bastion Client] Unable to connect to the server. Is the server running?")
    except Exception as e:
        print(f"[Bastion Client] Error: {str(e)}")

#  Run the Client
if __name__ == '__main__':
    bastion_client()
```

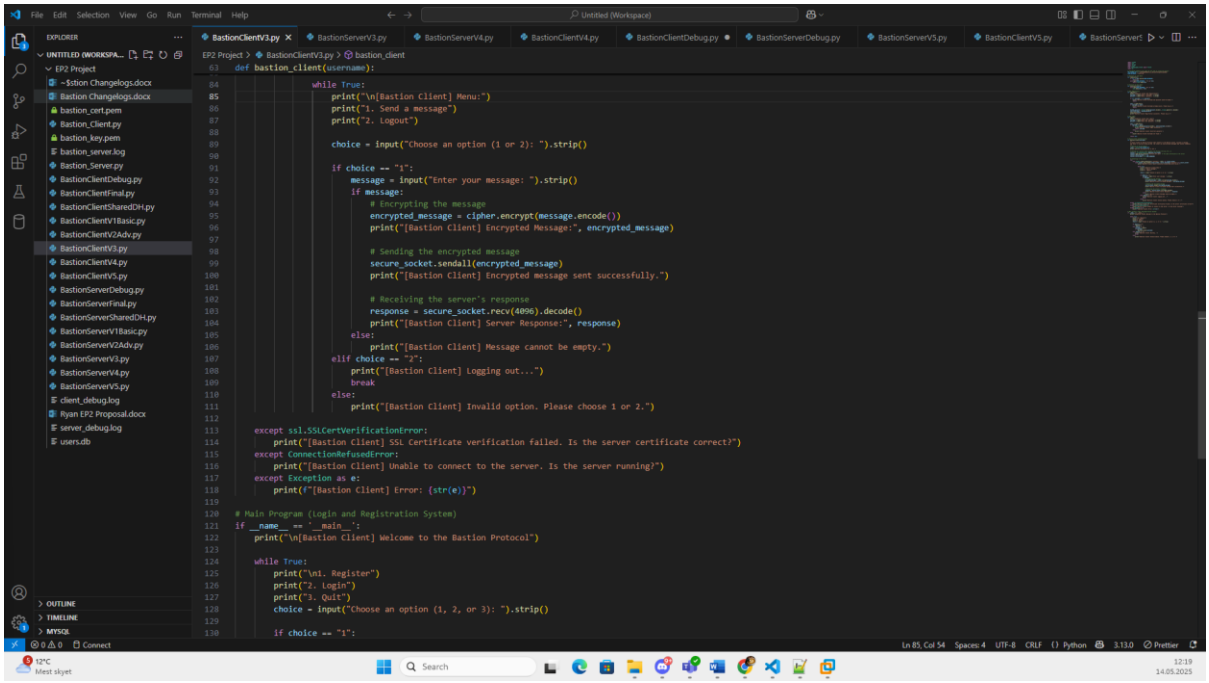
Bastion Client V3 (**BCV3**)



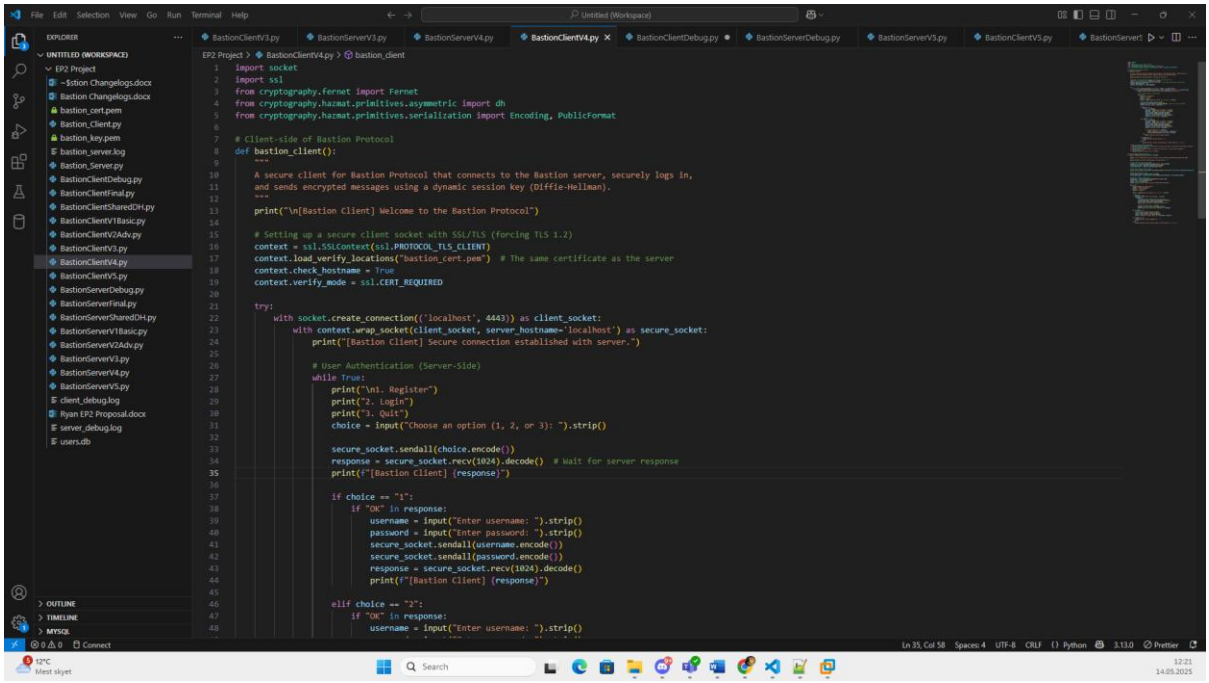
```
import socket
import ssl
import bcrypt
import json
from cryptography.fernet import Fernet
import os

# Pre-shared symmetric key (must be the same on client and server)
PRE_SHARED_KEY = b'2zEOPskXVWGDo8tCiz2_Dmlw1G-Qht2q5ISYja3X8X8='
USER_DATABASE = "users.db"

# Function to Load Users
def load_users():
    if not os.path.exists(USER_DATABASE):
        return {}
    with open(USER_DATABASE, "r") as file:
        return json.load(file)

# Function to Save Users
def save_users(users):
    with open(USER_DATABASE, "w") as file:
        json.dump(users, file)

# User Registration
def register():
    print("\n[Bastion Client] User Registration")
    username = input("Enter a username: ").strip()
    password = input("Enter a password: ").strip()

    if not username or not password:
        print("[Bastion Client] Username and password cannot be empty.")
        return

    users = load_users()
    if username in users:
        print("[Bastion Client] Username already exists. Please log in.")
        return

    hashed_password = bcrypt.hashpw(password.encode(), bcrypt.gensalt()).decode()
    users[username] = hashed_password
    save_users(users)
    print("[Bastion Client] Registration successful. Please log in.")

# User Login
def login():
    print("\n[Bastion Client] User Login")
    username = input("Enter your username: ").strip()
    password = input("Enter your password: ").strip()
```

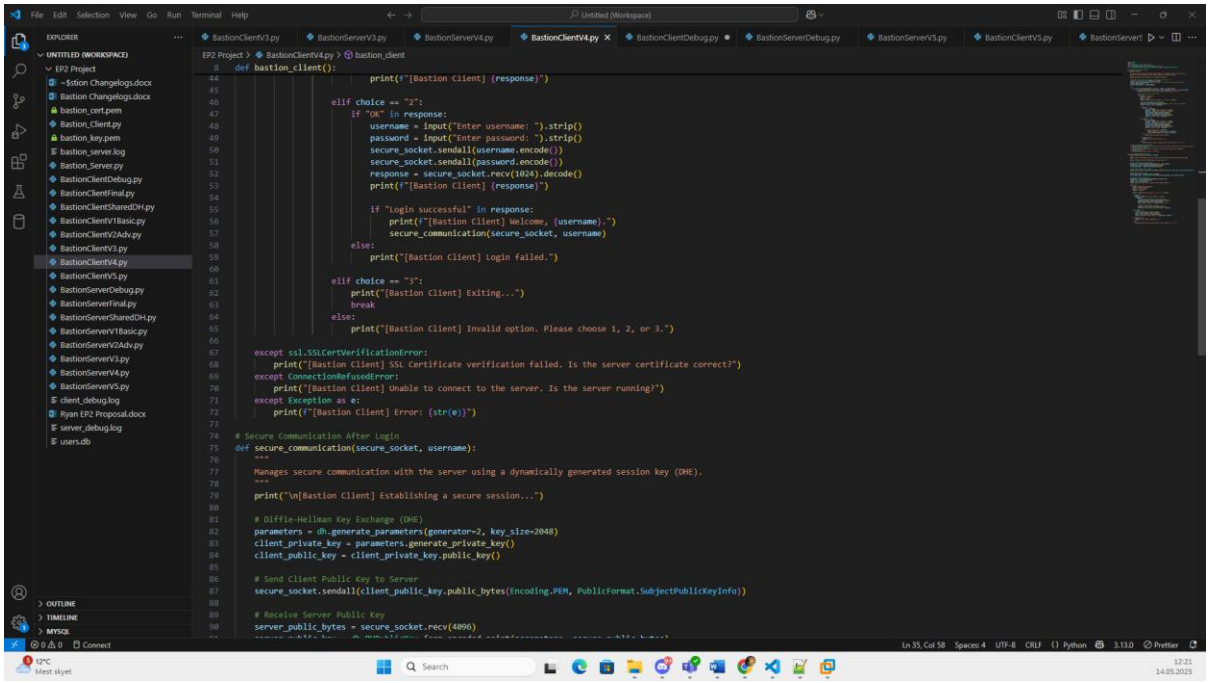


```
# User Login
def login():
    print("\n[Bastion Client] User Login")
    username = input("Enter your username: ").strip()
    password = input("Enter your password: ").strip()

    users = load_users()
    if username in users:
        if bcrypt.checkpw(password.encode(), users[username].encode()):
            print("[Bastion Client] Login successful.")
            return username
        else:
            print("[Bastion Client] Incorrect password.")
    else:
        print("[Bastion Client] Username not found.")

    return None

# Secure Client (Authenticated)
def bastion_client(username):
    """
    A secure client for Bastion Protocol that connects to the Bastion server, encrypts a message,
    and sends it securely using SSL/TLS. The client can send multiple messages and receive responses.
    """
    # Setting up the encryption
    cipher = Fernet(PRE_SHARED_KEY)
    print("[Bastion] Encryption key is set.")

    # Setting up a secure client socket with SSL/TLS (forcing TLS 1.2)
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.load_verify_locations("bastion_cert.pem")  # The same certificate as the server
    context.check_hostname = True
    context.verify_mode = ssl.CERT_REQUIRED

    # Connecting to the server
    try:
        with socket.create_connection(('localhost', 4443)) as client_socket:
            with context.wrap_socket(client_socket, server_hostname='localhost') as secure_socket:
                print("[Bastion Client] Secure connection established with server.")

                while True:
                    print("\n[Bastion Client] Menu:")
                    print("1. Send a message")
                    print("2. Logout")

