



C# SECURE CHAT SERVER

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document.]

When you're ready to add your content, just click here and start typing.]

Jessica Clara Fealy
18024092

| | |
|--|-----------|
| TABLE OF FIGURES | 2 |
| 1 INTRODUCTION | 3 |
| 1.1 Aim | 3 |
| 1.2 OBJECTIVES | 3 |
| 1.3 PROJECT DELIVERY | 3 |
| 1.3.1 PROJECT PLANNING | 3 |
| 1.3.1.1 Milestone one | 4 |
| 1.3.1.2 Milestone two | 4 |
| 1.3.1.3 Milestone three | 4 |
| 1.3.2 REQUIRED OUTCOME | 4 |
| 1.3.3 DESIRED OUTCOME | 5 |
| 2 BACKGROUND RESEARCH | 5 |
| 2.1 ACTORS, ATTACK VECTORS, AND MITIGATION | 5 |
| 2.2 ENCRYPTION | 5 |
| 2.2.1 DIFFERENT MODELS OF ENCRYPTION | 5 |
| 2.2.2 RSA AND AES | 6 |
| 2.2.2.1 RSA CRYPTOGRAPHY | 6 |
| 2.2.2.2 AES CRYPTOGRAPHY | 7 |
| 2.3 OTHER SECURITY FEATURES | 7 |
| 2.3.1 SSL / TLS | 7 |
| 2.3.2 AUTHENTICATION | 8 |
| 2.3.3 HASHING | 8 |
| 2.3.4 INPUT VALIDATION | 9 |
| 2.4 LIBRARIES AND OTHER SOFTWARE | 9 |
| 2.4.1 DEVELOPMENT TOOLS | 9 |
| 2.4.2 DATABASES | 10 |
| 2.4.3 LOGGING TOOLS | 10 |
| 3 PROTOTYPING | 10 |
| 3.1 SERVER | 10 |
| 3.1.1 THE EARLY SERVER | 11 |
| 3.1.2 THE EARLY CLIENT | 11 |
| 3.2 RSA ENCRYPTION | 11 |
| 3.3 RSA AND AES ENCRYPTION | 11 |
| 3.3 SECURE LOGGING | 12 |
| 3.4 HASH TESTING | 13 |
| 4 IMPLEMENTATION | 14 |
| 4.1 JSON MESSAGING SYSTEM | 15 |
| 4.2 RSA AND AES KEY EXCHANGE | 15 |

| | |
|---|-----------|
| 4.2 DATABASE AND PERSISTENT DATA STORAGE | 16 |
| 4.3 USER LOG IN AND ACCOUNT REGISTRATION | 16 |
| 4.4 SSL CONNECTION | 16 |
| 4.5 HASHING | 18 |
| 4.6 MULTITHREADING | 18 |
| 4.7 CLIENT HANDLER | 19 |
| 4.8 AES ENCRYPTION / DECRYPTION | 20 |
| 4.8.1 ENCRYPTION | 20 |
| 4.8.2 DECRYPTION | 21 |
| 4.9 AUTHENTICATION INFORMATION | 21 |
| 4.10 AUTHENTICATION | 22 |
| 4.11 LOGGING | 22 |
| 5 RESULTS | 23 |
| 6 FUTURE IMPROVEMENTS AND WORK | 23 |
| 7 CONCLUSION | 23 |
| 8 REFERENCES | 24 |
| 9 APPENDIX | 25 |
| ETHICS FORM | 25 |
| SOURCE CODE | 28 |
| SERVER | 29 |
| CLIENT | 29 |

Table of figures

| | |
|--|----|
| FIGURE 1 - MILESTONE ONE GANTT CHART | 4 |
| FIGURE 2 - MILESTONE TWO GANTT CHART | 4 |
| FIGURE 3 - MILESTONE THREE GANTT CHART | 5 |
| FIGURE 4 - RSA AES ENCRYPTION PROTOTYPE | 12 |
| FIGURE 5 - LOG4NET ROLLING FILE XML CONFIGURATOR PROTOTYPE | 12 |
| FIGURE 6 - READING FROM LOG4NET LOG FILE PROTOTYPE | 13 |
| FIGURE 7 - SALT GENERATION FUNCTION PROTOTYPE | 14 |
| FIGURE 8 - HASHING FUNCTION PROTOTYPE | 14 |
| FIGURE 9 - PROTOTYPE FOR TESTING HASHING | 14 |
| FIGURE 10 - HASHING TEST SHOW MATCH | 15 |
| FIGURE 11 - HASHING TEST SHOWING NON MATCH | 15 |
| FIGURE 12 - SSL CERTIFICATION CONSOLE GENERATION | 17 |
| FIGURE 13 - CONVERSION FROM CERTIFICATE TO PFX FILE | 18 |
| FIGURE 14 - AES ENCRYPTION | 21 |
| FIGURE 15 - AES DECRYPTION | 21 |
| FIGURE 16 - LOG4NET CONFIGURATION IMPLEMENTATION | 23 |

1 Introduction

1.1 Aim

The aim of this project is to research and determine the feasibility of creating a secure chat server using the C# .Net framework that provides both a secure and functional experience; and evaluating the feasibility of this both by the compliance to researched industry standard security features, and by the responses given by users on their experiences of using this project.

1.2 Objectives

1. To research various threats that a chat server, and other servers face daily to gain a complete understanding of the various methods and attack vectors that a malicious actor may take to breach server security.
2. To research encryption methodologies used in industry for message transmission and user data storage, this research will allow me to implement features to better secure data that is being sent to, sent from, and saved on the server.
3. To research pre-existing security technologies, protocols, and libraries that are either standard practice in industry, or are commonly used in industry. Doing this research will allow me to ensure that that final project better lies with what is standard, and to ensure that I am using libraries that will better help the overall security.
4. To design and implement the researched security features, practices, and protocols into a secure chat server using C#. The server will need to be able to securely protect against common attack vectors researched, and to also provide an easy to use client for message transmission and retrieval.
5. To test and evaluate the final solution of this project. After the server and clients have been finalised testing will be required to be under-taken to determine the success of the project. Evaluation of this testing and overall project will be completed.

1.3 Project Delivery

1.3.1 Project planning

This project has 3 distinct and defined milestone deadlines, each requiring specific work to be accomplished by their respective deadline. These milestone intervals are designed to ensure feedback can be given and corrections can be made at appropriate points to keep the project workflow as uninterrupted as possible. Alongside milestone deadline requirements, weekly meetings with the project supervisor will be conducted to ensure that intermediate deadlines and project transparency can be retained; these meetings will be designed to ensure issues can be raised and addressed in a frequent and timely manner. The deadlines required for each milestone deadline are as follows:

| TOPIC | DATE |
|-----------------|--------------------------------|
| MILESTONE ONE | 26 th November 2021 |
| MILESTONE TWO | 11 th February 2022 |
| MILESTONE THREE | 6 th May 2022 |

1.3.1.1 Milestone one

For the milestone one deadline the expectation is that all aims, and objectives will be completed, background research is expected to be complete, and the beginning of feature prototyping is expected to have started. Write ups for milestone one are expected to be rough drafts that will be expanded and finalised after the feedback in milestone 2. The Gantt chart for milestone one is as follows:



Figure 1 - Milestone one Gantt Chart

1.3.1.2 Milestone two

Milestone two will focus on finishing the development of prototype technologies and testing the performance of the prototypes; these prototypes will be written up and compiled into this report. Feedback will be received from milestone one submission, any recommended alternations made in the feedback will be appropriately made. Finally, the first implementations for the final product will be made in milestone two, these implementations will be written about prior to the milestone two submission. The Gantt chart for milestone two is as follows:

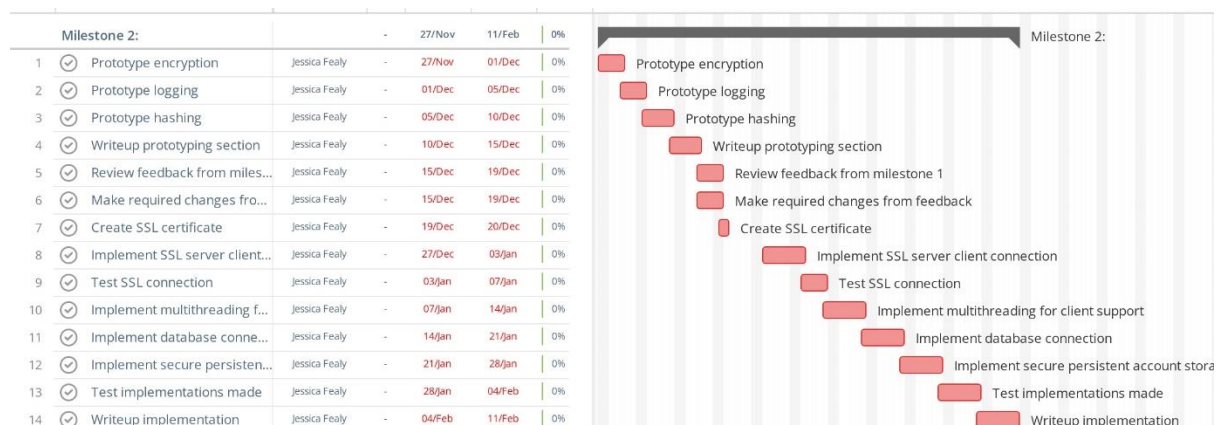


Figure 2 - Milestone two Gantt Chart

1.3.1.3 Milestone three

Milestone three is arguably the most important period, this is where all work is completed and finalised for submission. During this milestone project code implementation is to be completed, test, and written up. Then all user testing and critical analysis of the project success is required with mentions to potential improvements that could be made to the

project with future development, and finally conclude the project. The Gantt chart for milestone three is as follows:



Figure 3 - Milestone three Gantt chart

1.3.2 Project deliverables

Below is the required and desired outcome expected to be delivered by the completion of this project. These will be used to help evaluate the overall success for this project:

1.3.2 Required Outcome

- Research into encryption.
- Research into secure security, protocol, and libraries.
- Research into security threats.
- At least one chat channel per server instance.
- Persistent user log in data.
- Encrypted messages between client and server.
- Easy to use client interface.
- Implementation of other researched security features.
- Implement logging system.
- Server able to manage multiple clients simultaneously.
- Local admin server log in.

1.3.3 Desired Outcome

- Multiple chat channels on the same server instance.
- Remote admin server log in.
- Private client communication (one-on-one).
- Data stored securely to external database.
- Client messages stored locally securely.

2 Background Research

2.1 Actors, attack vectors, and mitigation

In terms of server security, a server is under threat by 3 main categories, human, nature, and technology (Jouini, Rabai and Aissa, 2014), in terms of this project only securing against the human and technological threats will be focused on, however with a larger scale solution nature threat would require more consideration.

This project will primarily focus on mitigating against these following vulnerabilities:

- Account hijacking – this is where a malicious actor gains unauthorised access to an authorised user account. This attack may occur against general users, or more seriously admin privileged users. There are many ways in which an attacker may gain access to an users account, this is primarily done via phishing or brute force attempts (Bamiah and Brohi, 2011). Way to mitigate against this form of attack, is by ensure users are aware of phishing attacks so they're better prepared for them and to ensure passwords are held to a standard to increase the time to brute force an attack.
- Man-in-the-middle (MITM) attacks – these attacks are caused when a malicious actor sits between the client and the server connection stream, this allows the actor to monitor, read, and interfere with messages sent between client and server (Swinhoe, D., 2021). A primary method to mitigate against MITM attacks is to set up a secure socket layer (SSL) between the server and the client (Irwin, 2021), this method ensures that both client and server can verify the message comes from the genuine sender.

2.2 Encryption

2.2.1 Different models of encryption

All encryption methods can be defined as either symmetric or asymmetric (Information Commissioners Office, n.d.), each category will give benefits and disadvantages depending on their use cases.

Symmetric cryptography is the method of using the same encryption key for both encryption and decryptions (Smirnoff and Turner, 2019). Symmetric can operated in two different modes, these are known as block ciphers and stream ciphers. Block ciphers segments all data (plain text) into static size blocks, these blocks have the encryption applied to them. Stream ciphers segments the data into separate bits, each individual bit then has the encryption applied (Panhwar, 2019). Symmetric encryption has a potentially fatal flaw, this regarding how the key is shared. This form of encryption has benefits over asymmetric methods by required less computation time for encryption and decryption, resulting in faster message transfer.

Asymmetric cryptography (also known as public key cryptography) is an encryption methodology that uses 2 related keys for data encryption and data decryption. These 2 keys are known as the private and public keys, the public key can be shared on the open internet as plain data – this key is used exclusively for data encryption. The private key is required to be kept secret and is responsible for data decryption. Asymmetric cryptography is a much more secure encryption process compared to symmetric option, while with this security comes a downside of the heavy computation time required to encrypt or decrypt data. However, due to the nature that asymmetric public key can be freely shared and used by third parties for data encryption there are no concerns with the secure distribution of keys as there is with symmetric variants (Simmons, 1979).

The speed advantages afforded to symmetric cryptography over asymmetric means while remaining a relatively high degree of security means that symmetric is the preferred method for bulk message encryption and decryption. Additionally, the primary flaw in symmetric cryptography being the ability to securely distribute their encryption keys can be answered by firstly using asymmetric cryptography. This can be done by machine A distributing their public asymmetric key, machine B encrypts their symmetric key using the public key form

machine A and then distribute the asymmetric encrypted message containing the symmetric key. Machine A can then decrypt the received message from machine B using the asymmetric private key, this decryption process will result with machine A having access to machine B's symmetric key. In this example the symmetric key has been securely transmitted to the desired location with only the person holding the private key with the ability to decipher the message (Anton, 2019).

2.2.2 RSA and AES

2.2.2.1 RSA Cryptography

Riverst-Shamir-Adleman (RSA) is an asymmetric encryption method that uses a key pair of a public and private key that are used for encryption and decryption of data. A public key used for data encryption; this key can be transferred over the internet as plain text with no security vulnerabilities arising. An RSA private key contains the information required to decrypt data; it is not considered secure to transfer an RSA private key in plain text over the internet. The process for the generation of an RSA key is as follows (Singh and Supriya, 2013):

1. RSA Modulus (n): the modulus is created from the multiplication of 2 large prime numbers with these prime numbers being labelled as q and p respectively. When multiplying prime numbers, the resultant value can only be factorised back into the original prime numbers, for smaller numbers a computer can relatively efficiently factorise to find q and p , however when larger prime numbers are used (+1024-bit prime numbers) this time to factorise too considerably longer.
2. RSA Derived number (e): This number is derived from the prime numbers used for the calculation of the modulus and uses the equation $(p - 1)(q - 1)$.
3. RSA Private key generation (d): This is the private key secret exponent and is calculated by using the Extended Euclidean algorithm which is stated as $d = e - 1 \bmod n$ (Zhou and Tang, 2011).
4. RSA Public key generation: The public key is derived from both the modulus and the derived number.

2.2.2.2 AES Cryptography

A common industry standard for symmetric encryption is known as the Advanced Encryption Standard (AES); where RSA cryptographic will use 2 keys either encryption or decryption, AES will use the same key and initialisation vector (IV) for both these processes. The IV within the AES is generated as a pseudo-random number and is used to ensure that the data is being encrypted / decrypted the same number of times, this security feature is to ensure that the same encrypted text containing the same data do not generate the same cipher text (Domingus, 2020). This means like all symmetric encryption both parties are required to know the key and IV to be able to understand messages sent between the pair, this raises security concerns of how to safely communicate these keys between the parties involved.

The AES cryptographic process follows 3 primary steps each with their own sub phases, these being (Heron, 2009):

- The initial round
 - During the initial round a XOR operation is performed between the plain text and the key.
- The intermediate round – this stage has 4 sub phases that are performed
 - Substitution byte

- Shift row
- Mix column
- Add round key

This step is repeated based on the key size of the AES key

- The final round
 - The final step does one last iteration of substitution byte, shift row, and add round key phases in the intermediate round.

2.3 Other security features

2.3.1 SSL / TLS

SSL (Secure Socket Layer) / TLS (Transport Layer Socket) is widely used in industry for secure communication across the internet, this protocol provides both the client and the server security, integrity, and authentication for all messages sent between them (SSL Support Team, 2021). For the process of connection SSL between a client and a server the client will begin by connecting to the server sending an initial connection message along with a list of cipher suites that is supported by the client, the server will respond to this message with an initial connection message, a selected cipher suite and a X.509 Certificate (Adeenze-Kangah, 2019). An X.509 certificate is a digital certificate that can give a client trust in ensuring a server's authenticity, an X.509 certificate will contain the server public encryption key and will contain the identity of the server. Using SSL and an X.509 certificate is a way to mitigate man-in-the middle attacks and gives the client certainty in the connection to the correct server.

This certificate can be generated in a two main way by using a Certificate Authority (CA) or by self-signing a certificate. Certificate Authority are the industry standard method to create X.509 certificates and are seen as the only secure way of doing so, this is because by using a certificate authority there is a trusted third party verifying the identity of the server. Where self-signing is not recommended to be used as there is not third party verifying the authenticity of the certificate. Self-signing is only recommended to be used in testing environments or on private networks (Sectigo, 2021). This project will be utilising self-sign X.509 certificates to initialise an SSL connection.

