RockyGuard Library - API Reference

Version 1.4.0

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

About this reference:

Describes the RockyGuard v1.4.0 C++ API. Three inline markers are used: "(v1.4+)" for APIs new in v1.4.0, "(v1.3+)" for those new in v1.3.0 and "(v1.2.1+)" for those new in v1.2.1; anything unmarked has been present since v1.2.0. Everything marked ships in the release named -- nothing here is a preview or a forward-copied draft note. Companion document: docs/Customer_Documentation.txt -- the operator-facing manual; this file is the engineer-facing API reference.

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

This reference documents the v1.4.0 C++ API. Symbols marked "(v1.4+)" do not exist in v1.3.x headers -- these are enum class LicenseErrorCode (section 4.5), the LicenseResult members code / detail / suggested_action and its member functions code_name() / to_string() / to_json() (section 4.4), and everything in the new header <rockyguard/license_error.h>: DiagnosticSeverity, LicenseErrorInfo, error_info(), error_code_name(), error_code_from_name(), status_name() and error_catalog() (section 4.6); and the load-bearing licence API: LicenseVerifier::unwrap_data_key() and unwrap_feature_key() (sections 6.10, 6.11) and the LicenseAsset class in the new header <rockyguard/license_asset.h> (section 12). Symbols marked "(v1.3+)" do not exist in v1.2.1 headers, and symbols marked "(v1.2.1+)" -- LicenseVerifier::check_version(), SignatureAlgorithm::AutoDetect, LicenseStatus::VersionMismatch, LicenseStatus::MachineSeatLimitReached, and the FloatingServerConfig fields max_leases_per_machine_id, log_max_bytes, log_keep_count -- do not exist in v1.2.0 headers. Referencing either against an older build produces "undeclared identifier" / "no member named" errors. To check the version you have linked, read ROCKYGUARD_VERSION_STRING from <rockyguard/version.h>.

v1.4 also adds the License Inspector: the CLIs rg_inspect and rg_report_read (section 9.7), and the diagnostics API they are built on (section 11) -- collection, sealing, the structured health check, and root-cause diagnosis. The CLIs are what you ask an end user to run and what your support desk runs on the file that comes back; the API is there for when you would rather build either half into your own application. v1.4 also ships the build side of the load-bearing pattern: the rg_bind_asset binder with its rockyguard_bind_asset() CMake helper (section 9.8) and the rg_bind_lint linter (section 9.9).

If your installed library is older than this reference, consult the Customer_API_Reference.pdf inside the customer zip you actually received -- that copy was packaged with your binary and matches it. Prior releases' references are also frozen under docs/archive/. The website (rockyguard.dev) publishes only the latest release, so a search result can land you on a newer reference than your build.

One change in v1.3.0 is a modification rather than an addition, so it carries no "(v1.3+)" symbol marker: LicenseVerifier::license() now returns License by value instead of const License&. See "Thread safety" in the section 6 introduction.

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 reference use the new names.

Table of Contents

1. Quick Start
2. Namespace and Headers
3. Diagnostic Output
4. Enums and Types
5. License Struct
6. LicenseVerifier Class
7. HardwareFingerprint Class
8. FloatingLicenseClient Class (Premium)
9. CLI Tools Reference
10. Integration Examples
11. Diagnostics API (v1.4+)
12. LicenseAsset Class (v1.4+)

1. QUICK START

```
#include <rockyguard/rockyguard.h>
#include <optional>

static constexpr char PUBLIC_KEY[] = R"(-----BEGIN PUBLIC KEY-----
MCowBQYDK2VwAyEA...your key...
-----END PUBLIC KEY-----)";

int main() {
    // CONSOLE-APP SHAPE. load() and check_node_locked() are
    // SYNCHRONOUS and can block for ~9s or more (3 HTTPS time-check
    // hosts x 3s, plus DNS) when the network is offline or behind a
    // captive portal -- see §6.2. Blocking main() is fine here; in a
    // GUI application run this off the UI thread and show the window
    // first, or the app appears frozen on first launch. §6.4 covers
    // the same hazard for periodic re-checks.
    //
    // The constructor is the ONLY verifier call that throws, and it
    // is guarded here even though PUBLIC_KEY is a compiled-in
    // literal. A literal that was truncated or line-wrapped when it
    // was pasted compiles cleanly and throws at construction;
    // uncaught, that is std::terminate() on your end user's machine,
    // with no message and no usable exit code. §6.1 gives the full
    // reasoning. LicenseVerifier is neither copyable nor movable,
    // which is why this is optional + emplace() rather than an
    // assignment.
    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() << "\n";
        return 1;
    }
    rockyguard::LicenseVerifier& verifier = *maybe_verifier;

    auto result = verifier.load("license.json");
    if (!result) { std::cerr << result.message << "\n"; return 1; }
    result = verifier.check_node_locked();
    if (!result) { std::cerr << result.message << "\n"; return 1; }
    // Licensed and running
}
```

Use the provided CLI tools to manage licenses. See §9.