                    choice = input("Choose an option (1 or 2): ").strip()
```

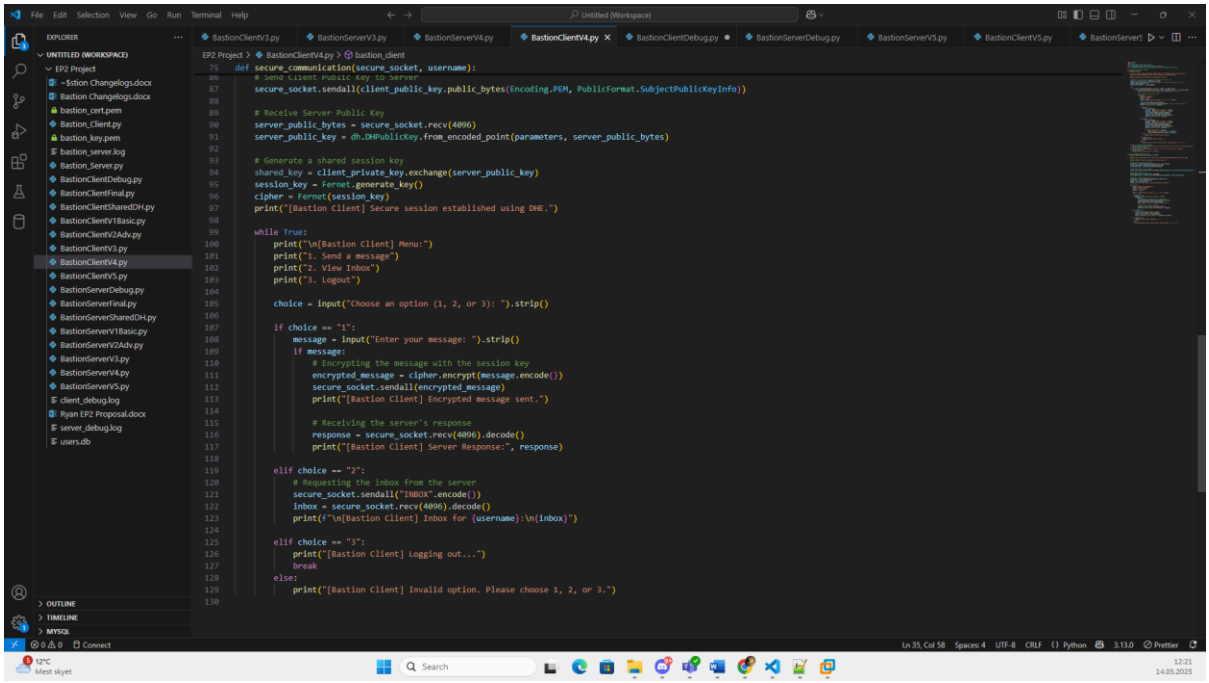


Bastion Client V4 (**BCV4**)



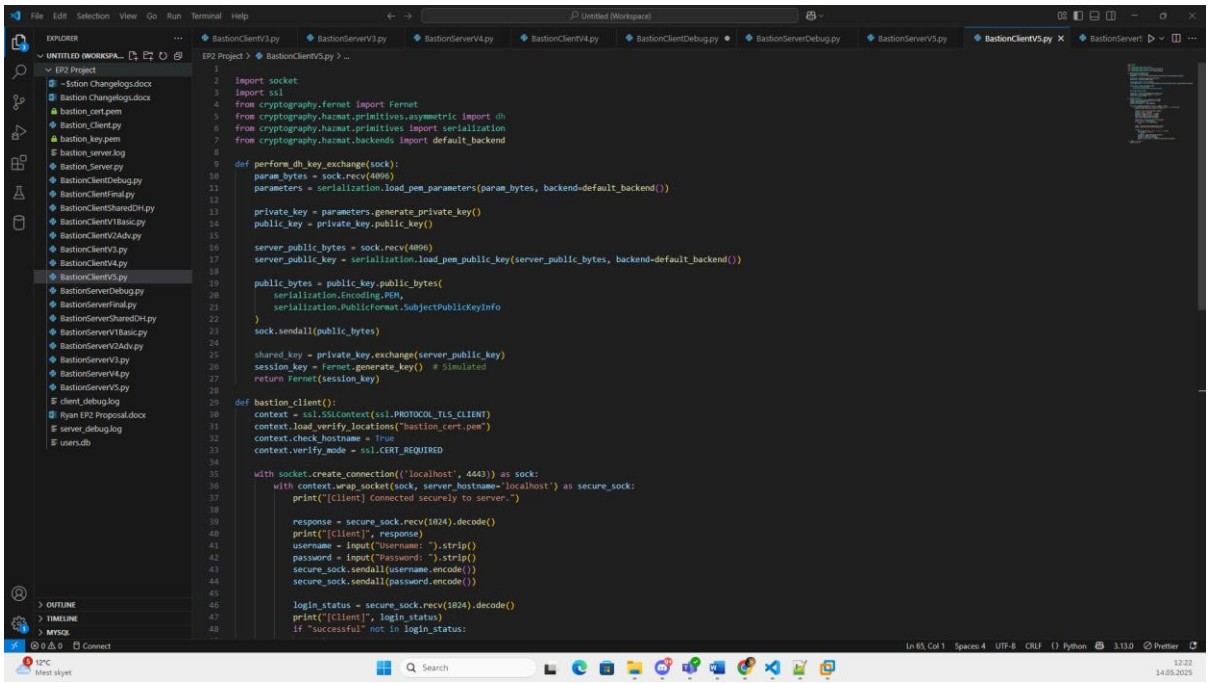


```
def bastion_client():
                        print(f"[Bastion Client] {response}")

                elif choice == "2":
                    if "OK" in response:
                        username = input("Enter username: ").strip()
                        password = input("Enter password: ").strip()
                        secure_socket.sendall(username.encode())
                        secure_socket.sendall(password.encode())
                        response = secure_socket.recv(1024).decode()
                        print(f"[Bastion Client] {response}")

                        if "Login successful" in response:
                            print(f"[Bastion Client] Welcome, {username}.")
                            secure_communication(secure_socket, username)
                    else:
                        print("[Bastion Client] Login failed.")

                elif choice == "3":
                    print("[Bastion Client] Exiting...")
                    break
                else:
                    print("[Bastion Client] Invalid option. Please choose 1, 2, or 3.")

    except ssl.SSLCertVerificationError:
        print("[Bastion Client] SSL Certificate verification failed. Is the server certificate correct?")
    except ConnectionRefusedError:
        print("[Bastion Client] Unable to connect to the server. Is the server running?")
    except Exception as e:
        print(f"[Bastion Client] Error: {str(e)}")

# Secure Communication After Login
def secure_communication(secure_socket, username):
    """
    Manages secure communication with the server using a dynamically generated session key (DHE).
    """
    print("\n[Bastion Client] Establishing a secure session...")

    # Diffie-Hellman Key Exchange (DHE)
    parameters = dh.generate_parameters(generator=2, key_size=2048)
    client_private_key = parameters.generate_private_key()
    client_public_key = client_private_key.public_key()

    # Send Client Public Key to Server
    secure_socket.sendall(client_public_key.public_bytes(Encoding.PEM, PublicFormat.SubjectPublicKeyInfo))

    # Receive Server Public Key
    server_public_bytes = secure_socket.recv(4096)
```

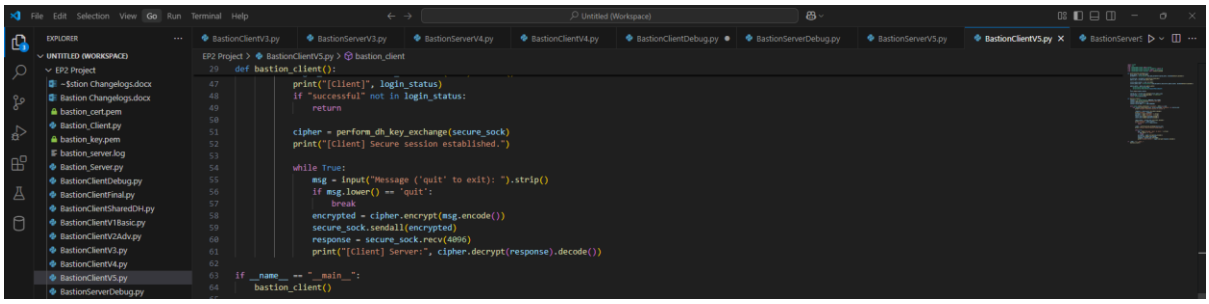


```
def secure_communication(secure_socket, username):
    # Send Client Public Key to Server
    secure_socket.sendall(client_public_key.public_bytes(Encoding.PEM, PublicFormat.SubjectPublicKeyInfo))

    # Receive Server Public Key
    server_public_bytes = secure_socket.recv(4096)
    server_public_key = dh.DHPublicKey.from_encoded_point(parameters, server_public_bytes)

    # Generate a shared session key
    shared_key = client_private_key.exchange(server_public_key)
    session_key = Fernet.generate_key()
    cipher = Fernet(session_key)
    print("[Bastion Client] Secure session established using DHE.")

    while True:
        print("\n[Bastion Client] Menu:")
        print("1. Send a message")
        print("2. View Inbox")
        print("3. Logout")

        choice = input("Choose an option (1, 2, or 3): ").strip()

        if choice == "1":
            message = input("Enter your message: ").strip()
            if message:
                # Encrypting the message with the session key
                encrypted_message = cipher.encrypt(message.encode())
                secure_socket.sendall(encrypted_message)
                print("[Bastion Client] Encrypted message sent.")

                # Receiving the server's response
                response = secure_socket.recv(4096).decode()
                print("[Bastion Client] Server Response:", response)

        elif choice == "2":
            # Requesting the inbox from the server
            secure_socket.sendall("INBOX".encode())
            inbox = secure_socket.recv(4096).decode()
            print(f"\n[Bastion Client] Inbox for {username}:\n{inbox}")

        elif choice == "3":
            print("[Bastion Client] Logging out...")
            break
        else:
            print("[Bastion Client] Invalid option. Please choose 1, 2, or 3.")
```

Bastion Client V5 (**BCV5**)

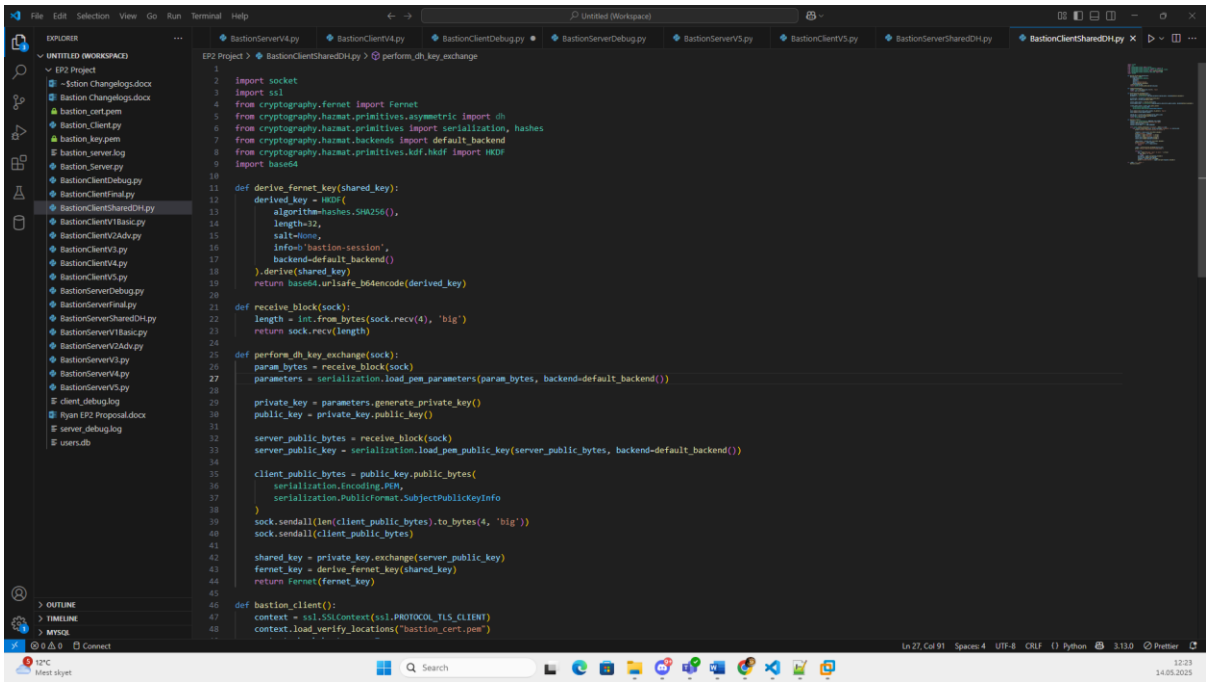


```python
import socket
import ssl
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend

def perform_dh_key_exchange(sock):
    param_bytes = sock.recv(4096)
    parameters = serialization.load_pem_parameters(param_bytes, backend=default_backend())

    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()

    server_public_bytes = sock.recv(4096)
    server_public_key = serialization.load_pem_public_key(server_public_bytes, backend=default_backend())

    public_bytes = public_key.public_bytes(
        serialization.Encoding.PEM,
        serialization.PublicFormat.SubjectPublicKeyInfo
    )
    sock.sendall(public_bytes)

    shared_key = private_key.exchange(server_public_key)
    session_key = Fernet.generate_key()  # Simulated
    return Fernet(session_key)

def bastion_client():
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.load_verify_locations("bastion_cert.pem")
    context.check_hostname = True
    context.verify_mode = ssl.CERT_REQUIRED

    with socket.create_connection(('localhost', 4443)) as sock:
        with context.wrap_socket(sock, server_hostname='localhost') as secure_sock:
            print("[Client] Connected securely to server.")

            response = secure_sock.recv(1024).decode()
            print("[Client]", response)
            username = input("Username: ").strip()
            password = input("Password: ").strip()
            secure_sock.sendall(username.encode())
            secure_sock.sendall(password.encode())

            login_status = secure_sock.recv(1024).decode()
            print("[Client]", login_status)
            if "successful" not in login_status:
```

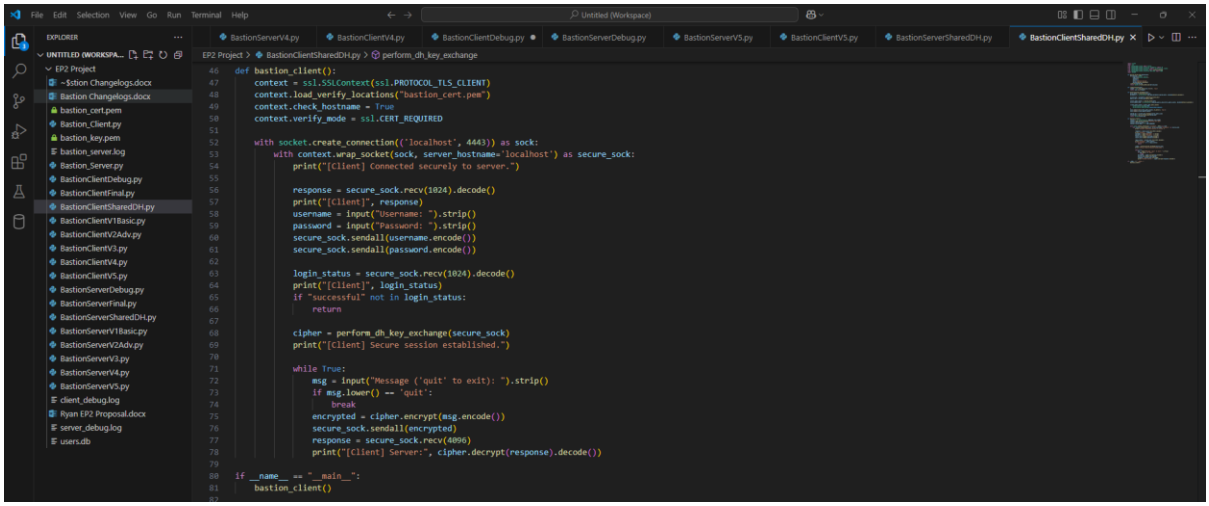


