RockyGuard Library - Documentation

Version 1.4.0

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

About this manual:

This document describes RockyGuard v1.4.0. Three inline markers are used: "(v1.4+)" for what arrived in v1.4.0, "(v1.3+)" for APIs, CLI flags, and behaviour added in v1.3.0, and "(v1.2.1+)" for what arrived in v1.2.1; anything unmarked has been present since v1.2.0. Every marked passage is a real, shipping part of the release named -- not a preview, and not a draft note copied ahead of itself.

v1.4.0 is one feature: every result a verification call returns now carries a stable, fine-grained error CODE alongside the coarse status it always carried. Nothing that worked in v1.3 behaves differently -- the code-to-status mapping reproduces exactly what v1.3.2 returned for the same input, so existing code needs no changes at all. What is new is that you can now tell apart conditions that used to share one status, branch on them, and show the user a remedy the library wrote. Start at 10.1.1 for the code reference; the recommended handling pattern is in 3.5 and the full catalog is in Customer_API_Reference sections 4.5 and 4.6.

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

This manual documents the v1.4.0 surface of the library. If your installed library is older, some of what is described here DOES NOT EXIST in your version -- calling it will fail to compile, or fail at the CLI argument parser. Read the inline markers against the version you actually have: passages marked "(v1.4+)" require v1.4.0, passages marked "(v1.3+)" require v1.3.0, and passages marked "(v1.2.1+)" require v1.2.1. Note that upgrading from v1.3.x to v1.4.0 requires a RECOMPILE and not merely a new shared library: LicenseResult gained three members, so the struct changed size. Nothing was reordered, so your source needs no edits -- but a v1.3-compiled binary must not be pointed at the v1.4 DLL. To check what you have linked, read ROCKYGUARD_VERSION_STRING from <rockyguard/version.h>.

If you hold a v1.2.1 release, this is not your manual. The v1.2.1 manuals are frozen under docs/archive/v1.2.1/, and the copy packaged inside your customer zip matches your binary exactly -- prefer that one. The website (rockyguard.dev) publishes only the latest release, so a search result can easily land you on a newer manual than your build.

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.

The general rule, whatever version you are on: the manual to trust is the one that came inside your customer zip under docs/Customer_Documentation.pdf, because it was packaged at the same time as the binary you have. Reading a newer manual against an older binary produces "function does not exist" or "unrecognized argument" errors that are easy to misread as bugs.

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 version-marker convention -- "(v1.3+)" and "(v1.2.1+)" -- 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 "(v1.3+)" or "(v1.2.1+)" describes behavior present since v1.2.0. A reader fresh to v1.3.0 can ignore the markers entirely and read the manual as a unified description of the current library. A reader upgrading from an earlier release can scan only the marked sections to see what is new, matching the marker to the version they are coming from, without re-reading the rest.
- NOT ALL CROSS-REFERENCED DOCUMENTS SHIP IN YOUR PACKAGE. Your docs/ directory contains exactly three files: this manual, the Customer_API_Reference, and the License Agreement. The following are cited in places and are VENDOR-INTERNAL -- do not go looking for them in the zip, and ask support if you need what they cover:

RockyGuard_Future_Roadmap.txt	deferred / post-v1.3 items
RockyGuard_Versioning_Policy.txt	SemVer + ABI policy
RockyGuard_v1_3_Plan.txt	internal release planning
docs/archive/, docs/archive/v1.2	.1/ prior-release doc snapshots

```
Citations to them are kept because they name where a decision is
recorded, which is useful when raising a support question. A
citation is not a promise that the file is in your hands. The
substance you actually need is in the three shipped documents; if
you hit a case where it is not, that is a documentation bug and
worth reporting.
```

- "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 evaluating RockyGuard. You receive a vendor license from Rocky Software Inc., generate end-user licenses on your build machine, and ship a RockyGuard-protected application to your end users.

YOUR END USER. The person whose machine runs your application. They never see RockyGuard directly. They receive an end-user license file plus your RockyGuard-protected executable. Their machine's hardware fingerprint is what node-locked licenses bind to.

ROCKY SOFTWARE INC. The library vendor (us). We sign your vendor license. After delivery we are not in the runtime path -- the RockyGuard library never phones home.

The three "licenses":

Vendor license (vendor_license.json) -- what YOU receive from Rocky Software Inc. Authorizes you to issue end-user licenses. Required by license_create and rg_floating_server. Stays on your build machine; never ships to your end users. See §2.2.

End-user license (license.json) -- what YOU issue to YOUR end user. Carries their machine's fingerprint (node-locked) or floating settings, plus expiry, features, and your signature. Your application's RockyGuard verifier reads this on their machine. See §4.

Floating license (a seat in a pool) -- a Premium-tier model in which one end-user license has type=floating and represents N concurrent seats served by a floating server. Individual end-user processes check out a seat at startup and release it on exit. See §5.

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:

- 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 x86_64 (Intel), macOS 11 "Big Sur" or newer. Shipped as rockyguard-vX.Y.Z-macos-x64-customer.zip. The binary statically links libc++ and OpenSSL, so it has NO Homebrew or other third-party runtime dependency: otool -L lists only /System frameworks and /usr/lib. On Apple Silicon the x86_64 build runs under Rosetta 2; a native arm64 build is available from the vendor on request. macOS-specific integration details are in the callout below.

macOS BUILD NOTES (read these if you are on the macOS package):

- The CLI tools have no filename extension (license_verify, not license_verify.exe). The shared library is librockyguard.dylib; the static library is librockyguard.a; there is no .lib import library.
- Static link (recommended): librockyguard.a + deps/lib/libssl.a + deps/lib/libcrypto.a (ssl before crypto) PLUS the frameworks "-framework IOKit" "-framework CoreFoundation" -- the macOS equivalents of Linux's pthread + dl. The bundled examples/ CMakeLists.txt already does this.
- Shared link: librockyguard.dylib has install name @rpath/librockyguard.dylib and OpenSSL statically inside it. There is no LD_LIBRARY_PATH on macOS: give your binary an rpath (CMake INSTALL_RPATH "@loader_path" and co-locate the .dylib, or point the rpath at lib/shared/) so it resolves the library at runtime. Ship librockyguard.sig next to the .dylib.
- Self-containment check: otool -L <your binary> must show only /System frameworks and /usr/lib -- never /usr/local/opt/openssl or /opt/homebrew. If it does, the link picked up a system OpenSSL.
- Building the examples: cmake -B build -S . && cmake --build build. The default generator is single-config Make, so --config Release is ignored and the binaries land directly in build/<name> (not build/Release/<name>).
- Hardware fingerprint sources: MAC from IOKit (en0/en1), CPU ID from the machdep.cpu.brand_string sysctl, disk serial from the IOKit NVMe/AHCI Device Characteristics, motherboard UUID from IOPlatformUUID. State/anchor files live under ~/Library/Application Support/RockyGuard/. No root or extra entitlements are required.

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

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

2. GETTING STARTED

2.1 Package Contents

The package layout is the same on every platform; the listing below is the Windows zip. The macOS and Linux zips differ only in the platform-specific names their own READMEs describe: the CLI tools have no .exe suffix, the shared library is librockyguard.dylib (macOS) or librockyguard.so (Linux) rather than rockyguard.dll, there is no .lib import library, and OpenSSL is bundled as static archives (deps/lib/libcrypto.a + libssl.a) with no separate runtime DLLs to ship. See the macOS BUILD NOTES in section 1 for the macOS specifics.

The Windows customer zip (rockyguard-v1.4.0-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.
	license_asset.h	LicenseAsset: opens an asset sealed under a key the licence yields , with no verdict to branch on -- the load-bearing pattern (v1.4+). Se e §9.11 and §12 of the API reference.
	hardware_fingerprint.h	Machine hardware fingerprinting.
	floating_client.h	Floating license client (Premium).
	license_error.h	Error-code catalog: LicenseErrorCode, error_info(), the stable code names (v1.4+). See §10.1.1.
	diagnostics.h	The Licence Inspector as an API: collection, sealing, the structured health check, and root-cause diagnosis (v1.4+). See §11 of the API reference.
	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 bu ild (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 (CLI).
	rg_fingerprint_gui.exe	Same fingerprint output, in a GUI window with a Copy button + auto -clipboard. Ship this to non-technical end users who would double-c lick an .exe rather than open a terminal. Windows only. v1.3+.
	rg_floating_server.exe	Floating license server (Premium).
	floating_server_config.yaml	Sample server config file.

Path	File	Description
	rg_floating_client.exe	Floating license client smoke-test binary (Premium).
	rg_inspect.exe	Licence Inspector: collects one sealed diagnostic report on an end user's machine for them to send you. Hand this to end users when licensing fails. v1.4+; see §7.7.
	rg_bind_asset.exe	Build-time asset binder for load-bearing licences: seals an asset under a data key and emits a C header of ciphertext for your build. v1.4+; see §7.8.
	rg_bind_asset.cpp	Source of the binder, so you can rebuild it against deps/ alone (it links nothing of ours).
	rg_bind_lint	Python script: lints your source for integrations that collapse the load-bearing pattern back into a flippable boolean. v1.4+; see §7.9.
	libcrypto-3-x64.dll	OpenSSL runtime for the tools above. A second copy of the same DLL that appears under lib/shared/; the tools load it from their own directory, so keep both next to the .exe files if you relocate them.
	libssl-3-x64.dll	As above. Both are required -- the tools fail to start with STATUS_DLL_NOT_FOUND (0xC0000135) and no message if either is missing.
portal/	start.bat / start.ps1 / start.sh	Launchers for the self-hosted licensing web app described in §4.8. OPTIONAL -- everything the portal does is license_create underneath (§4.3), so you can delete this directory outright if your team uses the CLI.
	run.py	Entry point the launchers call; run it directly if you prefer to manage the Python environment yourself.
	app/	FastAPI application (routes, templates, static assets).
	wheels/	Pre-downloaded Python dependency wheels, so first run works on a machine with no PyPI access. Platform-specific: the Windows zip carries win_amd64 wheels, the Linux zip manylinux2014_x86_64.
	requirements.txt	Dependency pin list matching wheels/.
	README.md	Portal setup and daily workflow. Start here rather than with §4.8 if the portal is all you need.
deps/	include/openssl/	Bundled OpenSSL headers. The exact version is stated in deps/THIRD-PARTY-NOTICES.txt and in the OPENSSL_FULL_VERSION_STR macro in include/openssl/opensslv.h. As of v1.3.0 the bundled version is OpenSSL 3.6.1.
	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).
	OPENSSL-LICENSE.txt	The Apache License 2.0, under which the bundled OpenSSL is distributed. You must carry this file, or its text, with any application you ship that redistributes OpenSSL -- which the shared-link path in §3.4 requires. See §3.4 "Third-party attribution".
	THIRD-PARTY-NOTICES.txt	Attribution for every bundled third-party component, with the exact version and what the licence asks of you as a redistributor. Generated at packaging time from the headers actually in this zip, so it cannot disagree with them.
examples/	CMakeLists.txt	Build script for the examples.
	node_locked_example.cpp	Node-locked license verification example.

Path	File	Description
	hidden_key_example.cpp	Node-locked example with the public key embedded as split/masked tables (no plain PEM blob in the binary; see §9.9). v1.4+.
	hidden_public_key.h	The masked key hidden_key_example uses -- a PLACEHOLDER built from the sample key; regenerate from your own key with embed_public_key.py.
	embed_public_key.py	Generator: turns a PEM public key into hidden_public_key.h. Re-run it to rotate the embedded key.
	rg_floating_client.cpp	Floating license client example source.
	health_check_example.cpp	The diagnostics API in an application: a cheap startup self-check, and the off-thread pattern for a "Help -> Diagnose Licence" menu item (v1.4+).
	loadbearing_asset_example.cpp	The load-bearing licence pattern end to end: --encrypt seals an asset at build time; at runtime LicenseAsset opens it with the key the licence yields and the feature runs on the bytes -- or on nothing (v1.4+). See §9.11.
	bound_asset.h	Header-only sealer behind the example's --encrypt and tools/rg_bind_asset. Build-time only -- your application opens assets with LicenseAsset, not this header. v1.4+.
	FEATURES.template.md	Feature-flag register template. Copy it to your repository root as FEATURES.md: the one place your code, your build (rockyguard_bind_asset) and licence minting (license_create --features / --wrap-feature-key) agree on every flag string. v1.4+.
	inspector/	Source for rg_inspect and rg_report_read, with their own CMakeLists.txt and README.txt, so you can build both with YOUR keys compiled in. rg_report_read is the support-desk half that opens the reports your end users send; only its SOURCE ships, never our build of it. v1.4+.
	banner.h	Shared header that the .cpp examples #include for their startup banner; copied into the package so the examples build standalone.
	README.txt	How to configure and build the two examples on their own, separately from your application.
cmake/	RockyGuardBindAsset.cmake	rockyguard_bind_asset(): one CMake call that runs rg_bind_asset at build time and hands your target the generated ciphertext header. v1.4+; see §7.8.
docs/	Customer_Documentation.txt/pdf	This document.
	Customer_API_Reference.txt/pdf	API reference.
	RockyGuard_License_Agreement.txt/pdf	The licence agreement governing your use of the library. Binding regardless of whether the portal or the CLI issues your end-user licences.
(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).
(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-v1.3.0-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 MINUS rg_fingerprint_gui (which is Windows-only in v1.3): license_keygen, license_create, license_verify, rg_fingerprint, rg_floating_server, rg_floating_client, rg_inspect, rg_bind_asset -- plus the same rg_bind_asset.cpp source and rg_bind_lint script, which are identical on both platforms. No OpenSSL runtime files here either -- they are statically linked.
portal/	start.sh	In place of start.bat / start.ps1. Same application; wheels/ carries manylinux2014_x86_64 wheels rather than win_amd64.
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.
deps/	OPENSSL-LICENSE.txt	Same file as the Windows zip. Note that it still applies on Linux even though no OpenSSL file ships beside your binary: OpenSSL is linked INTO librockyguard.so and into anything you static-link against librockyguard.a, so you are distributing OpenSSL code either way. See §3.4 "Third-party attribution".
	THIRD-PARTY-NOTICES.txt	Same file, generated per platform; its shipping note is worded for the Linux linkage described above.

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 links librockyguard.dylib directly too (no import library); give your binary an @rpath pointing at lib/shared/ rather than using LD_LIBRARY_PATH -- see the macOS build notes in §1.

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 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. RENAME IT TO vendor_license.json, or substitute your filename in every command that follows. Every example in 2.4, 3.3, 4.3, 5.1, 7.3, 9.7 and the portal walkthrough hardcodes vendor_license.json; renaming once is less error-prone than editing each command. 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 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.

Once it arrives, 2.3 explains where it is consumed and what your tier grants -- you do not need the vendor license in your shipped application, only in the tools that GENERATE licenses.

2.3 What Your Vendor License Does

2.2 covered how the file reaches you. This section covers what it is FOR: which of your programs need it, which do not, and what your tier changes. If you already have vendor_license.json in hand and only want to know where it plugs in, start here.

(These two sections used to overlap: 2.3 opened by re-announcing that you should have received the file, which reads as a restart rather than a continuation. The procurement detail lives in 2.2 and is not repeated here.)

```
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

Yes, Basic is the LESS restricted of the two on that middle column, and it is not a typo. The generation counter is a vendor-side control on how many end-user licenses you may mint, and it is applied where the commercial agreement calls for it -- typically Premium seat-pool deals, which are priced per concurrent seat. Basic is priced without such a cap, so none is enforced. Tier governs which FEATURES you get (floating licensing is Premium-only); it is not a single ranked axis where Premium is uniformly more permissive. See EULA 2.2(c).

Mechanically the cap is not the tier at all. It is one field on your vendor licence, max_end_user_licenses, enforced by license_create whenever it is greater than zero and ignored when it is zero; the tier never switches it on or off. "Basic:

unlimited" means Basic licences are issued with that field at zero. So do not infer your budget from the word Premium in either direction -- read it: license_create --show-count --vendor-license vendor_license.json prints the cap, the number minted so far and the remaining budget (§9.7), and the portal's dashboard card shows the same. When the budget runs out license_create refuses with GenerationLimitReached (§10.3.2); it does not degrade or warn beforehand, so check the count before a batch mint.

2.4 Quick Start

PATH SEPARATORS, AND ONE SHELL THAT WILL NOT ACCEPT THEM. Every command in this manual is written with forward slashes (tools/license_create). That form works in PowerShell, in Git Bash, and on Linux, which is why it is used here -- this manual serves both platforms and a backslash would break the Linux reader.

It does NOT work in cmd.exe. Pasting

```
tools/license_create --key private.pem ...
```

into a classic Command Prompt fails, because cmd.exe does not accept forward slashes in the program path. If you use cmd.exe, either switch the separators:

```
tools\license_create.exe --key private.pem ...
```

or run the commands from PowerShell instead, which is the default terminal in Windows 11 and needs no change. This affects only the path to the tool; the flags are identical in every shell.

WHERE THESE COMMANDS RUN, AND WHERE THEIR OUTPUT LANDS. Every path below is RELATIVE TO YOUR CURRENT WORKING DIRECTORY, and none of these tools writes to the package directory. Pick a working directory you control -- not inside the extracted zip -- and stay in it for all three steps:

```
cd C:\licensing\myproduct        (any directory you own)
```

SET ONE VARIABLE FIRST, AND EVERY COMMAND IN THIS MANUAL WILL WORK FROM THERE. Because your working directory is deliberately NOT the extracted package, a bare tools/license_keygen cannot resolve -- there is no tools/ directory where you are standing. Point a variable at the package once:

```
PowerShell:   $RG = "C:\rockyguard-v1.3.0-windows-x64-customer"
cmd.exe:      set RG=C:\rockyguard-v1.3.0-windows-x64-customer
bash / Linux: RG=$HOME/rockyguard-v1.3.0-linux-x64-customer
```

substituting wherever you actually unzipped it. Then every command that this manual writes as tools/<something> is run as:

```
PowerShell:   & "$RG\tools\license_keygen" --private private.pem ...
cmd.exe:      "%RG%\tools\license_keygen.exe" --private private.pem ...
bash / Linux: "$RG/tools/license_keygen" --private private.pem ...
```

THIS CONVENTION APPLIES THROUGHOUT THE MANUAL, not only to the Quick Start: sections 3.3, 4.1, 4.3, 4.4, 5.1, 8.2 and 9.7 all print tools/... in the same shorthand, and all of them mean "the tools directory inside your extracted package". The Quick Start below spells the prefix out in full; later sections use the shorthand to keep the commands readable. If a command fails with "not recognized as the name of a cmdlet" on Windows or "No such file or directory" on Linux, this is why -- you are missing the prefix, not the tool.

(Until 2026-08-14 this paragraph said "invoke the tools by path into the package, as the examples do" and then printed the bare form, which does not do that. Two independent first-time readers pasted step 1 verbatim from the directory this section tells you to create and got "not recognized" / exit 127; one of them then ran it from the package root instead,

which works but drops your private signing key inside the extracted zip -- the thing the paragraph above warns against. The convention is now stated rather than implied.)

With that cwd, step 1 writes private.pem and public.pem there, step 2 writes license.json there (license_create's default --output; see 7.3), and the anti-rollback state files described in 9.7 are written under your per-user application-data directory rather than the cwd as of v1.3. Keeping one stable working directory matters because private.pem from step 1 is the input to step 2, and losing track of which directory it landed in is the most common way this sequence goes wrong.

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):

```
"$RG/tools/license_keygen" --private private.pem --public public.pem
```

(PowerShell: prefix with & . cmd.exe: "%RG%\tools\license_keygen.exe". See the paragraph above if you have not set $RG yet.)

Keep private.pem SECRET. You will embed public.pem in your application. Note that both keys land in your CURRENT directory -- the one you cd'd to above -- and not in the package.

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):

```
"$RG/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() {
    // CONSOLE-APP SHAPE -- read this before pasting it into a GUI.
    // load() and check_node_locked() below are SYNCHRONOUS, and both
    // can block for roughly 9 seconds or more (three HTTPS
    // time-check hosts at 3s each, plus OS DNS resolution) when the
    // machine is offline or sitting behind a captive portal. See
    // section 9.2. Blocking main() in a console tool is harmless. In
    // a desktop application, do this on a background thread and show
```

```
    // your window first -- otherwise the app looks hung on first
    // launch, which is exactly when a new customer is deciding
    // whether your software works.
    //
    // The constructor is the only verifier call that throws
    // (std::runtime_error on an unparseable PEM or an unsupported
    // key type). Uncaught, that is std::terminate() -- a hard crash
    // rather than a message. A mis-pasted PUBLIC_KEY literal is the
    // usual cause, so guard it even when the key is embedded.
    // LicenseVerifier is not copyable or movable, hence optional +
    // emplace(); this also keeps the body below un-indented.
    // See Customer_API_Reference section 6.1.
    std::optional<rockyguard::LicenseVerifier> maybe_verifier;
    try {
        maybe_verifier.emplace(PUBLIC_KEY);
    } catch (const std::runtime_error& e) {
        std::cerr << "License system error: " << e.what() << std::endl;
        return 1;
    }
    rockyguard::LicenseVerifier& verifier = *maybe_verifier;

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;
}
```

Hardening the embedded key (optional). The PUBLIC_KEY literal above is the simplest form and is correct, but it leaves the key as a contiguous block of text in your binary that strings yourapp | grep -A2 "BEGIN PUBLIC KEY" finds in one command. The key is not secret -- a public key never is -- so an attacker READING it gains nothing. The concern is that a plain block is a clean target to OVERWRITE: replace it with the attacker's own public key and your app will happily verify licenses they signed themselves. To remove that target, the examples/ folder ships hidden_key_example.cpp with embed_public_key.py: run the generator on your public.pem to produce hidden_public_key.h, which stores the key as two masked byte tables reconstructed at runtime instead of a plain string. No PEM block appears in your binary, and the reconstruction cross-checks itself so a substitution has to patch both tables at once. This is a deterrent that raises attacker cost, not a guarantee (§9.9); it is most effective compiled under an obfuscator. If you do nothing, the plain literal

above is still a supported, working integration.

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
```

On macOS and Linux the default generator is single-config, so --config Release is ignored and the built examples land directly in build/ (for example build/node_locked_example); on a Windows Visual Studio build they are in build/Release/. The run commands below use the bare example name -- run them from the directory that holds the binary, or prefix the path shown above.

