

Менеджер паролей: настольное приложение на Python и Tkinter с симметричным шифрованием

Ран Максим Александрович

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В работе рассматривается разработка настольного офлайн-менеджера паролей, реализованного на языке Python с использованием библиотеки Tkinter. Приложение предназначено для локального хранения учётных данных (сайт, логин, пароль, заметка) с использованием симметричного шифрования Fernet. Ключ шифрования не хранится на диске и каждый раз выводится из мастер-пароля пользователя с помощью функции PBKDF2HMAC, после чего существует только в оперативной памяти в течение сеанса. Хранение данных организовано на основе встроенной СУБД SQLite с поддержкой выбора файла хранилища, применения дополнительного фактора в виде PIN-кода, блокировки при множественных неудачных попытках входа и автоматической блокировки по таймауту бездействия. Для каждой записи используется отдельная соль, что позволяет реализовать пер-записное ключевание и усложняет анализ содержимого при компрометации части данных. Интерфейс включает экран выбора и создания хранилища, форму аутентификации по мастер-паролю, основное окно с формой редактирования записи, таблицей записей на базе tkinter.Treeview, панелью поиска и фильтрации, а также модуль генерации паролей с оценкой их стойкости. Реализованы история паролей по каждой записи, журнал действий, а также импорт и экспорт данных в формате CSV. Показано, что связка Python + Tkinter + SQLite в сочетании с библиотекой cryptography позволяет создать удобное настольное приложение для безопасного локального управления учётными данными.

Ключевые слова: Python, Tkinter, менеджер паролей, симметричное шифрование, Fernet, PBKDF2HMAC, SQLite, безопасность данных.

Password Manager: a desktop application in Python and Tkinter with symmetric encryption

Ran Maksim Aleksandrovich

Sholom-Aleichem Priamursky State University

Student

Abstract

This paper presents the development of a desktop offline password manager implemented in Python using the Tkinter library. The application is designed for

local storage of user credentials (website, login, password, note) using symmetric Fernet encryption. The encryption key is not stored on disk: it is derived from a master password via PBKDF2HMAC and kept only in RAM during the active session. Data storage is based on the embedded SQLite database and supports selecting a vault file, using an optional PIN as a second factor, lockout after multiple failed login attempts, and automatic session locking after an inactivity timeout. Each entry has its own salt, which enables per-entry keying and makes large-scale analysis more difficult in case of partial data compromise. The user interface includes a storage selection and creation screen, a master-password login form, and a main window with an entry editor, a records table built on ttk.Treeview, a search and filtering panel, and a password generator with strength estimation. The application also provides per-entry password history, an audit log, and CSV import/export. The study demonstrates that the combination of Python, Tkinter, SQLite and the cryptography library allows building a convenient desktop application for secure local credential management.

Keywords: Python, Tkinter, password manager, symmetric encryption, Fernet, PBKDF2HMAC, SQLite, data security.

1 Введение

1.1 Актуальность

Современные пользователи ежедневно взаимодействуют с большим количеством веб-сервисов и приложений, требующих аутентификации: электронная почта, социальные сети, банковские системы, образовательные платформы и т.д. В таких условиях возникает проблема безопасного хранения и использования большого числа уникальных паролей. Распространённой практикой остаётся использование одинаковых или мало отличающихся паролей на разных ресурсах, а также хранение учётных данных «вручную» – в текстовых файлах, блокнотах или браузерах без достаточной защиты. Это существенно повышает риск компрометации аккаунтов при утечке данных хотя бы с одного сервиса.

Многие коммерческие менеджеры паролей предлагают облачные хранилища и синхронизацию между устройствами, однако не все пользователи готовы доверять свои критичные данные внешним сервисам или зависят от постоянного доступа к сети. В ряде случаев требуется простое и понятное настольное решение, работающее локально на одном компьютере, не отправляющее данные в Интернет и предоставляющее прозрачную модель безопасности.

В этой связи актуальной представляется разработка компактного настольного приложения, позволяющего безопасно хранить пароли и сопутствующие данные на локальном устройстве, обеспечивать защиту с помощью мастер-пароля и, по возможности, дополнительных факторов, а также предоставлять вспомогательные функции – генерацию стойких паролей, историю изменений и журнал действий пользователя.

1.2 Обзор исследований

Для более глубокого знакомства с языком программирования Python и его использованием при разработке графических интерфейсов целесообразно обратиться к следующим источникам. В.А. Машков и В.И. Литвиненко анализируют применение Python для решения задач самодиагностики на системном уровне [1]. Г.Д. Бухарова и П.С. Комельских акцентируют внимание на значимости включения Python в образовательный процесс [2]. Г.С. Сейдаметов раскрывает особенности использования Python в подготовке инженеров-программистов [3], а Э.А. Усеинов демонстрирует практику применения объектно-ориентированного подхода в рамках изучения данного языка [4]. Разработка приложений с графическим пользовательским интерфейсом (GUI) на Python с использованием библиотеки Tkinter также получила широкое освещение в научных работах. В частности, А.С. Матвеева в своей статье рассматривает применение паттерна «Команда» для построения модульных и структурированных приложений на Tkinter, что способствует повышению гибкости, расширяемости, надежности и удобства использования программных систем [5].

1.3 Цель исследования

Целью работы является разработка настольного приложения-менеджера паролей на языке Python с использованием библиотеки Tkinter и СУБД SQLite, реализующего безопасное локальное хранение учётных данных на основе симметричного шифрования Fernet и производного ключа от мастер-пароля.