```python
def bastion_client():
            print("[Client]", login_status)
            if "successful" not in login_status:
                return

            cipher = perform_dh_key_exchange(secure_sock)
            print("[Client] Secure session established.")

            while True:
                msg = input("Message ('quit' to exit): ").strip()
                if msg.lower() == 'quit':
                    break
                encrypted = cipher.encrypt(msg.encode())
                secure_sock.sendall(encrypted)
                response = secure_sock.recv(4096)
                print("[Client] Server:", cipher.decrypt(response).decode())

if __name__ == "__main__":
    bastion_client()
```

Bastion Client V6 (**BCV6**)

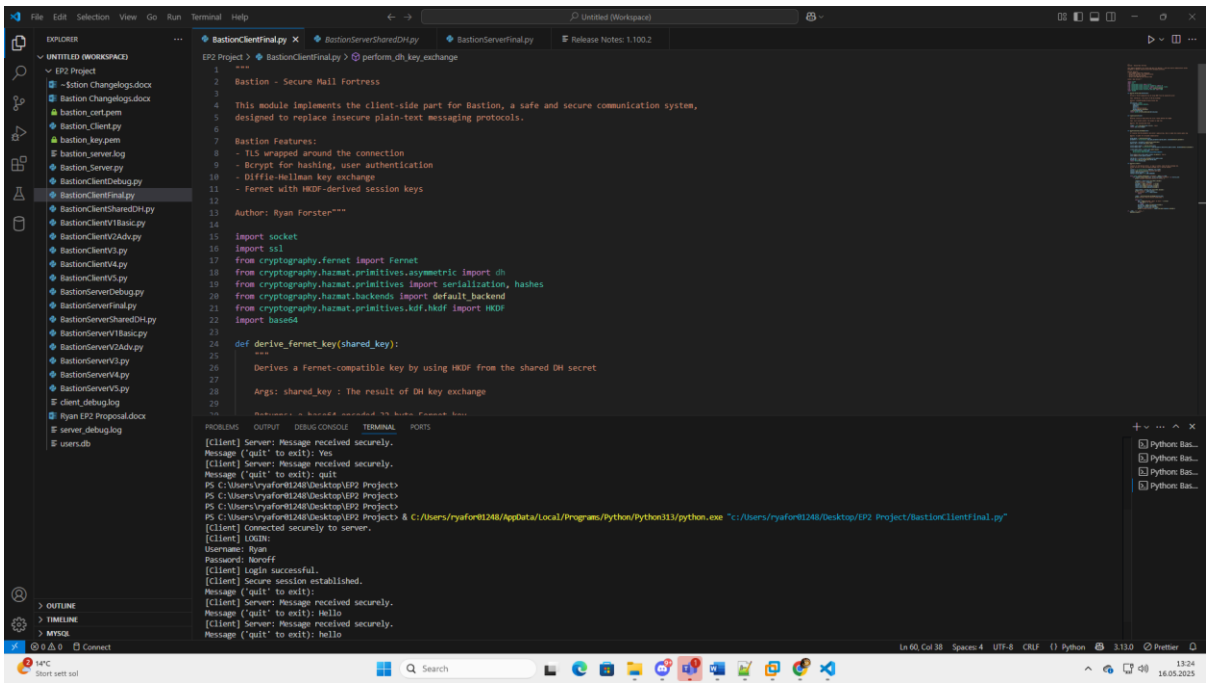


```python
import socket
import ssl
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import base64

def derive_fernet_key(shared_key):
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'bastion-session',
        backend=default_backend()
    ).derive(shared_key)
    return base64.urlsafe_b64encode(derived_key)

def receive_block(sock):
    length = int.from_bytes(sock.recv(4), 'big')
    return sock.recv(length)

def perform_dh_key_exchange(sock):
    param_bytes = receive_block(sock)
    parameters = serialization.load_pem_parameters(param_bytes, backend=default_backend())

    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()

    server_public_bytes = receive_block(sock)
    server_public_key = serialization.load_pem_public_key(server_public_bytes, backend=default_backend())

    client_public_bytes = public_key.public_bytes(
        serialization.Encoding.PEM,
        serialization.PublicFormat.SubjectPublicKeyInfo
    )
    sock.sendall(len(client_public_bytes).to_bytes(4, 'big'))
    sock.sendall(client_public_bytes)

    shared_key = private_key.exchange(server_public_key)
    fernet_key = derive_fernet_key(shared_key)
    return Fernet(fernet_key)

def bastion_client():
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.load_verify_locations("bastion_cert.pem")
```



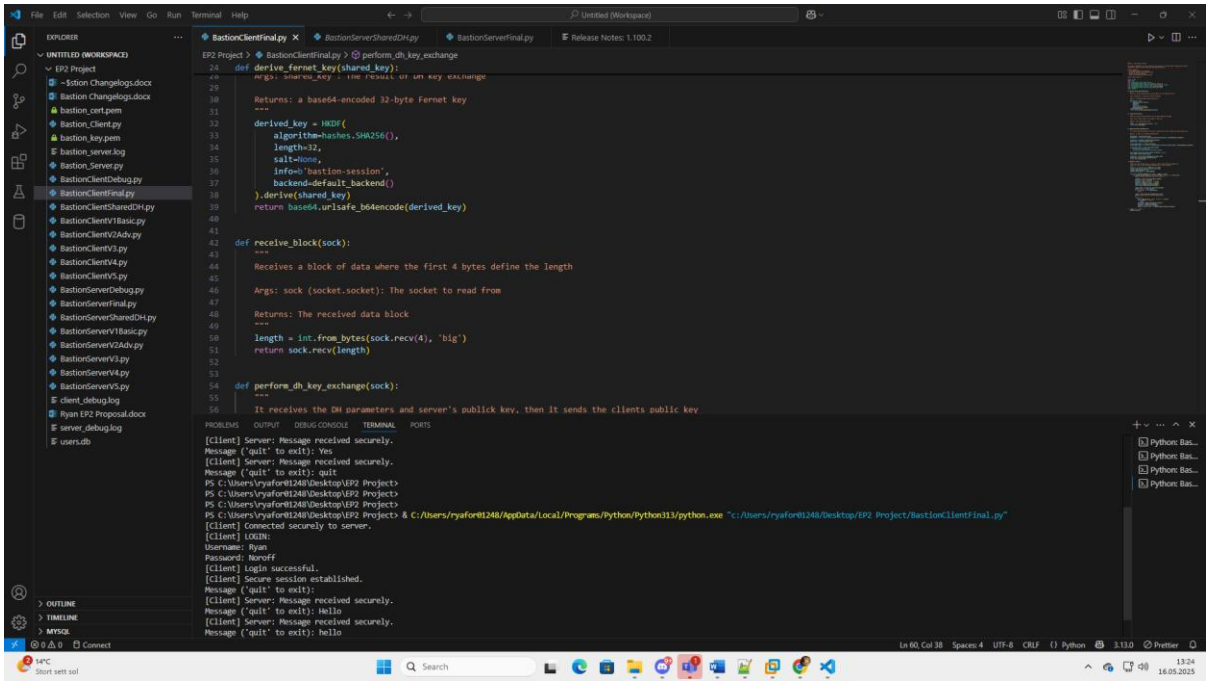
```python
def bastion_client():
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.load_verify_locations("bastion_cert.pem")
    context.check_hostname = True
    context.verify_mode = ssl.CERT_REQUIRED

    with socket.create_connection(('localhost', 4443)) as sock:
        with context.wrap_socket(sock, server_hostname='localhost') as secure_sock:
            print("[Client] Connected securely to server.")

            response = secure_sock.recv(1024).decode()
            print("[Client]", response)
            username = input("Username: ").strip()
            password = input("Password: ").strip()
            secure_sock.sendall(username.encode())
            secure_sock.sendall(password.encode())

            login_status = secure_sock.recv(1024).decode()
            print("[Client]", login_status)
            if "successful" not in login_status:
                return

            cipher = perform_dh_key_exchange(secure_sock)
            print("[Client] Secure session established.")

            while True:
                msg = input("Message ('quit' to exit): ").strip()
                if msg.lower() == 'quit':
                    break
                encrypted = cipher.encrypt(msg.encode())
                secure_sock.sendall(encrypted)
                response = secure_sock.recv(4096)
                print("[Client] Server:", cipher.decrypt(response).decode())

if __name__ == "__main__":
    bastion_client()
```

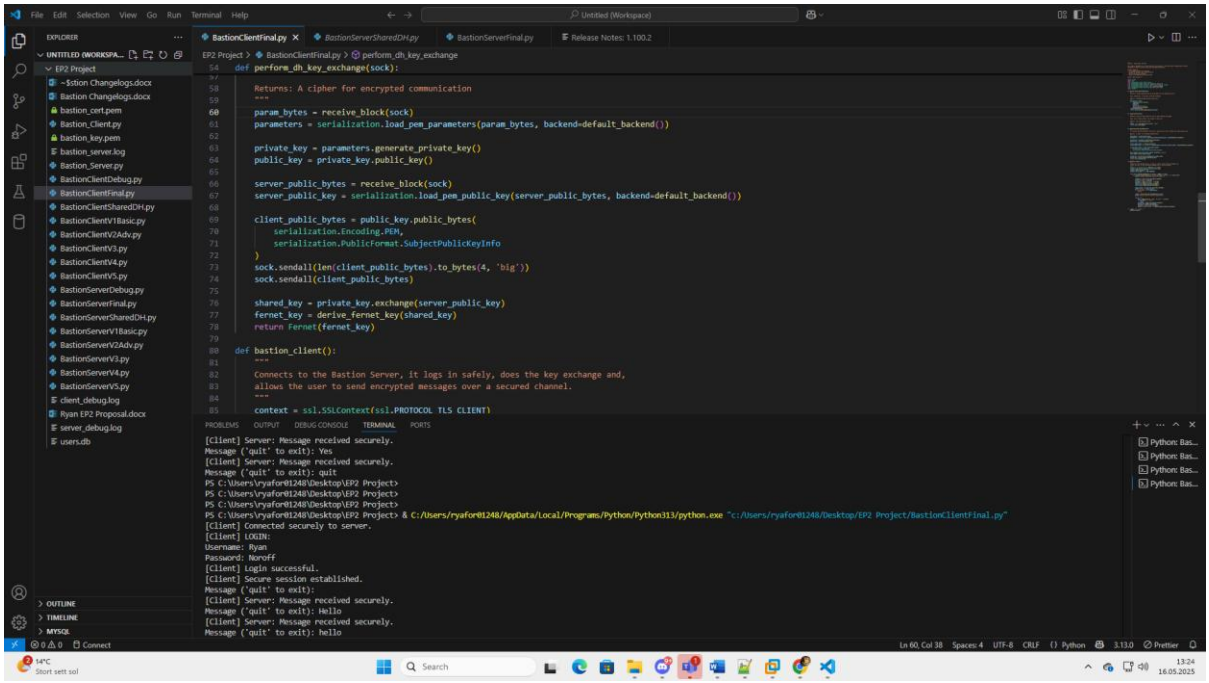
Bastion Client V7 (**BCV7**)

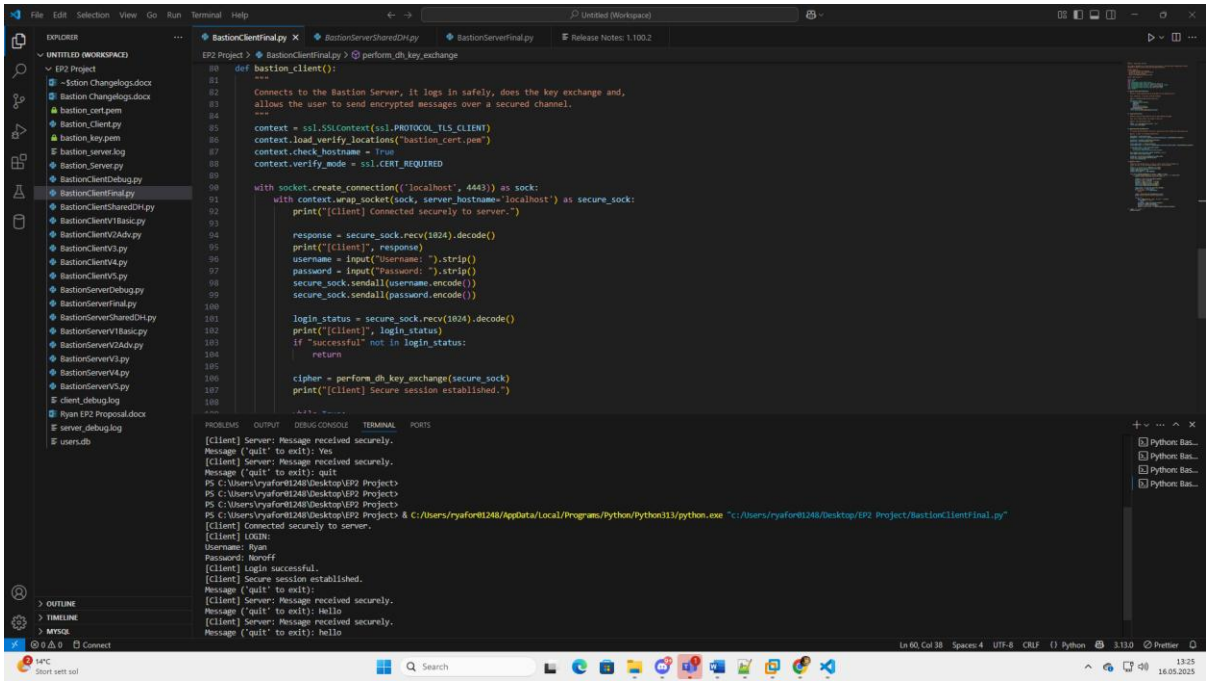


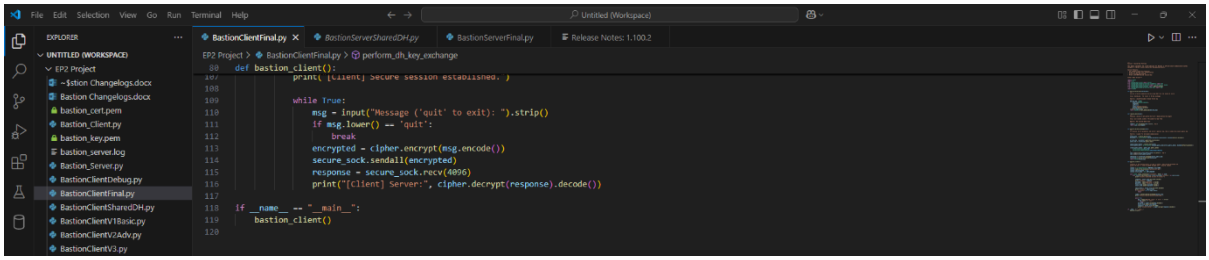
```python
"""
Bastion - Secure Mail Fortress

