

# RockyGuard Library Documentation

Version 1.2.1

RockyGuard - C++ License Checking Library

This document provides comprehensive documentation for the RockyGuard library,  
a C++17 library for node-locked and floating license management.

# RockyGuard Library - Documentation

Version 1.2.1-dev

Copyright (c) 2025-2026 Rocky Software Inc. All rights reserved.

## About this manual:

This document describes RockyGuard v1.2.1. The "-dev" suffix above marks an in-flight build during the v1.2.1 development cycle; the suffix is dropped at the release cut and a customer reading the shipped v1.2.1 manual sees just "Version 1.2.1". If you have a manual whose cover reads "1.2.1-dev", you are reading pre-release material that may still change before the v1.2.1 tag.

## Version applicability (READ THIS IF YOU ARE NOT SURE WHICH RELEASE YOU HAVE):

This manual documents the v1.2.1 surface of the library. If your installed library is v1.2.0, several of the APIs and CLI flags described here DO NOT EXIST in your version -- attempting to call them will fail to compile or fail at the CLI argument parser. Specifically, every passage marked "(v1.2.1+)" inline is v1.2.1-and-later only.

To check the version you actually have linked, read the `ROCKYGUARD_VERSION_STRING` macro from `<rockyguard/version.h>` in your application, or run any shipped CLI tool with `--help` -- the banner that prints carries the version. The library version, the CLI tool versions, and the manual you are reading should all match.

If your library is older than this manual (e.g., you installed v1.2.0 but landed on the v1.2.1 manual via a search result), the right manual to consult is the one that came inside your customer zip under `docs/Customer_Documentation.pdf` -- that copy was packaged at the same time as the binary you have. Mixing v1.2.1 docs with a v1.2.0 binary will lead to "function does not exist" or "unrecognized argument" errors that are easy to misread as bugs. The website ([rockyguard.dev](https://rockyguard.dev)) only publishes the latest release.

v1.2.1+ also renames the floating-license CLI binaries from `floating_server_example` / `floating_client_example` to `rg_floating_server` / `rg_floating_client`; references throughout this manual use the new names.

The v1.2.1+ markers convention is repeated below for completeness.

## Reading conventions:

- Where a feature is new since v1.2.0, the manual marks it explicitly with "(v1.2.1+)" inline. Examples: the per-machine floating-license cap (§5), Debug-CRT linking variant (§3.1), floating-server log rotation (§5.2), `--metadata` / `--show-count` flags on `license_create` (§7.3), the `LicenseVerifier::check_version()` API (Customer\_API\_Reference §6.7), `SignatureAlgorithm::AutoDetect`.
- Everything NOT marked "(v1.2.1+)" describes behavior present since v1.2.0. A reader fresh to v1.2.1 can ignore the markers and read the manual as a unified description of the current

library. A reader upgrading from v1.2.0 can scan only the marked sections to see what is new without re-reading the rest.

- "Future\_Roadmap §X" cross-references point at the companion docs/RockyGuard\_Future\_Roadmap.txt document (deferred items for v1.3 and beyond, not in this release).
- "Customer\_API\_Reference §X.Y" cross-references point at the companion docs/Customer\_API\_Reference.txt / .pdf -- the C++ API reference shipped alongside this manual.

## TABLE OF CONTENTS

1. Introduction
2. Getting Started
3. Library Integration
4. End-User License Management
5. Floating Licensing (Premium Tier)
6. Hardware Fingerprinting
7. CLI Tools Reference
8. License File Format
9. Anti-Tampering Features
10. Troubleshooting 11. Glossary 12. Production Checklist

# 1. INTRODUCTION

---

RockyGuard is a C++17 license checking library that lets you add license management to your software. It provides two licensing models for your end users:

- Node-Locked Licensing: Ties a license to a specific machine via hardware fingerprinting (MAC address, CPU ID, disk serial, motherboard ID). The license can only be used on the machine it was generated for.
- Floating Licensing (Premium tier): A central license server manages a pool of licenses. End-user applications check out a license when they start and return it when they stop.

Key concepts:

Three actors. Three meanings of the word "license". Get this upfront and the rest of the manual reads cleanly; the full glossary lives at §11 for lookup.

The actors:

YOU (the manual reader; the "library customer"). A software vendor who has bought or is evalu...

YOUR END USER. The person whose machine runs your application. They never see RockyGuard dire...

ROCKY SOFTWARE INC. The library vendor (us). We sign your vendor license. After delivery we a...

The three "licenses":

Vendor license (vendor\_license.json) -- what YOU receive from Rocky Software Inc. Authorizes ...

End-user license (license.json) -- what YOU issue to YOUR end user. Carries their machine's f...

Floating license (a seat in a pool) -- a Premium-tier model in which one end-user license has...

Whenever this manual says "the license" without a qualifier, the qualifier is usually clear from context (the §4 chapter is about end-user licenses; §2.2-§2.3 are about the vendor license; §5 is about the floating-license model). When ambiguity could matter, the docs spell out which one explicitly.

Key capabilities:

- Ed25519 and RSA-SHA256 digital signature algorithms
- Tamper-proof license files (signed JSON format)
- Cross-platform hardware fingerprinting (Windows, Linux)
- Virtual network adapter filtering (VMware, Docker, VPN, etc.)
- Fuzzy hardware matching with configurable threshold
- License expiry with configurable grace periods
- Per-feature license gating
- Floating license server with heartbeat and automatic lease expiry
- Clock manipulation detection
- Binary integrity self-check (shared library builds)
- Security hardening (ASLR, DEP, Control Flow Guard)

## Dependencies:

- OpenSSL 3.x (linked automatically)

## Supported platforms (v1.2.0):

- Windows 10 / 11 (x64), Windows Server 2019+
- Linux x64 on glibc 2.34 or newer. The shipped Linux binary statically links libstdc++, libgcc, and OpenSSL, so glibc is the only runtime ABI dependency. It runs on Ubuntu 22.04 LTS (glibc 2.35), Ubuntu 24.04 LTS (glibc 2.39), Debian 12 "Bookworm" (glibc 2.36), RHEL 9 / CentOS Stream 9 / Rocky Linux 9 (glibc 2.34), Amazon Linux 2023 (glibc 2.34), and any other x86\_64 Linux distribution with glibc 2.34 or newer. It does NOT run on Debian 11, Ubuntu 20.04, RHEL 8, or Amazon Linux 2 (all have glibc 2.31 or older). A glibc-2.28 build extending coverage to those older distributions is planned for a subsequent release.
- macOS (x64 and Apple Silicon): available upon request -- the library's macOS support code (IOKit hardware fingerprint, Mach-O integrity check) is present but a shipping macOS build is not included in v1.2.x. Contact the vendor if you need a macOS build ahead of the v1.3 release target.

To confirm your Linux distribution's glibc version, run:

```
ldd --version | head -1
```

## 2. GETTING STARTED

### 2.1 Package Contents

The Windows customer zip (rockyguard-v1.2.1-windows-x64-customer.zip) extracts to a folder of the same name and contains:

Path	File	Description
include/rockyguard/		Public headers (verification only).
	rockyguard.h	Single-include convenience header.
	version.h	Library version macros (auto-generated).
	types.h	Enums and result types.
	license.h	License data model.
	license_verifier.h	License verification.
	hardware_fingerprint.h	Machine hardware fingerprinting.
	floating_client.h	Floating license client (Premium).
	export.h	DLL export macros.
lib/static/	rockyguard.lib	Static library, /MD (Release-CRT) -- link from a Release consumer build.
	rockyguard_mdd.lib	Static library, /MDd (Debug-CRT) -- link from a Debug consumer build (v1.2.1+). See §3.1 for \$<CONFIG>-driven selection.
lib/shared/	rockyguard.dll	Shared library (DLL).
	rockyguard.lib	Import library for the DLL.
	rockyguard.sig	Integrity signature (ship with DLL).
	libcrypto-3-x64.dll	OpenSSL runtime.
	libssl-3-x64.dll	OpenSSL TLS runtime.
tools/	license_keygen.exe	Generate your Ed25519 or RSA keypair.
	license_create.exe	Create end-user licenses.
	license_verify.exe	Verify and inspect license files.
	rg_fingerprint.exe	Print machine hardware fingerprint.
	rg_floating_server.exe	Floating license server (Premium).
	floating_server_config.yaml	Sample server config file.
	rg_floating_client.exe	Floating license client smoke-test binary (Premium).
deps/	include/openssl/	Bundled OpenSSL headers.
	lib/libcrypto.lib	OpenSSL crypto import library (needed when static-linking against rockyguard.lib; see §3.1).
	lib/libssl.lib	OpenSSL TLS import library (needed when static-linking against rockyguard.lib; see §3.1).
examples/	CMakeLists.txt	Build script for the examples.
	node_locked_example.cpp	Node-locked license verification example.
	rg_floating_client.cpp	Floating license client example source.
	banner.h	Shared header that both .cpp examples #include for their startup banner; copied into the package so the examples build standalone.
docs/		This documentation and the API reference (PDF + .txt).
(zip root)	AI_INTEGRATION_GUIDE.md	AI-agent integration recipe. Hand this file to Claude Code, Cursor, Copilot, etc. to have them perform the integration end-to-end (see §2.6).

Path	File	Description
(zip root)	README.txt	One-page orientation: top-level package contents, quick start, support contact. Read this first if you're new.

Per-platform differences (Linux):

The Linux customer zip (rockyguard-v1.2.1-linux-x64-customer.zip) is identical in structure to the Windows zip above with the following file-level substitutions:

Path	Linux file	Notes
lib/shared/	librockyguard.so	In place of rockyguard.dll.
	librockyguard.sig	In place of rockyguard.sig. Same format; the name matches the .so.
	(no libcrypto / libssl files)	OpenSSL is statically linked into librockyguard.so on Linux, so there is no separate runtime dependency.
lib/static/	librockyguard.a	In place of rockyguard.lib.
tools/	(binaries with no .exe suffix)	Same tools as Windows: license_keygen, license_create, license_verify, rg_fingerprint, rg_floating_server, rg_floating_client.
deps/lib/	libcrypto.a	In place of libcrypto.lib. Needed when static-linking librockyguard.a.
	libssl.a	In place of libssl.lib. Needed when static-linking librockyguard.a.

No .lib import library is needed on Linux; the shared library is linked directly via -lrockyguard once LD\_LIBRARY\_PATH or RUNPATH points at lib/shared/. macOS customers: available upon request (see §1, Supported platforms).

## 2.2 How You Receive Your Vendor License

The vendor license is the file that authorizes your use of RockyGuard for issuing end-user licenses. You receive it from the library vendor (Rocky Software Inc.) once your purchase or trial agreement is in place; it is not bundled inside the customer zip you downloaded, and the library will not generate end-user licenses without it.

How to request it:

Use the contact form at <https://rockyguard.dev/contact> (preferred -- the form routes to the same inbox as the email below and pre-fills a few fields), or email [hello@rockyguard.dev](mailto:hello@rockyguard.dev) directly. Include the legal name of the licensee organisation, the product name the license is for, and the desired tier (Basic or Premium); see §2.3 for tier details.

Delivery format:

The vendor license arrives as an email attachment: a single signed JSON file, typically named after your organisation, e.g. rocky\_software\_license.json or acme\_inc\_license.json. Save it to a stable location on your build machine alongside your existing private signing key. Do NOT commit it to a public repository; treat it the same way you treat your private key.

Expected receipt time:

One business day from a complete request. If your request is received outside business hours (Eastern Time), expect delivery the next business day. The contact form's auto-reply confirms receipt; the actual license arrives in a follow-up

email.

If your license is missing or has not arrived:

Email [hello@rockyguard.dev](mailto:hello@rockyguard.dev) with your original request thread (subject line, sender, or approximate date is enough to locate the original request on our side) and we will resend or investigate. Same SLA as the initial request: one business day. If `license_create` or `rg_floating_server` reports that the vendor license file cannot be parsed when you point them at it (e.g., a "Vendor license missing" or "MalformedFile" error), do not modify the file -- ask for a fresh copy.

What the vendor license contains and how the library uses it:

The vendor license is a signed JSON document carrying your organisation name, license tier, generation-limit counter (Basic: unlimited; Premium: configurable), product binding, and a signature from the library vendor. `license_create` (called with `--vendor-license`) and `rg_floating_server` (`vendor_license`: in its YAML config) consume it at startup: the file's signature is verified against the vendor public key embedded in the library, the tier and limits are loaded into the running tool, and the matching code paths are unlocked. Every subsequent license-generation call then cross-checks the vendor license to enforce the tier and counter limits. `license_keygen` does NOT consume the vendor license -- it is a standalone tool that generates a fresh Ed25519 or RSA keypair on its own, run as the very first step of the workflow before you have anything from Rocky Software Inc. (see §2.4 Quick Start, step 1). The verifier-side library used by your end-user application does NOT need the vendor license either -- see §2.3.

## 2.3 Your Vendor License

You should have received a `vendor_license.json` file from the library vendor. This file authorizes your use of the library for generating end-user licenses.

### IMPORTANT:

- The vendor license is needed ONLY in your license generation tools and floating license server.
- Your end-user application does NOT need the vendor license file. It only verifies licenses using `LicenseVerifier` and `FloatingLicenseClient`.

