Contents

[Design Overview 1](#_Toc475734090)

[Design Patterns Used 2](#_Toc475734091)

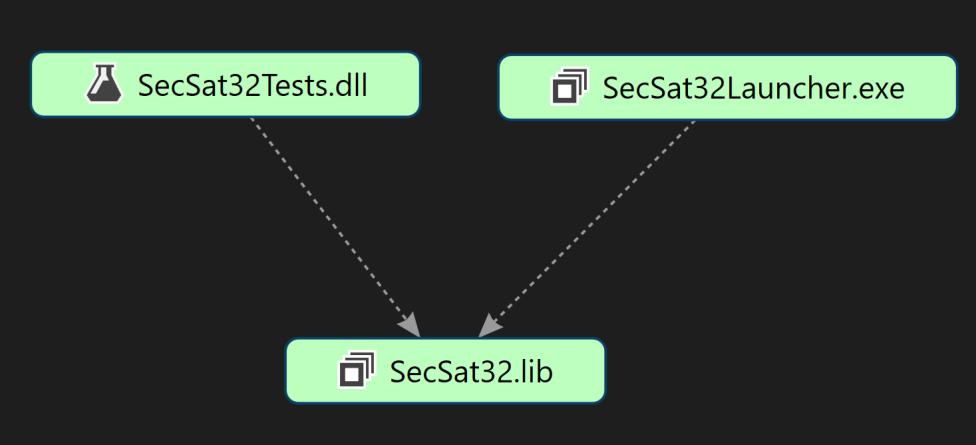
[Password Security 4](#_Toc475734092)

[Secure Types 6](#_Toc475734093)

[Requirements check 6](#_Toc475734094)

[Misc 8](#_Toc475734095)

# Design Overview



In order to allow the code to be unit tested, it has been written as a .lib static library project (SecSat32.lib) that both the test and launcher products can access. SecSat32Launcher.exe is an empty project that builds as an executable in order to allow the code in the library to run like an executable. Because there is no entry point (main) in SecSat32Launcher, it used the one in SecSat32.

The application has been built and tested using the x86 configuration for compatibility. It is not designed to be built using the x64 configuration and this has not been tested.

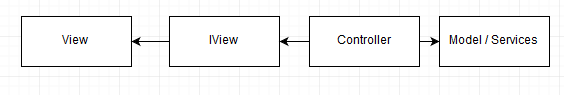
# Design Patterns Used

**Observer pattern:**

C:\Users\Mark\Downloads\Observer.png

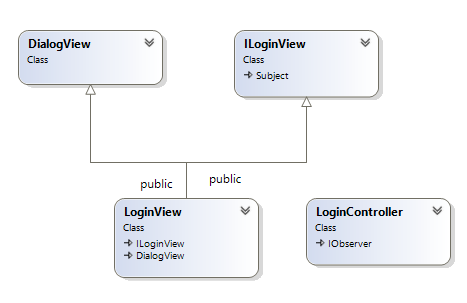
The observer pattern has been used to allow classes to observer other objects of interest, being notified when their state changes. This means that subjects may operate at their own pace, and may update as many observers as have subscribed. The above diagram shows a simplified view of how sensors owned by the satellite model inherit from Subject, and update the model whenever their state changes. The type of observer pattern used is a 'pull' pattern, where the it is the observers responsibility to interrogate the subject to gain the new information.