This module implements the client-side part for Bastion, a safe and secure communication system,
designed to replace insecure plain-text messaging protocols.

Bastion Features:
- TLS wrapped around the connection
- Bcrypt for hashing, user authentication
- Diffie-Hellman key exchange
- Fernet with HKDF-derived session keys

Author: Ryan Forster"""

import socket
import ssl
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import base64

def derive_fernet_key(shared_key):
    """
    Derives a Fernet-compatible key by using HKDF from the shared DH secret

    Args: shared_key : The result of DH key exchange
```

```
[Client] Server: Message received securely.
Message ('quit' to exit): Yes
[Client] Server: Message received securely.
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project> & C:/Users/ryafor01248/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ryafor01248/Desktop/EP2 Project/BastionClientFinal.py"
[Client] Connected securely to server.
[Client] LOGIN:
Username: Ryan
Password: Noroff
[Client] Login successful.
[Client] Secure session established.
Message ('quit' to exit):
[Client] Server: Message received securely.
Message ('quit' to exit): Hello
[Client] Server: Message received securely.
Message ('quit' to exit): hello
```

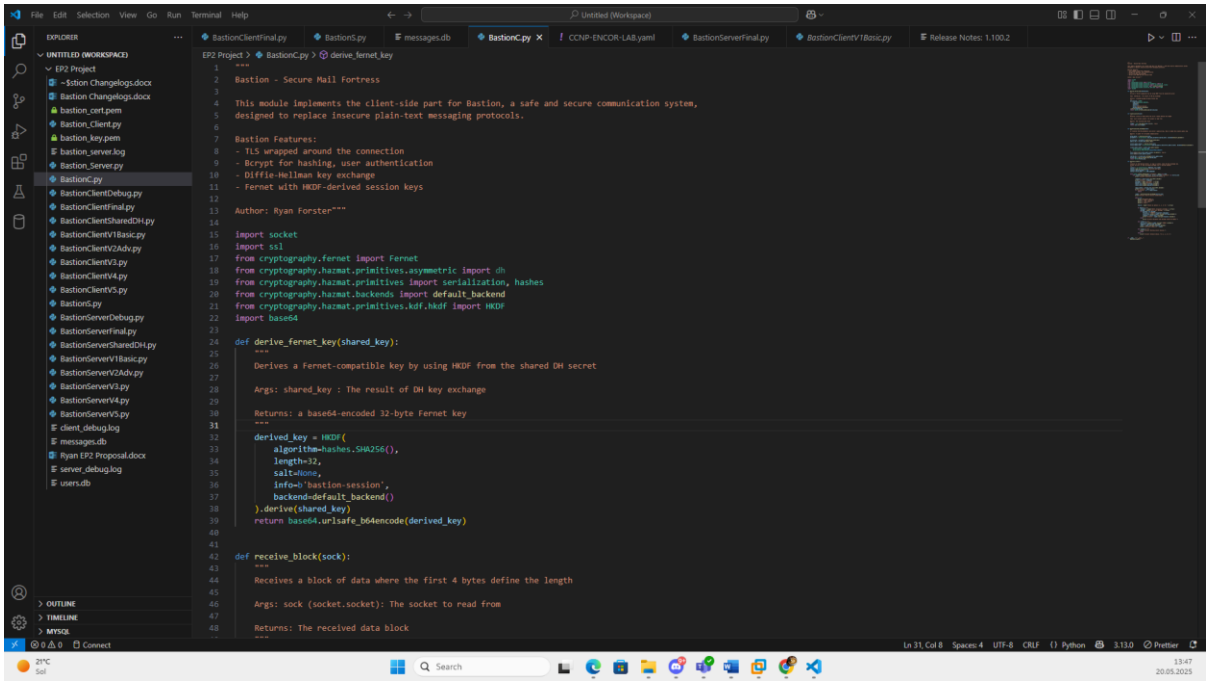

File Edit Selection View Go Run Terminal Help
Untitled (Workspace)
EXPLORER
UNTITLED (WORKSPACE)
EP2 Project
BastionClientFinal.py
BastionServerSharedDH.py
BastionServerFinal.py
Release Notes: 1.100.2
EP2 Project > BastionClientFinal.py > perform_dh_key_exchange
def derive_fernet_key(shared_key):
Args: shared_key : The result of dh key exchange
Returns: a base64-encoded 32-byte Fernet key
derived_key = HKDF(
algorithm=hashes.SHA256(),
length=32,
salt=None,
info=b'bastion-session',
backend=default_backend()
).derive(shared_key)
return base64.urlsafe_b64encode(derived_key)
def receive_block(sock):
Receives a block of data where the first 4 bytes define the length
Args: sock (socket.socket): The socket to read from
Returns: The received data block
length = int.from_bytes(sock.recv(4), 'big')
return sock.recv(length)
def perform_dh_key_exchange(sock):
It receives the DH parameters and server's publick key, then it sends the clients public key
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
[Client] Server: Message received securely.
Message ('quit' to exit): Yes
[Client] Server: Message received securely.
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project> & C:/Users/ryafor01248/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ryafor01248/Desktop/EP2 Project/BastionClientFinal.py"
[Client] Connected securely to server.
[Client] LOGIN:
Username: Ryan
Password: Noroff
[Client] Login successful.
[Client] Secure session established.
Message ('quit' to exit):
[Client] Server: Message received securely.
Message ('quit' to exit): Hello
[Client] Server: Message received securely.
Message ('quit' to exit): hello
OUTLINE
TIMELINE
MYSQL
Ln 60, Col 38 Spaces: 4 UTF-8 CRLF Python 3.13.0 Prettier
Search
13:24
16.05.2025

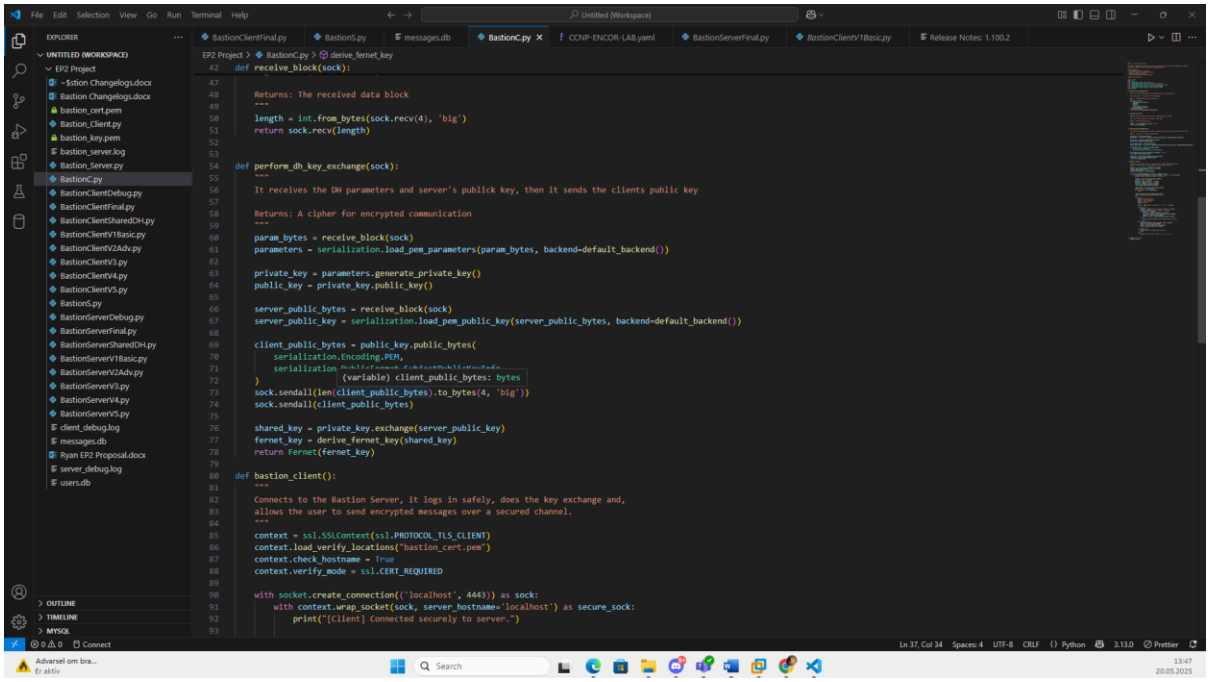

File Edit Selection View Go Run Terminal Help
Untitled (Workspace)
EXPLORER
EP2 Project
BastionClientFinal.py
BastionServerSharedDH.py
BastionServerFinal.py
Release Notes: 1.100.2
EP2 Project > BastionClientFinal.py > perform_dh_key_exchange
def perform_dh_key_exchange(sock):
Returns: A cipher for encrypted communication
param_bytes = receive_block(sock)
parameters = serialization.load_pem_parameters(param_bytes, backend=default_backend())
private_key = parameters.generate_private_key()
public_key = private_key.public_key()
server_public_bytes = receive_block(sock)
server_public_key = serialization.load_pem_public_key(server_public_bytes, backend=default_backend())
client_public_bytes = public_key.public_bytes(
serialization.Encoding.PEM,
serialization.PublicFormat.SubjectPublicKeyInfo
)
sock.sendall(len(client_public_bytes).to_bytes(4, 'big'))
sock.sendall(client_public_bytes)
shared_key = private_key.exchange(server_public_key)
fernet_key = derive_fernet_key(shared_key)
return Fernet(fernet_key)
def bastion_client():
Connects to the Bastion Server, it logs in safely, does the key exchange and,
allows the user to send encrypted messages over a secured channel.
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
[Client] Server: Message received securely.
Message ('quit' to exit): Yes
[Client] Server: Message received securely.
Message ('quit' to exit): quit
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project>
PS C:\Users\ryafor01248\Desktop\EP2 Project> & C:/Users/ryafor01248/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ryafor01248/Desktop/EP2 Project/BastionClientFinal.py"
[Client] Connected securely to server.
[Client] LOGIN:
Username: Ryan
Password: Noroff
[Client] Login successful.
[Client] Secure session established.
Message ('quit' to exit):
[Client] Server: Message received securely.
Message ('quit' to exit): Hello
[Client] Server: Message received securely.
Message ('quit' to exit): hello
OUTLINE
TIMELINE
MYSQL
Ln 60, Col 38 Spaces: 4 UTF-8 CRLF Python 3.13.0 Prettier
Search
13:24
16.05.2025



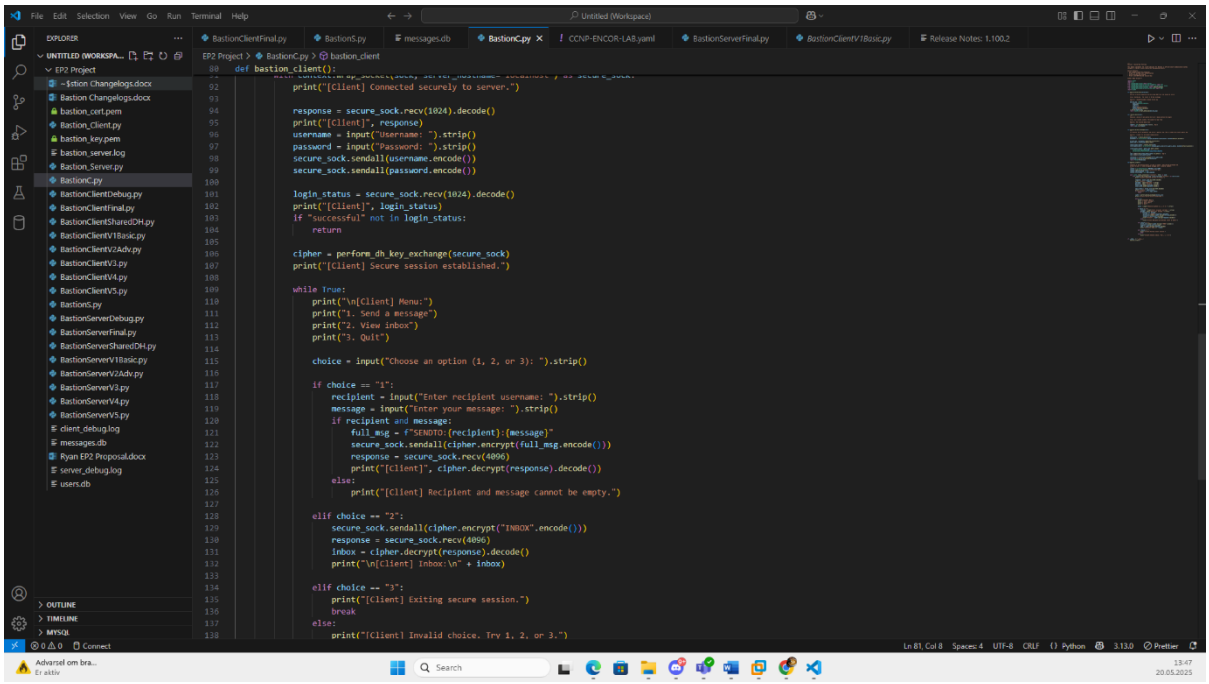


Bastion Client V8 (**BCV8**)

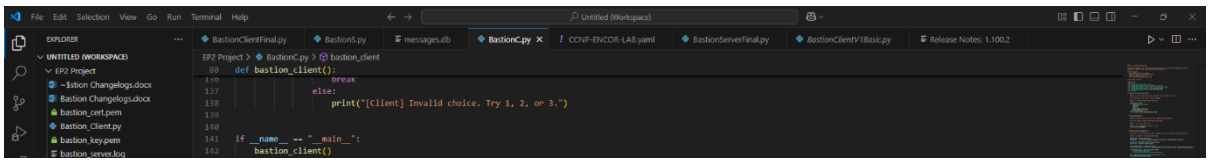




```python
def receive_block(sock):

    Returns: The received data block
    """
    length = int.from_bytes(sock.recv(4), 'big')
    return sock.recv(length)


def perform_dh_key_exchange(sock):
    """
    It receives the DH parameters and server's publick key, then it sends the clients public key

    Returns: A cipher for encrypted communication
    """
    param_bytes = receive_block(sock)
    parameters = serialization.load_pem_parameters(param_bytes, backend=default_backend())

    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()

    server_public_bytes = receive_block(sock)
    server_public_key = serialization.load_pem_public_key(server_public_bytes, backend=default_backend())

    client_public_bytes = public_key.public_bytes(
        serialization.Encoding.PEM,
        serialization.PublicFormat.SubjectPublicKeyInfo
    )
    sock.sendall(len(client_public_bytes).to_bytes(4, 'big'))
    sock.sendall(client_public_bytes)

    shared_key = private_key.exchange(server_public_key)
    fernet_key = derive_fernet_key(shared_key)
    return Fernet(fernet_key)

def bastion_client():
    """
    Connects to the Bastion Server, it logs in safely, does the key exchange and,
    allows the user to send encrypted messages over a secured channel.
    """
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.load_verify_locations("bastion_cert.pem")
    context.check_hostname = True
    context.verify_mode = ssl.CERT_REQUIRED

    with socket.create_connection(('localhost', 4443)) as sock:
        with context.wrap_socket(sock, server_hostname='localhost') as secure_sock:
            print("[Client] Connected securely to server.")
```

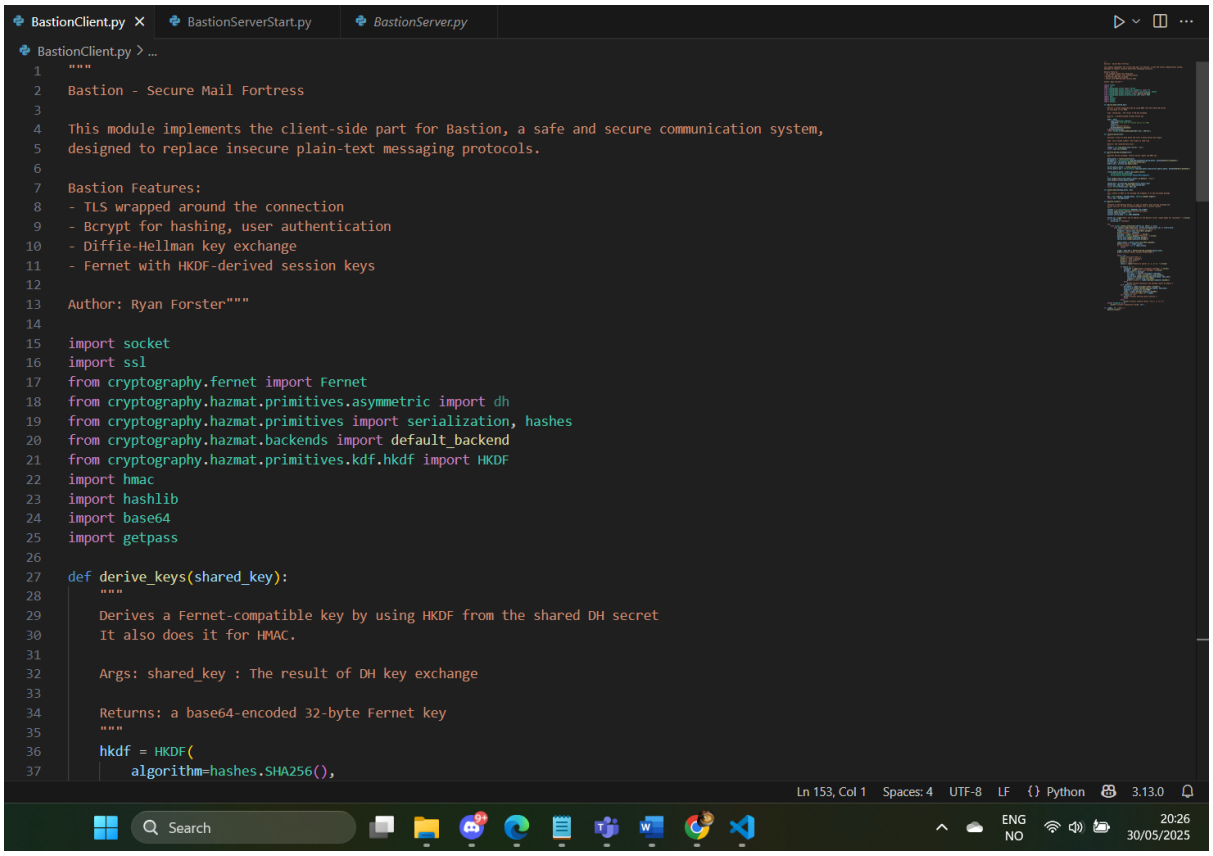


```python
            print("[Client] Connected securely to server.")

            response = secure_sock.recv(1024).decode()
            print("[Client]", response)
            username = input("Username: ").strip()
            password = input("Password: ").strip()
            secure_sock.sendall(username.encode())
            secure_sock.sendall(password.encode())

            login_status = secure_sock.recv(1024).decode()
            print("[Client]", login_status)
            if "successful" not in login_status:
                return

            cipher = perform_dh_key_exchange(secure_sock)
            print("[Client] Secure session established.")

            while True:
                print("\n[Client] Menu:")
                print("1. Send a message")
                print("2. View inbox")
                print("3. Quit")

                choice = input("Choose an option (1, 2, or 3): ").strip()

                if choice == "1":
                    recipient = input("Enter recipient username: ").strip()
                    message = input("Enter your message: ").strip()
                    if recipient and message:
                        full_msg = f"SENDTO:{recipient}:{message}"
                        secure_sock.sendall(cipher.encrypt(full_msg.encode()))
                        response = secure_sock.recv(4096)
                        print("[Client]", cipher.decrypt(response).decode())
                    else:
                        print("[Client] Recipient and message cannot be empty.")

                elif choice == "2":
                    secure_sock.sendall(cipher.encrypt("INBOX".encode()))
                    response = secure_sock.recv(4096)
                    inbox = cipher.decrypt(response).decode()
                    print("\n[Client] Inbox:\n" + inbox)

                elif choice == "3":
                    print("[Client] Exiting secure session.")
                    break
                else:
                    print("[Client] Invalid choice. Try 1, 2, or 3.")
```

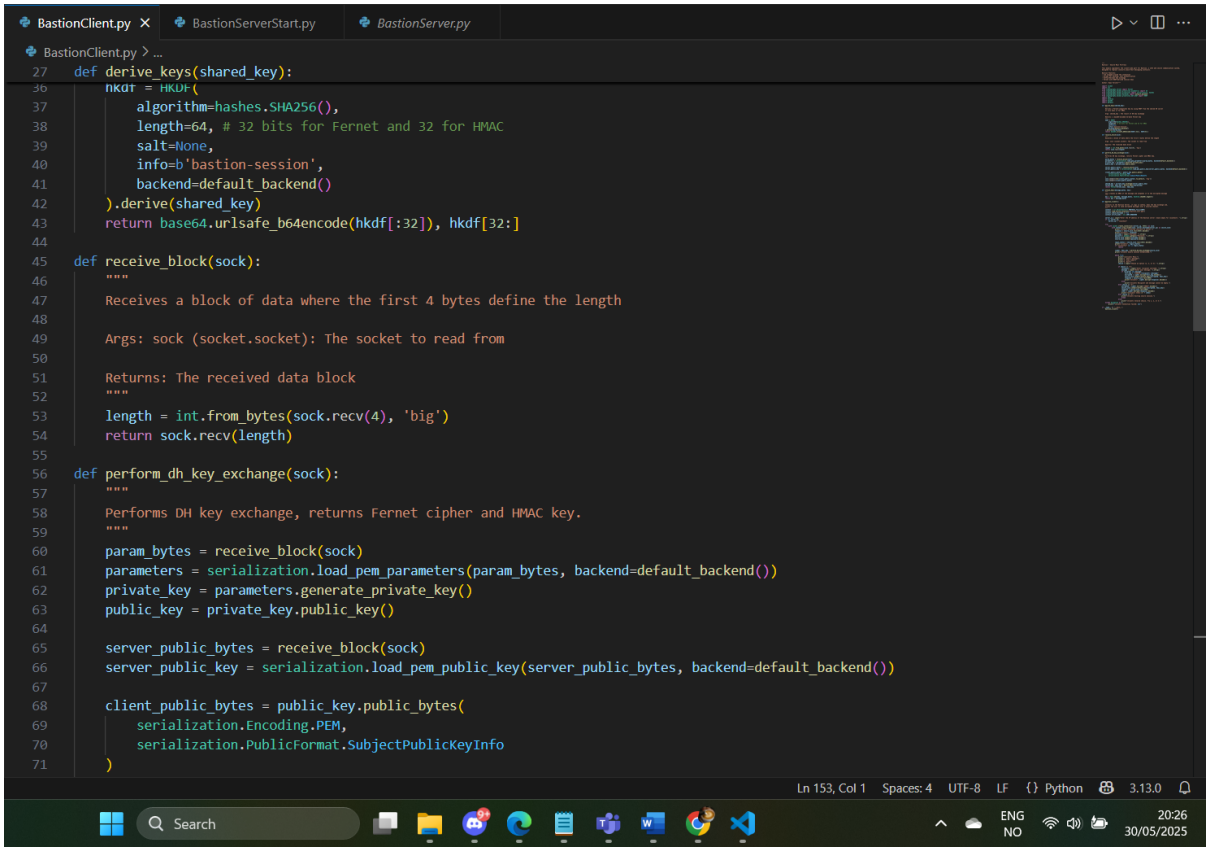


```python
                    break
                else:
                    print("[Client] Invalid choice. Try 1, 2, or 3.")