If you are integrating via an AI agent (Claude Code, Cursor, GitHub Copilot, ChatGPT, Gemini, Cody, or similar), point it at AI_INTEGRATION_GUIDE.md at the package root or at https://rockyguard.dev/AI_INTEGRATION_GUIDE.md. The guide is written in AI-imperative voice and walks the agent through the full integration deterministically (CMake wiring, public-key embedding, startup verification, feature gating). See Customer_Documentation §2.6 for the human-side description of that path.

2. NAMESPACE AND HEADERS

All API is in the rockyguard namespace.

```
#include <rockyguard/rockyguard.h>            // All-in-one include
#include <rockyguard/types.h>                 // Enums, LicenseResult
#include <rockyguard/license.h>               // License data struct
#include <rockyguard/license_verifier.h>      // License verification
#include <rockyguard/license_asset.h>         // LicenseAsset: load-bearing assets (v1.4+)
#include <rockyguard/hardware_fingerprint.h>  // Machine fingerprinting
#include <rockyguard/floating_client.h>       // Floating client (Premium)
#include <rockyguard/export.h>                // ROCKYGUARD_API export macro
#include <rockyguard/version.h>               // ROCKYGUARD_VERSION_* macros
```

The export.h header carries the ROCKYGUARD_API macro that decorates every public class and function. You normally do not include it directly -- the other headers pull it in -- but it is listed here for completeness so you know what is in include/rockyguard/. version.h, generated at library build time from CMakeLists.txt, provides ROCKYGUARD_VERSION_MAJOR / MINOR / PATCH integer macros and a VERSION_STRING constant for runtime display.

Linking with CMake (static, Windows; full link line with per-config CRT variant):

```
add_executable(your_app main.cpp)
target_include_directories(your_app PRIVATE
    ${ROCKYGUARD}/include
    ${ROCKYGUARD}/deps/include)
target_link_libraries(your_app PRIVATE
    $<IF:$<CONFIG:Debug>,
        ${ROCKYGUARD}/lib/static/rockyguard_mdd.lib,
        ${ROCKYGUARD}/lib/static/rockyguard.lib>
    ${ROCKYGUARD}/deps/lib/libssl.lib
    ${ROCKYGUARD}/deps/lib/libcrypto.lib
    ws2_32 crypt32 iphlpapi ole32 oleaut32 wbemuuid advapi32 shell32)
```

Where ${ROCKYGUARD} is the path to the extracted package directory. See Customer_Documentation §3.1 for the Linux equivalent (Linux ships a single static lib; the CRT-variant split is Windows-only).

CRT VARIANTS on Windows (v1.2.1+): the static-lib bundle ships TWO variants of the RockyGuard library so a consumer can build in either Release or Debug without LNK2038 / LNK1319 errors on _ITERATOR_DEBUG_LEVEL or RuntimeLibrary:

rockyguard.lib	/MD-built -- link from a Release consumer.
rockyguard_mdd.lib	/MDd-built -- link from a Debug 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. The customer-side CMakeLists template under examples/ uses IMPORTED_LOCATION_RELEASE / IMPORTED_LOCATION_DEBUG so a consumer running cmake --build build --config Debug automatically picks the right variant -- the explicit $<IF:$<CONFIG:Debug>,...> form above is only needed when wiring in a hand-rolled CMakeLists.

The two variants exist for the STATIC path only. rockyguard.dll is built /MD and has no /MDd counterpart, and the DLL boundary is where the mismatch stops being a link error: a /MDd consumer links the import library cleanly (import

libraries carry no CRT-mismatch record) and then exchanges std::string, std::vector and std::map objects with the DLL in a different layout -- measured with the shipped toolchain on x64, a std::string is 40 bytes under /MDd and 32 under /MD, a vector 32 against 24, a map 24 against 16 -- so LicenseResult and License are read through the wrong layout and the process corrupts memory or crashes at the first call. A Debug configuration on the shared path must therefore link the static rockyguard_mdd.lib or be compiled with the Release CRT (CMAKE_MSVC_RUNTIME_LIBRARY MultiThreadedDLL). Since v1.4.0 export.h enforces this: compiling any RockyGuard header with ROCKYGUARD_SHARED_LIB defined under the Debug CRT is a hard #error that names both remedies, so the mismatch is a one-line build failure instead of a first-call crash. It is deliberately silent on the static path, on the library's own build, and when ROCKYGUARD_ALLOW_DEBUG_CRT_WITH_DLL is defined by a consumer who has built their own Debug DLL. Customer_Documentation §3.1 and §3.2.