2 Материалы и методы

Для реализации приложения использовался следующий набор технологий и библиотек:

- Python – язык программирования общего назначения, используемый для реализации логики приложения, криптографических операций и взаимодействия с базой данных;
- Tkinter – стандартная библиотека Python для создания графического интерфейса: главное окно, формы ввода, кнопки, флажки, текстовые поля и модальные диалоги;
- tkinter.ttk (Treeview, Combobox и др.) – набор виджетов с расширенными возможностями для отображения табличных данных и элементов управления, применяемый для представления списка записей и панели фильтров;
- SQLite – встраиваемая реляционная СУБД, используемая для локального хранения конфигурации хранилища, записей с паролями, истории изменений и журнала действий;
- cryptography (Fernet, PBKDF2HMAC) – библиотека для криптографических операций: реализация алгоритма симметричного шифрования Fernet и функции PBKDF2HMAC на базе SHA-256 для вывода ключа шифрования из мастер-пароля;

- csv – стандартный модуль Python для импорта и экспорта записей в текстовом формате CSV с фиксированным набором полей;
- стандартные модули hashlib, os, datetime, base64, используемые для хэширования PIN-кода, работы с файловой системой, обработки времени и кодирования/декодирования двоичных данных.

Использование полностью встроенных средств Python и локального движка SQLite позволяет создать автономное настольное приложение, не зависящее от внешних серверов и работающее целиком на стороне пользователя.

3 Результаты и обсуждения

Файл main.py выполняет роль минималистичной точки входа (рис.1). В нём:

- импортируется модуль tkinter и основной класс приложения PasswordManagerApp из файла gui.py;
- создаётся корневое окно Tkinter;
- инициализируется экземпляр PasswordManagerApp, которому передаётся корневое окно;
- запускается главный цикл обработки событий root.mainloop().

```
import tkinter as tk

from gui import PasswordManagerApp

def main() -> None:
    root = tk.Tk()
    PasswordManagerApp(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

Рисунок 1. Код файла main

Вся бизнес-логика и работа с данными вынесены в другие модули, поэтому main.py отвечает только за запуск графического интерфейса, что соответствует принципу единственной ответственности.

Файл models.py содержит прикладные модели данных, оформленные в виде dataclass, что обеспечивает удобный доступ к полям и автоматическую генерацию служебных методов.

Модель конфигурации хранилища VaultConfig

Класс VaultConfig описывает конфигурацию конкретного файла-хранилища (vault) (рис.2):

- id – идентификатор конфигурации (в текущей реализации всегда 1 для одного хранилища);
- salt – строковое представление соли (base64), используемой при выводе ключа шифрования из мастер-пароля;

- kdf_iterations – число итераций алгоритма PBKDF2HMAC;
- created_at – дата и время создания хранилища;
- verify_token – зашифрованный контрольный токен (строка «vault-ok» в зашифрованном виде), применяемый для проверки корректности мастер-пароля;
- pin_hash – хеш PIN-кода (SHA-256) при использовании второго фактора аутентификации;
- fail_count – счётчик неудачных попыток входа;
- locked_until – момент времени, до которого вход в хранилище заблокирован (при превышении порога неудачных попыток).

```
from dataclasses import dataclass
from datetime import datetime
from typing import Optional

@dataclass
class VaultConfig:
    """Configuration for the password vault."""

    id: int
    salt: str
    kdf_iterations: int
    created_at: datetime
    verify_token: str
    pin_hash: str = ""
    fail_count: int = 0
    locked_until: Optional[datetime] = None
```

Рисунок 2. Импорт и класс VaultConfig

Данная модель отражает все параметры, необходимые для реализации политики безопасности и управления состоянием хранилища.

Модель записи PasswordEntry

Класс PasswordEntry представляет одну запись с учётными данными (рис.3):

- id – идентификатор записи;
- site – сайт или сервис (URL/домен);
- login – логин или адрес электронной почты;
- encrypted_password – зашифрованный пароль (токен Fernet в текстовом виде);
- notes – произвольная заметка;
- category – категория (например, «Банки», «Почта», «Соцсети»);
- tags – теги, записанные в виде строки;
- created_at и updated_at – даты создания и последнего изменения записи;
- favorite – логический флаг, отмечающий запись как «избранную»;
- entry_salt – индивидуальная соль записи (base64), используемая при выводе отдельного ключа шифрования на запись.

```
@dataclass
class PasswordEntry:
    """A single password record stored in the vault."""

    id: Optional[int]
    site: str
    login: str
    encrypted_password: str
    notes: str
    category: str
    tags: str
    created_at: datetime
    updated_at: datetime
    favorite: bool = False
    entry_salt: Optional[str] = None
```

Рисунок 3. Класс PasswordEntry

Таким образом, PasswordEntry инкапсулирует все данные, необходимые для отображения, фильтрации и криптографической обработки конкретной записи.

Файл `crypto_utils.py` содержит все криптографические примитивы и функции генерации паролей. Это позволяет централизовать операции, связанные с безопасностью, и упрощает их анализ и модификацию.

В начале задаются константы (рис.4):

- `DEFAULT_ITERATIONS = 390_000` – базовое количество итераций для PBKDF2HMAC;
- `KEY_LENGTH = 32` – длина ключа в байтах (256 бит).

```
import base64
import os
import string
from typing import Optional, Tuple

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.fernet import Fernet, InvalidToken

DEFAULT_ITERATIONS = 390_000
KEY_LENGTH = 32
```

Рисунок 4. Импорт и константы

Вывод ключа и работа с Fernet

Функция `generate_salt()` генерирует криптографически стойкую случайную соль заданной длины с использованием `os.urandom` (рис.5).

```
def generate_salt(length: int = 16) -> bytes:
    """Return cryptographically strong random salt bytes."""
    return os.urandom(length)
```

Рисунок 5. Функция generate_salt

Функция `derive_key(master_password, salt, iterations)` реализует вывод ключа по схеме PBKDF2HMAC (рис.6):

- в качестве алгоритма хеширования используется SHA-256;
- используется соль из конфигурации хранилища или комбинация солей;
- выполняется заданное количество итераций;
- результат кодируется в формате `urlsafe_b64encode` для совместимости с Fernet.

