Zuzana Baxendale

17047188

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document.   
When you’re ready to add your content, just click here and start typing.]

Documentation for secure sensor program

[Document subtitle]

Contents

[Design Patterns Used 2](#_Toc2348923)

[My Two Design Patterns 2](#_Toc2348924)

[Diagram of my two design patterns combined bellow 2](#_Toc2348925)

[Elements of Secure Software Development Used 3](#_Toc2348926)

[Secure Development of Strings and Integers 3](#_Toc2348927)

[Secure Development of File Reading/Writing 3](#_Toc2348928)

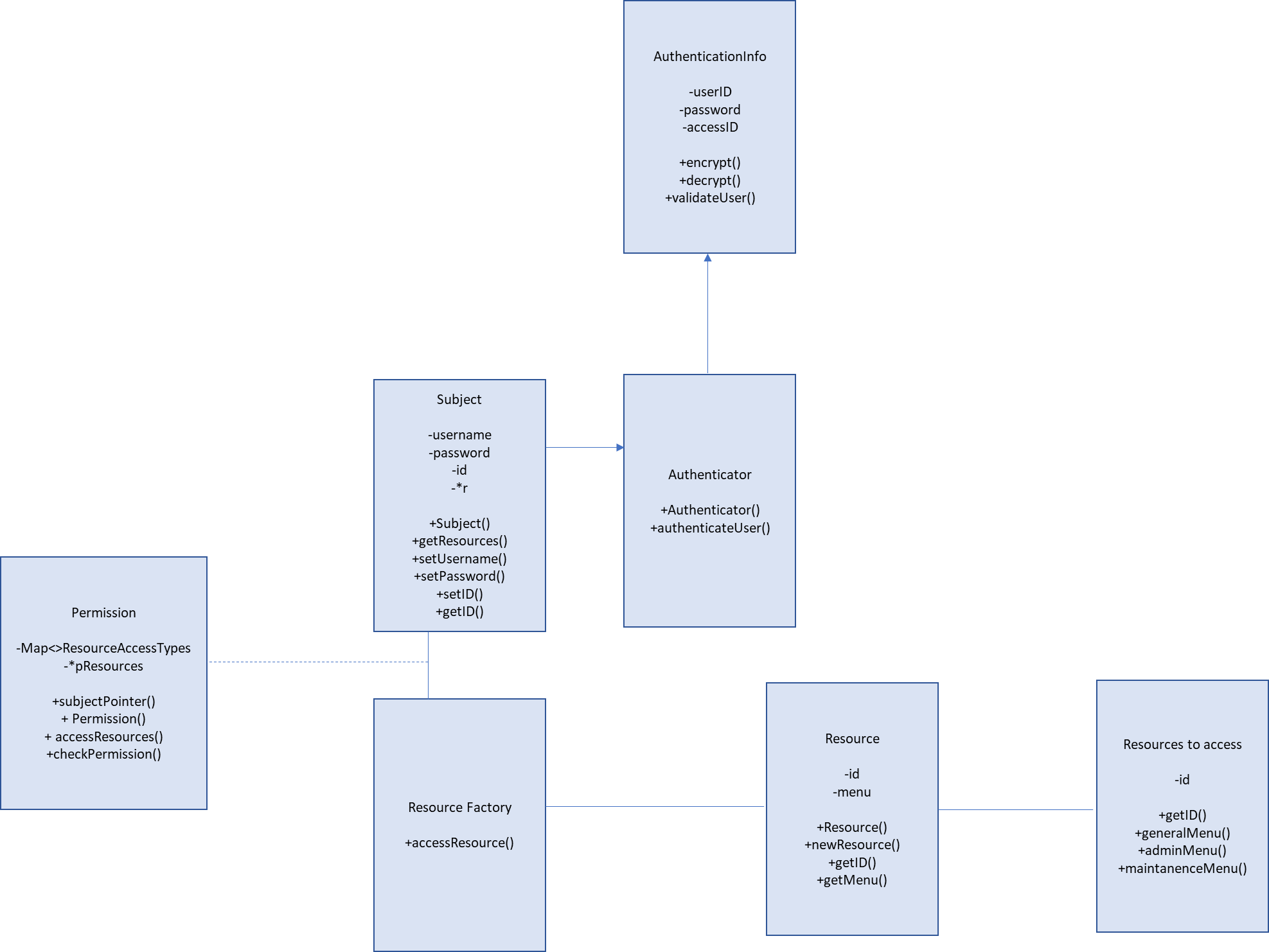
# Design Patterns Used

## My Two Design Patterns

I used a combination of the authorization and authentication pattern. I chose these two patterns because I felt they best fit the task at hand. The two design patterns worked well for my program as I knew I had to have a log in system and that different users would have different resources to access. It would be easy to create a new type of resource and a new user permission to access it. E.g. a lab tech user who can view the sensors diagnostics such as overheating for fans or how accurate a temperature reader is.

## Diagram of my two design patterns combined bellow

These design patterns can be linked with their common class ‘Subject’.