Planned for a future release -- not in v1.4.0, which ships only cmake/RockyGuardBindAsset.cmake: a rockyguard-config.cmake imported target so consumers can simply

```
find_package(rockyguard CONFIG REQUIRED)
target_link_libraries(your_app PRIVATE rockyguard::rockyguard)
```

with all transitive dependencies (OpenSSL, Windows system libraries, the right CRT variant) handled by the imported target. It has been held back so far because it changes the layout of the shipped package, which is cleaner to introduce alongside other packaging work than on its own. The explicit link line above remains the supported form, and will keep working alongside the imported-target form when that ships; the imported target is also the natural place to encode the CRT rule in the previous paragraph so the toolchain enforces it instead of this page.

For shared (DLL) linking, see Customer_Documentation §3.2.

3. DIAGNOSTIC OUTPUT

The library writes human-readable diagnostic messages to stderr in a small set of well-defined situations. Every line is prefixed with "[RockyGuard] WARNING: " so it is easy to identify, filter, or redirect.

Conditions that produce a stderr warning:

- License loading: license_id or product field is empty in the payload (per-license clock-manipulation detection is weaker without these). Emitted by load() / load_from_string().
- License creation: same fields empty when generating a license via the license_create CLI tool.
- Date parsing: a license issued_at or expires_at field is not valid ISO-8601. The library treats unparseable dates as Unix epoch (so any unparseable date is treated as expired); the warning makes the malformed input visible.
- Hardware fingerprint: any one of the four components (MAC, CPU, Disk, Motherboard) cannot be read on this machine. The fingerprint is still computed but is weaker.
- Integrity check (DLL builds): library binary path cannot be determined, binary cannot be read, or the integrity .sig file is not next to the DLL.
- Vendor license loading: a date field in the vendor license is not parseable.
- Floating server (Premium): logger cannot open the configured log file (server falls back to stderr-only logging).

These messages are diagnostic: they do not affect return values. The library still fails closed via LicenseStatus where the condition is security-relevant (e.g., an empty license file path returns MalformedFile regardless of whether a warning was emitted). A customer can therefore safely treat the absence of a warning as "all expected fields were valid" and can use return values for control flow.

How to suppress these warnings:

Process-wide redirect at startup (recommended for GUI hosts):

```
// C++
std::freopen("nul",       "w", stderr);  // Windows
std::freopen("/dev/null", "w", stderr);  // Linux
```

Note: this also suppresses your application's own stderr output. If you want to keep your own diagnostics, redirect stderr to a file instead and let the [RockyGuard] prefix filter the file at read time:

```
std::freopen("rockyguard.log", "a", stderr);
```

Shell-level redirect when launching the host:

```
your_app 2> /dev/null            # Linux
your_app.exe 2> NUL              # Windows cmd.exe
your_app.exe 2> $null            # PowerShell
```

A configurable logging callback (set_log_callback) that lets the host application route library diagnostics through its own log sink, with selectable severity levels, is planned for a future release; it is not in v1.4.0. It has been held back so far because it adds a new shipped public header and touches every diagnostic site in the library. The warnings remain on stderr until it ships.

4. ENUMS AND TYPES

4.1 enum class LicenseType

NodeLocked	License bound to specific hardware
Floating	License managed by a pool server (Premium)

4.2 enum class LicenseStatus

The coarse outcome, and the value to branch on. Since v1.4 every LicenseResult also carries a FINE-GRAINED code in LicenseResult::code -- see section 4.5 -- which names the specific condition behind the status. The two are orthogonal: this enum is unchanged in v1.4, and the code-to- status mapping reproduces exactly what earlier releases returned, so nothing here needs revisiting on upgrade. Reach for the code when you want to log, route, or explain; reach for the status when you want to branch.

Five of the values below have no counterpart in the v1.4 catalog, because no shipped code path returns them: NotYetValid (see the note under it), LibraryNotInitialized, TierNotAuthorized, GenerationLimitReached and MachineNotAuthorized. That omission is deliberate rather than an oversight -- inventing codes for unreachable conditions is how an enum accumulates dead values.