```
def derive_key(master_password: str, salt: bytes, iterations: int = DEFAULT_ITERATIONS) -> bytes:
    """Derive a Fernet-compatible key from a master password and salt."""
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=KEY_LENGTH,
        salt=salt,
        iterations=iterations,
        backend=default_backend(),
    )
    key = kdf.derive(master_password.encode("utf-8"))
    return base64.urlsafe_b64encode(key)
```

Рисунок 6. Функция `derive_key`

На основе этой функции реализованы:

- `build_fernet(master_password, salt, iterations)` – возвращает объект Fernet, общий для всего хранилища (рис.7);
- `derive_entry_fernet(master_password, vault_salt, entry_salt, iterations)` – формирует отдельный объект Fernet для конкретной записи, используя комбинацию соли хранилища и соли записи. Это обеспечивает пер-записное ключевание (per-entry keying) (рис.7).

```
def build_fernet(master_password: str, salt: bytes, iterations: int = DEFAULT_ITERATIONS) -> Fernet:
    return Fernet(derive_key(master_password, salt, iterations))

def derive_entry_fernet(master_password: str, vault_salt: bytes, entry_salt: bytes, iterations: int = DEFAULT_ITERATIONS) -> Fernet:
    """Per-entry key derived from master password + combined salts."""
    combined_salt = vault_salt + entry_salt
    return Fernet(derive_key(master_password, combined_salt, iterations))
```

Рисунок 7. Код

Шифрование и расшифровка

Функции `encrypt_text(fernet, plaintext)` и `decrypt_text(fernet, token)` обеспечивают (рис.8):

- шифрование текста в токен (строка, возвращаемая Fernet);
- обратную операцию, при которой в случае некорректного токена или неправильного ключа возвращается `None` без выброса исключения.

```
def encrypt_text(fernet: Fernet, plaintext: str) -> str:
    """Encrypt text and return base64-encoded token."""
    token = fernet.encrypt(plaintext.encode("utf-8"))
    return token.decode("utf-8")

def decrypt_text(fernet: Fernet, token: str) -> Optional[str]:
    """Decrypt token and return plaintext or None if invalid."""
    try:
        return fernet.decrypt(token.encode("utf-8")).decode("utf-8")
    except InvalidToken:
        return None
```

Рисунок 8. Код

Такая схема упрощает обработку ошибок в пользовательском интерфейсе: при неудачной расшифровке можно вывести пользователю понятное сообщение, не прерывая работу программы.

Генерация паролей и оценка их сложности

Функция `generate_password(...)` реализует гибкий генератор паролей с параметрами (рис.9):

- длина пароля;
- использование заглавных и строчных букв, цифр, специальных символов;
- опция исключения визуально похожих символов (O/0, l/1 и т.п.).

```
def generate_password(length: int = 16, use_upper: bool = True, use_lower: bool = True, use_digits: bool = True, use_symbols: bool = True, exclude_similar: bool = False) -> str:
    """Generate a password according to selected options."""
    upper = string.ascii_uppercase
    lower = string.ascii_lowercase
    digits = string.digits
    symbols = "!@#$%^&*()-_+=[]{}~`~'\",./\:\;\<>`~\`"

    if exclude_similar:
        similar = set("01l")
        upper = "".join(ch for ch in upper if ch not in similar)
        lower = "".join(ch for ch in lower if ch not in similar)
        digits = "".join(ch for ch in digits if ch not in similar)

    groups = []
    if use_upper:
        groups.append(upper)
    if use_lower:
        groups.append(lower)
    if use_digits:
        groups.append(digits)
    if use_symbols:
        groups.append(symbols)

    if not groups:
        return ""

    import random

    # Ensure at least one char from each selected group
    password_chars = [random.choice(g) for g in groups]
    alphabet = "".join(groups)
    password_chars.extend(random.choice(alphabet) for _ in range(max(length - len(password_chars), 0)))
    random.shuffle(password_chars)
    return "".join(password_chars[:length])
```

Рисунок 9. Функция `generate_password`

Гарантируется, что в пароле будет хотя бы по одному символу из каждой выбранной группы, а оставшиеся символы заполняются случайно из объединённого алфавита и перемешиваются.

Функция `password_strength(password)` вычисляет простую метрику сложности (рис.10):

- учитывает длину строки;
- наличие букв разных регистров, цифр и специальных символов;
- возвращает текстовую характеристику («слабый», «средний», «сильный») и числовой балл (0–100).

```
def password_strength(password: str) -> Tuple[str, int]:
    """Return textual strength and score 0-100."""
    score = 0
    if len(password) >= 8:
        score += 20
    if len(password) >= 12:
        score += 15
    if len(password) >= 16:
        score += 15
    if any(c.islower() for c in password):
        score += 10
    if any(c.isupper() for c in password):
        score += 10
    if any(c.isdigit() for c in password):
        score += 15
    if any(c in "!@#$%^&*()-_+=[]{};:,.?/\\|" for c in password):
        score += 15

    if score < 40:
        label = "слабый"
    elif score < 70:
        label = "средний"
    else:
        label = "сильный"
    return label, min(score, 100)
```

Рисунок 10. Функция Password_strength

Данный результат используется в графическом интерфейсе для отображения текущей «сложности» введённого или сгенерированного пароля.

Файл database.py реализует класс Database, инкапсулирующий всю работу с SQLite-базой данных. Подключение к файлу БД происходит в конструкторе: соединение создаётся с использованием параметра row_factory = sqlite3.Row для удобного доступа к полям по именам.

Создание и эволюция схемы БД

Метод _ensure_schema() создаёт (при необходимости) основные таблицы (рис.11.1-11.2):

- vault – конфигурация хранилища: соль, число итераций KDF, зашифрованный проверочный токен, дата создания, хеш PIN-кода, счётчик неудачных попыток и время разблокировки;
- entries – таблица записей с полями сайта, логина, зашифрованного пароля, заметок, категории, тегов, признака избранного, индивидуальной соли записи, дат создания и изменения;
- password_history – история паролей для каждой записи (идентификатор записи, зашифрованный прежний пароль, дата изменения);
- audit_log – журнал действий (тип действия, идентификатор записи, сайт, время события).