2.3.2 Authentication

A major aspect for this project is allowing for the infrastructure for a persistent user and admin login credentials. While there are libraries and third parties' solutions to managing this problem, an individual solution will be created for this project. The authentication system will function in a fashion where the sever never has access to the unencrypted password; to do this the server stores a username, an account salt, and a secure hash of the salted user password. When a client attempted to log into the chat server, they send their unique account name or code to the server, this is looked up in the server's database with the server replying with a generated session salt and the account salt. The client will salt the user password with the account salt and hash this, then again salt the password with the session salt and hash this again, the resultant is transmitted back to the server. The server will finally salt the received encrypted password with the session salt and hash this, then compared this final string with the stored salted password. Return a message to the client allowing or denying access to the server.

2.3.3 Hashing

While most secure system will implement practices to ensure difficulty for a malicious actor to breach any databases where confidential information is being stored, an assumption must be made that any malicious actor has unrestricted access to all systems and databases. With this assumption made security decisions must be made to ensure that even with access to the user databases they can retrieve no data that is critical to user account integrity.

For the protection of confidential data (in this case account passwords) it is common practice to implement hashing for the storage of persistent data (National Cyber Security Centre, 2018). Hashes are the result caused by using hash functions, these are one-way cryptographic algorithms that convert plain-text into a hash value; these functions need to be deterministic meaning with the same input the same output is provided. Many different hash functions exist and are often applied in industry all having their own advantages and disadvantages, the primary functions that will be compared in this report are the Secure Hash Algorithm 256 (SHA-256) and the Password-Based Key Derivation Function 2 (PBKDF2).

Going from plain text to hash value alone can result in some critical security vulnerabilities, a commonly used attack on hash values is called a rainbow table attack. In essence a rainbow table attack is a dictionary attack but optimised for hash values (Arias, 2019); by having commonly used passwords translated into hash values using common hashing algorithms, attacks can compare the rainbow table hash values to the server stored hash value to find any matching entries (Beyond Identity, n.d.). - If this process finds any matching entry, then the malicious actor has gained access to that account. Due to the efficiency that rainbow tables can compromise password authentication it is now common practice to incorporate a salt with the original value and then hash the salted password (a salt is a piece of randomly generated data that is near impossible to guess) (Arias, 2021). The inclusion of the salt makes rainbow attacks near impossible, meaning that attacks must rely purely on brute force attacks to breach security (LookingGlass, n.d.).

The two hashing functions mentioned, SHA-256 and PBKDF2, each have their advantages and disadvantages; however, the American National Institute of Standards and Technology (NIST) now recommends that all passwords be stored using a key-based hashing function such as PBKDF2 (NIST, 2017). In general, SHA-256 promotes a faster hashing algorithm for more efficient operations where PBKDF2 is designed to be slower and computationally difficult to generate hashes; for password security a hashing algorithm ideally should be slow, this is to increase the time it takes for a malicious actor to attempt to breach password security (Dharmadasa, 2017). For this reason, this program will be utilising a key-based hashing algorithm such as PBKDF2 to store passwords for persistent storage.

2.3.4 Input Validation

An attack vector for a malicious actor can be any text entry field available for user use, special concern for this attack vector must be taken into consideration due to the use of a database. A system lacking any form of input validation lays the risk of code injection, the biggest risk for this application is the potential for SQL injection. This form of code injection is where a malicious actor will attempt to enter SQL code to manipulate the back-end database, to protect against this form of attack all user inputs must be sanitised and validated to ensure that no entry is able to manipulate the database (William *et al*, 2006).

In recent months the improper implementation of input validation caused one of the worst exploits in recent years, this vulnerability was caused by the API Log4j. Log4j is used by many Java applications used this software to log events that happen within the system, this allowed the developers to better track any issues that would occur, CISA (Cybersecurity and Infrastructure Security Agency) officials stated that “potentially millions of devices are likely to be affected” (ABCNews, 2021). The vulnerability inside Log4j allowed for malicious actors to inject code within any system that used this logging software, this could mean that personal data could be stolen from databases and systems could have malicious software uploaded to them allowed the malicious actors to have a backdoor into that system (NCSC, 2022).

Another important factor that must be considered with any inputs is non-malicious user inputs, while most users will not intentionally attempt to cause malicious harm to the computer system human error must be taken into consideration. When enforcing a password policy any user given password when registering must be checked against this policy to ensure adherence. Additional when expecting data in certain formats such as emails, where you are expecting the usage of the ‘@’ character followed by an email service provider and then a domain. Emails entered must be validated against a criterion to ensure that emails entered are in the correct format, while the system cannot check whether the email is correct upon entry, the system can at least check whether the email address entered is within the correct formatting.

2.4 Libraries and other software

2.4.1 Development tools

The development of this project will occur using the Windows 10 operating system and Visual Studios Community Edition 2019 for the integrated development environment. The server will be using .Net Framework version 4.7.2, and the client created with .Net Framework 4.8 using Window Forms as they graphical user interface tool. These frameworks are specified to the Windows operating system and are unable to allow cross compatibility to Linux or Mac OS systems (Microsoft, 2022).

2.4.2 Databases

Databases will be heavily integrated into the development of this project; they will provide a vital role in the security of log in credentials and general logged data. The intention for this project is to have no sensitive information permanently stored on the computer that is hosting the server, doing so poses a risk to that data being stolen; instead, all sensitive data will be stored externally on a third-party server to help mitigate any risks. The chosen third party to host his database will be MongoDB, MongoDB provides a service that can be easily integrated with object-oriented programs, and uses a JSON message format, the same in which will be used for message communications between the server and clients (MongoDB, n.d).

2.4.3 Logging tools

In software development logging provide a critical service (TowardsDataScience, 2020); this is by keeping a record of all activities taken place on a system, this can track any errors, security issues, or general information. Such logging information can be vital when trying to pinpoint any issues or to find any security flaws. This system will utilise try and catch blacks around any areas of critical code that can fail, such examples can be data type conversions and input validations, if these areas of code fail the catch block will catch the exceptions and

a log will be generated providing a time stamp, location of error, and the error thrown. Additional logs entries will be created around all log in attempts (admin and users) and when any new accounts are added, these logs will be used to track the security of the programs and will allow the admins to view where a brute force attack has taken place.

Logging for this program will be handled by external logging libraries, the libraries being considered are Log4Net (Watson, 2019) and NLog (NLog, n.d). A few primary differences exist between Log4Net and NLog, primarily the set up for each with the former usually be configured using XML with limited ability to be programmatically configured, where NLog can be easily configured programmatically. Each logging libraries offers a similar set of logging functions, allows for logs to be made in a variety of severity levels and for logs to be made in multiple locations at once (Console and text files). Log4Net is the chosen logging library that is going to be used and implemented into this project, with the ability to perform thread safe operations and write logs to various locations at once; while in this project the logged data will remain to console or to a file future development may want to be able to send logging event to a database and Log4Net allows this via the appenders (Apache, n.d).

3 Prototyping

GitHub repo link: <https://github.com/JeCFE/Final-Year-Project>

3.1 Server

Early prototyping for this project consisted of creating an initial server that can communicate to several clients at the same time. When a client send a message to the server the server would respond to that client with the message of "Received ". This prototype was created to ensure that multiple clients can connect to the server, with the server broadcasting a message to only the client that initiated communication.

3.1.1 The Early Server

The server has a hard coded IP address and port number initialised to the hosting PC, in future iterations this will be adopted for the IP and port numbers to be assigned by an admin at run time for the server, this should also be adopted to allow the ability to save these settings. No multithreading has been used in this iteration with an infinite while loop being implemented to constantly listen to new clients connecting to the server, and in the same loop to listen and send messages to relevant clients.

3.1.2 The Early Client

The initial client, knowing that the server has consistent connection settings, would automatically connect to the client on run time. Unlike the server, the client has used multithreading to allow for listening to messages form the server, delegates have been used to communicate the messages received to outside the separate thread and update the relevant label.

3.2 RSA encryption

Early developments using RSA encryption was to encrypt and entire message stream using different functions to simulate the communication streams between server and client. Numerous issues occurred during this stage of prototyping, primarily RSA's difficulty to encrypt long data streams. Further research committed into industry standard encryption

methods concluded in the using RSA to encrypt an AES key for key sharing, then using AES encryption for all messages between client and server.

