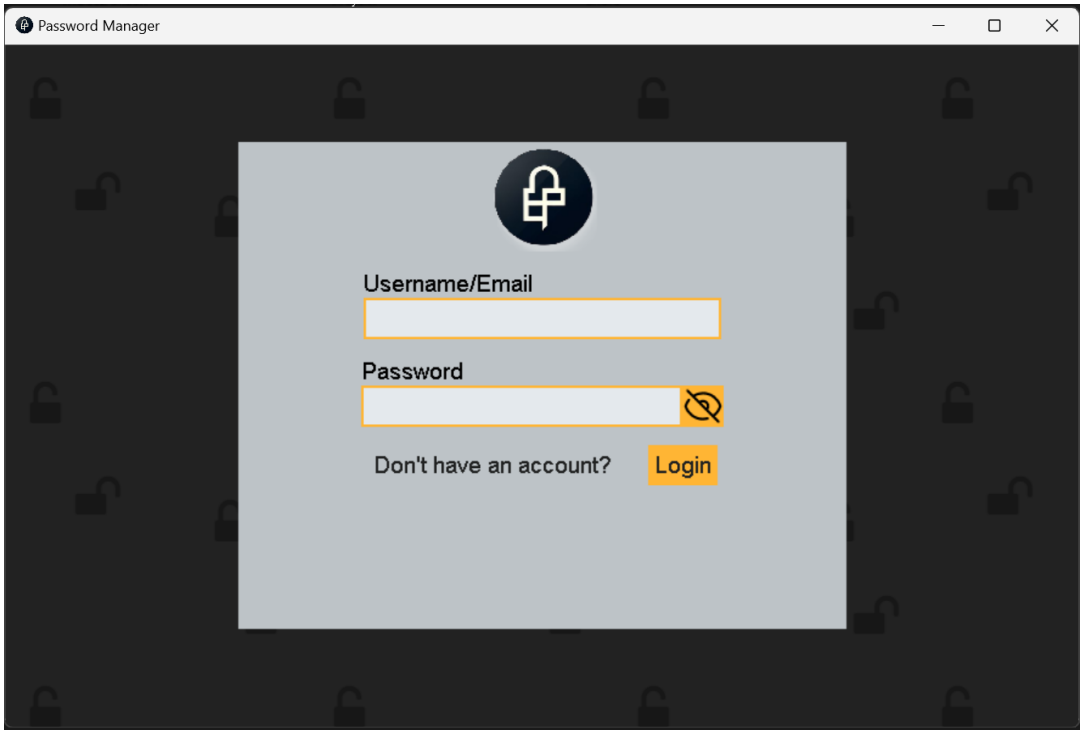
Projekt Informatik – Password Manager



Erstellt von:

1. Ahmad Abeer Ahsan(Matrkl no. 312896)
2. Moeez Muhammad Ahsan(Matrkl no. 312897)

Aufgabenstellung:

Ein Benutzer kann einen Passwort-Manager verwenden, um alle seine Passwörter an einem Ort zu speichern. Zunächst muss er ein Konto mit einer E-Mail/Username und einem Master-Passwort erstellen. Dieses Master-Passwort muss ein sicheres Passwort sein. Anschließend kann er sich mit dem erstellten Konto bei der Passwort-Manager anmelden. Nach der Anmeldung kann der Benutzer seine Anmeldedaten hinzufügen, entfernen oder bearbeiten, einschließlich der Plattform, auf der das Konto erstellt wurde, der Anmelde-ID und des Passworts. Der Benutzer kann auch ein starkes Zufallspasswort generieren. Die Zugangsdaten werden dann in verschlüsselter Form sicher online gespeichert. Der Benutzer kann seine Anmeldedaten von jedem beliebigen Computer aus abrufen, er muss sich nur an sein Login-Id und das Master-Passwort seines Passwort-Manager-Kontos erinnern.