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>
```

hidden_key_example:
The same node-locked flow, but the public key is embedded as split/masked byte tables (via embed_public_key.py -> hidden_public_key.h) instead of read from a file -- so no plain PEM block is left in the binary as a substitution target (§9.9). The key is embedded, so it takes only the license path. Run as:

```
hidden_key_example <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]
```

loadbearing_asset_example:
The load-bearing licence pattern (§9.11) end to end. Seal any file under a data key at build time, then open it at runtime with the key the licence yields -- and watch the feature run on the bytes with a genuine licence and on nothing with a foreign one. Run as:

```
loadbearing_asset_example --encrypt <key_hex> <in> <out.enc>
loadbearing_asset_example --pubkey <public_key.pem> --license <license.json> --asset <out.enc> [--feature
    <name>]
```

Without --feature the asset is opened with the product data key (license_create --wrap-data-key); with --feature, with that

feature's own key (--wrap-feature-key <name>=<hex>). Generate keys with license_create --gen-data-key. Exit status is 0 when the asset opened and 3 when it stayed sealed -- the latter is the expected result of the wrong-licence negative test. The examples build also produces rg_bind_asset (§7.8), the build-time binder behind cmake/RockyGuardBindAsset.cmake.

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 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-v1.3.0-windows-x64-customer/ on Windows or rockyguard-v1.3.0-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-v1.3.0-windows-x64-customer/.
Read rockyguard-v1.3.0-windows-x64-customer/AI_INTEGRATION_GUIDE.md
and follow it.
```

Replace the folder name with whatever your extracted zip actually produced. If you extracted into a subdirectory of your repo (downloads/, third_party/, or anywhere else), prepend that path so the agent knows where to look -- e.g. third_party/rockyguard-v1.3.0-windows-x64-customer/.

- 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 downloading and extracting it) before it touches your build files. The guide's §2 covers placement: it tells the agent to match your project's existing convention for vendored dependencies rather than invent one.

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: a per-user, user-writable location such as %APPDATA%\YourCompany\YourProduct; next to the executable only when that directory is user-writable -- see the writability rule in §3.4). 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.
- 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 shipped library is compiled as C++17, and your consumer project must be built with a C++17 (or later) standard so that your translation units agree with it on the standard library types crossing the boundary. 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 naming the exact fix -- no template error cascade. That guard lives in include/rockyguard/export.h, which every other public header reaches, so it fires whichever header you include first and whether or not you go through the rockyguard.h convenience header. The one deliberate exception is version.h, which includes nothing and carries only macros, so build scripts and standard-agnostic code can read your library version without tripping the guard.

(Correcting an earlier version of this paragraph: it said the public headers "use std::optional, std::variant, std::string_view, structured bindings, and inline variables". They do not -- those features are used in the library's own implementation, not in its interface, and a reader who checked the headers for them found nothing and could reasonably conclude the requirement was stale. The requirement is real; the reason given for it was wrong. Separately, until v1.3.0 the guard sat only in rockyguard.h, so including an individual header under a pre-C++17 standard compiled silently. Both were found by the Phase 4.1 v1.3.0 onboarding dry-run.)

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 iphlpapi ole32 oleaut32 wbemuuid advapi32 shell32)
```

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.

The DLL has no Debug variant, and that is the case the two static libraries can lull you into forgetting. rockyguard.dll is built /MD only. A Debug (/MDd) consumer links against its import library without any LNK2038 -- import libraries carry no CRT-mismatch record -- and then exchanges std::string, std::vector and std::map objects with the DLL (LicenseResult, License, every by-value return) whose layout differs between the two CRTs: on x64 with the shipped toolchain a std::string is 40 bytes under /MDd and 32 under /MD, a vector 32 against 24, a map 24 against 16. Nothing fails to build; the process corrupts memory or crashes at the first call. So on the shared path build your Debug configuration EITHER against the static rockyguard_mdd.lib (select it per-config with the same generator-expression pattern as above, and define ROCKYGUARD_SHARED_LIB only for the configurations that link the DLL -- a dllimport declaration against a static library is its own mismatch) OR with the Release CRT (CMAKE_MSVC_RUNTIME_LIBRARY MultiThreadedDLL for that configuration). A Debug-CRT DLL is recorded on our roadmap; until it ships, a /MDd host must not load rockyguard.dll.

Since v1.4.0 you do not have to remember this: <rockyguard/export.h> fails the build with an explanatory #error when a translation unit is compiled with the Debug CRT while ROCKYGUARD_SHARED_LIB is defined. The guard does not fire on the static path, and if you have built your own Debug rockyguard.dll from source you can define ROCKYGUARD_ALLOW_DEBUG_CRT_WITH_DLL to suppress it.

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 iphlpapi ole32 oleaut32 wbemuuid advapi32 shell32)
```

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)
```

NOT AVAILABLE TO YOU: add_subdirectory(). Earlier revisions of this section offered a third variant --

```
add_subdirectory(path/to/rockyguard)      # DOES NOT WORK HERE
target_link_libraries(your_app PRIVATE rockyguard)
```

-- described as letting the library's own CMake target export its transitive dependencies so you need not list them. That is true of the library's SOURCE tree, which is not what you have. Your package ships headers and prebuilt binaries; there is no CMakeLists.txt and no source to add_subdirectory() into, so this variant cannot resolve and there is no directory in the zip to point it at. It is documented here only because it appeared in prior revisions and is the variant a newcomer naturally reaches for -- it is the one that promises to spare you the manual link line.

If you want that ergonomics without a source tree, copy examples/CMakeLists.txt: it defines an IMPORTED target named rockyguard that carries the include directories, both OpenSSL libraries and the platform libraries as INTERFACE properties, so your own targets get them transitively from a single target_link_libraries(your_app PRIVATE rockyguard). That file is the supported equivalent, and it is the same file that builds the shipped examples, so it is known to work.

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(dllimport).
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.

RELEASE CRT ONLY. rockyguard.dll is built /MD and there is no /MDd DLL. A Debug (/MDd) consumer links its import library cleanly -- no LNK2038, because import libraries carry no CRT-mismatch record -- and then passes std::string / vector / map objects across the boundary in a different layout (see the CRT VARIANTS note in §3.1 for the measured sizes). That is memory corruption, not a link error. Build your Debug configuration against the static rockyguard_mdd.lib, or compile it with the Release CRT; do not load rockyguard.dll from a /MDd process. As of v1.4.0 this is enforced at compile time rather than left to the reader: including any RockyGuard header with ROCKYGUARD_SHARED_LIB

defined under the Debug CRT is a build error naming both remedies. ROCKYGUARD_ALLOW_DEBUG_CRT_WITH_DLL suppresses it for anyone who has built their own Debug DLL.

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 "$ORIGIN/../lib/shared"
    BUILD_WITH_INSTALL_RPATH TRUE)
# ...or ship your_app and librockyguard.so in the same directory
# and set INSTALL_RPATH "$ORIGIN" instead, with the same
# BUILD_WITH_INSTALL_RPATH TRUE line.
```

BUILD_WITH_INSTALL_RPATH TRUE IS REQUIRED, NOT OPTIONAL, AND ITS ABSENCE FAILS IN A WAY THAT PASSES YOUR OWN TESTING. CMake applies INSTALL_RPATH only to the binary produced by cmake --install. Without BUILD_WITH_INSTALL_RPATH TRUE, the binary in your build tree instead receives an automatic absolute RPATH pointing at wherever librockyguard.so sat when you linked -- typically the directory you unpacked this package into.

That binary then works on your machine and only on your machine. Copying it next to librockyguard.so in a deployment directory does not help and, worse, appears to: the absolute RPATH still resolves, because your extracted package is still where it was, so ldd shows the library found and the program prints its licence status normally. On an end user's machine, where that path does not exist, the dynamic loader fails before main() runs:

```
./your_app: error while loading shared libraries: librockyguard.so:
cannot open shared object file: No such file or directory
```

Exit status 127, no output of your own, and no RockyGuard diagnostic -- the library never gets control, so nothing in Section 10 applies. This is the Linux counterpart of the Windows STATUS_DLL_NOT_FOUND (0xC0000135) failure described above, and it is harder to catch because the Windows case fails on the developer's machine too.

Verify before you ship, from your build tree:

```
objdump -x build/your_app | grep -E 'RPATH|RUNPATH'
```

You want to see $ORIGIN. An absolute path containing your own home or build directory means BUILD_WITH_INSTALL_RPATH is missing, and the binary is not portable.

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() {
    // Guard construction: it is the only verifier call that throws.
    // See §2.4 and Customer_API_Reference §6.1 for why this matters
    // even when PUBLIC_KEY is an embedded literal.
    std::optional<rockyguard::LicenseVerifier> maybe_verifier;
    try {
        maybe_verifier.emplace(PUBLIC_KEY);
    } catch (const std::runtime_error& e) {
        std::cerr << "License system error: " << e.what() << std::endl;
        return 1;
    }
    rockyguard::LicenseVerifier& verifier = *maybe_verifier;

auto result = verifier.load("license.json");
// One structured line per outcome, pass or fail (v1.4+). This is
// what you want in your log: result.code is a stable contract,
// whereas result.message is prose we may reword. See 3.7.
your_logger(result.to_json());  // see 3.5
if (!result) {
    std::cerr << result.message << std::endl;
```

```
    return 1;
}

result = verifier.check_node_locked();
your_logger(result.to_json());  // see 3.5
if (rockyguard::error_info(result.code).severity ==
        rockyguard::DiagnosticSeverity::Warning) {
    // Passing, but tell somebody (v1.4+). Warning severity only
    // ever sits on a result that converts to TRUE, so this branch
    // must not refuse to start. Testing severity rather than
    // status == InGracePeriod also catches
    // OK_CLOCK_ANCHOR_REPAIRED, which before v1.4 only ever
    // appeared on stderr.
    std::cerr << "Warning: " << result.message << std::endl;
} 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. Cheap because they are pure lookups in the License object that load() cached: neither re-checks expiry, the clock or binary integrity, so in a process that stays up for days they keep returning Valid after the licence has expired or the clock has been rolled back. A tool that starts, checks and exits never notices. A service or desktop application must re-validate on a timer -- call check_expiry() hourly (or check_node_locked() if the hardware should be re-confirmed too), gate on that result, and treat check_feature() as the tier lookup it is; see §4.6. 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.lib, rockyguard.dll, rockyguard.sig, public.pem (it is embedded into your executable as a string constant, not shipped as a file), the deps/ folder, or the rest of the extracted package directory.

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: under %APPDATA%\YourCompany\YourProduct\license.json on Windows, or $XDG_CONFIG_HOME/YourCompany/license.json on Linux. Alongside the executable is fine ONLY when the executable itself lives somewhere the running user can write -- a portable or per-user install -- and not under %ProgramFiles% or /opt, for the reason in the next paragraph.

THE DIRECTORY HOLDING license.json MUST BE WRITABLE BY THE USER RUNNING YOUR APPLICATION. This constrains the choice above and is easy to miss. The anti-rollback machinery in §9.2 keeps a local cache file named .lck_cache_<tag> NEXT TO the licence, and it is written during verification -- check_expiry() and check_node_locked() both touch it, on every run, not only the first. One file appears per licence id; they are not cleaned up.

The consequence for the "alongside the executable" convention is the one that bites: a Windows application installed to %ProgramFiles%\YourApp\ runs read-only for a normal non-elevated user, so the cache write fails there every time. The same applies to /usr/local/bin or /opt on Linux, and to any read-only container image or signed app bundle.

What happens when the write fails: verification still SUCCEEDS. The cache is one of three anti-rollback witnesses (§9.2) and the other two live in per-user locations that remain writable, so losing it degrades tamper detection rather than breaking licensing -- your end user sees no error and your application keeps running. That is deliberate, and it is also why the problem is invisible in testing: nothing surfaces until someone audits why clock-rollback detection is weaker on deployed machines than on the developer's.

Recommendation: for an installed application, do NOT put license.json next to the executable. Use %APPDATA%\YourCompany\YourProduct\ on Windows or $XDG_CONFIG_HOME/YourCompany/ on Linux -- both are per-user and writable by construction. Reserve the alongside-the-executable convention for portable or single-user builds that live in a directory the user owns.

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.

Third-party attribution -- one more file, on every ship list above

This package bundles OpenSSL (Apache License 2.0). Every ship list in this section puts OpenSSL code into what you distribute, so every one of them makes you a redistributor of OpenSSL and Apache-2.0's attribution terms apply to your distribution, not only to ours.

On the Windows shared-link path this is visible: libcrypto-3-x64.dll and libssl-3-x64.dll are named on the list. On the two static paths and on the Linux shared path it is NOT visible, and that is the case people get wrong -- no OpenSSL file appears next to your binary, but OpenSSL is compiled or linked into it. ldd, dumpbin /dependents and the absence of a DLL beside your executable will all fail to remind you.

So add to every deployment:

- deps/OPENSSL-LICENSE.txt: the Apache License 2.0 text, verbatim, as shipped in this package.

Where you put it is your choice -- next to your executable, inside an installer's licence folder, or folded into your own THIRD-PARTY-NOTICES file with the other components you ship. What matters is that the recipient of your application can obtain the licence text. deps/THIRD-PARTY-NOTICES.txt in this package states the exact OpenSSL version this zip carries, restates these obligations, and records that OpenSSL 3.x ships no NOTICE file (so Apache-2.0 section 4(d) adds nothing here). If your legal review needs the version in writing, cite that file rather than this manual -- it is generated from the headers in the zip and cannot drift from them.

RockyGuard's own licence is a separate matter and is unaffected by any of the above; see docs/RockyGuard_License_Agreement.txt and §12.4.

3.5 Handling Verification Results (v1.4+)

Every verification call returns a LicenseResult. As of v1.4 it carries four things worth using, and the split matters because each has a different stability guarantee.

Member	Use it for	Stable?
status	Control flow. The coarse outcome, unchanged since v1.2.0.	Yes -- the enum is frozen; new precision goes into code instead.
code	Logging, routing, telemetry, support tickets. The specific condition.	Yes -- explicit numeric values in reserved blocks, never renumbered; names never reused.
message	Showing a human. One self-contained sentence.	No -- may be reworded in any release. Never match on it.
detail	Showing a human, or a support engineer. The particulars this call measured -- a path, a count, a drift in seconds, an HTTP status.	No.
suggested_action	Showing a human. What to do about it, in our words.	No, but the code it belongs to is.

The recommended shape is: branch on the STATUS, log the CODE, show the MESSAGE.

```
#include <rockyguard/rockyguard.h>   // pulls in license_error.h

auto result = verifier.check_node_locked();

// 1. LOG. to_json() renders code, status, severity, retryable,
//    message, detail, suggested_action and grace_days_remaining as
//    one line. Do this on every outcome, not only failures -- the
//    passing lines are what let you see how long a customer had
//    been in a grace period before they called you.

your_logger(result.to_json());

// 2. WARN. Warning severity sits only on results that convert to
//    TRUE. This branch must not refuse to start.
if (rockyguard::error_info(result.code).severity ==
        rockyguard::DiagnosticSeverity::Warning) {
    show_warning(result.message);
}

// 3. FAIL. Branch on status; the code refines the message you show.
if (!result) {
    show_error(result.message, result.suggested_action);
    switch (result.status) {
        case rockyguard::LicenseStatus::Expired:
            // GRACE_PERIOD_EXPIRED and LICENSE_EXPIRED share this
            // status. The code tells you which sentence the user
            // needs: "renew this" or "the grace you were already
            // given has run out".
            offer_renewal(result.code ==
                rockyguard::LicenseErrorCode::GRACE_PERIOD_EXPIRED);
            break;
        case rockyguard::LicenseStatus::HardwareMismatch:
            offer_rehost_request();
            break;
```

```
        default:
            break;
    }
    return false;
}
```

Why not just print message? You can, and nothing breaks if you do -- message is still the one self-contained sentence, now composed as summary + detail + suggested_action, so a v1.3 integration that logs only message loses nothing on upgrade. The reason to do more is that prose cannot be branched on. Three things become possible once you read the code:

- Retry only what is worth retrying. error_info(code).retryable marks the transient conditions -- the six NETWORK_* causes, a floating pool that is momentarily full, a library binary another process has open. A bad signature is never retryable. Loop on the flag rather than enumerating codes yourself:

```
for (int attempt = 0; attempt < 5; ++attempt) {
    result = client.checkout();
    if (result) break;
    if (!rockyguard::error_info(result.code).retryable) break;
    sleep_for_backoff(attempt);      // do not spin
}
```

- Route support traffic. "Recopy your activation key" (KEY_CHECKSUM_MISMATCH) and "your clock is wrong" (CLOCK_ROLLED_BACK) are different queues, and before v1.4 both were LicenseStatus values you could not reach the difference through.
- Tell apart conditions that share a status. Six of these are new in v1.4 and are the reason the feature exists; 10.1.1 lists them.

Do not show the user a code. It is for your logs and your support process. Show message, or show suggested_action next to your own headline -- those strings are written to be read by whoever is sitting in front of the application, and reusing them saves you writing (and translating) 50-odd remedies of your own.

Do not persist the numeric value of a status. LicenseStatus values are positional; the codes are the ones with explicit, permanent numbers. If you are writing a licensing outcome into your own database, write the code NAME (result.code_name(), e.g. "HWID_MISMATCH") or its numeric value, both of which are contractual, and read it back with rockyguard::error_code_from_name().

The full catalog of 56 codes, the reserved numeric blocks, and the stability contract are in Customer_API_Reference §4.5. The lookup functions -- error_info(), error_code_name(), error_code_from_name(), status_name(), error_catalog() -- are in §4.6. For diagnosing a failure you already have in hand, start at 10.1.1.