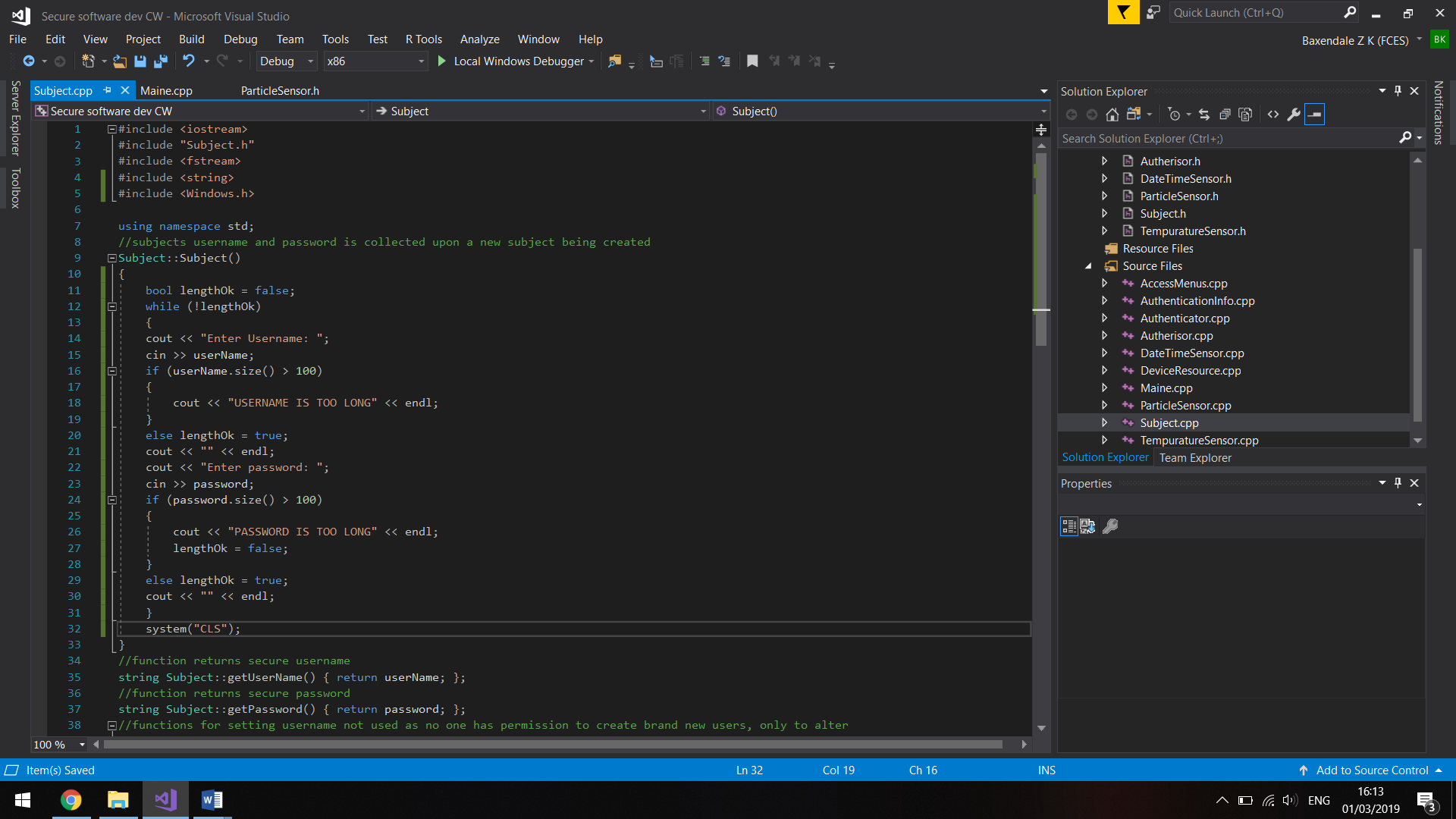
# Elements of Secure Software Development Used

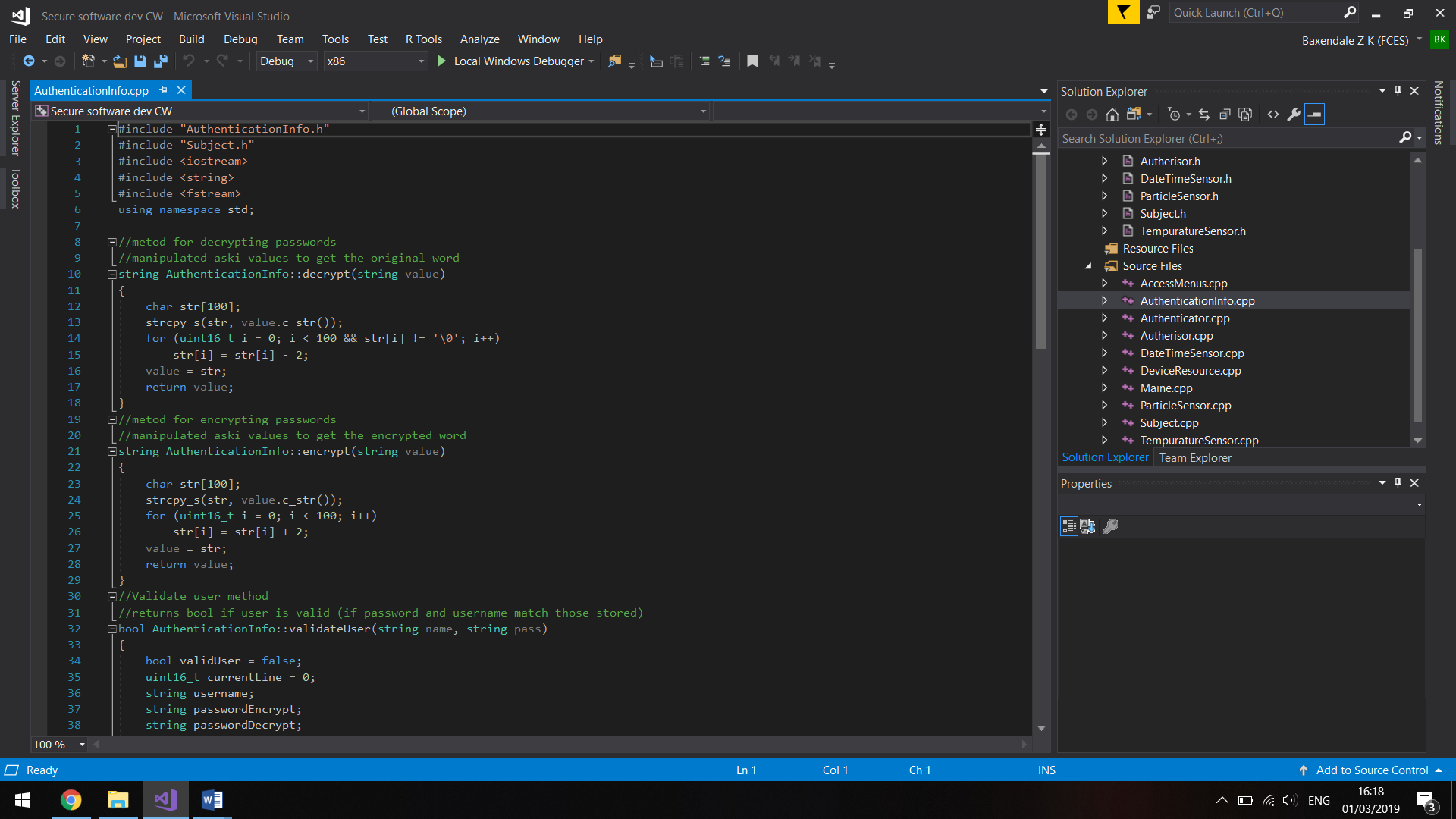
## Secure Development of Strings and Integers