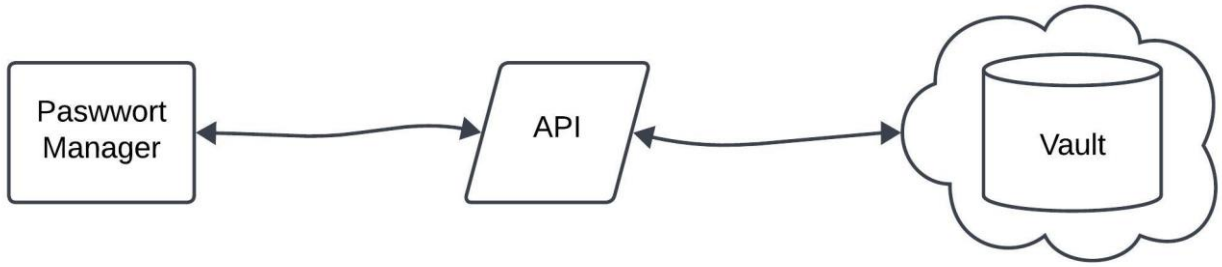
Verwendete Technologien:

C++, cpp-httplib(HTTP Bibliothek), wxWidgets(C++ GUI Bibliothek), PostgreSQL, libpqxx(postgreSQL C++ Bibliothek):

Projektordner herunterladen unter: ProjectInformatik.zip

Inhalt:

Konzept



Ist es sicher, alle seine Passwörter an einem Ort zu speichern? Es besteht ein gewisses Risiko, das man eingehen muss, wenn man seine Passwörter in einer verschlüsselten Datenbank im Internet ablegt. Die meisten Sicherheitsforscher verwenden Passwort-Manager(PM) und würden die Verwendung von Passwort-Managern(PM) empfehlen.

Warum PMs funktionieren: Sehr sichere Passwörter sind schwer zu merken. Es ist besser, mehrere sichere Passwörter für jedes einzelne Konto zu haben, aber sich viele starke Passwörter zu merken ist schwierig/unmöglich. Die einzige Möglichkeit, die einem bleibt, ist, ein oder zwei starke Passwörter für alle seine Konten zu verwenden oder sein Passwort einfach irgendwo auf dem Computer oder auf einer Seite aufzuschreiben. PMs bieten eine sehr sichere Möglichkeit, alle Passwörter in einem so genannten "Vault" zu speichern. Ein Vault ist einfach eine große Liste von Passwörtern in verschlüsselter Form.

Die Gefahr besteht darin, dass ein Angreifer sich Zugang zum Vault verschafft und alle Passwörter entschlüsselt. Ein PM macht nur Sinn, wenn es für alle oder viele Konten verwendet wird, ansonsten ist es weniger sinnvoll. Ein PM begegnet diesem Problem dadurch, dass alle Passwörter mit einem Verschlüsselungsschlüssel verschlüsselt werden. Die Frage ist nun: Wo ist dieser Schlüssel gespeichert und wer hat Zugang dazu?

Dieses PM, das wir entwickelt haben, ist ein cloudbasiertes PM der zweiten Generation, das eine symmetrische Verschlüsselung durchführt. Bei der symmetrischen Verschlüsselung wird derselbe Schlüssel zum Ent- und Verschlüsseln der Chiffre verwendet. Es speichert alle verschlüsselten Passwörter in einer Cloud-Datenbank. Die Cloud führt die Ver- und Entschlüsselung nicht selbst durch. Der gesamte Vault wird vom Benutzer auf der Client-Seite ver- und entschlüsselt und dann an die Cloud gesendet oder von dort abgerufen. Der Verschlüsselungs-/Entschlüsselungsschlüssel

wird niemals in der Cloud gespeichert. Das ist gut, denn wenn die Datenbank durchsickert oder sogar von einem unbekannten Administrator weitergegeben wird, sind die Passwörter immer noch sicher, da der Schlüssel nie in der Datenbank gespeichert wurde.

Es gibt nun zwei Probleme, die wir lösen müssen. Das erste ist, wie leiten wir einen Schlüssel ab, den der Server nicht kennt, den wir aber verwenden können? Die andere Frage ist, wie wir einen Server davon überzeugen, uns den Vault zu schicken, denn der gesamte Vault befindet sich in der Cloud. Ein Benutzer gibt seinen Login-Benutzernamen/E-Mail und sein Masterpasswort ein und bittet den Server, ihm den Vault zu schicken, damit er ihn entschlüsseln kann. Dieses Szenario ist gefährlich, weil der Benutzer gerade sein Masterpasswort an den Server geschickt hat.