Your license tier:

- Basic: Node-locked licensing, unlimited license generation
- Premium: Node-locked + floating licensing, configurable generation limit

## 2.4 Quick Start

If you would rather have an AI agent (Claude Code, Cursor, GitHub Copilot, ChatGPT, Gemini, Cody, or similar) perform the integration for you end-to-end instead of walking through the steps below by hand, see §2.6 "Integrating with an AI assistant" and skip to it now. The two paths produce equivalent results; choose whichever matches how you usually work.

Step 1: Generate your keypair (run once):



```
tools/license_keygen --private private.pem --public public.pem
```

Keep private.pem SECRET. You will embed public.pem in your application.

Step 2: Create an end-user license. Requires the vendor\_license.json your library vendor provided (see sections 2.2 and 2.3). Replace "<end user's fingerprint>" with the hash your end user obtained by running rg\_fingerprint on THEIR machine (see §4.2 for how to ask for it):

```
tools/license_create --key private.pem \  
  --vendor-license vendor_license.json \  
  --id "LIC-001" \  
  --licensee "Customer Name" \  
  --product "YourApp" \  
  --expires "2027-12-31T23:59:59Z" \  
  --feature "pro_feature" \  
  --fingerprint-value "<end user's fingerprint>"
```

Quick self-test alternative: use --fingerprint (no value) instead of --fingerprint-value to bind the license to YOUR OWN build machine. This is convenient for verifying the toolchain end-to-end without a real end user, but the resulting license will not run on any other machine. Do not ship a self-fingerprinted license to an actual customer.

Step 3: In your end-user application (no vendor license needed):

```
#include <rockyguard/rockyguard.h>  
  
static constexpr char PUBLIC_KEY[] = R"(-----BEGIN PUBLIC KEY-----  
MCowBQYDK2VwAyEA...your public key here...  
-----END PUBLIC KEY-----)";  
  
int main() {  
    rockyguard::LicenseVerifier verifier(PUBLIC_KEY);  
  
    auto result = verifier.load("license.json");  
    if (!result) {  
        std::cerr << result.message << std::endl;  
        return 1;  
    }  
  
    result = verifier.check_node_locked();  
    if (!result) {  
        std::cerr << result.message << std::endl;  
        return 1;  
    }  
  
    // Optional: enforce version_range at runtime (v1.2.1+).  
    // Skip if you don't issue licenses with --version, or if you  
    // want the field to remain informational only. See §3.3 point B  
    // for the uncommented version and Customer_API_Reference §6.7  
    // for the full syntax.  
    // result = verifier.check_version("3.1.0");  
    // if (!result) return 1;  
  
    // Application runs with valid license  
    return 0;  
}
```

```
}
```

## 2.5 Building and Running the Examples

The `examples/` folder contains working example source code and a `CMakeLists.txt` to build them. Use these to learn how the library works before integrating into your own application.

To build the examples (OpenSSL is bundled, no separate install needed):

```
cd examples
cmake -B build
cmake --build build --config Release
```

The `CMakeLists.txt` automatically finds the library in the parent directory (the package root that contains `include/`, `lib/`, `deps/`, etc. after you unzipped the customer zip). To point to a different location:

```
cmake -B build -DROCKYGUARD_DIR=/path/to/extracted/package
```

Available examples:

`node_locked_example`: Demonstrates license loading, signature verification, hardware fingerprint checking, grace period handling, and feature gating. Run as:

```
node_locked_example <public_key.pem> <license.json>
```

`rg_floating_client`: Demonstrates floating license checkout, heartbeat, and checkin. Premium tier only -- requires a running floating server (see §5). Run as:

```
rg_floating_client [server_host] [port]
```

## 2.6 Integrating with an AI Assistant

The customer zip ships an AI-agent-readable integration recipe at the package root: `AI_INTEGRATION_GUIDE.md`. The same file is also published online at [https://rockyguard.dev/AI\\_INTEGRATION\\_GUIDE.md](https://rockyguard.dev/AI_INTEGRATION_GUIDE.md) so an agent can fetch it by URL. It is a parallel path to the human-walked Quick Start (§2.4) and the build-and-run examples (§2.5); the end state is the same -- a working RockyGuard integration in your application -- produced by a different actor.

What it is:

A how-to written in AI-imperative voice ("you are an AI assistant... do X, ask the user Y") and structured for agentic execution: 11 numbered sections covering the full integration (CMake wiring, public-key embedding, startup verification, feature gating, optional floating-license setup), explicit input-collection prompts the agent must ask the user before touching code, and explicit "stop and ask the user" gates for situations the agent should not unilaterally resolve. The same content is readable to a human; the imperative voice exists so an AI agent executes it deterministically.

How to use it:

Open your AI agent of choice (Claude Code, Cursor, GitHub Copilot, ChatGPT, Gemini, Cody, etc.) and give it one of

these prompts. After extracting the customer zip you will have a folder named after the zip itself -- e.g. rockyguard-v1.2.1-windows-x64-customer/ on Windows or rockyguard-v1.2.1-linux-x64-customer/ on Linux (the version and platform parts match whichever zip you downloaded). That extracted folder is the package root the AI agent should be pointed at; you do not need to rename it, move it into a vendor/ or third\_party/ subdirectory, or otherwise reorganize it before running the prompts below.

- If you have already extracted the zip (recommended; the agent reads the local copy of the guide directly):

```
Please integrate the RockyGuard license library into my project.
The package is at rockyguard-v1.2.1-windows-x64-customer/.
Read rockyguard-v1.2.1-windows-x64-customer/AI_INTEGRATION_GUIDE.md
and follow it.
```

Replace the folder name with whatever your extracted zip actually produced. If you extracted ...

- If you have not yet downloaded the zip, or prefer the agent to fetch the guide directly from the website:

```
Please integrate the RockyGuard license library into my project.
Fetch https://rockyguard.dev/AI_INTEGRATION_GUIDE.md and follow it.
```

The agent will then ask you for the location of the extracted package (or instruct you on dow...

The agent will then ask you, in order, for the four inputs the guide requires before it modifies any code:

- Your public key (PEM string, from public.pem produced by license\_keygen).
- Which binary in your project should carry the license check (if you have more than one).
- Where license.json should be loaded from at runtime (default: next to the executable). This is the license file your application will read on the end-user's computer to verify the user is licensed; it is not the vendor license and it does not stay on your build machine.
- Your tier (Basic = node-locked only; Premium = node-locked + floating) and which features in your application should be paid-tier gated.

Once those are answered, the agent edits your CMakeLists.txt to link the library, embeds your public key as a string constant in the source file containing main(), inserts the verification block at the top of main(), and adds check\_feature() calls for each feature you named. It then runs your normal build to confirm nothing breaks.

What the agent will and will NOT do automatically:

The guide instructs the agent to STOP and ask you, rather than guess, in any of the following situations:

- It cannot unambiguously locate int main(...).
- Your project has multiple binaries and it is unclear which one to protect.
- Your build system is not vanilla CMake (Bazel, Meson, MSBuild, plain Make).
- You already have a different license-checking system in place (Sentinel, FlexLM, custom) -- whether to replace it or run side-by-side is your call.
- public.pem is not in the workspace and you have not pasted its contents.

- You have not told the agent which features should be paid-tier gated.

The guide also enforces a set of hard-no rules the agent will refuse to violate: never commit private.pem to git, never ship private.pem, never catch and ignore license verification errors, never embed license.json into the binary, never generate a fake public key, never silently switch your build system. These mirror the security and operational rules in §3, §9, and §12 of this manual.

Why we ship this:

C++ integration tasks -- adding a third-party library to a CMakeLists.txt, embedding a key, wiring a verification call into main(), defining feature gates -- are increasingly handed to AI agents. Most license-management libraries ship documentation that assumes a human reads a PDF and translates it into code by hand: a half-day to one-day onboarding even for an experienced engineer. AI\_INTEGRATION\_GUIDE.md exists to compress that to a single prompt by giving the agent a deterministic, opinionated recipe written for the way agents actually work (asking the user for ambiguities up front, refusing to guess, naming the failure modes that produce silent runtime bugs). It is a deliberate design choice, not a marketing layer over the human-readable docs; the same recipe produces a clean integration whether followed by a human or an agent.

Cross-references:

- The full guide is in your customer zip at AI\_INTEGRATION\_GUIDE.md and at [https://rockyguard.dev/AI\\_INTEGRATION\\_GUIDE.md](https://rockyguard.dev/AI_INTEGRATION_GUIDE.md).
- The human-walked equivalent is §2.4 (Quick Start) plus §3 (Library Integration).
- For feature gating (the half of the integration the agent will ask you to define), see §4.6 of this manual.
- For the floating-license additions the agent applies on Premium tier when explicitly requested, see §5.

## 3. LIBRARY INTEGRATION

REQUIREMENT: C++17. The library's public headers use `std::optional`, `std::variant`, `std::string_view`, structured bindings, and inline variables, all C++17 features. Your consumer project must be built with a C++17 (or later) standard.

In CMake:

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

With a direct compiler flag: `/std:c++17` (MSVC) or `-std=c++17` (gcc / clang). If this is missed, the first rockyguard header you include fires a `static_assert` with a message pointing at the exact fix -- no template error cascade.

### 3.1 Linking (Static - Recommended)

The static `rockyguard.lib` pulls in both `libcrypto` (license signing, hashing, HMAC, ed25519/RSA) AND `libssl` (TLS for the online time-anchor verification and the optional floating-license HTTPS). Your consumer must link BOTH, plus the Windows system libs those transitively need.

Static builds do not need a `.sig` file. The integrity-check signature shipped as `rockyguard.sig` in `lib/shared/` is paired with the rockyguard DLL only -- a static link compiles the library code directly into your executable, so there is no separate binary for the integrity check to verify against. §9.3 has the full picture of what the `.sig` file does and why static builds skip it.

CRT VARIANTS on Windows: the customer zip ships TWO static library variants so you can build your own executable in either Release or Debug without hitting LNK2038 / LNK1319 errors about mismatched `_ITERATOR_DEBUG_LEVEL` or `RuntimeLibrary`:

```
rockyguard.lib      built with /MD -- link from a Release
                    (/MD) consumer.
rockyguard_mdd.lib  built with /MDd -- link from a Debug
                    (/MDd) consumer.
```

The bundled OpenSSL import libs (`libssl.lib`, `libcrypto.lib`) are CRT-agnostic: `vcpkg's` Release and Debug import stubs are byte- identical, and OpenSSL's API does not pass C-runtime objects across the DLL boundary, so a single pair serves both consumer variants.

If your CMake selects per-config, the natural pattern is to feed the right variant via a generator expression:

```
target_link_libraries(your_app PRIVATE
  $<IF:$<CONFIG:Debug>,
    path/to/lib/static/rockyguard_mdd.lib,
    path/to/lib/static/rockyguard.lib>
  path/to/deps/lib/libssl.lib
  path/to/deps/lib/libcrypto.lib
  ws2_32 crypt32 iphlapi ole32 oleaut32 wbemuuid)
```

This restriction does not apply on Linux: glibc / libstdc++ ABI does not split per-config, so a single librockyguard.a serves both Release and Debug consumers.

In your CMakeLists.txt (Windows, Release-only example):

```
# Point to the library's include and lib directories. The
# bundled deps/ directory contains the exact OpenSSL build we
# linked against -- use it for the most predictable result.
target_include_directories(your_app PRIVATE
    path/to/include
    path/to/deps/include)
target_link_libraries(your_app PRIVATE
    path/to/lib/static/rockyguard.lib
    path/to/deps/lib/libssl.lib
    path/to/deps/lib/libcrypto.lib
    ws2_32 crypt32 iphlapi ole32 oleaut32 wbemuuid)
```

On Linux, the static path is structurally the same as Windows: the shipped librockyguard.a contains the RockyGuard code only, and you link the bundled OpenSSL archives from deps/lib/ (both libssl.a and libcrypto.a are needed; libssl supplies the TLS symbols used by the online time-anchor verification, libcrypto supplies the hashing / signature / HMAC symbols). pthread + dl are required by OpenSSL on Linux:

```
target_include_directories(your_app PRIVATE
    path/to/include
    path/to/deps/include)
target_link_libraries(your_app PRIVATE
    path/to/lib/static/librockyguard.a
    path/to/deps/lib/libssl.a
    path/to/deps/lib/libcrypto.a
    pthread dl)
```

Or if using the library as a CMake subdirectory (both OSes), the library's own CMake target exports the transitive dependencies so you do not need to list them yourself:

```
add_subdirectory(path/to/rockyguard)
target_link_libraries(your_app PRIVATE rockyguard)
```

Static builds do NOT require rockyguard.sig at runtime. The integrity signature is only relevant to shared (DLL / .so) builds, where the library image is a separate file that can be tampered with after install. When the library is statically linked into your\_app.exe, it is part of the executable image and any tamper invalidates the executable's own signature.

## 3.2 Linking (Shared / DLL)

Windows:

```
# Define ROCKYGUARD_SHARED_LIB before including headers so the
# public symbols are marked __declspec(dllexport).
target_compile_definitions(your_app PRIVATE ROCKYGUARD_SHARED_LIB)
target_include_directories(your_app PRIVATE path/to/include)
target_link_libraries(your_app PRIVATE
    path/to/lib/shared/rockyguard.lib)
```

At build time you only link `rockyguard.lib` (an import library that routes calls to `rockyguard.dll` at load time). You do NOT have to link any OpenSSL `.lib` files yourself; `rockyguard.dll` already imports the OpenSSL symbols it needs.

