TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Diana Anastassija Turks 233751IVSB

Secure Password Manager

Report

Tallinn 2025

Table of Contents

1 Introduction

Password managers must provide confidentiality, integrity, and availability for stored credentials, while minimizing exposure of secrets in memory, logs, and UI. For the Secure Programming (ICS0022) course, the focus in the creation of a password manager application resides on securely storing and managing user credentials, with an emphasis on encryption, secure memory handling, and protection against common security vulnerabilities like buffer overflows, improper input validation, and unauthorized access.

The present application has the following functionality implemented:

- Key derivation from a master password and randomly generated salt
- Encryption of a a serialized vault using an AEAD cipher with a per-write random nonce
- Atomic writes
- Load/decrypt on demand
- Prompt wiping of secure buffers and plaintexts
- Clipboard integration (best-effort to secure)
- Creator-only access per vault
- Action logging

This architecture is consistent with modern security trends of using Argon2 for password-based key derivation (specifically Argon2id, offering a balanced security profile) **Error! Reference source not found.** and XChaCha20-Poly1305 for authenticated encryption, due to memory-hardness and large nonce space mitigating nonce reuse risks [1]. In addition, structured logging and access-control enforcement are key to operational security and forensics.

2 Cryptographic design

2.1 Key derivation

The application derives the vault key from the master password using **Argon2id** via crypto_pwhash (libsodium) **Error! Reference source not found.**. The salt is randomly generated (crypto_pwhash_SALTBYTES), and OPSLIMIT/MEMLIMIT_MODERATE parameters are used to balance security and performance.

Validations of length (not 0 and not more than the maximum allowed) are implemented, as well as null pointer check. Any failure results in a log entry and is handled gracefully.

```
bool derive_key_from_password(
    const byte* pw, size_t pw_len,
    const byte salt[SALT_LEN],
    byte key[KEY_LEN]
) {
    if (crypto_pwhash(
        key, KEY_LEN,
        reinterpret_cast<const char*>(pw), pw_len,
        salt,
        OPSLIMIT, MEMLIMIT,
        crypto_pwhash_ALG_ARGON2ID13) == 0)
    {
    return true;
    }
}
```

Figure 2.1. Key derivation from password function (validation and logging omitted).

2.2 Vault encryption

The serialized vault is encrypted using **XChaCha20-Poly1305-IETF AEAD**, with a random 24-byte nonce generated per encryption. Authentication failures (wrong key/corruption) are correctly handled. XChaCha20-Poly1305-IETF provides a large nonce space (192 bits), reducing risk of reuse, and AEAD ensures both confidentiality and integrity.

Validations of length (not 0 and not more than the maximum allowed) are implemented, as well as null pointer check. Any failure results in a log entry and is handled gracefully.

```
bool encrypt_vault_blob(
    const byte key[KEY_LEN],
    const byte* plaintext, size_t plen,
    byte** out_ct, size_t* out_ct_len,
    byte nonce[NONCE_LEN]
) {
    randombytes_buf(nonce, NONCE_LEN);
    if (crypto_aead_xchacha20poly1305_ietf_encrypt(
        *out_ct, out_ct_len,
        plaintext, plen,
        nullptr, 0, nullptr,
        nonce, key) == 0;)
    {
    return true;
    }
}
```

Figure 2.2. Vault data encryption function (validation and logging omitted).

3 Memory management

The password manager uses locked memory to handle sensitive data – SecureBuffer class handles passwords, and SecureKey class handles keys. Sodium_mlock prevents swapping secrets to disk, sodium_mprotect_* allows dynamic access control (read-only/no-access), and immediate zeroization on cleanup prevents residual data leaks.

```
class SecureBuffer {
public:
    explicit SecureBuffer(size_t size) {
        ptr_ = static_cast<unsigned char*>(sodium_malloc(size));
        sodium_mlock(ptr_, size);
        sodium_mprotect_readwrite(ptr_);
        sodium_memzero(ptr_, size);
    }
    ~SecureBuffer() {
        sodium_mprotect_readwrite(ptr_);
        sodium_memzero(ptr_, size_);
        sodium_munlock(ptr_, size_);
        sodium_free(ptr_);
    }
    unsigned char* data() { return ptr_; }
private:
    unsigned char* ptr_;
    size_t size_;
};
```

Figure 3.1. SecureBuffer class (validation, cleanup and error handling omitted).