Die Antwort auf diese beiden Fragen gibt der Passwort-Manager, indem er aus dem Master-Passwort des Benutzers Schlüssel ableitet. Dieses Master-Passwort muss stark und sicher sein. Die meisten Passwörter sind nicht ausreichend, um als Master-Passwort verwendet zu werden. Wir werden zwei Ableitungen von diesem Master-Passwort vornehmen. Erstens werden wir eine Funktion auf das Master-Passwort anwenden, um unseren Vaultschlüssel zu erzeugen. Und zweitens werden wir unser Master-Passwort für einen Authentifizierungsmechanismus mit dem Server verwenden. Wir verwenden das Master-Passwort, um uns beim Server zu authentifizieren, und zwar so, dass das Master-Passwort dem Server nicht bekannt wird. Der Server wird "Ja" sagen. Der Server sendet uns dann den verschlüsselten Vault. Wir verwenden den Vaultschlüssel, den wir zuvor abgeleitet haben, um das Kennwort lokal zu entschlüsseln. Wir können beliebige Passwörter hinzufügen oder entfernen. Wir verschlüsseln den Vault erneut mit dem Vaultschlüssel und senden ihn dann zurück an den Server, wo er gespeichert wird.

Das letzte Problem, mit dem wir jetzt konfrontiert sind, ist, dass wir einen Schlüssel verwendet haben, der auf unserem Master-Passwort basiert, um uns anzumelden, und wir haben denselben Schlüssel für die Verschlüsselung verwendet. Wir werden zunächst den Benutzernamen (email) an das Master-Passwort (mpass) anhängen, um ein neues Master-Passwort (email|mpass) zu erstellen, und dann dieses neue Master-Passwort mit einer starken Hash-Funktion (H()) verschlüsseln. Die Hash-Funktion(H()) hat viele Iterationen, um zu verhindern, dass sie mit roher Gewalt(brute force) geknackt werden kann. Diese Hash-Funktion erstellt unseren Vaultschlüssel (V = H(email | mpass)) und dieser Vaultschlüssel (V) wird zur Entschlüsselung des Vault verwendet. Im Moment haben wir den Vault nicht, weil der Vault in der Cloud gespeichert ist. Jetzt nehmen wir unseren Vaultschlüssel (V), hängen ihn wieder an das Master-Passwort an und verwenden dieselbe Hash-Funktion, um einen Authentifizierungsschlüssel (A = H(H(email | mpass) | mpass)) zu erstellen. Auch diese Hash-Funktion wird mehrfach durchlaufen. Dieser Authentifizierungsschlüssel (A) wird für die Authentifizierung beim Server verwendet. Auf diese Weise kann ein Angreifer den Schlüssel nicht einfach abbilden, denn er müsste den Hash erraten, was langsam ist, oder den Hash rückgängig machen, was nicht möglich ist. Auf diese Weise erhält der Server niemals das Master-Passwort. Was passiert, ist, dass wir ein Master-Passwort verwenden, um einen Vaultschlüssel (V) abzuleiten, und dann verwenden Sie diesen Vaultschlüssel (V) und Ihr Master-Passwort(mpass) erneut, um einen Authentifizierungsschlüssel (A) abzuleiten, der auf dem Server verwendet wird. Nur die Person, die das Master-Kennwort hat, kann den Vaultschlüssel (V) und den Authentifizierungsschlüssel (A) erstellen, und nur diese Person kann den Vault vom Server anfordern und ihn entschlüsseln. Das Master-Kennwort darf niemals gespeichert werden, auch nicht in verschlüsselter Form. Der Vaultschlüssel ist nicht persistent und nur im Laufzeitspeicher vorhanden.

Beim Speichern von Passwörtern in den Vault erzeugt der Passwortmanager ein echtes Zufallssalz(Salt), aus dem der Vektor (IV) und der Chiffrierschlüssel abgeleitet werden, der das Passwort in eine Chiffre verschlüsselt. Die Chiffre und das Salz(Salt) werden beide gespeichert.

Zusammenfassend kann man sagen, dass die Passwort-Manager der zweiten Generation 3 wesentliche Punkte aufweisen.

1. Ableitung des Vaultschlüssels aus dem Master-Passwort mit einem PBKDF-Algorithmus.
2. Schutz der Daten mit einem starken symmetrischen Verschlüsselungsalgorithmus.
3. Niemals das Master-Passwort in irgendeiner Form speichern.

In unserer verteilten Anwendung wird eine API die Kommunikation zwischen dem Passwort-Manager und der Cloud, in der der Vault gespeichert ist, sicherstellen.

GUI/Frontend:

Wir haben eine Open-Source Cross-Plattform C++ GUI Bibliothek verwendet, um die Benutzeroberfläche unserer Anwendung zu steuern. Wir haben hier nicht alles darüber dokumentiert, wie die Benutzeroberfläche funktioniert, weil es zu lang wäre. Die wichtigen Dateien für die Benutzeroberfläche sind: PasswordManagerApp.h, MainFrame.h, RedisterationFrame.h, AccountPanel.h, MainAddEditAccount.h, MainViewAccount.h, SignInPanel.h, SignUpPanel.h .