```

import sqlite3
from datetime import datetime
from typing import List, Optional

from models import PasswordEntry, VaultConfig

class Database:
    def __init__(self, db_path: str = "vault.db") -> None:
        self.db_path = db_path
        self.conn = sqlite3.connect(self.db_path)
        self.conn.row_factory = sqlite3.Row
        self._ensure_schema()

    def _ensure_schema(self) -> None:
        cur = self.conn.cursor()
        cur.execute(
            """
            CREATE TABLE IF NOT EXISTS vault (
                id INTEGER PRIMARY KEY,
                salt TEXT NOT NULL,
                kdf_iterations INTEGER NOT NULL,
                verify_token TEXT NOT NULL,
                created_at TEXT NOT NULL,
                pin_hash TEXT DEFAULT '',
                fail_count INTEGER DEFAULT 0,
                locked_until TEXT
            );
            """
        )
        cur.execute(
            """
            CREATE TABLE IF NOT EXISTS entries (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                site TEXT NOT NULL,
                login TEXT NOT NULL,
                encrypted_password TEXT NOT NULL,
                notes TEXT,
                category TEXT DEFAULT '',
                tags TEXT DEFAULT '',
                favorite INTEGER DEFAULT 0,
                entry_salt TEXT,
                created_at TEXT NOT NULL,
                updated_at TEXT NOT NULL
            );
            """
        )

```

Рисунок 11.1. Метод _ensure_schema

```

cur.execute(
    """
    CREATE TABLE IF NOT EXISTS password_history (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        entry_id INTEGER NOT NULL,
        encrypted_password TEXT NOT NULL,
        changed_at TEXT NOT NULL,
        FOREIGN KEY(entry_id) REFERENCES entries(id) ON DELETE CASCADE
    );
    """
)

cur.execute(
    """
    CREATE TABLE IF NOT EXISTS audit_log (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        action TEXT NOT NULL,
        entry_id INTEGER,
        site TEXT,
        created_at TEXT NOT NULL
    );
    """
)

self.conn.commit()
self._add_missing_columns()

```

Рисунок 11.2. Метод _ensure_schema

После создания таблиц вызывается метод `_add_missing_columns()` (рис.12), который при помощи `PRAGMA table_info` проверяет наличие всех требуемых столбцов и при необходимости добавляет недостающие (например, поля `category`, `tags`, `favorite`, `entry_salt` в таблицу `entries` и `pin_hash`, `fail_count`, `locked_until` в таблицу `vault`). Это позволяет корректно работать с уже существующими файлами БД при расширении схемы.

```
def _add_missing_columns(self) -> None:
    cur = self.conn.cursor()
    # entries columns
    cur.execute("PRAGMA table_info(entries);")
    existing = {row["name"] for row in cur.fetchall()}
    column_defaults = {
        "category": ("TEXT", ""),
        "tags": ("TEXT", ""),
        "favorite": ("INTEGER", "0"),
        "entry_salt": ("TEXT", "NULL"),
    }
    for name, (ctype, default) in column_defaults.items():
        if name not in existing:
            cur.execute(f"ALTER TABLE entries ADD COLUMN {name} {ctype} DEFAULT {default};")

    # vault columns
    cur.execute("PRAGMA table_info(vault);")
    existing_vault = {row["name"] for row in cur.fetchall()}
    vault_defaults = {
        "pin_hash": ("TEXT", ""),
        "fail_count": ("INTEGER", "0"),
        "locked_until": ("TEXT", "NULL"),
    }
    for name, (ctype, default) in vault_defaults.items():
        if name not in existing_vault:
            cur.execute(f"ALTER TABLE vault ADD COLUMN {name} {ctype} DEFAULT {default};")
    self.conn.commit()
```

Рисунок 12. Метод _add_missing_columns

Работа с конфигурацией хранилища

Метод get_vault_config() выбирает первую (и единственную) строку таблицы vault, преобразует поля в объект VaultConfig и при необходимости преобразует строку locked_until обратно в объект datetime (рис.13).

```
def get_vault_config(self) -> Optional[VaultConfig]:
    cur = self.conn.cursor()
    cur.execute("SELECT * FROM vault LIMIT 1;")
    row = cur.fetchone()
    if not row:
        return None
    locked_until = row["locked_until"]
    return VaultConfig(
        id=row["id"],
        salt=row["salt"],
        kdf_iterations=row["kdf_iterations"],
        verify_token=row["verify_token"],
        created_at=datetime.fromisoformat(row["created_at"]),
        pin_hash=row["pin_hash"] or "",
        fail_count=row["fail_count"] or 0,
        locked_until=datetime.fromisoformat(locked_until) if locked_until else None,
    )
```

Рисунок 13. Метод get_vault_config

Метод create_vault_config(salt_b64, iterations, verify_token, pin_hash) (рис.14):

- вставляет новую запись в таблицу vault с идентификатором 1;
- сохраняет соль, число итераций, зашифрованный контрольный токен, дату создания и хеш PIN-кода;
- возвращает сформированный объект VaultConfig.