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).
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
```

Run it as the same user that will run your application, and on Linux do NOT prefix it with sudo. Since v1.4.0 the result is the same either way -- the library chooses its hardware sources by what any user could read, not by what the current process happens to be able to read -- but before v1.4.0 a root run could read the DMI board serial while the application, running unprivileged, could not, and the two disagreed on one fingerprint slot.

For non-technical end users on Windows: ship them rg_fingerprint_gui.exe from the tools/ directory of the customer zip instead of the CLI. It's a small GUI window (no terminal involved) that shows the component breakdown, displays the joined fingerprint hash in a copy-friendly box, auto-copies the hash to the clipboard on launch, and has a "Copy" button for manual recopying. Same fingerprint output as the CLI -- both tools compile the library's own hardware-collection source (v1.4.0; earlier releases kept a hand-maintained copy), so a license issued against the CLI's output verifies correctly against rg_fingerprint_gui's and against the library, and vice versa. See §7.1 for the full reference.

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), or reads as a firmware placeholder such as "To be filled by O.E.M." (v1.4.0; see "Placeholder serials" below), 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.

Containers -- the partial case. A container is rarely all-empty; it is usually one-or-two-of-four, which is worse because it looks healthy on the issuing machine and fails on the next restart:

- MAC address. The only interface inside a container is a veth peer (usually called eth0) with a MAC the runtime generates fresh on every container start. The library classifies it as virtual and uses it only as a last resort, so the slot is populated -- but with a value that changes each time the container is recreated. Pin it (docker run --mac-address, or the equivalent in your orchestrator) if you need it stable.
- Motherboard ID. /sys/class/dmi/id/board_serial is mode 0400 (root only) on mainstream distributions, so the library falls back to /etc/machine-id -- which a container usually lacks or leaves empty, hence an empty slot. Since v1.4.0 the choice is made from the file's permission bits, not from whether the current process can read it, so a root process and an unprivileged one pick the same source.
- Disk serial. Visible only if the runtime exposes the host's /sys/block, which many do not.
- CPU ID. Stable, but it is a model string shared by every machine of the same type.

So a node-locked license issued inside a container commonly matches 1 of 4 on the next restart and is rejected at the default threshold of 2. Node-locking is the wrong model for containerised deployments: use floating licensing (section 5), where the lease is tied to a server-side pool rather than to the container's hardware view. If you must node-lock a container, pin the MAC and run with access to /sys/class/dmi, and verify the fingerprint across a restart before issuing.

Deployment profiles -- choosing which slots to trust

The fingerprint has no weights and no per-component flags. It has a slot-skip rule and a threshold, and together they express every policy a deployment needs. match_count() ignores any slot that is the SHA-256 of the empty string on EITHER side. So the value you pass to --fingerprint-value may carry the empty hash

```
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
```

in any slot you do not want to count, and --threshold then says how many of the REMAINING slots must match. Blanking a slot on the licence side is a policy statement made once, at mint time; nothing changes in the application. It is the same rule that makes an unreadable component harmless, applied deliberately.

Bare-metal workstation. Use the four slots as rg_fingerprint reports them, threshold 2 (the default). This tolerates one NIC or one disk swap; two simultaneous changes need a re-issue. Raise the threshold to 3 for high-value licences (section 9, threat table).

Cloud VM (EC2, Azure, GCP). MAC addresses are assigned by the fabric and on some providers change across stop/start; disk serials are volume identifiers that follow the volume, not the instance; motherboard serials are per image and shared by every clone of it (the placeholder rule below blanks the worst cases). Lock on CPU plus motherboard-or-machine-id: put the empty hash in the MAC and disk slots and set --threshold 2, so BOTH remaining

slots must match. Expect a re-issue when the instance is rebuilt from a new image generation. If instances come and go, node- locking is the wrong model: use floating licensing (section 5).

Container. As described above, one or two slots survive a restart. Floating licensing is the answer. If you must node-lock, pin the MAC, blank the slots that are empty inside the container, and set --threshold to the number of slots you kept.

Sandbox or appliance with no readable identifiers. --threshold 0, the explicit "not hardware-locked" setting.

Run rg_fingerprint -v on the end user's machine before minting: it shows which slots are blank there, and those are the ones a profile must not count on.

Placeholder serials (v1.4.0)

Firmware that was never programmed with a serial does not report nothing; it reports a stand-in string: "To be filled by O.E.M.", "Default string", "System Serial Number", "None", "0", a run of one repeated character, or the SMBIOS specification's sample UUID 03000200-0400-0500-0006-000700080009. Budget boards ship this way and every VM built from the same template reports the same one. Before v1.4.0 the library hashed these as if they were identities, so a licence bound to such a board matched every other board with the same firmware default on that slot. From v1.4.0 the collectors treat a recognised placeholder as unreadable: the slot is empty, rg_fingerprint -v shows it blank, and match_count() skips it. The rule applies to the disk and motherboard slots on every platform; the MAC slot has its own virtual-adapter rules and the CPU slot is a model string by design.

Compatibility: a licence minted before v1.4.0 against a placeholder still verifies on the same machine -- the slot is skipped, not mismatched -- but it now counts one slot fewer towards the threshold. A licence that was passing at exactly 2 of 4 with the placeholder as one of the two needs a re-issue or --threshold 1. rg_fingerprint -v on the end user's machine shows which slots are blank before you decide.

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"
```

4.4.1 Activation-Key Form (v1.3+)

In addition to the .lic file form your library has used since v1.2.0, the verifier accepts an "activation key" -- the same signed envelope encoded as a single Base32 string with a recognizable RGAK- prefix and dash-grouped 5-character bodies, e.g.:

```
RGAK-A4DGB-2P67Q-RXKMN-4HEYS-...-Z9V77
```

This is a different transport for the same license -- it carries the same payload, the same Ed25519/RSA signature, and produces the same verification result as the .lic file. The advantage is delivery ergonomics: an activation key can be pasted into a settings dialog, sent inline in an email, or stored in a password manager, without file-attachment workflows. Choose whichever form your customer support flow prefers; both are equally valid.

Generating an activation key with the CLI:

```
tools/license_create --key private.pem \
  --vendor-license vendor_license.json \
  --id LIC-A4D2-2026 --licensee "Acme Corp" \
  --product "MyApp" --expires "2027-12-31" \
  --output       license.lic        \
  --output-key   license.key
```

Both --output and --output-key may be passed in the same run -- they emit the same minted license in two formats; the generation counter increments exactly once.

Generating an activation key from the portal: the license-issue page shows both the .lic file download and a Key link side-by-side after the license is minted; pick whichever channel fits the customer.

Loading an activation key in your application:

```
// Construction is guarded for the reason given in §12.2 and
// Customer_API_Reference §6.1: the constructor throws on a key it
// cannot parse, and an uncaught throw is std::terminate() on your
// end user's machine rather than a message. The rest of this
// snippet is about load_from_string(); the guard is not optional
// just because the surrounding example is about something else.
std::optional<rockyguard::LicenseVerifier> maybe_verifier;
try {
    maybe_verifier.emplace(public_key_pem);
} catch (const std::runtime_error& e) {
    std::cerr << "License system error: " << e.what() << "\n";
    return 1;
}
rockyguard::LicenseVerifier& verifier = *maybe_verifier;

auto result = verifier.load_from_string(activation_key_text);
// Same LicenseResult as load(path); same .status / .message
// semantics; the verifier sees the same License object.
```

load_from_string() accepts EITHER a raw JSON envelope (v1.2.x behavior, unchanged) OR a RGAK- activation key; the format is auto-detected by prefix. Whitespace and case are tolerated on activation keys -- a key pasted from email with stray line breaks or typed in lowercase decodes the same as the canonical form.

If the key arrives as a FILE on disk rather than as text, use load() and nothing else -- there is no need to read it into a string first:

```
verifier.load("license.lic");   // JSON envelope        - works
verifier.load("license.key");   // RGAK- activation key - also works
```

load() reads the file and hands the bytes to exactly the same detection logic as load_from_string(), so both forms work through either entry point. Detection is by CONTENT, not by file extension: the deciding factor is whether the first non-whitespace characters are "RGAK". The trailing newline that license_create --output-key leaves at the end of a .key file is stripped along with the dash separators, so such a file loads unmodified. In practice this means an application can accept whatever the end user hands it -- a .lic, a .key, or a pasted string -- and route all three through the same two lines of code.

What the verifier reports on input failures:

KeyMalformed	The string starts with RGAK but the body contains chars outside the Base32 alphabet, is truncated, or uses an unknown format version. Detail message names the precise failure mode.
KeyChecksumMismatch	The Base32 body decoded but the trailing CRC32 checksum does not match. Almost always means a single-character typo during transcription; recopy the full key from the source.
SignatureInvalid	The body decoded and checksums match, but the Ed25519/RSA signature on the inner envelope did not verify. The key has been tampered with, OR it was generated for a different vendor public key, OR the customer is loading it into the wrong product.
MalformedFile	The string is neither a valid activation key (no RGAK prefix) nor valid JSON.

The CRC32 is a customer-support aid, NOT a security primitive -- only the Ed25519/RSA signature on the inner envelope authenticates the license. The CRC exists so that "you mistyped a character" and "this key was tampered with" surface as distinguishable error states instead of both reporting SignatureInvalid.

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"
```

check_feature() is a lookup, not a re-validation. It reads the features list of the License that load() cached and nothing else: no expiry check, no clock check, no integrity check. In a tool that starts, checks and exits that distinction is invisible. In anything long-running it is the whole story: a service that loads a licence on Monday and gates on check_feature() alone keeps enabling every feature on Thursday, after the licence expired on Wednesday, because nothing has looked at the expiry date since Monday. Re-validate on a timer and gate on that:

```
// Hourly, on a timer thread. check_expiry() re-runs expiry, clock and
// integrity; use check_node_locked() if the hardware should be
// re-confirmed as well. Both write anchor state and may block on a
// network time check, so keep them off the UI thread (§3.3).
licensed.store(static_cast<bool>(verifier.check_expiry()));
```

```
// Feature gates, any thread: only meaningful while licensed is true.
if (licensed.load() && verifier.check_feature("export_pdf")) {
    enable_pdf_export();
}
```

The library has no built-in staleness policy: it will not force a re-check for you and does not track when the last one happened. Customer_API_Reference §6.5 has the full contract; tell us if you would use an opt-in maximum-age setting.

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.");
}
```

4.8 Web Portal (alternative to the CLI)

The customer zip ships a self-hosted FastAPI web portal under portal/ that wraps license_create in a browser UI. The portal is the recommended workflow for non-technical licensing-admin staff who would otherwise have to learn the license_create flag set. Everything the portal does is just license_create underneath, so CLI users and portal users can coexist on the same vendor private key without coordination.

What the portal does, at a glance:

- Products page -- define each product you license once (name, current version, default version range, the set of available features, the set of metadata keys you attach to licenses). Subsequent license generation pivots off the selected product so you never re-type product names or feature lists.
- Users page -- manage the end-user customers you issue licenses to (name, email, organization, trial / paid status).
- License generation -- pick a product, tick the feature checkboxes (auto-pre-filled from the most recently issued license for that product, so repeat-customer workflow is one click), fill metadata values, pick an expiry date from a calendar, click Generate. The user-detail page exposes each minted license in two delivery formats side-by-side: a .lic file download (the signed JSON file your library has used since v1.2.0) and a Key link (v1.3+) that opens the activation-key form (RGAK-prefixed Base32 string) in a plain-text browser tab for select-all + copy + paste-into-email delivery. Both forms are the same minted license -- the generation counter increments exactly once -- so pick whichever channel fits the customer.
- Batch generation -- issue many node-locked licenses in one submission. Shared product / features / metadata / expiry at the top of the form; a per-row table for (name, organization, email, fingerprint) at the bottom; "Paste CSV..." for large batches. Submit produces all licenses + a _manifest.json + a ZIP download.
- Dashboard generation-counter widget -- shows current end-user license count vs your vendor license's cap, with a progress bar that turns red below 20% remaining. Avoids the situation where minting hits the cap mid-workday because nobody was watching the counter.
- Vendor license tier auto-detection -- the Settings page reads your vendor license JSON, surfaces tier (Basic vs Premium), licensee, expiry, model, and max licenses. Basic tier vendor licenses automatically hide the Floating option from product and license forms.

- File-picker buttons in Settings for locating license_create.exe, your vendor license JSON, and your vendor private key -- no typing absolute paths.
- Login security -- bcrypt-hashed admin password, per-IP brute-force lockout with exponential backoff (see §4.8.6).

The portal is OPTIONAL. Everything it does can be done from the CLI as described in §4.3. Customers who prefer the CLI for scripting, audit logging, or CI integration can ignore the portal entirely.

4.8.1 Launching the portal + first-run setup

Launch (from the customer zip's portal/ directory):

Windows:	start.bat
Linux:	./start.sh
PowerShell:	./start.ps1

The start scripts create a Python virtualenv on first launch and install dependencies from a bundled offline wheels/ directory (no internet required), then prompt for an admin password. Subsequent launches are instant. The browser opens at http://127.0.0.1:8765/.

First-run Settings tasks (the portal will guide you):

```
1. Click Browse... next to "license_create executable" and pick it from the customer zip:
    tools/license_create.exe on Windows, tools/license_create on Linux.
2. Click Browse... next to "Vendor license file" and pick your vendor_license.json (provided separately by the
    library vendor).
3. Click Browse... next to "Vendor private key" and pick the private signing key you generated with
    license_keygen -- the file this manual calls private.pem everywhere else. The portal UI and
    portal/README.md label it "vendor private key", which is the same file under a different name; it is NOT
    something the library vendor sends you. (The vendor sends the vendor LICENSE, vendor_license.json. The
    signing KEY never leaves your machine.)
4. Leave Signing algorithm on "Auto-detect" unless you have a specific reason to force Ed25519 / RSA.
5. Optionally upload a logo to brand the sidebar.
```

The portal reads your vendor license and displays the tier (Basic / Premium), licensee, expiry, and max-end-user-licenses cap below the path fields. Green [OK] / red [X] pills next to each path indicate whether the file exists on disk.

Default network exposure is localhost-only (host: 127.0.0.1 in portal/config.json -- that file does not exist in the zip; it is generated on first run, so look for it only after you have started the portal once). The customer can flip to 0.0.0.0 for intranet access behind their own VPN if multiple licensing-admin staff need to share one portal instance; HTTPS via reverse proxy is recommended for any non-localhost deployment.

4.8.2 Defining your products

Open Products in the sidebar, click + New product, and fill the form:

- Product name (e.g. FooApp) -- used as the --product argument when minting licenses.
- Current version (e.g. 2.3.1) -- bump this each time you ship a release.
- Default version range (e.g. 2.) -- pre-fills the license form. Use 2. for major-compat, 2.3. for minor-compat, `` for any.
- Available features -- one per line. These become checkboxes on the license form. Names should match the strings your application checks against License::has_feature().
- Metadata keys -- one per line (e.g. region, channel, tier). Just the schema; actual values are entered at license-generation time. Stored on the license as License::metadata[key] and not interpreted by the library (see

§8.2).
- Default license type (Node-locked / Floating). Floating is hidden when your vendor license is Basic tier.
- Default grace days.

You can have any number of products. Each license is bound to one product. Deleting a product is safe -- existing issued licenses keep their snapshot of the product name and continue to verify; they just lose their link back to the product row.

4.8.3 Single license generation

Open Users, pick a user (or click + New user), then + Generate license. Select the product from the dropdown. The form populates:

- Version range -- pre-filled from the product (or the most recent license for that product if any).
- License type -- Node-locked / Floating (Floating only available on Premium-tier vendor licenses).
- Expiry date -- pick from a calendar widget OR check "Permanent". Past dates are rejected with a pop-up warning.
- Grace days -- pre-filled from the product default.
- Fingerprint match threshold -- 0-4 (default 2).
- Hardware fingerprint -- paste the value the end user got from rg_fingerprint / rg_fingerprint_gui.
- Features -- checkboxes generated from the product's available_features. Pre-checked from the most recent license issued for this product, so issuing the same license profile to another end user is one click.
- Metadata -- one row per metadata key declared on the product. Leave a value blank to omit that key from the license.

Submit. The .lic file lands at portal/licenses/<sanitized-email>/<timestamp>.lic and a download link appears on the user-detail page.

4.8.4 Batch license generation

For issuing many node-locked licenses in one submission (e.g., a 50-seat company rollout): sidebar Batch generate or + Batch generate on the Licenses page.

Shared fields (single product / features / metadata / expiry / grace / threshold) at the top. Per-row table at the bottom: Name, Organization, Email, Fingerprint. Click + Add row to extend or × on a row to remove. For large batches click Paste CSV... and paste:

```
Alice Lee,Acme Corp,alice@acme.com,ABC123-MAC-CPU-DISK-MOBO
Bob Singh,Acme Corp,bob@acme.com,DEF456-MAC-CPU-DISK-MOBO
```

Header row optional (auto-detected if the first cell is name). Click Replace rows with CSV -- the rows in the table get replaced and you can still edit any individual row before submitting.

Submit produces:

- One .lic file per row at portal/licenses/_batch/<batch_id>/<email>.lic.
- A _manifest.json capturing the shared-field snapshot and the per-row outcome (success / failure with reason).
- A result page with Download all as ZIP + per-row status pills.

Per-row license-create failures (e.g., bad fingerprint format, vendor-license error) don't abort the batch -- they're reported on the result page with the error message, and the rest of the batch proceeds.

Validation: name / email / fingerprint required per row; duplicate emails within one batch are rejected.

Batch is node-locked only (each row carries a unique fingerprint). For floating licenses, use single-license generation -- there is no batch-floating mode.

4.8.5 Generation-counter dashboard

The Dashboard shows a "Vendor license & generation counter" card with:

- Tier pill (Basic / Premium)
- Licensee + model + expiry
- Progress bar: current end-user licenses generated vs your vendor license's cap (max_end_user_licenses in the vendor JSON). Turns red when remaining drops below 20%.

The counter comes from shelling out license_create --show-count --vendor-license <path>. If your vendor license has max_end_user_licenses: 0, the cap is unlimited and the card shows the running count without a progress bar.

If the card says "Generation counter unavailable: ..." there's a Settings misconfiguration (most commonly license_create.exe or the vendor license JSON pointing at a wrong path). Open Settings and verify the green [OK] next to each path.

4.8.6 Login security (brute-force protection, v1.3+)

The portal's admin login is protected against brute-force attacks. After 5 consecutive failed password attempts from the same source IP, that source is locked out:

```
1st lockout: 1 minute
2nd lockout: 5 minutes
3rd lockout: 15 minutes
4th+ lockout: 60 minutes (capped)
```

Lockout state is persisted to portal/lockout.json and survives a portal restart -- killing the server does NOT reset the counter. Each failed attempt also incurs an artificial sleep (2 / 4 / 8 / 10 seconds, capped at 10) before the response is returned, which slows brute-force tools and removes the response-time side channel between "wrong password, bcrypt ran" and any internal short-circuit path.

A successful login from a previously failing IP clears that IP's full failure / lockout history; one proof-of-knowledge resets the penalty.

Recovery if you lock yourself out and don't want to wait: delete portal/lockout.json and refresh the login page. The file is regenerated automatically on the next failed attempt; no other state is lost.

X-Forwarded-For is honored when the portal sits behind a reverse proxy -- the lockout counter keys on the first entry of XFF (the original client IP) rather than the proxy's IP, so a single shared proxy in front of a multi-user deployment doesn't get all-users-locked-out by one bad actor.

Operational visibility: when a lockout fires, the portal logs a structured line to stderr in the form [portal] lockout: ip=<source> lockout_count=N locked_for=Ns. Pipe the portal's stderr to a log file if you want a persistent record of lockout events for audit / compliance review.

4.8.7 Backing up the portal

Two paths are worth backing up:

- portal/portal.db -- your customer + license history database (SQLite). Since v1.4.0 the database runs in SQLite's write-ahead-logging mode, so two sibling files, portal.db-wal and portal.db-shm, exist while the portal is running and may hold recent writes not yet in portal.db itself. Either stop the portal before copying, or copy all three files together. Keep the database on a local disk; WAL mode is not safe on network file systems.
- portal/licenses/ -- the .lic files themselves. Your end users may also have their own copies, but this is the master.

portal/config.json contains the hashed admin password and session secret; it can be regenerated by deleting it and

re-running the launch script for your platform (start.bat on Windows, ./start.sh on Linux), which prompts for a new admin password. Other regenerable files (lockout.json, .venv/) don't need backup.

For a fuller treatment of every portal feature including click-by-click walkthroughs, see portal/README.md shipped in the customer zip's portal/ directory.

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 v1.3 or any earlier release. Your customer zip contains floating_client.h but no floating_server.h, so there is nothing to include and nothing to link against: the server is a process you run, not a class you construct. 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).