At runtime, `rockyguard.dll` loads two OpenSSL DLLs from the same directory: `libcrypto-3-x64.dll` and `libssl-3-x64.dll`. These are vcpkg-built dynamic-linkage OpenSSL binaries; `rockyguard.dll` does NOT statically embed them. Without them in your application's load path, `rockyguard.dll` fails to load with an OS-level "missing dependency" error before any of your code runs.

Ship with your application:

```
- rockyguard.dll
- rockyguard.sig          (integrity signature - must be next to the DLL)
- libcrypto-3-x64.dll     (OpenSSL crypto runtime)
- libssl-3-x64.dll        (OpenSSL TLS runtime)
```

Place all four files in the same directory as `your_app.exe` (or anywhere on the OS DLL search path). The Linux shared-build behaves differently -- see the Linux subsection below -- so do not assume the Windows shipping list applies to the `.so`.

Linux:

```
target_compile_definitions(your_app PRIVATE ROCKYGUARD_SHARED_LIB)
target_include_directories(your_app PRIVATE path/to/include)
target_link_libraries(your_app PRIVATE
    path/to/lib/shared/librockyguard.so)

# Either set RPATH at build time so your_app finds the .so at
# runtime without LD_LIBRARY_PATH:
set_target_properties(your_app PROPERTIES
    INSTALL_RPATH "$ORIGIN/./lib/shared")
# ...or ship your_app and librockyguard.so in the same directory
# and set RPATH=$ORIGIN.
```

Ship with your application:

```
- librockyguard.so
- librockyguard.sig      (integrity signature - must be next to the .so)
```

OpenSSL is statically linked into `librockyguard.so` on Linux, so there is no separate `libcrypto` / `libssl` runtime to ship -- the Linux `.so` is self-contained for OpenSSL. (The Windows shared-build packages OpenSSL as separate DLLs because that is vcpkg's standard x64-windows convention; the Linux build configures OpenSSL for static archiving instead. Either choice is equivalent at the call-site level; the difference is only in which files ship next to `your_app`.)

### 3.3 Two Separate Integration Points

Your software will have TWO places where the library is used:

A) YOUR LICENSE GENERATION WORKFLOW (vendor license required):

```
This is the workflow you run when issuing a new end-user license.
The package ships license_create.exe (Windows) / license_create
```

(Linux) for this; you do NOT need to write or compile any C++ for license generation. Programmatic license generation requires headers that are not shipped in this package.

Typical invocation (shown in bash continuation style; Windows cmd.exe users substitute `^^` for `\\`, or PowerShell users substitute a backtick, or simply put the whole thing on one line):

```
tools/license_create --key private.pem \  
  --vendor-license vendor_license.json \  
  --id "LIC-001" \  
  --licensee "End User Corp" \  
  --product "YourApp" \  
  --version "3.*" \  
  --expires "2027-12-31T23:59:59Z" \  
  --grace-days 7 \  
  --fingerprint-value "<end user's fingerprint>" \  
  --threshold 2 \  
  --feature "feature_a" \  
  --feature "feature_b"
```

See §4.3 below for full flag documentation, including the floating-license and permanent-license forms. The end user's fingerprint is the value they get from running `rg_fingerprint` on their machine (§4.2).

## B) YOUR END-USER APPLICATION (no vendor license needed):

This is the application you ship to your end users.

```
#include <rockyguard/rockyguard.h>  
  
static constexpr char PUBLIC_KEY[] = R"-----BEGIN PUBLIC KEY-----  
...your public key...  
-----END PUBLIC KEY-----)";  
  
int main() {  
    rockyguard::LicenseVerifier verifier(PUBLIC_KEY);  
    auto result = verifier.load("license.json");  
    if (!result) {  
        std::cerr << result.message << std::endl;  
        return 1;  
    }  
  
    result = verifier.check_node_locked();  
    if (result.status == rockyguard::LicenseStatus::InGracePeriod) {  
        std::cerr << "License expiring soon: "  
                  << result.grace_days_remaining << " days left\n";  
    } else if (!result) {  
        std::cerr << result.message << std::endl;  
        return 1;  
    }  
  
    // Optional version gate (v1.2.1+). Include this block if you  
    // issue licenses with --version "3.*" (or similar) and want  
    // the library to ENFORCE the range at runtime; omit it if
```



```
// version_range should remain informational only.
// check_version() is opt-in: the library never consults
// version_range automatically. See Customer_API_Reference §6.7
// for the supported syntaxes (glob and comparator).
static constexpr char APP_VERSION[] = "3.1.0";
result = verifier.check_version(APP_VERSION);
if (!result) {
    std::cerr << "Version not allowed by license: "
               << result.message << std::endl;
    return 1;
}

// Feature gating
if (verifier.check_feature("feature_a")) {
    enable_feature_a();
}

return 0;
}
```

The recommended startup sequence is therefore: `load()` -> `check_node_locked()` -> (optional) `check_version()` -> per-feature `check_feature()` calls. `check_node_locked()` does the heavy lifting (signature, hardware match, clock-manipulation detection, integrity self-check, expiry / grace evaluation); `check_version()` and `check_feature()` are cheap data lookups that run after. See Customer\_API\_Reference §6 "What each check method re-evaluates" for the full per-method coverage matrix.

### 3.4 What to Ship to Your End User

The vendor license, your private signing key, and the `license_create` / `license_keygen` tools from your customer zip NEVER ship to end users. Those stay on your build machine. What goes in the customer-facing distribution depends on the link variant you chose:

Static link (§3.1, the recommended default):

- `your_application.exe` (or unsuffixed binary on Linux): your own executable with the RockyGuard library compiled in. No external RockyGuard files needed alongside it.
- `license.json`: the end-user license you generated for this customer. Conventional location is next to the executable; if you load from a different path in your code, document it for the end user. One file per end user / per machine.

That is the entire shipping list for the static path. Specifically NOT shipped: `rockyguard.li...`

Shared (DLL) link (§3.2):

- `your_application.exe`: your own executable.
- `rockyguard.dll`: the shared library. Same directory as the `.exe`.
- `rockyguard.sig`: integrity signature for the DLL. Same directory as the DLL. The library refuses to load without it (see §9.3).
- `libcrypto-3-x64.dll`, `libssl-3-x64.dll`: bundled OpenSSL runtime.

Same directory as the .exe (Windows DLL search rules).

- license.json: the end-user license, as in the static path.

NOT shipped: rockyguard.lib (it is the import library, link-time only), the deps/ folder, anything from the package's lib/static/ or tools/ directories.

Linux equivalents (same logical files, different extensions):

Linux static link:

- your\_application (unsuffixed binary): your own executable with the RockyGuard library compiled in.
- license.json: the end-user license, as in the Windows static path.

Linux shared link:

- your\_application: your own executable.
- librockyguard.so: the shared library. Same directory as the executable (or somewhere reachable via RUNPATH / LD\_LIBRARY\_PATH if you ship it elsewhere).
- librockyguard.sig: integrity signature for the .so. Same directory as the .so. The library refuses to load without it.
- license.json: the end-user license.

NO separate OpenSSL .so files are shipped on Linux: the Linux shared build statically links libssl and libcrypto into librockyguard.so itself (vcpkg's x64-linux triplet uses static OpenSSL), so the .so is self-contained for the crypto runtime. This is the inverse of the Windows shared build, which DOES ship libcrypto-3-x64.dll and libssl-3-x64.dll alongside rockyguard.dll. See §3.2 for the full discussion of the Windows-vs-Linux asymmetry.

End-user license file location is up to you. The library loads from whatever path you pass to LicenseVerifier::load(). Common conventions: alongside the executable (single-tenant desktop apps), under %APPDATA%\YourCompany\YourProduct\license.json on Windows, or \$XDG\_CONFIG\_HOME/YourCompany/license.json on Linux.

For a floating-licensing deployment (Premium tier, §5), the same per-customer "ship list" applies on the end-user side. The floating server itself is a SEPARATE deployment to a host inside your network, and it gets its own copy of the vendor license -- treat that machine with the same care as your build machine.

## 4. END-USER LICENSE MANAGEMENT

The end-to-end flow has three actors: YOU (the library customer running `license_create`), YOUR end user (the person whose machine the license will run on), and YOUR application (which embeds the verifier and reads `license.json` at startup). Two flags on `license_create` control which machine the license binds to and they are easy to confuse:

- `--fingerprint-value "<hash>"` The PRODUCTION path. Bind the license to the hash your end user sent you (they obtained it by running `rg_fingerprint` on their own machine; see §4.2). The license verifies on THEIR machine and only theirs.
- `--fingerprint` The SELF-TEST / TOOLCHAIN-CHECK path. Auto-detect THIS machine's fingerprint and bind the license to it. Convenient when you want to verify your keypair, vendor license, and `license_create` / verify pipeline end-to-end on your own build host without involving a real customer. Do NOT ship a self-fingerprinted license to an actual end user; it will fail verification on their machine because their hardware fingerprint is different.

If both flags are provided, `--fingerprint-value` wins. The Quick Start in §2.4 uses `--fingerprint-value` because that is what new customers actually want; the testing alternative is mentioned there as a one-liner.

### 4.1 Generating Your Keypair

Run once. Keep the private key secret; embed the public key in your app.

```
tools/license_keygen --private private.pem --public public.pem
```

For RSA instead of Ed25519:

```
tools/license_keygen --algo rsa --rsa-bits 2048 \  
--private private.pem --public public.pem
```

Choosing an RSA key size:

Ed25519 is the recommended default for new deployments -- it provides a 128-bit security level with smaller keys and signatures and faster verification, and there are no key-size choices to second-guess. The remainder of this sub-section applies only if you have a specific reason to use RSA (e.g., a corporate cryptographic-policy mandate, FIPS-validated module requirements, or interoperability with an existing RSA toolchain).

--rsa-bits	Security level	Recommendation
1024	~80 bits	NOT supported by OpenSSL 3 at default security level; rejected at key generation. Do not use.
2048	~112 bits	The library default. Adequate for license-signing workloads in 2026 and remains acceptable through the late 2020s per common guidance (NIST SP 800-57 Part 1, BSI TR-02102). Adequate for most products through the standard product lifecycle.
3072	~128 bits	Recommended for keys you expect to use beyond ~2030. Same security level as Ed25519. Verification is roughly 2-3x slower than 2048 (still sub-millisecond on modern hardware -- no practical impact at the license-verify call site, which fires once at application startup).

--rsa-bits	Security level	Recommendation
4096	~152 bits	Maximum margin against future cryptanalysis. Verification is roughly 5-7x slower than 2048; signing is several times slower than that. The cost is paid by YOUR license_create runs (signing) and your end users' application startup (verification). For a desktop application's once-at-launch verify the absolute latency is still in the low single-digit milliseconds; choose 4096 freely if your security policy mandates it.

The library accepts any --rsa-bits value OpenSSL accepts (multiples of 8, subject to OpenSSL's per-version minimum at the configured security level). 2048, 3072, and 4096 are the values to consider; everything else is either too weak or non-standard. The library does not impose its own ceiling.

A note on key longevity: license-signing keys are typically long-lived because rotating the signing key requires shipping a new application binary with the new public key embedded AND re-issuing every customer's existing license under the new key (Customer\_Documentation §9.1). Pick a key size strong enough to comfortably outlive the longest support window you expect to commit to for the products it signs.

Performance numbers above are approximate and intended only for rough sizing -- the actual throughput depends on your CPU, OpenSSL build, and whether hardware acceleration is available. Verification (the customer-facing path) is far cheaper than signing in all cases.

## 4.2 Getting an End User's Machine Fingerprint

Ask your end user to run:

```
rg_fingerprint -v
```

Or to save to a file they can send you:

```
rg_fingerprint -v -o fingerprint.txt
```

The output looks like:

```
MAC address:      6c:2f:80:5e:e8:45
CPU ID:           GenuineIntel-...
Disk serial:      ...
Motherboard ID:   ...
311867...|ff9f0f...|877f34...|f3b059...
```

The unlabeled line at the bottom is the composed hardware fingerprint, four SHA-256 component hashes joined with pipe ('|') in fixed order: MAC | CPU | Disk | Motherboard. Use that line verbatim when creating the end user's license (pass it to license\_create --fingerprint-value). If a component cannot be read on the target machine (for example, the disk serial is unavailable inside a VM), the corresponding slot is the SHA-256 of the empty string and the library's match\_count skips such slots when computing hardware-match scores.

What if every component is empty? On a stripped Docker container (no MAC adapter exposed, no /sys/class/dmi access, no SMART data), on a CI runner deliberately scrubbed of host identifiers, or on some sandboxed VM configurations, all four slots can come back empty. In that case the fingerprint is four copies of SHA-256("") joined by pipes, and on the verifier side match\_count returns 0 -- which fails the default fingerprint\_match\_threshold of 2, so a node-locked license issued for any other host is correctly rejected on this machine (LicenseStatus::HardwareMismatch).

