Access Control Software

­­­­\_\_\_\_\_\_\_\_

by

Saidhbh O’ Malley

(30007730)

\_\_\_\_\_\_\_\_

Supervised by

Janusz Kulon

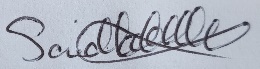
# Statement of Originality

This is to certify that, except where specific reference is made, the work described within this project is the result of the investigation carried out by myself, and that neither this project, nor any part of it, has been submitted in candidature for any other award other than this being presently studied.

Any material taken from published texts or computerized sources have been fully referenced, and I fully realize the consequences of plagiarizing any of these sources.

Saidhbh O Malley

Student Name (Printed) ………………………………..

Student Signature 

Computer Science

Registered Course of Study ……………………………….

04/05/2022

Date of Signing ……………………………….

# Table of Contents

[Statement of Originality 2](#_Toc102772627)

[Table of Contents 3](#_Toc102772628)

[List of Figures 4](#_Toc102772629)

[Nomenclature 5](#_Toc102772630)

[Abstract 6](#_Toc102772631)

[Project Aim 7](#_Toc102772632)

[Main Objectives 7](#_Toc102772633)

[Introduction 9](#_Toc102772634)

[Literature Review 10](#_Toc102772635)

[Security Patterns: 13](#_Toc102772636)

[Authenticator Pattern: 13](#_Toc102772637)

[Single Access Point (SAP) Pattern: 15](#_Toc102772638)

[Role-Based Access Control (RBAC) Pattern: 16](#_Toc102772639)

[Secure Logger (Auditor) Pattern: 17](#_Toc102772640)

[Reference Monitor Pattern: 18](#_Toc102772641)

[A Modular Design 20](#_Toc102772642)

[User Authentication 21](#_Toc102772643)

[Role-Based Authorization 24](#_Toc102772644)

[Reference Monitor 28](#_Toc102772645)

[Logger-Auditor 29](#_Toc102772646)

[createLog() 31](#_Toc102772647)

[readFromFile() 31](#_Toc102772648)

[AES Encryption & Base64 Encoding 35](#_Toc102772649)

[OpenSSL Encryption 35](#_Toc102772650)

[Base64 Encoding 37](#_Toc102772651)

[User Interface: Front-End Interactivity 39](#_Toc102772652)

[C# Client 39](#_Toc102772653)

[C++ Host 40](#_Toc102772654)

[Conclusion: 42](#_Toc102772655)

[References: 43](#_Toc102772656)

[Appendix: 45](#_Toc102772657)

[Appendix A: UML – RBAC/Authorization 45](#_Toc102772658)

[Appendix B: UML – Logger/Auditor 46](#_Toc102772659)

[Appendix C: UML – Authenticator 47](#_Toc102772660)

[Appendix D: UML – Reference Monitor 48](#_Toc102772661)

[Appendix E: UML – Main Diagram 49](#_Toc102772662)

[Appendix F: Gantt Chart 50](#_Toc102772663)

[Appendix D: Ethics Form: 50](#_Toc102772664)

# List of Figures

Fig. 1 - Basic Authenticator Pattern (Fernandez, 2013) -------------------------------------------- 13

Fig. 2 - Abstract Builder (Gamma et al. 1995) -------------------------------------------------------- 14

Fig. 3 - Proof of ID generation (Brown et al. 1999) -------------------------------------------------- 14

Fig. 4 - User-Privilege relationship (Yoder & Barcalow, 1998) ------------------------------------ 15

Fig. 5 - User-Role-Privilege relationship (Yoder & Barcalow, 1998) ----------------------------- 15

Fig. 6 - Role Based Access Pattern (Fernandez, 2013) ---------------------------------------------- 16

Fig. 7 - Secure Logger Pattern (Steel, Nagappan & Lai, 2006) ------------------------------- 17

Fig. 8 - Reference Monitor Pattern (Schumacher et al. 2006) ------------------------------ 17

Fig. 9 – Table of User Role-Privileges ----------------------------------------------------------------- 24

Fig. 10 – 128-Bit Block Diagram ------------------------------------------------------------------------ 35

Fig. 11 – Client-End User-Interface ------------------------------------------------------------------- 38

# Nomenclature

AES - Advanced Encryption Standard

CBC - Cipher-Block Chaining

DES - Data Encryption Standard

ECC - Elliptic Curve Cryptography

IOT - Internet of Things

IV - Initialization vector

RBAC - Role-Based Access Control

RSA - Rivest, Shamir and Adleman (Asymmetric Encryption)

SAP - Single Access Point

SMS - Short Message Service

SSL - Secure Sockets Layer

TLS - Transport Layer Security

# Abstract

The following work contains the research and implementation relating to the development of Access Control Software. Research has shown that the tried and tested method of designing a robust and secure system architecture is the implementation of a sequence of security design patterns. The final solution will demonstrate how, by developing a modular system consisting of four functionally independent security modules, that multilayered security can be implemented. The four modules consist of an Authenticator with which to verify a users credentials, a Role Based Authentication system so as to assign privileges to a subject based on their company role, a Reference monitor that will intercept all petitions to access confidential resources, and a robust logging system that securely stores all interactions in the system. A method of encryption and encoding will be incorporated to add a further element of security, ensuring that only employees of sufficiently elevated privileges can view confidential records.

# Project Aim

The aim of my project is to show it is possible to develop a secure Access Control System that can be incorporated into any Internet of Things device, thus filtering access to the internal file system using a combination of role-based authorization and reference validation.

# Main Objectives

The primary objective of the project is to develop a role-based user-authentication system that will filter access to sensitive areas using a password/username style authentication. The implementation of the Authenticator Pattern here will allow potential users to have their credentials verified before access is granted. As a prerequisite to resource access, authentication is the process of confirming the identity of the ‘user’ attempting to access the system, regardless of whether the entity is a human user or a device. While there are well established protocols in existence to maintain a high level of security during machine-by-machine authentication, this process provides little in the way of assurance that the human behind the machine is who they claim to be. This is where the task of user (human-by-machine) authentication comes into play (O’Gorman, 2003, pp. 2021-2022).

The Authenticator Pattern creates a Single Access Point through which a subject’s interactions may be received. Fernandez (2013, pp. 52-56) outlines how specific protocols can be implemented to prevent a malicious entity from impersonating a legitimate user, particularly relating to situations where the subject being impersonated has elevated privileges.

Once a user’s identity and credentials have been verified, an authorization level is determined and access permissions are granted relating their role-based privileges. Whenever you are working in an environment where there is a need to classify users in this manner it is useful to implement the Role-Based Access Control Pattern. User access will be filtered into in-built and pre-defined groupings, with each of these groups being assigned specified privileges (Fernandez, 2013, pp.107-110). This allows privileges to be assigned in a more uniform and structured manner. Rather than assigning individualized access permissions to the user on a person by person basis, the user is assigned to a grouping where it is the group that defines what privileges are assigned.

It is intended that a sequence of secure design patterns (outlined below) will be used to provide multi-level security. While patterns alone do not provide provable security, they are a means to an end when it comes to developing a unified architecture based on secure principles.

A fully secure logging management system will be implemented such that all attempts to enter the secure system, both successful and unsuccessful, will be recorded. The purpose of this is to create an auditable database of all system events and user interactions to enable any incidents or security breaches to be forensically analyzed post-hoc. The logging system will record data relating to which user enacted the event, what resource was attempted to be accessed and when the event took place. All logging details will be stored in a fully encrypted .txt file with access to said file log restricted to users of sufficiently elevated privileges. The secure logger will create a level of abstraction between the implementation of the actual logging and the access itself, with a logger factory class incorporated such that multiple bespoke loggers can be created according to what event is required to be logged.

Finally, an encryption facility shall be implemented to securely store all sensitive user credentials. The purpose of this file encryption is to hold user password/username pairs for use during the user authentication process, to store all references to user role and privilege assignment, and for all logging data to be stored in an encrypted state. This encrypted data will be easily written to and read from file by using encryption and decryption algorithms, in conjunction with a method of encoding data prior to saving and reading from file.

# Introduction

Most modern devices and applications on the market today are distributed and connected via some kind of network, be it the Internet or across a LAN (Anand, Ryoo & Kazman, 2014, p. 476). With the ever-increasing complexity of software and computer systems, coupled with our reliance on day-to-day access to our confidential data being available at our fingertips, it is of the utmost importance to develop robust security solutions before vulnerabilities can be exploited. (Fernandez, 2013, p. 1-3).

Access control, as a broader concept serves many functions within a software security framework, with the main objective being to protect sensitive system resources from inappropriate or malicious access. As a security model, it is primarily concerned with the task of authentication, ensuring that a subject or user is actually whom they claim to be and if that entity is permitted to enter the system legitimately. While authentication of valid users is an essential aspect of access control, once authenticated it is necessary to have the tools available to define what operations each individual user should or should not be capable of performing; what tasks the user is authorized to perform and how these tasks relate to their job-title or role.

The overarching objective of Information Technology is to create a safe environment where users can access and share resources and data. The interconnected nature of modern society requires greater and greater levels of connectivity. On the surface, restrictions could appear to impede this objective with imposed restrictions on the free access of information being filtered to a chosen few. The reality is that a well-managed and effective access control system actually enhances and facilitates ‘sharing’ by creating a provably secure platform in which allows ‘selective sharing of information’ among verified legitimate users (Ferraiolo, Chandramouli & Kuhn, 2007).

To quote Fernandez (2013), “secure systems need to be built in a systematic way in which security is an integral part of the software lifecycle”. It’s not enough to simply develop a secure platform and expect an application running on it to remain secure in isolation. Nor is it good practice to attempt to manage security issues as they arise, patching holes in the metaphorical dam and expect the system to remain robust. Attempting to retrospectively fix vulnerabilities after-the-fact, not to mention the associated risk of these vulnerabilities being discovered after software has been released, can be incredibly expensive for both users and developers alike.

While no system is truly 100% secure, it is commonly accepted among software professionals that security must be addressed at the design stage and implemented at the architectural level. Security design patterns facilitate the inclusion of tried and tested methods of structuring elements of code. They deliver reusable solutions to common problems, and unlike algorithms which primarily address computational issues, patterns provide the developer with well-known templates that can be employed in many different security situations (Dougherty et al. 2009).

# Literature Review

Prior to the era of computing, the act of confirming the identity of another person was based primarily on recognition. If two people met and required an exchange of information, the confirmation of each other’s identity was not a complex task. Humans can visually recognize each other; they can ask each other questions and they can implement the whole range of human interaction skills that help us to determine if we ‘believe’ a person is who they say they are. Once we entered the computing era, where the vast majority of these information request interactions do not take place face-to-face, the concept of ‘user authentication’ had to evolve. With the bulk of data being stored digitally, both locally on servers or in cloud storage, security relating to device access has become one of the most important elements to consider when developing a software system.

To consider developing a robust and substantial security system, Fernandez (2013, p. 4) outlines the key types of countermeasures which should be implemented:

* Identification and authentication where any potential user’s identity must be verified and authenticated and a set of credentials can be used to describe the attributes associated with their authenticated access.
* Authorization and access control to define the authenticated user’s permissions (read/write/execute etc.) so as to filter their access to resources and prevent un-authorized access to confidential data.
* Logging and auditing all events and actions that may be relevant for later analysis.
* Hiding of information or encryption such that the data itself is protected.
* Intrusion detection system to alert, in real time, if a system is under attack so defensive measures can be taken.

The purpose of the identification/authentication service of a software system is to circumvent the need for physical human interaction in the security process. This is achieved by implementing a method to remotely recognize and validate an individual’s identity to establish trust. As users have become more mobile their access requirements have become more complex. Equipment is vulnerable to theft, and with most modern embedded devices belonging to the Internet of Things (IoT) with an online or networked element, the risk of eavesdropping or intercepting of transmissions is an ever-growing source of vulnerability. Hence, the establishment of a degree of trustworthiness relating to a user’s identity must be established before access to a secure system can be granted (Fernandez, 2013, pp. 32-33).

If authentication is looked at more broadly as a security concept, taking into account both face-to-face and computer-based authentication practices, O’Gorman (2003) and Avasthi & Sanwal (2016) group these factors into three categories.

1. Knowledge based authentication
2. Object-based authentication
3. ID-based (or Biometric) authentication.

If looked at individually it is possible to see that the concept of authenticating a user is quite a complex one.

Knowledge-based authentication is the idea of security characterized by secrecy, relying on the idea of ‘what you know’. This involves verification of identity based on passwords, pin-numbers and secret information known only to the human user. As the most commonly used form of validation, it is also arguably the most vulnerable as it is largely memory-based, relying on the user to memorize their proof-of-ID. When relying purely on memory there is a tendency for users to generate passwords or pin numbers relating to something that could be easy to remember, and the simpler the password the less secure it is. Poorly selected, short and easily memorized passwords that may be in use across multiple platforms are extremely insecure and highly susceptible to theft. Password based security is vulnerable to any number of security breaches from man-in-the-middle, phishing and social engineering attacks to communication eavesdropping and basic human error. People tend to choose passwords that are easy to remember (personalized passwords such as a date, phone number, pets name) as the average user finds difficulty in memorizing long and complex passwords (Sood, Sarje & Singh, 2009).