Die Klasse PasswordManagerApp ist für die Verwaltung der Master-Account Registration(RedisterationFrame) und der Hauptfenster(MainFrame) zuständig.

RedisterationFrame: Der RedisterationFrame enthält Methoden zur Validierung von Benutzereingben, zur Erstellung von Master-Accounts und zur Authentifizierung bereits erstellter Master-Accounts.

MainFrame: Der MainFrame ist der Ort, an dem die eigentliche Ver- und Entschlüsselung der Passwörter gespeicherter Konten stattfindet. Er enthält mehrere Steuerelemente, die durch die Übergabe eines Account-Objekt miteinander verbunden sind. Die GUI-Elemente "AccountPanels" und "MainAddEditAccount/MainViewAccount" enthalten Account-Objekte und übergeben diese, wenn mit ihnen interagiert wird.

Backend:

Die wichtigen Dateien sind: MasterAccount.h, Account.h, Password.h, HttpClient.h, Crypto.h .

Klassen:

MasterAccount -Klasse: Speichert die folgenden Daten: *Master-Benutzername*, *Master-Kennwort, Authentifizierungsschlüssel*(es wird vom Master-Kennwort abgeleitet und zur Authentifizierung an den Server gesendet), *Vault-Key*(es wird ebenfalls vom Master-Kennwort abgeleitet, aber es wird zur weiteren Ableitung von Ver- und Entschlüsselungsschlüsseln verwendet).

Zur Erstellung des Vault-Key werden der Benutzername und das Master-Kennwort angehängt und 20000 Mal gehasht. Zur Erstellung des Auth-Key werden der Vault-Key und das Master-Kennwort angehängt und 10000 Mal gehasht. Sie werden unterschiedlich oft gehasht, da wir den Schlüssel(hier Master-Key), der zur Ableitung der Ver- und Entschlüsselungsschlüssel verwendet wird, geheim halten wollen. Der Auth-Key wird über das Internet gesendet und ist daher nicht mehr geheim.

Account -Klasse: Speichert die folgenden Daten: *Id*(den id-Wert, der dem id-Wert in der Datenbank entspricht), *masterAccountIdentity*, (Master-Benutzername(Link zu MasterAccount)), *platform*(den Dienst, bei dem das zu speichernde Konto erstellt wird) *identifier*(den Benutzername/E-Mail-Adresse des zu speichernden Kontos), *createdAt(*das Datum, an dem das Konto erstellt wurde), *updatedAt*(das Datum, an dem das Konto zuletzt aktualisiert wurde), *passwort*(Passwort-Objekt, um auf das Passwort im Klartext zuzugreifen). Die Account enthält die folgenden Methoden, um ihre Informationen in der Datenbank zu bearbeiten:
DeleteFromDB(HttpClient* httpCleint); //löscht das gespeicherte Konto aus der Datenbank
AddToDB(HttpClient* httpCleint); //fügt ein gespeichertes Konto in die Datenbank ein
UpdateInDB(HttpClient* httpCleint); //aktualisiert die Felder des Kontos in der Datenbank

Die AccountPanel Klasse hat eine statische Methode, die beim Start der Anwendung alle gespeicherten Accounts des angemeldeten Benutzers abfragt: AccountPanel::FillFromDB(...): Hier werden alle gespeicherten Accounts abgerufen und in ein AccountPanel-Gui-Steuerelement eingepackt.

Diese Methoden kommunizieren nicht direkt mit der Datenbank, sondern senden Anfragen an eine API, die die Anfragen verarbeitet und dann eine entsprechende Antwort ausgibt. Aus diesem Grund benötigen diese Methoden ein HttpCleint-Objekt als Eingabeparameter.

HttpClient: Das Objekt, das Anfragen an die API sendet und dann die Antwort zurückgibt.

Password -Klasse: Speichert die folgenden Daten: *plain_text*(das Passwort als Klartext), *cipher_text*(Chiffretext(wie es in der Datenbank gespeichert ist*)), salt*(Eingabetext zur Ableitung des Schlüssels und des Vektors), *Secret-Key*(aus MasterAccount), *Key, IV*(den geheimen Verschlüsselungs- und Entschlüsselungsschlüssel und den Vektor(IV), der zur Umwandlung von Klartext in Chiffretext oder Chiffretext in Klartext verwendet wird).