**MVC**

****

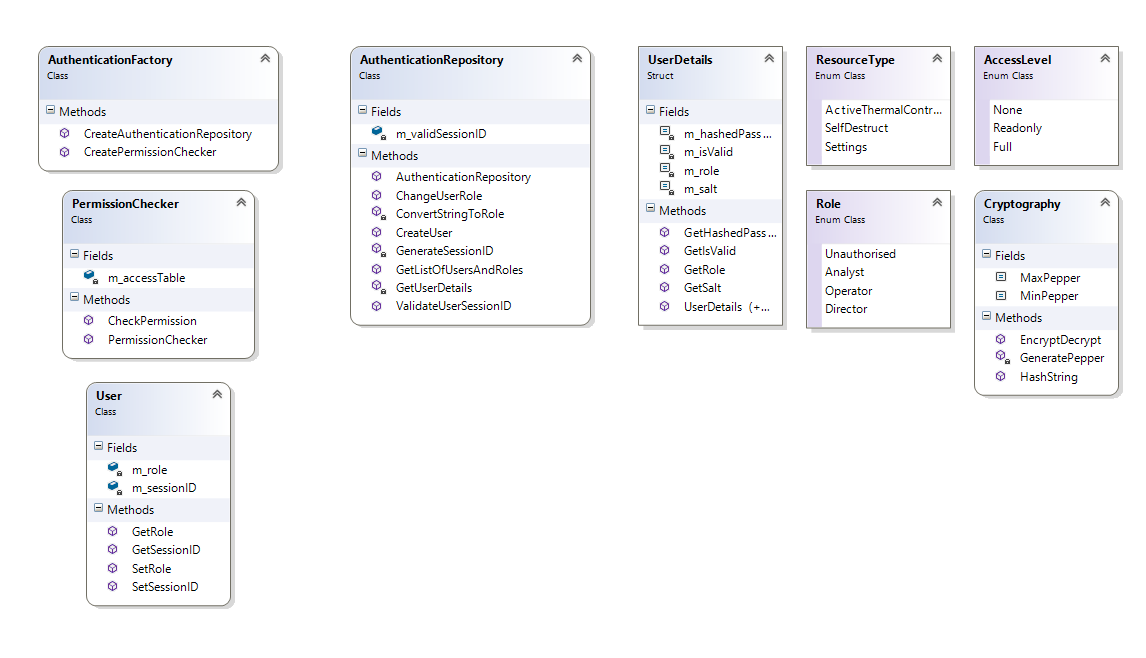
This is a pattern used to separate concerns between classes. An interface to the view is used so that it can be easily swapped out for any other kind of view. For example, Win32 dialogs have been used for this application but they could easily be changed to console windows, as long as the new view implements the IView interface.

To give an example when there is no model class, but instead the controller uses other classes such as the AuthenticationRepository:



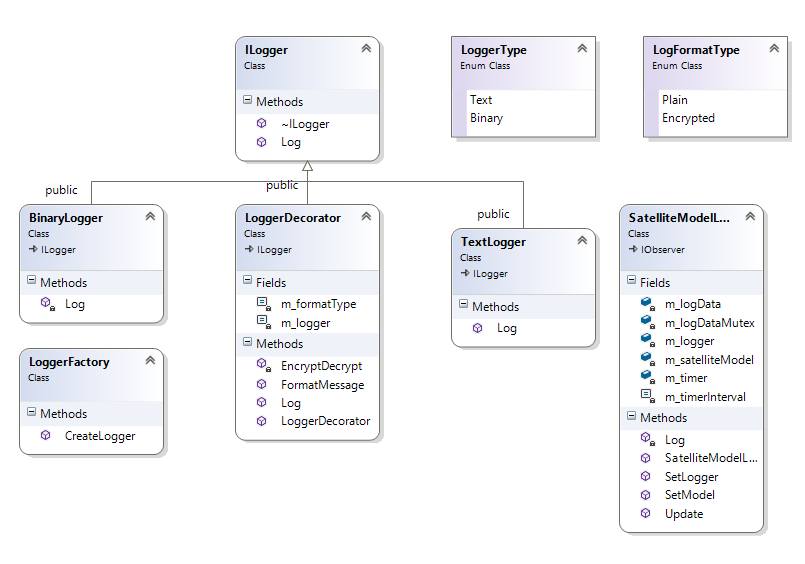
**Authenticator / Role Based Authorisation**

The authenticator/role based authorisation compound pattern has been implemented using the following classes:



**SecureLogger**

The secure logger pattern has been implemented using the following classes:



# Password Security

The following assumptions have been made about the threats to the software:

* Attackers have access to the source code, but not to the user database (with this, they could easily create a new user with full permission).
* Attackers may have a degree of access to the memory that the application is using to run.