Object-based authenticators rely on the concept of ‘what you have’; a physical object such as a key or fob, or more commonly nowadays, a cryptographic key. The obvious drawback of a physical-object key is that it can be lost or stolen quite easily. Although, as O’Gorman (2003, p. 2024) points out, when a physical object is lost the owner becomes aware almost immediately and can act accordingly, whereas with a digital key its loss might only become apparent after a security breach incident.

ID-based authentication relates to ‘who you are’ and is primarily related to a person’s physical uniqueness within the general population. Biometrics use a method of identity recognition based on a user’s physiological or behavioral characteristics. Physically there are many characteristics that are said to be unique to each person and hence are considered to be a highly secure method of confirming a user’s identity; examples include fingerprints, hand geometry, iris and retinal scanning and facial recognition (Avasthi & Sanwal, 2016, pp. 215-216). Behavioral biometrics on the other hand relate more to how a person uses their body, rather than the body’s physical traits. Patterns such as a person’s voiceprint, signature recognition, mouse and keystroke dynamics can be used quite effectively to establish a user’s unique identity. Avasthi & Sanwal (2016) focus primarily on the idea of keystroke dynamics where they establish that how a person types, their finger position, keystroke pressure, key-press delay, can all be used to construct a unique ‘signature’ for that individual. They argue that, when used in combination with another verification method such as a password, that a person’s unique template of how they enter a specific sequence of characters can be beneficial for logical access control.

When dealing with security-sensitive user details such as passwords and credentials, it is essential to implement a robust cryptographic solution to obfuscate and encrypt all files containing confidential information. While it is possible to develop simple algorithms using techniques such as bitwise encryption, these are inherently rather insecure as it is not difficult for a would-be intruder to use techniques such as ‘dictionary words’ to discover the encryption key and create a cipher.

Modern cryptography in general use today uses advanced mathematical principles to store and transmit data, such that only the intended sender and recipient can read or process the contained message data. While there are many forms of encryption algorithms available, in general they can be broadly categorized into symmetric or private key encryption (AES, DES etc.) and asymmetric or public key encryption (RSA, ECC etc.). It is this combination of cryptographic keys in conjunction with the mathematics at the heart of the algorithms themselves that make encryption possible (Thakkar, 2020)

DES (Data Encryption Standard) is regarded as one of the original and traditionally the most widely used forms of symmetric encryption. Data is encrypted in 64-bit blocks using a short 56-bit encryption key, with the input plaintext and output ciphertext both being of the same 64-bit length. Based on an earlier algorithm design by the German-American cryptographer Horst Feistel, the DES algorithm was approved as federal standard in 1976, with its first publication in 1977 (Hazma & Kumar, 2020, p. 335 and Zebaree, 2020, pp775-777). The single encryption key and a plaintext block are taken as algorithm inputs, with the same private key being used for both encryption and decryption (Rihan, Khalid & Osman, 2015, p 151). While the short 56-bit key is considered too short for modern applications, it laid the foundations for many of the more secure algorithms to come.

Within 30 years the AES (Advanced Encryption Standard) algorithm superseded its DES predecessor as the go-to form of symmetric encryption. Based on the Rijndael block cipher, its longer key lengths of 128, 192 or 256- bits and the implementation of an initialization vector as a starting variable mean it provides a far superior level of encryption to its short key predecessor. AES is a block cipher, a means to encrypt/decrypt where a string of plaintext is treated as a single ‘block’. This is used to generate a block of ciphertext of the same size (Rihan, Khalid & Osman, 2015, p 152). By using CBC (cipher-block chaining) an initialization vector adds an element of randomness to each block such that even identical blocks will not use the same cipher to decrypt them. It works on the principle of substitution and permutation, with the encryption process consisting of various sub-processes, such as ‘sub-bytes’, ‘shift rows’, ‘mix columns’ and ‘add round keys’. Depending on the size of the encryption key, either 10, 12 or 14 rounds of processing occur, with each round including the four aforementioned sub-processes (Thakkar, 2020).

One disadvantage of symmetric key encryption is that the key distribution is done primarily through a third party, which can negate the essence of encryption if the key is compromised (Agoyi & Seral, 2010, pp. 449-449). In contrast, asymmetric encryption involves multiple keys in both the encryption and decryption of data, with the keys mathematically related to one another. The most obvious benefit to this method of encryption is the security it offers, with the publicly known key being used in the encryption of the data and the private key being used in the decryption.

Commonly used in the transmission of SMS messages, email encryption and SSL/TLS certificates, RSA encryption is one of the most commonly used form of asymmetric encryption in use today, implementing Leonard Euler’s totient theorem (aka the Fermat-Euler theorem) of modular arithmetic. Two huge and randomly generated prime numbers of the same bit size *p* and *q* are selected as the basis for the key generation, with the RSA modulus *N* being equal to the product of *p* and *q*. The idea is that the public and private key exponents *e* and *d* must satisfy the congruence *ed* ≡ *1(mod φ(N))*, where *φ(N)* = *(p – 1)(q – 1)* is Euler’s totient function and is evaluated using the two random prime numbers above. The public key is chosen to be a random number *e* such that it is both less than the Euler’s totient *φ(N),* and the greatest common divisor or ‘gcd’ of *φ(N)* and *e* is 1. Next the private key is calculated using the above number *e*, such that the private key *d* is congruent to *e-1(mod φ(N))* (Nitaj, 2012, pp. 3-8).

The primary advantage of RSA encryption over its symmetric counterparts is in its scalability. Encryption key lengths vastly outstrip the 256-bit keys of AES encryption, having a range from 768-bits to over 16,384-bits, with any key length above 3072-bits considered to be the most secure. Despite the mathematical approach to key generation, its implementation in the public key infrastructure is actually quite straightforward, something that has allowed RSA encryption to become the most widely used encryption algorithm in regular use today (Thakkar, 2020).

## Security Patterns:

When designing a robust security system, it is impossible to always solve every issue that may arise from its first-principles. When a specific code design works, is tried and tested, it gets reused over and over. These recurring patterns of classes and objects, successful elements of architectural structures, became what is now referred to as design patterns. They provide ready-made and easily implemented structural solutions to well-known and recurring design problems, helping to avoid the pitfalls of design glitches (Heyman et al. 2007, p. 1). Design patterns are essentially generic templates that exist independent of their implementation. They do not describe an actual solution, rather an abstract arrangement of classes and components that define the solution concept. Patterns help developers to understand both the problems and their solutions, rather than providing an oven-ready answer to an architectural quandary (Schumacher et al., 2006, pp. 2-4).

The idea of re-usable design patterns was first introduced by the British-American architect and design theorist Christopher Alexander. He proposed that all complex architectural developments, from buildings and cities to object-oriented software, were developed using reusable building-blocks for architecting space and structure (Dangler, 2013, p. 23). In is seminal 1977 book, ‘A Pattern Language’, Alexander explains how “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in a way that you can use this solution a million times over, without ever doing it the same way twice.”

In their book, ‘Design Patterns: Elements of Reusable Object-Oriented Software”, Gamma et al. (1995) formalized this architectural concept, compiling these well-known structures in the first true catalog of software patterns. Here they described each of the known design patterns in a formal and structured manner, providing the names, classifications, applications, uses and implementations that remain essential, influential and familiar to software engineers to this day. As structural patterns their popularity stems from how they can easily be visually understood and how effectively they can be adapted and implemented. There is a huge amount of ready-to-go source code available and how they interact with one another is well defined.

As design patterns have been so successful among software professionals it was probably inevitable that a similar methodology would be extended to the task of threat mitigation. The natural assumption would be to presume that, like design patterns, they can be easily described using UML and implemented using familiar sample source code. While this does remain true, with many interesting and powerful patterns in common use being presented this way, there are many others that can be considered more conceptual rather than structured. These relate more closely to the actual processes and methods involved in the developing security-focused software and are more likely to impact the organizational and managerial aspects of project development (Kienzle, 2002, pp 3-4).

The following are a selection of Security Design Patterns which are intended to make up the main structure of the final Access Control Software that will be developed.

### Authenticator Pattern:

The goal of implementing this pattern is to prevent imposters from gaining access to the system by impersonating a legitimate user. Additionally, user authentication must be achieved in a reliable and secure way so as to prevent an imposter circumventing the security protocols (Fernandez, 2013, p.53).

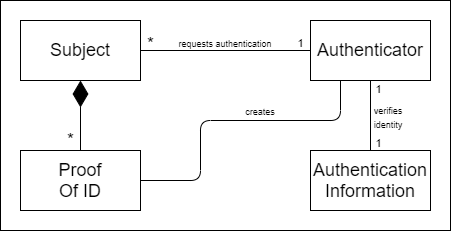


Fig. 1 – Basic Authenticator Pattern (Fernandez, 2013)

When a user requests system access, the Authenticator requests verification of the user’s identity by cross-referencing the user-supplied data with the Authentication Info stored in a persistent and encrypted database of legitimate user credentials. Once the subject’s credentials have been confirmed and the user is deemed “verified”, a ‘Proof of ID’ token is assigned to that user’s access permissions so as to allow them to retain their designated access privilege throughout their logged-on session (Fernandez, 2013, p.54). Fernandez refers to this variant on the Authentication pattern as the “Single Sign On”, whereby this ‘token’ is assigned to the user’s session to qualify them for any future access required throughout their active session.

Brown et al. (1999) take the traditional authenticator pattern structure outlined above a little further. While their work relates primarily to situations where the subject is attempting to access remote objects, it may be possible to implement elements of their structure to enhance the original pattern. They describe a server system that acts as a repository of objects, with user authentication being used to restrict user access based on each request.

Their structure implements an abstract authenticator class to be the interface that provides authentication and communication with the concrete authenticator classes that inherit the abstract class. This abstract class allows multiple concrete authenticator classes to be instantiated, providing the system with the capability to allow a more bespoke authentication system depending on the subject’s requirements.

The Abstract Factory Design pattern (Gamma et al., 1995, pp. 83-91) could be used, where the concrete authenticator implements an abstract factory class as in interface to create authentication objects in the inherited concrete factory classes.

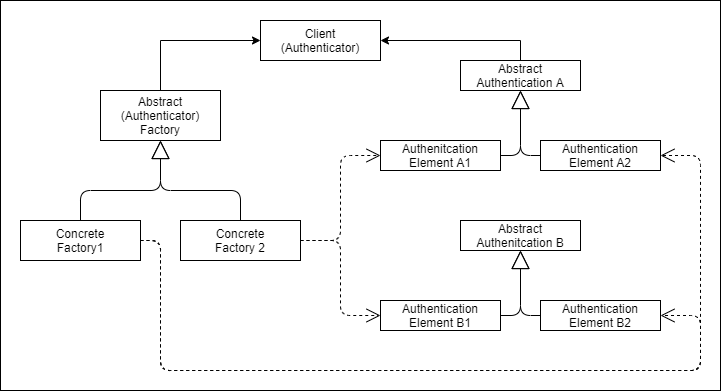


Fig. 2 – Abstract Builder (Gamma et al. 1995)

Once a successful authentication has been recognized, Brown et al. (1999) implement an ‘Object-Factory’ pattern to create their ‘protected object’, but also identify that it is here that additional actions as required by the authenticator class can be performed. This structure lends itself quite nicely to providing the authenticator with a proof-of-ID token. The concept of an ID token enhances the authentication process by providing each recognized user with the proof of this legitimacy. This can be now associated with this user and carried throughout their validated session.

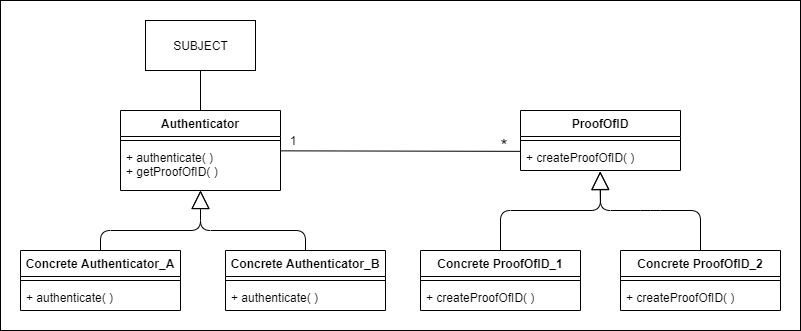


Fig. 3 – Proof of ID generation (Brown et al. 1999)

While a simplified version of the Factory pattern above would suit the implementation of this, a similar result could be achieved by implementing the Builder Design Pattern (Gamma et al. 1995, pp. 93-101), where elements of the more complex authentication object can be constructed. If each concrete authenticator class is constructed in such a way that it functions as a ‘Director’ class, it can have an abstract ‘Builder’ interface composed with it that calls the inherited concrete builders to assemble the components required to fully authenticate a user for each concrete form of authentication required.

### Single Access Point (SAP) Pattern:

As an addition to the authenticator pattern it is beneficial to incorporate the Single Access Point (SAP) pattern as a unique interface for all system access requests to enhance security monitoring and control. The SAP pattern creates a gateway of separation between the external entities requesting access and the internal secure system, with any inbound ‘traffic’ being routed through a single-entry point. As all validated users must pass through this access point it can be used as an appropriate place to capture an access log to store information relating to all successful and unsuccessful authentication attempts (Rosado et al., 2006).

### Role-Based Access Control (RBAC) Pattern:

According to Anand, Ryoo and Kazman (2014) the RBAC pattern serves to group multiple subjects according to their pre-defined roles based on their duties. In this way, rather than assigning each subject/user their own unique access privileges, the user is assigned group access rights as permitted to their group. In this way legitimate users’ access to secure systems can be filtered such that only users with very specific credentials are authorised to perform certain duties within the secure system.

When dealing with a large database of legitimate users the one-to-many user-privilege relationship can become difficult to manage without some method of role assigning.

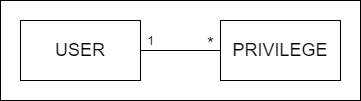


Fig. 4 – User-Privilege relationship (Yoder & Barcalow, 1998)

By splitting the user-privilege relationship into ‘user-role’ and ‘role-privilege’ relationships it is possible to maintain the one-to-many connection outlined above, while reducing the total number of actual relationships involved. Maintenance of user roles becomes an easier task to manage and can facilitate a higher level of security. As job titles and their associated privileges change over time, the actual role-privilege relationship can now be directly edited. Similarly in situations where a user’s job title changes their role relationship can be modified rather than manually updating every user-privilege relationship associated with them (Yoder & Barcalow, 1998).

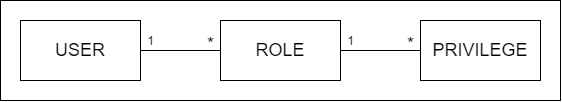


Fig. 5 – User-Role-Privilege relationship (Yoder & Barcalow, 1998)

Yoder and Barcalow expand this concept further to take into account where a user’s job title may encompass more than one concrete role (i.e. where the subject is both an accountant and a manager). They argue that, rather than creating a bespoke role to facilitate every potential hybrid-job combination, that it would be more efficient and secure to make each user’s role variable “store a set of roles”. Role sets can be created describing editing privileges and viewing privileges etc., with more complex role-privilege systems perhaps implementing a hierarchy of role-types and their corresponding subtypes. In this way a composite role can be thought of as containing many sub-roles.

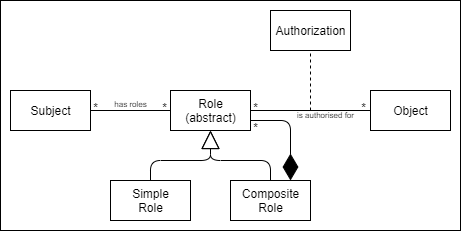


Fig. 6 – Role Based Access Pattern (Fernandez, 2013)

An access control matrix is one method of specifying the individual relationships between the individual subjects requesting access, and the secure system to which they require access (Saunders, Hitchens & Varadharajan, 2001, pp. 6-7). The basic elements of a role-based system can be separated into Users, Roles and Permissions, with legitimate users easily being switched between roles and their permissions accordingly changing depending on the assigned role. A 2-dimensional matrix or grid can therefore be created using subjects (those users who request access) and the corresponding objects (the systems or files etc being requested), with each subject/object pair indicating the unique permissions applicable.

### Secure Logger (Auditor) Pattern:

While not an essential component of Access Control Software as a whole, it is believed that implementing a secure file logging facility in the system can certainly aid in maintaining a record of any incidents that may occur.

Fernandez (2013, p. 111) recommends the use of this pattern as a method of tracking users’ accounts to determine which users accessed what material within a secure system. With any system that deals with sensitive data it is essential to keep a record of who did what and when, and this must be done in a way such that the data is faithfully stored in a persistent state and securely encrypted. As these data logs contain information that could be compromised during an attack by an imposter it is essential that any adversary that enters (or indeed anyone who inadvertently accesses) the secure system must have limited/no visibility of the content of the log files (Dougherty et al, 2009, pp. 75-76).

By implementing a secure logging system, a process is in place to prevent an attacker from altering files or logs and prevents them from hiding their actions. Each time a user accesses the system, this action is recorded taking note of the identity of the user, what was accessed (both successfully and unsuccessfully) with a date and time stamp of when this occurred. As an auditing system it allows for the forensic process to be possible. If an incident occurs it may be necessary to audit user actions to ascertain how the attack occurred in the first place, to prevent similar occurrences in the future. This can also aid in future security improvements as if a pattern of system misuse becomes apparent in the system logs, more robust security measures can be implemented to target specific exploitations (Fernandez, 2013, pp. 111-114).

Another use of a secure logging system is in the processing of confidential data such as user credentials or actual files on a system. As before, this is done so as to maintain a log for forensic purposes, but also to allow data to be securely encrypted in persistent storage. It is a critical requirement of any security system that logged data should not be viewed by any unauthorised users.

Steel, Nagappan & Lai (2006) consider the logging process to consist of two parts. Firstly, the securing of the data being logged, and secondly the logging of the actual data in a secure manner. They propose a system where a Secure Logger class provides centralised control of the logging functionality by decoupling the actual implementation of the logger itself from the code that will use it. When a call is made to securely log an access attempt or a data file, it is the secure logger itself that handles how the events etc are logged and the user has no access to that process.

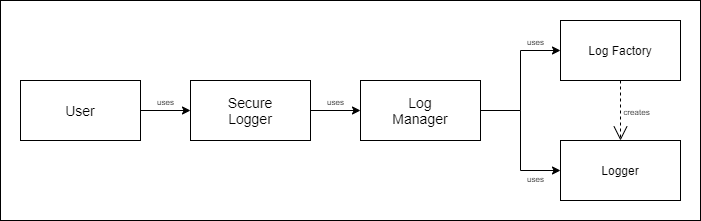


Fig. 7 – Secure Logger Pattern (Steel, Nagappan & Lai, 2006)

Any time a user requests access, either to the system itself or to a resource, the secure logger secures the data to be recorded and sends the event string to the log manager. The log manager in turn obtains an actual logger from the Log Factory where a logger instance of the specific logger intended to be used is created. Finally the log manager sends a request to the actual logger to log the event securely. The principle here is that each part of the logging process is handled by a separate part of the logging system; the secure logger class deals with securing the data through encryption while the log manager deals with the actual logging process itself (steel, Nagappan & Lai, 2016, pp. 577-579).

### Reference Monitor Pattern:

Any time a user makes a request for access to data or resources on a secure system, it is important to define access restrictions to that information whenever an entity, human or otherwise, makes that request (Schumacher et al., 2006, p. 256). The Reference Monitor pattern defines how an abstract process can be implemented to intercept all petitions from resources, confirming they have authorization for access and are in compliance with authorizations (Rosado et al., 2006, pp. 141-142)

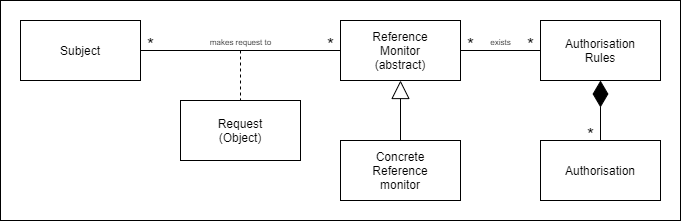


Fig. 8 – Reference Monitor Pattern (Schumacher et al. 2006)

Heckman and Schell (2016, pp. 2-3) define the implementation of this pattern in the form of a ‘security kernel’. They describe the interaction between subjects (the active entities, processes or devices that instigate information flow) and objects (the passive entities that store or receive the information), where the subject makes reference to the object “based on a set of current authorisations”.

In his 2004 thesis, Erlingsson (2004, pp. 4-5) describes some of the basic requirements involved in implementing this pattern:

1. Completeness, in that the mediation of all operations must be performed by the security kernel.
2. Integrity of the kernel must be maintained and protected from external tampering by isolating the reference monitor itself.
3. Correctness must be guaranteed with the monitor/kernel being small enough to test or analyse, with verifiable correspondence between the security policy and the implementation of the security kernel (Heckman & Schell, 2006, p. 3).

By implementing a Reference Monitor at every entry point through which resource requests can be made, it is possible to intercept and verify their compliance with authorisation prior to access being granted (Schumacher et al., 2013, pp. 256-258). The implementation of complete mediation allows the reference validation mechanism to authorise each operation against an access control policy in real-time. As it is only the security-sensitive operations that could potentially violate access control, this mediation ensures user behaviour and processes can only perform authorised actions. Tamper-proofing doubles down on this mediation process by ensuring malicious entities cannot directly access or modify the behaviour of the reference validation kernel. By preventing unauthorised subjects from interfering with this mechanism it ensures the access control policy cannot be surreptitiously circumvented, ensuring every action or request passing through the validation process itself. A reference validation mechanism can only be considered ‘correct’ if it always generates the desired access control query, correctly processes that query against the access control policy and successfully implements the resultant decision (Jaeger, 2011).

# A Modular Design

The concept of modularity in software design is the idea that the differing functionalities of a program can be separated into independently functioning modules, each with its own purpose, and operating relatively independently of the other modules within the program. Modules can be seen as the physical ‘containers’ in which we declare classes, create objects and can be the structural basis of the logical software design. The goal of each module is such that “it should be possible to change the implementation of other modules without knowledge of the implementation of other modules and without affecting the behavior of other modules”. Each block of code should be functionally independent in such a way that its code could potentially be recompiled as a single unit without the need for compiling the entire code base (Booch et al, 2007). The packaging of classes and functionalities in this way allows for greater code reuse across any future applications and allows for entire code modules to be extended or even replaced with minimal disruption to the legacy code.

The architecture of this design consists of four separate modules of functionality: An Authenticator to validate a subject’s supplied credentials, a Role-Based Authorizer to allocate a subject’s pre-defined role within the system and assign unique access privileges to determine their permissions while logged into the system, a Reference Monitor to intercept all petitions for resources by a subject, determining if their credentials and permissions are sufficient to allow access to protected resource objects, and an Activity Logging system to track and log user activity and allow the processing of sensitive data through file encryption.

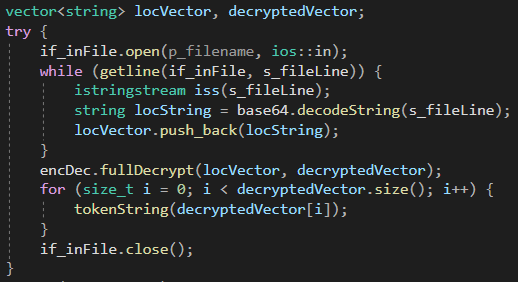
## User Authentication

The authentication process allows the user to be validated prior to being permitted to enter the system. The user is prompted to input their personal credentials into the console in the form of a username/password pair, which will be validated against an encrypted file of verified users. The getLoginDetails() member function of the Subject class accepts these user details as provided and passes them to their corresponding private member variables.

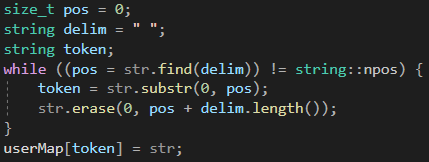
The actual authentication process is performed by one or more concrete authenticator classes, with the design making use of the polymorphic nature of object-oriented programming ([see Appendix C).](#_Appendix_C:_UML) By allowing multiple concrete authenticators to be derived from an abstract base class, it is possible for a specific version of the authentication process to be instantiated and implemented depending on what or who is being authenticated and how they wish to be verified. In the final development one concrete authenticator is implemented to verify user credential pairs, but given this “plug and play” style of architecture, it is possible to add or remove authenticators with minimal code additions/alterations as necessary without affecting the overall functionality of the Authentication process as an independent module.

Once the user has provided their credentials to the Subject class, the SystemLogin class begins the actual user authentication process. An Authenticator pointer has been created as a private variable in this class, allowing any potentially available concrete authenticators to be selected for implementation. In the current development the ‘authenticate’ variable is instantiated as a pointer to the ConcreteAuthenticator1 class, where username/password validation occurs. By calling the ‘authenticate()’ method, it is possible to call the version of this method contained in the concrete authenticator class. The current subject’s details are passed to the method by reference as a class object.

