KVPStorage – Secure Key-Value Store

A lightweight encrypted key-value storage for desktop and embedded Linux systems

Author: Paulo Braga
Version: 1.0
Date: April 2025

Project developed using C++, Buildroot, QEMU and GoogleTest
Target Platform: ARM VersatilePB (QEMU Emulated)
Host Platform: Ubuntu 22.04.5 LTS

Table of Contents

1. Introduction

The KVPStorage project is a secure and lightweight key-value storage system designed for use in both desktop environments and embedded systems. It provides a simple interface for storing, retrieving, and deleting data, while ensuring confidentiality and integrity through cryptographic primitives.

The system is implemented in C++ and emphasizes modularity and clarity. It is structured as a reusable library that can be linked into multiple applications, such as a command-line interface or socket-based services.

The project is particularly useful in resource-constrained environments where data persistence and security are required, such as custom Linux distributions for IoT devices, QEMU-emulated targets, or embedded Linux boards.

This document covers the design of the system, its dependencies, usage instructions for desktop and embedded scenarios, and guidelines for extending or integrating the project into other systems.

---

2. Design Overview

The architecture follows a layered approach aiming separation of concerns and ease of maintenance. The system is organized into the following components:

2.1 Command Processor

At the core of the system lies the ***CommandProcessor***, which receives textual commands (e.g., ***SET key value***, ***GET key***, ***DELETE key***) and translates them into storage operations. It acts as a bridge between the user interface and the storage backend.

This module enables interface-agnostic integration. Whether the input comes from a terminal, a Unix domain socket, or another IPC mechanism, the processor interprets commands uniformly.

2.2 Storage Facade

The ***KVStorageFacade*** provides a simplified API to the underlying storage. It encapsulates a concrete implementation of the ***IKVStorage*** interface and serves as the entry point for ***set***, ***get***, and ***delete*** operations.

By isolating storage operations behind a facade, the system ensures that business logic and command interpretation remain decoupled from file management and cryptography.

2.3 File-Based Secure Storage

The default storage backend, **FileKVStorage**, persists key-value pairs to disk using a simple JSON format, which is then encrypted and signed using the libsodium library. Encryption ensures confidentiality, while HMAC ensures integrity.

The data is stored in a secure file, and the cryptographic keys are loaded from a configurable path using environment variables.

2.4 Utilities

A set of utility classes assist with key loading, path resolution, and secure file generation. These include:

- **KeyLoader**: Loads encryption and HMAC keys from the filesystem.
- **StoragePathResolver**: Determines the location of the secure storage file and key directory based on environment variables.

2.5 CLI Interface

The command-line interface (**CLI**) offers an interactive shell for users to interact with the storage system. It connects to the **CommandProcessor** and reads input from the user, providing immediate feedback for each command.

2.6 Build and Modularity

The system is structured as a reusable CMake-based project, with clear separation between:

- **libkvp/**: the core library,
- **apps/cli/**: the command-line application,
- **include/:** public headers,
- **tst/**: unit tests (not included in embedded builds),
- **third_party/**: external dependencies (**nlohmann/json**).

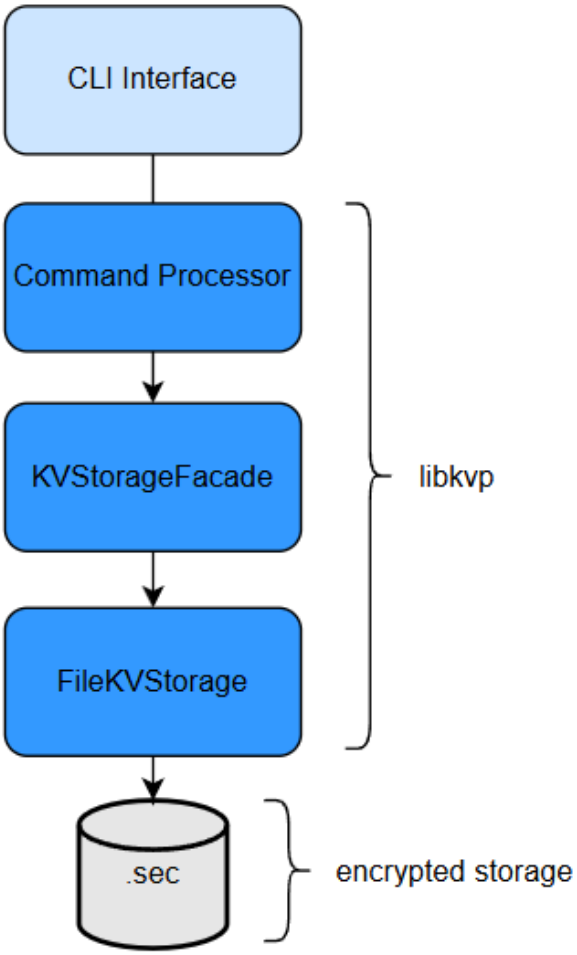

Figure 1 – Layered Architecture

---

3. Build and Runtime Dependencies

This chapter outlines the tools, system packages, and structure required to build and run the KVPStorage project on a Linux development host.