ON THE C++ FIELD NAMES IN THIS SECTION. YAML is the only configuration surface available to you, and every setting below is reachable from it. Where this manual also names a C++ field -- "YAML key lease_timeout; C++ struct field FloatingServerConfig::lease_timeout_sec" -- that is a CROSS-REFERENCE, not a second way to configure the server. The names are given because the server's own log lines, error messages, and this manual's glossary use the C++ spelling, which differs from the YAML spelling often enough (lease_timeout vs lease_timeout_sec, private_key vs signing_private_key_pem) to be worth mapping. Read them as a translation table for interpreting output, not as an API you are expected to call. If you find yourself wanting to set one of these fields from C++, you want the YAML key instead.

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. As shipped it is configured for LOCAL EVALUATION -- it enables allow_insecure_http rather than TLS, because a first-run config cannot assume you already have a certificate. Before putting the server on any network you do not fully control, set tls_cert / tls_key and delete the allow_insecure_http line.

```
# 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: uncomment BOTH to enable HTTPS.
# tls_cert: server.crt
# tls_key: server.key

# Cleartext opt-in (v1.3+, default false). REQUIRED IF YOU LEAVE
# tls_cert / tls_key COMMENTED OUT. As of v1.3 the server REFUSES
# TO START with neither a cert/key pair nor this flag; pre-v1.3 it
# fell through to plain HTTP with only a log warning. Set it true
# only for local evaluation or an already-encrypted transport
# (loopback, service mesh, WireGuard). See the discussion below.
allow_insecure_http: true

# Belt-and-braces (v1.3+, default false). When true, startup
# requires tls_cert/tls_key even if allow_insecure_http is set, so
# an operator cannot launch a plaintext server by mistake in an
# environment where encryption is mandatory.
# require_tls: true

# Per-IP rate limiting (v1.3+). Token bucket per client IP; one
# token is consumed per incoming connection BEFORE the TLS
# handshake, so floods are rejected cheaply. rate_limit_rps is the
# steady refill rate, rate_limit_burst the bucket capacity.
# Defaults shown are generous -- normal traffic is one heartbeat
```

```
# per client per heartbeat_interval. Set rate_limit_rps <= 0 to
# disable the limiter entirely.
# rate_limit_rps: 50
# rate_limit_burst: 250

# 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.

Per-IP rate limiting (v1.3): rate_limit_rps (default 50) and rate_limit_burst (default 250) apply a token bucket per client IP address, checked before the TLS handshake so a flood is rejected cheaply. rate_limit_rps is the steady refill rate in requests/second; rate_limit_burst is the largest instantaneous spike tolerated from one IP. Over-limit connections receive HTTP 429 (plain HTTP) or are dropped (TLS). The defaults are generous -- normal traffic is one heartbeat per client per heartbeat_interval, so even a large NAT'd site sits far below the steady rate and the burst absorbs a mass cold-start. Set rate_limit_rps to 0 to disable the limiter entirely (for example if you front the server with a reverse proxy that already does rate limiting). Note that the unauthenticated /status endpoint is throttled by the same limiter; it exposes only pool counts.

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=127.0.0.1" \
  -addext "subjectAltName=IP:127.0.0.1,DNS:localhost"
```

THE CERTIFICATE IDENTITY MUST MATCH THE HOST YOUR CLIENTS DIAL. The -addext line above is not optional decoration. OpenSSL 3 verifies the server's hostname against the certificate's subjectAltName extension, and a certificate with no SAN is rejected no matter what its CN says. Replace the SAN values with the host your clients actually pass as FloatingClientConfig::server_host -- add DNS:license.example.com for a DNS name, IP:10.0.0.5 for an address, or list several comma-separated as above. The command as written is for a client dialling 127.0.0.1 or localhost, which is what the shipped floating_server_config.yaml and the rg_floating_client examples use.

Getting this wrong produces a failure that is easy to misread, so it is worth recognising. The status is LicenseStatus::ServerUnreachable, which reads like the server is down or firewalled; the TCP connection actually succeeded and only certificate verification failed. Since v1.4 the result also carries LicenseErrorCode NETWORK_TLS_HANDSHAKE_FAILED (1705) and its message says the TLS handshake failed, so log result.code and route on it rather than on the status or its text (§3.5); before v1.4 the message was the generic "Cannot reach license server". What the library still cannot tell you is WHY the handshake failed -- a name missing from the SAN, an unknown issuer and an expired certificate all arrive under the same code, because the client discards OpenSSL's verify result. rg_inspect's network probe does capture it: its floating-server TLS detail reads, for this case, "hostname mismatch". So if you see that code -- or the generic message on a pre-v1.4 build -- while the server log says the process started and is listening, run the Inspector and check the certificate's SAN before you check the network:

```
openssl x509 -in server.crt -noout -ext subjectAltName
```

An empty result means the certificate is the problem. Section 10.5.2 lists the other causes.

PREREQUISITE: this command needs the OpenSSL command-line tool, which is NOT included in your customer package -- the package bundles the OpenSSL libraries the library links against, not the CLI. On Linux install it from your

distribution (openssl on Debian/Ubuntu/RHEL). On Windows it ships with Git for Windows at C:\Program Files\Git\mingw64\bin\openssl.exe, or install it separately. Any tool that can emit a PEM certificate with a subjectAltName works; openssl is simply the one this command uses.

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.

As of v1.3, a client with use_tls = true and an empty tls_ca_cert_path fails closed: the FloatingLicenseClient constructor throws, because TLS without a CA certificate encrypts traffic but cannot verify the server identity (it is vulnerable to man-in-the-middle attacks). If you genuinely want to accept any certificate -- for example in a controlled pre-production test -- set allow_insecure_tls = true; the client then proceeds but prints a one-shot MITM warning to stderr. Prefer setting tls_ca_cert_path for any real deployment.

Two related server-side knobs (v1.3):

- require_tls (config key require_tls): when true the server refuses to start unless both tls_cert and tls_key are set. Use it to guarantee an operator cannot accidentally launch a plaintext server.
- allow_insecure_http (config key allow_insecure_http): BREAKING CHANGE IN v1.3. The server now refuses to start when tls_cert/tls_key are not both configured unless you set allow_insecure_http: true. Up to v1.2.x, omitting the certificate silently produced a working cleartext server with only a log warning. If you upgrade a plaintext deployment, the server will not start until you either configure TLS or add this key; the startup error names the key so the fix is one line. The client half is symmetric: FloatingClientConfig::use_tls = false now throws from the FloatingLicenseClient constructor unless FloatingClientConfig::allow_insecure_http = true.

Why this fails closed rather than warning: the checkout response carries session_secret, the HMAC key that signs every subsequent request. An eavesdropper who captures one plaintext checkout can forge any later request, so cleartext does not merely weaken client authentication, it nullifies it. Response signing does not compensate, because it provides authenticity, not confidentiality.

Note that allow_insecure_http and allow_insecure_tls are different settings on different axes, and confusing them is easy. allow_insecure_tls relaxes certificate VERIFICATION once TLS is already enabled; allow_insecure_http governs whether there is any encryption at all. Leaving allow_insecure_tls false does not protect you from cleartext.

Set allow_insecure_http only for local development, or where the transport is already encrypted by other means (loopback, a service mesh, a WireGuard tunnel).

- When TLS is disabled the client and server each emit a plaintext warning at startup, because the session secret and license traffic travel in the clear. Enable TLS for production.

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;

// The SERVER SIGNING public key -- NOT the license-verification
// public key you pass to LicenseVerifier. These are two different
// keypairs serving two different trust roles; see 5.2 above. Naming
// them distinctly in your own code is worth the small effort, since
// pasting the wrong one here silently collapses that separation.
```

```
config.server_public_key_pem = SERVER_SIGNING_PUBLIC_KEY;

// TLS is required as of v1.3: constructing a client with
// use_tls = false throws unless allow_insecure_http is set. See 5.2.
config.use_tls = true;
config.tls_ca_cert_path = "server.crt";     // Verify server certificate

// Local development against a plaintext server only -- never ship
// this. Replaces the two lines above rather than joining them.
// config.allow_insecure_http = true;

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
```

v1.3+: the client also offers four heartbeat-lifecycle callbacks (on_connection_lost, on_connection_restored, on_license_revoked, on_give_up) plus a retry-budget knob (max_heartbeat_failures). Wire these on FloatingClientConfig::callbacks to drive UI state from lease events instead of polling is_checked_out(). See Customer_API_Reference §8.7 for the full semantics, threading model, and re-entrancy contract. The one rule to carry in your head: callbacks run on the client's own heartbeat thread, so from inside one never call checkout() or checkin() on that client (rejected with ReentrantCall since v1.4) and never destroy the client (not guarded: a self-join that terminates the process). Set a flag and act from another thread. §10.5.4 has the details.

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, available }
```

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 (Linux: only where
                 world-readable; otherwise /etc/machine-id -- the same
                 source whether or not the process runs as root, v1.4.0)
```

On Windows and Linux (Linux since v1.4.0), wired Ethernet is preferred over Wi-Fi, and on all three platforms (macOS also since v1.4.0) any adapter whose MAC has the locally-administered bit set is treated as virtual regardless of its name -- that one rule catches Docker, veth pairs, Hyper-V dynamic NICs and MAC-randomising Wi-Fi without a name list. Named virtual adapters (VMware, VirtualBox, Hyper-V, Docker, CNI / Calico / Flannel / Cilium bridges, LXD, Podman, VPN, WireGuard, Bluetooth, etc.) are filtered as well. On Linux and macOS the choice among several physical NICs is made in sorted-name order, so it does not depend on the order in which the kernel happened to register the drivers at this boot. On macOS every en* adapter is a candidate (before v1.4.0 only en0 and en1 were, so a Mac whose Ethernet is a Thunderbolt or USB adapter reported no MAC); wired and Wi-Fi are not distinguished there, so en0 -- the built-in adapter, Wi-Fi on a laptop -- is chosen first, exactly as before.

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 (CLI)

```
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

# Capture it in a script (v1.3+)
rg_fingerprint > fingerprint.txt
FP=$(rg_fingerprint)
```

Output streams. The fingerprint is the tool's only data output and goes to STDOUT, alone and unadorned. Everything else -- the startup banner, the -v component dump, error text, the double-click pause prompt -- is diagnostic and goes to STDERR. So rg_fingerprint > fp.txt and FP=$(rg_fingerprint) both capture the bare fingerprint and nothing else, and rg_fingerprint -v > fp.txt writes the hash to the file while showing you the component breakdown on screen. (The banner additionally suppresses itself when stderr is not a terminal, and honours ROCKYGUARD_NO_BANNER.)

Before v1.3 the fingerprint went to stderr along with everything else, so both redirect idioms produced an empty file. If you have a script that worked around this by capturing stderr, it will now capture the banner and any -v output instead of the hash; read stdout instead.

When launched by double-click from Windows Explorer (rather than from an open command prompt), rg_fingerprint.exe detects this -- via the GetConsoleProcessList API -- and pauses with Press Enter to close... before exiting, so the user has a chance to read the output before the console window closes. The pause is suppressed when run from an existing terminal session so dev workflows are unchanged.

7.1.1 rg_fingerprint_gui (Windows GUI; v1.3+)

rg_fingerprint_gui.exe is a Windows-native GUI counterpart to the CLI tool above, shipped alongside it in tools/. Same fingerprint output (the two tools share their hardware-collection / virtual-adapter-filter / LAA-bit-check code via the rg_fingerprint_shared.h header, so output is identical), different audience: end users who would double-click an .exe from Explorer rather than open a terminal.

What the window shows:

- "Your hardware fingerprint" page title.

- A short instruction line: "Send this to your software vendor so they can issue a license bound to this machine."
- The four component values (MAC address, CPU ID, Disk serial, Motherboard ID) in a labeled grid.
- The joined fingerprint hash in a boxed read-only edit field (selectable text, monospace font).
- A "Copy" button (copies the fingerprint hash to the clipboard; flashes "Copied!" for visual feedback) and a "Close" button.
- A "[OK] Copied to clipboard" status message after Copy is clicked.
- The Rocky Software brand logo in the top-right corner.

On launch the GUI auto-copies the fingerprint hash to the clipboard, so a user who immediately Alt-Tabs to their email client and hits Ctrl+V doesn't need to click the Copy button first.

Implementation notes useful to downstream support:

- Compiled with /SUBSYSTEM:WINDOWS so no console window flashes on launch.
- Embeds a Common-Controls v6 manifest so buttons / edit fields render with the modern Windows 10/11 themed look rather than the Win9x 3D bezel.
- PerMonitor-V2 DPI aware -- crisp on 4K laptops without bitmap stretching. The window resizes after creation based on the actual monitor's DPI.
- Source compiled with /utf-8 so the U+2713 CHECK MARK glyph in "[OK] Copied" renders correctly regardless of the build machine's system code page.

Customer support tip: when a license issued to a customer's fingerprint suddenly fails to verify after a hardware change, ask the customer to re-run rg_fingerprint_gui.exe and send the new fingerprint. The match-count threshold (default 2 of 4) is forgiving for a single component change (one new MAC, one new disk), but two simultaneous changes (motherboard + disk replacement, or VM migration) will trip below the threshold and require a re-issued license.

There is currently no Linux / macOS GUI build of this tool; Linux / macOS end users use the CLI rg_fingerprint. The GUI is on the v1.4+ roadmap for cross-platform expansion when a customer asks.

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)
--output-key <file>         ALSO write a short activation key to
                            <file> (v1.3+). This is the flag that
                            produces the Base32 activation key
                            described in §4.4.1 -- the headline v1.3
                            feature -- and it is easy to miss because
                            a run without it silently produces only
                            the .json. Both --output and --output-key
                            may be passed in one invocation; see
                            §4.4.1 for when to hand an end user a key
                            rather than a file, and §9.8 for the
                            trailing-newline note when transcribing.
--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.

```
--wrap-feature-key <name>=<hex>
```

	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.
--wrap-data-key <hex>	Seal a 32-byte data key (64 hex chars) into the license for the "load-bearing license" pattern (v1.4+). At runtime the app opens an asset sealed under this SAME key with LicenseAsset(verifier).open(), so a patched-out check yields no key -- garbage, not a working unlock. Seal the asset at build time with tools/rg_bind_asset (§7.8) or the rockyguard_bind_asset() CMake helper, and reuse ONE key per product (the encrypted asset ships in the binary, identical for all end users). Bound to the license signature: only a genuinely-signed license yields the key. Round-trips into --output-key for free. See §9.11, Customer_API_Reference §6.10 / §12 and AI_INTEGRATION_GUIDE §6.5. Seal a SEPARATE 32-byte key for one feature into the license (v1.4+). Repeatable. Only the features you name here carry a key, so a license you did not entitle to <name> has nothing for LicenseAsset(verifier, "<name>") to recover -- and no patched check_feature() changes that. Use this, not the product-wide key, for anything tier-gated: the product key is in every license, so it gives no tier separation on its own. Pair it with the same <name> in --features; license_create prints a notice for any feature that is entitled but carries no sealed key.
--allow-eval-keys	Permit --wrap-data-key / --wrap-feature-key on a short evaluation license (v1.4+). Refused by default: wrapped keys are bound to the signature, not to time, so a trial carrying your production keys hands them to anyone who downloads it.
--gen-data-key	Print a fresh random 32-byte data key (hex) and exit (v1.4+). No key or vendor license needed. Run it once per product AND once per bound feature, and save each output: the build seals under it and every mint wraps it.

7.4 license_verify

```
Usage: license_verify [options]
```

```
--key <file>                Public key PEM file (required)
--license <file>            License file (required)
--algo <auto|ed25519|rsa>   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.
--check-hardware            Verify hardware fingerprint
--check-feature <name>      Check if feature is licensed
--check-version <ver>       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.
```

7.5 rg_floating_server (Premium tier)

```
Usage: rg_floating_server <config.yaml>
```

Takes exactly one argument: the path to a YAML config file. It has no other flags -- every setting is a YAML key, and the full key list with defaults is in 5.2 rather than repeated here. Run with no argument and it prints a sample config, which is the quickest way to see the recognised keys on a machine where you do not have this manual. That output goes to STDERR, not stdout, and the process exits 1 -- so rg_floating_server > server_config.yaml produces an EMPTY file. Redirect stderr instead:

```
rg_floating_server 2> server_config.yaml
```

(This paragraph said "to stdout" until 2026-08-14, which made the obvious way to capture it silently produce nothing.)

Refuses to start if neither tls_cert/tls_key nor allow_insecure_http is configured (v1.3+). See 5.2.

7.6 rg_floating_client (Premium tier; example binary)

```
Usage: rg_floating_client [host] [port] [public_key.pem]
                          [--tls] [--ca-cert <file>]
                          [--insecure-http] [--help]
```

This is an EXAMPLE and a smoke-test tool, not something to ship. Your application embeds FloatingLicenseClient directly (Customer_API_Reference section 8); this binary exists so you can confirm a server is reachable and issuing leases before you write any client code. Source is in examples/rg_floating_client.cpp.

Defaults are 127.0.0.1 and port 8080. Without --tls you must pass --insecure-http, or the client refuses to construct (v1.3+); the same fail-closed rule the library applies to your own code.

Note on the two tools above: section 7 is titled "CLI Tools Reference" and until v1.3 documented only the five node-locked tools, leaving the two floating executables discoverable only from 5.2 and the API reference. Both are in your tools/ directory, so both are listed here.

7.7 rg_inspect / rg_report_read (License Inspector; v1.4+)

The diagnostic pair you build and hand to YOUR end users. Source ships in examples/inspector/; the API behind it is documented in Customer_API_Reference section 11.

Why you build these rather than running ours. There are three parties:
Rocky Software writes this library, YOU embed it in your product, and YOUR END USER runs your product. Licensing problems happen on the end user's machine, and the support desk they contact is yours -- so the report has to come back to YOU, be readable by YOU, and carry YOUR product's name. Our build cannot do any of that, because it is sealed to our key. So you compile your own keys in and ship the result:

rg_inspect	give to your end users
rg_report_read	keep on your support desk (it holds the private key)

One-time setup:

```
tools/license_keygen --algo rsa \
    --private support_priv.pem --public support_pub.pem

cd examples/inspector
cmake -B build \
  -DROCKYGUARD_INSPECT_RECIPIENT_KEY=/path/support_pub.pem \
  -DROCKYGUARD_INSPECT_VERIFY_KEY=/path/your_license_pub.pem \
  -DROCKYGUARD_INSPECT_PRODUCT_NAME="Your Product"
cmake --build build --config Release
```

The recipient key must be RSA: an Ed25519 key is a signature key with no encryption operation, so your license signing key will not work here and the build tells you so. Keep support_priv.pem on your support desk -- it is the only thing that opens the reports you receive. Both compiled-in keys are PUBLIC keys; the build refuses a file containing "PRIVATE KEY".

ROCKYGUARD_INSPECT_VERIFY_KEY is the license-verification public key already embedded in your product. Supplying it is what makes the report carry the real verification verdicts your application produces -- usually the most useful section in it. Without it the license is described but not verified.

With both keys compiled in, your end user runs the tool with no arguments and no key files:

```
rg_inspect --license their.lic
```

They get a one-screen summary on their terminal -- often enough to fix an obvious problem on the spot -- plus one sealed file to send you. Useful extras when you are talking them through it: --note "<what happened>", --server <host:port> to probe your floating server, --no-network for no connections at all, --full-hardware when you need raw hardware ids.

What it collects, in one pass:

* The hardware fingerprint, component by component: which components the platform could read, their hashes, and

the raw values masked.

* The license file: size, mtime, SHA-256, and every field of the signed payload except the signature, which travels as a digest only.

* The verification verdicts your own application would get, each with its error code (section 4.5 of Customer_API_Reference).

* The system clock, and its measured drift against a trusted host.

* Network reachability by stage: DNS, then TCP, then the TLS handshake and certificate verification, each timed.

* The state of the anti-rollback anchor and license-cache files: present or missing, size, mtime, and who can write them.

* The library's own binary integrity self-check.

Reading the report, on your side:

```
rg_report_read --in report.rgdr --header-only
```

Provenance without decrypting -- when it was made, by which build, and the id of the key that opens it. Needs no key at all. Use it to pick the right key if you rotate them, and to confirm an attachment is a real report before opening it.

```
rg_report_read --in report.rgdr --key support_priv.pem --summary
```

The same verdict list your end user read off their own screen, rendered by the same library code -- so the two cannot disagree while you are on the phone with them. Nothing is ever appended to this mode, and that restraint is the mode: its whole value is being identical to what they are reading aloud.