|  |  |  |  |
| --- | --- | --- | --- |
| **UserID** | **Salt (512 bit)** | **Password** | **Hashed Password (Password + Salt + Pepper (8 bit))** |
| AldrinE | dC8MNLJN\*O\_Mgg#UX9Lr6tP@k3tNGvE%Vw-8q0Q4lIp&Mg$90dLo#Q0Vy1\*5PfyI | Fortune59Fairs13Dilemmas | 163f2224253e3565691631392223616314393c353d3d31233413681d1e1c1a1e7a1ff1d37377358691c2266240103b63241e172615756277d6821601643c1920761d37746960341c3f73160629617a6503629195e |
| ArmstrongN | p@ZIUFNRLNcp&CWrhkkBJ0l&0lTzN)DJKclbQ7li\_04fIrw8DNHOqsz!d9pF!4Ti | Siegmund92Dismayed95Painful | 33935373d253e3469621439233d312935346965031393e36253c2010a195161e21c1e33207613722383b3b121a603c76603c42a1e79141a1b333c321673c39f60643619222768141e181f21232a713469201671644397c |
| ConradP | &680vc1LWTM\_7847!LDcV$gKq\_5qtBeQ%4@ke3QQAHWW(kCpF#TJ85uB^-qObhC2 | Wielded65brothels54archly | 739353c343534666532223f2438353c236564312233383c29766668602633611c741df67686467711c1433674371b21f652124123517564103b356311111877783b1320167341a68652512e7d211f32381362ffffff9c |
| DevP | f-UDkanDt68rtnwM3&vZUkd7TgwvS(3v#S-G%dB9abFcPE&9YVuXGFKwCwy(nYpz | Reviver71Starred18precise | 235263926352267613243122223534616820223533392335367d5143b313e1424666822243e271d637626a53b3467437272637863267337d17753412693132163301576699625817161b27132729783e9202a72 |
| KulonJ | TRZm1-\_A5D--OcYb2vNhG$M^-8agAqiRyEefttOrnKBMQko\*%v%Qp\*&Yr96UFwn^ | Eligible12Tanya33Lawsuits | 153c393739323c3561624313e293163631c3127232539242342a3d617df1165147d7d1f3393262261e3817741de7d68313711213922915353624241f223e1b121d13b3f7a7526751207a769226966516273ee1c |
| PlassmannP | qta$mXafp0qvez&X6DipuPH)8z!K^LyLI4clDUDcETv#VXLzHZe-5Tza!mz%5ryL | wager13Diode11Swam | 2731373522616314393f34356161327313d212431743d8313620602126352a768661439202501879682a711be1c291c1964333c14514331542673681c2a18a357d6542a31713d2a756522291cffffff8e |

Roles have not been included because they can be changed, but DevP will have the highest level of access at the time of submission.

Passwords should not be stored in the source code of the application and they should not be stored in plain text. The accepted way of storing passwords is to salt and hash them using a proven hashing algorithm, optionally including a pepper as well as a salt.

Salt

The salt is a random string, ideally at least 128 bits in length and generated with a cryptographically secure pseudorandom number generator. This string is added to the password before hashing, which effectively defends against rainbow table attacks on the database. Using this technique has the added benefit of ensuring the resulting hash is unique to a high degree of probability.

The salts and passwords for this application have been generated using online random generators.

Hashing Algorithm

String = password + salt + pepper

For each character c in String

c = c XOR ‘P’, converted to hexadecimal

The hashing algorithm used is clearly not suitable for security purposes, but it serves as an example, without needing to include a third party cryptography library. More specifically, it lacks the following features of a good hashing algorithm:

* Practically irreversible
* Memory intensive
* Slow
* Same length output for any input

# Secure Types

In order to protect against common vulnerabilities, a number of secure classes have been created.

SecureDouble

The SecureDouble class is a wrapper around the native type 'double', which throws a SecurityException if an overflow is detecting when performing an operations such as an addition. The class makes use of the FE\_OVERFLOW and FE\_UNDERFLOW flags to detect errors.