> (NOTE ON ORDER: this table is grouped by TOPIC for reading, which is NOT the declaration order in types.h. Since this same section warns that deleting a mid-enum value would renumber later values and break anyone persisting numeric codes, do not infer numeric values from the sequence below -- read them from types.h, which is the only authority. NotYetValid in particular is declared between MalformedFile and FeatureNotLicensed in the header, not near the end where it appears here.)

Valid	License is valid
Expired	License has expired
InGracePeriod	Expired but within grace period
HardwareMismatch	Machine doesn't match the license
SignatureInvalid	License file has been tampered with
MalformedFile	License file cannot be parsed
FeatureNotLicensed	Requested feature not in this license
NoLicensesAvailable	Floating: all licenses in use (pool exhausted)
MachineSeatLimitReached	Floating: this machine reached its per-machine cap
VersionMismatch	Application version does not satisfy version_range
ServerUnreachable	Floating: cannot reach server
LibraryNotInitialized	Vendor license not loaded (generation tools only)
TierNotAuthorized	Feature requires Premium tier
GenerationLimitReached	License generation limit exceeded
MachineNotAuthorized	Machine not authorized for generation
ClockManipulated	System clock rolled back detected
IntegrityCheckFailed	Library binary has been modified

NotYetValid	Reserved -- unreachable by design, not by oversight. No shipped code path returns it in any release to date. A switch that handles every LicenseStatus should include this case (route it the same way as MalformedFile) to silence "unhandled enum" warnings. Why issued_at is recorded but NOT enforced: comparing it against the local clock would reject a legitimate licence on any machine whose clock runs a few minutes slow -- a paying customer locked out on day one, with an error that reads like a vendor mistake. Expiry carries the same skew exposure but fails in the safe direction (slightly too much access rather than none). There is also no adversary: the only party who can mint a future-dated licence is whoever holds the vendor private key, so the check would defend against nothing and fire only on issuance errors or wrong clocks. Enforcing it is really an activation-date FEATURE ("valid from 1 March"), which needs its own design for skew tolerance and interaction with grace periods. The enumerator is held open so that feature can land without renumbering the enum. It will not be removed for tidiness: it sits mid- enum, so deleting it would renumber every later value and break anyone who persists or transmits numeric status codes.
KeyMalformed	(v1.3+) load_from_string() was given a string that begins with the activation-key prefix "RGAK-" (case-insensitive) but whose body is unrecoverable: Base32 chars outside the alphabet, truncated body, or an unsupported format-version byte. The .message field names the precise failure mode. See §6.3.
KeyChecksumMismatch	(v1.3+) An activation key decoded as Base32 but its trailing CRC32 disagreed with the body. Almost always a single-character transcription error -- a flipped char in an email-pasted key. Distinct from SignatureInvalid (which fires on cryptographic failure): on this status the customer needs to recopy the key; on SignatureInvalid the key is tampered or wrong-product. See §6.3.
ReentrantCall	(v1.4+) FloatingLicenseClient::checkout() or checkin() was called from inside one of that same client's heartbeat callbacks. Both methods stop and join the heartbeat thread, which is the thread the callback runs on, so the call is rejected rather than self-joined. The client's state is unchanged. This is a host-programming error, not a licensing condition; the fix is in your code. See §8.7.2.

4.3 enum class SignatureAlgorithm

Ed25519	Ed25519 (recommended)
RSA_SHA256	RSA with SHA-256
AutoDetect	(v1.2.1+; default for the LicenseVerifier constructor) Inspect the loaded key and pick the matching algorithm. Eliminates the customer-tracked "must remember which algo I used" state. Pass Ed25519 or RSA_SHA256 explicitly to override.

4.4 struct LicenseResult

Member	Type	Description
status	LicenseStatus	Result status (see default below)
message	std::string	Human-readable description
grace_days_remaining	int	Grace days left (0 if N/A)
code	LicenseErrorCode	(v1.4+) Fine-grained cause; see section 4.5
detail	std::string	(v1.4+) Case-specific particulars, or empty
suggested_action	std::string	(v1.4+) What the user should do

Member functions:

```
const char* code_name() const
    The stable wire name of code, e.g. "HWID_MISMATCH". Never null.
```

```
    Shorthand for error_code_name(code).

std::string to_string() const
    One line for a log: "HWID_MISMATCH (HardwareMismatch): <message>".
```

std::string to_json() const The whole result as one JSON object, for structured logs, telemetry and support tickets. Keys, in this order: code, status, severity, retryable, message, detail, suggested_action, grace_days_remaining. All strings are JSON-escaped; bytes at or above 0x80 are passed through unchanged, so a filesystem path from a non-UTF-8 codepage cannot make reporting an error throw. No trailing newline, so the object can be embedded in a larger document.

What v1.4 changed about message -- and what it did not. message keeps its role exactly: one self-contained, human-readable sentence. It is now COMPOSED as

```
summary [+ " " + detail] [+ " " + suggested_action]
```

where the summary and the suggested action come from the catalog entry for code and the detail is whatever the failing call measured. Everything a v1.3 caller read out of message is therefore still in there, and code that logs only message loses nothing on upgrade. The new members are that same text decomposed, so it can be branched on, routed, or replaced with your own wording instead of merely printed. The exact wording of message is not a stable contract and may be reworded; code is.

operator bool() returns true if status is Valid or InGracePeriod:

```
if (auto result = verifier.load("license.json")) {
    // License is valid
}
```

Default value of status. As of v1.3.1 status is default-initialised to a NON-PASSING value (currently LicenseStatus::MalformedFile). This is worth knowing because LicenseStatus::Valid is the first enumerator and therefore has the value 0: with no default, a value-initialised LicenseResult reported Valid and converted to true through operator bool, i.e. it failed OPEN. Every library entry point assigns a real status before returning, so the default is only ever observable in your own code -- typically a result declared up front and assigned on some branches but not all. Treat the specific default status as unspecified; rely only on the guarantee that it is not a passing one.

4.5 enum class LicenseErrorCode (v1.4+)

The fine-grained diagnostic code that accompanies every LicenseResult, in LicenseResult::code. Where LicenseStatus (4.2) is the coarse outcome you BRANCH on, LicenseErrorCode is the specific condition you LOG, route, and show the user a remedy for.

The two are orthogonal, not alternatives. Every code maps onto exactly one status, and the mapping reproduces what pre-v1.4 releases returned for the same input -- so upgrading changes no existing status-based behaviour. Seven distinct INTEGRITY_ codes all still report LicenseStatus::IntegrityCheckFailed; six NETWORK_ codes all still report ServerUnreachable. The library has a test (ErrorCodeApiTest.StatusAlwaysAgreesWithCatalog) whose only job is to keep that promise.

Several conditions became distinguishable for the first time in v1.4 because they had been sharing a status with no way to tell them apart: KEY_UNSUPPORTED_VERSION vs KEY_MALFORMED, GRACE_PERIOD_EXPIRED vs LICENSE_EXPIRED, VERSION_RANGE_MALFORMED vs a corrupt license file, and the six transport causes behind what used to be a single "cannot reach license server".

Stability contract

- Values are assigned EXPLICITLY, in the reserved blocks listed below. A value, once shipped, never changes, and a new code takes the next free number in its block -- so adding one never renumbers another. You may persist the numeric value or the name.
- The SCREAMING_SNAKE_CASE name is part of the contract, returned by error_code_name() and parsed by error_code_from_name(). A name is never reused for a different code.
- A code may be RETIRED -- no shipped path returns it any more -- but its value and name are kept forever, so archived logs stay decodable.
- The MEANING and remedy text below is NOT a stable contract and may be reworded. Branch on the code; display the text.

Each entry below shows the code, then its mapped LicenseStatus in brackets together with any non-default severity and the retryable flag, then what it means and what to do about it. Numeric values are deliberately not printed here -- read them from <rockyguard/types.h>, the only authority.

Block 0 - 9: sentinels

Code	Status	Meaning	Action
`OK`	Valid	License is valid.	-
`UNSPECIFIED`	MalformedFile	No verification result was recorded.	Treat this as a failure: it means a LicenseResult was declared but never assigned, so check the branch that produced it.

Block 10 - 99: passing outcomes

Code	Status	Meaning	Action
`OK_NOT_NODE_LOCKED`	Valid	Not a node-locked license, so the hardware check was skipped.	-
`OK_FEATURE_ENTITLED`	Valid	The requested feature is included in this license.	-
`OK_VERSION_ENTITLED`	Valid	The application version satisfies the license's version range.	-
`OK_GRACE_PERIOD_ACTIVE`	InGracePeriod, severity warning	The license has expired but is still inside its grace period.	Renew the license before the grace period ends to avoid an interruption.
`OK_CLOCK_ANCHOR_REPAIRED`	Valid, severity warning	The license is valid; local time-anchor state was ahead of the system clock and was re-baselined after the online time pool confirmed the clock is correct.	No action is needed now, but if this repeats the system clock is being set forward -- a failing CMOS battery is the usual cause.
`OK_LIBRARY_INITIALIZED`	Valid	The vendor license loaded and the library is initialized.	-
`OK_ALREADY_CHECKED_OUT`	Valid	A floating license is already checked out on this client.	-
`OK_CHECKED_OUT`	Valid	A floating license was checked out successfully.	-
`OK_CHECKED_IN`	Valid	The floating license was checked back in.	-
`OK_NOT_CHECKED_OUT`	Valid	No floating license was checked out, so there was nothing to check in.	-