if __name__ == "__main__":
    bastion_client()
```

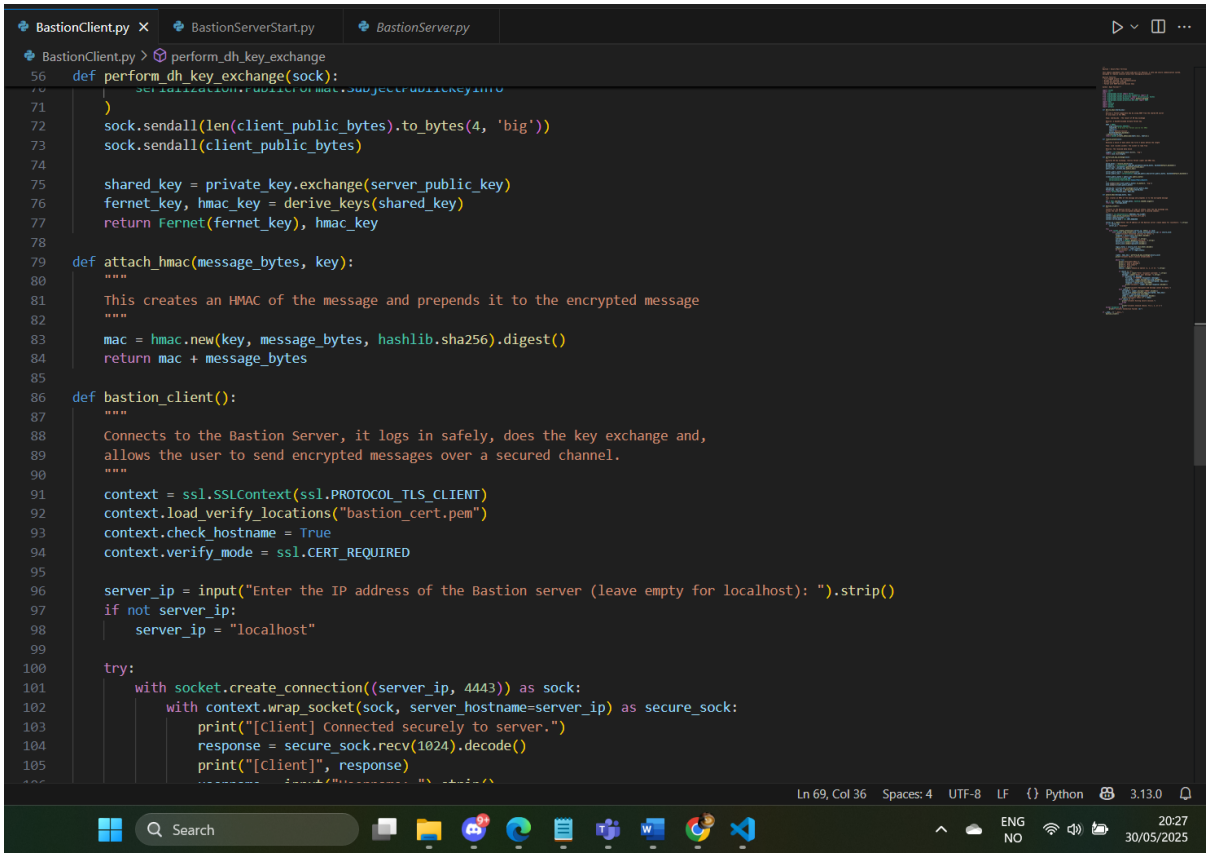
Bastion Client Final (**BCF**)

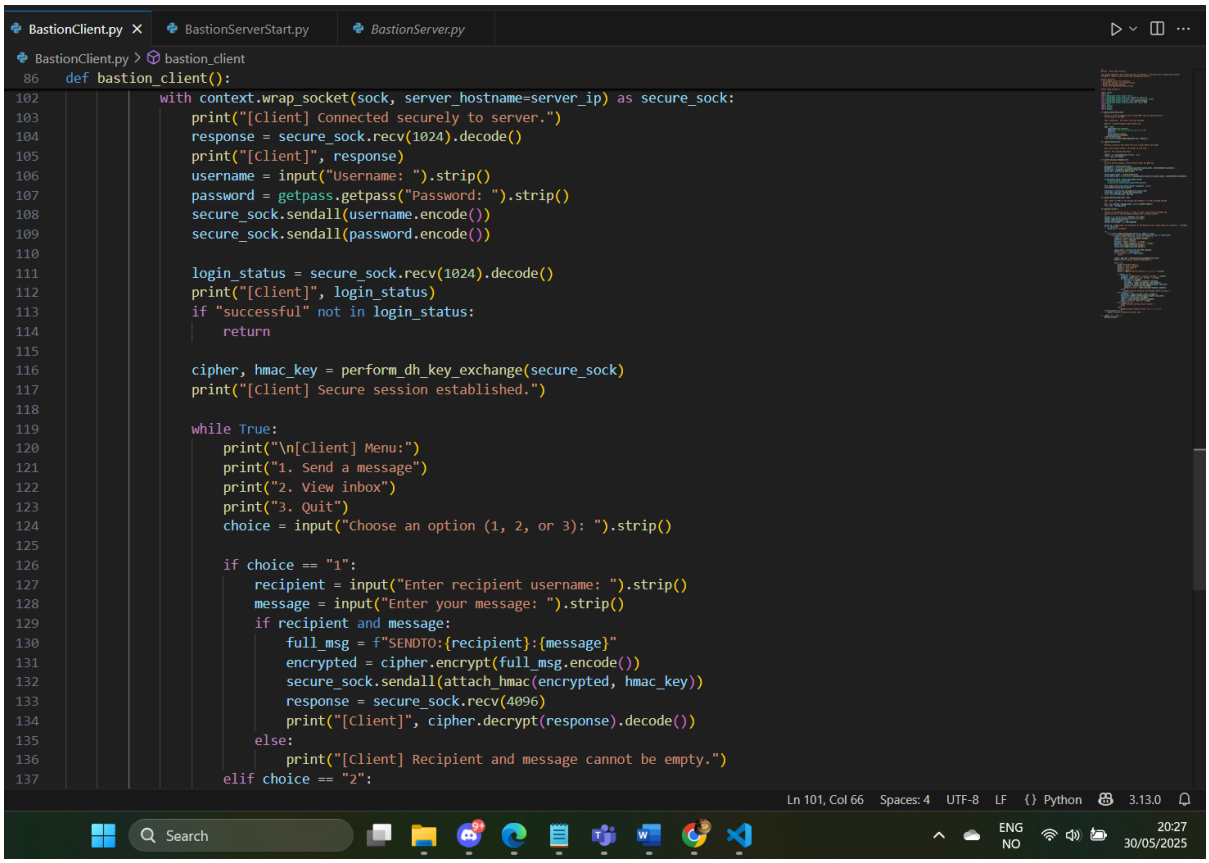


```python
"""
Bastion - Secure Mail Fortress

This module implements the client-side part for Bastion, a safe and secure communication system,
designed to replace insecure plain-text messaging protocols.

Bastion Features:
- TLS wrapped around the connection
- Bcrypt for hashing, user authentication
- Diffie-Hellman key exchange
- Fernet with HKDF-derived session keys

Author: Ryan Forster"""

import socket
import ssl
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import hmac
import hashlib
import base64
import getpass

def derive_keys(shared_key):
    """
    Derives a Fernet-compatible key by using HKDF from the shared DH secret
    It also does it for HMAC.

    Args: shared_key : The result of DH key exchange

    Returns: a base64-encoded 32-byte Fernet key
    """
    hkdf = HKDF(
        algorithm=hashes.SHA256(),
```



```python
def derive_keys(shared_key):
    hkdf = HKDF(
        algorithm=hashes.SHA256(),
        length=64, # 32 bits for Fernet and 32 for HMAC
        salt=None,
        info=b'bastion-session',
        backend=default_backend()
    ).derive(shared_key)
    return base64.urlsafe_b64encode(hkdf[:32]), hkdf[32:]

def receive_block(sock):
    """
    Receives a block of data where the first 4 bytes define the length

    Args: sock (socket.socket): The socket to read from

    Returns: The received data block
    """
    length = int.from_bytes(sock.recv(4), 'big')
    return sock.recv(length)

def perform_dh_key_exchange(sock):
    """
    Performs DH key exchange, returns Fernet cipher and HMAC key.
    """
    param_bytes = receive_block(sock)
    parameters = serialization.load_pem_parameters(param_bytes, backend=default_backend())
    private_key = parameters.generate_private_key()
    public_key = private_key.public_key()

    server_public_bytes = receive_block(sock)
    server_public_key = serialization.load_pem_public_key(server_public_bytes, backend=default_backend())

    client_public_bytes = public_key.public_bytes(
        serialization.Encoding.PEM,
        serialization.PublicFormat.SubjectPublicKeyInfo
    )
```


BastionClient.py
BastionServerStart.py
BastionServer.py
BastionClient.py > perform_dh_key_exchange
def perform_dh_key_exchange(sock):
)
sock.sendall(len(client_public_bytes).to_bytes(4, 'big'))
sock.sendall(client_public_bytes)
shared_key = private_key.exchange(server_public_key)
fernet_key, hmac_key = derive_keys(shared_key)
return Fernet(fernet_key), hmac_key
def attach_hmac(message_bytes, key):
"""
This creates an HMAC of the message and prepends it to the encrypted message
"""
mac = hmac.new(key, message_bytes, hashlib.sha256).digest()
return mac + message_bytes
def bastion_client():
"""
Connects to the Bastion Server, it logs in safely, does the key exchange and,
allows the user to send encrypted messages over a secured channel.
"""
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations("bastion_cert.pem")
context.check_hostname = True
context.verify_mode = ssl.CERT_REQUIRED
server_ip = input("Enter the IP address of the Bastion server (leave empty for localhost): ").strip()
if not server_ip:
server_ip = "localhost"
try:
with socket.create_connection((server_ip, 4443)) as sock:
with context.wrap_socket(sock, server_hostname=server_ip) as secure_sock:
print("[Client] Connected securely to server.")
response = secure_sock.recv(1024).decode()
print("[Client]", response)
Ln 69, Col 36 Spaces: 4 UTF-8 LF {} Python 3.13.0
Search
ENG
NO
20:27
30/05/2025


BastionClient.py
BastionServerStart.py
BastionServer.py
BastionClient.py > bastion_client
def bastion_client():
with context.wrap_socket(sock, server_hostname=server_ip) as secure_sock:
print("[Client] Connected securely to server.")
response = secure_sock.recv(1024).decode()
print("[Client]", response)
username = input("Username: ").strip()
password = getpass.getpass("Password: ").strip()
secure_sock.sendall(username.encode())
secure_sock.sendall(password.encode())
login_status = secure_sock.recv(1024).decode()
print("[Client]", login_status)
if "successful" not in login_status:
return
cipher, hmac_key = perform_dh_key_exchange(secure_sock)
print("[Client] Secure session established.")
while True:
print("\n[Client] Menu:")
print("1. Send a message")
print("2. View inbox")
print("3. Quit")
choice = input("Choose an option (1, 2, or 3): ").strip()
if choice == "1":
recipient = input("Enter recipient username: ").strip()
message = input("Enter your message: ").strip()
if recipient and message:
full_msg = f"SENDTO:{recipient}:{message}"
encrypted = cipher.encrypt(full_msg.encode())
secure_sock.sendall(attach_hmac(encrypted, hmac_key))
response = secure_sock.recv(4096)
print("[Client]", cipher.decrypt(response).decode())
else:
print("[Client] Recipient and message cannot be empty.")
elif choice == "2":
Ln 101, Col 66 Spaces: 4 UTF-8 LF {} Python 3.13.0
Search
ENG
NO
20:27
30/05/2025

```
BastionClient.py  BastionServerStart.py  BastionServer.py
BastionClient.py > bastion_client
86  def bastion_client():
                    print("[Client] Recipient and message cannot be empty.")
137             elif choice == "2":
138                 encrypted = cipher.encrypt("INBOX".encode())
139                 secure_sock.sendall(attach_hmac(encrypted, hmac_key))
140                 response = secure_sock.recv(4096)
141                 inbox = cipher.decrypt(response).decode()
142                 print("\n[Client] Inbox:\n" + inbox)
143             elif choice == "3":
144                 print("[Client] Exiting secure session.")
145                 break
146             else:
147                 print("[Client] Invalid choice. Try 1, 2, or 3.")
148     except Exception as e:
149         print(f"[Client] Connection failed: {e}")
150
151 if __name__ == "__main__":
152     bastion_client()
153
```

8.2 Bastion Testing

Bastion Client Certificate (False) (BCCF)