```
def create_vault_config(self, salt_b64: str, iterations: int, verify_token: str, pin_hash: str = "") -> VaultConfig:
    created_at = datetime.utcnow().isoformat()
    cur = self.conn.cursor()
    cur.execute(
        "INSERT INTO vault (id, salt, kdf_iterations, verify_token, created_at, pin_hash, fail_count, locked_until) VALUES (1, ?, ?, ?, ?, ?, 0, NULL);",
        (salt_b64, iterations, verify_token, created_at, pin_hash),
    )
    self.conn.commit()
    return VaultConfig(
        id=1,
        salt=salt_b64,
        kdf_iterations=iterations,
        verify_token=verify_token,
        created_at=datetime.fromisoformat(created_at),
        pin_hash=pin_hash,
        fail_count=0,
        locked_until=None,
    )
```

Рисунок 14. Метод create_vault_config

Метод `update_vault_fail(fail_count, locked_until)` обновляет счётчик неудачных попыток и время блокировки в таблице `vault` (рис.15). Этот механизм используется при реализации защиты от перебора мастер-пароля и PIN-кода.

```
def update_vault_fail(self, fail_count: int, locked_until: Optional[datetime]) -> None:
    cur = self.conn.cursor()
    cur.execute(
        "UPDATE vault SET fail_count = ?, locked_until = ? WHERE id = 1;",
        (fail_count, locked_until.isoformat() if locked_until else None),
    )
    self.conn.commit()
```

Рисунок 15. Метод update_vault_fail

Операции с записями

Для работы с отдельными записями реализованы следующие методы:

- `add_entry(...)` – добавляет новую запись, устанавливает дату создания и изменения, сохраняет индивидуальную соль записи и зашифрованный пароль, добавляет запись в историю паролей и фиксирует действие в журнале. Возвращает объект `PasswordEntry` для только что созданной записи (рис.16);

```
def add_entry(self, site: str, login: str, encrypted_password: str, notes: str, category: str, tags: str, entry_salt: str, favorite: bool = False) -> PasswordEntry:
    now = datetime.utcnow().isoformat()
    cur = self.conn.cursor()
    cur.execute(
        """
        INSERT INTO entries (site, login, encrypted_password, notes, category, tags, favorite, entry_salt, created_at, updated_at)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?);
        """
    )
    (site, login, encrypted_password, notes, category, tags, int(favorite), entry_salt, now, now),
    self.conn.commit()
    entry_id = cur.lastrowid
    self.add_history(entry_id, encrypted_password, now)
    self.log_action("add", entry_id, site)
    return PasswordEntry(
        id=entry_id,
        site=site,
        login=login,
        encrypted_password=encrypted_password,
        notes=notes,
        category=category,
        tags=tags,
        created_at=datetime.fromisoformat(now),
        updated_at=datetime.fromisoformat(now),
        favorite=favorite,
        entry_salt=entry_salt,
    )
```

Рисунок 16. Метод add_entry

- `update_entry(entry_id, ...)` – обновляет существующую запись, фиксирует новую дату изменения, добавляет новый зашифрованный пароль в таблицу `password_history` и записывает действие «update» в журнал (рис.17);

```
def update_entry(self, entry_id: int, site: str, login: str, encrypted_password: str, notes: str, category: str, tags: str, favorite: bool) -> None:
    now = datetime.utcnow().isoformat()
    cur = self.conn.cursor()
    cur.execute(
        """
        UPDATE entries
        SET site = ?, login = ?, encrypted_password = ?, notes = ?, category = ?, tags = ?, favorite = ?, updated_at = ?
        WHERE id = ?;
        """
    )
    (site, login, encrypted_password, notes, category, tags, int(favorite), now, entry_id),
    self.conn.commit()
    self.add_history(entry_id, encrypted_password, now)
    self.log_action("update", entry_id, site)
```

Рисунок 17. Метод update_entry

- `delete_entry(entry_id)` – удаляет запись по идентификатору и записывает действие «delete» в журнал (рис.18);

```
def delete_entry(self, entry_id: int) -> None:
    cur = self.conn.cursor()
    cur.execute("DELETE FROM entries WHERE id = ?;", (entry_id,))
    self.conn.commit()
    self.log_action("delete", entry_id, None)
```

Рисунок 18. Метод `delete_entry`

- `get_entry(entry_id)` – возвращает конкретную запись в виде объекта `PasswordEntry` (рис.19);

```
def get_entry(self, entry_id: int) -> Optional[PasswordEntry]:
    cur = self.conn.cursor()
    cur.execute("SELECT * FROM entries WHERE id = ?;", (entry_id,))
    row = cur.fetchone()
    return self._row_to_entry(row) if row else None
```

Рисунок 19. Метод `get_entry`

- `get_all_entries()` – выбирает все записи, сортируя их по дате изменения (рис.20);

```
def get_all_entries(self) -> List[PasswordEntry]:
    cur = self.conn.cursor()
    cur.execute("SELECT * FROM entries ORDER BY updated_at DESC;")
    rows = cur.fetchall()
    return [self._row_to_entry(row) for row in rows]
```

Рисунок 20. Метод `get_all_entries`

- `search_entries(query, category_filter, favorites_only)` – выполняет поиск по сайту, логину, заметкам и тегам, с учётом выбранной категории и флага «только избранные», возвращая список объектов `PasswordEntry` (рис.21);

```
def search_entries(self, query: str, category_filter: str = "", favorites_only: bool = False) -> List[PasswordEntry]:
    like = f"%{query}%"
    sql = """
        SELECT * FROM entries
        WHERE (site LIKE ? OR login LIKE ? OR notes LIKE ? OR tags LIKE ?)
    """
    params = [like, like, like, like]
    if category_filter:
        sql += " AND category = ?"
        params.append(category_filter)
    if favorites_only:
        sql += " AND favorite = 1"
    sql += " ORDER BY updated_at DESC;"
    cur = self.conn.cursor()
    cur.execute(sql, params)
    rows = cur.fetchall()
    return [self._row_to_entry(row) for row in rows]
```

Рисунок 21. Метод `search_entries`

- `get_categories()` – возвращает список уникальных категорий, присутствующих в записях (рис.22).