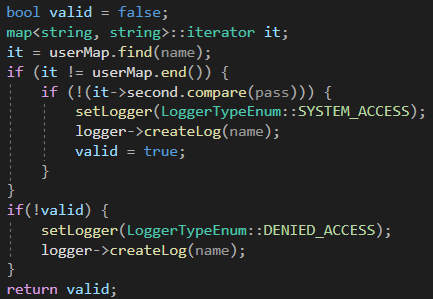
The elegance of the Authenticator pattern is that functionality is divided across the authenticator classes and the authentication information class, with each performing its own end of the user authentication process. The AuthenticationInfo class performs all actions relating to the actual validation itself, by comparing the supplied user strings to the validation information stored persistently in an encrypted text file. Each time this class is instantiated, it takes the file name as argument into the parameterized constructor. A try/catch block tests if the file itself has been successfully opened before any operations can be performed on the file data. Next, each individual file line is then read in using stringstream as an encrypted and encoded string.



First the file line is binary-to-text decoded into a usable ‘encrypted text’ format using Base 64, with each decoded line being pushed back into a local vector. Next that vector is passed to the fullDecrypt() function where the decoded vector of encrypted file lines can be decrypted into a user-readable format. Finally, each individual file line is passed, one at a time to the tokenString() function, where the string is tokenized using a delimiter into username and associated password substrings. This method of ‘tokenizing’ strings into their composite sub-strings is a technique used throughout the system as the default method for deconstructing file lines as they are read in from persistent storage. Each credential pair is then inserted into a ‘map’ so as to retain the username/password association in a searchable format.

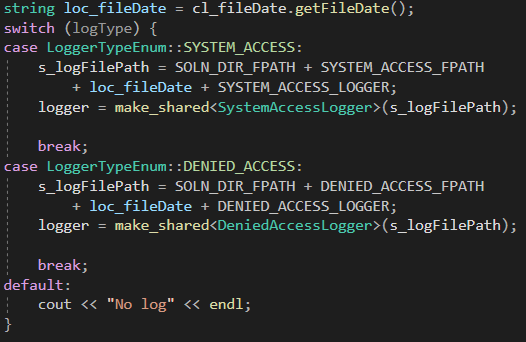


The actual validation is performed by the AuthenticationInfo member function, validateUser(). First a Subject object is passed to the authenticate() method by reference, where the user supplied credentials are extracted and in turn passed to the validation method. Using a map iterator, the function takes the provided username and iterates through the ‘userMap’ populated above to check if the name supplied corresponds to one of the validated usernames on file.



The validateUser() method itself returns a boolean depending on the outcome of the validation process, so if the name supplied is not found it returns false. Provided the name is found the iterator checks the corresponding password to confirm the username/password pair supplied matches one of the entries saved to file.

Finally, the Authentication process calls the Logger Auditor module to create a system log recording the outcome of the user authentication, specifically whether the authentication process was successful or if the user was denied access. By calling the setLogger() method, authentication outcome is passed as an enum to a switch menu where the correct form of file logging is chosen. A smart pointer ‘logger’ is set to make a shared logger of the desired type, with a file string passed to the logger constructor to assign the correct file path to execute the save.



## Role-Based Authorization

The role-based authorization module forms one part of the main subject authorization process and allows each user’s designated ‘role’ to be created prior to a resource access request. This module’s design was largely based on the Composite Design Pattern (see [Appendix A](#_Appendix_A:_UML)), with classes being composed into tree structures where individual objects and compositions of objects can be treated uniformly and can be represented as whole-part hierarchies. These class hierarchies are defined as consisting of primitive objects and composite objects, where the primitives can be further composed into more complex objects, which in turn can be re-composed and so forth recursively (Gamma et al., 1995, pp. 157-167). As such, the specialized role privileges (read, write, execute etc) are treated similarly to the composition of the privilege objects, with the RoleBuilder class being further composed into these specialized role building classes that inherit from it.

The design consists of a hierarchical structure where multiple roles can be inherited through an abstract ‘Role’ class so as to allow a variety of permissions or actual roles to be assigned depending on how ‘Role’ itself is instantiated. Once a subject has been authenticated, the authorizeSubject() method is then called to recursively iterate through a series of inherited sub-role classes to build their personal authorization string. As each privilege is assigned, the role itself is recursively constructed by the CompositeRole class, where its getRole() method retrieves each role-part one at a time.

Two role pointers are declared to implement the construction of a user role authorization: ‘rolePtr’ that follows the role construction by pointing to each of the components of the final composite role object, and ‘buildRole’ that iterates recursively through each assigned derived privilege class. The process begins by initializing buildRole as a shared pointer to the CompositeRole class as a start point, with rolePtr being pointed to that same instance of CompositeRole.

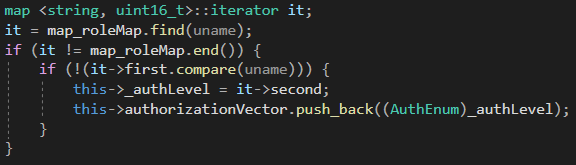


The authorization process consists of two parts, assigning a simple role where the user’s actual privilege level is assigned, and the composite role where the actual privileges associated with that role are aggregated. Within the system there are four predefined role types that can be assigned to a user (low, medium, high or admin) with each user only permitted to be assigned to a single role at a time within the hierarchy. With each increasing authority level, the user will be allowed a greater level of privileges relating to how they can interact with protected resources and perform object requests. While a user’s individual privileges will not yet be assigned, it is in SimpleRole class that they get allocated their broad role designation which will later determine which default privileges can be allocated. In addition to the aforementioned four actual roles there is an additional role of None provided so as to allow default initialization of ‘no role’. This is so as to facilitate the principle of Least Privilege so no subject can enter the secure system with a privilege level any greater than None and will not gain any privilege greater than what will be assigned. It will also allow the security levels to default to zero-access to lock out all users in the case of a security breach.

When the simpleAuthorisation() method is called, the buildRole pointer is now set to SimpleRole and the user’s unique username is passed through to the class constructor along with rolePtr itself.



The class constructor uses a try/catch method of validation to ensure that the relevant file has been opened successfully. It accesses the role assignment textfile, decodes and decrypts each file line before tokenizing each file string into username/authorization pairs and inserting them into a map so as to preserve their association. Finally, a map iterator is used to iterate over the entries to locate the correct username. Once found, the corresponding authorization integer representing a user’s role is assigned to the ‘\_authLevel’ member variable.



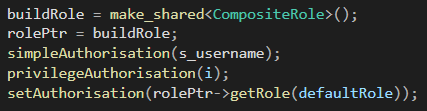
The functionality of assigning a role to a user and the actual assigning of their corresponding privileges are separated into separate entities so as to simplify the one-to-many relationships between them. While a user can only have one role within this particular system, it is entirely possible for this process to be expanded such that a verified user could be assigned multiple roles within the system, with each of these roles being allocated their own related privileges.

Each current role has an associated list of access privileges relating to their specific access permissions regarding how they are permitted to interact with each of the 6 unspecified file types.

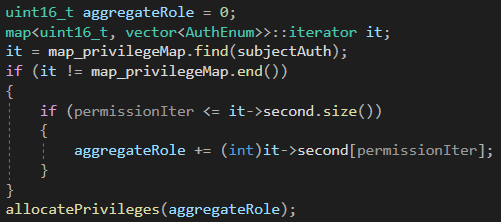
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **Permissions** | | | | | |
| * NA: No Access | **Role** | **Role int.** | **File 1** | **File 2** | **File 3** | **File 4** | **File 5** | **File 6** |
| * R: Read Only | **None** | 0 | NA | NA | NA | NA | NA | NA |
| * W: Write | **High** | 1 | FA | FA | FA | FA | FA | FA |
| * E: Execute | **Medium** | 2 | R|W|E | R|W | R|E | R | R | R |
| * FA: Full Access | **Low** | 4 | R|E | R|E | R | R | NA | NA |
|  | **Admin** | 256 | FA | FA | FA | FA | FA | FA |

Fig. 9 – Table of User Role-Privileges

Next the corresponding privilegeVector that relates to that user’s specific role is selected, and it is this that is to be used to assign the user’s individual privileges in the RoleBuilder. As each user role has specific and different roles relating to each file type, it is necessary to iterate through each part of the user’s vector of file privileges so as to assign each one individually. Each iteration of a for-loop calls the privilegeAuthorisation() function passing the loop index ‘i’ in as argument.



A map iterator selects the vector of permissions that correspond to the users assigned role. Next it selects the ‘second’ variable that corresponds to the for-loop iteration value (from 0 to 5) and fetches its corresponding permission setting this integer value to the local aggregateRole variable. Finally, that integer value that represents one of the 6 file permissions is passed to the allocatePrivileges() method where the actual assignment of privileges is created.

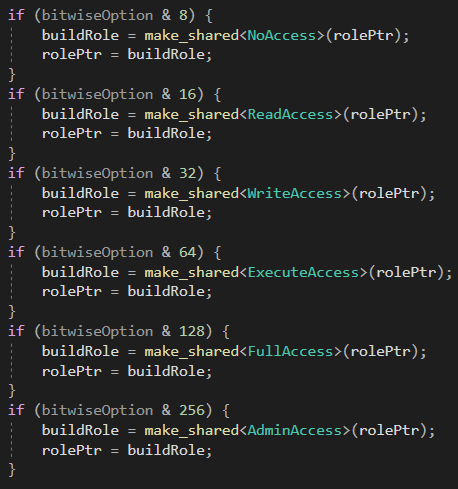


This process makes use of an enumerated class, AuthEnum. Each enumerated value in the class represents a specific type of permission which can potentially be incorporated into the overall aggregated permission allocation, with each numeric value corresponding to a specific integer value from the binary number sequence (1, 2, 4, 8,…). By using bitwise operator calculations, it is possible to determine from the composite supplied integer value representing the aggregated permissions, which individual permissions can be assigned. This works as follows:

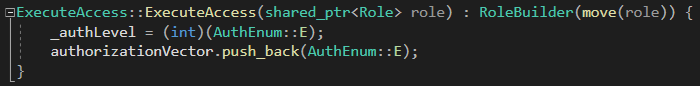
* Read Access = 0x10 (corresponding to 00010000 in binary or 16 in decimal format)
* Write Access = 0x20 (corresponding to 00100000 in binary or 32 in decimal format)
* Supplied integer = 48, requesting Read Access permission.
* Using bitwise addition, only the bit values that are 1 in both binary numbers are kept:
  + - 48 = 00110000
    - 16 = & 00010000
* So (48 & 16) gives a value of 16dec or 00010000, allowing the permission of Read Access to be allocated.

The theory here is that for any composite number supplied (e.g. 112 = 16 + 32 + 64) the correct permissions can be thus allocated according to the role level allocated.

Each privilege part is constructed by calling its own derived privilege class (e.g. ExecuteAccess) that contains all the relevant information needed to add that particular permission type, with each of the privilege classes being derived from the RoleBuilder base class. As such, series of IF statements filters the flow through to the correct privileges using this system of bitwise operation.



Rather than simply reassigning the pointer to each additional access sub-class, the move() function is implemented, taking the existing pointer to the current derived class and redirecting it to the next class. What this does is transfer the ownership of the pointer to a different location, so rather than overwriting or deleting the original class pointer, ownership of the heap memory is transferred to the secondary pointer. This essentially nullifies the original pointer rather than copying it, incrementing the reference count, allowing the pointer to recursively iterate through each of the selected derived access classes in turn as required to construct each assigned permission.



The implementation of this method of ‘building’ permissions in this way is intended to perform multiple functions within the overall system. An additional functionality can be implemented to allow a system admin to provide additional (temporary or single-use) permissions to users on request. Thus, a subject that has only been allocated low-level read-only default permissions during the authentication and authorization process can request additional ‘upgrades’ to their permission levels. A user with sufficiently elevated (e.g. Admin) permission, can manually select the additional permissions requested and, provided they are not already assigned to that subject, decorate that subject’s permissions with additional elevated permissions. The theory here is that the permissions could be requested on a file-by-file basis, with the elevated privileges being either added as a one-off access or as a permanent role ‘enhancement’ for that particular user. While this has not been full implemented yet, it is certainly a feature that could be incorporated in the future with minimal effort.

Each of the user’s file permissions are stored in a vector, with each vector slot directly corresponding to the file type it is associated with. It would, therefore, be possible for an admin to view the user’s current privilege level as assigned by the authorization process, and further ‘decorate’ that permission with additional privileges using the same process as when the original permissions were assigned.

## Reference Monitor

The implementation of a Reference monitor module allows each individual request for resources to be vetted and each access request petition to be assessed prior to providing the subject with authorization to view and interreact with an object ([see Appendix D](#_Appendix_D:_UML)). Each resource that is stored in the secure system will have its own permissions such that depending on a user’s role they may be restricted in their interactions with said resource, or find themselves denied access completely.