3.3 RSA and AES encryption

For the prototype, both an AES key and RSA key is generated, to simulate and validate the potential of encrypting an AES key using RSA on the server and communicating this to the client for the client to decrypt the AES key using RSA for all message communication. An AES key was both encrypted and decrypted using RSA before attempting to decrypt a message to ensure that is no loss in data or ability for encryption through this process.

```
var csp = new RSACryptoServiceProvider(2048);
var privKey = csp.ExportParameters(true);
var pubKey = csp.ExportParameters(false);

Console.WriteLine("INPUT MESSAGE");
string message = Console.ReadLine();
Aes aesKey = Aes.Create();
Console.WriteLine(System.Text.Encoding.Default.GetString(aesKey.Key));
byte[] encryptedAESKey = csp.Encrypt(aesKey.Key, RSAEncryptionPadding.Pkcs1);
byte[] encryptedAESIV = csp.Encrypt(aesKey.IV, RSAEncryptionPadding.Pkcs1);
string strEAesKey = System.Text.Encoding.Default.GetString(encryptedAESKey);
string strEAesIV = System.Text.Encoding.Default.GetString(encryptedAESIV);
Console.WriteLine(strEAesKey);
Console.WriteLine(strEAesIV);

byte[] decryptedAESKey = csp.Decrypt(encryptedAESKey, RSAEncryptionPadding.Pkcs1);
byte[] decryptedAESIV = csp.Decrypt(encryptedAESIV, RSAEncryptionPadding.Pkcs1);
string strDAesKey = System.Text.Encoding.Default.GetString(decryptedAESKey);
string strDAesIV = System.Text.Encoding.Default.GetString(decryptedAESIV);
Console.WriteLine(strDAesKey);
Console.WriteLine(strDAesIV);
byte[] encrypted = EncryptStringToBytes_Aes(message, aesKey.Key, aesKey.IV);

string plainText = DecryptStringFromBytes_Aes(encrypted, aesKey.Key, aesKey.IV);
Console.WriteLine(plainText);
```

Figure 4 - RSA AES Encryption Prototype

3.3 Secure Logging

A prototype logging system was created to test race conditions when a read request is performed to a file that is being actively written to by Log4Net. The tests performed had 10 threads committing 100000 logs to the same text files concurrently with user random log reads requests being committed, these tests showed that there was no data loss in the logs when the user was reading from the file.

For these tests to happen a Log4Net logger was created in XML with the following statements:

```
<appender name="RollingFile" type="log4net.Appender.RollingFileAppender">
  <file value="logs/Log_" />
  <datePattern value="dd-MM-yyyy'.txt'"/>
  <appendToFile value="true" />
  <staticLogFileName value="false"/>
  <lockingModel type="log4net.Appender.FileAppender+InterProcessLock"/>
  <layout type="log4net.Layout.PatternLayout">
    <conversionPattern value="%date [%thread] %-5level %logger - %message%newline"/>
  </layout>
</appender>
```

Figure 5 - Log4Net Rolling File XML Configurator Prototype

Where the XML file highlights the lockingModel using type InterProcessLock, this line ensure that the logger has exclusive access to writing to the file for each write attempts, once the logger has finished writing the lock is released. While the write access to the file has been lock whenever the logger is writing to the file, the read access has been left unlocked this allows the program to read from the file whilst the logger continues to write. For the read access from the file the following code was implemented:

```
try
{
    using (FileStream fileStream = new FileStream(
        "..\\Debug\\logs\\Log_27-01-2022.txt",
        FileMode.Open,
        FileAccess.Read,
        FileShare.Write))
    {
        using (StreamReader streamReader = new StreamReader(fileStream))
        {
            log.Info("Begin");
            Console.WriteLine("-----");
            Console.WriteLine(streamReader.ReadToEnd());
            Console.WriteLine("-----");
            log.Info("End");
            streamReader.Close();
        }
        fileStream.Close();
    }
}
```

Figure 6 - Reading From Log4Net Log File Prototype

Using FileStream allows the files to be locked in limited use, this statement opens the file, set the read access to read only and then allows the files to be continuously written to by the logging system. Through testing there is no data loss or integrity issues when read and writing to a file at the same time.

3.4 Hash Testing

For the final development of the chat server all account passwords will be salted and hashed for both transmission and persistent storage of these passwords. This following code was

created to test and simulate the salting and hashing process that was later incorporated onto the final solution. This test code will attempt to follow the authentication process highlighted in section 2.3.2 Authentication of this report. The process of this code is to first generate 2 random salt values named as accountSalt and sessionSalt; The user is then asked to enter a string password, this password and accountSalt is then hashed using the function called Hasher, this function uses a PBKDF2 hashing function using 10000 iterations, the return hash value is return and called persistantFirstHash (in real application this is the hash value that is stored to a database). This persistantFirstHash and sessionSalt then uses the Hasher function again with the resultant being called persistantFinalHash (on the server this hash value will be the hash valued used for authenticating user log in). The next stages asks for the user for the password again, this is to simulate the user logging into an account with these new hashes being used to authenticate the user. The new hashes go through the same stages as the persistent hashes, using the same hash values the plain text password is hashed once with the accountSalt and then the result is hashed again using the sessionSalt. In application the loginFinalhash will be transmitted to the server for comparison to the persistantFinalHash, in this test code a simple if statement is used to compare these two values. The code below shows the test code solution:

```
2 references
public static byte[] GenSalt()
{
    RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
    byte[] salt = new byte[128];
    provider.GetBytes(salt);
    return salt;
}
4 references
public static string Hasher(string message, byte[] salt)
{
    Rfc2898DeriveBytes hash = new Rfc2898DeriveBytes(message, salt, 10000);
    return Convert.ToBase64String(hash.GetBytes(128));
}
```

Figure 7 - Salt Generation Function Prototype

```
public static string Hasher(string message, byte[] salt)
{
    Rfc2898DeriveBytes hash = new Rfc2898DeriveBytes(message, salt, 10000);
    return Convert.ToBase64String(hash.GetBytes(128));
}
```

Figure 8 - Hashing Function Prototype

```

References
static void Main(string[] args)
{
    byte[] accountSalt = GenSalt();
    byte[] sessionSalt = GenSalt();
    Console.WriteLine("Enter password");
    string password = Console.ReadLine();
    string persistantFirstHash = Hasher(password, accountSalt);
    string persistantFinalHash = Hasher(persistantFirstHash, sessionSalt);
    Console.WriteLine("Password hash: " + persistantFinalHash);
    Console.WriteLine("Enter password");
    string loginPassword = Console.ReadLine();
    string loginFirstHash = Hasher(loginPassword, accountSalt);
    string loginFinalHash = Hasher(loginFirstHash, sessionSalt);
    Console.WriteLine("Test hash: " + loginFinalHash);
    if (persistantFinalHash == loginFinalHash){ Console.WriteLine("Match"); }
    else{ Console.WriteLine("Non Match"); }
    Console.ReadKey();
}
}

```

Figure 9 - Prototype For Testing Hashing

The below test cases show first when the passwords entered match, and then again when the password entered do not match:

```

Enter password
Testcase1
Password hash: DHa6g1u0HIB9LE1BjK500Iz/jYzx6V2cKTSytRePXkdm1MRvf4pok6uGYOT7fsiPDxvVhBiTDHw/kN9ux3QyLfZ7aj+fmQGIYOIFTE3qbBGL+B5ZhnEBBedsnDXRYqaDklfBaD9YT0G5eWxNIreM2yMgJzQDRiabbKYMymPTw=
Enter password
Testcase1
Test hash: DHa6g1u0HIB9LE1BjK500Iz/jYzx6V2cKTSytRePXkdm1MRvf4pok6uGYOT7fsiPDxvVhBiTDHw/kN9ux3QyLfZ7aj+fmQGIYOIFTE3qbBGL+B5ZhnEBBedsnDXRYqaDklfBaD9YT0G5eWxNIreM2yMgJzQDRiabbKYMymPTw=
Match

```

Figure 10 - Hashing Test Show Match

```

Enter password
Testcase2Password
Password hash: T0D0t7zsUuaqAozjNcv+HUC9fCK8hdC8SbaeNPRUYE1bR1IABjeIPAYh1Z0BqDJWvXoMjM0Bd9CCu++pwAGWMC9L90ayUbJpKc6u3ZzsWhiX4kCyz6/47vR03U+zJ5BZ111vixp01btw+sz6K5YUM6BcbdFdDuRHI1FKLUGH0M=
Enter password
Testcase2IncorrectPassword
Test hash: aI/Y4xf2ZZ2EDRDGG3BV8s4Xkd196lUsewa0gkcHrKbRpwSyzwvGn1CwbG62mh0/N4Z101po09Xdn1M87smZVBH5rjrL87gMoIIBNPG0+1X6iA6fiXvwQ/fkPiZwJrt5Ev5dnqly13o1NyU6tvwbW3jJG3DK0TOuldFiGnySdY=
Non Match

```

Figure 11 - Hashing Test Showing Non Match

4 Implementation

The implementation of the completed C# secure chat server can be considered as the second iteration of development, each prototyped technology will be adopted and implemented into a seamless completed project. Each prototyped stage has been tested in a closed system with no interaction with other sections of prototyped technology. With each implemented prototype into the second iteration of development further testing will be completed to ensure that the adoptions have been implemented correctly before additional sections of prototypes are incorporated. The development will between server and client will be done concurrently due the nature of testing messaging and encryption infrastructure.