SecureBuffer and SecureKey are very similar in structure, with the difference being SecureBuffer including <cstring> for memcpy (used when copying password input). Both employ size validation, automated cleanup with a destructor and throw error messages in case of an error.

4 Storage

4.1 Atomic file writes and permissions

Encrypted vault writes use a write-temp – fsync – rename pattern to prevent partial writes on crash. Files are set to owner-only access (0600) and permissions are validated for directories and files.

```
bool atomic_write_file(const std::string& path, const byte* buf, size_t len)
{
    std::string tmpl = path + ".tmpXXXXXX";
    int fd = mkostemp(temp.data(), O_CLOEXEC);
    fchmod(fd, S_IRUSR | S_IWUSR);
    write(fd, buf, len);
    fsync(fd);
    close(fd);
    rename(temp.data(), path.c_str());
}
```

Figure 4.1. Atomic write function (validation and logging omitted).

Validations of size (not 0 and not more than the maximum allowed) are implemented, file permissions checks and . Any failure results in a log entry or prints a short error message to the terminal (since log writing failure = no way of producing a log entry), and is handled gracefully.

4.2 Metadata

Salt and nonce are Base64-encoded for a layer of obfuscation and written into vault.meta. These are non-secret parameters necessary for KDF and decryption, and storing them separately aids rotation and recovery.

```
bool save_meta(const byte salt[SALT_LEN], const byte nonce[NONCE_LEN]) {
    if (!salt || !nonce) return false;
    std::string b64salt = to_base64(salt, SALT_LEN);
    std::string b64nonce = to_base64(nonce, NONCE_LEN);
    std::string content = b64salt + "\n" + b64nonce + "\n";

    if (!atomic_write_file(g_meta_filename,
        reinterpret_cast<const byte*>(content.data()),
        content.size())) {
        audit_log_level(LogLevel::ERROR,
            "save_meta: atomic write failed",
            "io_module",
            "failure");
        return false;
    }
    return true;
}
```

Figure 4.2. Metadata writing function (validation and logging included).

5 Input validation and sanitization

Generally, inputs for labels, usernames, passwords, vault names are length-bounded and reject control characters or all-whitespace values; log fields are sanitized to remove CR/LF to avoid multi-line injection. Previous and upcoming sections will mention specific validation controls used for each of the internal modules. These controls reduce injection, log forging, and parsing failures.

```
bool valid_label_or_username(const std::string& s) {
    if (s.empty() || s.size() > MAX_LABEL_LEN) return false;
    if (contains_control_or_tab_or_null(s)) return false;
    if (std::all_of(s.begin(), s.end(), { return std::isspace(c); })) return
false;
    return true;
}
```

Figure 5.1. Validation of credential entry labels and usernames: size must not be 0 or exceed maximum allowed, must not contain breaking symbols and must not be all spaces.

6 Authentication

On unlock, the application derives a key from the entered master password and salt, then attempts to decrypt the vault with the stored nonce. Success implies the master password was correct; failure implies wrong key or tampered ciphertext. No verifier (bcrypt/Argon2id string) is stored. In this implementation, the vault itself acts as the verifier because authenticated encryption guarantees integrity.

```
const unsigned MAX_ATTEMPTS = 5;
unsigned attempts = 0;
bool authenticated = false;
SecureBuffer master = get_password_secure("Master password: ");
if (!derive_key_from_password(master.data(), master.size(), salt,
key.data())) {
    attempts++;
}
Vault tmp;
if (load_vault_decrypted(key. data(), nonce, tmp)) {
    authenticated = true;
}
else {
    attempts++;
}
```

Figure 6.1. Master password vault unlock sequence (validation and logging omitted).

A brute-force guard of maximum attempt number is implemented. If at any point the number of attempts reaches it – the application logs the user out.

```
if (!authenticated) {
    std::cerr << "Too many failed attempts; exiting.\n";
    audit_log_level(LogLevel::ALERT,
        "Too many failed master password attempts - lockout",
        "load_vault",
        "failure");
    return cleanup_and_exit(3, salt, nonce);
}
```

Figure 6.2. Lockout sequence if the user reaches maximum number of attempts.