The Role-Based Authorization module outlined above serves to assign each user that enters the system with access privileges according to their pre-determined role, with more elevated interactions permitted only to those users with equally elevated role-based privileges. Each resource request received by the monitor is passed through a single ‘security kernel’ function that performs the task of comparing a user’s role-based security privileges and credentials with the necessary permissions required for that particular user to access the protected object in the manner requested.

The ReferenceMonitor class itself functions as an abstract class through which the concrete monitor can be called. This is done so as to uphold the principle that the reference process must always be ‘invoked’ rather than directly called from the base class. As such it is the derived ConcreteReferenceMonitor that is instantiated from the SystemLogin class to allow the monitoring process to take place. When this object is created, the class constructor opens the access matrix text file, reading in each file line, decrypting and decoding the string and storing each file line in a local vector. As before, each file line is then tokenized into substrings, pushed back into a permission vector and the data is inserted into a map to preserve the connection between each user role and its associated vector of file permissions. As a subject’s role privileges increase, so do their file access permissions for each file.

The Reference Monitor kernel itself consists of a single function validateResourceAccess(), that when called, processes each resource request in real-time acting as a single access point through which all requests must be filtered. It uses an IF/ELSE statement to determine if access will be granted or denied. A bitwise operator calculation is performed (similarly to the privilege assignment implemented in the Composite Role class described above. The returned integer value from the checkPermission() method represents the aggregated value for the permissions the user has relating to the specific resource they are requesting. A bitwise AND calculation is performed to determine if the user has either Full Access privileges OR has been allocated the desired privileges by the authorizer module.

For example, if a user has been assigned ‘User’ access to resources, and requests Write Access to a file type 2 object. If their permissions only allow read-only access to that resource the request will be denied as follows:

* Role-Based permission to file type 2 = Read Only = 0x08
* Write access to file type 2 = 0x04
* (0x08 & 0x04) = 0x00 or No Access, therefore write access is denied.

As this is a single access point, it is essential that a file log of all activity through the security kernel be logged persistently so all resource access can be fully monitored, regardless of whether they are successful or unsuccessful.

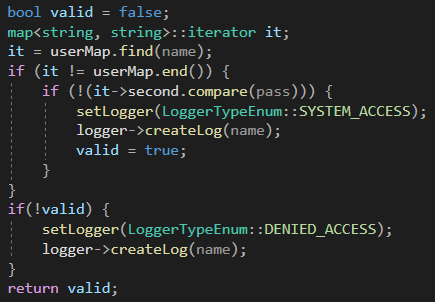
## Logger-Auditor

With any secure system dealing with potentially sensitive data, it is essential to have a method of recording and keeping track of all activity within the system itself. This can include user logins, resource requests and data modification attempts, regardless of whether they were successful or unsuccessful. This implementation of a logging and auditing system can help in the prevention of any would-be attacker maliciously accessing or altering content, as any activity that occurs within the system is recorded faithfully in a persistent and encrypted state.

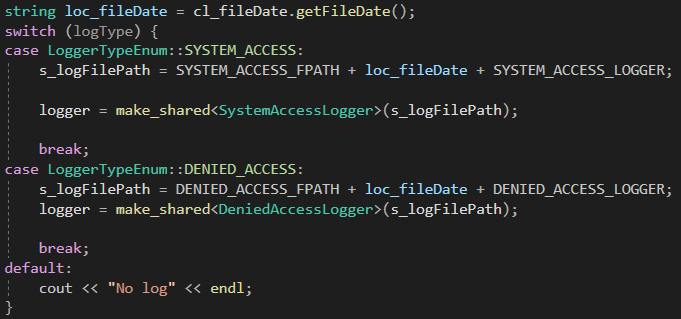
The architecture of the Logger-Auditor module ([see Appendix B](#_Appendix_B:_UML_1)) incorporates an abstract logger class, so any number of concrete loggers can be derived from the base class so as to allow for a multi-functional and bespoke logging system that directly relates to the system requirements. The concept here is that, regardless of the data being sent to a log file, a single abstract logger can direct the functionality towards the correct file logger class. The LoggerAbs base class contains three fully virtual methods, createLog(), readFromFile() and returnFileDate(), that can be overridden as necessary so as to allow the system to call the correct version of each method. By instantiating a specific logger, that derived class’s version of the virtual method is called provide the desired technique of creating a new log or reading from a log file.

The system as it stands contains four concrete loggers; ‘SystemAccessLogger’ and ‘DeniedAccessLogger’ that deal with recording the success/failure outcome of a user’s login and authentication, and ‘ResourceAccessSuccess’ and ‘ResourceAccessDenied’ that relate to the success/failure of all resource requests that have passed through the Reference Monitor validation process. A fifth derived class from LoggerAbs base class is ‘LoggerViewer’ that is called when a user with sufficiently elevated admin privileges requests to audit file logs.

The log class selection itself is performed by pointing a smart pointer, ‘logger’, to the desired logging class as determined by the ‘success/failure’ outcome of a user’s interaction with the system. Looking more closely at the authentication process, the logger to be instantiated is determined by whether the user’s supplied credentials at the log-in screen match an entry in the encrypted user credentials file, as inserted into a map member variable, ‘userMap’.



A map iterator searches the ‘userMap’ of valid credentials for the username. If the associated password matches the user-supplied one, the subject is deemed to be authenticated and is permitted into the system. Conversely, an authentication failure will obviously disallow the user entry. This successful or unsuccessful outcome allows the ‘setLogger()’ method below to be called, passing the correct log type in as an enum. A file-name string is generated by concatenating the directory file-path, today’s date and the type of log being created, and it is to this specific file location that the actual access log is created.

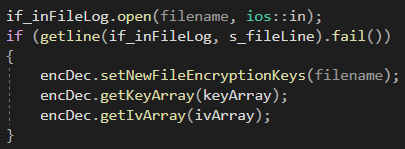


Finally, the logger smart-pointer is instantiated as the correct log type, passing the filename string as parameter.

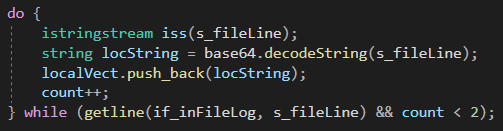
The first four logging systems themselves can be examined together as their functionality is virtually identical. When one of the class constructors is instantiated, it receives the aforementioned filename and, using a try/catch’ block, attempts to open the chosen file. If for any reason this process fails, the ofstream failure exception is caught, ending the process and preventing the system crashing.

Each concrete logger class has two parametrized constructors. The first has a single ‘file name’ parameter and is called when a new file log is being generated for logging. The second has two parameters, a pointer to the Logger Abs class and the file name, with this constructor coming into play when an admin level user requests to view/audit existing file logs.

There are two file-open scenarios catered for when either constructor is called, regardless of which concrete logger is required: either a new and empty log file has just been created, or an existing file has just been opened. As a new log text file is generated each calendar day that an access attempt of any kind has been made, when the very first daily log entry is to be entered the file itself will be completely empty. This is indicated when the first attempt to get a file line, getline(), fails. In this scenario it is essential to generate a new set of encryption keys and save them to the empty file, with each new pair generated being completely unique to the data stored in that particular file.

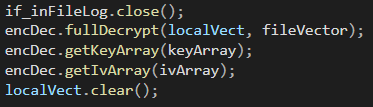


In the scenario where there are already logs saved to file, a do-while loop iterates through each file line, reading each in with a ‘stringstream’, before decoding it into a decryptable format. One at a time, they are pushed back into a local vector in preparation for passing to the fullDecrypt() method where the entire vector of decoded strings is decrypted.



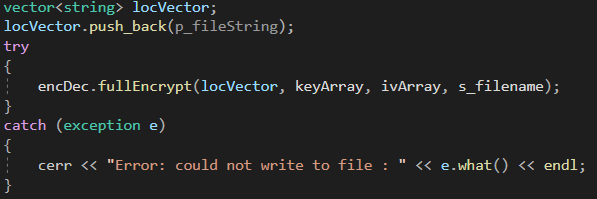
It is here that the two constructors differ slightly, in the case of the single parameter constructor, only the first two file lines are read in from text file and decrypted, the two strings that contain the encryption key and its associated initialization vector (both of which are essential for securely encrypting all logging data). As this scenario is for creating new file logs it is only these two strings that are necessary as all new file logs will be appended to the existing secure file.

The second constructor is implemented when a user wishes to read from a file log, so it is essential for the entire log file to be read in and decrypted. A pointer to the LoggerAbs class allows for the log data to be formatted or ‘decorated’ by ‘moving’ the pointer along to the next logging class in sequence prior to being presented to the user in the finished format.



### createLog()

Each time a user interacts with the system by attempting to log in or requesting resource access, a version of the createLog() method is called. In each of the four scenarios outlined above an identical process takes place, with only the file name and file save path differentiating between their functionalities. The string encryption method, ‘fullEncrypt()’, being implemented requires that a vector of strings be passed to it, regardless of whether it is a vector containing a single string or a vector set of multiple strings.



First the file string passed as parameter, ‘p\_fileString’, is pushed back into a local vector, then the Encryption class member function fullEncrypt() is called. The local vector, along with the stored encryption key array, initialization vector array and destination file-name are then passed to the function. There each vector string of unencrypted data is converted to an unsigned character array then passed into the encrypt() method. This is where the actual AES encryption algorithm obfuscates the data string. Lastly, it is encoded using base64 encoding and appended to the specified text file log.

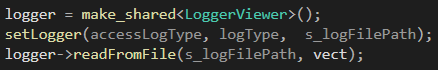
### readFromFile()

The readFromFile() method is somewhat more complicated than simply creating a file log, as the process involves filtering the search results according to a chosen parameter. The returned data-set is formatted for user readability using ‘decoration’.

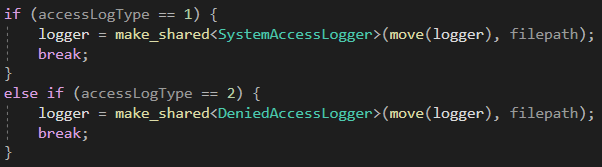
As the data stored in a log file is considered to be sensitive information, only a user with sufficiently elevated privileges is permitted to access and view these records. Furthermore, the option to actually interact with and audit log files is only provided as a menu option to those users who have been assigned ‘admin’ status. In this way it is impossible for a user without sufficient privileges to accidentally interact with any restricted content while logged in.

Prior to viewing file logs, the user is prompted to select which access logs they require: User Authentication or Resource Access logs, then whether they wish to view the related successful or unsuccessful log files. Next the file date is inputted, with the user supplying the day, month and year relating to the actual file. It is with these three elements that the file path to the required log is generated.

It is here that the fifth LoggerAbs derived class is implemented: LoggerViewer. First the logger pointer is instantiated as a shared pointer to the logger viewer class. Next the setLogger() method function is called, passing the user selection data and file path through as parameter.

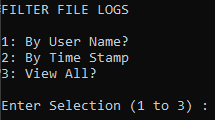


This setLogger() functions similarly to the previous version of setLogger() in that it uses the user-inputted viewing choices to direct the flow to direct the smart pointer towards the correct logging class. The main difference is in how this is done. As with the creation of the aggregate user privilege vectors above in the RBAC module, the move() function is implemented here in a similar manner. The existing pointer to the LoggerViewer class is redirected to the second logging class, linking the classes recursively and allowing the pointer to iterate back through each of the derived logger classes in turn.

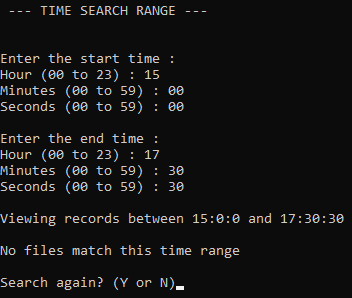


As an example, suppose the user wishes to view successful authentication logs from the 5th of May 2022. First the smart pointer is initialized as a LoggerViewer, then the pointer is reassigned to the SystemAccessLogger class. The order here is important as the pointer will recursively iterate from the last assigned class back to the first. In order to present user data to console first it needs to generate a vector of all possible log entries on that date. This is done by instantiating the doubly parameterized constructor from the SystemAccessLogger class where a vector is populated with all the relevant file strings that were generated on that date. Next that class’s readFromFile() function is called with the log vector being passed to the function. The pointer then directs functionality to the next class, the LoggerViewer, and its version of readFromFile() is called, passing the vector of file logs on to be formatted.

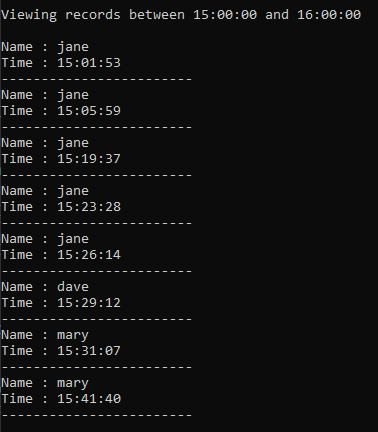
Once the file log vector has been set to the member variable ‘fileVector’, the user is prompted to decide if or how they wish the file logs to be filtered for them. If the user does not wish to view all records at once (this could be a huge number of actual log files), they can filter by user name, by time stamp or by both at the same time to fully narrow down a search.