4.1 JSON messaging system

During early stages of development an issue was raised where not all messages sent between server and client were intended to be distributed to all connected clients and were instead intended for direct communication of data or instructions between server and client. Such types of communication include a need for a recognised encryption handshake, login request, registration request, and general messaging. On both server and client contains classes for the communication types mentioned before, these will be used serialising and deserialising JSON data into a string. Another master JSON class exists called Message. This class contains 2 string variables, one called id and the other called message; id is used refer to the which communication type is being sent, such as id 0 will refer the message contain information required for the encryption handshake. The message variable is used to store the JSON string from the serialisation from one of the other classes, knowing that id 0 means that message is for handshaking the message variable can be deserialised into the HandShakeMessage class seamlessly. This messaging system allows for future growth and additions when needed, with the only changes being required for a new addition is the addition of another class to handle the data for JSON serialisation and deserialization.

4.2 RSA and AES key exchange

Under common systems both server and client will already know the RSA keys used for encryption via the certification used during the SSL handshake, however due to the nature of this certificate nothing signed by a certification authority the decision was made to create a custom secure handshaking protocol whilst still using the initialised SSL tunnel. The protocol required the server to generate 2048-bit RSA key, this has been created and stored within an authentication class to separated RSA encryption from the ClientHandler class. The client would be responsible for the creation of their unique AES key and IV. Some issues arise from the implementation from the prototype RSA AES exchange to the chat server, this error existed when trying to convert the RSA public key into an XML string for transmission to the client, when the server attempted to decrypt messages sent from the client (when the client used import the RSA key from XML) there was an issue with a mismatch of keys. This error was solved simply by not converting the RSA public key into an XML string and rather transmitting the raw byte data instead the lack of conversion back and forth being byte and string solved the errors being experienced. The general transmission used in the key exchange used the JSON messaging system and the HandShakeMessage. Upon a client connecting the server, the server would send the client the RSA public key, the client would then encrypt their AES key and IV storing this data into the HandShakeMessage class, the client would also then encrypt a common word known between both client and server using AES. The server upon receiving the message from the client would decrypt the AES key and IV using the RSA private key and store the AES data inside the client handler class, the server would then decrypt the test message using the AES decryption, if the word matches what was expected the server will send the client confirmation, else the server would send the client notification that the process had not worked.

4.2 Database and persistent data storage

The database integration was not prototyped due to the database provider MongoDB provide supporting material which instructions on how to connect a C# program to the database server. This project uses 2 separate tables for data storage, one named Accounts and is responsible for storage of general user log in details, this includes a name or nickname, an account salt, last logged in, and a hash value of the account password. The other table used

is called Admin, this table contains information pertinent to the login for admin users, this table include admin id codes, last logged in, and hash values for the admin.

4.3 User log in and account registration

For both the client and server all login attempts, and related communications are handled using the JSON messaging system using the LoginInformation class, for all registration and related communications the RegistrationInformation class is used. Descriptions of the stage ID codes used for each can be found under appendix ENTER APPENDIX ID.

A login process for a user will require the user to enter an email address and password, the client will then send the server the enter email address. This email address is looked up in the database to ensure that the email address has been previously registered, if there is an account matching the submitted email address then a random session salt is created by the server and all account details are downloaded, including account salt and hash value. The account salt, session salt, and a confirmation that an account is found is transmitted back to the client. On receipt of the client receiving this message from the server, the client will proceed to salt the entered password using the account salt, the client will then salt and hash this value using the session salt. After the client has generated the hash value of users' password this is then transmitted back to the server for comparison. Once the hashed password from the client has reached server side, the server will then have the saved hash value with the session hash – this process ensures that both the saved hash and the client hash has all gone through the same process. Both the server-side hash value and the client hash value are compared, if these values are equal then the user had entered the correct password, else the client has entered the incorrect password; the result of this comparison is transmitted back to the client, either the client has access to the chat server, or they must go through all the stages of logging in again.

Where a user does not have an account or wishes to create a new account on the server, they will be required to enter a nickname, email address, and password, all this data will be transmitted to the server using the RegistrationInformation class in JSON format. The server will look up the email address provided from the client on the accounts database, if the email address already exists in the database the registration cannot continue and the user must enter a different email address, else the server will generate an account salt and transmit this back to the client. Upon receipt of the account salt the client will salt and hash the entered password, the client will then proceed to create a JSON string from the RegistrationInformation containing the information of the hashed password, the email address, and the nickname – all this information will transmit back to the server. On server side, these details will be deserialised and a new user account will be created on the MongoDB.

4.4 SSL Connection

As was mentioned throughout the background research topic covering SSL / TLS server and client connections the method being used is a self-signed X509 certificate, this certificate has been created with the open-source technology of OpenSSL. The OpenSSL commands used for the generation of the SSL certificate can be seen below:

```

C:\WINDOWS\system32>openssl genrsa -aes256 -out server.key 4096
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:

C:\WINDOWS\system32>openssl req -new -key server.key -out server.csr
Enter pass phrase for server.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:UK
State or Province Name (full name) [Some-State]:Wales
Locality Name (eg, city) []:Treforest
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Student
Organizational Unit Name (eg, section) []:Student
Common Name (e.g. server FQDN or YOUR name) []:SecureChat
Email Address []:jessicaf091@outlook.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:SecureChat
An optional company name []:Student

C:\WINDOWS\system32>openssl x509 -req -sha256 -days 365 -in server.csr -signkey server.key -out server.crt
Enter pass phrase for server.key:

```

Figure 12 - SSL Certification Console Generation

These commands first create an RSA 4096 bit key and save this under server.key, then create a certificate signing request file; this file requires some information about the certificate authority due to this being a self-signed certificate the data entered to simply dummy data. The main important section of the second command is related to the challenge phrase, this is the password each client will need to echo back to the server when required to verify the SSL connection. Finally, the last command is related to the creation of the X509 certificate, this can then be installed on the machine that the server is being hosted. This certificate is only valid for a period of 365 days the expires, after a year a new certificate will be required; if the challenge phrase is changed the new time the certificate is generated then all clients will require an update to match the new phrase.

The server can only understand X509 certificates in the format of PFX, this requires a final command using OpenSSL to convert the generated .crt certificate to a .pfx certificate, the command required is as follows:

```

C:\WINDOWS\system32>openssl pkcs12 -export -out server.pfx -inkey server.key -in server.crt
Enter pass phrase for server.key:
Enter Export Password:
Verifying - Enter Export Password:

```

Figure 13 - Conversion From Certificate To PFX File

The created server.pfx file is then copied into the directory containing the server executable and then everything else is managed internal of both the server and client. Upon the admin creating a new server, the server class finds the stored pfx SSL certificate and saves this to a X509Certificate2 variable. Each time the server received a client connection a new ClientHandler object is initialised, within this object the SetupConn function is run to setup the SSL connection between the server and client; for this connection to be set up the ClientHandler is required to define a new SslStream object, this containing the client Network stream which allows the server to read and write data on the stream, the SslStream must

then authenticate the SSL stream by passing the SSL certificate and security protocol to the client.

On client side, the client follows similar steps of setting up the SslStream with the Network Stream, where the requirements differ is that client is required to verify the SSL certificate, when using an official certificate this would require called to the Certificate Authority that signed the certificate to ensure validity, however with this being a self-signed certificate this step must just return true. This is a security flaw within the server connection as realistically a fraudulent certificate could be passed to the client and the client wouldn't attempt to verify the validity of the connection, however for this circumstance this isn't avoidable. Once the client has verified the validity of the SSL certificate, the challenge phrase of "SecureChat" is echoed back to the server, once the server accepts this then the stream is verified, and the SSL tunnel is established.

4.5 Hashing

The implementation of hashing in the final solution of this project follows the design choices made for both the salt generation and the PBKDF2 functions shown in the prototyping stages has been directly exported in the authorisation class in both the server and admin. Testing was performed in the prototyping stages for the validity of the hash values generated when using the same input stream; in server application this means that hash values generated by using the PBKDF2 function can be reliably recalculated when the original passwords and salt values are entered into the system. Client-side hashing is responsible for hashing the entered user password twice, once with the account salt and again with the session salt. As mentioned during the research and prototype stages, hashing using the session salt ensure that the server can be certain of the sender of the login request. The server will take the hashed password from the user file and hash this using the same session salt the client used; comparison of these generated hash values are a key principle of the authorisation system. PBKDF2 hashing requires the definition of hashing iterations

The salt generation is also held within the authorisation class and generates a pseudo-random 128-byte value using the RNGCryptoServiceProvider available within the .Net framework, after generation this salt value is returned either packaged to be communicated between server and client and/or used within the PBKDF2 function.