```
def get_categories(self) -> List[str]:
    cur = self.conn.cursor()
    cur.execute("SELECT DISTINCT category FROM entries WHERE category IS NOT NULL AND category != '' ORDER BY category;")
    return [row[0] for row in cur.fetchall()]
```

Рисунок 22. Метод `get_categories`

Для работы с историей и журналом предусмотрены методы (рис.23):

- `add_history(entry_id, encrypted_password, changed_at)` и `get_history(entry_id)` – добавление и выборка истории паролей по конкретной записи;
- `log_action(action, entry_id, site)` и `get_audit_log(limit)` – фиксирование действий пользователя (добавление, обновление, удаление) и выборка последних N записей журнала для отображения в интерфейсе.

```
def add_history(self, entry_id: int, encrypted_password: str, changed_at: Optional[str] = None) -> None:
    timestamp = changed_at or datetime.utcnow().isoformat()
    cur = self.conn.cursor()
    cur.execute(
        "INSERT INTO password_history (entry_id, encrypted_password, changed_at) VALUES (?, ?, ?);",
        (entry_id, encrypted_password, timestamp),
    )
    self.conn.commit()

def get_history(self, entry_id: int):
    cur = self.conn.cursor()
    cur.execute(
        "SELECT encrypted_password, changed_at FROM password_history WHERE entry_id = ? ORDER BY changed_at DESC;",
        (entry_id,),
    )
    return cur.fetchall()

def log_action(self, action: str, entry_id: Optional[int], site: Optional[str]) -> None:
    cur = self.conn.cursor()
    cur.execute(
        "INSERT INTO audit_log (action, entry_id, site, created_at) VALUES (?, ?, ?, ?);",
        (action, entry_id, site, datetime.utcnow().isoformat()),
    )
    self.conn.commit()

def get_audit_log(self, limit: int = 200):
    cur = self.conn.cursor()
    cur.execute("SELECT * FROM audit_log ORDER BY created_at DESC LIMIT ?;", (limit,))
    return cur.fetchall()
```

Рисунок 23. Код

Внутренний метод `_row_to_entry(row)` преобразует результирующую строку SQLite в экземпляр `PasswordEntry`, конвертируя строки дат в объекты `datetime` и приводя логические поля к нужному типу (рис.24).

```
def _row_to_entry(self, row: sqlite3.Row) -> PasswordEntry:
    return PasswordEntry(
        id=row["id"],
        site=row["site"],
        login=row["login"],
        encrypted_password=row["encrypted_password"],
        notes=row["notes"] or "",
        category=row["category"] or "",
        tags=row["tags"] or "",
        created_at=datetime.fromisoformat(row["created_at"]),
        updated_at=datetime.fromisoformat(row["updated_at"]),
        favorite=bool(row["favorite"]),
        entry_salt=row["entry_salt"],
    )
```

Рисунок 24. Метод _row_to_entry

Файл `gui.py` является наиболее объёмным и отвечает за реализацию графического интерфейса на Tkinter, а также за связывание всех подсистем: криптографии, базы данных и пользовательских сценариев. Основным элементом – класс `PasswordManagerApp`.

Инициализация приложения и общие настройки

В конструкторе `__init__` (рис.25):

- настраивается главное окно (заголовок, размер, возможность изменения);
- инициализируются параметры хранилища: путь к файлу БД, объект `Database`, конфигурация `vault`, объект `Fernet`, текущее значение мастер-пароля и соли хранилища;
- задаются параметры безопасности: максимальное число неудачных попыток входа, длительность блокировки, тайм-аут бездействия;
- создаются переменные состояния для полей формы (сайт, логин, пароль, категория, теги, заметки, флаги избранного, фильтров и т.п.);
- настраиваются переменные для опций генератора паролей и переменная для отображения сложности пароля;
- создаётся структура для сортировки записей в таблице (по дате изменения, с возможностью переключения направления);
- вызываются методы привязки обработчиков активности и построения первого экрана выбора хранилища.

```

class PasswordManagerApp:
    def __init__(self, root: tk.Tk, db_path: str = "vault.db") -> None:
        self.root.title("Менеджер паролей")
        self.root.geometry("1100x680")

        self.db_path = db_path
        self.db = None
        self.vault_config = None
        self.fernet = None
        self.master_password: str | None = None
        self.vault_salt_bytes: bytes | None = None
        self.selected_entry_id = None

        # Security options
        self.max_fail = 3
        self.lock_delay_seconds = 30
        self.idle_timeout_ms = 5 * 60 * 1000
        self.idle_job = None

        # UI state
        self.storage_frame = None
        self.login_frame = None
        self.main_frame = None
        self.status_var = tk.StringVar(value="")
        self.site_var = tk.StringVar()
        self.login_var = tk.StringVar()
        self.password_var = tk.StringVar()
        self.category_var = tk.StringVar()
        self.tags_var = tk.StringVar()
        self.search_var = tk.StringVar()
        self.category_filter_var = tk.StringVar(value="Бcd")
        self.favorites_only_var = tk.BooleanVar(value=False)
        self.favorite_var = tk.BooleanVar(value=False)

        # Generator options
        self.gen_length = tk.IntVar(value=16)
        self.gen_upper = tk.BooleanVar(value=True)
        self.gen_lower = tk.BooleanVar(value=True)
        self.gen_digits = tk.BooleanVar(value=True)
        self.gen_symbols = tk.BooleanVar(value=True)
        self.gen_exclude_similar = tk.BooleanVar(value=False)
        self.strength_var = tk.StringVar(value="Сложность: -")

        # Sorting
        self.sort_state = ("updated", False) # column, reverse

        self._bind_activity_listeners()
        self._build_storage_ui()

```

Рисунок 25. Конструктор __init__

Методы `_bind_activity_listeners()` и `_reset_idle_timer()` реализуют отслеживание активности пользователя (нажатие клавиш, клики мышью) и сброс таймера автоматической блокировки. По истечении заданного интервала бездействия вызывается `_auto_lock()`, который очищает ключи из памяти и возвращает приложение к экрану входа (рис.26).