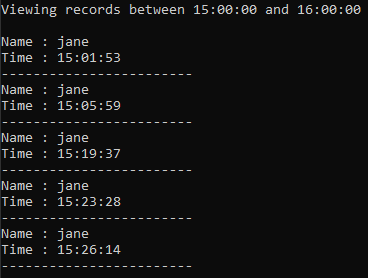
To filter by time stamp, a range of time values is required so the user is prompted to enter the start time or lower search bound, and the end time or upper search bound. This will then present only the file logs that exist within that given range to console. Before each time stamp is accepted as valid, it is validated to ensure that the numbers supplied are not outside the suggested bounds. If no records exist on file within that range the user is prompted to try again for a different range if they so wish.



Once a valid range of dates has been chosen the user name and time stamp relating to the logs are formatted and presented to the user via the console.



If the search parameters for the same date and time period are re-filtered to search for just entries for employee ‘jane’, you can see below how the output changes accordingly.



## AES Encryption & Base64 Encoding

As with any secure system, it is absolutely essential that some form of encryption be implemented to obfuscate sensitive user/client data and to protect against file tampering etc. It was decided that every single piece of data that shall be read in or saved to file should be fully encrypted using AES-256 cipher-block chaining (CBC) as encryption involving a larger 256-bit encryption key using a 128-bit initialization vector is deemed to be very secure and robust.

### OpenSSL Encryption

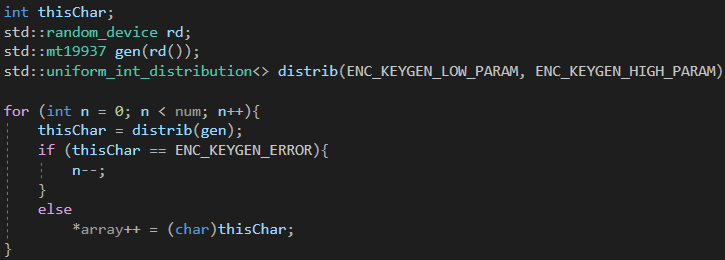
The encryption library that was chosen for implementation in this development was OpenSSL’s symmetric encryption/decryption library as it has a vast range of tools on hand to create a solid encryption methodology. OpenSSL provides two primary libraries that deal with the fundamentals of its cryptographic routines:

* ‘Libcrypto.lib’ provides the functions for performing symmetric encryption and decryption operations across a wide variety of algorithms and modes
* ‘Libssl.lib’ is a tool-kit that primarily provides the functionalities of Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols and depends on ‘Libcrypto’

The encryption and decryption methods implemented in the finished software were developed with reference to sample code provided by OpenSSL (OpenSSL(b), n.d.).

#### Key Generation

As with any AES symmetric encoding system, it is necessary to generate a new, unique and completely randomly generated set of encryption keys (the encryption key itself and the initialization vector). The initialization vector (IV) is essentially a random ‘first block salt’ that is used to initialize the feedback of the encryption algorithm. The benefit of using encryption keys with an IV is that regardless of whether or not a fresh set of keys is generated with each encryption, a completely different ciphertext will be produced from an identical plaintext input. That said, it is believed that by generating new keys with each instance of encryption that this adds an additional level of security as it would be impossible to guess or predict the keys based on previous encryptions.



The key and IV used in the system’s encryption and decryption process have been chosen as 256-bit and 128-bit respectively. Despite having a far smaller encryption key than its asymmetric RSA counterpart, given current technology it is still considered impossible to brute-force a 256-bit encryption key. With a powerful quantum computer, it would take approximately 2 trillion years to crack (for reference the universe is a paltry 13.7 billion years old), hence the AES-256 block-cipher has, to-date, never been cracked (Scott, 2020). The 128-bit length of the IV directly relates to the 128-bit block size used by AES. The data to be encrypted is split into a 4-by-4 array of 16 bytes as demonstrated below, so with 8 bits per byte giving a total block size of 128-bits, the size of both the plaintext and ciphertext output will remain the same.

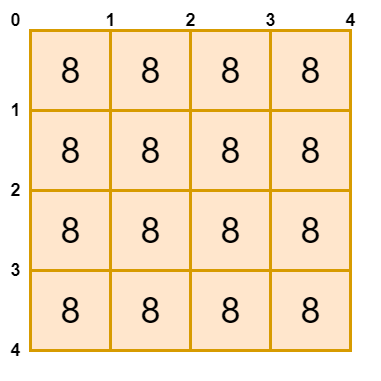


Fig. 10 – 128-Bit Block Diagram

To generate a number that is truly random number it is essential that the engine generating it is not based on a pseudo-random algorithm. While functions such as rand() appear to produce randomness, its values are not actually uniformly distributed and its sequence is inherently deterministic. Instead the ‘random\_device’ class is instantiated to generate a single uniformly-distributed unsigned integer of 32-bit length. It does this by either accessing the OS’s entropy pool or by using a hardware random number generator when available.

The Mersenne Twister engine is a strong pseudo-random number generator based on the Mersenne prime number 219937 – 1 (the 51 known Mersenne Primes are the set of prime numbers that are one less than a power of two, i.e. Mn = 2m – 1). While its stand-alone number generation is not completely random, the act of seeding it with a unique and truly random number initializes it to a state whereby each new number seed produces a fully randomized 32-bit output. As 219937 – 1 is an unimaginably large integer, it is considered highly unlikely that any two numbers produced would be identical.

By instantiating the ‘mt19937’ class and seeding it with the ‘random\_device’ number generated above, a truly random number can be generated.

Next the ‘uniform\_int\_distribution’ class is instantiated to provide the system with a closed interval distribution over which the actual key values may be selected. An upper and lower bound of ASCII values is chosen such that the value range falls between 33 and 126. This range of characters consists of all upper and lowercase letter, the numbers from 0 to 9, and all keyboard special characters, but does not include the range of ASCII escape codes such as ‘\n’ or ‘\0’ that could insert erroneous behavior into an encryption key.

Finally, a for-loop iterates over the number of characters required for the given key (either 16 for the IV or 32 for the encryption key), taking each randomly generated character and adding it to the character key array (Assencio, 2015).

#### Encryption

The encrypt() method takes a char\* to the unencrypted plaintext string, the key and IV, and the ciphertext buffer in as parameter, along with the length of the plaintext string to be encrypted. A sequence of conditional statements is used to format and transform the plaintext from a human-readable string into an obfuscated string.

First an EVP\_CIPHER object is instantiated as ‘ctx’ to use EVP\_aes\_256\_cpc() to implement the AES algorithm with a 256-bit key in CBC mode, then this context is initialized using EVP\_CIPHER\_CTX\_new().



Next the cipher context ‘ctx’ is set up to initialise the encryption context with a cipher type by calling the EVP\_EncryptInit\_ex() library function. This ensures that the key sizes set are appropriate for the selected cipher type and block size.



EVP\_EncryptUpdate() is then called to encrypt the actual plaintext string. The provided input message bytes are read in from the buffer, encrypted and outputted through the encrypted buffer, ciphertext. An integer value for the length is also returned by reference parameter so as to ensure the actual data length is verifiable and is no greater than the block size.



If ‘padding’ is enabled, as is set by default, EVP\_EncryptFinal\_ex() is called to finalise the process by encrypting any data that remains from a partial block. This is done in situations where the data is not of a complete block size and further ciphertext bytes may need to be written to ensure that the final encrypted data length is of the correct size.



Finally, EVP\_CIPHER\_CTX\_free() is called to clear all information from the cipher context, thus freeing any allocated memory, including ‘ctx’ itself. It is essential for security purposes that this be the final step in the encryption process so the sensitive data being obfuscated does not continue to remain in memory after the function has completed (OpenSSL(a), n.d.).



#### Decryption

The string decryption process functions almost identically to the encryption process above, only in reverse.

As before, a new context ‘ctx’ is created and instantiated by calling the EVP\_CIPHER\_CTX\_new() library function. The decryption operation is then initialized as AES 256-bit CBC as before, this time calling the EVP\_DecryptInit\_ex() library function to initialize the decryption operation.

Next EVP\_DecryptUpdate() is called to perform the actual decrypting of the ciphertext back into readable plaintext and EVP\_Decrypt\_Final\_ex() deals with any padding such that the decrypted data out buffer must have enough room for the required number of bytes, unless the cypher block size is 1.

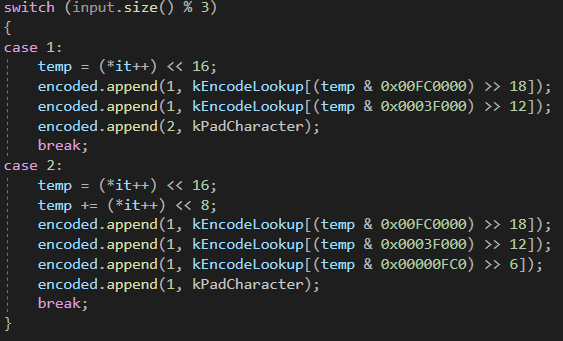
Finally as with the encryption process, the EVP\_CIPHER\_CTX\_free() library function is called to free all allocated memory and remove all references to the decryption data (OpenSSL(a), n.d.).

### Base64 Encoding

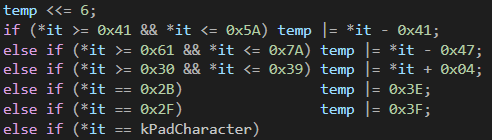
To avoid the insertion of erroneous and unwanted escape characters finding their way into the encrypted file lines it was decided to enhance the file security by using Base 64 Encode to alter the binary data of the strings into ASCII text. By doing this it avoids the concern that binary data may be misinterpreted as control characters or special characters such as a null terminator ‘\0’ or new line ‘\n’.

The Base64 algorithm takes 6-bit binary values that represent numbers from 0 to 63 then maps them to a set of printable ASCII characters, with the 64 characters themselves representing all upper and lowercase letters, the numbers 0 through 9 and the characters ‘+’ and ‘/’. Base64 encoding encodes 6 bits of data at a time, so every 6-bit portion of data is mapped directly to a single character output from the 64-character supplied array.

Where the length of the input string is not a multiple of 3 bytes, the missing length is padded with zeroes in multiples of sixes allowing that final character to be encoded. In a situation where one bytes is required in the input to make up a multiple of three, the character ‘=’ is added to the end of the string as padding. In situations where two bytes are required, the same character ‘=’ is added twice. This is done by testing the result of the input size modulo 3. If there is no remainder, then the input string is determined to be of the correct size and requires no padding. If the remainder is either 1 or 2, then some padding is required so as to create a string of the correct length.



Decoding of a string works similarly to the encoding process only in reverse. An iterator ‘it’ is set so it is pointed to the beginning of the input string. Next it cycles through the inputted string, iterating over it in 4-character chunks with the understanding that the final characters may be padding characters added during the encoding process. Each character is tested to ascertain which of 4 given ASCII ranges it falls within, with this determining how it will be successfully decoded into the original text.



As each character is decoded, it is pushed back into a vector byte array which can be returned as a single object once the decoding is completed.

# User Interface: Front-End Interactivity

A basic graphical user interface has been developed as proof-of-concept to investigate if it were possible for a GUI developed in one higher level language such a Python or C# could successfully communicate with a lower level language such as C++. As multiple programming languages would need to be implemented so as to send data strings between them, it was important that a method of communication should be implemented such that the string values being transmitted between them can be received and understood.

One of the concepts behind this is that by using a separate user interface that could be installed on multiple client machines, it may be possible for multiple users to be ‘logged on’ simultaneously and securely to the underlying software.

The technique implemented was a basic TCP client/server, with the front-end client being developed in the C# and the back-end server being developed in C++ to correspond with the language of the main software code base. TCP as a communications protocol, provides a secure method of exchanging messages over a network, guaranteeing the integrity of data transmission and ensuring end-to-end delivery of messages in both directions to and from the server. For our purposes here both server and client are local to the same machine, but in theory a remote server could be implemented where the client and server communicate over a LAN via sockets.