1. When holding strings as character arrays there poses the issue of ensuring that there is enough space for that string to be held. If not enough space is allocated this creates a classic buffer overflow. Pointers will start trying to access memory from the heap that was not allocated to them and will cause a program to crash.

In order to avoid this I tried not to use character arrays where possible and used strings instead which expand dynamically.

However when I needed to assign a string to a character array in order to encode/decode passwords I gave the array 100 characters of space and ensured that no password was greater than 100:

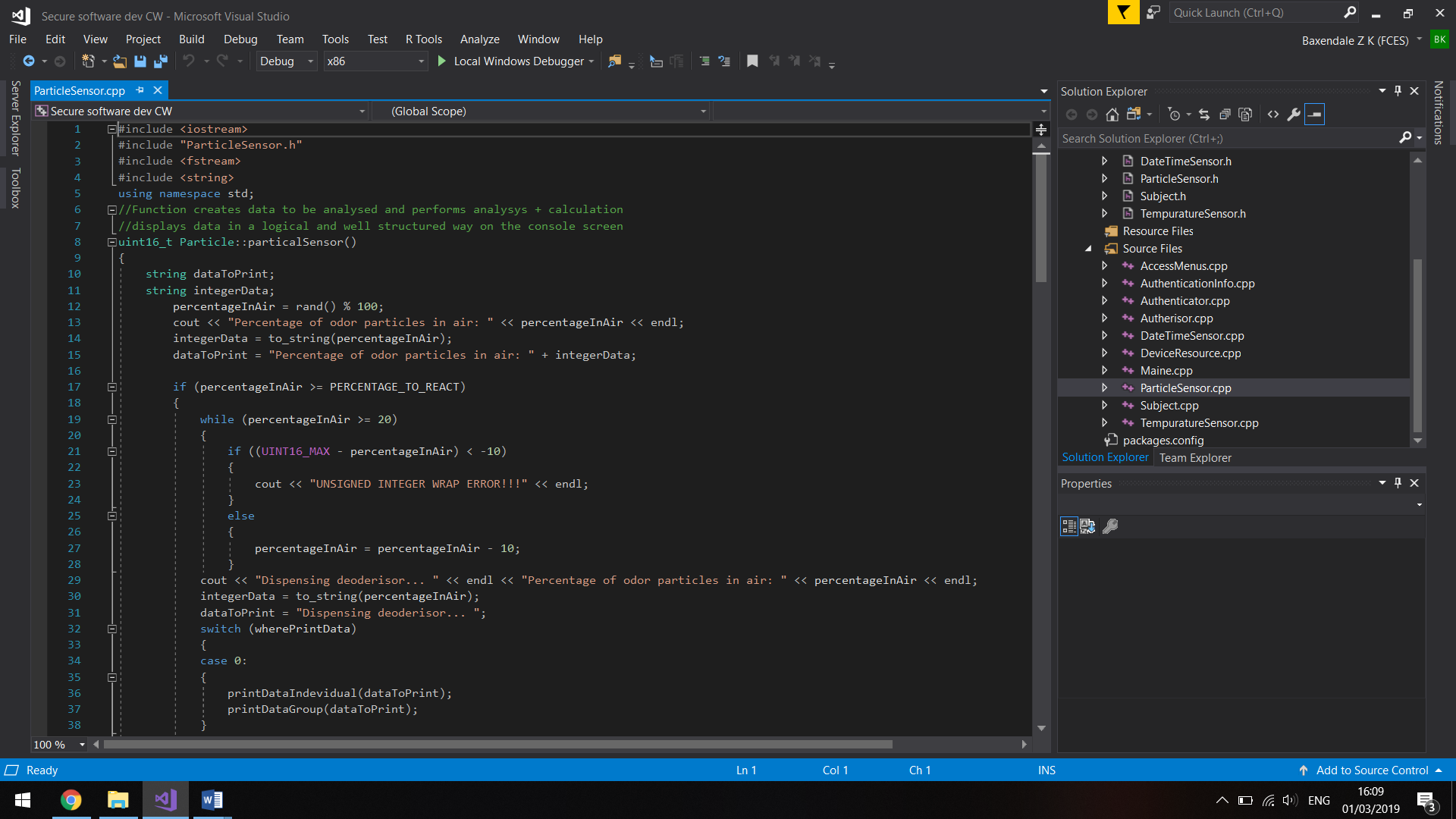
won’t let username of password exceed 100 characters because when password is searched for it must be cast into a character array in order to perform ASCII encryption. I allowed 100 characters in order to make sure no one caused pointers to access the heap.