Password hat zwei Konstruktoren. Der erste Konstruktor nimmt den Master-Key und das Kontopasswort als Eingabeparameter. Er generiert einen zufälligen Salt, der dann zusammen mit dem Master-Key verwendet wird, um den Chiffrierschlüssel und den Vektor abzuleiten. Mit dem Chiffrierschlüssel und dem Vektor wird das Kontopasswort verschlüsselt, um den Chiffretext zu erhalten.

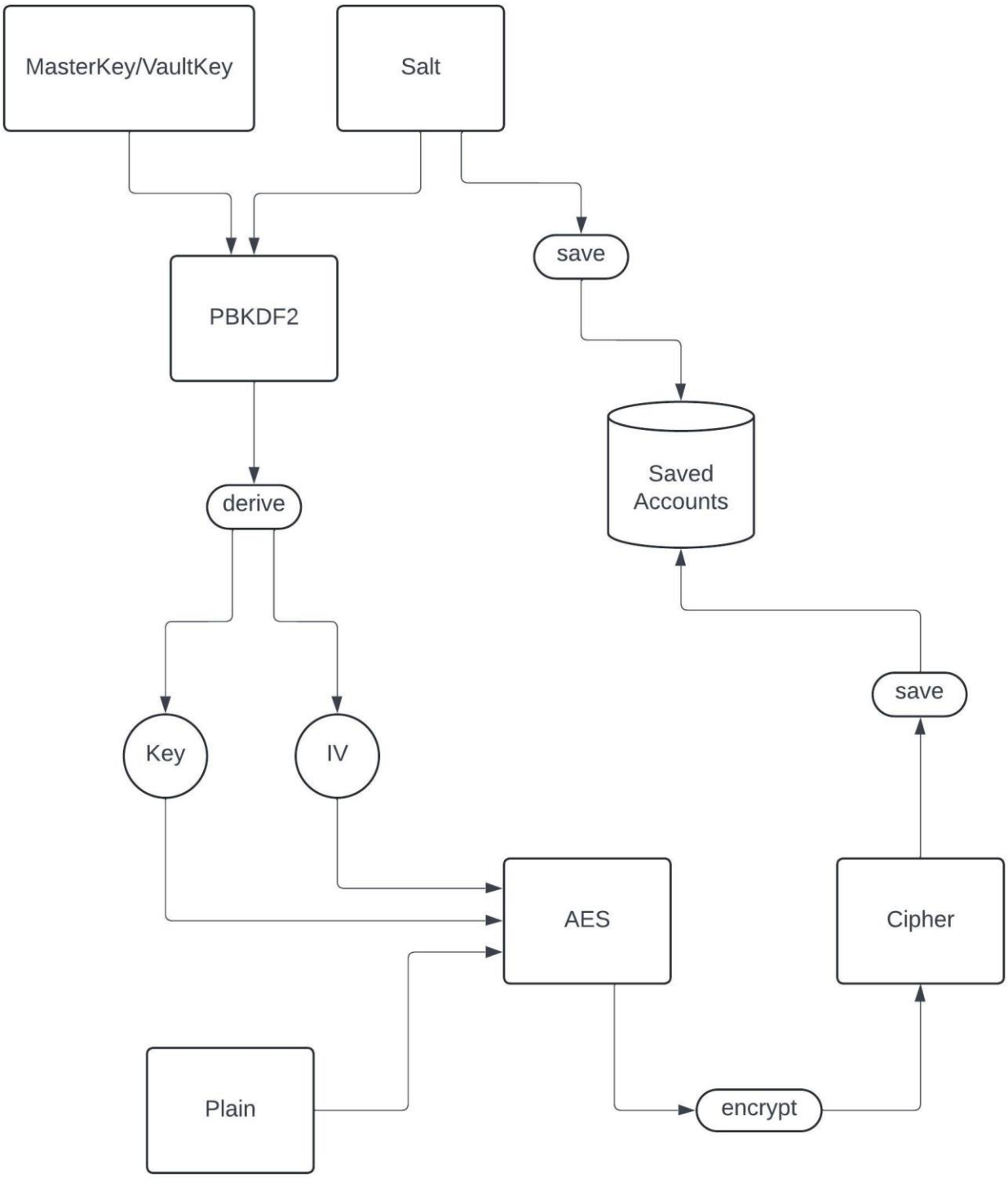
Der zweite Password-Konstruktor nimmt den Master-Key, den Chiffriertext und das Salt als Eingabeparameter. Der Entschlüsselungsschlüssel und der Vektor werden aus dem Master-Key und dem Salt abgeleitet. Anschließend wird der verschlüsselte Text mit Hilfe des Entschlüsselungsschlüssels und des Vektors in den Klartext entschlüsselt. In ähnlicher Weise hat auch die Klasse Account zwei Konstruktoren. Der erste wird aufgerufen, wenn ein Account auf der Cleint-Seite erstellt wird. Dieser ruft den ersten Passwort-Konstruktor auf. Der zweite Account-Konstruktor ruft den zweiten Password-Konstruktor auf, wenn Daten aus der Datenbank gelesen werden.