## C# Client

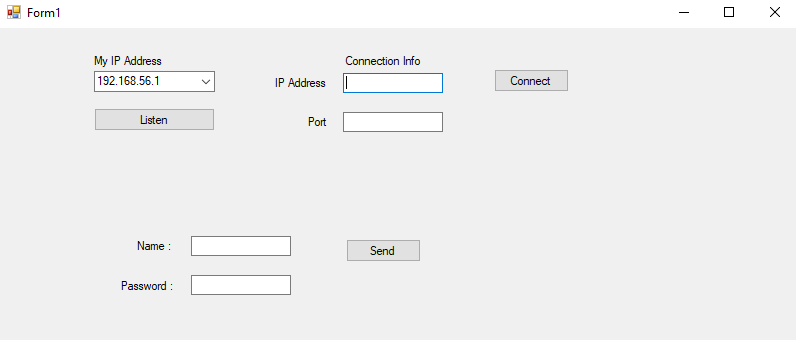
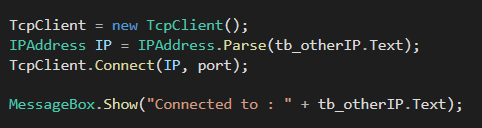


Fig. 11 – Client-End User-Interface

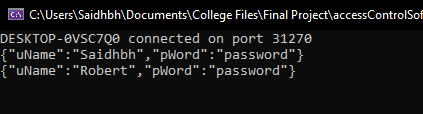
The Client first gets the host name for the local machine by calling the ‘getHostName()’ function. This is in turn supplied the getHostEntry() function to return a list of IP addresses relating to the host name which can be used to access the network and are used to populate a drop-down menu on the user interface. The user can then supply a port number (any free port number will suffice) and select an IP address from the menu to use as the local IP for connecting to the server.

Once the server side is running and ‘listening’ for any devices attempting to connect, the user can initiate a connection to the server by clicking ‘Connect’ on the interface. On clicking, an instance of the TcpClient class is created and the IP address text contained in the GUI textbox is parsed to an ‘IPAddress’.



TcpClient then calls the ‘Connect()’ method, connecting the client to the remote host using the supplied IP address and port number. It is now possible for the client to send message strings directly to the host.

The C# language lends itself quite nicely to data transmission via a network connection, providing a script serialization library that can be included in the program. When a user inputs a name/password pair into the provided text boxes and slicks the ‘Send’ button, both string values are used to populate the member variable of the LoginDetails class. The ‘JavaScriptSerializer’ method, Serialize(), is then called to convert the class object to a JSON string which gets written to the Network Stream as an encoded byte array.



Provided the client/host connection was successful, the user object containing the login details for a user can be received as a raw JSON string.

## C++ Host

Implementing a successful C++ server was a moderately more complicated task as many of the functionalities provided by the C# language are not as intuitively presented in C++. To implement a TCP/IP connection in C++ it was necessary to include the ‘WS2tcpip.h’ header file to access the relevant libraries.

First it is necessary to initialise the use of Windows Sockets (Winsock) before making additional function calls using WSAStartup() to initiate the use of Winsock DLL and by passing the Winsock version specification and a pointer to WSADATA that receives details of the Winsock implementation.

Next a listening socket is created using socket(AF\_INET, SOCK\_STREAM, 0), with AF\_INET indicating the address family that the socket is permitted to communicate (i.e. IPv4 addresses) and SOCK\_STREAM being a TCP connection data-stream providing two-way byte stream transmission. We use SOCK\_STREAM here as opposed to SOCK\_DGRAM as we wish the data-stream to be reliable and delivered completely via a connection with packet delivery confirmed.

Next we must bind an IP address and port to a socket. First we set the address family to AF\_INET as mentioned above, the port number to be any available port and assign the value INADDR\_ANY to the IP address. This is done in cases, such as this, where don’t wish to bind the socket to any specific IP as we don’t know the address of the machine. Then the port/address data provided is bound to the listening socket using bind().

The server/host must now wait for a client to attempt to make a connection. The sockaddr\_in struct is initialised for the client, and a client socket is created ready to permit an incoming connection attempt on this socket.

Once a connection has been successfully made to the client it is possible to retrieve the host name and port number through which the client is connected. It is now a matter of waiting for the connected client to send a message. Once a message has been sent to the host, the recv() function accepts the message as a character array, through the specified client socket. As shown above, it is possible to view the actual message contents by printing to console.

While not yet implemented, it is possible to instigate a two-way conversation between client and server by calling the send() function to return a message back to the client in the same manner in which a message was received. This TCP communication can now be maintained indefinitely until the client end disconnects from the server.

Regarding the messages being received by the C++ server in JSON String format, it might be preferable to implement a different method of encryption or serialization when communicating over the network. While it is incredibly easy to both serialize and deserialize a message string using C# as the functionality is already available in the available libraries, the reverse cannot be said for C++. There doesn’t appear to be any easily implemented method of deserializing a message sent in JSON format, and while it is not the most difficult thing to decode, it may require the developing of a decoding algorithm to translate the sent strings into usable variables and strings.

# Conclusion:

As software systems become increasingly more complex the need for robust and dependable security measures to be incorporated into the overall system architecture is undeniable. Access Control Software in many ways acts as a front-line defense against unwanted intruders, and is arguable the most fundamental security mechanism in use in software development today. It is a single access point through which all movement into the more sensitive areas of a system flows and is regulated. It filters users through authorization and authentication filters to verify users and assign permissions, deciding who can do what.

Security implementation should never be an afterthought. When security measures are implemented at the architecture level of development, they become part of the core structure. The use of strong secure code practices needs to be factored into the fundamental design so every element of the system is fortified.

Security design patterns have long been the tried and tested method of upholding the principles of CIA; confidentiality, integrity and availability. Confidentiality being the need to keep information and user details secure away from prying eyes, integrity of knowledge that information is protected from unwanted modification and destruction and availability, where all information is readily available to any user who is fully authorized to access it.

By successfully developing an interconnected series of security patterns it is possible to successfully protect any device or system, while allowing ease of access to any legitimate and verified user.

# References:

Agoyi, M. and Seral, D. (2010) “SMS Security: An Asymmetric Encryption Approach”, *2010 6th International Conference on Wireless and Mobile Communications*, pp. 448-452. IEEE.

Alexander, C. (1977), *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Anand, P., Ryoo, J. and Kazman, R. (2014) "Vulnerability-Based Security Pattern Categorization in Search of Missing Patterns", 2014 Ninth International Conference on Availability, Reliability and Security, pp. 476-483, doi: 10.1109/ARES.2014.71.

Assencio, D. (2015), “Generating Random Numbers in C++”. Available at: https://diego.assencio.com/?index=6890b8c50169ef45b74db135063c227c [Accessed 5 April 2022]

Avasthi, S. and Sanwal, T. (2016). “Biometric authentication techniques: a study on keystroke dynamics”. *International Journal of Scientific Engineering and Applied Science (IJSEAS)*, *2*(1), pp.215-221.

Booch, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Connallen, J. & Houston, K.A. (2007) *Object-Oriented Analysis & Design with Applications.* (3rd ed.) Pearson Education Inc.

Brown, F.L., DiVietri, J., de Villegas, G.D. and Fernandez, E.B. (1999) “The Authenticator Pattern”, *Proceedings of Pattern Language of Programs, 6th Conference*, pp. 15-18.

Dougherty, C., Sayre, K., Seacord, R.C., Svoboda, D. and Togashi, K., (2009). *Secure Design Patterns*. 47, Carnegie-Mellon University Pittsburgh PA. Software Engineering Inst.

Erlingsson, U. (2004) *The Inlined Reference Monitor Approach to Security Policy Enforcement*. Cornell University.

Fernandez, E. B. (2013) *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. West Sussex, United Kingdom: Wiley.

Ferraiolo, D. F., Chandramouli, R. & Kuhn D. R (2007) *Role-Based Access Control*. 2nd ed. Boston: Artech House.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. and Patterns, D., 1995. *Elements of reusable object-oriented software*. Reading, Massachusetts: Addison-Wesley.

Heckman, M.R. and Schell, R.R. (2016), “Using Proven Reference Monitor Patterns for Security Evaluation”. *Information*, *7*(2), doi:10.3390/info7020023.

Hamza, A. and Kumar, B. (2020), “A Review Paper on DES, AES, RSA Encryption Standards”. *2020 9th International Conference System Modelling and Advancement in Research Trends (SMART)* (pp. 333-338). IEEE.

Heyman, T., Yskout, K., Scandariato, R. & Joosen, W. (2007) “An Analysis of the Security Patterns Landscape”. *Third International Workshop on Software Engineering for Secure Systems (SESS'07: ICSE Workshops 2007)*. IEEE.

Jaeger, T. (2011), “Reference Monitor”. *Systems & Internet Infrastructure Lab. Pennsylvania State University.*

Kienzle, D.M., Elder, M.C., Tyree, D. & Edwards-Hewitt, J. (2002), “Security Patterns Repository Version 1.0”, *DARPA, Washington DC*.

Nitaj, A. (2012) “The Mathematical Cryptography of the RSA Cryptosystem”, *Protocols, Design and Applications* (pp. 159-186). Nova Science Publishers, Inc.

O'Gorman, L. (2003), "Comparing Passwords, Tokens, and Biometrics for User Authentication", Proceedings of the IEEE, 91(12), pp. 2021-2040, doi: 10.1109/JPROC.2003.819611.

OpenSSL(a) (n.d.), “EVP\_CYPHER\_CTX\_new” Available at: https://www.openssl.org/docs/manmaster/man3/EVP\_CIPHER\_CTX\_new.html [Accessed 28 April 2022]

OpenSSL(b) (n.d.), “EVP Symmetric Encryption and Decryption”, Available at: https://wiki.openssl.org/index.php/EVP\_Symmetric\_Encryption\_and\_Decryption [Accessed 5 April 2022]

Rihan, S.D., Khalid, A. and Osman, S.E.F. (2015) “A Performance Comparison of Encryption Algorithms AES and DES”, *International Journal of Engineering Research & Technology (IJERT)*, *4*(12), pp.151-154.

Rosado, D.G., Fernandez-Medina, E., Piattini, M. and Gutierrez, C. (2006) “Comparison of Security Patterns”, *International Journal of Computer Science and Network Security*, *6*(2), pp.139-146.

Saunders, G., Hitchens, M. and Varadharajan, V. (2001) “Role-Based Access Control and the Access Control Matrix”, *ACM SIGOPS Operating Systems Review*, 35(4), pp. 6-20, doi: 10.1145/506084.506085.

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F. and Sommerlad, P. (2013) *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons.

Scott, B. (2020) “Military-Grade Encryption Explained”. Available at: https://nordpass.com/blog/military-grade-encryption-explained/ [Accessed 30 April 2022]

Sood, S. K., Sarje, A. K. & Singh, K. (2009), "Cryptanalysis of Password Authentication Schemes: Current Status and Key Issues," 2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS), pp. 1-7, doi: 10.1109/ICM2CS.2009.5397977.

Steel, C., Nagappan, R. & Lai, R. (2006), *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Pearson Education.

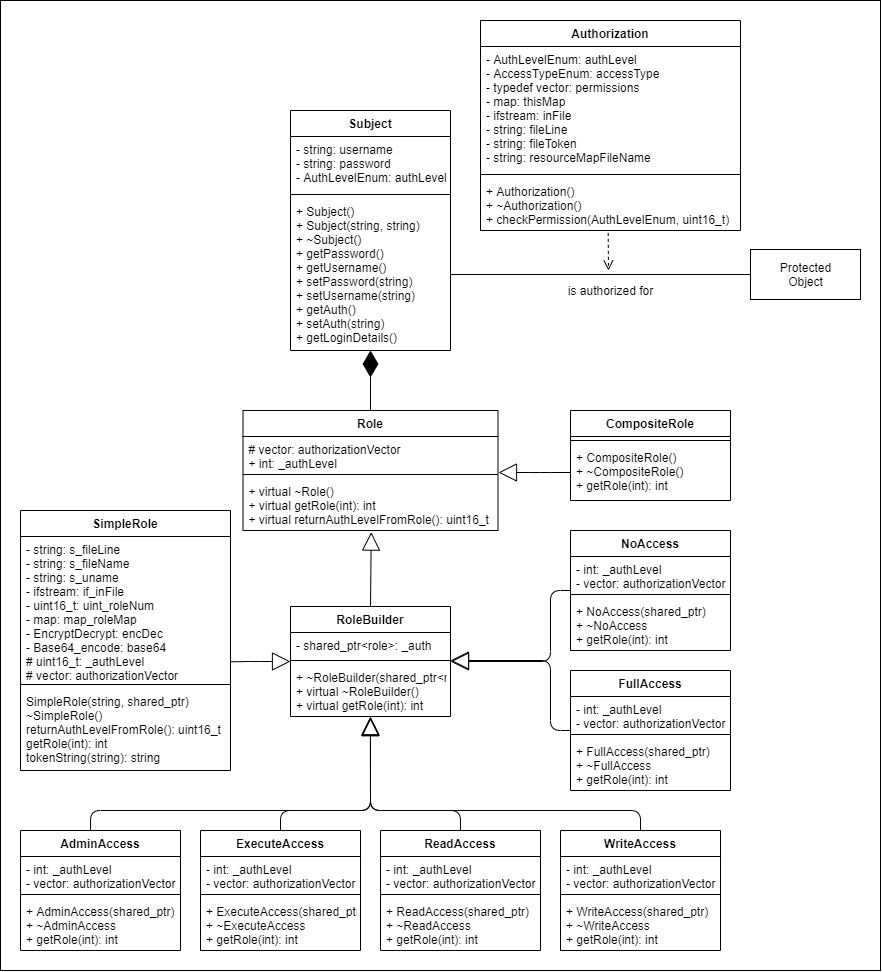
Thakkar, J. (2020) “Types of Encryption: 5 Encryption Algorithms & How to Chose the Right One”. Available at: https://www.thesslstore.com/blog/types-of-encryption-encryption-algorithms-how-to-choose-the-right-one/ [Accessed 25 April 2022]

Yoder, J. and Barcalow, J. (1998) “Architectural Patterns for Enabling Application Security”, *Proceedings of the 4th Conference on Patterns Language of Programming*, vol. 2.