4.6 Multithreading

The core of this project rotates around the ability for multithreading programming, all key systems in this project requires the use of multithreading. Server-side multithreading lives on 2 layers, the first instance being the server itself; when an admin logs in and starts an instance of the server, this instance lives on a separate thread besides the primary thread. This multithreading has allowed the ability for the server thread to manage all clients and server related managements, where the primary thread is where the admin has access to adjust various factors of the server, primarily regarding adding and remove users and admin details, as well as monitoring log data. The server thread can also spawn multiple threads itself; these threads are related to the TcpListener thread that is required to listen and handle any new connections from client machines; when this thread detects a new connection from a client another thread is spawned, this thread is related to handling all client data and communication and is handled by creating a thread in an instance of ClientHandler. The final thread handled server side is within the ClientHandler thread, this new thread is responsible for listening for messages from that individual client. These threads mean that there is

theoretically infinite number of clients that the server can manage, as each client will not interfere with each other directly, however practically the limit on the quantity of client is bound by the CPU performance and the dedicated RAM as each new thread will require greater resources.

Client-side development for this project is much more simplistic on the multithreading requirements, client side only requires 2 threads to run in parallel – with the new thread being spawned to handle listening for message from the server.

Using multithreading on any project imposes the risk of race conditions with the potential of multiple thread attempting to access shared resources concurrently, this can incur the risk of one thread changing the contents of a variable before the first thread has finished. This project only uses 2 shared pools, these being the logging data files (client and server) and the database provide from MongoDB.

Regarding the shared databased mutex locks have been implemented on functions calls on the server class to only allow one thread at a time to have access to the database. The primary risk for when 2 clients are attempting to register the same user details onto the database, without any mutex locks this can result in some unforeseen errors. Mutex locks on key areas of shared pools means that only one thread can manage shared data and others are places onto a queue to wait their turn. Managing race condition runs into the issue of decreasing overall performance and deadlocks, these are managed by only allowing threads to access shared memory, when necessary (which in the case of this program is only when logging in or registering), with deadlocks being managed by having the mutex locks timeout after a sufficient period if they haven't been released.

4.7 Client Handler

The Client Handler class has the primary responsibility for handling communications between the server and client, upon connection to the server a new Client Handler instance is created and appended to a vector of Client Handlers. Upon initialisation of the Client Handler class a new thread is created that is used to setup SSL connection between server and client, to start a new thread dedicated to listening to for messaged from the unique socket assigned to each instance of this class, and the finally to begin the hand shaking process – The handshake process has already been detailed previously. Other responsibilities of each Client Handler instance includes to manage special communications such as log in and registration requests, these protocols have been previously detailed as well as the JSON messaging system. Additionally, each Client Handler contains the initialisation for unique logging, this logging is used to log data as when a client's connection is authorised, if any errors occur with encryption or decryptions or in the event where any errors are likely to occur.

Each Client Handler has the unique responsibility of encrypting and decrypting messages using that connected clients unique AES encryption data, the design decision to contain the encryption and decryption internally of Client Handler retains to the time efficiency of multithreading. If the encryption and decryption was held internally of the Authenticator class then each time a ClientHandler received a message or prepares to dispatch a message a lock would be required on the Authenticator function pertaining the relevant function, this would ultimately lead to severe computational times for message sending. Additionally, having the encryption and decryption held within the Authenticator class would require each client AES keys to be shared internally of the server, this leads to unnecessary security risks. With each

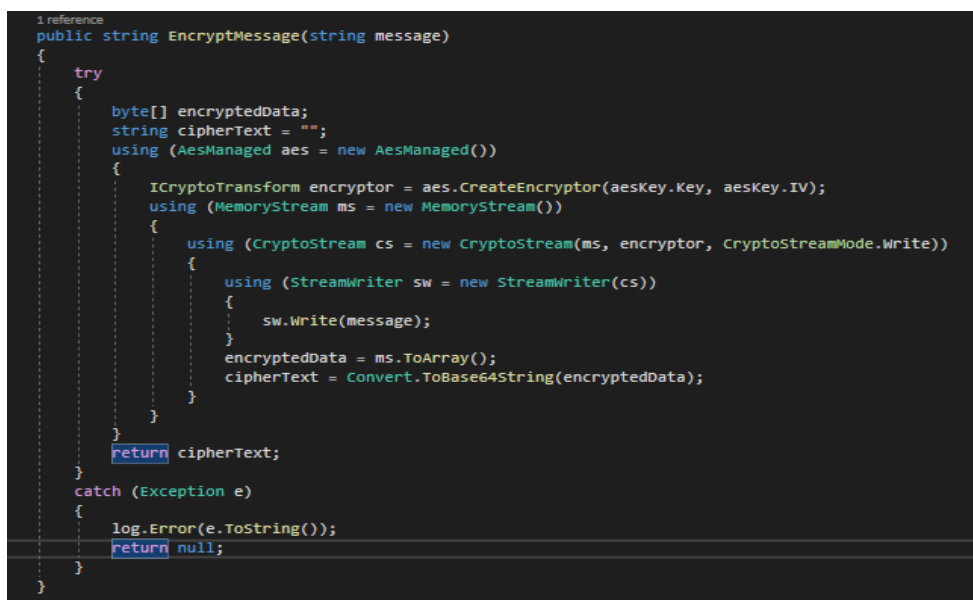
Client Handler running its own thread, and the AES key information already being stored within the class it made logically sense to contain the encryption and decryption processes internal of the class. The actual AES encryption and decryption processes have already been detailed.

4.8 AES Encryption / Decryption

Both server and client side have the same functions for handling the encryption and decryption of AES messages; on server side these functions are contained within the ClientHandler class, as this class already runs on a unique thread per client, so having AES management here will ensure that there is not a black log of AES management attempts when using the servers shared authorisation class. Whereas, for the client AES management is contained within the authorisation class. Microsoft provides guidance for the implementation of AES encryption and decryption when using the .Net framework, this guidance has been followed with changes name to adopt the guidance to merge into this project's development.

4.8.1 Encryption

The entire encryption process is wrapped in a try catch, this is to ensure any errors that might be thrown in the process are correctly logged and handles. The encryption process takes use of the C# using keyword, this ensure that the created object is destroyed when the program leaves the scope. The process requires 4 levels of objects existing with the using keyword, these are in order of AesManaged, MemoryStream, CryptoStream, and StreamWriter. Inside the AesManaged scope an encryptor is created using the clients AES key and IV values; the CryptoStream is configured to use the MemoryStream, generated encryptor and is configured to Write mode. The StreamWriter is configured to read the CryptoStream, the StreamWriter then write the entire CryptoStream is written to a byte [] variable. This variable is then converted to base 64 string and finally returned; the returned string value is considered the cipher text. Any errors detected whilst encrypting will result in an error being caught and an error log is made.

A screenshot of a code editor showing a C# method named `EncryptMessage`. The method takes a `string message` as input and returns a `string`. It is wrapped in a `try` block. Inside the `try` block, a `byte[] encryptedData` and a `string cipherText` are declared. An `AesManaged` object is created. An `ICryptoTransform` encryptor is created using the `AesManaged` object's `Key` and `IV`. A `MemoryStream` is created. A `CryptoStream` is created, wrapping the `MemoryStream` and the `ICryptoTransform`, and set to `CryptoStreamMode.Write`. A `StreamWriter` is created, wrapping the `CryptoStream`. The `message` is written to the `StreamWriter`. The `encryptedData` is obtained by calling `ms.ToArray()` on the `MemoryStream`. The `cipherText` is obtained by calling `Convert.ToBase64String(encryptedData)`. The `cipherText` is returned. A `catch` block for `Exception e` is shown, which logs the error and returns `null`.

```
1 reference
public string EncryptMessage(string message)
{
    try
    {
        byte[] encryptedData;
        string cipherText = "";
        using (AesManaged aes = new AesManaged())
        {
            ICryptoTransform encryptor = aes.CreateEncryptor(aesKey.Key, aesKey.IV);
            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor, CryptoStreamMode.Write))
                {
                    using (StreamWriter sw = new StreamWriter(cs))
                    {
                        sw.Write(message);
                    }
                    encryptedData = ms.ToArray();
                    cipherText = Convert.ToBase64String(encryptedData);
                }
            }
        }
        return cipherText;
    }
    catch (Exception e)
    {
        log.Error(e.ToString());
        return null;
    }
}
```

Figure 14 - AES Encryption

4.8.2 Decryption

The AES decryption process is very similar to that of the encryption process with some notable differences, primarily the decryption process requires the input of a byte variable over string variable. The levels of objects remains the same, with a decryptor being configured inside the AesManagement scope using both the AES key and IV. The MemoryStream is configured using the provided cipher text, the CryptoStream is configured with the MemoryStream, decryptor, and configured to read mode. Finally, inside the StreamReader the stream is read to end and saved into a string value. This value is now reverted to plain text and returned to the calling function. Like the encryptor if an error is thrown in this function this is caught and an error log is made.

```
1 reference
public string DecryptMessage(string message)
{
    try
    {
        byte[] encryptedData = Convert.FromBase64String(message);
        string plainText = "";
        using (AesManaged aes = new AesManaged())
        {
            ICryptoTransform decryptor = aes.CreateDecryptor(aesKey.Key, aesKey.IV);
            using (MemoryStream ms = new MemoryStream(encryptedData))
            {
                using (CryptoStream cs = new CryptoStream(ms, decryptor, CryptoStreamMode.Read))
                {
                    using (StreamReader reader = new StreamReader(cs))
                    {
                        plainText = reader.ReadToEnd();
                    }
                }
            }
        }
        return plainText;
    }
    catch (Exception e)
    {
        log.Error(e.ToString());
        return message;
    }
}
```

Figure 15 - AES Decryption

4.9 Authentication Information

The authentication information class is used for management of secure storage for sensitive information, this is both the admin and basic user admin details. This class contains 2 protected variables these being a list of items containing admin login information and the other being a list containing items containing user login information, the other variable held by this class is the initiation for the logger - the logger for this class is primarily used to log any errors detected when connecting to the database.