```

def _bind_activity_listeners(self) -> None:
    self.root.bind_all("<Any-KeyPress>", lambda event: self._reset_idle_timer())
    self.root.bind_all("<Any-Button>", lambda event: self._reset_idle_timer())

def _reset_idle_timer(self) -> None:
    if self.idle_job:
        self.root.after_cancel(self.idle_job)
    self.idle_job = self.root.after(self.idle_timeout_ms, self._auto_lock)

def _auto_lock(self) -> None:
    if self.fernet:
        messagebox.showinfo("Блокировка", "Сеанс завершён из-за бездействия.")
        self._logout()

```

Рисунок 26. Код

Экран выбора и создания хранилища

Метод `_build_storage_ui()` формирует начальный экран, на котором пользователь (рис.27):

- указывает путь к существующему файлу хранилища и нажимает кнопку «Подключить», либо
- выбирает «Создать новое» и задаёт путь для нового файла.

```
def _build_storage_ui(self) -> None:
    if self.main_frame:
        self.main_frame.destroy()
    if self.login_frame:
        self.login_frame.destroy()
    if self.storage_frame:
        self.storage_frame.destroy()

    self.storage_frame = ttk.Frame(self.root, padding=24)
    self.storage_frame.pack(fill="both", expand=True)

    ttk.Label(self.storage_frame, text="Выберите файл хранилища", font=("Arial", 15, "bold")).pack(pady=(0, 20))

    path_row = ttk.Frame(self.storage_frame)
    path_row.pack(fill="x", pady=4)
    self.path_var = tk.StringVar(value=self.db_path)
    ttk.Entry(path_row, textvariable=self.path_var, width=60).pack(side="left", fill="x", expand=True)
    ttk.Button(path_row, text="Обзор...", command=self._choose_existing_path).pack(side="left", padx=6)

    btn_row = ttk.Frame(self.storage_frame)
    btn_row.pack(pady=12)
    ttk.Button(btn_row, text="Подключить", command=self._connect_storage).pack(side="left", padx=6)
    ttk.Button(btn_row, text="Создать новое", command=self._create_new_storage).pack(side="left", padx=6)
```

Рисунок 27. Метод _build_storage_ui

Обработчики _choose_existing_path(), _connect_storage() и _create_new_storage() работают с диалогами выбора/сохранения файла, создают объект Database, загружают (или не загружают) конфигурацию хранилища и переходят к экрану аутентификации (рис.28).

```
def _choose_existing_path(self) -> None:
    path = filedialog.askopenfilename(title="Выберите vault.db", filetypes=[("SQLite", "*.db"), ("All files", "*.*)])
    if path:
        self.path_var.set(path)

def _connect_storage(self) -> None:
    path = self.path_var.get().strip()
    if not path or not os.path.exists(path):
        messagebox.showerror("Хранилище", "Файл не найден.")
        return
    self.db_path = path
    self.db = Database(self.db_path)
    self.vault_config = self.db.get_vault_config()
    self._build_login_ui()