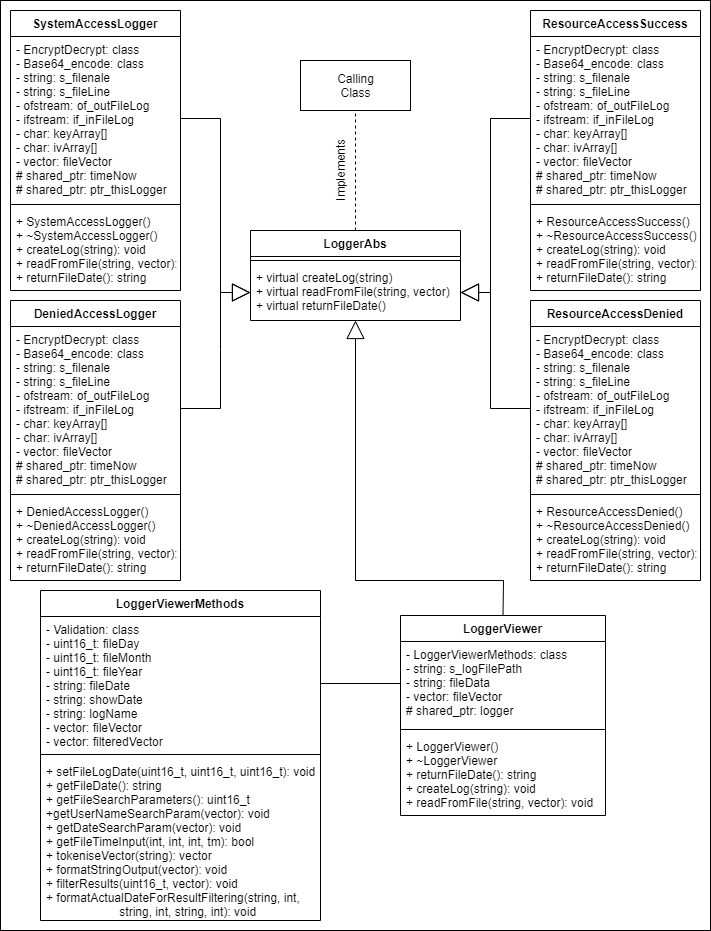
Zeebaree, S.R. (2020), “DES Encryption and Decryption Algorithm Implementation Based on FPGA”. *Indones. J. Electr. Eng. Comput. Sci*, *18*(2), pp.774-781.

# Appendix:

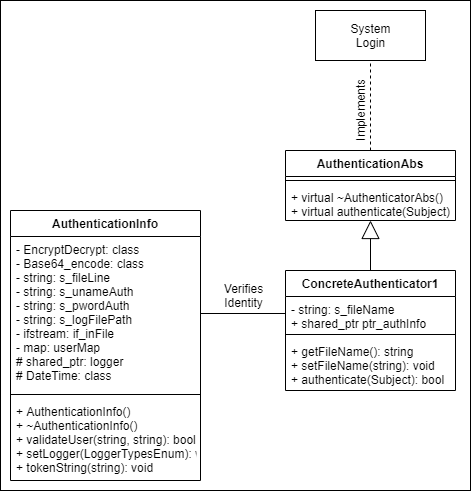
## Appendix A: UML – [RBAC/Authorization](#_Role-Based_Authorization)



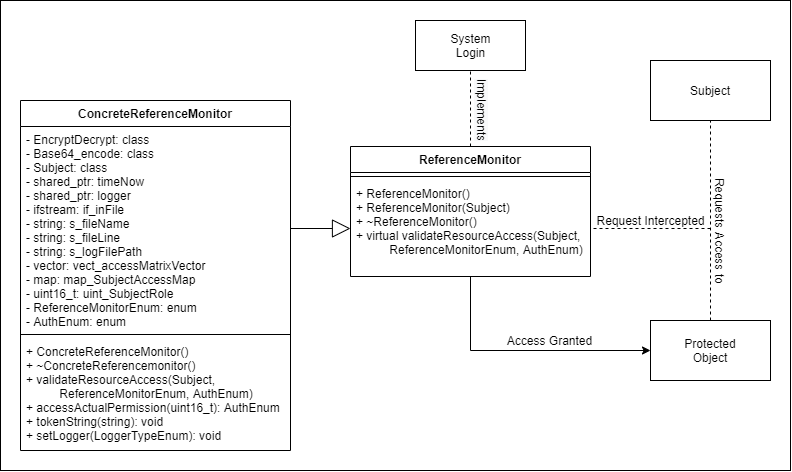
## Appendix B: UML – [Logger/Auditor](#_Logger-Auditor)



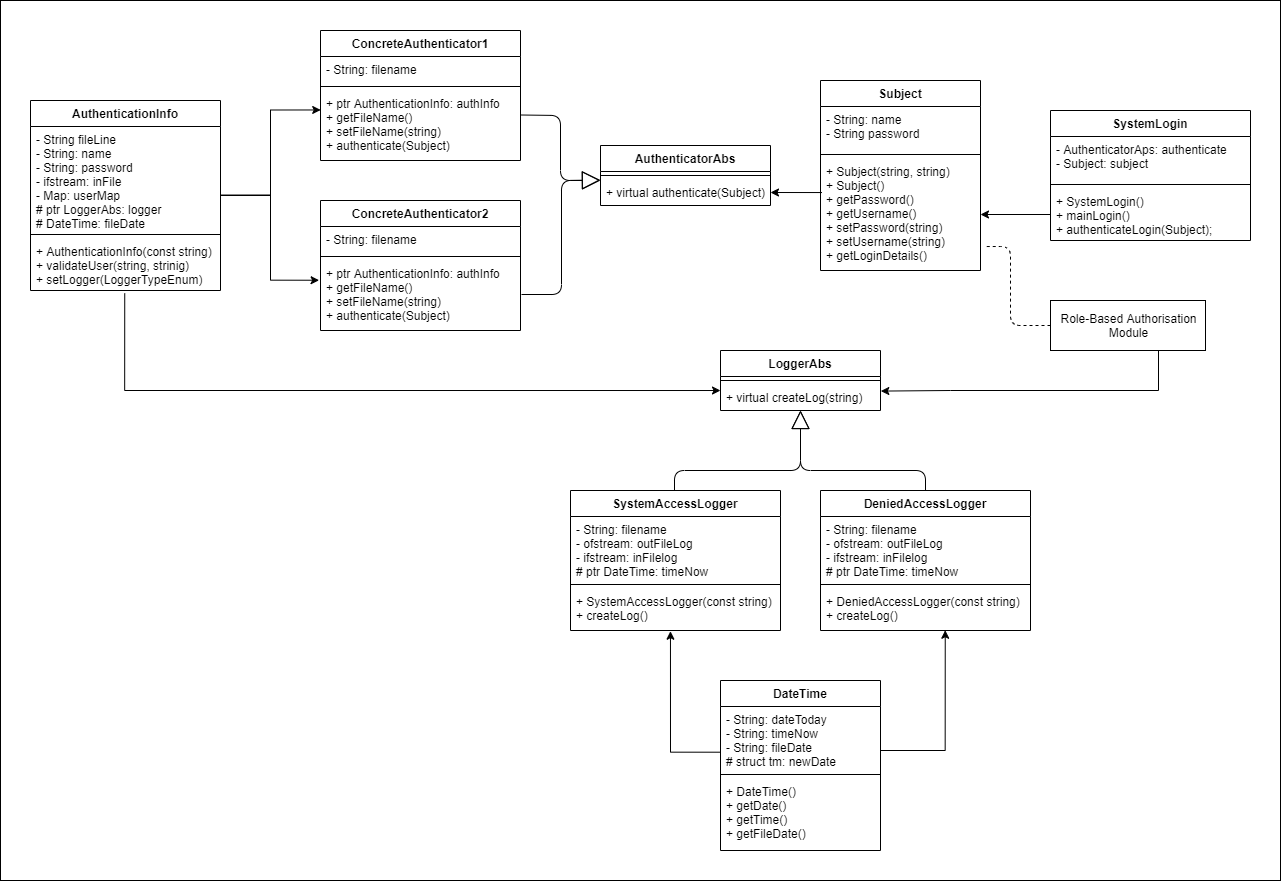
## Appendix C: UML – [Authenticator](#_User_Authentication)



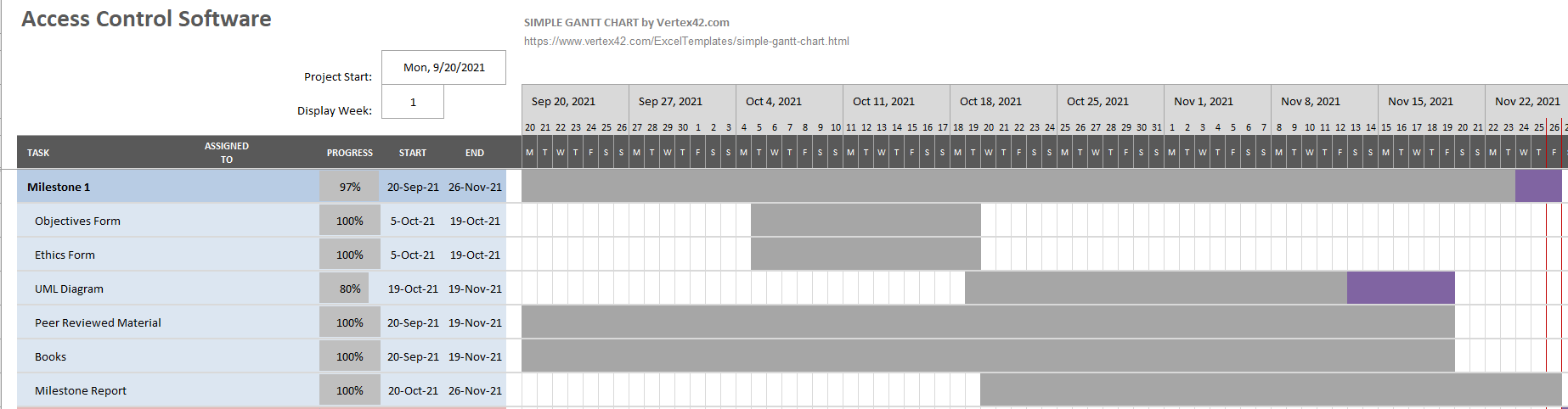
## Appendix D: UML – [Reference Monitor](#_Reference_Monitor)



## Appendix E: UML – Main Diagram



## Appendix F: Gantt Chart



(Original file attached)

## Appendix D: 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 | Saidhbh O Malley |
| Student Number | 30007730 |
| Student E-mail Address (please use University e-mail) | [30007730@students.southwales.ac.uk](mailto:30007730@students.southwales.ac.uk) |
| Name of Principal Project Supervisor | Janusz Kulon |
| Project Title | Access Control Software |
| 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. | The aim of the project is to develop software that controls access to any device from the Internet of Things by checking a user’s credentials. I do not foresee any elements of my research and/or testing to be relevant to an Ethical Evaluation |
| 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. | I intend to create software which will determine a user’s authorisation to access a protected IoT system. Depending on their access level and/or role they will be authorised/denied entry to the system. All attempts, both successful and unsuccessful will be logged for monitoring in an encrypted permanent record. Any testing performed during the final stages of development will most likely be undertaken by me, my fellow students and/or staff members on a voluntary basis. No personal data will be stored or entered into the system. |

**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.

1. 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

1. 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.
2. Requires full NHS ethical approval via the Integrated Research Application System.
3. Requires a Human Tissue Act license.
4. Involves “covert” procedures as in covert observation studies.
5. 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).
6. 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.
7. Involves administration of drugs, placebos or other substances (such as food substances or vitamins) as part of this study.
8. Involves invasive procedures (not limited to blood sampling, collection of biological samples, or passing current through a participant’s body, etc.).
9. Offers any financial inducements to participate in the study.
10. 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:  Date: |
| **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: *Janusz Kulon*  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.