Block 1000 - 1099: license source and envelope

Code	Status	Meaning	Action
`LICENSE_FILE_UNREADABLE`	MalformedFile	The license file could not be opened.	Check that the path is spelled correctly, that the file exists, and that this process has permission to read it.
`LICENSE_NOT_LOADED`	MalformedFile	No license has been loaded yet.	Call load() or load_from_string() and check its result before calling any check_* method.
`LICENSE_ENVELOPE_INCOMPLETE`	MalformedFile	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	The license file could not be parsed.	Re-obtain the license file, and do not edit it by hand -- any change invalidates the signature.

Block 1100 - 1199: activation keys

Code	Status	Meaning	Action
`KEY_MALFORMED`	KeyMalformed	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	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.
`KEY_CHECKSUM_MISMATCH`	KeyChecksumMismatch	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.

Block 1200 - 1299: signatures

Code	Status	Meaning	Action
`SIGNATURE_INVALID`	SignatureInvalid	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.
`VENDOR_SIGNATURE_INVALID`	SignatureInvalid	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.
`SERVER_SIGNATURE_INVALID`	SignatureInvalid	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.

Block 1300 - 1399: hardware binding

Code	Status	Meaning	Action
`HWID_MISMATCH`	HardwareMismatch	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	Status	Meaning	Action
`HWID_MISSING_IN_LICENSE`	HardwareMismatch	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.
`VENDOR_HWID_MISMATCH`	HardwareMismatch	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	The node-locked vendor license carries no hardware fingerprint.	Obtain a correctly issued vendor license from the RockyGuard vendor.

Block 1400 - 1499: validity window and system clock

Code	Status	Meaning	Action
`LICENSE_EXPIRED`	Expired	The license has expired and has no grace period.	Renew the license and install the replacement file.
`GRACE_PERIOD_EXPIRED`	Expired	The license expired and its grace period has now also run out.	Renew the license and install the replacement file; no further grace remains.
`VENDOR_LICENSE_EXPIRED`	Expired	The vendor license has expired.	Renew your RockyGuard library license with the vendor.
`CLOCK_SKEWED`	ClockManipulated	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.
`CLOCK_ROLLED_BACK`	ClockManipulated	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	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	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	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. (Since v1.4.0 a MALFORMED tag -- a torn write, a half-restored file -- is classified as damage, treated as an absent anchor, and does not raise this code.)	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.

Block 1500 - 1599: entitlement

Code	Status	Meaning	Action
`FEATURE_NOT_ENTITLED`	FeatureNotLicensed	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	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	Status	Meaning	Action
`VERSION_RANGE_MALFORMED`	MalformedFile	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.

Block 1600 - 1699: library binary integrity

Code	Status	Meaning	Action
`INTEGRITY_CHECK_FAILED`	IntegrityCheckFailed	The library's binary integrity self-check failed.	Re-install the library from the original distribution.
`INTEGRITY_SIGNATURE_MISSING`	IntegrityCheckFailed	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	The library binary has been modified since it was signed.	Re-install the library from the original distribution.
`INTEGRITY_SIGNATURE_INVALID`	IntegrityCheckFailed	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_SIGNATURE_ERROR`	IntegrityCheckFailed	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_BINARY_UNREADABLE`	IntegrityCheckFailed, retryable	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_PATH_NOT_FOUND`	IntegrityCheckFailed	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.
`INTEGRITY_RUNTIME_MODIFIED`	IntegrityCheckFailed	The library's in-memory code was modified after it loaded, which indicates a runtime patch, hook, or attached debugger. Distinct from INTEGRITY_HASH_MISMATCH (the on-disk file differing from its signature): this is the loaded image differing from itself. Reported only when the library is built with the ROCKYGUARD_RUNTIME_INTEGRITY option enabled.	Detach any debugger, profiler, or code-injection tool from this process and run the application from a clean, unmodified installation; if you use a function-hooking APM or security agent, add this process to its exclusion list.