```
rg_report_read --in report.rgdr --key support_priv.pem --diagnose

That same verdict list, then the ranked ROOT CAUSES behind it.
This is the one to run on a ticket; --summary is the one to read
aloud on the phone. Add --json to get the same thing as one
machine-readable document for a ticketing system to route on.

rg_report_read --in report.rgdr --key support_priv.pem
```

The whole report as JSON, for an editor or for jq.

Root causes, not symptoms (--diagnose). The verdict list reports one finding per section, each judged on its own. That is the right shape for a verdict list and the wrong shape for a diagnosis, because the interesting failures are correlated across sections and no single-section reader can join them.

The case worth knowing, because it costs real money: a licence reported LICENSE_EXPIRED on a machine whose clock runs three days fast has NOT expired. The expiry verdict is in the licence section; the measured drift that explains it is in the clock section. --summary reports both and is not wrong to; --diagnose reports that the first is caused by the second, and tells you not to issue a replacement -- which would read as expired too the next time that clock skips.

Every cause carries the report fields it was drawn from, a confidence, and one action. The verdict list is always printed above it, deliberately: a diagnosis presented without the facts behind it is worse than a symptom list, because a support engineer will act on it. Where a cause explains another, the one it explains is kept in the list and marked rather than hidden, so you can see what the engine discarded if you disagree with it.

Where correlation does not change the answer, no rule fires and you get the library's own remedy for the code, unchanged -- section 10.1.1. The engine is not a second copy of that table; it is the set of cases where looking at two

sections at once gives a different answer than looking at one. Section 10.2 lists them.

Exit status is 0 whenever the report was read, whatever it says, exactly as for rg_inspect -- diagnosing a broken licence is the tool succeeding. Branch on --json, not on the exit code.

Diagnosing a hardware mismatch. This is the part you cannot get by asking. check_node_locked() tells you "2 of 2 components matched", which is the right answer for the library and a useless one for a human: it cannot say WHICH component moved. The report breaks the comparison down per component, so it shows at a glance that the disk serial changed and the other three did not -- distinguishing a replaced disk from a reimaged machine from a license issued against the wrong fingerprint entirely. Three different conversations. It also surfaces the case where verification still PASSES because the threshold is met while a component has already drifted: an end user one hardware change from a lockout, with nothing else to warn them. A component neither side could read is reported as absent rather than as matching, because the library does not count it either.

Run it BEFORE trying again. Verifying WRITES anti-rollback state: it creates the witnesses when it finds none, and re-baselines them after a confirmed forward clock excursion. Those two states are exactly what a field failure looks like, so the tool inventories them before it verifies anything and records in the report that it did. The practical consequence for your support script: have the end user run rg_inspect BEFORE they launch your application again, or the evidence explaining the original failure may already be gone.

--no-clock-check skips the clock/anchor verdict, but it does NOT make the run side-effect free: whenever a verification key is available the tool verifies, and verification reaches the same clock machinery on its own, creating the witnesses if it finds none. (Measured, and corrected here -- this passage previously claimed a "strictly read-only pass", which was wrong.) If you need a run that writes nothing at all, run it without a verification key: the licence is then described but not judged. Either way the inventory records the state as it was BEFORE anything ran, which is the part that matters for diagnosis.

Exit status is 0 whenever a report was written, even when the report says the license is invalid -- diagnosing a broken license is the tool succeeding. Non-zero means it could not run or could not write. This is deliberately the opposite of license_verify, which is the one to branch on in scripts.

What is in the file, and what is not. The report never contains a private key, a passphrase, or a floating-license session secret. The license signature travels as a SHA-256 digest and a length, never verbatim. Raw hardware identifiers and license metadata VALUES are masked to their first and last two characters plus a length; the metadata keys are kept, since we have no way to know what your integration put in the values. The fingerprint component HASHES are included in full, because they are what the license binds to and what a mismatch is diagnosed against. So are license id, licensee, product, features, dates, entitlements, error codes and file paths -- a masked path cannot diagnose a permissions fault.

These rules live in the LIBRARY, not in the shipped source, on purpose:
they are a promise this document makes, and a promise that shipped as editable source would be neither testable nor reliably true of any particular build. Change what is collected by setting fields on rockyguard::diagnostics::Options, not by editing the collection.

File paths are why --plaintext is an opt-out rather than the default: on Windows they routinely contain the operator's account name.

Encryption. A fresh random AES-256-GCM key per report, wrapped to your recipient key with RSA-OAEP/SHA-256, with the container's own metadata (format, scheme, cipher, timestamp, tool version, recipient key id) bound in as additional authenticated data -- so a report cannot be relabelled as having been collected at a different time or by a different build and still open.

One scheme, deliberately. Sealing to your RSA key needs no shared secret and nothing transmitted out of band, so there is nothing else to choose and nothing for an end user to invent, transcribe, or leak.

If a build has no recipient key compiled in, the tool stops and says so rather than silently producing something nobody can open. Supply one with --recipient-key, or use --plaintext to write the report unencrypted -- reasonable when you are reading it yourself, not something to mail, and the tool warns before writing one. Compiling the key in removes the choice entirely, which is the configuration to ship.

Putting the diagnostic INSIDE your application. You do not have to ship a separate executable at all. The same collection is available as a library call, which is how you build a "Help -> Diagnose Licence" menu item or a quiet self-test at startup:

```
#include <rockyguard/diagnostics.h>
namespace rgd = rockyguard::diagnostics;

rgd::Options o;
o.license_path   = "license.lic";
o.public_key_pem = kMyProductPublicKey;   // already in your binary

rgd::HealthReport h = rgd::health_check(o);
// h.overall   -- worst severity, for colouring the dialog
// h.headline  -- one sentence for the title
// h.items     -- one row per finding: label, value, detail, code
// h.report_json -- the full report, ready for seal() if they want to
//                  send it
```

Each item carries a code -- the same error code the rest of the API uses -- so you branch on that rather than on wording that may be reworded:

```
for (const auto& item : h.items)
    if (item.code == "HWID_MISMATCH") offer_reactivation();
```

For a menu item, run it off your UI thread. Collection can take seconds (the network stage probes several hosts, and DNS cannot be bounded), so there is an async form that calls nothing back -- you poll it, which means nothing has to be marshalled into your event loop:

```
rgd::AsyncHealthCheck job(o);            // returns immediately
// on a timer tick:
status_label = job.progress();
if (job.ready()) show(job.take());
```

For a startup check, set o.probe_network = false. That skips every socket and brings the cost down to roughly a tenth of a second, most of it hardware fingerprinting. If all you want at startup is "is this licence valid", though, use LicenseVerifier directly (Customer_API_Reference section 6) -- it is cheaper still and that is what it is for. Reach for health_check() when you want the surrounding context as well: clock, cache state, integrity, network.

See examples/health_check_example.cpp for both patterns in full, and Customer_API_Reference sections 11.6 and 11.7 for the reference.

Root causes in your own support tooling. The same correlation engine rg_report_read --diagnose uses is a library call:

```
rgd::DiagnosisReport d = rgd::diagnose(h.report_json);
for (const rgd::RootCause& c : d.causes) {
    if (c.suppressed) continue;          // explained by an entry above
    show(c.headline, c.remediation);     // branch on c.cause_id
}
```

diagnose() takes the report JSON -- HealthReport carries it, so no second collection is needed -- and explain() renders the same thing as text if you would rather print it. diagnosis_json() serialises it for a ticketing system. All three are pure functions of the report and never read the system clock, so a report diagnosed a year later gives the identical answer it gave on the day.

Note what this is FOR. It is written for whoever reads a support ticket, not for the end user's own screen: some causes name your issuance pipeline, and the anti-rollback remedies double as a tamper recipe. Causes carrying reasoning you would not want an end user reading are flagged support_only so you can filter them, but nothing filters them for you. The shipped rg_inspect deliberately does not diagnose; putting diagnosis in front of your end users is your decision to make, not ours to make for you.

Customising. Each tool is roughly 400 lines over <rockyguard/diagnostics.h> and nothing else. Rename them, restyle the output, put rg_inspect in your installer, wrap it in a GUI, or call the API directly from your own application and skip the CLI entirely. See examples/inspector/README.txt.

7.8 rg_bind_asset (build-time asset binder; v1.4+)

The build-time half of the load-bearing licence pattern (§9.11). It seals a plaintext asset under a 32-byte key into the library's AES-256-GCM blob layout -- nonce(12) || ciphertext || tag(16), the format LicenseAsset::open() reads -- and optionally writes it out as a C/C++ header so your build embeds ciphertext only. The plaintext never enters your binary.

```
Usage: rg_bind_asset --in <asset> [--out <asset.enc>]
                     [--emit-header <hdr.h> --symbol <c_identifier>]
                     [--feature <name>]
                     (--data-key <64hex> | --data-key-env <ENVVAR>)

--in <asset>                The plaintext to seal (required).
--out <asset.enc>           Write the raw blob to a file.
--emit-header <hdr.h>       Write the blob as a header: a
  --symbol <ident>          static const unsigned char <ident>[] plus
                            <ident>_len, with the runtime call to copy
                            in a comment at the top. --symbol is required
                            with --emit-header.
--feature <name>            Documentation only: names the feature whose
                            key you are sealing under, so the generated
                            header's comment shows
                            LicenseAsset(verifier, "<name>"). Omit it
                            when sealing under the product-wide key.
--data-key <64hex>          The key. Prefer --data-key-env <VAR>, which
--data-key-env <VAR>        reads it from the environment so it stays out
                            of build scripts and shell history.
```

Which key: the product data key (license_create --gen-data-key, wrapped into every licence with --wrap-data-key) for product-wide assets, or a separate key per feature (another --gen-data-key, wrapped only into entitled licences with --wrap-feature-key <name>=<hex>) for anything tier-gated. Per-feature keys are wrapped by you at mint time, never derived from the product key -- §9.11 explains why. The binder seals under whichever key you hand it and derives nothing.

Exit status: 0 on success; 2 on any usage or I/O error, with a one-line reason on stderr. Deliberately OpenSSL-only: it does not link the RockyGuard library, so it runs in a build step with no licence context, and its source (tools/rg_bind_asset.cpp, with examples/bound_asset.h) rebuilds against deps/ alone -- the examples CMakeLists builds it as target rg_bind_asset. On Windows the prebuilt tools/rg_bind_asset.exe needs the two OpenSSL DLLs beside it, exactly like the other tools.

From CMake you normally do not call it by hand. cmake/RockyGuardBindAsset.cmake provides rockyguard_bind_asset(), which runs the binder as a custom command, regenerates the header whenever the plaintext changes, and adds the header to your target:

```
list(APPEND CMAKE_MODULE_PATH "${ROCKYGUARD_DIR}/cmake")
include(RockyGuardBindAsset)
rockyguard_bind_asset(TARGET myapp FEATURE export_pdf
    ASSET  ${CMAKE_SOURCE_DIR}/assets/export_tables.bin
    HEADER ${CMAKE_BINARY_DIR}/generated/export_tables.enc.h
    SYMBOL kExportTables
    DATA_KEY_ENV RG_KEY_EXPORT_PDF)
```

The build reads the key from $RG_KEY_EXPORT_PDF (a CI secret, never committed). The helper finds the binder in <package>/tools, else uses the rg_bind_asset target if your tree builds it, else PATH.

7.9 rg_bind_lint (integration linter; v1.4+)

A Python 3 script (standard library only) that scans C/C++ source for the ways a load-bearing integration silently regresses to a flippable boolean:

```
python tools/rg_bind_lint [PATH ...]      # default: current directory
python tools/rg_bind_lint --quiet src/    # ERROR findings only
```

ERROR findings (exit status 1): a branch on unwrap_data_key() / unwrap_feature_key(); a shared bool is_licensed()-style predicate; a branch on LicenseResult.message text; and, across the whole tree, a licence key that is recovered but never used to open an asset. WARN findings: an if (key.empty()) refuse; or if (asset.empty()) refuse; guard, and no wrong/foreign-licence test anywhere in the tree. INFO: exit()/abort() calls, which are fine in a CLI and fatal inside a plugin or GUI.

It is a lint, not a proof. Zero errors is necessary, not sufficient: the negative test in §9.11 step 5 -- a foreign licence must leave every bound feature visibly broken, not merely "invalid" -- is what actually establishes that a binding is real.

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.

As of v1.3, verifier.license() returns a License by VALUE rather than a const reference, so that a snapshot cannot be torn by another thread reloading the verifier (Customer_API_Reference section 6.11 has the full rationale). The examples below are unaffected and remain correct: binding "const auto& meta = verifier.license().metadata;" to a member of the returned temporary extends that temporary's lifetime for as long as the reference lives, and using verifier.license() directly inside a larger expression keeps it alive to the end of that expression. The only form that stops compiling is a non-const reference binding such as "auto& lic = verifier.license();" - deliberately, because a compile error is a clearer signal than a silent change in behaviour. If you hold license data for a long time, prefer copying it outright: "rockyguard::License snap = verifier.license();".

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 the customer is allowed to operate -- "EU", "NA", "APAC-no-CN"), edition (which SKU this license was issued for), end_user_id (your customer's own identifier for their end user).

```
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 without invalidating the signature. The leak is tamper-evidence.

- 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 is a free-form string map, not a typed config.)

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

KEY ROTATION (cited from 4.1 and the 12.5 checklist). Two places in this manual point here for the rotation procedure, and until v1.3 this section contained none -- it is about signature verification. The procedure, in full:

1. Generate a new keypair with license_keygen. Keep the old private key; you will need it only if you must re-sign anything under the old identity, and you should destroy it once step 4 completes.
2. Ship a new application build with the NEW public key embedded. Until a customer installs that build, their binary trusts only the old key -- this is the step that sets your rotation timeline, not the key generation.
3. Re-issue every outstanding end-user license, signed with the new private key. Licenses signed with the old key will NOT verify against the new public key; there is no dual-key grace period in v1.3, by design (accepting two signing identities at once would weaken the property the signature exists to provide).
4. Set an end-of-life date for the old application version, after which you stop issuing old-key licenses.

Because step 2 requires a shipped binary and step 3 touches every customer, rotation is a release-scale operation, not a maintenance task. That is why 4.1 advises choosing a key strong enough to outlive your longest support commitment. If the private key is COMPROMISED rather than merely aging, the same four steps apply but on an emergency timeline, and you should assume any license mintable with that key already exists.

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. THE POOL, so you can write the firewall rule (all port 443, outbound only, no inbound and no other port):

www.google.com	www.microsoft.com	www.cloudflare.com
www.apple.com	www.amazon.com	www.github.com
www.wikipedia.org	www.mozilla.org	www.w3.org
www.ietf.org	www.debian.org	www.kernel.org

```
One reachable host is enough -- a single successful check
re-anchors the clock -- so an allow-list covering any subset still
works, with fewer hosts meaning more retries when some are blocked.
The list is compiled in; treat it as fixed for v1.3 and re-check
this section on upgrade rather than assuming it is stable forever.
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.
```

Deploying on intranet-only or air-gapped networks:

There is currently NO configuration flag or API setting to disable the outbound time check, and no way to redirect it at an internal NTP or HTTPS host. This is a real gap for locked-down deployments and we would rather say so than let you search for a switch that does not exist. It did not make v1.4; it is recorded on our roadmap (item 4.18), shaped by one constraint worth understanding before asking for a particular switch. The machine running your application belongs to the end user, who in node-locked licensing is the party the clock check defends against. A runtime setting, environment variable or configuration file that redirects the check at YOUR time server is a setting the end user can point at THEIRS, and a switch that disables the check is a switch they can flip -- which is why a hosts-file redirect of the pool already fails TLS certificate verification and is treated as offline rather than trusted. A sound version therefore has to be fixed where the end user cannot reach it: in your build (a pinned internal HTTPS host with its CA compiled into your binary) or as an explicit offline-only build that accepts losing the online rollback defence and says so in its diagnostics. Those are the two shapes the roadmap item takes. Tell us which you need; the answer decides the order.

What actually happens on a network with no route out matters more than it sounds, because the check is SYNCHRONOUS and blocks the calling thread. The library tries up to 3 hosts at 3 seconds each, so the worst case inside RockyGuard is roughly 9 seconds added to the call that triggered it -- typically load() at startup. That is the library's own budget and does not include OS-level DNS resolution, which on some configurations adds several seconds of its own before the connection attempt even begins. A network that DROPS packets silently produces the worst case; one that actively REFUSES the connection, or has no DNS entry at all, usually fails in milliseconds. If your environment can be made to refuse rather than blackhole traffic to port 443, startup latency largely disappears.

The check fires on 100% of first runs (or any run where the anchor files are missing) and on about 10% of subsequent runs, so a stalled startup is intermittent rather than constant -- which makes it easy to misdiagnose as a hang somewhere else. If you see occasional multi-second startup delays on an isolated network, this is the first thing to rule out.

Practical mitigations available today: allow outbound HTTPS to the time-check pool if policy permits, since one successful check re-anchors the clock; call load() off the UI thread so the stall is not user-visible; or accept the latency knowing it is bounded and intermittent. Note that the check speaks TLS directly to the pool hosts and has no HTTP proxy support: on a network where all egress must pass through a mandatory proxy, every host is unreachable, the check reports offline, and the machine is indistinguishable from an air-gapped one as far as the library can tell. rg_inspect (§7.7) probes the pool stage by stage and its diagnosis names the proxy case explicitly, so run it before concluding that a site is air-gapped.

One security consequence is specific to air-gapped machines and is a documented, accepted limitation rather than an

oversight. The online check is what catches a rollback when the local anchors are gone. On a permanently isolated machine, a user who deletes ALL anchor locations AND sets the clock back gets one false pass, because the library cannot distinguish that from a legitimate first run on a new offline machine -- refusing would break every genuine air-gapped install. See the threat-model table in section 11.2 and Customer_API_Reference section 6.6. For air-gapped fleets the practical countermeasure is shorter licence durations, so re-issuance forces clock truth on a regular cadence. A second countermeasure is recorded on the same roadmap item and is NOT in v1.4, so do not rely on it yet: every licence already carries its issue date (issued_at, set by license_create at mint time), and a system clock earlier than that date is impossible for a legitimate install; treating it as ClockManipulated would cost nothing offline and would bound the one false pass to the licence's own term measured from its mint date, though it cannot catch a rollback to a date after minting.

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.

Anchor integrity does not depend on where the licence lives. Each witness carries an HMAC computed over its timestamp plus a stable identifier for the LOCATION, not over its filesystem path. That distinction matters whenever an installation moves. Before v1.3.1 the path was part of the integrity tag, so renaming an install directory, spelling the licence path relatively on one run and absolutely on the next, or (on Windows) reaching one directory through a different drive-letter case, a mapped drive, or a junction turned an intact anchor into a failed integrity check -- reported as tampering, which the library deliberately never self-repairs. Anchors written by v1.3.0 and earlier still verify and are rewritten in the current form on the next successful run, so upgrading needs no action and loses no rollback history.

Because that third witness is written NEXT TO license.json, the directory you choose for the licence must be writable by the user running your application, on every run and not only the first. If it is not -- the common case being an application installed under %ProgramFiles% or /opt and run non-elevated -- verification still succeeds and your end user sees nothing, but that witness is permanently absent and rollback detection falls back to the two per-user backends. Nothing tells you either: the failed write produces no stderr line, no log-callback message and no result code, and rg_inspect currently records writability only for witnesses that exist, so its report shows one absent witness with no explanation. Both gaps are recorded on our roadmap (a Warning-severity result code for an unwritable witness location, and an Inspector probe that names the directory); until they ship, the only way to notice is to look for the file. §3.4 covers the deployment choice this implies; the short version is to put license.json under a per-user directory rather than beside the

executable, and to check on your own test install that .lck_cache_<tag> actually appears next to it.

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 -- plus the opt-in runtime check below, which is the one integrity mechanism a static build CAN run for itself.

Runtime in-memory code check (opt-in, v1.4+):
Separately from the on-disk .sig check above, the library can verify at runtime that its own critical functions have not been PATCHED IN MEMORY after loading -- an inline hook, a software breakpoint, or a debugger write, none of which change the file on disk and so none of which the .sig check can see. On the first validation the library records a SHA-256 baseline over the entry of each enforcement-critical function (the signature verifier, the clock check, and the integrity checker itself); every later validation re-hashes those spans and compares. Because the process is compared against its own baseline, address-space randomization and load-time relocations do not matter -- a mismatch means the running code was modified, and the validation fails with INTEGRITY_RUNTIME_MODIFIED (§10.1.1, §10.4.3).