Crypto.h enthält die kryptografischen Funktionen, die von der Password-Klasse zum Ableiten von Schlüsseln sowie zum Ver- und Entschlüsseln von Kennwörtern verwendet werden.

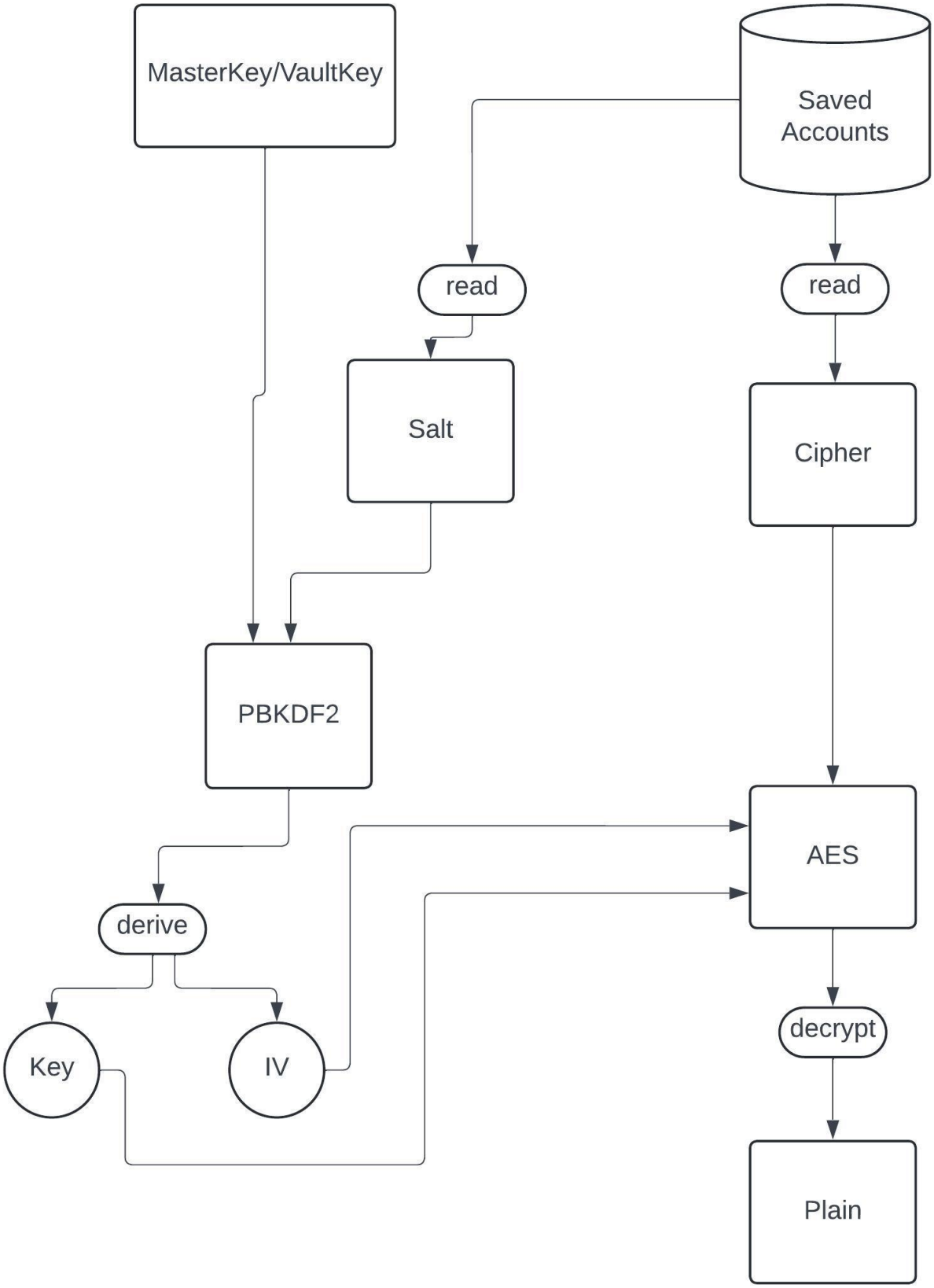
Die verwendeten kryptographischen Algorithmen sind:

- SHA256: Produziert ein Hashwert(Datensatz) aus einer festen Länge(32 Bit),die zur Verifizierung von Benutzers dient.
- PBKDF2 (Password-Based Key Derivation Function 2): ist eine genormte Funktion, um von einem Master-Passwort einen Schlüssel abzuleiten, der in einem symmetrischen Verfahren eingesetzt werden kann.
- AES-256-CBC ist ein hochsicherer Verschlüsselungsalgorithmus, der zur Ver- und Entschlüsselung von Passwörtern verwendet wird.

Encryption


MasterKey/VaultKey
Salt
save
PBKDF2
Saved Accounts
derive
save
Key
IV
AES
Cipher
Plain
encrypt

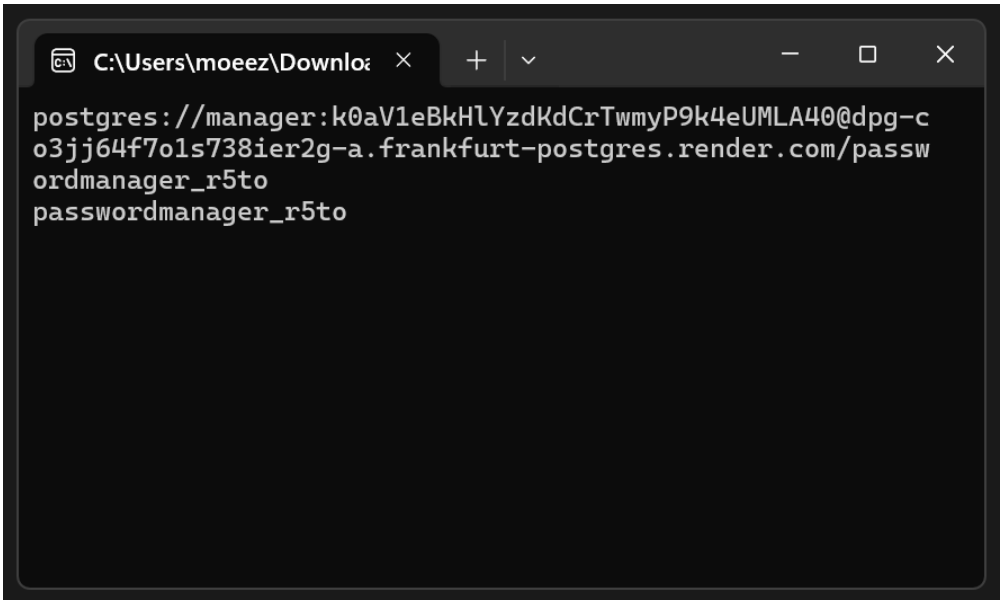
Decryption


MasterKey/VaultKey
Saved Accounts
read
read
Salt
Cipher
PBKDF2
AES
derive
decrypt
Key
IV
Plain

API/Server:

- In der Main.cpp-File wird eine Instanz der Klasse Server erstellt. -Es handelt sich um ein Objekt aus der httplib-Bibliothek. -Das Server-Objekt lauscht auf eingehende Request an Port:1234. -Es liest die Request und löst die entsprechenden Get- oder Post-Methoden aus. -In den Get- oder Post-Methoden werden Eingabeparameter aus der Request-URI oder dem Request-Body extrahiert und als Eingabe-Argumente an Methoden eines PostgreSQLCleint-Objekts übergeben.

PostgreSQLCleint: Diese Klasse ist für die Erstellung von Abfragen(Queries) zuständig und sendet diese dann an die Datenbank, damit sie ausgeführt werden können.



Database:

Wenn ein Benutzer versucht, sich anzumelden, sendet er seinen Benutzernamen und seinen Authentifikation-Key an den Server. Der Server(API) hasht den Authentifikation-Key, um einen Wert zu erzeugen, der mit der Identität(user_identity) in der Master_Accounts-Tabelle verglichen wird, um zu prüfen, ob der Benutzer existiert und ob die richtigen Anmeldedaten angegeben wurden.