Block 1700 - 1799: floating licensing and network transport

Code	Status	Meaning	Action
`NETWORK_TIMEOUT`	ServerUnreachable, retryable	The license server did not respond within the timeout.	Check that the server is running and reachable, then retry. Note that a call which HANGS for much longer than the 5-second socket budget without producing this code is usually stuck in DNS resolution, not in the connection -- the library cannot bound the system resolver. An IP literal for server_host removes that term entirely. Persistent timeouts usually mean a firewall is dropping the connection rather than rejecting it.
`NETWORK_CONNECTION_REFUSED`	ServerUnreachable, retryable	The license server refused the connection.	Check that the floating server is running and listening on the configured port, then retry.

Code	Status	Meaning	Action
NETWORK_HOST_UNRESOLVED	ServerUnreachable, retryable	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, retryable	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, retryable	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, retryable	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, retryable	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, retryable	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, retryable	This machine has reached its per-machine floating-license cap.	Close another session on this machine before checking out a new one.
FLOATING_REENTRANT_CALL	ReentrantCall	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 §8.7.2.

Block 1800 - 1899: vendor license (library activation)

Code	Status	Meaning	Action
VENDOR_LICENSE_FILE_UNREADABLE	MalformedFile	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	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	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.

Statuses with no code. Five LicenseStatus values have no catalog entry, because no shipped code path returns them: NotYetValid, LibraryNotInitialized, TierNotAuthorized, GenerationLimitReached and MachineNotAuthorized. They are not omitted by oversight -- see the notes on them in 4.2. If one ever becomes reachable it will gain a code in the appropriate block at that point.

4.6 <rockyguard/license_error.h> (v1.4+)

The catalog lookups. Every function here is a pure table lookup: none allocate, none throw, none touch the filesystem or the network, and all are safe to call from any thread at any time -- including before init() and while handling a failure. That is deliberate. Rendering a diagnostic must never be the thing that fails while you are trying to report a failure.

```
enum class DiagnosticSeverity { Info, Warning, Error }
```

Info	The outcome is a pass with nothing to act on.

Warning	The outcome is a PASS (operator bool() is true and your application should run) but something needs a human so on. Currently OK_GRACE_PERIOD_ACTIVE and OK_CLOCK_ANCHOR_REPAIRED. Do not gate startup on the se.
Error	The outcome is a failure. Carried by every code whose status is not Valid or InGracePeriod.

```
struct LicenseErrorInfo {
    LicenseErrorCode    code;
    const char*         name;              // "HWID_MISMATCH"
    LicenseStatus       status;
    DiagnosticSeverity  severity;
    bool                retryable;
    const char*         summary;
    const char*         suggested_action;
};
```

An aggregate whose string members point at static storage with program lifetime, so a LicenseErrorInfo -- or the reference error_info() returns -- can be held indefinitely and shared across threads without copying the strings.

retryable means "the same call, repeated later without anyone changing anything, could plausibly succeed". It marks transient conditions for backoff logic; it does not mean "retry immediately in a tight loop". The network codes, the two pool-contention codes, SERVER_UNEXPECTED_RESPONSE and INTEGRITY_BINARY_UNREADABLE (another process may hold the file open) are retryable. A bad signature is not.

```
const LicenseErrorInfo& error_info(LicenseErrorCode code)
```

Total function. An unrecognized value -- read back from an old log, or produced by a newer RockyGuard than the one you linked -- yields the UNSPECIFIED row rather than throwing or returning null.

```
const char* error_code_name(LicenseErrorCode code)

Shorthand for error_info(code).name. Never null.
```

bool error_code_from_name(const std::string& name, LicenseErrorCode* out)

Parses a name back into a code, for reading persisted logs or a code quoted in a support ticket. Returns true and writes out on an exact, case-SENSITIVE match; returns false and leaves out untouched otherwise. out may be null, which makes this a "do I know this name?" predicate.

```
const char* status_name(LicenseStatus status)
```

The enumerator spelling of a coarse status, e.g. "HardwareMismatch". Never null; an unrecognized value yields "Unknown". Provided so printing a status does not mean writing the switch by hand in every host.

```
const LicenseErrorInfo error_catalog(std::size_t count)
```

The whole catalog, for building a support page, a CLI reference table, or a lookup map in another language. Writes the row count to *count (which must not be null) and returns the first row; the array is static, contiguous, program-lifetime, and sorted by ascending numeric code.

```
Returns a pointer and a count rather than a std::vector on purpose.
This symbol crosses the shared-library boundary, and handing a
std::vector across it would tie your C++ standard library to ours.
```

Recommended handling shape

Branch on the STATUS for control flow, and use the CODE for everything you record or display. That keeps your switch small and stable while your logs stay specific.