Strings cast as a char array of 100.

(only time character strings were used in the program because they can be dangerous and I used strings where possible)

1. Unsigned integer wraps occur as a result of an integer trying to hold a number greater than UINT\_MAX (32,768) once this number is exceeded the integer will roll over to 0 and start incrementing from there.

This unsigned integer wrap can occur from the following operations being performed on an integer value, addition, subtraction and multiplication. In order to prevent this checks can be performed before any calculation checking that the outcome will not be an integer wrap. This is done by comparing whether the calculation would exceed the value of 32,768, as seen bellow:



Lines Used:

Particlesensor.cpp

Line 21

DateTimeSensor.cpp

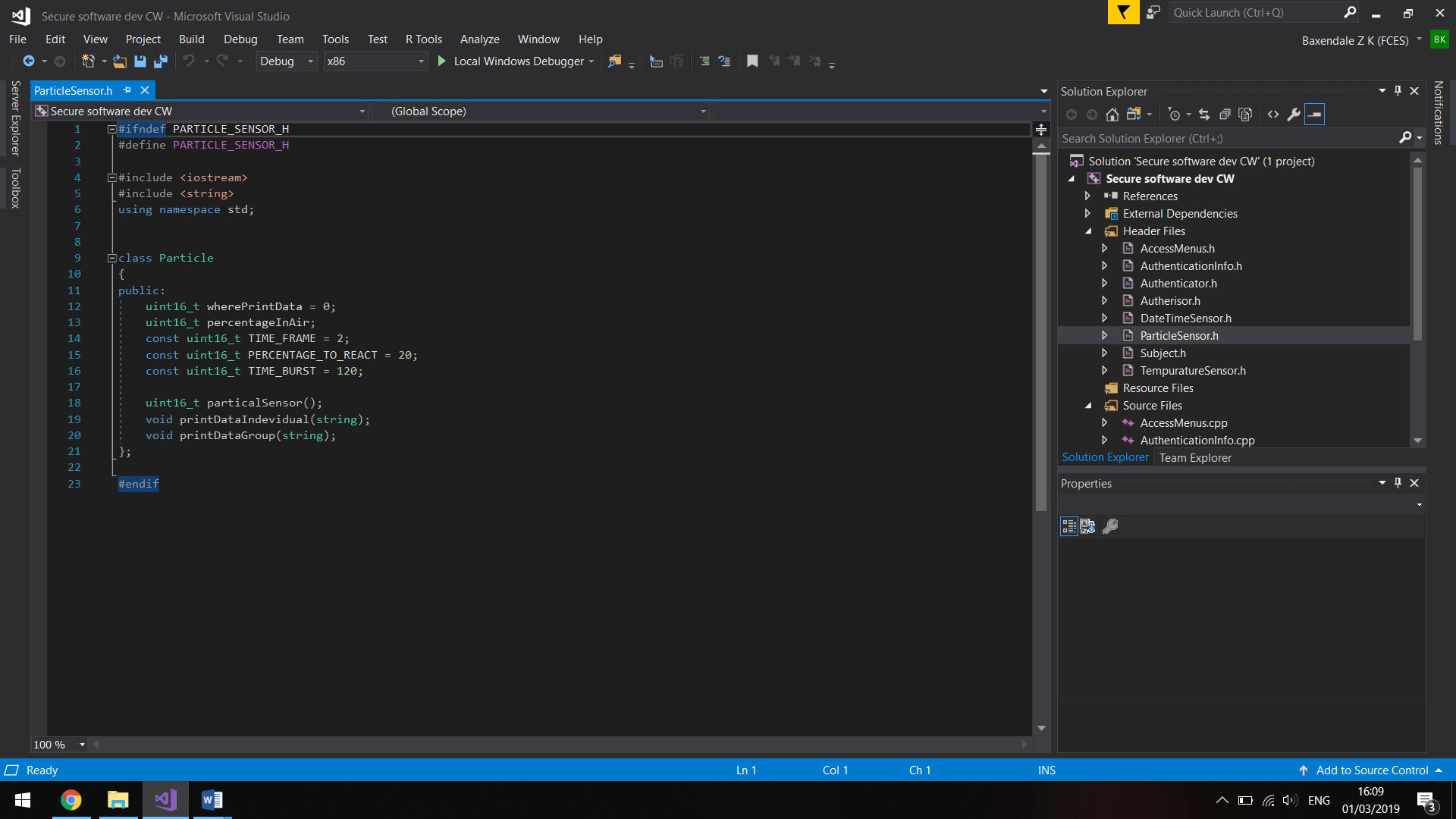
Line 135

AccessMenus.cpp

Line 482

Line 487

Also uint16\_t and uint32\_t are now used to ensure that the integer can only ever be unsigned:



Used through the whole program wherever an integer is sued that has no need to ever be signed.