All information relating to admin and user login details are stored securely on a remote database being hosted by MongoDB, for the effective use of this information up creation of the Authentication Information class a call is made to the remote database requesting all the information held relating to login details, these details are passed to the server using the BsonDocument data type. This data is extracted and converted to a string format that is saved into the class AccountData and AdminLoginAuthentication respectively, this is then appended to the protected list being stored by this class.

When a user or admin logs into the server their username or emails addresses are fed into this class where their details are checked against stored records, if a match is found then the stored account details are relayed back to the requesting origin point, if no match is found then a log is made reflecting this. When a user has passed checks relating to the authentication of password the Authentication Information class is informed that a particular

user has been given permission to log in and a Boolean flag is changed in the account details to ensure that each account can only have a single login at one time.

This class has the additional responsibility of the addition of new users and admin into the server, with most of the required registration information being confirmed with database lookups to ensure there are no duplicate email or username entries. Once confirmations that there are no pre-existing accounts with matching credentials exists then a connection to the database is made and a new entry is given with all the new accounts relevant details, the account is also added to the live list hosted by the authorisation class so that user can log in straight away without a server restart being required.

4.10 Authentication

Both the server and client contain an authorisation class, with the server-side authorisation containing greater responsibility than the client-side. The client-side authorisation class is responsible for generating the clients AES details, this occurs when the constructor for the class is called. Additionally, this class is responsible for both the client encryption and decryption of both AES messages and handling the RSA handshake. Using both the functions `GetEncryptedAesKey` and `GetEncryptedAesIV` will take the public RSA key from the server and using the `RSACryptoServiceProvider` will encrypt both the AES key and IV using the provided RSA public key, this step is vital in the encryption handshake between client and server and will allow for future messages to be communicated via AES.

Within the server-side the authorisation class hosted many more responsibilities, primarily used to act as a barrier between the main program and the authorisation information class. Hosting many functions class between server/`ClientHandler` classes and the authorisation information class. Additionally, the server authorisation is responsible for RSA encryption and decryption, validating admin and user login attempts, hashing, salt generation, RSA key generation, and new account registration.

4.11 Logging

As was discussed in the background research and prototyping sections of this report the logging framework used for the final development is Log4Net, this framework boasts thread friendly file logging on a various severity level. During the prototyping stage the XML configuration was constructed using guidance from Apache on how to configure a rolling appended logging file, the configuration made means that when each day rolls over a new log file is created and stored in the same folder with all other logs; this allows for the ability for admins to view logs with daily intervals rather than every logged event at one time. Each major class (a class that have actual function code) is initialised with a logger global variable, this variable is initialised with the filename of the class. This means any logs made using this variable will be writing to the log file will be associated with the class making the log. Additional each log made will contain a time stamp, the thread the log was made on, the severity of the log, and the logged message. The configuration of the logger use can be found below:


```

<appender name="RollingFile" type="log4net.Appender.RollingFileAppender">
  <file value="logs/Log_" />
  <datePattern value="dd-MM-yyyy'.txt'" />
  <appendToFile value="true" />
  <staticLogFileName value="false" />
  <lockingModel type="log4net.Appender.FileAppender+InterProcessLock" />
  <layout type="log4net.Layout.PatternLayout">
    <conversionPattern value="%date [%thread] %-5level %logger - %message%newline" />
  </layout>
</appender>

```

Figure 16 - Log4Net Configuration Implementation

The class FileHandler is responsible for the management of reading data from and managing any data request from the log files, this includes updating the console with the most recent message and displaying specific logs request by the admin. With the log level being contained within squared brackets this allows the code to use the brackets as delimiters to be able to read what severity each log is. For instance, when an admin requests to view all INFO severity logs on a specific date, each log from that day will be analysed looking for the severity level, when a log has the INFO severity level this is saved to string list; once the entire file has been completely read from all the logs with the required log severity is displayed onto the console. Admins are presented with options to read logs from any stored log folder and specifically looks for ERROR, INFO, FATAL, or to view all logs from that day.

5 Results

The success of this project can be weighed by critical analysis of the final chat server produced in terms of functionality and performance. These tests are done either by direct analysis of the security features and testing edge cases to ensure they continue to follow expect results, and additional tests can be performed by gaining independent subjective opinion of peers for them to tests, assess, and to user the server in way that were not originally intended to ensure that the server is well suiting to manage unexpected activities; these subjective opinions are gathered via a Microsoft forms document, querying the user to attempts various different activities such as registering an account or asking the user to attempt to breach the server documenting how they done so. Finally, open ended questions are asked to gain subjective opinions on topics such as the appearance of the chat client, and what features they would desire to see added or subtracted from any additional work.

6 Future Improvements and Work

7 Conclusion

8 References

- David Wagner, B. S., 1996. Analysis of the SSL 3.0 protocol. *Proceedings of the Second USENIX Workshop on Electronic Commerce*, 1(1), pp. 29-40.
- Muhammad Aamir Panhwar, S. A. k. G. P. K. A. m., 2019. SACA: A Study of Symmetric and Asymmetric Cryptographic. *International Journal of Computer Science and Network Security*, 19(1), pp. 48-55.

Bamiah, M. and Brohi, S., 2011. Seven Deadly Threats and vulnerabilities in Cloud Computing. *INTERNATIONAL JOURNAL OF ADVANCED ENGINEERING SCIENCES AND TECHNOLOGIES*, 9(1), pp.087-090.

Swinhoe, D., 2021. *What is a man-in-the-middle attack? How MitM attacks work and how to prevent them*. Available at: <https://www.csoononline.com/article/3340117/what-is-a-man-in-the-middle-attack-how-mitm-attacks-work-and-how-to-prevent-them.html> (Accessed 19 October 2021).

Irwin, L., 2021. *How to defend against man-in-the-middle attacks - IT Governance Blog En*. Available at: <https://www.itgovernance.eu/blog/en/how-to-defend-against-man-in-the-middle-attacks> (Accessed 19 October 2021).

Information Commissioners Office. n.d. *What types of encryptions are there?* Available at: <https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/encryption/what-types-of-encryption-are-there/> (Accessed 19 October 2021)

Justice Adeenze-Kangah, Y. C., 2019. Detecting Proper SSL/TLS Implementation with Usage Patterns. *Journal of Physics: Conference Series*, 1176(2), p. 022045.

Sectigo, 2021. *Self-Signed Certificate vs CA Certificate — The Differences Explained*. [Online] Available at: <https://sectigostore.com/page/self-signed-certificate-vs-ca/> [Accessed 19 November 2021].

Smirnof, P. and Turner, D., 2021. *Symmetric Key Encryption - why, where and how it's used in banking*. Available at: <https://www.cryptomathic.com/news-events/blog/symmetric-key-encryption-why-where-and-how-its-used-in-banking> (Accessed 19 October 2021).

Ncsc.gov.uk. 2022. *Log4j vulnerability - what everyone needs to know*. [online] Available at: <https://www.ncsc.gov.uk/information/log4j-vulnerability-what-everyone-needs-to-know> (Accessed 16 January 2022).

ABCNews, 2021. Luke Barr. Available at: <https://abcnews.go.com/Politics/cybersecurity-official-warns-software-vulnerability-affect-hundreds-millions/story?id=81751264#:~:text=On%20the%20call%2C%20CISA%20Director,actors%2C%20according%20to%20that%20official.> (Accessed 16 January 2022).

Zhou, X. and Tang, X., 2011. Research and implementation of RSA algorithm for encryption and decryption. *Proceedings of 2011 6th International Forum on Strategic Technology*, 2(1), pp.1118-1121.

Singh, G. and Supriya, S., 2013. A Study of Encryption Algorithms (RSA, DES, 3DES and AES) for Information Security. *International Journal of Computer Applications*, 67(19), pp.33-38.

NIST, 2017. *NIST Special Publication 800-63B*. Washington DC: NIST, p.15.

Dharmadasa, K., 2022. *How to store passwords securely with PBKDF2*. Medium. Available at: <https://medium.com/@kasunpdh/how-to-store-passwords-securely-with-pbkdf2-204487f14e84> [Accessed 28 January 2022].