Two practical implications:

- Do NOT issue a node-locked license against an all-empty fingerprint expecting it to be accepted on the same machine later. The verifier still returns HardwareMismatch even against a matching all-empty fingerprint, because match\_count of 0 is below the threshold of 2 by design.
- If you genuinely want a license to verify on machines with no readable hardware identifiers (server-room appliances, fully sandboxed end users), issue the license with --threshold 0 -- this is the explicit "not hardware-locked" setting; the verifier then accepts any fingerprint, including the all-empty one. See Customer\_API\_Reference §7.5 for the underlying match\_count semantics.

## 4.3 Creating End-User Licenses

All license\_create invocations require --vendor-license pointing at the vendor\_license.json you received from your library vendor. The tool will refuse to run without it.

Node-locked license:

```
tools/license_create --key private.pem \  
  --vendor-license vendor_license.json \  
  --id "LIC-001" \  
  --licensee "End User Corp" \  
  --product "YourApp" \  
  --version "3.*" \  
  --expires "2027-12-31T23:59:59Z" \  
  --grace-days 7 \  
  --fingerprint-value "<end user's fingerprint>" \  
  --threshold 2 \  
  --feature "export_pdf" \  
  --feature "advanced_reports"
```

Floating license (Premium tier):

```
tools/license_create --key private.pem \  
  --vendor-license vendor_license.json \  
  --id "FLOAT-001" \  
  --licensee "End User Corp" \  
  --product "YourApp" \  
  --type floating \  
  --expires "2027-12-31T23:59:59Z" \  
  --max-users 10
```

Permanent license:

```
tools/license_create --key private.pem \  
  --vendor-license vendor_license.json \  
  --id "PERM-001" \  
  --licensee "End User Corp" \  
  --product "YourApp" \  
  --expires permanent \  
  --fingerprint-value "<fingerprint>"
```

License with metadata (v1.2.1+; pass --metadata key=value, repeatable):

```
tools/license_create --key private.pem \  
  --vendor-license vendor_license.json \  
  --id "LIC-2026-0042" \  
  --licensee "Acme Inc." \  
  --product "YourApp" \  
  --expires "2027-12-31T23:59:59Z" \  
  --fingerprint \  
  --metadata "customer_po=PO-2026-0042" \  
  --metadata "region=NA" \  
  --metadata "channel=direct"
```

See §8.2 for the canonical description of the metadata field (semantics, recommended use cases, read-side examples, size guidance) and §7.3 for the --metadata flag's parse rules (delimiter handling, empty-value behavior, repeated-key behavior, shell-quoting tips).

## 4.4 Verifying a License File

Quick check (signature + payload only):

```
tools/license_verify --key public.pem --license license.json
```

Full check (also re-runs the hardware fingerprint match and confirms a specific feature is licensed):

```
tools/license_verify --key public.pem --license license.json \  
  --check-hardware --check-feature "export_pdf"
```

## 4.5 Fuzzy Hardware Matching

The fingerprint contains 4 hardware component hashes separated by '|'. When verifying, each component is compared independently. The fingerprint\_match\_threshold (default: 2) controls how many must match.

If an end user replaces one component (e.g. new network adapter), the license still validates as long as enough other components match. Set --threshold 2 for a good balance between security and convenience.

Unavailable hardware components (e.g., no disk serial on a virtual machine) are automatically skipped during matching - they don't count as a match or a mismatch. The library warns when a component is unavailable during fingerprint generation.

## 4.6 Feature Gating

Add features to a license with --feature (can repeat). In your app:

```
if (verifier.check_feature("export_pdf")) {  
    enable_pdf_export();  
}  
  
if (verifier.check_feature("advanced_reports")) {
```

```
    enable_advanced_reports();  
}
```

Create different license tiers by varying the feature list:

```
Basic:      --feature "basic"  
Pro:        --feature "basic" --feature "export_pdf" --feature "reports"  
Enterprise: --feature "basic" --feature "export_pdf" --feature "reports"  
            --feature "api_access" --feature "sso"
```

## 4.7 Grace Periods

Set `--grace-days N` when creating a license. After expiry, the license continues to work for N days with status `InGracePeriod`. Your app can show a renewal warning:

```
if (result.status == rockyguard::LicenseStatus::InGracePeriod) {  
    show_warning("License expired. " +  
                std::to_string(result.grace_days_remaining) +  
                " days remaining before deactivation.");  
}
```

## 5. FLOATING LICENSING (PREMIUM TIER)

Floating licensing requires a Premium tier library license. If you have a Basic tier, contact the library vendor to upgrade.

### 5.1 Creating a Floating License

```
tools/license_create --key private.pem \  
  --vendor-license vendor_license.json \  
  --id "FLOAT-001" --licensee "Corp" --product "App" \  
  --type floating --max-users 10 \  
  --expires "2027-12-31T23:59:59Z"
```

--vendor-license is required by every license\_create invocation (see §4.3); the tool refuses to run without it. The example above shows only the floating-specific flags; for the full flag set see §4.3.

### 5.2 Running the License Server

The floating license server requires a vendor license (Premium tier). Deploy it on a machine accessible to all end users. The supported customer path is the shipped rg\_floating\_server binary configured via a YAML file (described in this section); ship the binary plus its YAML config to the host you want to run the server on. The programmatic FloatingLicenseServer C++ class is NOT part of the customer-distributed header set in v1.2.x. If you have a strong need to embed the server inside your own service rather than running it as a separate process, contact the vendor to discuss a custom build that exposes the header; this is a v1.4+ scope question (see Future\_Roadmap).

Configuration file:

Create a config file (e.g. server\_config.yaml). The example below shows every recognised key. Required keys are marked; everything else is optional with the default value shown.

```
# Vendor license file (REQUIRED, Premium tier).  
vendor_license: vendor_license.json  
  
# License pool size (REQUIRED).  
max_users: 10  
  
# Per-machine seat cap (v1.2.1+; defends against ghost-checkout  
# exhaustion). 0 (default) = uncapped, preserving v1.2.0 behavior.  
# Set above the expected steady-state concurrency on one workstation  
# to leave headroom for crash-recovery (ghost leases count toward  
# this cap until the reaper TTL fires).  
max_leases_per_machine_id: 0  
  
# Server settings.  
port: 8080  
heartbeat_interval: 60  
lease_timeout: 120 # Seconds. The YAML key is
```



```
# lease_timeout (no _sec suffix);
# the corresponding C++ struct
# field is FloatingServerConfig::
# lease_timeout_sec. Both names
# appear in this document depending
# on whether the discussion is
# about operator tuning (YAML) or
# the underlying API field.

# Thread pool and client timeout.
thread_pool_size: 4
client_timeout: 5                                # Seconds. Same YAML/struct split
                                                # as lease_timeout above: the C++
                                                # field is FloatingServerConfig::
                                                # client_timeout_sec.

# Response signing (recommended): server signs every response.
private_key: private.pem                        # Path to PEM file. The C++
                                                # struct field that holds the
                                                # loaded PEM contents is
                                                # FloatingServerConfig::
                                                # signing_private_key_pem.

# TLS encryption (optional): uncomment to enable HTTPS.
# tls_cert: server.crt
# tls_key: server.key

# Server mode: release or debug.
mode: release

# Log to file (uncomment to enable).
# log_file: server.log

# Log rotation (v1.2.1+; only applies when log_file is set).
# Defaults shown below: rotate at 100 MiB, keep 5 archives. Set
# log_max_bytes <= 0 to disable rotation (v1.2.0 unbounded growth).
# log_max_bytes: 104857600
# log_keep_count: 5
```

Run the server:

```
rg_floating_server server_config.yaml
```

A sample config file (floating\_server\_config.yaml) is included in the tools/ directory.

Server modes:

The mode key controls log verbosity. Two values are accepted.

In release mode (the default), the server logs basic events only: server start and stop, license checkouts and checkins, lease expirations, and errors. This is suitable for production use.

In debug mode, the server logs everything from release mode plus full request and response details, client connection info (IP and port), heartbeat events, periodic lease-table dumps with heartbeat ages, worker thread activity, and client

timeouts. Use for troubleshooting; the volume is high.

Log output:

By default, all log output goes to stderr. To log to a file, set the `log_file` field in the config; logs are appended, not overwritten. The format is:

```
2026-04-09 20:25:23.533 [INFO] Server started on port 8080
2026-04-09 20:25:24.100 [DEBUG] Connection from 192.168.1.5:52341
2026-04-09 20:25:24.102 [INFO] Checkout: client abc-123 - license granted (1/10)
```

Log rotation (v1.2.1+, file-backed loggers only):

When `log_file` is set, the active log file rotates as soon as it reaches `log_max_bytes` (default 100 MiB). The rename cascade is:

```
server.log    -> server.log.1
server.log.1  -> server.log.2  (prior .2 promoted to .3, and so on)
server.log.N  -> dropped      (where N = log_keep_count)
```

A fresh `server.log` is opened immediately after the rename. Worst-case disk usage per `log_file` path is bounded by `log_max_bytes * (log_keep_count + 1)` -- 600 MiB at the defaults. Set `log_max_bytes <= 0` to disable rotation entirely (preserves the v1.2.0 unbounded-growth behavior). `log_keep_count <= 0` means "rotate but keep no archives" -- each rotation truncates the log. `log_keep_count` is silently clamped to the range `[0, 100]` -- values above 100 are pinned to 100 so a misconfigured "keep a million archives" cannot turn each rotation into an  $O(N)$  filesystem walk. Rotation runs under the writer mutex, so log lines are never lost or torn across the rename cascade. Stderr-backed loggers do not rotate; the host's logging system handles that path.

Per-machine seat cap (v1.2.1+):

`max_leases_per_machine_id` bounds the number of concurrent leases any single workstation can hold. Without it, a flaky client that crashes and restarts before sending checkin can multiply its lease hold (the ghost lease persists until `lease_timeout` fires). With it set, the  $(cap+1)$ th checkout from the same machine is denied with the dedicated error "machine seat limit reached" (`LicenseStatus::MachineSeatLimitReached` on the client side), even if the global pool still has seats free for other machines.

Sizing guideline: set the cap ABOVE the steady-state concurrency you expect on one workstation. Ghost leases count toward this cap until the reaper TTL (`lease_timeout`) fires, so a tight cap means a crashed client temporarily blocks one slot for the reaper TTL after restart. A reasonable default for a CAD-style product where one user usually runs at most three concurrent sessions is `cap=4` or `cap=5`: legitimate concurrency works, but a single machine still cannot drain a 50-seat pool by crashing in a tight loop.

Thread pool and connection handling:

The server uses a thread pool. The main thread accepts connections and dispatches them to a pool of worker threads. Each worker sets a receive timeout on the client socket to protect against slow or malicious clients.

`thread_pool_size` (default 4) sets the number of worker threads. Each handles one connection at a time; raise it for high-concurrency environments.

`client_timeout` (default 5 seconds) drops a connection if the client connects but does not send data within this time. Protects against denial-of-service attacks where a client connects and holds the connection open.

The server and client both handle TCP fragmentation correctly: HTTP requests and responses that arrive in multiple packets are reassembled using Content-Length before parsing.

#### Response signing and TLS:

The server can sign every response with a private key; the client verifies the signature using the corresponding public key (`FloatingClientConfig::server_public_key_pem`). This prevents server spoofing (e.g., someone redirecting traffic to a fake server that always returns "checked\_out"). Set `private_key` in the server config to the path of an Ed25519 or RSA private key, and ship the corresponding public key with the client.

Recommended: use a SEPARATE keypair for server response signing, not the keypair used for license signing.

The server's signing key and the license-signing key are independent fields in the library: the server takes its key via `private_key` in the YAML config; the license-signing key is the one you pass to `license_create --key`. Nothing enforces they be the same. Using a dedicated server keypair is the recommended configuration because it separates two distinct trust roles:

- The license-signing key authorizes minting of new end-user licenses. Compromise mints unauthorized licenses but does not let an attacker impersonate your live floating server.
- The server-signing key authenticates live server responses (checkout, heartbeat, checkin). Compromise lets an attacker run a fake server but does not let them mint licenses.

If you reuse one keypair for both roles, a single compromise gives the attacker both capabilities at once. Generating a second keypair is a one-time cost (`tools/license_keygen --private server_signing_priv.pem --public server_signing_pub.pem`); the operational handling is otherwise identical to the license keypair. Embed the server signing PUBLIC key in your end-user binary as `FloatingClientConfig::server_public_key_pem` (alongside, but distinct from, the license-verification public key embedded in `LicenseVerifier`).

Acceptable for very small or pre-production deployments: reusing the license-signing keypair is a supported path -- the field is independent precisely so you can choose. If you go that route, do it with the explicit understanding that the two compromise scenarios above collapse into one. For any production deployment exposed to a network you do not fully control, generate a separate server keypair.

Rotation: you can rotate the server key independently of the license-signing key as long as you ship a new application binary carrying the new server public key (clients pin server identity via the embedded key). Rotating the license-signing key has the heavier cost called out in §9.1 (re-issuing every existing customer's license under the new key). Separating the two keypairs lets you respond to a server-side incident without that license-re-issuance cost.

For additional traffic encryption, enable TLS by providing a certificate and key. Generate a self-signed certificate:

```
openssl req -x509 -newkey ec -pkeyopt ec_paramgen_curve:prime256v1 \
-days 3650 -nodes -keyout server.key -out server.crt \
-subj "/CN=license-server"
```

Then set `tls_cert` and `tls_key` in the server config file. On the client, set `use_tls = true` and `tls_ca_cert_path = "server.crt"` to enable certificate verification. Without `tls_ca_cert_path`, TLS encrypts traffic but does not verify the server identity, and the library logs a warning about MITM vulnerability.

## 5.3 Client Integration (No Vendor License Needed)

In your end-user application (see `examples/rg_floating_client.cpp` for a complete working example):

```
rockyguard::FloatingClientConfig config;
config.server_host = "license-server.internal";
config.server_port = 8080;
config.heartbeat_interval_sec = 30;
config.server_public_key_pem = PUBLIC_KEY; // Verify server responses
// config.use_tls = true; // Enable if server uses TLS
// config.tls_ca_cert_path = "server.crt"; // Verify server certificate

rockyguard::FloatingLicenseClient client(config);

auto result = client.checkout();
if (!result) {
    std::cerr << result.message << std::endl;
    return 1;
}

// Application runs... heartbeat is automatic
// License is returned on client.checkin() or destructor
```

## 5.4 Server HTTP Endpoints

```
POST /checkout { "client_id": "uuid", "machine_id": "..." } -> 200 or 403
POST /checkin { "client_id": "uuid", "machine_id": "..." } -> 200
POST /heartbeat { "client_id": "uuid", "machine_id": "..." } -> 200
GET /status -> { total, active, avail...
```

The `machine_id` field carries the host's hardware fingerprint -- the same |-joined SHA-256 hashes (MAC, CPU ID, disk serial, motherboard ID) used for node-locked verification. The server stores the hashes only; raw hardware values never leave the host. v1.2.1+ servers use `machine_id` to enforce `max_leases_per_machine_id`; v1.2.0 servers logged the field but did not act on it. Three further fields are sent on every POST but elided here for clarity: a fresh nonce, a monotonic sequence counter, and a session-secret-keyed HMAC (replay protection + client authentication). See [Customer\\_API\\_Reference §8.3](#) for the full wire-data inventory.

Leases are automatically released if a heartbeat is not received within the `lease_timeout` window (YAML key `lease_timeout`, default 120 seconds; the matching C++ struct field is `FloatingServerConfig::lease_timeout_sec`).

## 6. HARDWARE FINGERPRINTING

---

### 6.1 Collected Components

Component	Source
-----	-----
MAC address	Physical NIC (virtual adapters filtered out)
CPU ID	Vendor + brand string
Disk serial	Boot disk serial number
Motherboard ID	Serial number or platform UUID

On Windows, Ethernet NICs are preferred over Wi-Fi. Virtual adapters (VMware, VirtualBox, Hyper-V, Docker, VPN, WireGuard, Bluetooth, etc.) are automatically filtered out.

### 6.2 Using the Fingerprint API

```
auto hw = rockyguard::HardwareFingerprint::collect();
std::cout << "MAC:  " << hw.mac_address << "\n";
std::cout << "CPU:  " << hw.cpu_id << "\n";

// Get full fingerprint string
std::string fp = rockyguard::HardwareFingerprint::fingerprint();

// Compare fingerprints (returns 0-4 matching components)
int matches = rockyguard::HardwareFingerprint::match_count(fp1, fp2);
```

## 7. CLI TOOLS REFERENCE

---

### 7.1 rg\_fingerprint

Usage: rg\_fingerprint [options]

Print the machine's hardware fingerprint.

-v, --verbose            Show individual hardware components  
-o, --output <file>    Write fingerprint to file (default: stdout)

Examples:

```
# Print fingerprint to screen
rg_fingerprint

# Show detailed components + fingerprint
rg_fingerprint -v

# Save to file (to send to vendor or use with license_create)
rg_fingerprint -o fingerprint.txt
```

### 7.2 license\_keygen

Usage: license\_keygen [options]

--algo <ed25519|rsa>    Algorithm (default: ed25519). Ed25519 is recommended; RSA is provided for policy / interoperability reasons only. There is no "auto" choice here -- auto-detection is not applicable to key generation (you are creating a key, not loading one).

--rsa-bits <bits>       RSA key size when --algo rsa (default: 2048). Ignored when --algo ed25519. Recommended values: 2048 for typical deployments through the late 2020s; 3072 for keys expected to live past ~2030; 4096 for maximum margin / policy mandates. Smaller values are rejected by OpenSSL at default security level. See §4.1 "Choosing an RSA key size" for the full recommendation table including security levels and verification-cost notes.

--private <file>        Private key output (default: private.pem)

--public <file>         Public key output (default: public.pem)

### 7.3 license\_create

Usage: license\_create [options]

--key <file>            Private key PEM file (required)

`--vendor-license <file>` Vendor license file (required for every invocation that generates or inspects an end-user license, including `--show-count`; see §4.3). Same `vendor_license.json` the library vendor sent you (§2.2).

`--algo <auto|ed25519|rsa>` Signature algorithm (default: `auto`; `v1.2.1+`). "auto" inspects the private key and picks `ed25519` or `rsa` automatically. Pass an explicit value only if you want to ASSERT one algorithm (e.g., a security policy that rejects RSA after migrating to `Ed25519`). Unknown values are rejected; in `v1.2.0` a typo silently fell back to `ed25519`.

`--output <file>` Output file (default: `license.json`)

`--id <id>` License ID (required, must be unique)

`--licensee <name>` Licensee name

`--product <name>` Product name (required)

`--version <range>` Version range. Glob form ("`3.*`", "`3.1.*`", "`3.1.5`") or comparator form ("`>=3.0,<4.0`", "`!=3.5.0`"). Empty or "`*`" matches any version. Enforced by `LicenseVerifier::check_version()` (`v1.2.1+`); see `Customer_API_Reference §6.7` for the full syntax.

`--type <node_locked|floating>` License type (default: `node_locked`)

`--expires <date>` Expiry (ISO 8601) or "permanent"

`--grace-days <n>` Grace period in days (default: 0)

`--max-users <n>` Max concurrent users (floating only)

`--feature <name>` Add a feature (repeatable)

`--metadata <key=value>` Add a metadata pair (repeatable; `v1.2.1+`). Round-tripped to verifier-side as `License::metadata[key]`; useful for customer-internal IDs (PO, account ID), region tags, channel labels, anything you want signed alongside the license without changing the schema. The library does not interpret these.

#### Parse rules:

- The FIRST '=' is the delimiter. Key is everything before it; value is everything after it. A value may therefore contain '=' freely:  
`--metadata "formula=a=b+c"` sets `key="formula"`, `value="a=b+c"`.
- An argument with no '=' is rejected with "Error: --metadata requires key=value form" and the tool exits non-zero. Pass `key=` (with a trailing '=' and nothing after) for a presence-flag entry.
- An empty key (the argument starts with '=', e.g. `--metadata "=foo"`) is rejected with "Error: --metadata key cannot be empty".

- An empty value is ALLOWED:  
--metadata "is\_trial=" sets the key with an empty string value. Use this when the presence of the key is the signal and the value does not matter.
- The same --metadata key passed twice on one command line is last-write-wins (License::metadata is a std::map<string,string>).
- No character restrictions on keys or values: any byte sequence your shell can deliver is accepted. Quote arguments that contain spaces, '=' (in the value), '\$', '|', '&', or any other shell meta-character so the WHOLE key=value pair reaches the tool as one argv entry.

#### Quoting tips:

bash / zsh:

```
--metadata 'po_number=PO-12345'  
--metadata "region=North America"
```

Windows cmd.exe:

```
--metadata "po_number=PO-12345"  
--metadata "region=North America"
```

PowerShell:

```
--metadata "po_number=PO-12345"  
Use single quotes around values  
that contain '$' to suppress  
PowerShell variable expansion:  
--metadata 'tag=$revision'
```

--fingerprint

Auto-detect THIS MACHINE's fingerprint and bind the license to it. Useful for self-licensing or end-to-end testing of the toolchain on your own build host. NOT what you want when issuing to a remote end user -- their machine has a different hardware fingerprint, so the license will not verify on their host. Use --fingerprint-value for that case.

--fingerprint-value <v>

Bind the license to a specific hardware fingerprint string. This is the standard production path: the end user runs rg\_fingerprint on their own machine and sends you the resulting hash; you pass that hash here.

--threshold <n>

Fingerprint match threshold (default: 2)

--show-fingerprint

Print machine fingerprint and exit

--show-count

Print the current end-user license generation count and exit (read-only; requires --vendor-license; v1.2.1+). Output includes the count, the vendor-license id, and the cap with remaining budget. Use to monitor approach to the



cap before hitting "End-user license generation limit reached" (§10.3.2). See §9.7 for storage details.

## 7.4 license\_verify

Usage: license\_verify [options]

<code>--key &lt;file&gt;</code>	Public key PEM file (required)
<code>--license &lt;file&gt;</code>	License file (required)
<code>--algo &lt;auto ed25519 rsa&gt;</code>	Signature algorithm (default: auto; v1.2.1+). "auto" inspects the public key and picks ed25519 or rsa automatically; this matches the LicenseVerifier C++ default (Customer_API_Reference §6.1). Pass an explicit value only if you want to ASSERT one algorithm. Unknown values are rejected; in v1.2.0 a typo silently fell back to ed25519.
<code>--check-hardware</code>	Verify hardware fingerprint
<code>--check-feature &lt;name&gt;</code>	Check if feature is licensed
<code>--check-version &lt;ver&gt;</code>	Check that <ver> satisfies the license's version_range (v1.2.1+; corresponds to LicenseVerifier::check_version() at Customer_API_Reference §6.7). Prints "Version '<ver>' against range '<range>': PASS FAIL - <message>" and exits non-zero on FAIL. Empty version_range matches any <ver>. Glob form ('3.*', '3.1.5') and comparator form ('>=3.0,<4.0,!=3.5.0') are both accepted; full syntax in Customer_API_Reference §6.7. Useful for validating an issued license against a target build's version before shipping.

## 8. LICENSE FILE FORMAT

### 8.1 Structure

```
{
  "payload": "<JSON string>",
  "signature": "<base64-encoded Ed25519/RSA signature>"
}
```

The signature covers the exact payload bytes. Any modification to the payload invalidates the signature.

### 8.2 Payload Fields

Field	Type	Description
license_id	string	Unique license identifier (REQUIRED)
licensee	string	Licensed entity name
product	string	Product name (REQUIRED)
version_range	string	Version pattern: "3.*" / "3.1.*" / "3.1.5" (glob form) or ">=3.0,<4.0" / "!=3.5.0" (comparator form). Empty or "*" = match any. Enforced by LicenseVerifier::check_version() v1.2.1+; informational only in v1.2.0.
type	string	"node_locked" or "floating"
hardware_fingerprint	string	Component hashes (pipe-separated)
fingerprint_match_threshold	int	Min matching components (default: 2)
issued_at	string	ISO 8601 issue date
expires_at	string	ISO 8601 expiry or "permanent"
grace_period_days	int	Days after expiry before block
max_concurrent_users	int	Pool size (floating only)
features	string[]	Feature flag names
metadata	object	Arbitrary string->string pairs

The metadata field is a customer-defined string -> string map that the library round-trips through the signed license file without inspecting it. Set values from license\_create with --metadata key=value (repeatable; v1.2.1+); read them on the verifier side via verifier.license().metadata, which returns a std::map<std::string, std::string>. The library does not enforce or interpret any key; the JSON object is stored exactly as you supplied it.

Why use metadata instead of a sidecar config file:

- It is signed: any byte change invalidates the license signature, so an end user cannot edit metadata after the fact without breaking license verification entirely.
- It is bound to the license: you cannot pair Customer A's license with Customer B's metadata, since both travel in the same signed payload.

- It is one file to deliver and one file to deploy.

#### Common use cases:

- Audit and accounting trail. Bind business data to the license so finance and legal can verify it later. Example keys: `customer_po` (the purchase order the license was issued against), `contract` (an MSA / quote / SO reference), `account_id` (your CRM identifier for the licensee), `channel` (reseller, region, or salesperson code).

Example invocation (set at issuance):

```
tools/license_create --key private.pem \  
  --vendor-license vendor_license.json \  
  --id "LIC-2026-0042" --product "YourApp" \  
  --licensee "Acme Inc." \  
  --expires "2027-12-31T23:59:59Z" \  
  --fingerprint \  
  --metadata "customer_po=PO-2026-0042" \  
  --metadata "contract=MSA-2025-Acme-v3" \  
  --metadata "account_id=ACC-91827"
```

Read on the support side (e.g., from a debug-info dump in your application):

```
const auto& meta = verifier.license().metadata;  
auto it = meta.find("customer_po");  
if (it != meta.end()) std::cout << "PO: " << it->second << "\n";
```

- Pass-through data your application needs to know about the licensee. Many products have an internal taxonomy independent of the license itself: tier or plan names, regions for data-residency compliance, edition / SKU identifiers. Metadata is the supported channel for carrying that data signed.

Example keys: `plan` (your product's tier name -- "free", "pro", "enterprise"), `region` (where t...