Unlike the .sig check, this needs no separate binary and no .sig file, so it is the one integrity mechanism that also covers STATIC builds. It is OFF by default and enabled at build time with the ROCKYGUARD_RUNTIME_INTEGRITY CMake option (§12.1); an application that merely rebuilds against a newer RockyGuard sees no change in behavior until you turn it on. Weigh it before enabling: a legitimate function-hooking APM/EDR agent, a profiler, or running your own application under a debugger with software breakpoints can trip it. It raises the cost of an in-memory patch; like any client-side check running on the end user's machine, it does not make one impossible, and it cannot see tampering already present before its first-run baseline.

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. As of v1.3 this is enforced rather than advised: a server with no tls_cert/tls_key refuses to start unless allow_insecure_http: true is set, and a client with use_tls = false throws from its constructor unless allow_insecure_http = true is set. See the allow_insecure_http entry above for the upgrade impact and the reasoning.

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)	On-disk binary integrity self-check (DLL/SO builds): rockyguard.sig is verified at load against the binary's SHA-256. Optional runtime in-memory code check (ROCKYGUARD_RUNTIME_INTEGRITY, §9.3): re-hashes critical functions on every validation to catch in-memory patches and hooks, and covers static builds too. As a further, independent layer the licence-decision paths can be compiled with code obfuscation (control-flow flattening, string encryption; a vendor build-time option, §9.8) that makes those branches hard to locate and patch in the first place.	Residual: the .sig check is bypassed by modifying BOTH the binary AND the .sig in coordinated fashion, and it cannot see a patch applied only in memory (a debugger, an injected hook). The optional runtime check closes that in-memory gap, but running on the end user's machine it can itself be patched out by a determined attacker and does not catch tampering already present before its first-run baseline. Obfuscation raises the cost of finding the branch but, like any offline check, does not make it unbreakable on hardware the attacker controls. Mitigation: enable the runtime check where its false-positive surface (debuggers, function-hooking APM/EDR) is acceptable; request an obfuscated build if your threat model warrants it; ship the application's own code signing so the OS / loader detects pre-launch modification before RockyGuard runs. Static-library builds embed the library code 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)	TLS encryption, REQUIRED as of v1.3 (use_tls + tls_ca_cert_path; running without it needs an explicit allow_insecure_http opt-in on both client and server); REQUIRED Ed25519 payload signature on every server response (server_public_key_pem)	Residual only if the opt-in is used: a passive sniffer captures the session secret issued at checkout and can forge subsequent HMAC-authenticated requests, so response signing does not help -- it gives authenticity, not confidentiality. Mitigation: do not set allow_insecure_http outside local development; 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.

Attack	RockyGuard defense	Residual risk and mitigation
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.
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 vlid. 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>:<vlid>:<hmac>
```

A single text line. <count> is a non-negative integer. <vlid> is the vendor_license_id. <hmac> is HMAC-SHA-256 over <count> + <vlid> with a per-location salt (which of the storage locations the file is; v1.4.0 -- earlier releases salted with the file's own path string, so on Linux/macOS a differently spelled $HOME, such as a trailing slash or an autofs alias, made the same file read as tampered; files written by earlier releases still verify and are rewritten under the new salt on the next license_create run), 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. Target the per-user application-data location (%LOCALAPPDATA% on Windows, $HOME/.local/share/rockyguard on POSIX), which needs no admin or registry access to read. Do not target the working-directory file: as of v1.3 it is no longer written, so an audit script reading it will report a stale count or find nothing at all.

Local-cache file note:
CHANGED IN v1.3: the counter is no longer written to the current working directory. Through v1.2.x a third copy was written to <cwd>/.rg_gencount_<tag>, so a file appeared in whatever directory you happened to run license_create from -- including version-controlled trees, where it became untracked noise you had to remember not to commit.

It was dropped because it was the weakest of the three locations, not merely the most annoying. Being a RELATIVE path, its contents depended on where the process was invoked: run license_create from a different directory and the file was simply absent, contributing nothing. Anyone deliberately shedding it needed only to change directory. The two remaining locations are absolute, and each has to be found and deleted individually, which is where the redundancy actually comes from.

Existing files are still READ, so upgrading cannot lower your count: if the highest value happens to sit in an old <cwd>/.rg_gencount_<tag> -- for example because %LOCALAPPDATA% was unwritable on the run that incremented it -- load() still picks it up. Nothing writes those paths any more, so no new ones appear.

Housekeeping after upgrading: any .rg_gencount_ files already sitting in your working trees are now inert once the count is reflected in the two permanent locations, which happens on the next license_create run. You can delete them at that point. If you added .rg_gencount_ to .gitignore on the previous advice, it is harmless to leave in place.

9.8 Code Obfuscation of the Licence Logic (build option)

The digital signature (§9.1) makes a licence file impossible to forge, but the code that ACTS on a valid/invalid result -- the signature, expiry and hardware-match branches inside the library -- is ordinary conditional code. An attacker who cannot forge a licence may instead try to patch that code: locate the "is this licence valid?" branch in the binary and flip it. The integrity checks (§9.3) are designed to DETECT such a patch; obfuscation is the complementary layer that makes the branch hard to FIND in the first place, by compiling the licence-decision paths with control-flow flattening (the branch structure is turned into a dispatcher-driven state machine) and string encryption (the log strings that would let a reverser grep straight to the check are removed from the binary).

This is a VENDOR BUILD-TIME option, off by default. It changes no API and no observable behaviour -- an obfuscated library and a plain one verify licences identically and are used from your application in exactly the same way -- so whether the specific library you received is obfuscated is a choice your vendor makes at build time, not something visible in the public headers or the API. If your product's threat model includes skilled reverse-engineering of the shipped binary, ask your vendor whether an obfuscated build is available for your platform.

To be concrete about the default, since "off by default" is easy to read as "probably on in the real product": the standard RockyGuard release zips, on Windows and on Linux alike, contain UNOBFUSCATED binaries. An obfuscated build is produced only on request and, at present, only for Windows -- the obfuscating toolchain has no Linux build yet. If you need to know which you received, ask; it is recorded per artifact on our side.

Honest scope. Obfuscation is deterrence, not prevention: it raises the cost and skill needed to locate and patch the check, and defeats one-click or scripted cracks, but any offline licence check can ultimately be broken on hardware the attacker fully controls. Just as important, the single most effective thing YOU can do is on your own side of the boundary. If your application gates everything on one line --

```
if (result.status == LicenseStatus::Valid) { unlock_all_features(); }
```

-- then an attacker patches THAT one branch, in YOUR binary, and never touches RockyGuard at all. Prefer to (a) check the result in several places -- but note the choke point is the VERDICT, not the number of sites: eight gates that all call one predicate (or one bool is_licensed() wrapper) are still one patch, and byte-identical inlined checks get folded back together by the compiler (CSE) and linker (identical-code-folding), so vary the real work at each site; (b) let the outcome feed real program logic rather than a single boolean gate -- strongest of all, DERIVE a key from the validated licence and decrypt data you need with it, so a patched-out check yields garbage rather than a working unlock; the library supports this directly as of v1.4 (§9.11) -- and (c) ship your own executable code-signing so the OS detects modification of your application before it runs. See §9.10 below on symbol exposure and stripping, §9.11 on load-bearing licences, and the AI Integration Guide's Section 9 (Hardening) for worked patterns.

9.9 Public-Key Hiding (Anti-Substitution)

Your application embeds the license-verification PUBLIC key (§2.4, step 3). That key is not secret and does not need to be -- an attacker who reads it cannot forge a license, because forging needs your PRIVATE key, which never leaves your control. The attack this subsection addresses is different: SUBSTITUTION. If the key is embedded as a plain PEM string, it sits in your binary as a contiguous block that strings yourapp | grep "BEGIN PUBLIC KEY" locates instantly -- and, more to the point, OVERWRITES easily. Replace those bytes with the attacker's own public key and your application verifies licenses the attacker signed themselves. A plain embedded key is a clean overwrite target.

The examples/ folder ships a pattern that removes that target:

embed_public_key.py turns your public.pem into hidden_public_key.h, which stores the key as two independently masked byte tables that hidden_key_example. cpp reconstructs at runtime. No PEM block appears in the binary, so there is nothing for a strings scan to find or cleanly overwrite; and the accessor reconstructs BOTH tables and refuses to return a key unless they agree, so a substitution attack must patch both consistently rather than change one blob. Run the generator whenever you rotate your key -- it is deterministic, so an unchanged key produces an unchanged header.

Honest scope. As with obfuscation (§9.8), this is deterrence, not prevention. The masking is not encryption: the values needed to reconstruct the key are in the binary, so a determined reverser can still recover the key and, with more effort, produce a consistent patch. Its value is removing the low-effort grep-and-overwrite attack and giving an obfuscated build (§9.8) a reconstruct path to protect -- the two layers are strongest together. The durable protections remain the ones on your side of the boundary: node-locking, the floating server (§5), and gating real functionality on license-unlocked data rather than a single boolean -- strongest, the load-bearing pattern in §9.11.

9.10 Symbol Exposure and Stripping

The integrity self-check (§9.3), obfuscation (§9.8) and public-key hiding (§9.9) each protect a specific asset. This subsection covers a cheaper win an attacker takes first: the SYMBOL NAMES in your shipped binary. If check_node_locked, check_feature and the verifier glue appear in the symbol table, a reverser runs nm once and jumps straight to the licence logic instead of reading disassembly to find it. Removing those names does not make the check unbreakable; it removes the map that tells the attacker where to start.

Measure before you change anything -- the right action differs by how you link, and guessing wastes effort:

```
nm -D --defined-only <artifact> | grep -ci rockyguard   # export table (.dynsym)
nm    --defined-only <artifact> | grep -ci rockyguard   # symbol table (.symtab)
strings -a <artifact> | grep -c 'BEGIN PUBLIC KEY'      # greppable key? (see §9.9)
file <artifact>                                         # "stripped" / "not stripped"
```

The .dynsym export table is what visibility flags govern; .symtab is what a plain nm reads and what stripping removes. They are different tables and respond to different actions.

Static-linked executable (you link librockyguard.a into your own .exe -- the recommended case, §3.1). RockyGuard's functions are internal to your executable, so they are already ABSENT from the export table: nm -D returns nothing to hide, and -fvisibility=hidden changes little here on its own. What leaks the names is .symtab, so STRIPPING is the high-value action -- -Wl,-s at link time, a strip pass, or a CMake install-strip step. Keep -fvisibility=hidden -fvisibility-inlines-hidden anyway as hygiene (it costs nothing and pays off the day you build a plugin or link the shared library), ensure the release build carries no -g, and keep a detached debug file for your own crash analysis (objcopy --only-keep-debug plus --add-gnu-debuglink) rather than shipping symbols.

Shared library or plugin (you ship a .so / .dll, or wrap RockyGuard in one). Here the export table IS your ABI surface and visibility does real work. Default to hidden and export only the handful of entry points the host genuinely calls; the licence functions (check_, the key accessor, the verifier glue) must NOT be exported. On ELF, a linker version script that whitelists your real exports and marks everything else local: gives precise control (verify with nm -D). On Windows the default is already export-nothing, so the mistake to avoid is the reverse of ELF: do not put __declspec(dllexport) or a .def entry on any licence symbol, and ship without the .pdb. If you consume RockyGuard as its shared librockyguard.so, the vendor already export-controls the library's own symbols -- your job is your wrapper's exports plus shipping librockyguard.sig beside the .so for the integrity check (§9.3).

Honest scope. As with §9.8 and §9.9, this is deterrence, not prevention:
stripping and visibility remove the attacker's cheap reconnaissance -- the symbol map and the one-command jump to the check -- but a determined reverser can still locate the logic in disassembly on hardware they control. The value is forcing that work and combining with the other layers: an attacker who cannot nm their way to the branch, cannot strings-and-overwrite the key (§9.9), and meets control-flow-flattened code when they arrive (§9.8) is spending real effort where a plain build gave it away. The AI Integration Guide's Section 9 (Hardening) carries the build-system snippets (CMake, qmake) and a worked before/after measurement.

9.11 Load-Bearing Licences (Consume the Result, Don't Branch on It)

Every measure above (§9.3, §9.8, §9.9, §9.10) makes the license DECISION harder to find, patch, or read. This subsection removes the decision as the thing worth patching -- and it is the single most effective step you can take, because it moves the attacker off your enforcement code and back onto the cryptography. The others are deterrence; this one changes the shape of the attack.

The universal weakness of any offline check is that "valid or invalid?" is answered on the attacker's CPU, so the answer

is a branch they can flip. A strong signature on the license does not help when your program's RESPONSE to it is one conditional. The fix is to stop branching and start DERIVING: pull a key out of the validated license and use it to decrypt data or code your program genuinely needs -- a required lookup table, a resource blob, a critical routine. Patch the check out and the key never appears, so the bytes stay encrypted: a running application that produces garbage, not a working unlock. A branch can be flipped; a missing decryption key cannot be wished away.

As of v1.4 the library and the package do the key handling and the sealing for you, so you do not invent your own scheme:

```
1. Make the keys (keep them on your build machine / in CI secrets):
       tools/license_create --gen-data-key            # 64 hex chars = 32 bytes
   Run it once for the PRODUCT data key, and once more for EACH feature
   you will bind (see "Per-feature keys" below).

2. Seal the asset your program needs, under the right key, at build time:
       tools/rg_bind_asset --feature export_pdf --in tables.bin \
           --emit-header generated/tables.enc.h --symbol kTables \
           --data-key-env RG_KEY_EXPORT_PDF
   or, from CMake, one call to rockyguard_bind_asset() from
   cmake/RockyGuardBindAsset.cmake (§7.8). Either way the output is the
   exact AES-256-GCM blob the runtime expects, embedded as a byte array;
   the plaintext never ships.

3. Wrap the SAME keys into each end-user license at mint time:
       tools/license_create ... --wrap-data-key <product hex> \
           --wrap-feature-key export_pdf=<feature hex> --output license.json
   Only the features that license is entitled to get a --wrap-feature-key.
```

4. At runtime, open the asset and USE it. There is no key to test and no verdict to branch on:

```
#include <rockyguard/license_asset.h>
#include "tables.enc.h"                      // kTables / kTables_len
rockyguard::LicenseAsset asset(verifier, "export_pdf");   // after load()
auto table = asset.open(kTables, kTables_len);            // empty on any failure
load_tables(table);            // real work; garbage/nothing without the key