(Pseudo code)

Clear all current flags

Perform the operation

Check the overflow and underflow flags

If there was an overflow or underflow, throw an exception

SecureInt32

The SecureInt32 and SecureUint32 classes throw exceptions if a wrap or overflow is detected.

SecureString

The SecureString class is designed to solve the problem of sensitive information remaining in memory after classes are destructed. It has been heavily inspired by the CERT [example](https://www.securecoding.cert.org/confluence/display/cplusplus/MEM03-CPP.+Clear+sensitive+information+stored+in+returned+reusable+resources).

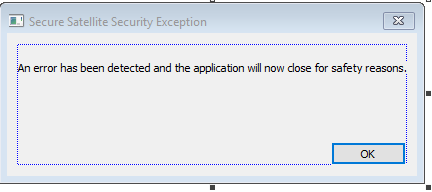
# Requirements check

**Requirement: Three sensor data inputs simulated**The three sensors simulated in the program measure temperature, solar radiation, and gravity anomaly.

**Requirement: Data processing of input from one sensor**The temperature sensor gives values in Kelvin. This is converted to Degrees Celsius after it is read.

**Requirement: State of one sensor changeable**Disabling the Active Thermal Control System (ATCS) stops the temperature simulator from only giving readings in the approximate range of the operating temperature of the satellite. Re-enabling the ATCS will cause the readings to steadily return back to an optimum temperature.  
In addition, destroying the satellite will cease readings from all sensors.

**Requirement: Persistent storage of sensor and device data in a log**Sensor readings are outputted to the log file (SensorData.txt) at a regular interval. Each log entry also includes the status of the ATCS, whether the satellite is operational or destroyed, and whether the satellite temperature is outside of its operational range.   
  
**Requirement: Interface to display sensor/device data and to manage the device**The Win32 user interface displays the current sensor readings , as well as a warning when the temperature is out of the operating range of the satellite. The interface can also be used to toggle the state of the ATCS, and to cause the satellite to self-destruct. When a security exception is caught, a dialog is show explaining what has happened (but not in so much details as to give away sensitive information):

****

**Requirement: User permissions implemented/Permissions are determined by the system administrator**These can be managed through the settings view, after selecting the settings button on the SatelliteView with the correct permissions.

**A number of references have been left as comments to show secure practices in use:**

DR1: Calling reset on the smart pointer before reassigning it prevents the memory it is pointing to from being leaked.  
If reset was not called, the pointer within the smart pointer would point to a new location in memory and we could no longer free what it was previously pointing to.

DR2: Performing checks when writing to a file.

DR3: Use of make\_unique in case of an exception between allocating memory and transferring ownership to a smart pointer.

D4: Checking to see if type has overflown or underflown

D5: Checking that incrementing the value will not cause overflow

D6: Checking that addition will not cause overflow

D7: Using secure version of strnlen to determine the size of the string

D8: Deleting memory before reassignment to prevent memory leak

D9, D10, D15: Ensuring the string is null terminated] CERT MEM03-CPP (Clear sensitive information stored in returned reusable resources)

D11: Correct use of delete[]

D12: Setting to nullptr to avoid dangling pointer

D13: Getting length of text to prevent buffer overflow  
The length of text in the textbox is obtained so that the buffer size can be allocated correctly.

D14: Use of smart pointer to handle memory management

D16: Initialising pointer to nullptr

D17: using cin for input into a string, with set width

D18: Checking file stream before and after use

D19: Avoiding dangerous cast by performing checks first

In addition, enum classes have been used. This prevents callers of methods from providing values that are out of range of the enumeration.

# Misc

Unit tests have been written but code coverage is very low.

Compiler and linker flags have been set correctly where possible. Because SecSat32 is a .lib project, it has not options to set the linker flags. The executable that calls it has all the correct linker flags set. /RTC1 has been turned off for release builds because it is only compatible with debug configurations.

Built in code analysis in Visual Studio has been used, using Microsoft recommended rules for native code. Cppcheck has also been used and all errors corrected, with the exception of a few false positives:

