# Implementation

## UI

As the interface designs were created, it provided a clear structure of what the application should visually look like, and implementing these designs were also very easy, this was due to the use of graphical class library known as “Windows Forms”, which provided the necessary components known as “controls” needed to adhere to the standards of the designs.

Below shows the implemented “Overview” section:

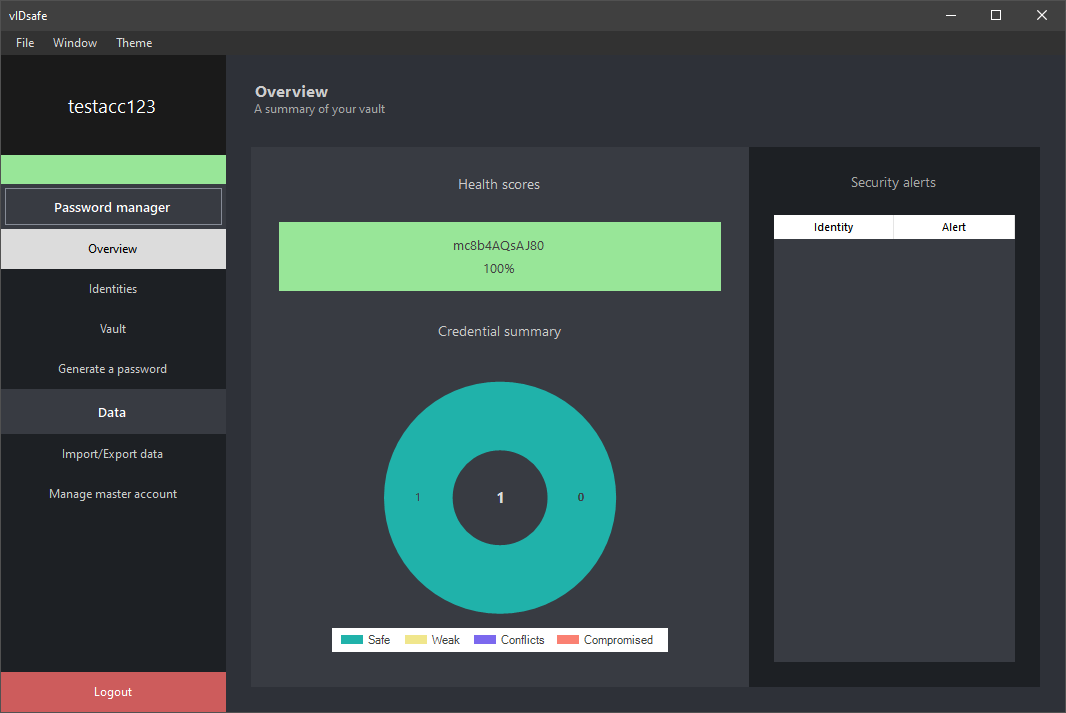


Figure 5‑1 Implemented overview section

Much like the designed screen in Figure 4‑6, above shows a slightly modified version in which the layout is identical although the colour scheme was changed for consistency. When looking further into this screen, it’s also noticeable that just about everything is placed into a panel, which is a control given by WinForms, the reason for this was because it helps organise the layout of the screen through grouping its sub controls as well as aligning and making it more responsive through the use of properties such as “docking” and “anchoring” as well as “margin” and “padding”; the title panel for an example is docked at the very top meaning regardless of the screen size, it will always be anchored to the top. The inner panel showing the “health scores” and the “credential summary” as well as the “security alerts” use a padding of 25 for all directions to provide a 25-pixel gap to separate them from the rest of the components in the screen. Something else that’s also noticeable is the design of the buttons; these all use the “flat” style as their appearance which diverts from the average windows button which may have borders, which wouldn’t compliment the overall design of the screen.

As previously said, just about everything is inside of a panel, and this includes the different sections of the application as well, which are all part of a “navigation” or a “home” form/screen, this is shown below:

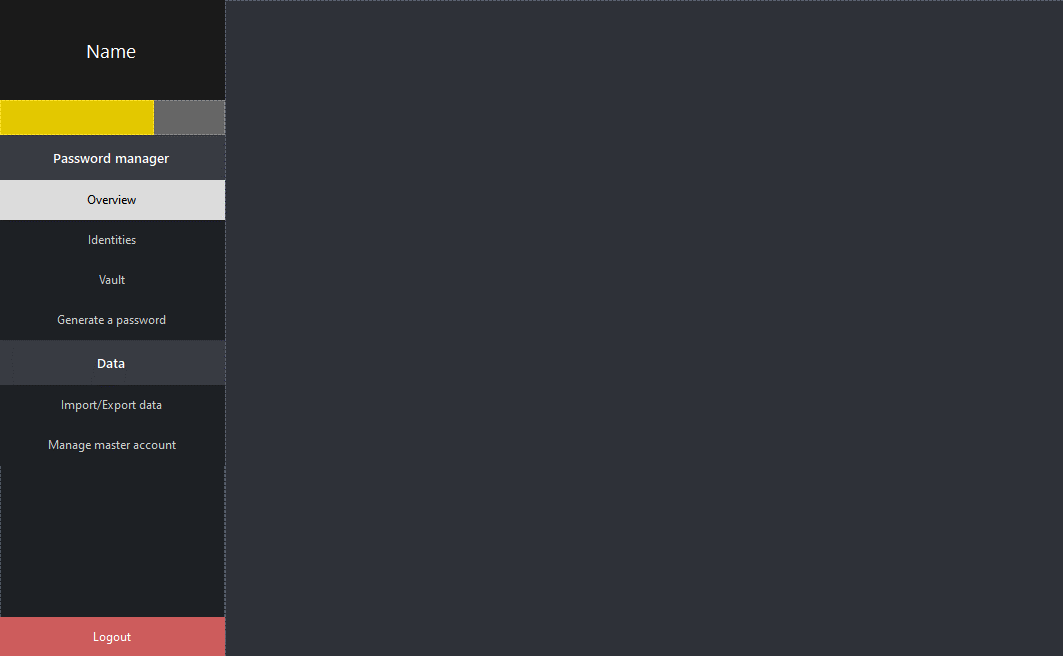


Figure 5‑2 Navigation screen

Each section is split into a form of its own, and when navigated to it by one of the buttons on the navigation panel on the left, it simply opens that form onto the panel on the right inside of this navigation form, this is known as a child form and it was made possible through code in which the child form based on the navigation button is added as a control inside of the panel. The reason why it was done this way was so that it wouldn’t be necessary to then duplicate the navigation panel onto each form which would create redundancy as it’d also mean that the navigation code would be duplicated for each. This also meant that a lot of memory would be saved considering these child forms are closed before opening another, this was also done in code by assigning the current child form as the “active” form.

Regarding the sections/screen of the application, one in particular had to be altered separately, this was the “Application settings” section which was converted into a toolbar:

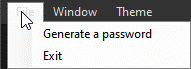


Figure 5‑3 Settings toolbar

The reason for this change was because it felt unnecessary to have it as a part of the navigation panel as it seemed “out of place” due to it having fewer content within, hence why the “ToolStrip” was used as a solution to make the options as simple and easily accessible.

## Themes

Moreover regarding the interfaces, inclusively with the application settings, it was decided earlier on that it should contain themes as well as the option for the user to switch between them, although it wasn’t an easy task considering that the custom theme would need to be applied to each control and have a separate set of colours for various controls. To solve this, an abstract class was created known as “Theme” which is shown below:

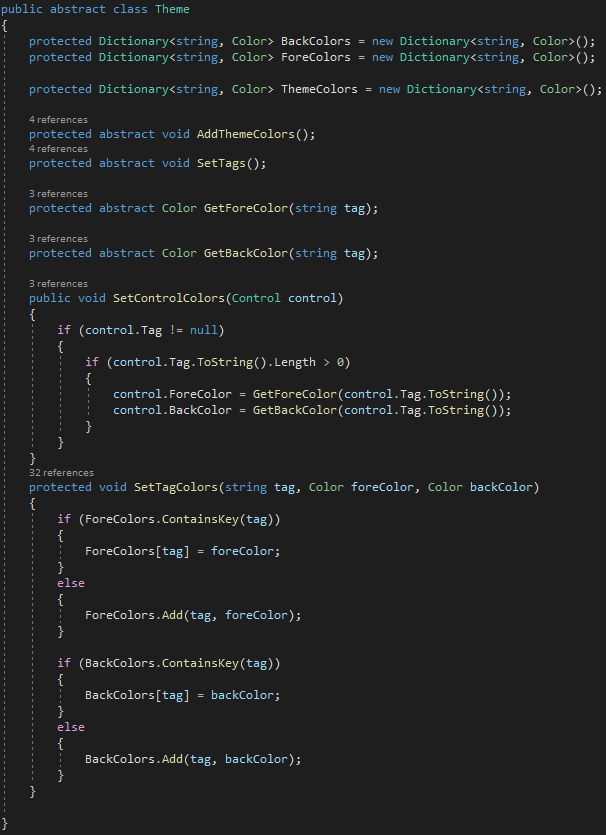


Figure 5‑4 Theme class

This class was used to inherit and implement the shown methods and fields into the custom themes in which most of these members are overridden with specific details accustomed to its theme. A key thing noticeable here is that the controls and their tags are passed in when setting and getting their colours, this measly eliminates most of hard coding for the themes as the specified tags are set to their respective controls through the forms themselves where these tags are shared between each theme class and are assigned a background and foreground colour using the selected theme; due to this, it simplifies the creation of future themes and also the modification of existing ones.

Upon loading the initial “home” screen/form, these themes are automatically appended to a list using the advantage of polymorphism introduced by the use of an abstract class, and so when the user selects a theme, it uses the selected index from the theme select combo box to get the corresponding theme from the list to set as the current theme.

## Credential generation

A simple yet important feature is randomised credentials, a credential can refer to any of the following in this application: a username, a password, a passphrase, or all of them together. The generation of such credentials take place in a class named “CredentialGeneration”, below shows the method for generating a username:



Figure 5‑5 GenerateUsername method

To enforce compartmentalisation, the generation of the username is based on a set of characters which are taken from a string that’s passed in, and so when an identity is created, any credential generated under it has its username based on the identity name using the method above. From the snippet, it shows that there’s three types of characters that gets used in the generation of a username, one of them being numbers from 0 to 9 with the other two as the lowercase and uppercase characters from the derived name (Identity name in this case).

Generating a password works very similarly as well, although the only difference is that the chosen characters for the password generation is user-selected, although with passphrases, these chosen characters are instead words and they are randomly taken from a predefined list of words. The randomisation for the generation of these credentials is determined by an external class known as “CryptoRandom” which returns a securely randomised value, and as for adding a random character/word to build the credential, a StringBuilder is used as opposed to a primitive string for mutability.

## Health scores & Credential statuses

A crucial feature needed to be implemented was health scores, which indicates how strong your credentials are by counting the unsafe credentials against the safe credentials, these health scores are split by the identity as well as the vault, meaning that each identity has its own health score, and the vault contains an overall health score which combines all identity health scores, this is shown in “CalculateHealthScore” and “CalculateOverallHealthScore” methods shown in Figure 8‑2 and Figure 8‑3 respectively.

There are four statuses of credentials that affect an identity’s health score:

* Compromised
* Conflicted
* Weak
* Safe

These statuses are defined using an enumeration known as “CredentialStatus” shown in Figure 8‑4; by default, every credential is set to the “safe” status, however, each credential contains a public method known as “CalculateStatus” within to get its status as shown in Figure 8‑5, this is done by calling the below shown private methods within each credential. To check if a credential is breached, it first uses the “CheckBreached” method as shown:

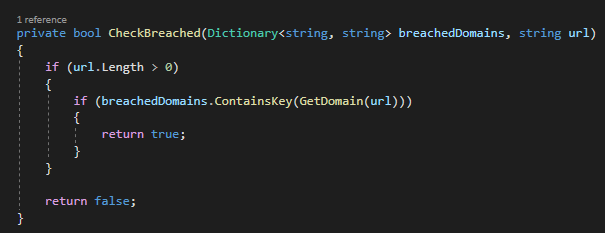


Figure 5‑6 CheckBreached method

This method uses the domain extracted from the credential’s URL and gets it checked against the breached domains to see if it exists, which marks it as “Compromised” if it does. The method for extracting a domain from the URL, “GetDomain”, is shown in Figure 8‑6.

When the user creates an identity and selects an email, it automatically gathers the compromised data for that email using the method shown in Figure 8‑7. This method gives the option to either get existing breach data or re-fetch using a Boolean; the way it checks for these breached domains is through an API known as “Enzoic”, this allows a credential to be sent to the Enzoic servers to check if it contains any compromised data for that credential, in this case, the application simply uses the email, which when called on the API returns a list of details of each breach for that email.

Additionally to the health scores, checking if a credential is conflicted with another worked differently, since if one credential is checked for conflicts, it scans through each identity if the same credential exists and sets them to “Conflicted” as well if they do exist. This is shown below:

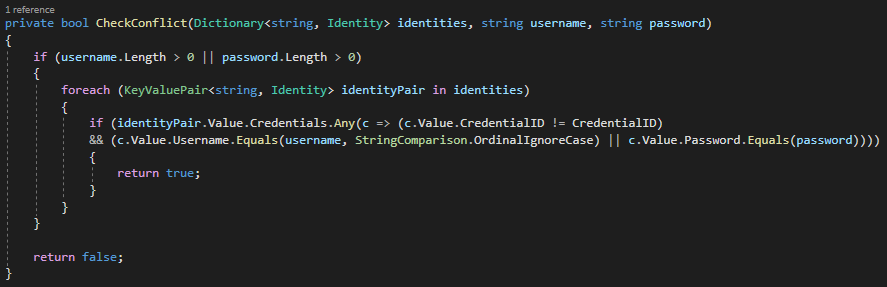


Figure 5‑7 CheckConflict method

This method iterates through each identity in the list of identities that is passed in and iterates through their credentials using LINQ to check if any credential that isn’t the current credential (Using the credential ID) contains the same username (Regardless of case) or the same password (Case sensitive).

Finally, there’s the method to check if a password is weak, which simply takes a password and checks if its strength is below a threshold value to set to as “Weak”, and it is shown below:

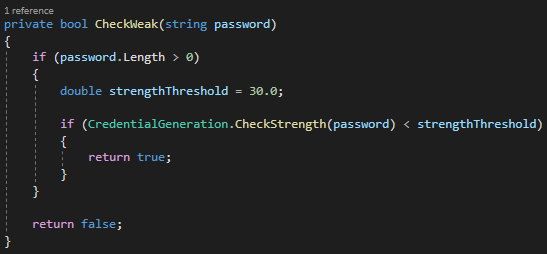


Figure 5‑8 CheckWeak method

The threshold by default is set to 30.0 (30%), as it is deemed to be reasonable for the average password as it considers having a good password length and/or having various types of characters, this is determined from the “CheckStrength” method shown in Figure 8‑8; this takes in a password and iterates through selected regular expressions and accumulates a score which is initialised as the password length, and returns a percentage using the accumulated score and the maximum score.

## Vault encryption/decryption

Initially, it was planned to use bcrypt to hash the user’s master password and use that as the key to encrypt the vault using AES encryption as discussed in section 2.3, this however did not go as planned as later conducted research concluded that regardless of the input when hashing with bcrypt, the output will always be different and unique due to an in-built salt parameter which is randomised in each hash, and thus defeating the whole purpose of using it as an encryption key as these hashes aren’t stored anywhere to be linked and verified. This was when a similar, but a key derivation function was found known as PBKDF2, which allows the customisation of the salt for which it was decided that the user’s username would be used which meant that both the master password as well as the username would be used to create an encryption key which is even better. This also still includes the work factor, which in this case is the hash iterations where the higher it is the longer it takes to hash a password, this is shown below on the implementation:

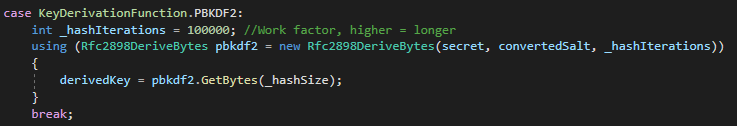


Figure 5‑9 PBKDF2 with 100000 hash iterations

The mode use for AES encryption/decryption is CBC, cipher block chaining, this particular method uses a chaining mechanism that causes the decryption of a block of ciphertext to depend on all preceding ciphertext blocks, which makes it harder to crack as one mistake affects the rest of the ciphertext. It also introduces a variable known as an initialisation vector which is commonly referred to as a “nonce”, it is used as an initiator for the encryption and it helps randomise the output of the encryption so that in this case the vault’s data cannot be pinpointed and will be randomised on every encryption, therefore it should be unique and random but not necessarily a secret.

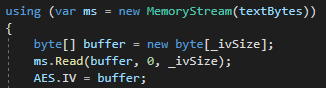


Figure 5‑10 Extracting IV from ciphertext

The issue with this is that when decrypting, this IV is still needed to be used alongside the key, as a workaround for this, the IV is stored before the ciphertext itself in an array of bytes before being converted to a string, that way when the ciphertext is being decrypted, as shown above, the IV is extracted from it by reading the “N” bytes where the “N” is the IV size (Usually the key size divided by 8 where the key size is the selected size of the secret in bits, 256).

For security reasons, the vault is automatically stored locally under a folder known as “Vaults” by default in the same directory as the application. Every time the user logs out or closes the application, it automatically saves their vault to that directory as a file under their name, this file contains the “Vault” object which is initialised upon registering for an account, and this is where the encryption takes place as this vault is serialised as a string which is encrypted with AES (With the PBKDF2 key as mentioned before). Upon logging in, this string is decrypted using the same key and serialised back into the “Vault” object for the user to use, this means that as long as you have your vault file under your login username, it will be decryptable.

An example of such an encrypted vault is shown below:

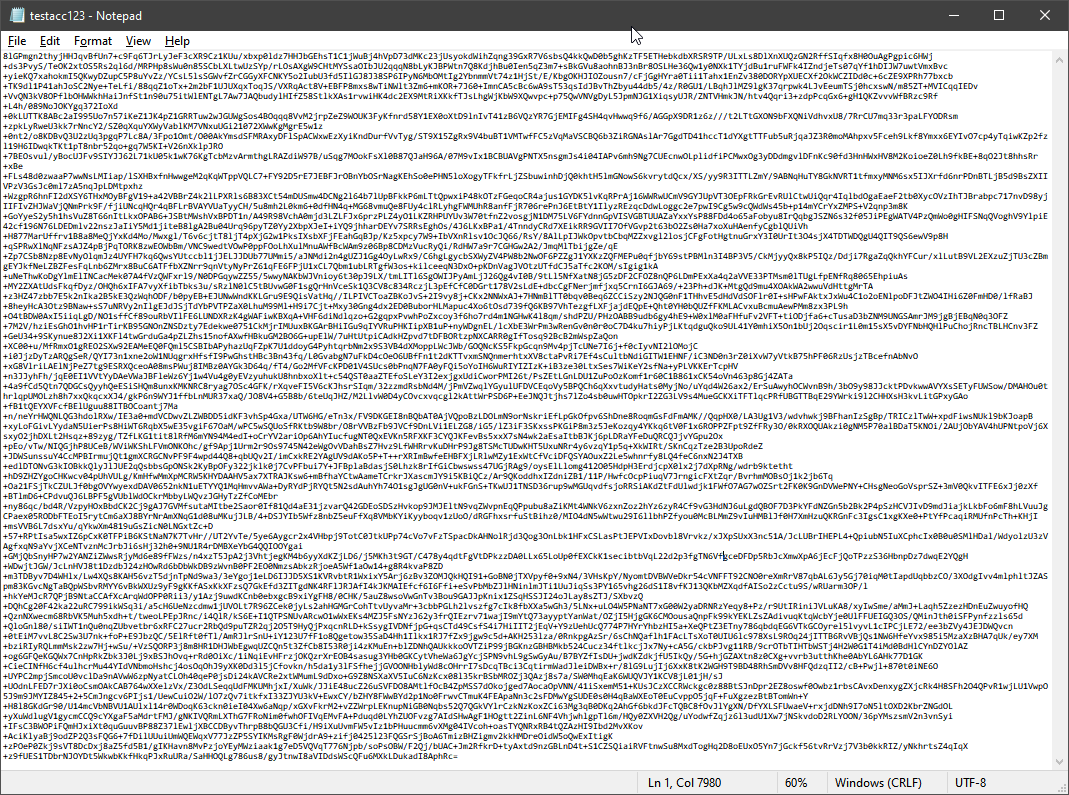


Figure 5‑11 An encrypted vault

It is however worth noting that due to limitations with .NET development, the application may be vulnerable for memory attacks as the credentials are stored in clear text while the vault is decrypted, although it is up to the user to ensure that their system is secure before using this application. Furthermore, upon making changes to the “Vault” class internally, it seems to corrupt any further uses of the “Vault” objects in previous versions, this again is because the object itself gets serialised hence when reading a file that contains the same object but with new or fewer additions it won’t work.

## Portability

Another requirement in this application is portability where data within should be exportable and importable, this is done in three ways; CSV files, JSON files, and encrypted files. The encrypted files simply work the same way as the normal vault encryption where you need a salt and a key to decrypt, except the file name selectable by the user, the reason why it’s done this way is so that it can be used more of as a “backup” and “restore” as the entire vault gets serialised as described in section 5.5 and thus can only be used by this application. With CSV files, each credential is stored in a row alongside its identity details, this is done through the help of the library “CsvHelper”, although it is unencrypted meaning that all the data will be exported in plain text, an example is shown below of such an exported file:



Figure 5‑12 CSV export

An issue with this is that there’s some redundancy being produced where upon exporting multiple credentials for the same identity, the identity details are still repeated, this is because as opposed to the original idea of exporting identities and credentials individually, they are exported together, and furthermore, the attribute names for the columns needs to be manually set inside of the class that is being serialised. This however isn’t the same for JSON files, where instead the library “Newtonsoft.Json” is used, which handles all these problems by aggregating credentials by identities as the key, which was possible by the use of a dictionary which is used to store the identities and the credentials by the email and a unique GUID respectively, the same vault is exported into a JSON file and is shown below:

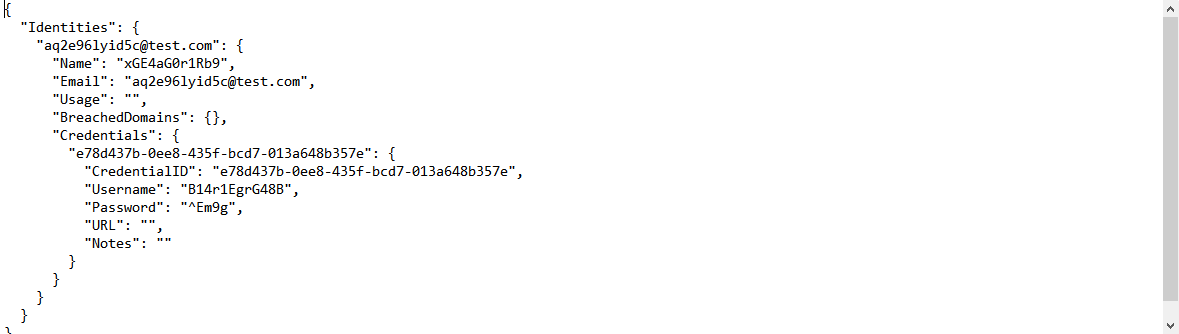


Figure 5‑13 JSON export

As shown from above this also allows the addition of some other attributes such as “BreachedDomains”, and it overall improves readability as the aggregation categorises lists where its public attributes are shown under it.

There is a common drawback to all three ways of exporting, and this is that it may not be easily imported into another password manager, for the CSV file you may simply rename, remove, and move columns although for the JSON file it’s slightly difficult as the format may not comply with the other password managers.

## Code structure & quality

Throughout the development of this system, code was written with reusability and maintenance in mind, hence why object-oriented programming principles were adhered; initially it planned to create just four classes as only just four entities were identified as shown from Figure 4‑3, although there were more created for different purposes. This also introduced several design patterns in order to ensure that there’s less redundant code and more extendable and maintainable code; an example being the “Theme” class as described and shown in section Themes5.2 where this abstract class is inherited to create a theme; therefore it can be used to create customised themes while it’s essentially still a “Theme” object through polymorphism. There were several other instances in which inheritance was used such as the “Health” class as shown below:

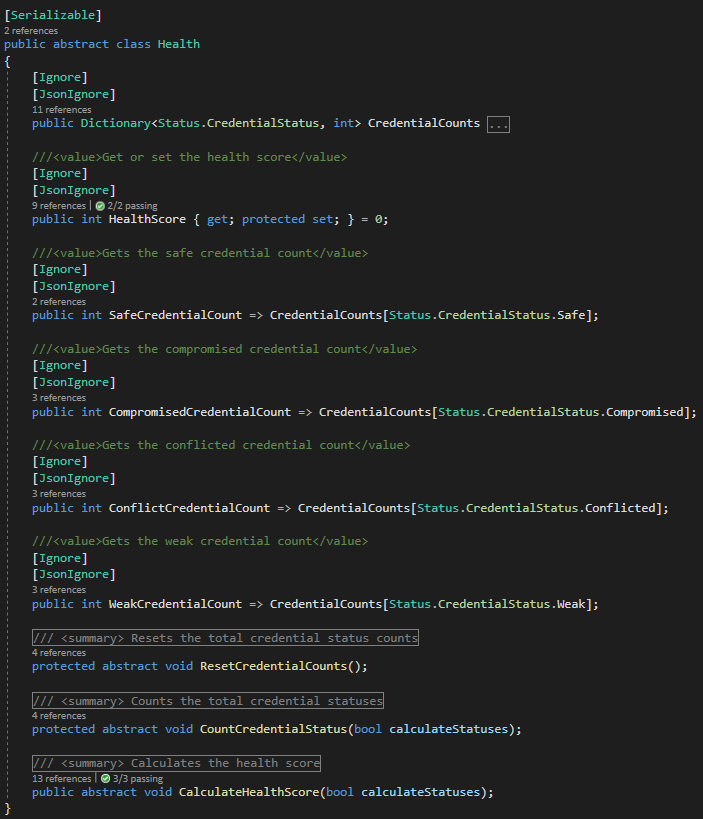


Figure 5‑14 Health class

This is another abstract class which is inherited by the “Vault” and the “Identity” class as they both share one thing in common; health scores, however, while the variables are the same, the methods should be implemented differently, hence why they were abstract methods. Each of these classes contain members (variables and methods) where the variables that are used by other classes are created as properties, although to ensure that only the needed properties are modified and accessed, encapsulation was used by making setting the modifiers of these properties to public, protected, or private as shown by the “HealthScore” variable in the class above.

Regarding the vault object, while it is in memory, it is stored in such a way that it is only accessible through the master account object; this object is initialised from the beginning of the application as a singleton with a private constructor, this is done to ensure thread safety when accessing it through multiple threads. Furthermore, the vault object inside the master account object contains the identities and these identities contain the credentials which displays a one-to-many relationship between them, and so to store these, a concurrent dictionary in both the vault and in each identity is created to both ensure thread safety as well as the ease of access where each identity/credential is identified by a key, an email and a credential ID respectively, thus disallowing duplicates.

It is also worth noting that an identity has a reference to the vault it is created from and a credential has references to both the vault and the identity it’s created from; this implements the dependency injection principle as it helps access the attributes in another object without having to re-instantiate it. This is done through the constructor, below shows the constructor for a credential object:

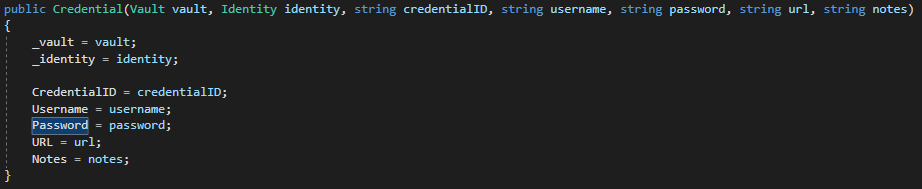


Figure 5‑15 Credential constructor

Upon creating a credential as shown above, the vault and identity it’s created from is automatically assigned as it’s passed in through the parameters, and in this case the vault is used to obtain the identities to check for similar credentials whereas the identity is used for checking breached domains.

When talking about the status of these credentials, it was noted that an enumeration known as “CredentialStatus” was assigned to each credential to determine its status as described in section 5.4, this implements the state pattern as this enumeration it makes it easier to calculate the health score by counting using the “state” of each credential, which is either safe, weak, conflicted or compromised. Going back to the aforementioned modifiers, when calculating the status, private methods were used for individual calculation although they are encapsulated within a single public method known as “CalculateStatus” shown in Figure 8‑5, this is for abstraction where the user can call this method to calculate a credential’s status as opposed to going through each calculation method. This also introduces the single-responsibility principle, which is used throughout all classes to separate functionalities which helps maintainability as it’s easier to pinpoint and fix errors.

In addition to the code quality, the codebase was documented well as the project is open source, this was done through a few ways, one of which being through comments; in each class all methods were commented using XML, an example of this is shown below:

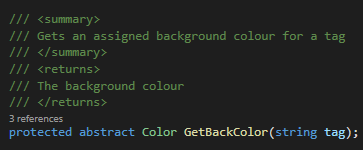


Figure 5‑16 GetBackColor method's documentation

In all methods, a “summary” tag is used to describe the functionality of the method in a simple sentence followed by a “return” tag if the method returns a value, it’s done this way so that when hovering over the method when it’s being used, it displays a tooltip of this XML documentation as shown below:

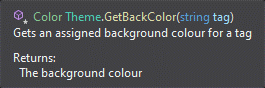


Figure 5‑17 GetBackColor method's tooltip

This makes it simpler as opposed to commenting line by line which not only is time consuming for the developer but for any future developers reading the code. This is also applied to properties of each class but instead done by using a “value” tag. These classes were also all separated by folders pertaining to their purpose in order to exclude the “form” classes from them as they are dependent on the other classes.

Using the same example, another way of documentation was through variable naming; for each method and public field in a class, pascal case is used for the naming as it’s enhances readability by enabling the distinction of the words used for the name to display its purpose. Private fields on the other hand starts with an underscore followed by its name in camel case to denote that it’s unusable outside of the class it’s in.

It’s also notable that most of these methods in these classes were designed for customisability where for an example the previously mentioned feature for portability was separated by a different class in which an enumeration is used to determine the format to use for importing and exporting the vault, and so to add a new format, a developer could simply define a new format in the enumeration and create a method for it. This is also seen in the password hashing method as described in section 5.5 where each hashing function is predefined in order to ensure that only the specified functions can be used, and this is independent of any other class hence also improving reusability.

## Changes to the requirements

While the project was carried out almost exactly as planned, there were some changes amended to the requirements as some were considered to be not suitable or feasible. For instance, throughout the development it was revealed that bcrypt could not be used to as an encryption key due to its inbuilt-salt parameter and thus PBKDF2 was used which still produced the same result, this is described in section 5.5.

It was also initially planned to have a password hint for the user when they first login, although this introduced a security flaw as this hint would have to be stored somewhere, however, since the vaults are encrypted and stored under the user’s account name, storing it alongside it would defeat the point of a password hint as it’d make it far less difficult to guess the actual password as the account name is already present. Even if the password hint was stored somewhere else hidden, it would still be abusable as an attacker can enter an incorrect password for an account only to be given a password hint, which the user might choose something similar to the actual password.

Lastly, an important requirement removed was the ability to scrape public information for given details; this would also later be considered futile as it may return irrelevant results. Furthermore, this may only do more harm than good for the user considering that searching these terms may allow some of these search engines to extract identifiable information using the entered search terms as well as their IP address, which would ultimately defeat the purpose of this application, this is mentioned in section 2.8.

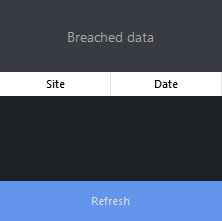


Figure 5‑18 Breached data table

As shown above, due to these reasons, this was instead replaced with the list of breached domains gathered upon entering an email address, that way the user is aware of the sites that they have possible compromised accounts in.

Appendices

More relevant material

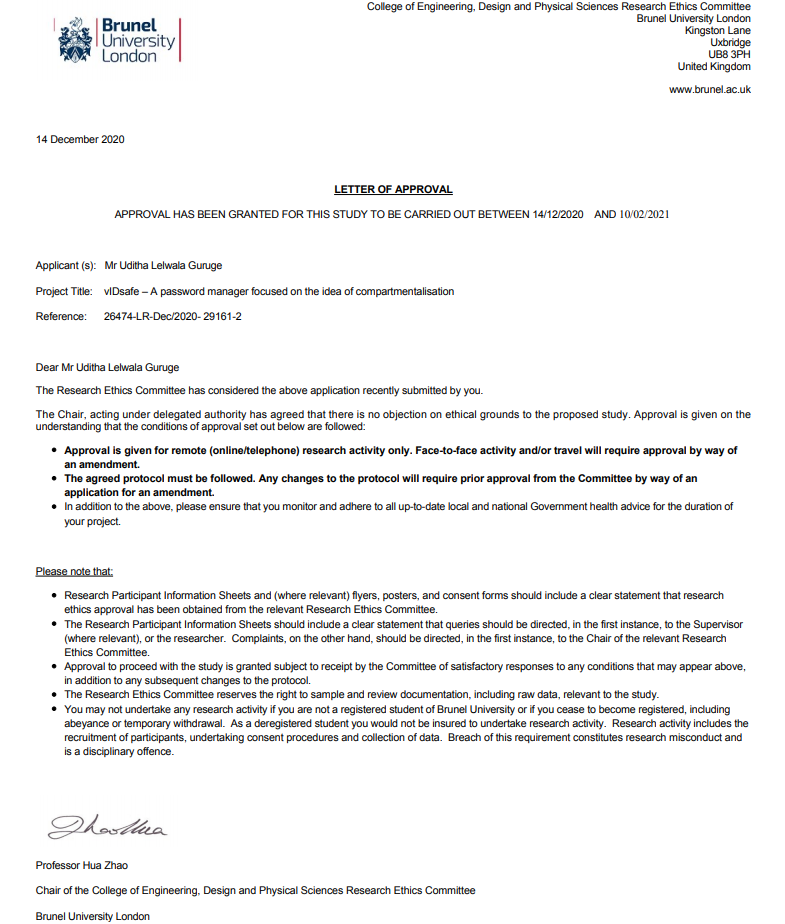


Figure 8‑1 Letter of ethics approval

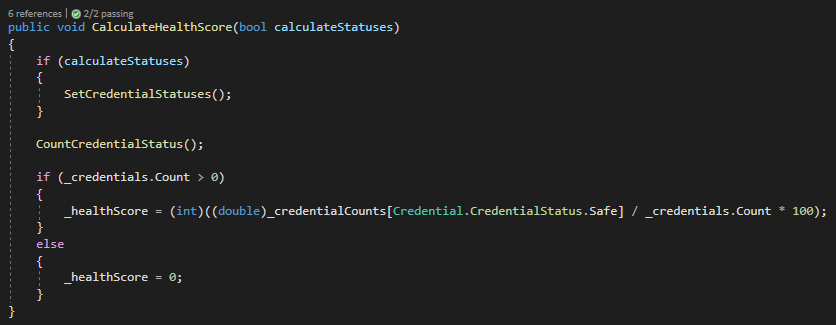


Figure 8‑2 CalculateHealthScore method

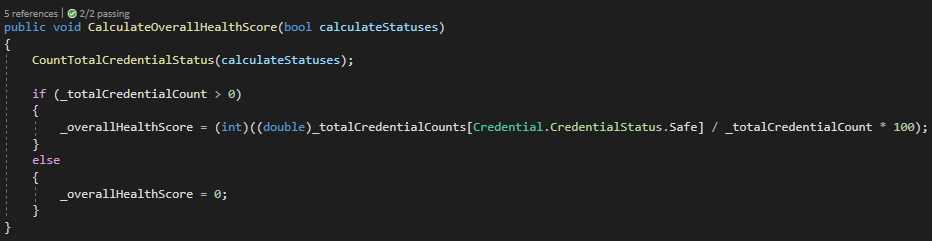


Figure 8‑3 CalculateOverallHealthScore

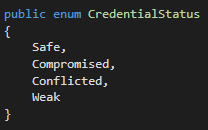


Figure 8‑4 CredentialStatus enum

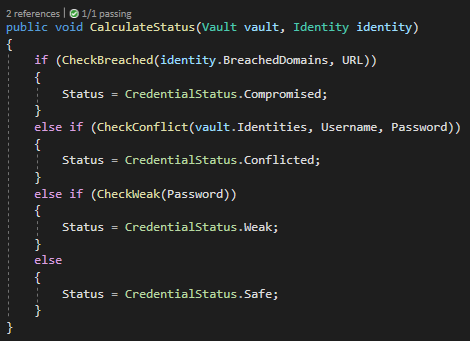


Figure 8‑5 CalculateStatus method

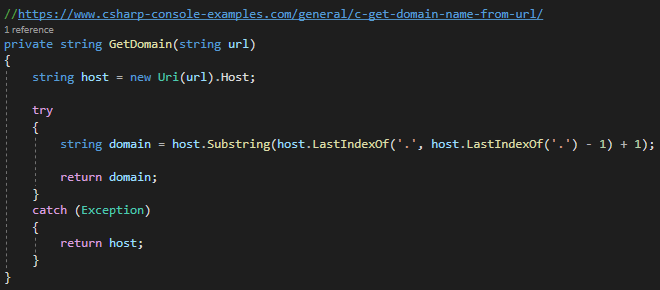


Figure 8‑6 GetDomain method



Figure 8‑7 GetBreaches method



Figure 8‑8 CheckStrength method

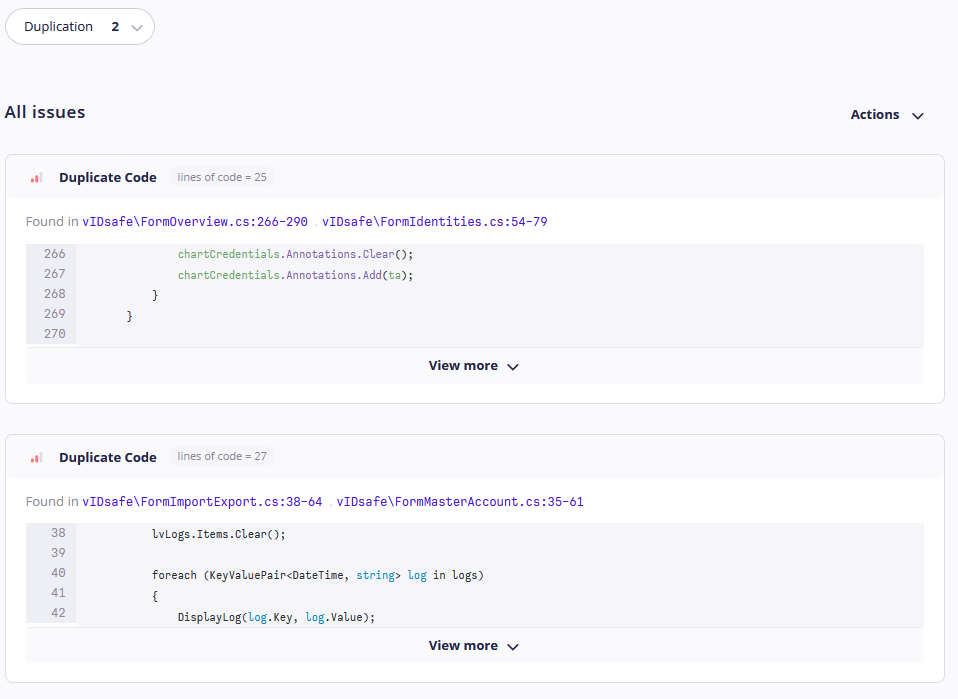


Figure 8‑9 CodeFactor issues