1. Signed integer overflows can be triggered by many more operations, however my program did not require the use of any signed integers, but if I had I would have performed checks to ensure that the minimum and maximum limitations of the signed integer where not exceeded.
2. Type casting double variables can result in a loss of accuracy, for example, 50.654 becoming 50 when cast as an integer could be an issue. I did not encounter this but would prevent this type casting error by only casting integers as doubles if some comparison or compatibility is needed between the two

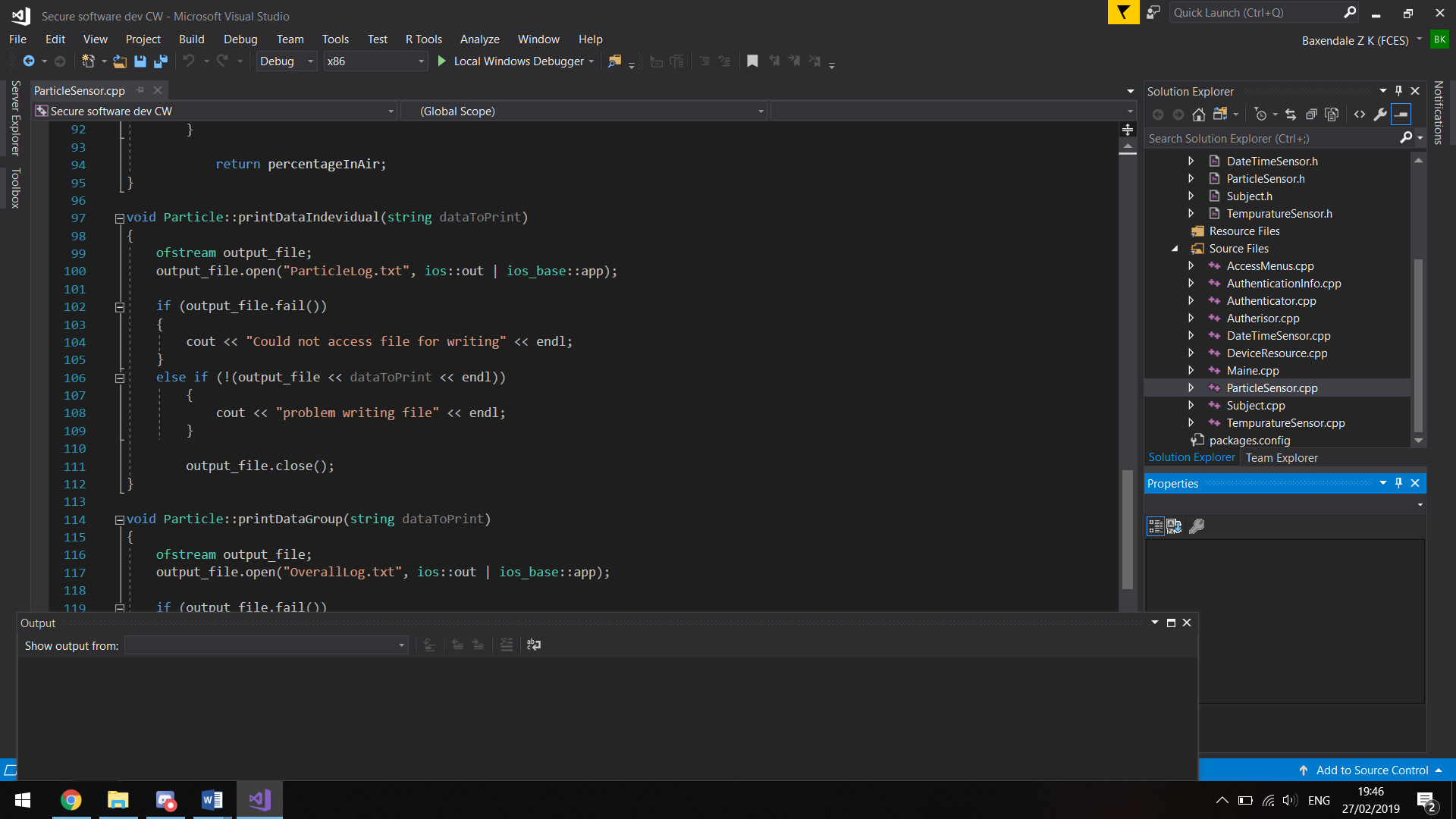
## Secure Development of File Reading/Writing

The checks performed when writing/reading to/from a file:

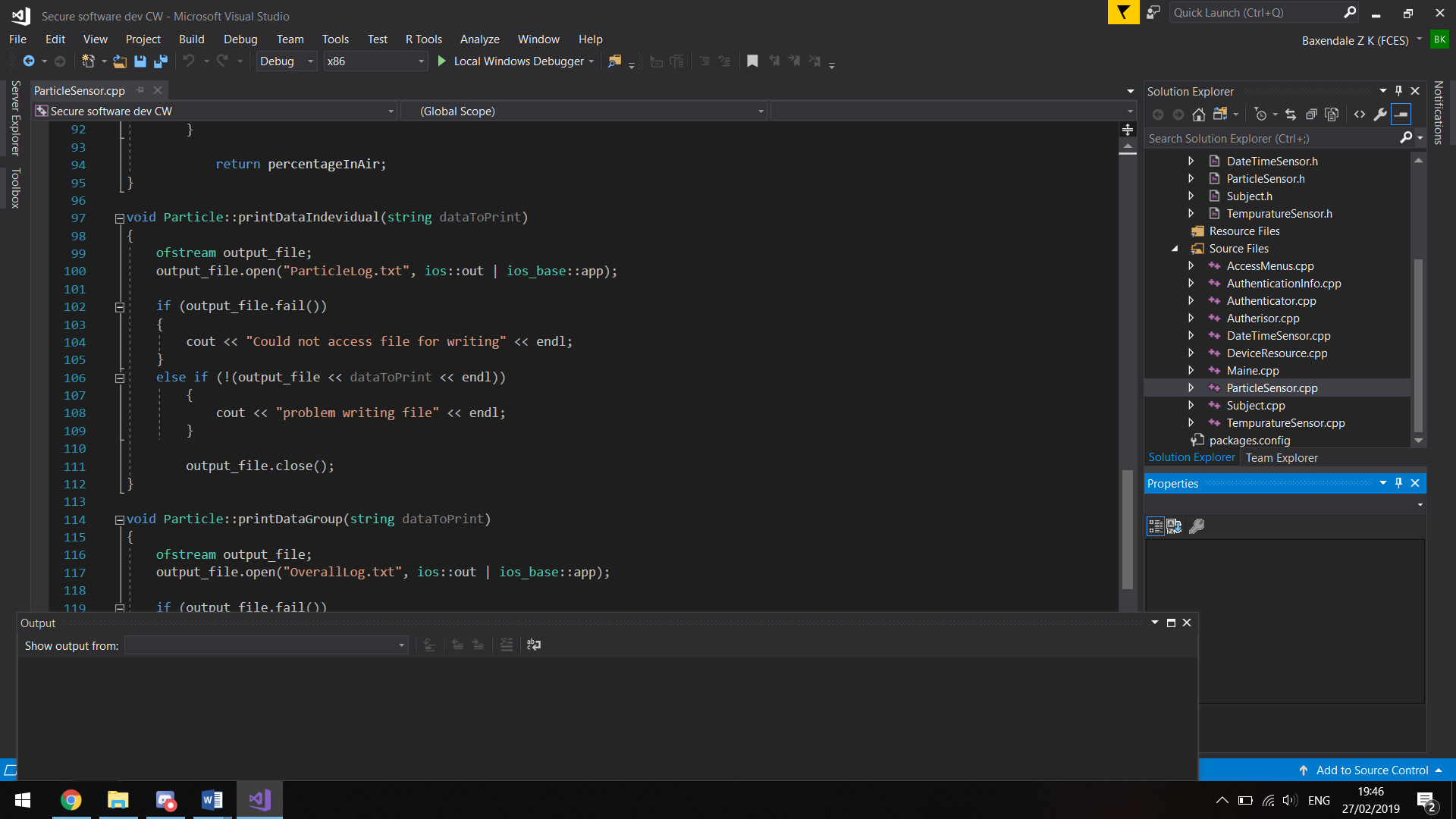
1. Can the file be opened? If the file cannot be opened it may have been moved, deleted or had its name changed. The security issue that commonly occurs with file accessing is TOCTOU race conditions (time of check time of use). This means that you should always perform checks the first time a file is accessed and also the next access (e.g. the next use of the file). This prevents a person maliciously editing or deleting a file to try and cause you to open a different file that could be dangerous.

I ensured this would not happen during any ifstreams and ofstreams as demonstrated below:

Check file can be opened (this ensures file with that name exists and that as a user we have permission to use it:



Check that writing to a file is possible (again checks our permissions and that a stable connection has been established):



Lines where it is used:

TempuratureSensor.cpp

Line 45

Line 61

ParticleSensor.cpp

Line 107

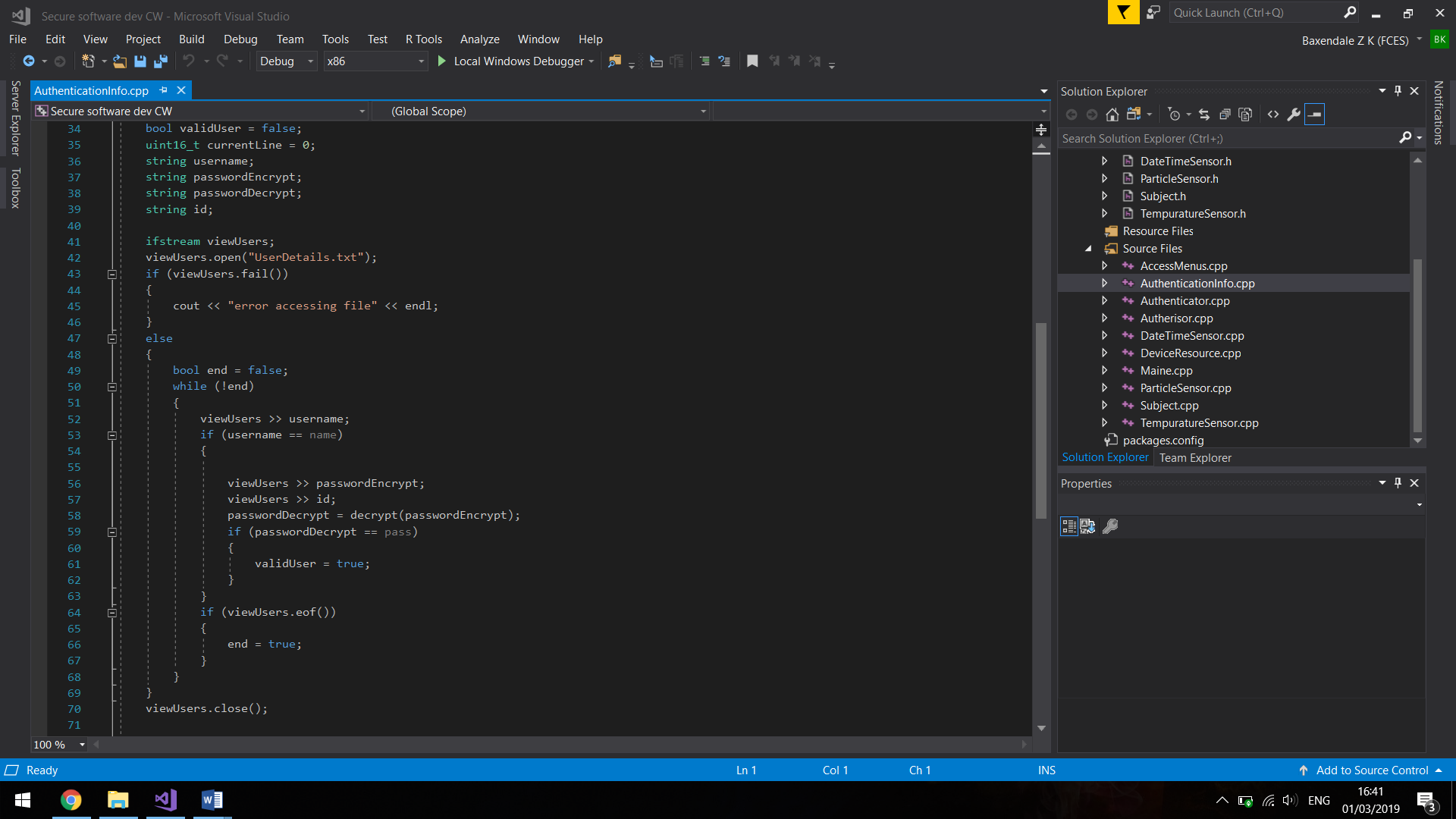
Line 124

DateTimeSensor.cpp

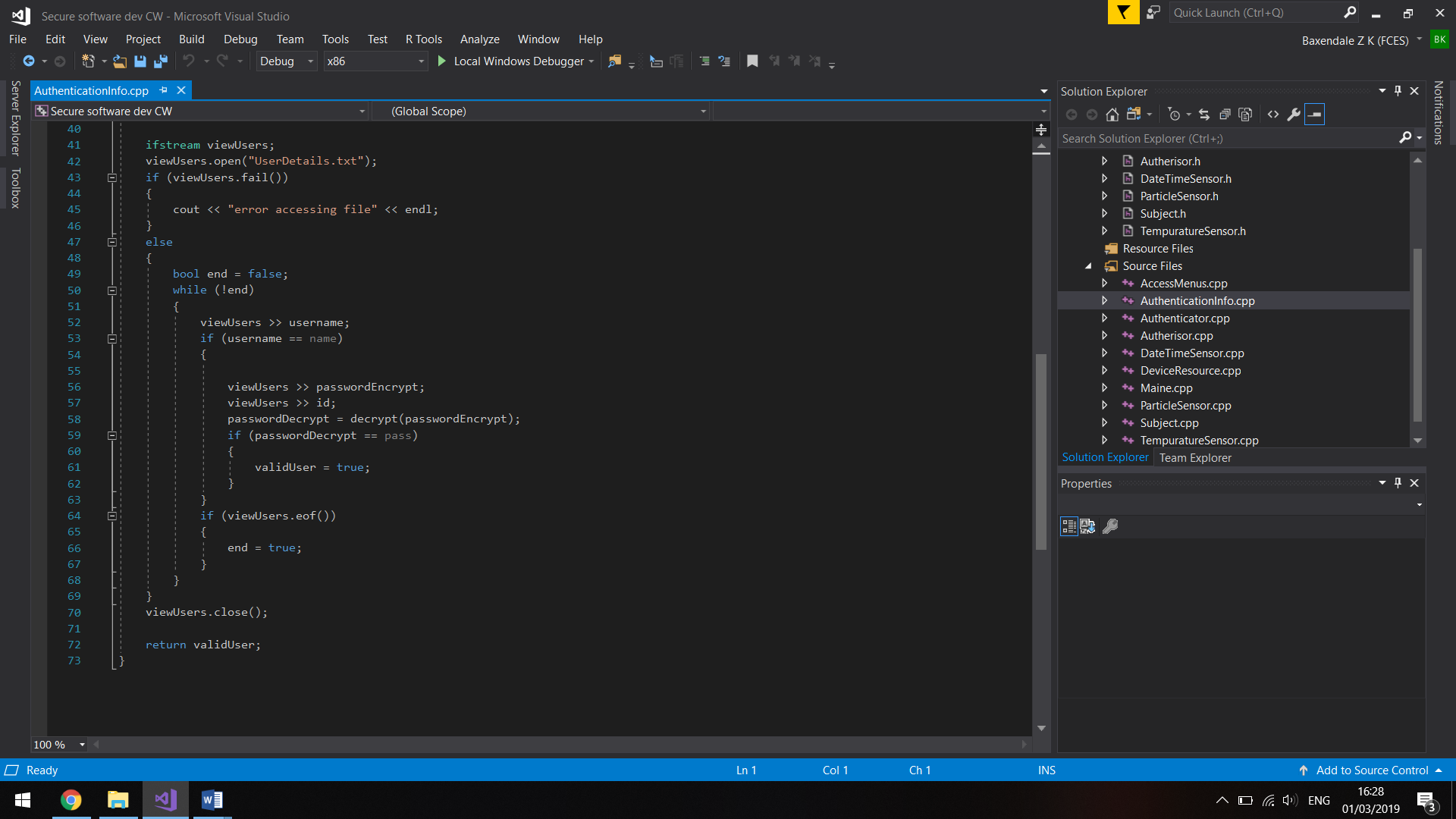
Line 148

Line 165

Check that data read from file is what you want and you are using information that is expected:



1. Reaching chars up to EOF (end of file). EOF is a negative character and if a program uses signed integers it can be difficult to distinguish the difference