Beyondidentity.com. n.d. *Rainbow Table Attack | Beyond Identity*. Available at: <https://www.beyondidentity.com/glossary/rainbow-table->

[attack#:~:text=A%20rainbow%20table%20attack%20is,password%20hashes%20in%20a%20database.&text=After%20the%20user%20enters%20their,to%20look%20for%20a%20match](#). [Accessed 28 January 2022].

LookingGlass. n.d. *How to Thwart a Rainbow Table Attack*. Available at: <https://lookingglasscyber.com/blog/security-corner/thwart-rainbow-table-attack/#:~:text=Keys%20to%20Preventing%20Rainbow%20Table,is%20hashed%20the%20same%20way.&text=You%20solve%20this%20issue%20with%20password%20salting>. [Accessed 28 January 2022].

Anton, T., 2019. *The need to manage both symmetric and asymmetric keys*. [online] Cryptomathic.com. Available at: <<https://www.cryptomathic.com/news-events/blog/the-need-to-manage-both-symmetric-and-asymmetric-keys#:~:text=Typically%2C%20once%20a%20secure%20connection,is%20known%20as%20hybrid%20encryption.>> [Accessed 23 December 2021].

Domingus, A., 2020. *An Introduction to the Advanced Encryption Standard (AES)*. [online] Medium. Available at: <[https://medium.com/swlh/an-introduction-to-the-advanced-encryption-standard-aes-d7b72cc8de97#:~:text=An%20initialization%20vector%20\(or%20IV,is%20an%20added%20security%20layer.>](https://medium.com/swlh/an-introduction-to-the-advanced-encryption-standard-aes-d7b72cc8de97#:~:text=An%20initialization%20vector%20(or%20IV,is%20an%20added%20security%20layer.>) [Accessed 15 February 2022].

Heron, S., 2009. Advanced Encryption Standard (AES). *Network Security*, 2009(12), pp.8-12.

Simmons, G., 1979. Symmetric and Asymmetric Encryption. *ACM Computing Surveys*, 11(4), pp.305-330.

SSL Support Team. 2021. *What is SSL?* [online] Available at: <<https://www.ssl.com/faqs/faq-what-is-ssl/>> [Accessed 27 April 2022].

Arias, D., 2019. *How to Hash Passwords: One-Way Road to Enhanced Security*. [online] Auth0. Available at: <<https://auth0.com/blog/hashing-passwords-one-way-road-to-security/>> [Accessed 27 April 2022].

National Cyber Security Centre. 2018. *Password administration for system owners*. [online] Available at: <<https://www.ncsc.gov.uk/collection/passwords/updating-your-approach>> [Accessed 27 April 2022].

Arias, D., 2021. *Adding Salt to Hashing: A Better Way to Store Passwords*. [online] Auth0. Available at: <<https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/>> [Accessed 27 April 2022].

Microsoft. 2022. *.NET Framework system requirements - .NET Framework*. [online] Available at: <<https://docs.microsoft.com/en-us/dotnet/framework/get-started/system-requirements>> [Accessed 27 April 2022].

MongoDB. n.d. *Why Use MongoDB and When to Use It?* [online] Available at: <<https://www.mongodb.com/why-use-mongodb>> [Accessed 27 April 2022].

TowardsDataScience. 2020. *Why Should You Care About Logging?* [online] Available at: <<https://towardsdatascience.com/why-should-you-care-about-logging-442a195b80a1>> [Accessed 27 April 2022].

Watson, M., 2019. *Log4net for .NET Logging: The Only Tutorial and 14 Tips You Need to Know*. [online] Stackify. Available at: <<https://stackify.com/log4net-guide-dotnet-logging/>> [Accessed 27 April 2022].

NLog, n.d. *NLog*. [online] Available at: <<https://nlog-project.org/#:~:text=NLog%20is%20a%20flexible%20and,configuration%20on%2Dthe%2Dfly.>> [Accessed 27 April 2022].

Logging.apache.org. n.d. *Apache log4net – Apache log4net: Features - Apache log4net*. [online] Available at: <<https://logging.apache.org/log4net/release/features.html>> [Accessed 27 April 2022].

9 Appendix

Ethics Form

SECTION A: Project Definition

FOR UNDERGRADUATE & TAUGHT POSTGRADUATE ONLY

Complete the following table with full and relevant information relating to your research.

| | |
|---|---|
| Student Name | Jessica Clara Fealy |
| Student Number | 18024092 |
| Student E-mail Address (please use University e-mail) | 18024092@students.southwales.ac.uk |
| Name of Principal Project Supervisor | Simon Payne |
| Project Title | C# Secure Chat Server |
| Briefly describe the project, being sure to identify any aspects that are relevant to the Ethical Evaluation in Section B. NOTE: A project determined to be High Risk will need to include additional information in Section B to fully-specify the risks and mitigations. | To research, design, and developed a C# chat server using secure programming principles that utilises industry standard libraries, encryption methodologies, and other pre-existing technologies. |
| Please add an explanation of your study in plain English, with particular focus on any parts of your study which involve human participants. No more than 100 words. This is to help the Faculty Research Ethics Committee (FREC) to understand the project. | Creating a chat server. Questionnaire friends, family, and fellow students about the functionality of the security and features of the chat server. Questionnaire people in industry. |

SECTION B: Ethical Evaluation FOR UNDERGRADUATE & TAUGHT POSTGRADUATE ONLY

Consider the following points to determine the level of ethical risk your research presents:

1. Involves those who are considered vulnerable such as:
 - Children under 16.
 - Adults with learning difficulties.Unless in an accredited setting, accompanied by a carer or professional with a duty of care.
2. Involves those who are considered highly vulnerable such as:
 - Adults or children with diagnosed mental illness/terminal illness/dementia/in a residential care home.
 - Adults or children in emergency situations.
 - Adults or children with limited capacity to consent
3. Involves those who are “dependent” on others (such as teacher or lecturer to student). Unless in an accredited setting associated with normal working conditions or routines and within normal operating hours, such as a cultural institution, pre-school, school, or youth club where the research is carried out as part of professional practice such as curriculum development.
4. Requires full NHS ethical approval via the Integrated Research Application System.
5. Requires a Human Tissue Act license.
6. Involves “covert” procedures as in covert observation studies.
7. Involves anything considered “sensitive”. For example, does not carry a risk of those involved disclosing information which compromises the research (e.g., illegal activities; activities where moral opinion may differ, potential professional misconduct – work errors).
8. Induces significant psychological stress or anxiety or produce humiliation or cause more than fleeting harm / negative consequences beyond the risks encountered in the normal life of the participants (and where the potential for fleeting “harm” is clearly detailed in the participant information sheet). If in doubt regarding definition of the above terminology, please contact the research governance office.
9. Involves administration of drugs, placebos, or other substances (such as food substances or vitamins) as part of this study.
10. Involves invasive procedures (not limited to blood sampling, collection of biological samples, or passing current through a participant’s body, etc.).
11. Offers any financial inducements to participate in the study.
12. Intends to recruit serving prisoners or serving young offenders via Her Majesty’s Prison & Probation Service.

For your course, there may be specific requirements in **addition** to these, depending on the nature of the subject and how your project is assessed. You must also complete those requirements.

If **none** of the 12 points above apply, then the research can be considered **Low Risk**, *unless your course identifies additional criteria relevant to your subject that would render it High Risk*. This Section is then signed off by yourself and your supervisor and held on file for review by FREC.

If **any** of the 12 points applies, then the research is considered **High Risk** and students must bring the matter to the attention of their research supervisor immediately. **Research cannot then commence until mitigations for the risk are agreed by FREC**. Seek advice from your supervisor, who can help you identify mitigations of the risk or redesign as a Low Risk project.

All students must complete the section below, in collaboration with their supervisor.

Please strike through the statement that **does not** apply.

1. An ethics review has been completed, and the project has been identified as Low Risk.
- ~~2. An ethics review has been completed, and a High Risk was identified. I agree to explain how they may be mitigated below and agree to abide by any conditions identified at this stage, by my Project Supervisor, the School or the Faculty. I understand that High Risk projects can only proceed with approval from the Faculty Research Ethics Committee.~~

Issues: (Include as much information as possible to help FREC members to understand the issues. Extend onto additional pages as necessary.)

Proposed mitigations: (Include as much information as possible to help FREC members to understand the mitigations. Extend onto additional pages as necessary.)

Student's Signature: Jessica Clara Fealy
Date: 19/10/2021

Supervisor's statement: I have ensured due diligence and accountable decision making by the student. I have sought appropriate advice where required to support my judgment in this.

Supervisor's Signature: Simon Payne
Date: 19/10/2021

Any false or mis-represented information contributing to this Ethical Evaluation, including attempting to pass off a High Risk project as a Low Risk project, is subject to the Student Misconduct Regulations and may also have legal repercussions.

Both signatures are **required** for all projects, both Low Risk and High Risk.

Source Code

Server

Client