def _create_new_storage(self) -> None:
    path = filedialog.asksaveasfilename(
        title="Создать хранилище",
        defaultextension=".db",
        filetypes=[("SQLite", "*.db"), ("All files", "*.*)]),
    )
    if not path:
        return
    self.db_path = path
    self.db = Database(self.db_path)
    self.vault_config = None
    self._build_login_ui()
```

Рисунок 28. Код

Внешний вид приложения:

1. Выбор БД (рис.29)

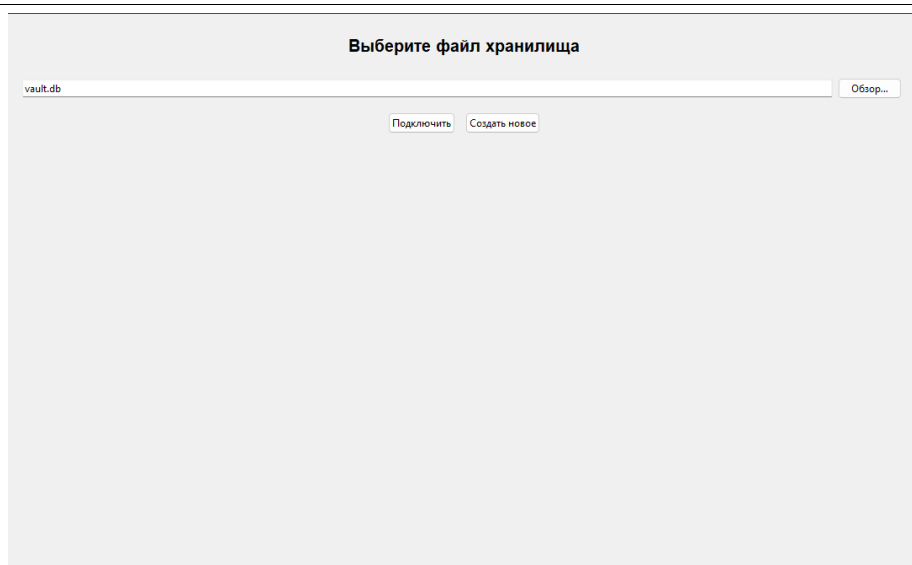


Рисунок 29. Выбор БД

2. Ввод данных для входа (рис.30)

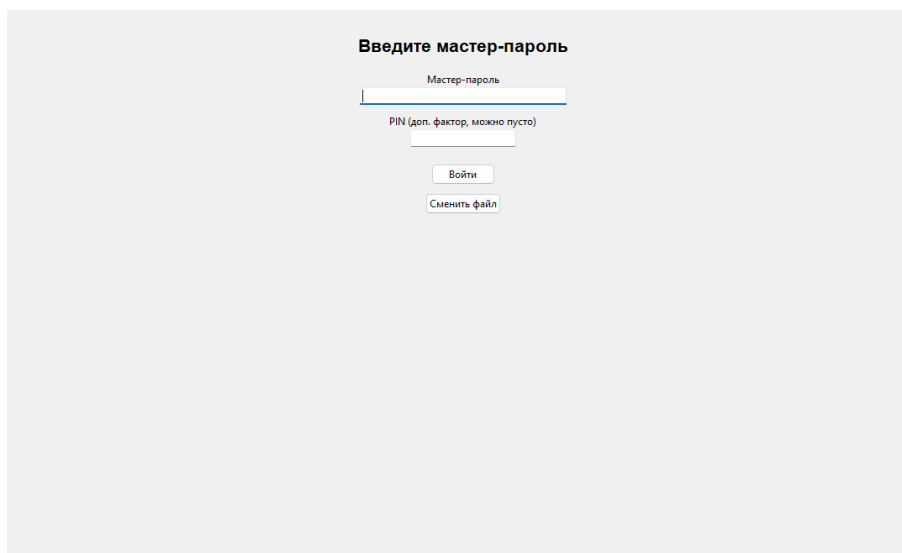


Рисунок 30. Ввод мастер пароля и PIN

3. Главный экран приложения (рис.31).

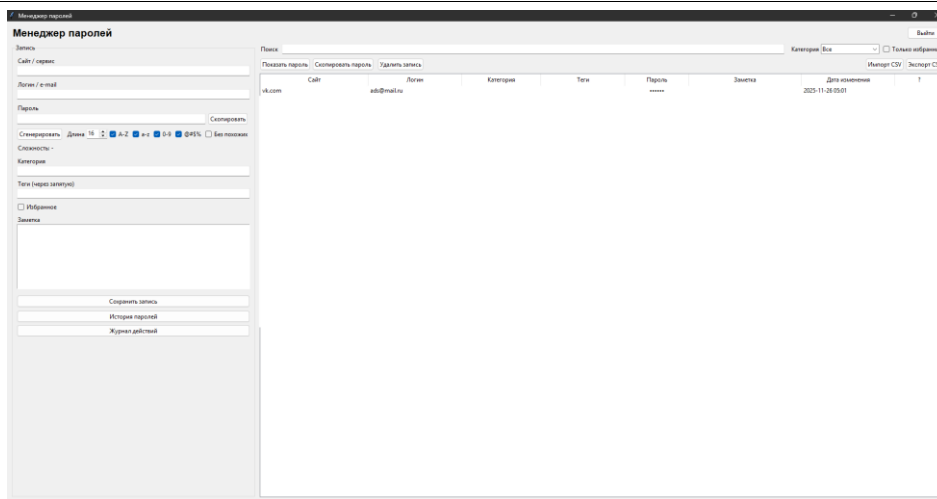


Рисунок 31. Рабочая область

4 Выводы

В ходе работы разработано настольное приложение-менеджер паролей на языке Python с использованием библиотек Tkinter и cryptography, а также СУБД SQLite для локального хранения данных. Приложение обеспечивает:

- выбор и создание файлов хранилища с сохранением конфигурации на уровне отдельного vault;
- аутентификацию пользователя по мастер-паролю с выводом ключа шифрования через PBKDF2HMAC и хранением на диске только соли и проверочного токена;
- опциональный дополнительный фактор в виде PIN-кода, механизм блокировки при множественных неудачных попытках и автоматическую блокировку по таймауту бездействия;
- пер-записное ключевание с индивидуальными солями для каждой записи, что повышает стойкость схемы при анализе зашифрованных данных;
- хранение и редактирование записей, включающих сайт, логин, зашифрованный пароль, заметки, категории и теги, а также выделение избранных записей;
- удобный графический интерфейс с формой ввода, таблицей записей, поиском и фильтрацией;
- генерацию паролей с настраиваемыми параметрами и оценкой их сложности;
- ведение истории паролей и журнала действий пользователя;
- импорт и экспорт данных в формате CSV с осознанным предупреждением о рисках работы с открытыми паролями.

Полученный прототип может быть использован как простое средство локального управления учётными данными, а также как учебный пример при изучении разработки настольных приложений на Python, реализации симметричного шифрования и организации безопасного хранения конфиденциальной информации на стороне клиента.

Библиографический список

1. Машков В.А., Литвиненко В.И. Использование языка программирования python 3 и системы компьютерной алгебры sympy на факультативных занятиях по теории чисел // Электротехнические и компьютерные системы. 2017. С. 48-54
2. Бухарова Г.Д., Комельских П.С. Важность и необходимость внедрения языка программирования python в процесс обучения студентов // Новые информационные технологии в образовании Материалы VII международной научно-практической конференции. 2014. С. 40-42.
3. Сейдаметов Г.С. Особенности использования языка программирования python в подготовке будущих инженеров-программистов // International scientific review. 2019. №. 9(10). С. 50-51
4. Усеинов Э.А. Объектно-ориентированное программирование в рамках дисциплины «Язык программирования python» // Ученые записки Крымского инженерно-педагогического университета. 2012. №. 34. С. 157-160.
5. Матвеева А. С. Реализация паттерна поведения «Команда» в графическом интерфейсе пользователя на Python // Постулат. 2025. №1.