between the EOF and the files data. I avoided using signed integers within my text files.



Lines where the .eof() security protocol is used:

AccessMenus

Line 486

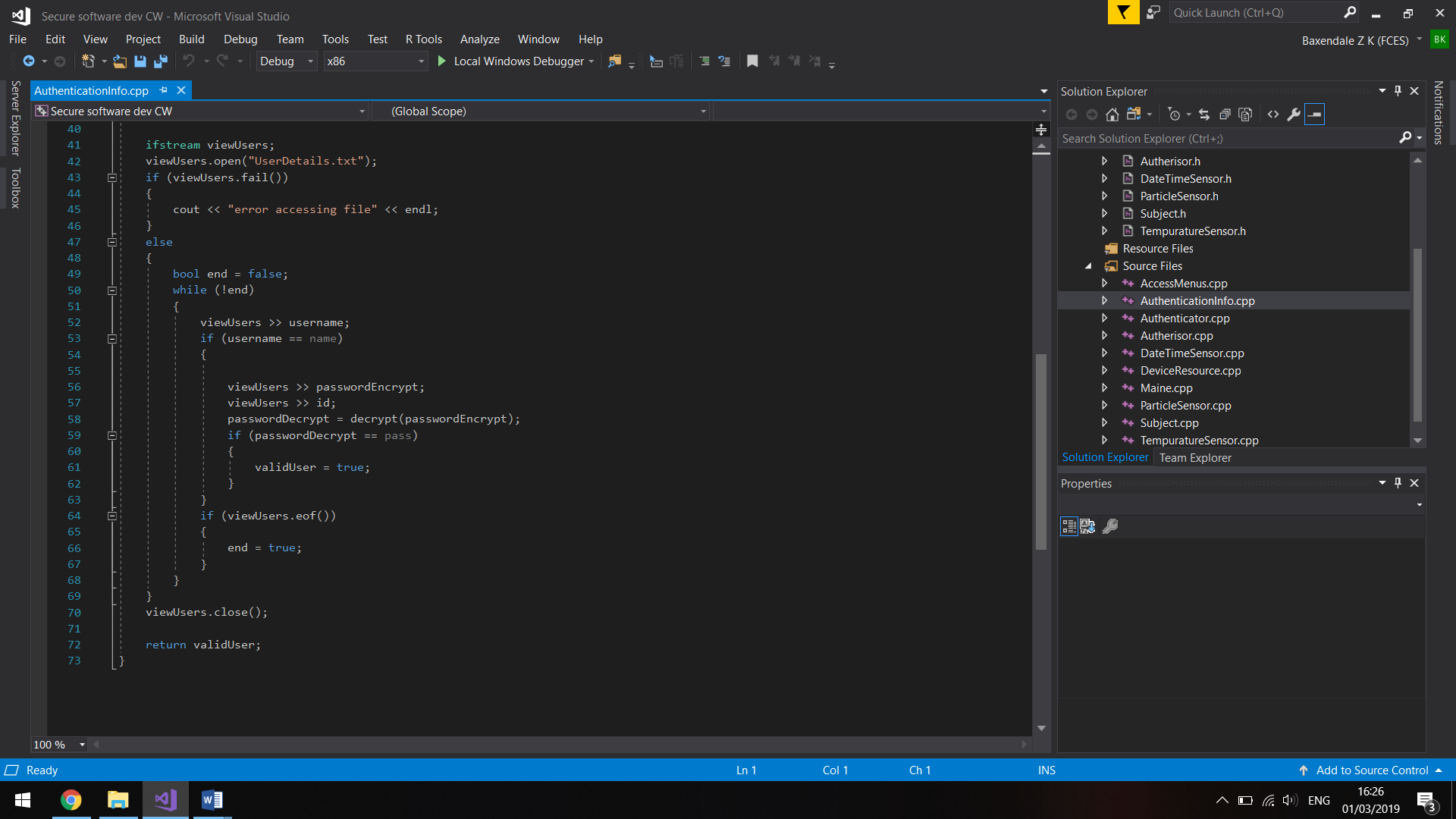
Line 494

Line 529

AuthenticationInfo

Line 64

1. Ensure files are closed when not needed as seen below:



Lines where .close() security protocol is used:

AuthenticationInfo.cpp

Line 70

AccessMenus.cpp

Line 501

Line 506

Line 526

Line 540