Es gibt keinen Fremdschlüssel in saved_accounts, der sie mit der Tabelle master_accounts verbindet. Dies ist beabsichtigt. Auf dem Server wird der Authentifikation-Key weiter gehasht, um einen eindeutigen Master-Identity-Key für jedes Master-Account zu erzeugen, und dieser wird mit jedem Saved-Account gespeichert, um später festzustellen, welches Saved-Account zu welchem Master-Account gehört. Damit soll sichergestellt werden, dass im Falle einer Kompromittierung der Datenbank niemand weiß, wem ein bestimmtes Passwort gehört.

master_accounts:

user_identity: dies ist der eindeutige Identifikator jedes Hauptkontos (E-Mails/Namen).

pass: der gehashte Authentifizierungsschlüssel.

saved_accounts*:*

saved_account_id: automatisch inkrementierte ID zur eindeutigen Identifizierung jedes gespeicherten Kontos.

master_account_identity: speichert die gehashte user_identity, um sie mit dem Hauptkonto zu verknüpfen.

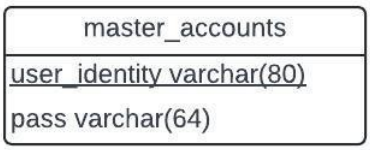
platform: die Plattform, auf der das gespeicherte Konto gespeichert ist, z. B. Facebook, Gmail, Studip.

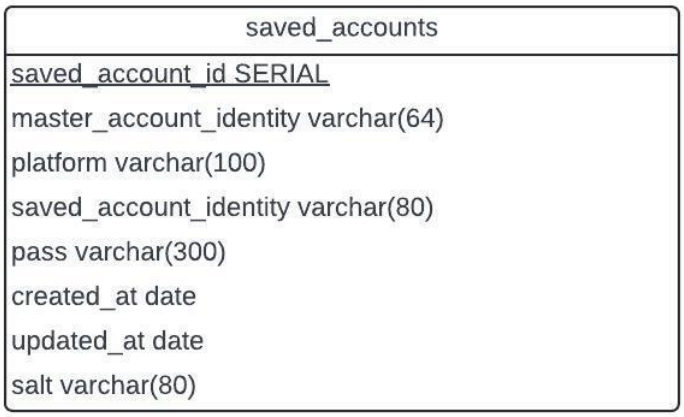
saved_account_identity: den Benutzernamen oder die Identität, die zur Anmeldung bei der jeweiligen Plattform verwendet wird.

pass: das verschlüsselte Passwort.

created_at, *updated at*: das Datum, an dem das Konto erstellt und zuletzt aktualisiert wurde unter.

salt: ein zufälliges Salt für jedes Saved_account, das zum Ableiten des Entschlüsselungsschlüssels verwendet wird.





Setup:

Sie können den Projektordner herunterladen unter:

https://1drv.ms/u/s!AgwwB1qPV5NIhbIRhqdbj1vS0ewibg?e=0NPN88

Oder unter:

https://drive.google.com/drive/folders/1C_6_TZIEBVjurY-qVQcF00OBdLdO1YLs?usp=drive_link

Bevor die Passwortmanager-Anwendung gestartet wird, muss die API bereits laufen.

Die API/Server-Anwendung befindet sich unter ProjectInformatik\API-Executable\APIKennwort.exe.

Die Passwortmanager-Anwendung befindet sich unter ProjectInformatik\PasswordManager-Executable\PasswordManager.exe.

Derzeit wird die Datenbank im Internet über ein kostenloses Tariff von Render.com gehostet. Um den Speicherort der Datenbank zu ändern, fügen Sie einfach ein Connection-String in die Datei DB_Connection.txt im Ordner Files ein. Dort muss nur die erste Zeile geändert werden, da die erste Zeile vom Programm gelesen wird. Ein Beispiel für einen Connection-String, um eine Verbindung zu einer lokalen Postgres-Datenbank herzustellen, sollte unten in die Datei geschrieben werden.

Um den Quellcode zu sehen, wählen/klicken Sie die Projekt-Solution-File (.sln). Um das Projekt zu erstellen, müssen die folgenden Bibliotheken installiert werden. Die Quellen für diese Bibliotheken sind:

OpenSSL: https://github.com/openssl/openssl

WxWidgets: https://github.com/wxWidgets/wxWidgets

libpqxx for postgreSQL: https://github.com/jtv/libpqxx

cpp-http(Http-Bibliothek): https://github.com/yhirose/cpp-httplib