Example application-side use, after `verifier.load()` succeeds:

```
const auto& meta = verifier.license().metadata;  
std::string plan = meta.count("plan") ? meta.at("plan") : "free";  
if (plan == "enterprise") enable_enterprise_features();
```

- Anti-leak forensics. If a license file ends up on a public file-share or in a leaked archive, metadata can identify which cohort it was issued to. Example keys: `issued_by` (an internal staff member ID), `cohort` (a quarterly batch label), `watermark` (a unique-per-license tag for tracing).

Example:

```
--metadata "issued_by=staff-91" \  
--metadata "cohort=2026-Q1-acme"
```

Because the license is signed, an attacker who copies the file cannot strip the watermark wit...

- Display strings your application can trust. The licensee field is the legal entity ("Acme Inc."); metadata can carry a friendlier display string for user-visible UI ("Acme Manufacturing, Munich"). Both are signed and both come from your issuance pipeline.

Example: `--metadata "display_name=Acme Manufacturing, Munich".`

- Numeric limits beyond binary feature flags. The features array is binary (a flag is either listed or not). When you need quantitative caps -- `max_seats` specific to your product, an API rate limit, a storage quota -- metadata carries them as strings; your application parses and enforces. RockyGuard guarantees the value came from a signed license, but does not enforce it itself.

Example:

```
--metadata "max_seats=50" \  
--metadata "api_rate_limit_qps=100" \  
--metadata "storage_gb=500"
```

Application side:

```
int seats = std::stoi(verifier.license().metadata.at("max_seats"));
```

(Wrap `stoi` in `try/catch` and pick a safe default if the key is absent or unparseable; `metadata...`

A note on size: the entire license is signed and re-parsed on every verification call. Keep individual values small (kilobytes, not megabytes); a multi-megabyte metadata payload makes every license check slow and has no good reason to be there. If you need to ship a large config blob to your end user, ship it next to the license as a separate file, but reference it from metadata (e.g., a SHA-256 of the config you trust).

## 9. ANTI-TAMPERING FEATURES

---

The library includes several anti-tampering mechanisms that protect end-user licenses from bypass attempts. These work automatically and require no configuration.

### 9.1 Digital Signatures

Every license file is digitally signed with Ed25519 or RSA-SHA256. The signature is verified BEFORE the payload is parsed. Modifying any byte in the payload invalidates the signature.

The library is safe against malformed license files: invalid JSON, wrong value types, or missing fields all return a clean error (MalformedFile) without crashing the host application.

### 9.2 Clock Manipulation Detection

The library detects system clock rollback to prevent end users from setting their clock backward to extend an expired license.

- UTC timestamps are stored in multiple hidden locations on the machine (see "Storage locations" below)
- Anchor files are unique per license (derived from license\_id, product, and licensee), so different products don't interfere with each other. The per-license tag is SHA-256(license\_id + "|" + product + "|" + licensee), abbreviated to 12 hex characters; you'll see it as the suffix on every storage location below. The "|" separators matter -- they prevent collisions between fields that would otherwise concatenate ambiguously (e.g., id="AB"+product="CD" vs id="ABC"+product="D")
- On each license check, all locations are read and cross-verified
- If the current time is more than 1 hour behind the maximum stored timestamp, clock manipulation is detected
- The stored timestamps are integrity-protected; editing them is detected
- Deleting some storage locations doesn't help - surviving ones still catch the rollback
- Online time verification: the library verifies the system clock via HTTPS (port 443) against a pool of 12 trusted internet servers, randomly selected. Uses TLS certificate timestamps (CA-signed, can't be forged) and HTTP Date headers. Hosts file redirects fail because the TLS handshake rejects invalid certificates. Mandatory when anchors are missing, 10% random during normal operation. Silently skipped if offline.

Storage locations:

The library writes the anchor timestamp to two backends per platform plus a local cache file that is shared across platforms. Every path or key carries the per-license <tag> suffix described in the bullet list above.

On Windows:

```
HKCU\Software\RockyGuard\ta_<tag>          (registry value, REG_SZ)
%LOCALAPPDATA%\RockyGuard\.timeanchor_<tag> (per-user file)
```

On Linux:

```
$HOME/.local/share/rockyguard/.timeanchor_<tag> (per-user file)
$HOME/.lck_svc_<tag>                             (hidden in $HOME)
```

On both platforms, in addition to the above:

```
<directory of license.json>/.lck_cache_<tag> (local cache, next
                                              to the license file)
```

The dual-backend design means a savvy end user who deletes one location does not defeat the rollback check -- the surviving backend still catches the manipulation. The local cache (.lck\_cache\_<tag>) is the third witness and travels with the license file.

If clock manipulation is detected, the check returns `LicenseStatus::ClockManipulated`.

**IMPORTANT:** `license_id` and `product` are required fields. They are used to generate unique anti-tampering anchor identifiers. The library will print a warning if these fields are empty, and the CLI tools enforce them as required parameters.

### 9.3 Binary Integrity Self-Check (Shared Library / DLL)

For DLL builds, the library verifies its own binary on every validation:

- The library's SHA-256 hash is computed and compared against a signed hash file (`rockyguard.sig`) shipped alongside the DLL
- If the DLL has been patched or modified, the check fails
- The `.sig` file cannot be forged (signed with vendor's private key)

If detected, the check returns `LicenseStatus::IntegrityCheckFailed`.

Ship `rockyguard.sig` next to `rockyguard.dll` in your distribution.

Static builds do not have or need a `.sig` file. The library's code is compiled directly into your executable, so there is no separate binary for the integrity check to verify and the package's `lib/static/` directory ships no `.sig`. Customers shipping their applications statically linked against `rockyguard.lib` (or `rockyguard_mdd.lib` on Windows Debug) get the integrity properties of their own executable instead -- whatever code-signing or binary-attestation flow they apply to that `.exe` / ELF covers the RockyGuard code transitively.

### 9.4 Floating Server Authentication

The floating license protocol includes multiple security layers:

- Response signing: every server response is signed with the server's private key. The client verifies using the public key. Prevents server spoofing via /etc/hosts or DNS manipulation.
- Client request authentication (non-TLS mode): the server issues a cryptographic session secret at checkout time (delivered inside the signed response). The client includes an HMAC of each subsequent request using this secret. This prevents attackers from forging checkin/heartbeat requests to evict other users, even if they sniff the client\_id from network traffic.
- Nonce-based replay protection: each request and response includes a cryptographically random nonce. Prevents replay attacks.
- Cryptographic random: all client IDs and nonces are generated using OpenSSL RAND\_bytes (hardware entropy), not predictable PRNGs.
- Optional TLS: encrypts all traffic between client and server. HMAC authentication is ALWAYS required, even under TLS, because TLS authenticates the transport but not the application-level client identity. Without HMAC binding each request to a session-specific secret, a peer who learns another client's client\_id (which is not secret) could forge checkin / heartbeat requests over the same TLS channel.

**WARNING** - Without TLS, session secrets are sniffable on the wire.

Without TLS, session authentication tokens flow as plaintext over the network. Response signing still prevents server spoofing (the client verifies every response), but a PASSIVE listener on the same LAN segment can capture a client's session secret as it leaves the server, then forge checkin or heartbeat requests against that client's lease -- evicting the legitimate user, holding the seat indefinitely, or both.

For any deployment reachable from a network you do not fully control (Wi-Fi, cross-VLAN, public internet, anything not a wired LAN with controlled access), enable TLS by setting `tls_cert` and `tls_key` in the server config and `tls_ca_cert_path` on the client. The server logs a warning at startup when response signing is enabled but TLS is not.

## 9.5 Security Hardening

The library is compiled with OS-level security protections:

- ASLR: DLL base address randomized on each load
- DEP: Non-executable stack and heap
- Control Flow Guard (Windows) / CET (Linux): Validates all indirect function calls at runtime, preventing control flow hijacking

These are enabled automatically and require no configuration.

## 9.6 Threat Model

The table below summarizes the concrete attacks RockyGuard is designed to defend against, the mechanism each

defense relies on, and the residual risk that remains after the defense is applied. The "Residual risk" column is what an attacker can still attempt; the column "Mitigation" describes what your application or operations team should add on top of the library to close that gap.

Attack	RockyGuard defense	Residual risk and mitigation
License file tampering (editing dates, machine fingerprint, features inside the JSON)	Ed25519 / RSA-SHA256 signature over the payload, verified at load()	None for the file itself: any byte change invalidates the signature and load() returns SignatureInvalid. Residual: an attacker who steals your private key can sign anything; mitigation = keep private.pem on a secure machine; rotate keys per the versioning policy if compromise is suspected.
Keygen / counterfeit license generator	Ed25519 / RSA private key required to sign a valid payload; public-key-only verifier on the customer side	None: the verifier cannot produce a valid signature without the private key. Same private-key custody mitigation as above.
Patching or cracking the shipped binary (NOP-out license checks)	Binary integrity self-check (DLL/SO builds): rockyguard.sig is verified at load against the binary's SHA-256	Residual: an attacker who modifies BOTH the binary AND the .sig in coordinated fashion bypasses the integrity check. Mitigation: ship the application's own integrity check (code signing, executable signature) so the OS / loader detects pre-launch modification before RockyGuard gets a chance to run. Static-library builds embed the library code directly into your executable, raising the bar further.
Clock rollback (set the system clock back to before expiry)	Multi-location time anchors (file + Windows registry where applicable) cross-checked against current clock; HTTPS time-anchor pool consulted on first run and 10% of subsequent runs	Residual: a determined attacker on a fully air-gapped machine who deletes ALL anchor locations AND rolls the clock back gets one false-pass at first run (documented in Customer_API_Reference §6.6). Mitigation: ship to non-air-gapped customers when possible; for air-gapped fleets, issue shorter-duration licenses so re-issuance forces clock truth on a regular cadence.
MITM on the floating-server connection (intercept and forge checkout / heartbeat / checkin responses)	Optional TLS encryption (use_tls + tls_ca_cert_path); REQUIRED Ed25519 payload signature on every server response (server_public_key_pem)	Residual without TLS: passive sniffer can capture the session secret issued at checkout and forge subsequent HMAC-authenticated requests. Mitigation: enable TLS in production deployments; pin the server cert via tls_ca_cert_path. With TLS + server_public_key_pem set, the residual is essentially zero.
Server spoofing (rogue floating server pretending to be the real one)	Client verifies every server response against server_public_key_pem	Residual: if server_public_key_pem is left empty (Development-only configuration), the client trusts any server. Mitigation: never ship Development-only configuration to end users; embed the production server's public key in the client at build time.
Eviction / heartbeat forgery on the floating server (peer who knows another client's client_id evicts them)	Per-checkout session secret issued by the server; HMAC-authenticated heartbeat / checkin requests bound to that secret	Residual without TLS: session secret travels in cleartext during checkout response and a sniffer can capture it. Mitigation: enable TLS; the captured-secret window closes.
Hardware fingerprint forgery (running a license bound to one machine on another machine that mimics the original's MAC / CPU / disk / motherboard IDs)	Four independent component hashes; configurable threshold (default: 2 of 4 must match)	Residual: a sufficiently determined attacker can spoof MAC and CPU id; spoofing all four including motherboard serial is much harder. Mitigation: keep the default threshold of 2 unless customer hardware constraints force you to lower it; consider raising threshold to 3 for high-value licenses.



Attack	RockyGuard defense	Residual risk and mitigation
Memory dumping / runtime extraction of the loaded license object	Out of scope: a library cannot defend against attackers with kernel-level access to the host process	None: the License struct lives in process memory after load() succeeds. Mitigation: this is an operating-system / hosting-environment problem, not a library problem. RockyGuard accepts this; if the threat model includes attackers with kernel-level access, license enforcement at the application layer is the wrong defense.

## 9.7 Generation Counter Storage

Every successful `license_create` call increments a persistent counter so the library can enforce the end-user-license generation cap on your vendor license (the "End-user license generation limit reached" error in §10.3.2). The counter is HMAC-protected (tamper attempts force the value to 999999999, blocking further generation) and stored in three locations per platform; the library reads the maximum across all locations, so deleting some but not all does not reset the counter.

Scope:

The counter is per-vendor-license and per-user-on-this-machine. The storage tag is derived from your `vendor_license_id`, so different vendor licenses on the same machine have independent counters; copying counter files between vendor licenses does not work because the HMAC binds to the `volid`. Counters do NOT sync across machines or users -- each generation host counts independently.

Storage locations:

The library writes the counter to three backends per platform. The `<tag>` suffix is derived from your `vendor_license_id` (a SHA-256 hash, abbreviated to 12 hex characters); you'll see it on every storage path below.

On Windows:

```
HKCU\Software\RockyGuard\gc_<tag>      (registry value, REG_SZ)
%LOCALAPPDATA%\RockyGuard\.gencount_<tag> (per-user file)
<cwd>\.rg_gencount_<tag>                (file in license_create's
                                         current working directory)
```

On Linux:

```
$HOME/.local/share/rockyguard/.gencount_<tag> (per-user file)
$HOME/.rg_gc_<tag>                             (hidden file in $HOME)
<cwd>/\.rg_gencount_<tag>                       (file in license_create's
                                         current working directory)
```

Storage format (file-based locations):

```
<count>:<volid>:<hmac>
```

A single text line. `<count>` is a non-negative integer. `<volid>` is the `vendor_license_id`. `<hmac>` is HMAC-SHA-256 over `<count> + <volid>` with a per-path salt, base-16-encoded. Editing the count or removing the HMAC invalidates the file: the library treats it as a tamper attempt and forces the counter to 999999999.