7 User interface

7.1 CLI

General cleartext messages are used to display the application menu and messages. A logic of clearing the terminal window from the action results before the latest attempt is implemented to not leak sensitive information. All password inputs are hidden, even when adding a credential to the vault. Username and notes are not treated as sensitive secrets, so their inputs are cleartext.

7.2 Timed clipboard

An option of copying the password of the credential from the vault (looked up by the label) is offered. A timer is set to clear the clipboard if it still contains the secret to reduce exposure window, and WSL systems users are notified in case their clipboard history is enabled.

```
void copy_with_timed_clear(const std::string& secret, unsigned seconds) {
    if (!clipboard_set(secret)) return;

    std::thread([secret, seconds]() {
        std::this_thread::sleep_for(std::chrono::seconds(seconds));
        std::string current;
        if (!clipboard_get(current)) return;
        if (current == secret) {
            clipboard_clear();
        }
        }).detach();
}
```

Figure 7.1. Copy with timed clear function (logging omitted).

8 Error handling and structured logging

The application uses a **structured audit log** with level, timestamp, userId, IP, sessionId, event, outcome, and sanitized message. The log file is set to owner-only permissions. Errors presented to users are generic and do not leak internals or secrets.

```
void audit_log_level(
    LogLevel lvl,
    const std::string& entry,
    const std::string& event,
    const std::string& outcome
)
{
    FILE* f = std::fopen(path, "a");
    // ...
    std::fprintf(
        f,
        "%s | %s | user=%s | ip=%s | session=%s | event=%s | outcome=%s |
%s\n",
        tbuf,
        log_level_str(lvl),
        g_log_ctx.userId.c_str(),
        g_log_ctx.ip.c_str(),
        g_log_ctx.sessionId.c_str(),
        s_event.c_str(),
        s_outcome.c_str(),
        s_entry.c_str()
    );
}
```

Figure 8.1. Logging function (most parts omitted).

9 Access control

Vault directories and files are placed under the user’s home (~/.passman/vaults/<name>) and must be owned by the effective user with no group/other permissions. Violations are logged and treated as errors. On Windows, checks are stubbed.

```
bool check_dir_ownership_and_perms(const std::string& path) {
    struct stat st;
    if (stat(path.c_str(), &st) != 0) return false;
    uid_t uid = geteuid();
    if (st.st_uid != uid) return false;
    mode_t perms = st.st_mode & 0777;
    if ((perms & 0077) != 0) return false;
    }
    return true;
}
```

Figure 9.1. Directory ownership check function (logging omitted).

10 Secure deletion

The application implements a secure deletion routine that overwrites file contents with zeros (up to a safe size), avoids deleting symlinks, flushes/syncs, and unlinks the file.

```
void secure_delete_file(const char* path) {
    FILE* f = fopen(path, "r+");
    // determine size, write zeros, fflush, fsync, fclose
    std::remove(path);
}
```

Figure 10.1. Secure deletion function (most parts omitted).

11 Conclusions and considerations

The application was developed with a cryptographic design in mind (Argon2id KDF, XChaCha20-Poly1305 AEAD), and many secure coding practices were inplemented (zeroization, input validation, atomic writes, structured logging, permission checks). However, there is much space left for improvement.

After a brief risk analysis, options for further development would be:

1. Considering store a master-password verifier (bcrypt or Argon2id string) while continuing to derive the AEAD key via Argon2id
2. Locking memory even for less sensitive secrets like added credential username and notes
3. Adding directory fsync() for atomicity
4. Researching and implementing more thorough secure deletion with considerations of modern SSDs

The list is not exshaustive because of lack of knowledge and experience in programming risk analysis of the developer team, but can be considered a start. However, the application may be considered functional and ready for use on local systems.

References

[1] JumpCloud, “What Is Argon2? Password Hashing Explained.” JumpCloud IT Index. [Online]. Available: https://jumpcloud.com/it-index/what-is-argon2. Accessed: Nov. 30, 2025.

[2] Libsodium, “XChaCha20-Poly1305.” Libsodium Guide. [Online]. Available: https://libsodium.net/guide/XChaCha20-Poly1305.html. Accessed: Nov. 30, 2025.

[3] Libsodium, “Password Hashing (Default PHF).” Libsodium Documentation. [Online]. Available: https://doc.libsodium.org/password_hashing/default_phf. Accessed: Nov. 30, 2025.