LicenseAsset (Customer_API_Reference §12) captures whatever key the
license yields -- the product key with LicenseAsset(verifier), one
feature's key with LicenseAsset(verifier, "name") -- and cannot be
asked whether it got one: no ok(), no operator bool, no optional.
open() returns the plaintext or an empty vector, and the bytes go
straight into the work. Do NOT write if (table.empty()) refuse;
around it; that reintroduces the flippable branch this exists to
remove. If you use the lower-level unwrap_data_key() /
unwrap_feature_key() directly (§6.10 / §6.11 of the API reference),
the same rule applies to their std::optional.
```

5. Run the NEGATIVE test before you ship. Mint a second license from DIFFERENT keys (or use a foreign one) and exercise every bound feature. Expected: the feature is visibly broken, garbage or inert -- NOT a clean "license invalid" while the rest of the application runs. A binding that passes only the happy path may be fake. Then run python tools/rg_bind_lint <src> (§7.9) and treat its ERRORs as blocking. examples/loadbearing_asset_example.cpp demonstrates both runs; examples/FEATURES.template.md is the register that keeps the flag strings in code, build and minting identical.

unwrap_data_key() (Customer_API_Reference §6.10) -- and LicenseAsset on top of it -- returns the exact bytes you

passed to --wrap-data-key, and only when a GENUINE signature is present: the wrap key is HKDF-SHA256 over the license signature -- the one field an attacker cannot produce for a payload of their choosing -- and the sealed blob is AES-256-GCM, so a tampered signature, a corrupted blob, a foreign license, or an absent one all yield nothing (authentication failure), never a forged key. One encrypted asset ships to every customer; each license carries its own wrapped copy of the key.

Per-feature keys, and why they are wrapped rather than derived. The product data key answers "was this license genuinely signed". It does not answer "is this license entitled to THIS feature": every license carries the product key, so an asset sealed under it opens for every licensee, and entitlement would be back to the check_feature() boolean. So for each tier-gated feature you generate a SEPARATE key and wrap it with --wrap-feature-key only into the licenses entitled to that feature. A Basic license physically carries no "premium" key; LicenseAsset(verifier, "premium") captures nothing; and a patched check_feature() cannot insert a key the envelope never carried. The alternative -- deriving per-feature subkeys from the product key, HKDF(DK, feature) or similar -- was evaluated and rejected: every input to that derivation is public or comes from the one product-wide key, so a license of ANY tier yields EVERY tier's key offline. It looks like tier separation and is not. rg_bind_asset therefore seals under whichever key you hand it and derives nothing; the separation comes from what you wrap into each license.

Many gates, no shared verdict. This is also how you kill a central choke point. If several features gate on one predicate -- check_node_locked(), or a bool is_licensed() wrapper you write around it -- that one function is a single patch: flip its return and every gate opens. Instead give EACH gated feature its own data dependency: decrypt that feature's own required asset, under that feature's own key, woven into its real work. Eight features become eight independent decryptions with no shared boolean anywhere; patch any one site and only that feature breaks, into garbage, not a bypass. Routing them all through the library's single unwrap path is safe even though it is one function -- you cannot patch a function to emit a 32-byte key you do not possess, whereas a bool has only two values. That is the difference between returning a verdict and returning key material.

Do not fold a code hash into the key. It is tempting to extend the idea to binary self-integrity: encrypt the asset under KDF(data key, hash(code region)) and recompute that hash at runtime, so a patched .text derives the wrong key with no integrity branch to NOP. We evaluated this and advise AGAINST it. The reason the data key above is load-bearing is that the value folded in -- the signature -- is a SECRET the attacker must hold a genuine license to obtain. A code hash is not a secret: it is a deterministic function of the legitimate binary the attacker already has, so they never touch the crypto -- they keep a pristine copy of the region (or pin the known-good constant) and redirect the hasher to read that instead of the running image. Worse, the self-hash routine is a more distinctive, more greppable target than the branch it replaced: a routine that reads its own code section and feeds a KDF is a recognizable shape, arguably an EASIER landmark than the original jne. A mechanism whose net effect is to move the patch target somewhere more obvious has negative value. Use obfuscation to make code hard to locate; keep the shipped .sig and runtime check as the DETECTION layer (trust-on-first-use, so blind to a disk patch made before first run); and rely on the signature-bound key above for the load-bearing property. The AI Integration Guide's Section 9.6 records this analysis in full.

Honest scope -- state this to yourself plainly and do not oversell it to anyone else. This defeats the cheap, high-volume attack: the free branch-flip and the keygen-less "patch and run" crack that scripted tools automate. It forces an attacker to possess a GENUINE license and reverse your derive-and-decrypt path (compile that path into an obfuscated build, §9.8, so it cannot be trivially located). It does NOT, for an offline scheme, force signature forgery: the signature sits in the plaintext .lic the attacker holds, so one leaked genuine license carries the material that yields the key. It is also SIGNATURE-bound, not machine-bound -- a genuinely-signed but expired or wrong-machine license still unwraps -- so keep check_node_locked() and check_expiry() for hardware and expiry enforcement; this adds a crypto floor under the branch, it does not replace those checks. Neither binding CAN live in the key: node-locking tolerates 2-of-4 fingerprint drift so a replaced disk does not strand a customer, and a key derived from the live fingerprint would break on exactly

that drift; and a fixed ciphertext needs the same key on every valid day, so a key that depended on "not yet expired" would need a trusted clock that an offline scheme does not have -- it would be a branch on the end user's clock wearing a KDF. Customer_API_Reference §12.4 has the full argument. Pick a SMALL, MANDATORY asset (a lookup table the output depends on, not a cosmetic resource) so wrong-key decryption is visibly broken rather than a quietly disabled feature. Worked end to end in examples/loadbearing_asset_example.cpp and the AI Integration Guide's Section 6.5, which the shipped tooling (§7.8, §7.9) and examples/FEATURES.template.md back.

10. TROUBLESHOOTING

Start here, before working through the symptoms below: run rg_inspect (section 7.7) on the affected machine. It answers, in one pass and without any back-and-forth, most of the questions the rest of this section asks you to check by hand -- what the fingerprint actually is and which component moved, whether the license verifies and with which error code, whether the clock is right, what the network can and cannot reach, and whether the anti-rollback state files are intact. Run it BEFORE running your application again: a verification writes anchor state, which can erase the evidence that explains the original failure.

If the answer is not obvious from its summary, the encrypted report it writes is the attachment to put on the support ticket -- and when it arrives, rg_report_read --diagnose is the mode to run on it. That is the one that correlates the report's sections against each other, which is where the answers this section cannot give you by hand actually live: whether an apparent expiry is really a wrong clock, whether a hardware mismatch is a replaced adapter or the wrong licence file entirely, and whether a TLS failure is a firewall or a date. See §10.1.2.

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)
KeyMalformed (v1.3+)	LicenseVerifier::load_from_string() given a RGAK- activation key whose body is unrecoverable (non-Base32 char, truncation, unsupported format version)	10.2.6
KeyChecksumMismatch (v1.3+)	LicenseVerifier::load_from_string() given a RGAK- activation key whose CRC32 trailer disagrees with the body (almost always a single-char typo)	10.2.7
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

LicenseStatus / message	Where it surfaces	Detailed entry
ServerUnreachable	FloatingLicenseClient::checkout() / checkin()	10.5.2
MachineSeatLimitReached (v1.2.1+)	FloatingLicenseClient::checkout()	10.5.3
ReentrantCall (v1.4+)	FloatingLicenseClient::checkout() / checkin() called from inside one of that client's own heartbeat callbacks	10.5.4
InGracePeriod	LicenseVerifier::load() / check_expiry() / check_node_locked()	Not an error: license is expired but still usable; check grace_days_remaining and warn the user.
Valid	Any verification call	Not an error: success.
NotYetValid	(never returned)	Reserved enum value, unreachable by design rather than by oversight. No shipped code path returns it in any release to date. The license's issued_at field is recorded and signed but deliberately NOT compared against the local clock: doing so would reject a legitimate license on any machine whose clock runs a few minutes slow, and it would defend against no attacker, since only the holder of the vendor private key can mint a future-dated license. Enforcing it is an activation-date feature ("valid from 1 March") needing its own design for clock-skew tolerance; the enumerator is held open so that can land without renumbering. It will not be deleted for tidiness -- it sits mid-enum, so removing it would renumber every later value. If your code switches on every LicenseStatus, include this case (route it as you would MalformedFile) 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).

Since v1.4 there is a second, finer lookup keyed by the stable error CODE rather than by the status or the message wording -- see 10.1.1 immediately below. Prefer it. The sub-sections from 10.2 onwards are titled by the message text as it currently stands, which is convenient when you already have a message in hand, but that wording is not a stable contract and must not be matched on in code. The code is.

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.

10.1.1 Error Code Reference (v1.4+)

Start from the CODE, not from the sentence. As of v1.4 every result a RockyGuard call returns carries a stable error code in LicenseResult::code, printed by license_verify as (for example) "HWID_MISMATCH (HardwareMismatch)" and available programmatically via result.code_name() or result.to_json(). Look that code up in the tables below.

This is the recommended entry point in preference to the message text. The wording of a message is not a stable contract and may be reworded in any release -- the sub-sections from 10.2 onwards are titled by the wording as it stands, which is convenient for a message you already have in hand but is not something to match on in code. The code is the contract.

The tables are grouped by the symptom you arrive with rather than by the code's numeric block. Each row gives the

coarse LicenseStatus the code maps to (which is what your switch should branch on), whether the condition is transient enough to be worth retrying, and what it means and what to do. The passing codes -- OK, OK_GRACE_PERIOD_ACTIVE and the rest -- are omitted here and listed in Customer_API_Reference §4.5, which is also the full catalog and the authority on the stability contract. §4.6 documents the lookup functions, including error_info(code).retryable and the severity that marks a PASSING result still worth warning about.

The license file will not load

Code	LicenseStatus	Retry?	What it means and what to do
LICENSE_FILE_UNREADABLE	MalformedFile	no	The license file could not be opened. Check that the path is s pelled correctly, that the file exists, and that this process has permission to read it.
LICENSE_ENVELOPE_INCOMPLETE	MalformedFile	no	The license file is missing its 'payload' or 'signature' field. Re-obtain the license file; this one was truncated or edited. If it arrived pasted into an email, ask for it as a file attachment instead.
LICENSE_MALFORMED	MalformedFile	no	The license file could not be parsed. Re-obtain the license file, and do not edit it by hand -- any change invalidates the signature.
LICENSE_NOT_LOADED	MalformedFile	no	No license has been loaded yet. Call load() or load_from_string() and check its result before calling any check_* method.
SIGNATURE_INVALID	SignatureInvalid	no	The license's signature did not verify. Confirm the license was issued for this product by this vendor and that the file has not been edited; if both hold, request a replacement license.

An activation key is rejected

Code	LicenseStatus	Retry?	What it means and what to do
KEY_CHECKSUM_MISMATCH	KeyChecksumMismatch	no	The activation key's checksum does not match its body, which almost always means a character was mistyped. Re-copy the activation key from the original message rather than retyping it.
KEY_MALFORMED	KeyMalformed	no	The activation key's structure is unrecoverable. Re-copy the whole activation key including the leading 'RGAK-'. Spaces, line breaks and letter case are all tolerated, so the likeliest cause is a missing or extra character.
KEY_UNSUPPORTED_VERSION	KeyMalformed	no	The activation key uses a format version this build of RockyGuard does not understand. Upgrade to an application build carrying a newer RockyGuard, or ask your vendor for a key in the older format.

The license is refused on this machine

Code	LicenseStatus	Retry?	What it means and what to do
HWID_MISMATCH	HardwareMismatch	no	This machine's hardware does not match the machine the license is bound to. Run rg_fingerprint on this machine and ask your vendor to re-issue the license for the fingerprint it reports; a partial hardware change such as a replaced disk or network adapter still needs a re-issue.

Code	LicenseStatus	Retry?	What it means and what to do
HWID_MISSING_IN_LICENSE	HardwareMismatch	no	The license is node-locked but carries no hardware fingerprint to compare against. Ask your vendor to re-issue the license bound to this machine's fingerprint; a license intended not to be hardware-locked must be issued with a fingerprint match threshold of 0.

Expiry, grace periods and the system clock

Code	LicenseStatus	Retry?	What it means and what to do
LICENSE_EXPIRED	Expired	no	The license has expired and has no grace period. Renew the license and install the replacement file.
GRACE_PERIOD_EXPIRED	Expired	no	The license expired and its grace period has now also run out. Renew the license and install the replacement file; no further grace remains.
CLOCK_ROLLED_BACK	ClockManipulated	no	The system clock is behind real time, verified against the online time pool. Set the system clock to the correct time -- ideally enable automatic network time -- and retry.
CLOCK_AHEAD_OF_REAL_TIME	ClockManipulated	no	The system clock is ahead of real time, verified against the online time pool. Set the system clock to the correct time -- ideally enable automatic network time -- and retry.
CLOCK_ANCHOR_AHEAD	ClockManipulated	no	Local time-anchor state records a time later than the current system clock, which means either the clock was rolled back or stale anchor files were left behind by a previous install. Confirm the system clock is correct, then delete the time-anchor state files named in the message detail to re-baseline them; see Customer_Documentation section 10.4.
CLOCK_ANCHOR_TAMPERED	ClockManipulated	no	Local time-anchor state failed its integrity check: the stored state carries a well-formed integrity tag that does not match it, which means it was edited. Delete the anchor state named in the message detail, then retry with network access so the anchors are re-baselined against verified time. This case is never repaired automatically, by design. (Since v1.4.0 a malformed tag -- damage rather than editing -- is treated as an absent anchor and does not raise this code.)
CLOCK_SKEWED	ClockManipulated	no	The system clock is inconsistent with the recorded time-anchor state. Set the system clock to the correct time -- ideally enable automatic network time -- and retry.

A feature or version is not covered

Code	LicenseStatus	Retry?	What it means and what to do
FEATURE_NOT_ENTITLED	FeatureNotLicensed	no	The requested feature is not included in this license. Ask your vendor for a license that includes the feature, then install the replacement file.
VERSION_NOT_ENTITLED	VersionMismatch	no	This application version is outside the range the license permits. Run a version within the license's range, or ask your vendor for a license covering this version.

Code	LicenseStatus	Retry?	What it means and what to do
VERSION_RANGE_MALFORMED	MalformedFile	no	A version string could not be parsed, so the version check could not be decided. Pass your application version as a dotted numeric string such as "3.1.5"; if the license's own version range is the unparseable one, ask your vendor to re-issue it.

LicenseAsset or unwrap_data_key() returns nothing on a trial license

This one produces no error code at all, which is why it belongs here: the license verifies, every check passes, and only the KEY is absent.

Licenses expiring within 7 days are EVALUATION MINTS, and no data key or feature key is wrapped into one. That is deliberate, not a defect. A wrapped key is bound to the license SIGNATURE rather than to time, so it keeps working after the license expires -- which means a 7-day trial carrying production keys would hand those keys out permanently to everyone who downloaded it. The keys are therefore left out at mint time, by the license_create tool and by the licensing portal alike.

The symptom on your side is that unwrap_data_key() and unwrap_feature_key() return an empty std::optional, and LicenseAsset::open() yields nothing, while check_node_locked(), check_feature() and everything else succeed normally.

If you are BUILDING a load-bearing integration, test it with a license that expires more than 7 days out. Ask your vendor for a development license with a longer expiry; a trial license cannot exercise this path, and no amount of debugging your own code will change that.

If you are SHIPPING and your production licenses behave this way, that is a real problem -- check with your vendor that the license was minted with --wrap-data-key, since a license minted without it also yields nothing here, and the two cases look identical from the API.

The library binary itself is refused

Code	LicenseStatus	Retry?	What it means and what to do
INTEGRITY_SIGNATURE_MISSING	IntegrityCheckFailed	no	The library's integrity signature file was not found. The shared library ships with a .sig file alongside it; verify both were copied to the install directory.
INTEGRITY_HASH_MISMATCH	IntegrityCheckFailed	no	The library binary has been modified since it was signed. Re-install the library from the original distribution.
INTEGRITY_SIGNATURE_INVALID	IntegrityCheckFailed	no	The library's integrity signature did not verify. The .sig file may have been tampered with; re-install both the library and its .sig from the original distribution.
INTEGRITY_BINARY_UNREADABLE	IntegrityCheckFailed	yes	The library binary could not be read for the integrity check. Check that the file is present and not locked by another process such as a backup agent or anti-malware scanner, then retry.
INTEGRITY_SIGNATURE_ERROR	IntegrityCheckFailed	no	Verifying the library's integrity signature raised an exception. Re-install the library and its .sig from the original distribution; if it persists, send the message detail to support.
INTEGRITY_PATH_NOT_FOUND	IntegrityCheckFailed	no	The library could not determine its own file path for the integrity check. Re-install the library from the original distribution; if it persists, send the message detail to support.

Code	LicenseStatus	Retry?	What it means and what to do
INTEGRITY_CHECK_FAILED	IntegrityCheckFailed	no	The library's binary integrity self-check failed. Re-install the library from the original distribution.
INTEGRITY_RUNTIME_MODIFIED	IntegrityCheckFailed	no	The library's in-memory code was modified after it loaded -- a runtime patch, hook, or attached debugger. Reported only when the library is built with ROCKYGUARD_RUNTIME_INTEGRITY (§9.3, §12.1). Detach any debugger, profiler, or code-injection tool and run from a clean, unmodified installation; if a function-hooking APM or security agent is in use, add this process to its exclusion list. See §10.4.3.

Floating licenses and the network

Code	LicenseStatus	Retry?	What it means and what to do
NETWORK_TIMEOUT	ServerUnreachable	yes	The license server did not respond within the timeout. Check that the server is running and reachable, then retry. Persistent timeouts usually mean a firewall is dropping the connection rather than rejecting it.
NETWORK_CONNECTION_REFUSED	ServerUnreachable	yes	The license server refused the connection. Check that the floating server is running and listening on the configured port, then retry.
NETWORK_HOST_UNRESOLVED	ServerUnreachable	yes	The license server's hostname could not be resolved. Check the spelling of server_host and this machine's DNS configuration, then retry.
NETWORK_UNREACHABLE	ServerUnreachable	yes	The network route to the license server is unavailable. Check this machine's network connection and any VPN required to reach the server, then retry.
NETWORK_TLS_HANDSHAKE_FAILED	ServerUnreachable	yes	The TLS handshake with the license server failed. Check that tls_ca_cert_path names the certificate or issuing CA the server presents, and that server_host matches a name on that certificate.
NETWORK_SOCKET_ERROR	ServerUnreachable	yes	A local socket could not be created for the request. This is a local resource failure rather than a server problem; check the process's file-descriptor limit and retry.
SERVER_UNEXPECTED_RESPONSE	ServerUnreachable	yes	The license server returned a status this client does not handle. Check the floating server's log for the corresponding request, then retry.
NO_LICENSES_AVAILABLE	NoLicensesAvailable	yes	Every license in the floating pool is currently in use. Wait for another user to release a license and retry, or ask your vendor to enlarge the pool.
MACHINE_SEAT_LIMIT_REACHED	MachineSeatLimitReached	yes	This machine has reached its per-machine floating-license cap. Close another session on this machine before checking out a new one.

Code	LicenseStatus	Retry?	What it means and what to do
FLOATING_REENTRANT_CALL	ReentrantCall	no	checkout() or checkin() was called from inside one of this client's own heartbeat callbacks, which is not allowed. Move the call out of the callback: set a flag or post to a queue from the callback, and call checkout() / checkin() from another thread. See Customer_API_Reference section 8.7.2.
SERVER_SIGNATURE_INVALID	SignatureInvalid	no	The floating server's response failed signature verification, so the server may not be authentic. Check that server_public_key_pem matches the key the floating server signs with, then confirm you are pointed at the intended host.

The library will not initialize (vendor license)

Code	LicenseStatus	Retry?	What it means and what to do
VENDOR_LICENSE_FILE_UNREADABLE	MalformedFile	no	The vendor license file could not be opened. Check the path; if the file is genuinely absent, obtain a valid vendor license from the RockyGuard vendor.
VENDOR_LICENSE_ENVELOPE_INCOMPLETE	MalformedFile	no	The vendor license file is missing its 'payload' or 'signature' field. Obtain an intact vendor license file from the RockyGuard vendor.
VENDOR_LICENSE_MALFORMED	MalformedFile	no	The vendor license file could not be parsed. Obtain an intact vendor license file from the RockyGuard vendor, and do not edit it by hand.
VENDOR_SIGNATURE_INVALID	SignatureInvalid	no	The vendor license's signature did not verify. Obtain a valid vendor license from the RockyGuard vendor; the embedded vendor key cannot authenticate this file.
VENDOR_LICENSE_EXPIRED	Expired	no	The vendor license has expired. Renew your RockyGuard library license with the vendor.
VENDOR_HWID_MISMATCH	HardwareMismatch	no	The vendor license does not authorize this machine. Obtain a vendor license issued for this machine, or one using the developer-seats model that includes it.
VENDOR_HWID_MISSING	HardwareMismatch	no	The node-locked vendor license carries no hardware fingerprint. Obtain a correctly issued vendor license from the RockyGuard vendor.

Codes not listed above. The vendor-license codes appear under "The library will not initialize" because that is when they surface -- at rockyguard::init(), before any end-user license is involved -- and they are a VENDOR-side problem: the integrator's own RockyGuard license, not the license their customer was issued. An end user should never see one.

UNSPECIFIED is also omitted. No RockyGuard code path returns it; seeing it means a LicenseResult was declared in host code and some branch returned without assigning it. It maps to a failing status deliberately, so that mistake fails closed rather than granting an unlicensed run.

10.1.2 Root-Cause Reference (v1.4+)

The table above answers "what does this code mean". It cannot answer "why did this happen", because a code is produced by one check looking at one thing, and the failures that cost the most support time are the ones where two things interact.

rg_report_read --diagnose (section 7.7) reads a full diagnostic report and correlates ACROSS its sections. Each

conclusion carries a stable cause id you can branch on, the report fields it rests on, a confidence, and one action. The same engine is available as an API -- diagnose(), Customer_API_Reference section 11.8 -- if you would rather build it into your own support tooling.

Two things worth knowing before you read the list.

First, this is deliberately NOT a second copy of 10.1.1. A rule ships only when it either correlates evidence from two or more report sections, or splits one error code into distinct causes with distinct remedies. Where correlation does not change the answer, no rule fires and you get the catalog's own suggested_action from the table above, unchanged. So the list below IS the set of cases where a cross-section reading gives you something a per-code table cannot -- and it is short on purpose.

Second, where a cause below contradicts the remedy in 10.1.1, the cause is right and the table is not wrong: the table cannot see the machine's clock. RG_CLOCK_AHEAD_FAKES_EXPIRY is the case to know.

Cause id	What it says
RG_CLOCK_AHEAD_FAKES_EXPIRY	The licence has NOT expired; the machine's clock runs ahead of real time and crossed the date early. Contradicts LICENSE_EXPIRED in 10.1.1: do not issue a replacement.
RG_LICENCE_EXPIRED_CORROBORATED	The licence really has expired, confirmed against an independent time source. Renew.