How to inspect:

The supported way is to run the v1.2.1+ CLI flag:

```
tools/license_create --vendor-license vendor_license.json --show-count
```

This prints the current count, the vendor-license id, and the cap (with remaining budget if a cap is set, "unlimited" otherwise). The flag is read-only -- no side effects, no counter increment.

If you need to read a specific storage location directly (e.g., as part of an audit script), the count is the substring before the first ':' on the single-line file format. The `.rg_gencount_<tag>` file in the current working directory is the most accessible (no admin / registry access required to read it).

Local-cache file note:

The third location -- `<cwd>/rg_gencount_<tag>` -- means the counter file shows up in whatever directory you run `license_create` from. This is by design (the library wants a third independent witness on the same volume as the vendor license) but means a customer running `license_create` from version-controlled trees should add `.rg_gencount_*` to their `.gitignore`, or run `license_create` from a dedicated state directory, to avoid accidentally committing counter files.

## 10. TROUBLESHOOTING

### 10.1 Error Reference Table

The table below is the canonical lookup: every `LicenseStatus` enum value and every CLI tool error message your customers might see, mapped to the human-readable message text and the sub-section in this document where the cause and resolution are explained in full.

LicenseStatus / message	Where it surfaces	Detailed entry
SignatureInvalid	LicenseVerifier::load() / load_from_string()	10.2.1
HardwareMismatch	LicenseVerifier::check_node_locked()	10.2.2
Expired	LicenseVerifier::load() / check_expiry() / check_node_locked()	10.2.3
FeatureNotLicensed	LicenseVerifier::check_feature()	10.2.4
VersionMismatch (v1.2.1+)	LicenseVerifier::check_version()	10.2.5
MalformedFile	LicenseVerifier::load() (invalid JSON or missing fields)	10.2.1 (related)
LibraryNotInitialized	license_create / rg_floating_server CLI	10.3.1
GenerationLimitReached	license_create CLI	10.3.2
MachineNotAuthorized	license_create / rg_floating_server CLI	10.3.3
TierNotAuthorized	rg_floating_server CLI	10.3.4
ClockManipulated	LicenseVerifier::check_expiry() / load() / check_node_locked()	10.4.1
IntegrityCheckFailed	LicenseVerifier::load() / check_expiry() / check_node_locked() (DLL builds)	10.4.2
NoLicensesAvailable	FloatingLicenseClient::checkout()	10.5.1
ServerUnreachable	FloatingLicenseClient::checkout() / checkin()	10.5.2
MachineSeatLimitReached (v1.2.1+)	FloatingLicenseClient::checkout()	10.5.3
InGracePeriod	LicenseVerifier::load() / check_expiry() / check_node_locked()	Not an error: license is expired but still usable; check <code>grace_days_remaining</code> and warn the user.
Valid	Any verification call	Not an error: success.
NotYetValid	(none in v1.2.1)	Reserved enum value: present in the public header so a future release can add a "license not yet active" path without an ABI break. No shipped v1.2.1 code path returns it. If your code switches on every <code>LicenseStatus</code> , include this case (route it as you would <code>MalformedFile</code> ) to silence "unhandled enum" warnings.

The full enum definition and operator `bool()` semantics are in [Customer\\_API\\_Reference §4.2](#) (enum class `LicenseStatus`) and [§4.4](#) (struct `LicenseResult`).

Each entry below is structured as:

- **Error:** the literal message text the library or CLI tool prints.
- **Cause:** why that message was produced (the underlying condition).
- **Resolution:** the action you take to fix it.

## 10.2 License Errors

### 10.2.1 "License signature verification failed"

Cause: One of three things has gone wrong.

- The license file or its signature was modified after signing.
- The public key embedded in your application does not correspond to the private key that signed the license.
- The SignatureAlgorithm passed explicitly to LicenseVerifier differs from the one used at signing time. (v1.2.1+: the constructor's default is now SignatureAlgorithm::AutoDetect, which inspects the loaded key and picks the right algorithm automatically; this case only fires if you pass an explicit value that disagrees with the key.)

Resolution: Address the matching cause from the list above.

- Replace the license with the genuine signed copy from your license\_create run.
- Verify that PUBLIC\_KEY in your application is the exact PEM string from public.pem produced by license\_keygen alongside the private key used at signing.
- Drop the explicit SignatureAlgorithm argument and let v1.2.1+'s AutoDetect default pick it from the key. If you have a deliberate reason to pass an explicit value, make sure it matches the algorithm used at signing time.

### 10.2.2 "Hardware mismatch: N of M required components matched"

Cause: The license was issued against a different machine's fingerprint than the one currently running it. N (matched components) is below the license's fingerprint\_match\_threshold (M, default 2).

Resolution: Re-issue the license bound to the correct machine: have the end user run rg\_fingerprint on their machine and send you the resulting hash, then call license\_create --fingerprint-value "<that hash>". Alternatively, if the same machine had only a partial hardware change (e.g., a NIC swap or disk replacement), lower fingerprint\_match\_threshold via license\_create --threshold so fewer components are required to match.

### 10.2.3 "License has expired"

Cause: The current system time is past the license's expires\_at, and the grace period (if any) has also elapsed.

Resolution: Issue a new license with a later --expires date and ship it to the end user. If the end user's system clock is wrong (e.g., dead CMOS battery resetting the date), correcting the clock is sufficient; no re-issue is needed.

### 10.2.4 "Feature not licensed: <name>"

Cause: Your application called check\_feature("<name>") but that name is not in the license's features array.

Resolution: Re-issue the license with `--feature "<name>"` added (one `--feature` flag per feature). If the feature should not have required a license, remove the corresponding `check_feature()` call from your application instead.

## 10.2.5 "Version <X> not allowed by license version range <range>" (v1.2.1+)

Cause: Your application called `LicenseVerifier::check_version("<X>")` (Customer\_API\_Reference §6.7), and the version string you passed does not satisfy the license's `version_range` field. The range is set at issuance time via `license_create --version`, and supports two notations: glob form ("`3.`", "`3.1.`", "`3.1.5`", "`***`") and comparator form ("`>=3.0,<4.0,!<=3.5.0`", with comma-AND'd clauses). An empty `version_range` matches anything (the v1.2.0 default; see Customer\_API\_Reference §5.1).

The most common ways this surfaces:

- The end user upgraded your application to a major version newer than what their existing license allows (e.g., license issued for "`3.*`", application now reports "`4.0.0`"). This is the intended use of `version_range` -- gate paid major-version upgrades.
- The application passed the wrong `current_version` string (e.g., a build-tag suffix slipped through: "`3.1.0-rc1`" against a license that accepts "`3.*`"). Glob and comparator matching are strict; "`3.1.0-rc1`" is NOT "`3.1.0`".
- The license was issued with an overly narrow range by mistake (e.g., "`3.1.5`" exact when the intent was "`3.1.*`").

Resolution: Pick the matching cause from the list above.

- For a legitimate version-gated upgrade, either re-issue the license with a wider `--version` range (`license_create --version "3.*" -> "3.* || 4.*"` via two clauses, or simply "`***`" if you want to drop the gate entirely) or instruct the end user to stay on a supported version.
- If the `current_version` string is wrong, normalize it before passing to `check_version()` (strip build suffixes, take only the major.minor.patch).
- For a too-narrow license, re-issue with the corrected range. The `version_range` field is part of the signed license payload, so editing the JSON by hand will trip `SignatureInvalid` (§10.2.1) -- you must regenerate.

If you do not want runtime version enforcement at all, simply do not call `check_version()` in your application and leave `version_range` empty at issuance. The library does not consult `version_range` automatically; `check_version()` is opt-in.

## 10.3 Vendor License Errors

These errors appear when running the vendor-side CLI tools shipped in this package: `license_create` (issues end-user licenses) and `rg_floating_server` (Premium tier; runs the floating license server).

### 10.3.1 "Vendor license missing. Re-run with --vendor-license vendor\_license.json"

Cause: The CLI tool was invoked without the `--vendor-license` flag, so it has no vendor license to authenticate license

generation against. (In v1.2.0 this error read "Library not initialized. Call rockyguard::init()..."; renamed in v1.2.1 because the original wording named an internal API the customer does not call.)

Resolution: Re-run the tool with `--vendor-license vendor_license.json` (pointing at the `vendor_license.json` file your library vendor sent you). For the `rg_floating_server` variant of this error, set `vendor_license: <path>` in the YAML config file instead.

### 10.3.2 "End-user license generation limit reached"

Cause: Your vendor license carries a maximum number of end-user licenses you may issue, and the persistent generation counter (preserved across CLI tool runs and reboots) has hit that cap. The counter is HMAC-protected: deleting state files does not reset it. See §9.7 for the storage layout and `tools/license_create --show-count (v1.2.1+)` for a read-only way to inspect the count and remaining budget before issuance.

Resolution: Contact the library vendor to renew the cap, raise the limit, or upgrade your vendor license to a higher tier.

### 10.3.3 "This machine is not authorized for license generation"

Cause: Your vendor license is bound to one or more specific generation machines (by hardware fingerprint), and the machine you are running the CLI tool on is not in that set.

Resolution: Run `rg_fingerprint` on the new generation machine, send the hash to your library vendor, and request a re-issued vendor license that includes the new machine.

### 10.3.4 "Floating licensing requires a Premium tier library license"

Cause: `rg_floating_server` was invoked under a Basic-tier vendor license. Floating licensing is gated behind the Premium tier.

Resolution: Contact the library vendor to upgrade your vendor license from Basic to Premium.

## 10.4 Anti-Tampering Errors

### 10.4.1 "System clock manipulation detected"

Cause: The library compared the current system clock to multiple stored time anchors and detected a backward jump beyond the 1-hour tolerance. Three common ways this happens:

- The end user genuinely set the clock back to extend an expired license.
- The clock was set FORWARD at some earlier point (deliberately, or because of a dead CMOS battery showing a wrong year) while the application ran -- the library wrote anchors at that future time -- and then the clock was

corrected. The current correct time is now "behind" the future-dated anchor, even though the clock is right.

- A timezone change wrapped a date boundary, or anchors were corrupted on disk.

Resolution: Set the system clock to the correct current time first. If that alone resolves it, the rollback was straightforward.

If the system clock is correct but the error persists, the anchor was likely written at a future time during a previous forward-clock excursion. The recovery path is to delete the per-license anchor files at the storage locations listed in §9.2 and re-run the application while connected to the internet. With every anchor location empty, the library treats the situation as a first run: the mandatory online time-anchor verification fires (HTTPS to a rotating pool of trusted servers), the corrected clock is validated against that real time, and fresh anchors are written at the correct time. The license then verifies cleanly.

Concrete steps:

```
Windows -- delete HKCU\Software\RockyGuard\ta_<tag> (registry value),
          %LOCALAPPDATA%\RockyGuard\timeanchor_<tag>, and
          <directory of license.json>\.lck_cache_<tag>.
Linux   -- delete $HOME/.local/share/rockyguard/timeanchor_<tag>,
          $HOME/.lck_svc_<tag>, and
          <directory of license.json>/.lck_cache_<tag>.
```

The <tag> is the 12-hex-character abbreviation of SHA-256(license\_id + "|" + product + "|" + ...

Pre-condition: the host must be online for at least the first run after deletion, OR the user...

If the host cannot reach the internet at all (air-gapped fleets) and the corrupted-anchor state is real, the library cannot self-recover from this state by design -- contact your support so you can re-issue a license out of band. v1.3 will add active forward-jump self-repair so this manual recovery is not needed in the online case (Future\_Roadmap §3.10).

## 10.4.2 "Library integrity check failed"

Cause: The shipped library binary (rockyguard.dll on Windows, librockyguard.so on Linux) was modified, OR the integrity .sig file next to it is missing, stale, or does not match the binary's current bytes.

Resolution: Restore BOTH the library binary AND its .sig file from the original shipped copies (they must come from the same build). The .sig file is paired with the binary at build time; mixing a binary from one build with a .sig from another build will continue to fail this check.

## 10.5 Floating License Errors

### 10.5.1 "No floating licenses available"

Cause: All seats in the floating server's pool are currently held by other clients (max\_users reached).

Resolution: Wait for one or more existing clients to release their leases. The server's heartbeat-timeout path will reclaim leases from clients that have stopped sending heartbeats automatically. If all seats are legitimately in use, increase

max\_users in the floating server's configuration and restart the server.

### 10.5.2 "Cannot reach license server"

Cause: The FloatingLicenseClient could not establish a TCP connection (or, with use\_tls, a TLS handshake) to the configured FloatingClientConfig.server\_host : server\_port.

Resolution: Verify the server is running and listening on the expected port. Check network connectivity (ping, traceroute) and firewall rules between client and server. Confirm server\_host and server\_port in FloatingClientConfig point at the right host. With use\_tls = true, also confirm tls\_ca\_cert\_path is correct and the server's certificate is valid.