```
    return Fernet(fernet_key)

def bastion_client():
    """
    Connects to the Bastion Server, it logs in safely, does the key exchange and,
    allows the user to send encrypted messages over a secured channel.
    """
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.load_verify_locations("bastion_cert.pem")
    context.check_hostname = False
    context.verify_mode = ssl.CERT_REQUIRED

    with socket.create_connection(('localhost', 4443)) as sock:
        with context.wrap_socket(sock, server_hostname='localhost') as secure_sock:
            print("[Client] Connected securely to server.")
```

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

21°C Sol Search

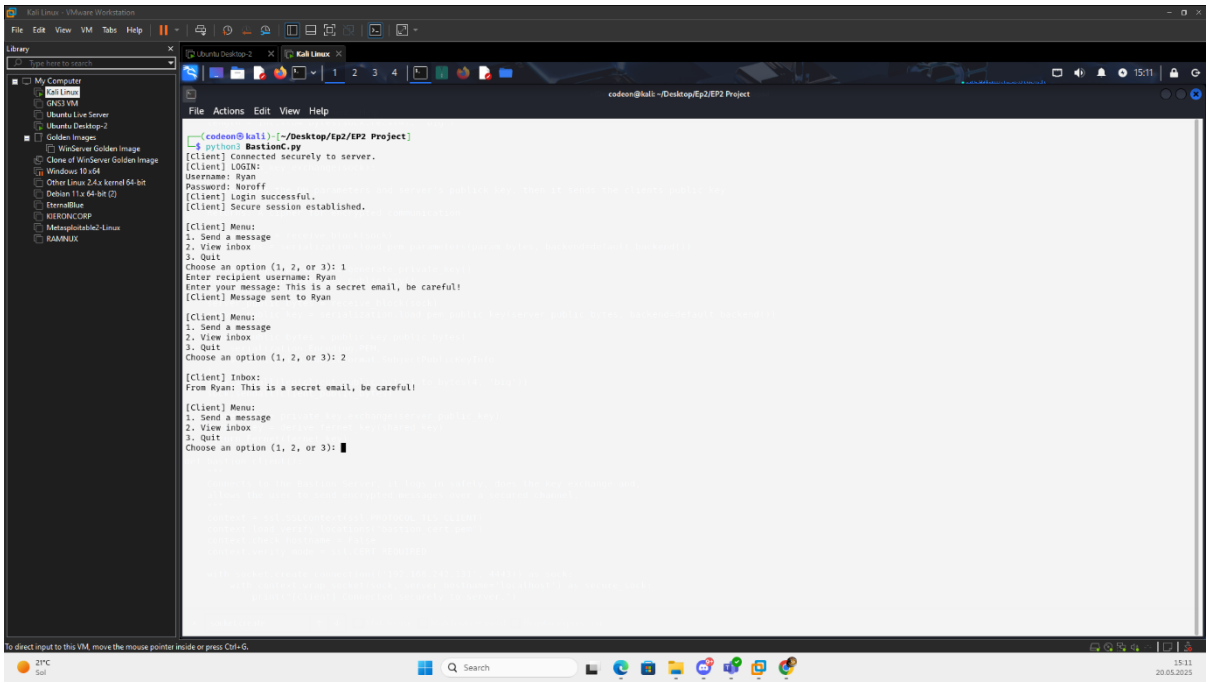
Bastion Client Certificate (TRUE) (BCCT)

```
    return Fernet(fernet_key)

def bastion_client():
    """
    Connects to the Bastion Server, it logs in safely, does the key exchange and,
    allows the user to send encrypted messages over a secured channel.
    """
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.load_verify_locations("bastion_cert.pem")
    context.check_hostname = True
    context.verify_mode = ssl.CERT_REQUIRED

    with socket.create_connection(('localhost', 4443)) as sock:
        with context.wrap_socket(sock, server_hostname='localhost') as secure_sock:
            print("[Client] Connected securely to server.")
```

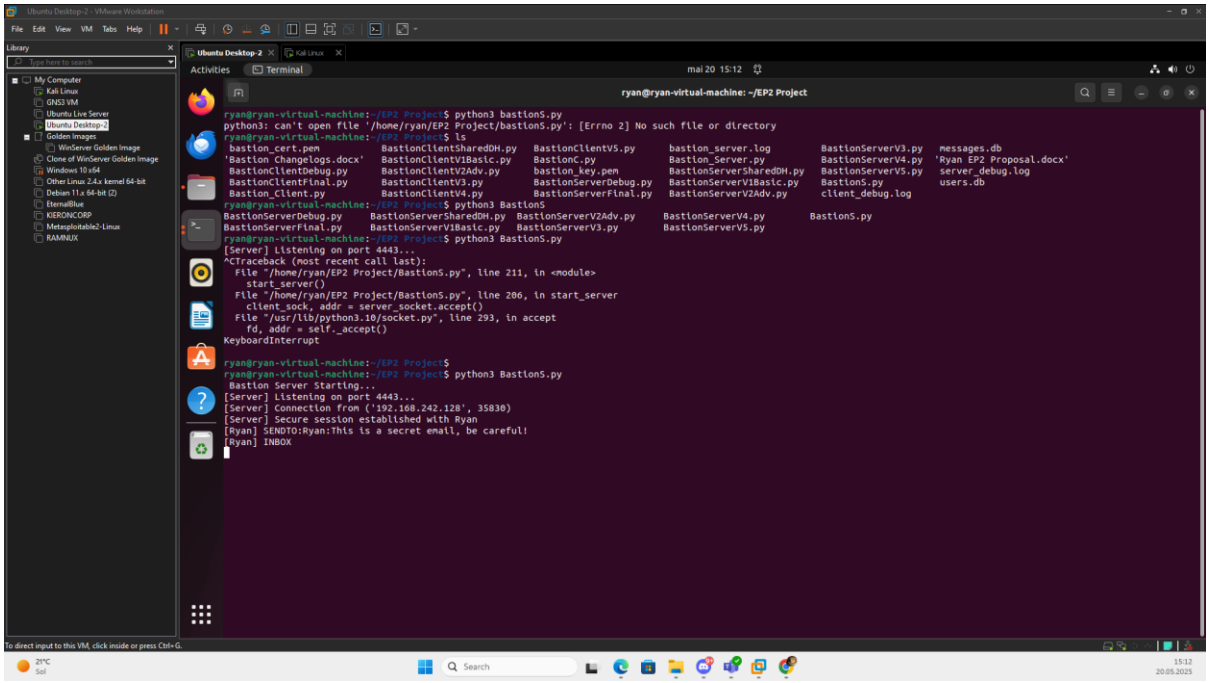
o direct input to this VM, move the mouse pointer inside or press Ctrl+G.

21°C Sol Search

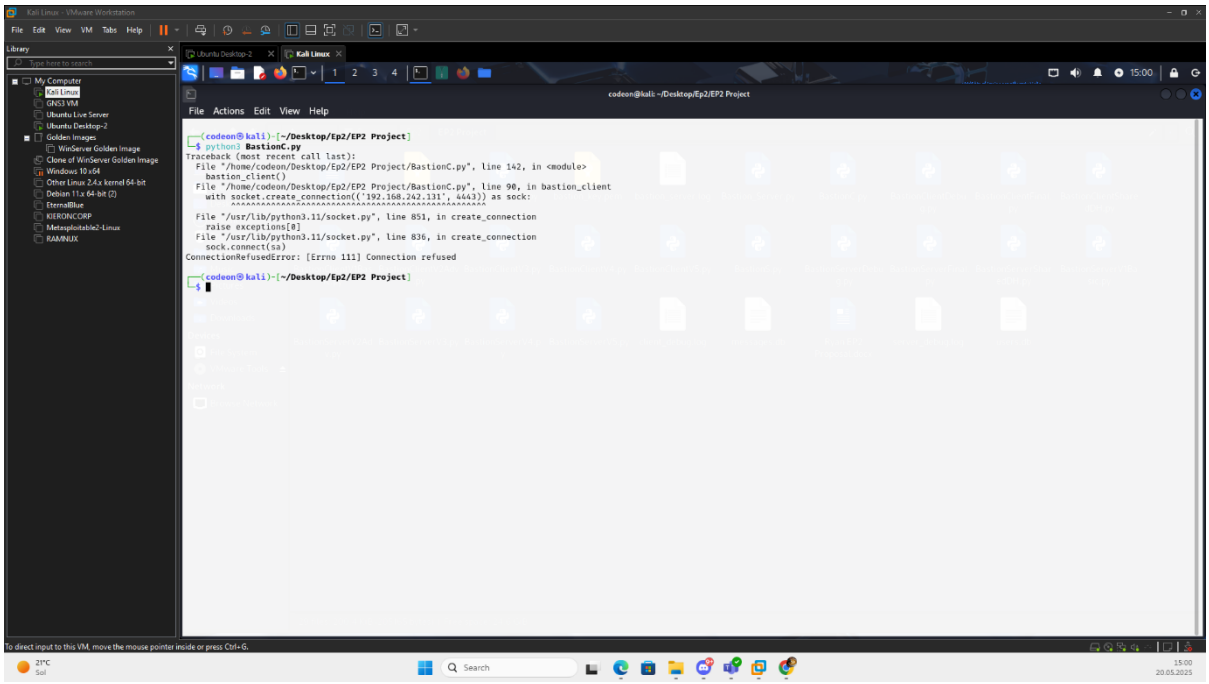
Bastion Client Basic Functionality (BCBF1)



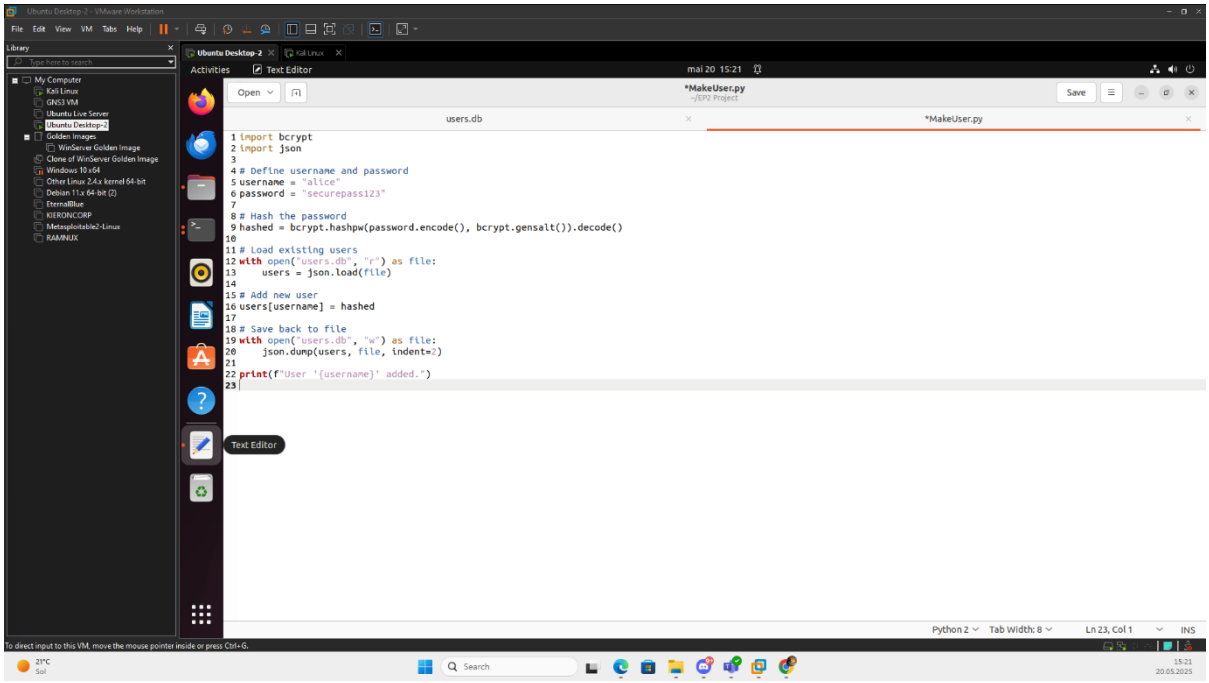
Bastion Server Basic Functionality (BSBF)



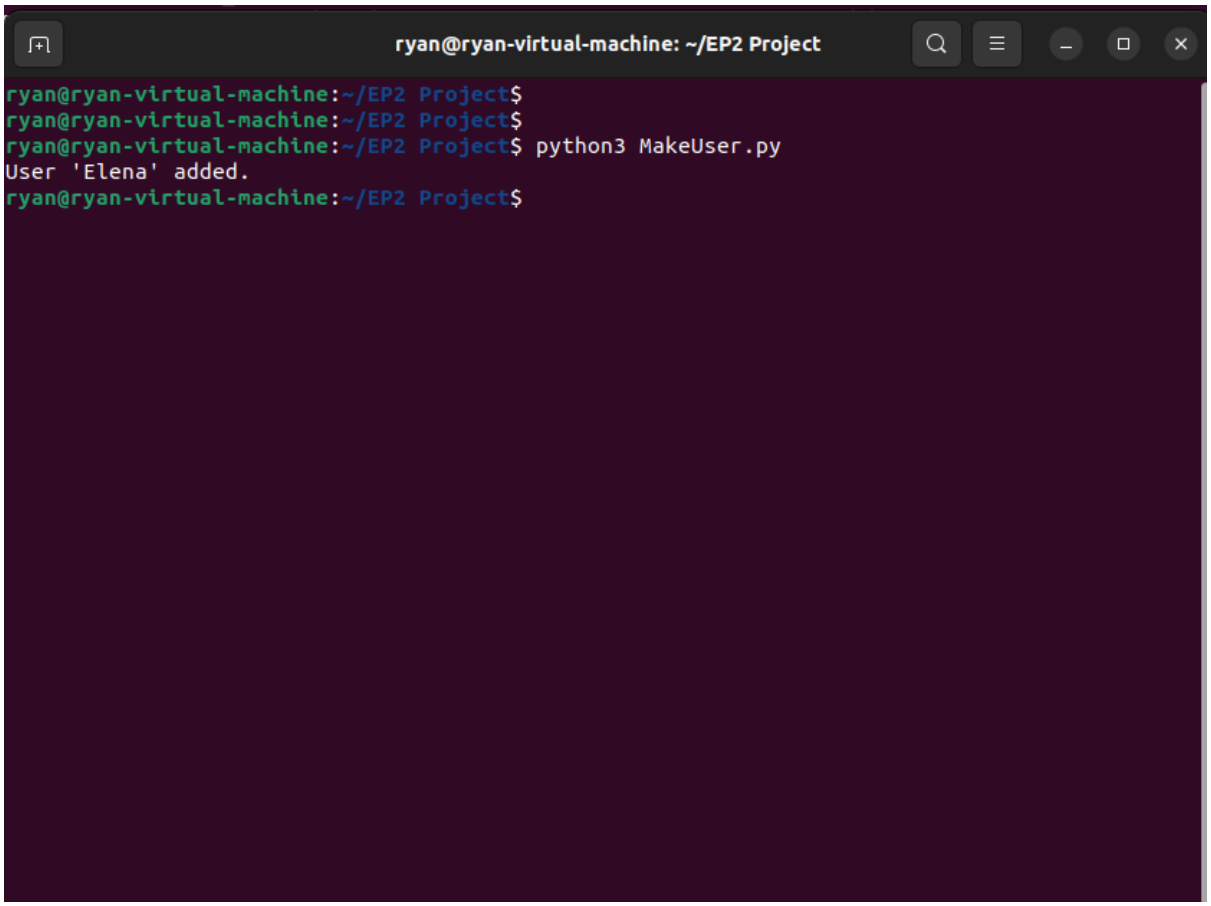
Bastion Client Connection Refused (BCCR)



Bastion Server Python Script to Add user (BSPS)



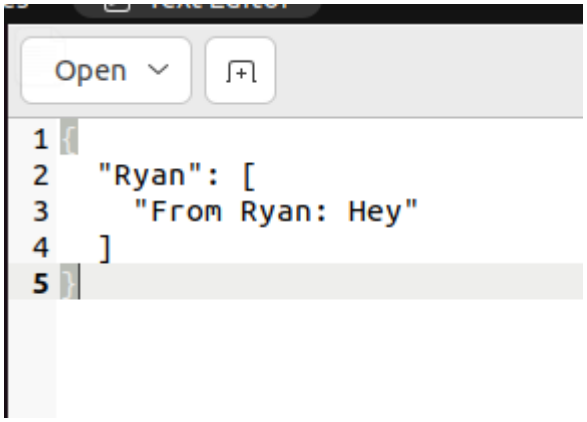
Bastion Server Running Add User Script (BSRA)



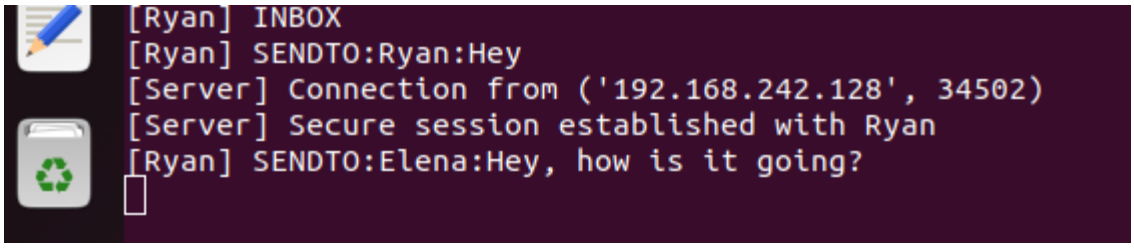
Bastion Client Wrong Username (BCWU)