3.1 Build Dependencies (Host)

To build the project on a typical Linux desktop, the following packages must be installed:

- cmake (≥ 3.14)
  Build system generator.
- g++ or clang++
  A compiler with C++17 support.

- libsodium-dev
  Development headers for the libsodium cryptographic library.
- pkg-config
  Used by CMake to detect and configure libsodium.
- make
  Standard GNU build tool.

3.2 Runtime Dependencies

At runtime, only the **libsodium** shared library is required to decrypt and verify the secure storage file.

It is typically provided by a package named libsodium18, libsodium23, or similar, depending on the distribution.

3.3 Third-Party Dependencies

The project includes nlohmann/json as a **header-only** third-party dependency, already embedded in the source tree:

```
third_party/
└── nlohmann/
    └── json.hpp
```

No external installation or linking is needed for this library.

3.4 Project Layout

The project is organized into clearly separated directories for application logic, the library core, build scripts, and tests:

```
KVPStorage/
├── apps/cli/              # CLI application: entry point and CLI interface
├── buildroot/             # Buildroot integration (defconfig, package definition)
├── docs/                  # Project documentation and build quick references
├── include/kvp/           # Public headers for use by external applications
├── keys/                  # Pre-generated HMAC and encryption keys
├── libkvp/                # Core library: storage, command processor, cryptography
├── scripts/               # Host-side scripts for configuration and installation
├── third_party/           # Header-only external libraries (e.g., nlohmann/json)
├── tst/                   # Unit tests (excluded from embedded builds)
└── CMakeLists.txt         # Top-level CMake configuration
```

This layout ensures a clean separation of concerns:

- **‘libkvp/’** + ‘**include/’** form a reusable library.
- **apps/cli/** links against it as a standalone binary.

- **tst/** contains GTest-based unit tests.
- **scripts/** offers utilities to simplify usage and installation.
- **buildroot/** packages everything for embedded system deployment.

---

4. Building and Running on Desktop (Host Linux)

This chapter explains how to build and run the KVPStorage project locally on a Linux desktop. It covers environment preparation, build steps, and how to use the CLI to interact with the secure key-value storage. All development and testing was performed on **Ubuntu 22.04.5 LTS**. The steps below are expected to work on other Debian-based systems as well.

4.1 Setup and Environment Configuration

Before compiling the project, it's recommended to run the provided helper script:

```
$ ./scripts/configure.sh
```

This script checks for required packages and configures two environment variables used at runtime:

- **KVP_STORAGE_PATH**: Location of the encrypted key-value store file.
- **KVP_KEYS_PATH**: Location of the encryption and authentication keys.

If not set manually, these variables will default to:

```
KVP_STORAGE_PATH -> $HOME/.config/kvpstorage/kvpstorage.secure
KVP_KEYS_PATH    -> $HOME/.config/kvpstorage/keys
```

To override the defaults, export your own values:

```
$ export KVP_STORAGE_PATH="/custom/path/data.secure"
$ export KVP_KEYS_PATH="/custom/path/keys"
```

4.2 Compiling with CMake

To build the project using CMake:

```
$ mkdir -p build
$ cd build
$ cmake ..
$ make
```

After successful compilation:

- The CLI binary will be located at: build/apps/cli/kvpstorage_cli
- The test suite (optional) will be at: build/tst/unit_tests

4.3 Running the Application

Run the CLI by executing:

```
$ ./apps/cli/kvpstorage_cli
```

You can now use commands such as:

```
> SET DeviceName TestDevice_123
OK
> GET DeviceName
TestDevice_123
> DELETE DeviceName
OK
> GET DeviceName
>
```

On first execution, the secure file will be created automatically (if it doesn't already exist).

4.4 Running Tests (Optional)

Unit tests are compiled and executed automatically when you build the project. To run the test suite manually:

```
$ ctest –output-on-failure
```

This will execute all GoogleTest-based tests located in tst/.

---

5. Building for Embedded Target (QEMU - ARM VersatilePB)

This chapter describes how to build and run the kvpstorage application in an embedded environment using Buildroot and QEMU.