### 10.5.3 "machine seat limit reached" (LicenseStatus::MachineSeatLimitReached, v1.2.1+)

Cause: The floating server's max\_leases\_per\_machine\_id cap is set to N, and this workstation already holds N concurrent leases (from a mix of currently-running clients and any ghost leases still waiting to be reclaimed by the reaper). Distinct from "no licenses available": the global pool may still have seats free for OTHER machines, but this machine has hit its per-host ceiling.

Resolution: Close another running session on the SAME workstation and retry the checkout. If you know the prior client was force-killed (no graceful checkin) and you do not want to wait for the lease\_timeout reaper, restarting the server clears all leases. If legitimate concurrency on one machine genuinely exceeds the cap, ask the operator to raise max\_leases\_per\_machine\_id. Operator sizing guideline: set the cap above expected steady-state concurrency to absorb crash-recovery; a tight cap means a crashed client temporarily blocks one slot until the reaper fires.



## 11. GLOSSARY

---

A consolidated reference for terms used throughout this manual and the API reference. Defined inline elsewhere in the docs; consolidated here for readers who reach a section in isolation.

**Anchor file:** A small file that stores a UTC timestamp plus an HMAC over that timestamp, used by the clock-manipulation detector. Anchor files are unique per license (per-license tag derived from `license_id`, `product`, and `licensee`). See §9.2 for storage locations.

**Basic tier:** The lower of the two RockyGuard library tiers. Allows node-locked license generation and verification. Does not allow floating-license server / client. Configured by the vendor license file you receive from Rocky Software Inc.

**Client ID:** A cryptographically random UUID generated at construction time by `FloatingLicenseClient`. Identifies one in-process client to the floating server. Distinct from machine ID (one process: distinct `client_id`; one host running N processes: same `machine_id`, N different `client_ids`).

**End-user license:** The signed license file you (the library customer) issue to YOUR end users. Contains licensee name, product name, expiry, hardware fingerprint (for node-locked) or floating settings, feature list, and signature. Filename conventionally `license.json`. Generated with `license_create`.

**Feature flag:** A string identifier in the end-user license's features list. Your application calls `verifier.check_feature("export_pdf")` (or similar) to gate code paths. Names are arbitrary; you choose the taxonomy.

**Floating license:** A licensing model where N concurrent seats are managed by a server on the customer's LAN. Clients check out a seat at startup, send heartbeats while running, and release on exit. Premium tier only.

**Ghost lease:** A floating-server lease still held by the server because the client that owned it crashed without sending a checkin. Reclaimed by the reaper thread after the `lease_timeout` window (YAML key `lease_timeout`; C++ struct field `FloatingServerConfig::lease_timeout_sec`). Counts toward `max_leases_per_machine_id` until reclaimed.

**Grace period:** A configurable number of days after `expires_at` during which the license still verifies but with status `InGracePeriod` (not `Valid`). Lets your application warn the user without blocking access at the moment of expiry.

**Hardware fingerprint:** A "|" -joined string of four SHA-256 hashes -- MAC address, CPU id, disk serial, motherboard serial -- used to bind a node-locked license to a specific machine. Computed by `HardwareFingerprint::fingerprint()`. Hashes only; no raw hardware identifiers leave the host.

**Heartbeat:** A periodic POST sent by `FloatingLicenseClient` to the floating server to keep its lease alive. Default interval 60 seconds. If the server stops receiving heartbeats for the `lease_timeout` window (YAML key `lease_timeout`; C++ struct field `FloatingServerConfig::lease_timeout_sec`), the reaper reclaims the lease (it becomes a "ghost" briefly, then is dropped).

**Integrity self-check:** A SHA-256 hash over the rockyguard.dll / librockyguard.so binary, signed by the vendor private key and shipped as rockyguard.sig. Verified at runtime by the library to detect post-build modification of the DLL. Static builds have no .sig file (§3.1, §9.3).

**Machine ID:** The hardware fingerprint string sent by FloatingLicenseClient to the floating server on every checkout / heartbeat / checkin. Used by v1.2.1+ servers to enforce max\_leases\_per\_machine\_id. Multiple clients on one host share the same machine\_id but each has its own client\_id.

**Node-locked license:** A licensing model where the license binds to one machine via hardware fingerprint. Available on both Basic and Premium tiers. Verified offline; no server contact.

**Per-license tag:** The 12-hex-character abbreviation of SHA-256(license\_id + "|" + product + "|" + licensee). The "|" separators are part of the tag formula; omitting them yields the wrong hash. Used as the suffix on every anchor-file storage location so different licenses on one machine do not interfere. See §9.2.

**Premium tier:** The higher of the two RockyGuard library tiers. Allows everything Basic does, plus floating-license server / client. Configured by the vendor license file.

**Public key (verification):** The Ed25519 or RSA public key embedded in your end-user application as a string constant. Pairs with the private signing key used by license\_create. Safe to commit and ship; does not leak the ability to forge licenses.

**Private key (signing):** The Ed25519 or RSA private key used by license\_create to sign end-user licenses. Lives ONLY on your build machine. Never ships to end users. Never committed to public source control.

**Reaper / reaper TTL:** The floating server's background thread that drops leases whose last heartbeat is older than the lease\_timeout window. Runs on an interval of lease\_timeout / 4, minimum 1 second. (The YAML key is lease\_timeout; the matching C++ struct field is FloatingServerConfig::lease\_timeout\_sec. The "TTL" in informal usage refers to this same window.)

**Session secret:** A short random string the floating server returns on a successful checkout. Used by FloatingLicenseClient to HMAC subsequent heartbeat and checkin requests so a third party who learns a client\_id cannot forge eviction or heartbeat requests. Required regardless of TLS, because TLS does not authenticate application-level client identity.

**Time anchor:** Any of the persistent storage locations the clock-manipulation detector writes timestamps to. See "anchor file" and §9.2. Multiple anchors are written per license; surviving anchors detect rollback even if some are deleted.

**Vendor license:** The signed JSON file YOU (library customer) receive from Rocky Software Inc. that authorizes your use of RockyGuard for issuing end-user licenses. Required by license\_create (every end-user license issued must be authorized) and by rg\_floating\_server / FloatingLicenseServer (Premium-tier gating). NOT required by license\_keygen, which is a standalone tool that generates a fresh Ed25519 / RSA keypair before you receive any vendor license. NOT shipped to your end users; not needed by LicenseVerifier or FloatingLicenseClient. See §2.2.

**Vendor public key:** A public key embedded in the RockyGuard library that verifies the vendor license signature at vendor-tool startup. You do not control this key; it is set by Rocky Software Inc. and ships inside the compiled library.

**Worker thread / thread pool:** The floating server uses a fixed pool of worker threads to handle incoming client connections. Each worker handles one connection at a time. Configured by `thread_pool_size` (default 4).

## 12. PRODUCTION CHECKLIST

---

A copy-paste preflight list before you ship a RockyGuard-protected build to a customer. Each item links to the section that has the full discussion; this list exists so you do not miss any one of them. Run through every item once per release cycle, not just on the first deployment.

### 12.1 Build configuration

- [ ] Consumer compiled with the matching CRT variant: link `rockyguard.lib` for Release, `rockyguard_mdd.lib` for Debug. Use the per-config `$<IF:$<CONFIG:Debug>,...>` generator expression in your CMakeLists, or rely on the `IMPORTED_LOCATION_RELEASE / IMPORTED_LOCATION_DEBUG` mechanism the shipped examples/CMakeLists.txt template uses. (§3.1; v1.2.1+ feature.)
- [ ] Public key embedded as a `const char*` string literal in your binary. NOT loaded from a file at runtime; NOT fetched over the network. (See §3.3, end-user application path / point B.)
- [ ] `license_id` and `product` set on every issued end-user license. Empty values weaken clock-manipulation defense and emit a stderr warning at load time. (§9.2 paragraph after §9.2 anchor table.)
- [ ] Decision made on runtime version enforcement (v1.2.1+). If you issue licenses with `--version "3.*"` (or similar) AND you want the library to BLOCK applications outside that range, your end-user code calls `LicenseVerifier::check_version(YOUR_APP_VERSION)` after `check_node_locked()`. If `version_range` is purely informational, you do nothing -- the library does not consult `version_range` automatically. Pick one and document it in your release notes so you do not silently change behavior between releases. (See §3.3 end-user application path / point B; Customer\_API\_Reference §6.7; troubleshooting at §10.2.5.)

### 12.2 Runtime / threading

- [ ] `LicenseVerifier::load()` is called off the UI thread on a desktop GUI app. The online time-anchor verification can take tens of seconds worst case if the network is flaky. Wrap in `std::async` or your application's job queue. A non-blocking `load_async()` API is on the v1.3 roadmap. (Customer\_API\_Reference §6.2.)
- [ ] If the verifier is shared across threads, you have read the thread-safety contract (Customer\_API\_Reference §6 "Thread safety"): publish after `load()`, call the truly read-only methods (`is_loaded`, `license`, `check_feature`, `check_version`) freely from many threads, but treat `check_expiry()` AND `check_node_locked()` as single-writer (both perform filesystem / registry writes for clock-manipulation anchors -- `check_node_locked()` tail-calls `check_expiry()` on the hardware-pass path).
- [ ] `LicenseVerifier` constructor is wrapped in `try/catch` ONLY if you load the public key from an untrusted source. For the standard pattern (PEM as a string literal) the throw cannot fire and `try/catch` is unnecessary. (Customer\_API\_Reference §6.1.)

- [ ] If you use FloatingLicenseClient (Premium tier only), you have read its threading contract (Customer\_API\_Reference §8 "Thread safety"). The client is internally synchronized: checkout() and checkin() serialize on an internal mutex and are safe to call from any thread; is\_checked\_out() is an atomic read with no lock and is advisory. The destructor is NOT safe to run concurrently with in-flight checkout() / checkin() / is\_checked\_out() from another thread (standard C++ object-lifetime rule). Construct once, publish to whatever threads need it, stop using it before destroying it. For genuinely-concurrent checkouts from many threads on one host, prefer one client instance per thread.

## 12.3 Floating-server deployment (Premium tier only)

- [ ] Server has a vendor license file at the path named by vendor\_license: in the YAML config. Same file you use for issuing end-user licenses; treat it with the same care as your private signing key. (§5.2.)
- [ ] private\_key: set in the server config so every response is signed and the client can verify the server identity. Without it, a peer who reaches the server's port can mint fake responses (e.g., always "checked\_out") to a client. Required regardless of whether TLS is enabled. (§5.2 "Response signing and TLS".)
- [ ] The server's private\_key is a SEPARATE keypair from the license-signing key (the one you pass to license\_create --key). Reusing the license-signing key for server response signing is supported but collapses two distinct trust roles into one: a single compromise gives the attacker both license-minting and server-impersonation capabilities. Generate a dedicated server keypair with tools/license\_keygen and embed its PUBLIC key in your end-user binary as FloatingClientConfig::server\_public\_key\_pem. (§5.2 "Response signing and TLS".)
- [ ] tls\_cert: and tls\_key: set if the server is reachable over a network a third party could intercept (Wi-Fi, the public internet, anything not a wired LAN you control). On the client, set tls\_ca\_cert\_path to pin the server certificate. Without tls\_ca\_cert\_path the library logs a warning about MITM vulnerability. (§5.2 "Response signing and TLS".)
- [ ] max\_leases\_per\_machine\_id set ABOVE the steady-state concurrency you expect on one workstation. Ghost leases count toward the cap until the reaper TTL fires; a tight cap means a crashed client temporarily blocks one slot for the reaper window. (§5.2 "Per-machine seat cap".)
- [ ] log\_max\_bytes / log\_keep\_count appropriate for the deployment's disk budget. Defaults are 100 MiB / 5 archives = ~600 MiB worst case per log\_file path. Set log\_max\_bytes <= 0 only if your host has its own log-rotation pipeline. (§5.2 "Log rotation".)

## 12.4 Deployment artifacts (what ships to your end users)

- [ ] Static-link path: just your\_application.exe + license.json. NO rockyguard.lib, NO rockyguard.dll, NO rockyguard.sig, NO public.pem file (it's embedded in your executable). (§3.4.)
- [ ] Shared-link path: your\_application.exe + rockyguard.dll + rockyguard.sig + libcrypto-3-x64.dll + libssl-3-x64.dll + license.json, all in the same directory. The library refuses to load without the matching .sig. (§3.4 + §9.3.)
- [ ] Vendor license, private signing key, and the license\_create / license\_keygen tools from your customer zip

NEVER ship to end users. Stay on your build machine. (§3.4.)

## 12.5 Operational and security

- [ ] You have read the threat model and accepted the residual risks called out there (notably: an air-gapped attacker who deletes ALL anchor locations and rolls the clock back gets one false-pass at first run; memory-dump attacks are out of scope). (§9.6.)
- [ ] You have a key-rotation plan if your private signing key is ever compromised. The plan is generally: generate new keypair, ship a new application version with the new public key embedded, re-issue end-user licenses signed with the new private key, set an EOL date on the old version's license-acceptance window. (§9.1; Versioning Policy doc.)
- [ ] You know which build of RockyGuard your release ships against (track ROCKYGUARD\_VERSION\_STRING from version.h in your release notes). Helps when correlating customer-side reports with library changelogs. (Customer\_API\_Reference §2.)