```
┌──(codeon㉿kali)-[~/Desktop/Ep2/EP2 Project]
└─$ python3 BastionC.py
[Client] Connected securely to server.
[Client] LOGIN:
Username: Elea
Password: Cisco
[Client] Invalid username or password.
```

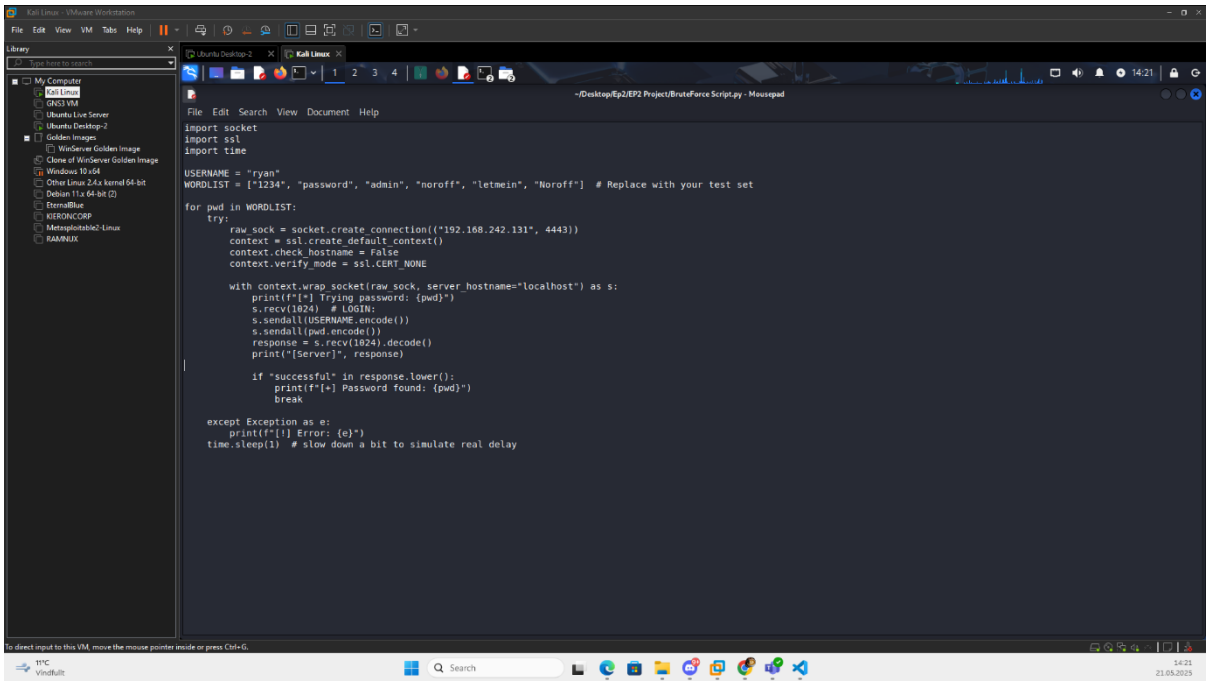
Bastion Server messages.db example (BSM)



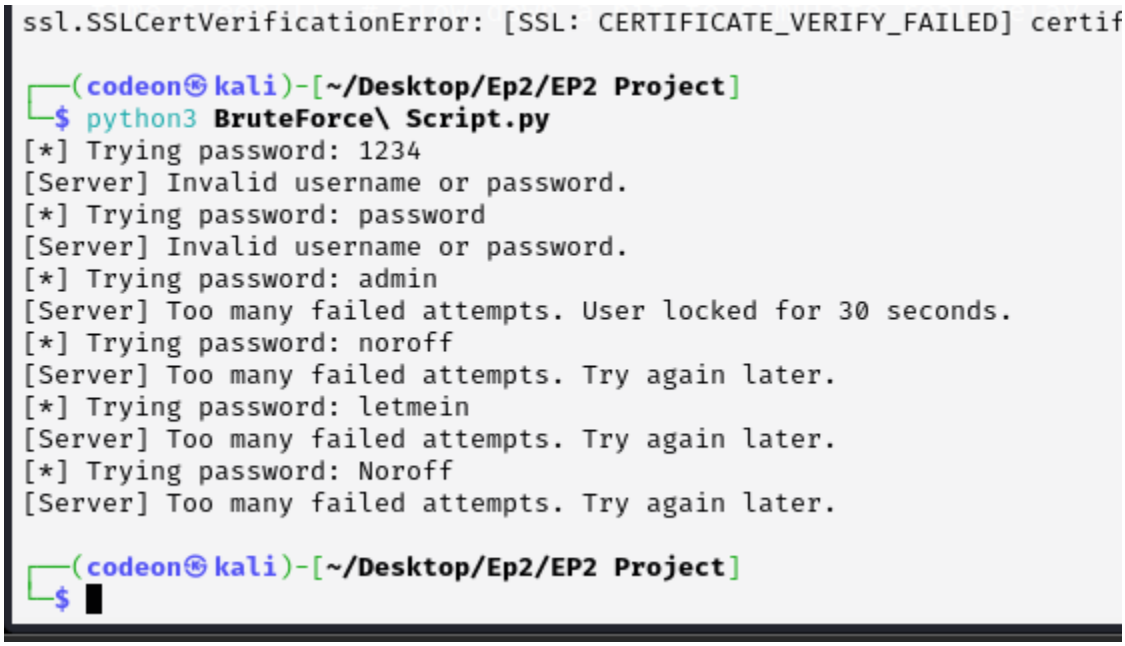
Bastion Server Message Confirmed (BSMC)



Bastion Client Brute Force Script (BCBF)



Bastion Client Brute Force Attack (BCBFA)



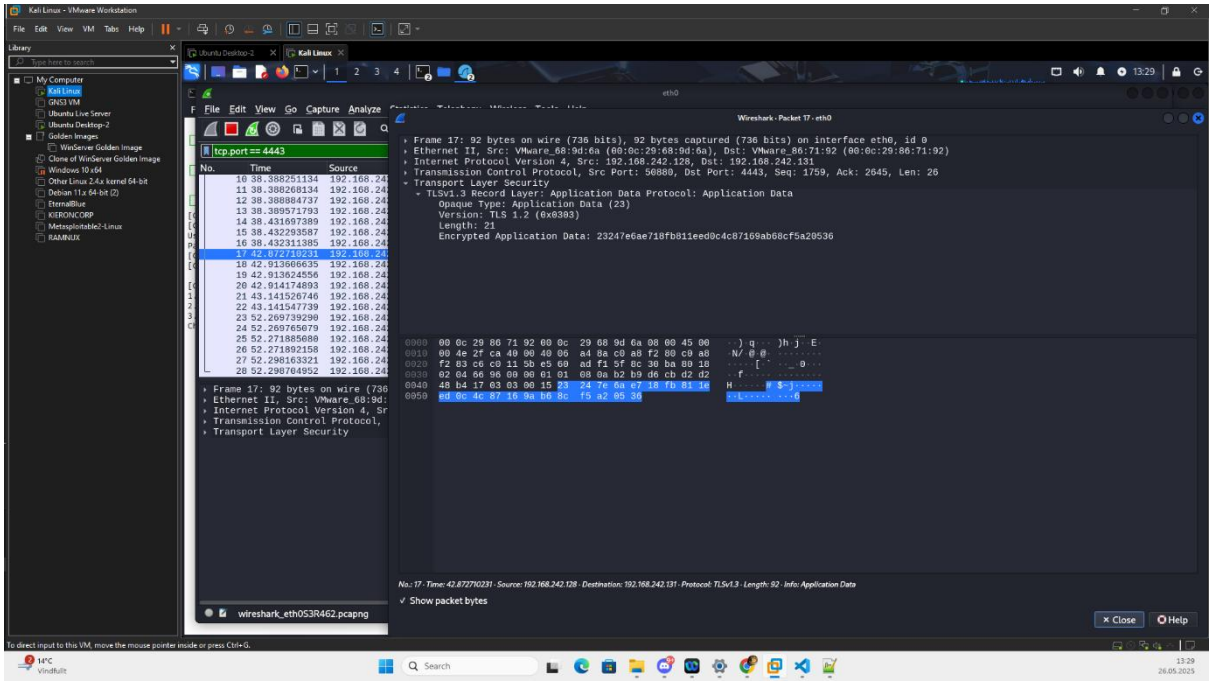
Bastion Server Down after brute force (BSBD)

```
ryan@ryan-virtual-machine:~/EP2 Project$ python3 BastionServerFinalVersion.py
[Server] Listening on port 4443...
[Server] Loaded locked users: []
[Server] Connection from ('192.168.242.128', 54284)
[Server] Secure session established with Ryan
[Server] Connection from ('192.168.242.128', 59662)
[Server] Secure session established with Elena
Traceback (most recent call last):
  File "/home/ryan/EP2 Project/BastionServerFinalVersion.py", line 294, in <module>
    start_server()
  File "/home/ryan/EP2 Project/BastionServerFinalVersion.py", line 290, in start_server
    secure_sock = context.wrap_socket(client_sock, server_side=True)
  File "/usr/lib/python3.10/ssl.py", line 513, in wrap_socket
    return self.sslsocket_class._create(
  File "/usr/lib/python3.10/ssl.py", line 1100, in _create
    self.do_handshake()
  File "/usr/lib/python3.10/ssl.py", line 1371, in do_handshake
    self._sslobj.do_handshake()
ssl.SSLError: [SSL: TLSV1_ALERT_UNKNOWN_CA] tlsv1 alert unknown ca (_ssl.c:1007)
ryan@ryan-virtual-machine:~/EP2 Project$
```

Bastion Server After Second Brute Force, survived (BSBD2)

```
ryan@ryan-virtual-machine:~/EP2 Project$ python3 BastionServerFinalVersion.py
[Server] Listening on port 4443...
[Server] Loaded locked users: []
[Server] TLS handshake failed from ('192.168.242.128', 43748): [SSL: TLSV1_ALERT_UNKNOWN_CA] tlsv1 alert unknown ca (_ssl.c:1007)
```

Bastion Client Wireshark Sniffing (BCWS)



8.3 Bastion – Secure Email Fortress

Bastion – Secure Mail Fortress (BSMF)