RG_EXPIRY_UNCORROBORATED	It reads as expired, but nothing checked the clock it was judged against. Check the date before quoting a renewal.
RG_CLOCK_BEHIND_MASKS_EXPIRY	Verification passed, but only because the clock is slow. The machine stops working the moment the clock is corrected.
RG_ANCHOR_REPAIRED_RECURRING	A forward clock excursion was healed and the anchors re-baselined. Recurs until the underlying clock fault is fixed. Reported even when a grace-period code took the single verdict slot.
RG_ANCHOR_AHEAD_STALE	Anchors record a future time but the clock is independently confirmed correct, so they are leftovers from a previous install rather than a rollback. Names the files to delete.
RG_ANCHOR_AHEAD_AMBIGUOUS	The same symptom with no trusted time available. The engine says outright that it cannot tell, and names the one step that decides it.
RG_ANCHOR_TAMPERED	Anchor state failed its integrity check. Adds the innocent explanations -- a backup agent or a file-sync client covering the state directory -- which are commoner than the guilty one.
RG_ANCHORS_ABSENT_FIRST_RUN	All witnesses missing. Benign on a new machine, and NOT repeatable: anything that verifies creates them.
RG_ANCHOR_WORLD_WRITABLE	POSIX only. Another local account can rewrite the anti-rollback state.
RG_HW_ADAPTER_REPLACED	Exactly one component changed and it is the network adapter -- a dock, a USB adapter or a VPN virtual adapter does this with nobody touching the machine.
RG_HW_ONE_COMPONENT_CHANGED	Exactly one component changed: disk, CPU or mainboard, with the story for each.
RG_HW_PARTIAL_DRIFT	Several changed but not all: a major upgrade or a VM moved between hosts.
RG_HW_DIFFERENT_MACHINE	Nothing matches. Usually the wrong .lic file rather than a changed machine -- check your issue records before re-issuing.
RG_HW_COMPONENTS_UNREADABLE_NOW	The machine did NOT change; the platform stopped producing an identifier the licence binds to. Re-issuing here bakes the degradation in permanently -- fix readability first.

Cause id	What it says
RG_HW_DRIFT_PRE_FAILURE	Verification still passes, but a component has already moved. The only pre-failure warning in the system.
RG_HW_LICENCE_UNBOUND	An issuance fault: node-locked with a non-zero threshold and no fingerprint, so the check can never pass anywhere.
RG_CLOCK_SKEW_BREAKS_TLS	DNS and TCP work but every handshake fails -- the signature of a clock so far out that every certificate is outside its validity window. Fix the date, not the firewall.
RG_NET_TLS_INTERCEPTED	A TLS-inspecting middlebox is presenting its own certificate. Caught even though the probes report OK, because the handshake itself succeeded.
RG_NET_TRUST_STORE_EMPTY	The process loaded no certificate authorities at all.
RG_NET_DNS_BROKEN	Nothing resolves. Offline verification is unaffected.
RG_NET_EGRESS_BLOCKED	DNS works, no outbound connection does. Distinguishes refused from dropped.
RG_FLOATING_HOST_UNRESOLVED / _PORT_CLOSED / _FILTERED / _TLS_MISCONFIGURED	Your floating server specifically, split by which stage failed -- name, port, filtering, or handshake.
RG_INTEGRITY_PARTIAL_UPGRADE	Library and signature came from different builds, told apart by their file timestamps. Nothing was tampered with.
RG_INTEGRITY_BINARY_MODIFIED	Same timestamps, still mismatched: a genuine modification, or anti-malware rewriting the file.
RG_INTEGRITY_SIGNATURE_MISSING	The .sig was never deployed. Usually an installer copying by glob.
RG_INTEGRITY_BINARY_UNREADABLE	The check could not run -- a scanner or backup agent holding the file. Retryable.
RG_INTEGRITY_NOT_APPLICABLE	A static build reports ok=true and code=OK for a check that never ran. This says so, because the report invites the opposite reading.
RG_LICENCE_FILE_ABSENT / _PERMISSIONS / _LOCKED	The file could not be opened, split by which of the three reasons applies, with the Windows and POSIX wording differing because the underlying check does.
RG_LICENCE_MALFORMED	Not a readable licence -- truncated, mangled in transit, or the wrong file. Points at the digest to compare against your records.
RG_LICENCE_ENVELOPE_INCOMPLETE	Valid JSON, but missing its payload or its signature.
RG_LICENCE_SIGNATURE_INVALID	Well-formed and signed by a key this build does not verify against. Stated as the disjunction it is -- the report carries no key id and cannot decide -- with a remedy that resolves both branches in order.
RG_LICENCE_NOT_VERIFIED	This inspector build has no verification key, so nothing was judged. Explains an empty verdict list that otherwise reads as healthy.
RG_LICENCE_NOT_SUPPLIED / RG_NETWORK_NOT_PROBED	The report does not cover what you are asking about. Prevents a wrong "nothing is wrong here".
RG_UNDIAGNOSED_FAILURE	The report contains a failure no rule explains. Names the code and its site, and carries the library's own remedy through unchanged.
RG_NO_FAULT_FOUND	Nothing in the report explains a licensing fault.
RG_REPORT_UNREADABLE / _NOT_A_REPORT / RG_REPORT_COLLECTION_FAILED / RG_DIAGNOSIS_FAILED	The report itself could not be used.

Cause ids are stable in the same sense error codes are -- a spelling is never reused for a different cause, and a retired id is never reissued -- with one difference: a rule's trigger may be TIGHTENED in a patch release, so it fires on fewer reports. It is never loosened onto a different condition; that takes a new id. rule_catalog_revision in the --json output tells you whether two diagnoses ran the same rules.

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.

10.2.6 "Activation key is malformed: <reason>" (v1.3+)

Cause: LicenseVerifier::load_from_string() was given a string that begins with the "RGAK-" activation-key prefix (or its case-insensitive equivalent) but whose body is not a recoverable activation key. The detail message names the exact failure:

- "Activation key contains characters outside the Base32 alphabet ..." -- one or more characters in the body are not in A-Z / 2-7 / '-' / whitespace. The most common cause is a character substitution during transcription (e.g., '0' for 'O', '1' for 'I'). Note that the Base32 alphabet OMITS '0', '1', 'I', and 'O' precisely to avoid this confusion.
- "Activation key decodes to fewer than 5 bytes -- key is truncated." -- the body was too short to contain even the version byte + CRC32 trailer. Almost always means the user copied only part of the key.
- "Activation key uses format version <N> which this version of RockyGuard does not understand." -- the key was generated by a newer release of RockyGuard with a future format extension. Upgrade your application's RockyGuard library to a matching version, or have the vendor reissue the key in the v1.3 format.

Resolution: Recopy the full key from the source. If the source is an email, expand any auto-shortened "view full message" lines; some email clients truncate long strings at the visual boundary. If recopying fails repeatedly with the same error, ask the vendor to reissue the activation key.

10.2.7 "Activation key checksum (CRC32) does not match the key body" (v1.3+)

Cause: LicenseVerifier::load_from_string() decoded the Base32 body successfully, but the trailing 4-byte CRC32 inside the decoded payload does not match the bytes preceding it. A single-character flip in transcription is the overwhelmingly most common cause -- the CRC32 was added in v1.3 specifically to catch this case BEFORE the signature verification step runs, so the customer-support diagnostic is unambiguous.

Distinguish from §10.2.1 SignatureInvalid: SignatureInvalid means the cryptographic signature did not verify (the key was tampered with, was generated against a different vendor public key, or you are loading it into the wrong product). KeyChecksumMismatch means the customer mistyped or partially-pasted the key; the cryptographic check did not even run.

Resolution: Recopy the full key from the original source verbatim. If the key was transcribed from a phone call or printed page, paste from a digital source instead. If recopying produces the same error every time, ask the vendor to reissue.

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" / "Local time-anchor state reports a timestamp N seconds ahead of the current system clock"

In v1.3+ the library produces three distinct customer-facing messages here depending on the underlying detection path, replacing the single generic "System clock manipulation detected" wording v1.2.x emitted for all three cases (v1.2.1 Phase 4.1 Report finding DR3-1):

- "System clock is behind real time by N seconds (verified against the online time pool). Set the system clock to the correct time and retry." -- the online check found the clock significantly behind real time. Confident clock-rollback finding. Resolution: set the system clock correctly.
- "System clock is ahead of real time by N seconds (verified against the online time pool). Set the system clock to the correct time and retry." -- the online check found the clock significantly ahead. Same resolution.

- "Local time-anchor state reports a timestamp N seconds ahead of the current system clock..." -- the local anchor files report a timestamp ahead of "now". Ambiguous: this is consistent with genuine clock rollback BUT it is ALSO what happens when stale anchor files from a prior install (or from a session where the clock was set forward and later corrected) are still on disk. The message includes the recovery procedure below.

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.
- Stale .lck_cache_ / .licensecheck_anchor_ / .rg_gencount_* files from a prior install of a DIFFERENT product (sharing the same RockyGuard library) are still on disk and now look "ahead" relative to the current run. Common when uninstalling/reinstalling without cleaning the per-user state directory.

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 + "|" + licensee); §9.2 explains the derivation. If you do not know the tag, deleting every .timeanchor_ / .lck_svc_ / .lck_cache_* entry under those locations is safe -- only the per-license entries the customer's installation has touched will be regenerated, and the next run rewrites them at the correct time.

Pre-condition: the host must be online for at least the first run after deletion, OR the user can defer until they next have network connectivity. The mandatory online check that fires when all anchors are missing is what authoritatively validates "the clock is now correct" before fresh anchors are written.

As of v1.3 the library repairs this state on its own whenever it can prove the current clock is trustworthy. If the online time pool confirms the system clock agrees with real time in both directions, then the clock is demonstrably not rolled back, the anchor is simply stale, and the library re-baselines every anchor location and continues -- no manual intervention, no support call. A genuine rollback is unaffected: it fails the online drift check before reaching the repair path, and a tampered anchor (one whose integrity check fails) is never repaired. You will see a one-line WARN naming the drift that was healed; if it recurs, the system clock is being set forward, and a failing CMOS battery is the usual cause.

A different message -- the time-anchor state "failed its integrity check (HMAC mismatch)" -- is a different diagnosis, and the self-repair just described does not apply to it: an anchor's contents no longer match its integrity tag, and that verdict is never healed automatically. As of v1.3.1 the message lists every anchor location for that licence by full path, so the remedy is to delete exactly those, plus any .rg_gencount / .licensecheck_anchor state beside the licence, and re-run

once with network access. Setting ROCKYGUARD_DEBUG_TIME_ANCHOR=1 in the environment prints one line per witness -- path, stored timestamp, and verdict -- so you can see WHICH location failed rather than deleting all of them on spec. Note this diagnostic covers the anchor reads themselves and therefore prints even when verification is denied before any network access is attempted.

If the host cannot reach the internet at all (air-gapped fleets), there is no evidence either way, so the library keeps the conservative behavior and refuses -- delete the state files listed above to re-baseline, or contact your support so you can re-issue a license out of band.

Corruption versus tampering (v1.4.0). The integrity check now distinguishes two things that used to share one verdict. A tag that is the right shape -- 64 hexadecimal characters, which is what the library writes -- but does not match its timestamp is TAMPERING: the file was edited, and the permanent denial above applies. A tag that is not even the right shape (truncated, non-hex, missing) is DAMAGE: a torn write on a network share, a file-sync client that restored half a file, an anti-virus quarantine round-trip, a crash on a filesystem that does not make small writes appear whole. Damage is treated exactly like a missing anchor -- the other locations still count, and if every location is unusable the mandatory online check re-baselines them -- so it clears itself on the next run with network access and no longer needs the deletion procedure. This gives an attacker nothing: truncating an anchor was always exactly as powerful as deleting it, and deletion has always been handled. Separately, since v1.4.0 the anchor and counter files are written by atomic replace (write to a temporary, then rename over the old file) rather than truncate-and-write, so another instance of your application reading an anchor while one is being updated sees the previous value or the new one, never a partial file. (On Windows the swap can collide with a reader that has the file open at that instant; the writer retries briefly and, if the collision persists, falls back to the previous write pattern, whose worst case is an empty file -- which is the handled "absent" state, not a tamper verdict.)

10.4.2 "Library integrity check failed" (and the v1.3 precise variants)

In v1.3+ the library produces a distinct trailing customer-facing message per failure mode, replacing the single generic "Library integrity check failed. The library binary appears to have been modified." wording v1.2.x used for all of them (v1.2.1 Phase 4.1 Report finding V121-7). The variants:

- "Integrity signature file not found (<path>). The library DLL ships with a .sig file alongside it; verify both are present after copying / installing." -- the .sig file is missing. The library binary itself may be perfectly intact; this is a packaging / deployment issue, NOT actual tampering. Resolution: copy the rockyguard.sig file from your customer zip to the directory holding rockyguard.dll. (The v1.2.x generic "binary appears to have been modified" wording for THIS case was the source of finding V121-7.)
- "Library binary has been modified since signing (<path>). Re-install the library from the original distribution." -- the binary's SHA-256 does not match the hash signed in the .sig file. Genuine tampering OR an incomplete copy / partial file-write. Resolution: re-install from the original distribution.
- "Library integrity signature did not verify (<path>). The .sig file may have been tampered with; re-install both the library and its .sig from the original distribution." -- the binary hash matched what's stored in .sig, but the .sig's vendor signature is invalid. Someone replaced the .sig file with a forgery. Resolution: re-install both files from the original distribution.
- "Library binary could not be read for integrity check (<path>). The file may be locked by another process or missing on disk." -- the library file exists on disk but cannot be opened for hashing. On Windows this typically means another process opened the file with exclusive share mode (an anti-tampering tool the customer also installed, or an antivirus mid-scan). Resolution: identify the locking process; on persistent failures, allowlist the rockyguard.dll path in any host security tool.

- "Could not determine the library's own DLL path for integrity check." -- the runtime could not find its own module path. Rare; typically indicates the library was loaded in an unusual way (e.g., via a manual VirtualAlloc + memcpy rather than LoadLibrary).

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.

10.4.3 "The library's in-memory code was modified" (INTEGRITY_RUNTIME_MODIFIED, opt-in, v1.4+)

This message appears only when the library was built with the ROCKYGUARD_RUNTIME_INTEGRITY option (§12.1); the default build never emits it. It is a different diagnosis from §10.4.2: those variants are about the library FILE on disk differing from its .sig, whereas this one is the LOADED code differing from the baseline the library recorded on its first validation. A static build, which has no .sig at all, can still produce this one.

Cause: one of the library's enforcement-critical functions was patched in memory after the process started -- an inline hook, a software breakpoint, or a debugger write. The message detail names the function whose bytes changed.

Resolution: detach any debugger, profiler, or code-injection tool from the process and run the application from a clean, unmodified installation. The most common BENIGN cause is a legitimate tool the customer is running -- a function-hooking APM or EDR agent, or a profiler that instruments loaded libraries. If that is expected on the customer's machines, either add the application process to the tool's exclusion list, or ship a build with ROCKYGUARD_RUNTIME_INTEGRITY disabled (it is off by default). Note the check compares the running code against a baseline captured on the first validation, so it cannot flag a patch that was already in place at that point; launching once from a clean install is part of the remedy. This is a client-side defense running on the end user's machine: it raises the cost of an in-memory patch rather than making one impossible.

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, a handshake failure is reported under this SAME status but with LicenseErrorCode NETWORK_TLS_HANDSHAKE_FAILED (1705), and in that case the TCP connection succeeded, so ping, traceroute and firewall checks will find nothing wrong. The usual causes are a certificate whose subjectAltName does not contain the exact server_host value the client dials (the CN is not consulted), or a tls_ca_cert_path that does not name the certificate or its issuing CA. Check the SAN with the openssl command in §5.2, and run rg_inspect: its TLS detail names the exact verification error ("hostname mismatch", "unable to get local issuer certificate", "certificate has expired"), which

the library's own result does not yet carry. Since v1.4.0 both IPv4 and IPv6 are supported end to end: server_host may be an IPv6 literal or a name that resolves to IPv6 only, and the server listens on both families (its start-up log line reports "IPv4+IPv6" or "IPv4 only" if the host has IPv6 disabled). If you run a pre-v1.4.0 server, clients must reach it over IPv4.

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.

10.5.4 "called from inside one of this client's own heartbeat callbacks" (LicenseStatus::ReentrantCall, v1.4+)

Cause: Your application called FloatingLicenseClient::checkout() or checkin() from inside one of the heartbeat callbacks (on_connection_lost, on_connection_restored, on_license_revoked, on_give_up) wired on that same client. Those callbacks run on the client's internal heartbeat thread, and both methods stop and join that thread -- so an unguarded call would join the calling thread itself. Since v1.4.0 the library detects this and returns FLOATING_REENTRANT_CALL without changing anything; before v1.4.0 the same call terminated the process or deadlocked. This is a defect in the host application, not a licensing condition, and it is not something an end user can fix.

Resolution: Do not call checkout() or checkin() from a callback. Have the callback record the intent -- set an atomic flag, post a message to your UI thread, push onto a work queue -- and make the call from a different thread. is_checked_out() may be called from a callback. If you want a re-checkout after the server restarts, enable recheckout_on_lease_lost (Customer_API_Reference section 8.7.4) rather than re-implementing it in on_license_revoked. The full threading contract is in Customer_API_Reference section 8.7.2. The same rule covers destruction: do not delete or reset() the client, or let its owning scope end, from inside a callback. The destructor joins the heartbeat thread; a self-join throws out of a destructor, and that terminates the process -- and unlike checkout() / checkin() it cannot be guarded, because a destructor has no status to return. Tear the client down from the thread that owns it.

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. On the SHARED path there is no Debug DLL: a /MDd consumer links rockyguard.dll's import library without error and corrupts memory at the first call, so a Debug configuration must use the static rockyguard_mdd.lib or be compiled with the Release CRT. Since v1.4.0 the headers refuse that combination at compile time, so this item is now checked for you by the build. (§3.1, §3.2; v1.2.1+ feature, guard v1.4.0+.)
- [ ] 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.)
- [ ] Decision made on runtime code-integrity enforcement (v1.4+, opt-in). The ROCKYGUARD_RUNTIME_INTEGRITY CMake option (OFF by default) makes the library verify at runtime that its own critical functions have not been patched in memory -- catching inline hooks, software breakpoints and debugger patches the on-disk .sig check cannot see, and giving STATIC builds integrity coverage they otherwise lack. Enabling it is a behavior change: a legitimate function-hooking APM/EDR agent, a profiler, or an end user running your application under a debugger can trip INTEGRITY_RUNTIME_MODIFIED (§10.4.3). Turn it on where that false-positive surface is acceptable and record the choice in your release notes; leave it off if your customers routinely run instrumentation that hooks loaded code. (§9.3, §10.4.3.)

12.2 Runtime / threading

- [ ] LicenseVerifier::load() AND check_node_locked() are called off the UI thread on a desktop GUI app. Both run the online time-anchor verification, which can take tens of seconds worst case if the network is flaky: roughly 9 seconds inside the library (three hosts at 3 seconds each) plus OS DNS resolution time, and check_node_locked() adds a hardware collect on top. Wrap them in std::async or your application's job queue. A non-blocking load_async() API is planned for a future release; Customer_API_Reference §6.2 explains why it has not shipped yet. Note that both calls hold an internal writer lock for the duration, so other calls on the same verifier block until they return -- keep status polling off the UI's critical path rather than assuming a background load frees the object

up. On a server, call check_node_locked() at startup and on a timer, never per request (Customer_API_Reference §6.4).

- [ ] 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).
- [ ] Long-running hosts (services, daemons, desktop applications that stay open) re-validate on a timer. check_feature() and check_version() are pure lookups in the License cached by load() and never re-check expiry, the clock or integrity; a process that gates on them alone keeps every feature enabled after the licence expires. Schedule check_expiry() (hourly is a reasonable default; check_node_locked() if the hardware should be re-confirmed too) off the UI thread, and gate feature code on that result (§4.6; Customer_API_Reference §6.5).
- [ ] LicenseVerifier construction is wrapped in try/catch. This applies to the STANDARD pattern too -- a PEM compiled in as a string literal. What the compiler guarantees is that the literal is present and unmodified at runtime, not that its contents parse as a key: a literal that was truncated, wrapped wrong, or pasted without its BEGIN/END armor compiles cleanly and throws at construction. Pasting public.pem into a source file is the most common way the standard pattern goes wrong, and an AI assistant doing the paste makes it more likely, not less. Uncaught, the throw is std::terminate() on the end user's machine -- a hard crash with no actionable message. LicenseVerifier is neither copyable nor movable, so hold it in a std::optional and emplace() if it must outlive the try block; §2.4's Quick Start shows the pattern, Customer_API_Reference §6.1 the rationale. (Earlier revisions of this checklist said the guard was unnecessary for the embedded-literal case. That was wrong and contradicted both §2.4 and API §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. As

of v1.3, use_tls = true with no tls_ca_cert_path is a hard error unless you also set allow_insecure_tls = true (which accepts any certificate and warns). Consider setting require_tls: true on the server so it refuses to start without a cert/key. Also as of v1.3, a server with no cert/key will not start at all unless allow_insecure_http: true is set, and a client with use_tls = false will not construct unless allow_insecure_http = true is set - so a plaintext v1.2.x deployment needs one of those two edits before it will run on v1.3. (§5.2 "Response signing and TLS".)

- [ ] rate_limit_rps / rate_limit_burst reviewed for the deployment. Defaults (50 rps / 250 burst per IP) suit typical LAN use. If many end-user machines share one egress IP (large NAT) and cold-start simultaneously, raise rate_limit_burst; set rate_limit_rps to 0 only if a fronting proxy already rate-limits. (§5.2, "Per-IP rate limiting".)
- [ ] 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.)
- [ ] deps/OPENSSL-LICENSE.txt (or its text, folded into your own third-party-notices document) ships with your application. This applies to EVERY path above, including the static ones and the Linux shared one where no OpenSSL file appears beside your binary -- OpenSSL is linked into it, so you are redistributing OpenSSL and Apache-2.0's attribution terms bind you. deps/THIRD-PARTY-NOTICES.txt names the exact version this zip carries. (§3.4 "Third-party attribution".)
- [ ] Linux artifacts, if you ship Linux: static path is your_application + license.json; shared path adds librockyguard.so and librockyguard.sig, and NO separate OpenSSL files (they are inside the .so). The three checkboxes above name Windows filenames because Windows is this manual's worked example; §3.4 carries the Linux ship lists in full. Note that item 1 of §12.1's build checklist -- the Debug/Release CRT split -- does not apply on Linux at all (§3.1).
- [ ] license.json is deployed to a directory the RUNNING user can write, on every end-user machine. "+ license.json" in the items above lists what ships, not where it sits: the anti-rollback witness .lck_cache_<tag> is written next to the licence on every verification, the write fails silently when it cannot, and an install under %ProgramFiles% or /opt run non-elevated loses that witness permanently. Per-user config directories are the safe default; verify on a test install that the witness file appears. (§3.4, §9.2.)

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.)

End of Document