5.1 Overview of the Embedded Build

The embedded target is based on:

- **Architecture**: ARM (arm926ej-s)

- **Platform**: VersatilePB (emulated via QEMU)
- **Build system**: Buildroot

All components, including the Linux kernel, root filesystem, and kvpstorage app, are compiled using Buildroot.

5.2 Enabling the kvpstorage Package

To make the kvpstorage package available inside Buildroot, you can run the provided helper script:

```
$ ./scripts/install_kvp_storage.sh <path-to-buildroot>
```

This script automates the following steps:

- **Copies the package**: Places the kvpstorage directory into the package/ folder of your Buildroot tree.
- **Adds the defconfig**: Installs the custom kvp_defconfig into the configs/ directory.
- **Modifies the menu (packages/Config.in)**: Inserts the line

  ```
  source "package/kvpstorage/Config.in"
  ```

  under the "menu "Miscellaneous" section of package/Config.in, unless already present.

After running the script, you can apply the configuration with:

```
$ cd path/to/buildroot
$ make kvp_defconfig
```

Important notes:

- You may also run '**make menuconfig**' to verify that **kvpstorage** is enabled under '**Target packages > Miscellaneous**'.
- This setup assumes a **clean and unmodified Buildroot environment**. If you're integrating into a customized or existing setup, manual adjustments may be necessary.

5.3 Building the Image

To build the entire system (kernel, rootfs, and application):

```
$ make
```

This process will take a few minutes and generate the output files under output/images/.

5.4 Running in QEMU

To boot the image using QEMU:

```
qemu-system-arm \
  -M versatilepb \
  -kernel output/images/zImage \
  -dtb output/images/versatile-pb.dtb \
  -drive file=output/images/rootfs.ext2,format=raw,if=sd \
  -append "root=/dev/mmcblk0 console=ttyAMA0" \
  -nographic
```

If you don’t have qemu installed yet, you can install it with the following commands:

```
$ sudo apt update
$ sudo apt install qemu-system-arm
```

Once booted, you'll get a shell prompt. The login password is ‘root’:

```
Welcome to Buildroot
buildroot login: root
#
```

5.5 Automatic Initialization

The image includes a script (/etc/kvpstorage/init_kvpstorage.sh) to start the kvpstorage application.

This script:

- Sets the environment variables:

```
export KVP_STORAGE_PATH="/var/lib/kvpstorage/kvpstorage.secure"
export KVP_KEYS_PATH="/etc/kvpstorage/keys"
```

- Starts the application:

```
exec /usr/bin/kvpstorage
```

The script is deployed to the rootfs during the Buildroot install step with execution permission. To run the script and start kvpstorage application:

```
# /etc/kvpstorage/init_kvpstorage.sh
```

6. Extending the Project

The project was designed with modularity in mind. New interfaces, backends, or external integrations can be added with minimal changes to existing code.

6.1 Adding New Interfaces (e.g., IPC, REST)

The core logic is decoupled from the user interface. This means the CommandProcessor class can be reused by any frontend that needs to process text-based commands.

To add a new interface:

1. Create a new component in apps/ (e.g., apps/ipc/, apps/rest/).
2. In this new component:
    - Instantiate KVStorageFacade.
    - Instantiate CommandProcessor.
    - Read input (from socket, HTTP request, etc.).
    - Pass input lines to CommandProcessor::handle_command(...).
    - Send output back to client.

Example: IPC (Unix Domain Socket)

A Unix socket interface can reuse the same logic as the CLI:

```
CommandProcessor processor(facade);
std::string response = processor.handle_command("SET foo bar");
```

This design ensures no logic duplication.

6.2 Integrating with Other Applications

External applications can integrate by directly linking against the libkvp static library and using the public API:

```
#include <kvp/storage/KVStorageFacade.hpp>

std::unique_ptr<IKVStorage> storage = std::make_unique<FileKVStorage>(file_path);
KVStorageFacade facade(std::move(storage));

facade.set("username", "admin");
auto value = facade.get("username");
```

All business logic and storage encryption are encapsulated inside libkvp. No need to reimplement parsing, encryption, or file handling.

6.3 Creating Alternative Storage Backends

To support other forms of storage (e.g., database, in-memory, remote), implement the IKVStorage interface:

```
class MyCustomStorage : public IKVStorage {
public:
    void set(const std::string& key, const std::string& value) override;
    std::optional<std::string> get(const std::string& key) override;
    void del(const std::string& key) override;
};
```

Then use it transparently with the KVStorageFacade.