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

rockyguard::LicenseVerifier v(kEmbeddedPublicKeyPem);
auto r = v.load(license_path);

// One line per outcome, machine-parseable, safe to ship to a
// log aggregator or paste into a support ticket.
log_line(r.to_json());

if (!r) {
    // The user sees OUR remedy rather than your paraphrase of
    // it -- these strings are written for exactly this.
    show_error(r.message, r.suggested_action);

    switch (r.status) {
        case rockyguard::LicenseStatus::Expired:
            // GRACE_PERIOD_EXPIRED and LICENSE_EXPIRED share
            // this status; the code says which conversation
            // to have.
            offer_renewal(r.code ==
                rockyguard::LicenseErrorCode::GRACE_PERIOD_EXPIRED);
            break;
        case rockyguard::LicenseStatus::HardwareMismatch:
            offer_rehost_request();
            break;
        default:
            break;
    }
    return false;
}

if (rockyguard::error_info(r.code).severity ==
        rockyguard::DiagnosticSeverity::Warning) {
    // Passing, but say something: expiring soon, or a clock
    // anomaly that was repaired and will probably recur.
    show_warning(r.message);
}
```

Retrying only the transient failures

A floating checkout can fail for reasons that will clear on their own -- a server still starting up, a seat about to be released -- and for reasons that never will. The retryable flag separates them without you enumerating codes:

```
rockyguard::LicenseResult r;
for (int attempt = 0; attempt < 5; ++attempt) {
    r = client.checkout();
    if (r) return true;
    if (!rockyguard::error_info(r.code).retryable) break;
    sleep_for_backoff(attempt);   // do not spin
}
report(r.to_json());
return false;
```

5. LICENSE STRUCT

Defined in: <rockyguard/license.h>

5.1 Fields

Member	Type	Default
license_id	std::string	"" (REQUIRED *)
licensee	std::string	""
product	std::string	"" (REQUIRED *)
version_range	std::string	""
type	LicenseType	NodeLocked
hardware_fingerprint	std::string	""
fingerprint_match_threshold	int	2
issued_at	std::string	""
expires_at	std::string	""
grace_period_days	int	0
max_concurrent_users	int	0
features	std::vector<std::string>	{}
metadata	std::map<std::string, std::string>	{}

(*) license_id and product are REQUIRED by the license generator (license_create CLI returns an error and refuses to write the license if either is empty). They are STRONGLY RECOMMENDED on the verifier side: a license that somehow reaches load() with either field empty will load successfully but the per-license clock-manipulation defense becomes weaker, and the library emits a stderr warning at load time (see §3 "Diagnostic Output"). The asymmetry is deliberate: forward compatibility with future schema versions that might omit these fields takes precedence over hard-rejecting an otherwise-valid signed license.

The metadata field:

A customer-defined string -> string map (std::map<std::string, std::string>) that the library round-trips through the signed license file without inspecting it. The full discussion -- semantics, why metadata instead of a sidecar config, recommended use cases (audit trails, anti-leak forensics, display strings, numeric limits beyond binary feature flags), copy-pasteable issuance and read-side examples, and size guidance -- is in Customer_Documentation §8.2 "Payload Fields". On the API side: set values from the license_create CLI with --metadata key=value (Customer_Documentation §7.3 has the parse-rule details); read them at verification time via verifier.license().metadata.

Date format for issued_at and expires_at:

The canonical format is ISO 8601 with explicit time and UTC timezone -- "2027-12-31T23:59:59Z" -- and that is the form license_create writes when called with --expires. The parser also accepts a date-only form ("2027-12-31"), which is interpreted as 23:59:59 UTC on that date (i.e. the license expires at end-of-day UTC on the named day). Mixing the two within one license set is fine; the verifier compares parsed UTC instants regardless of which form produced them. Any other shape -- local-timezone offsets without "Z", date-only with US-style separators, etc. -- is treated as unparseable, which the library treats fail-closed as already expired, with a warning emitted to stderr (see §3). For new licenses, prefer the full ISO 8601 form: it is unambiguous and matches what license_verify and the customer-facing CLI tools display.

5.2 from_json()

```
static License from_json(const std::string& json_str)
```

Parses a JSON payload string into a License. Throws std::exception on malformed JSON or wrong field types. Most customers never call this directly: LicenseVerifier::load() uses it internally and reports parse failures via LicenseStatus::MalformedFile rather than exceptions.

5.3 to_json()

```
std::string to_json() const
```

Serializes this License back to a JSON string. The result is the unsigned PAYLOAD form (no envelope, no signature). Used by the license generator on the vendor side; on the verifier side this is mainly useful for diagnostics ("what license is currently loaded?") via the License object returned by LicenseVerifier::license().

5.4 is_expired()

```
bool is_expired() const
```

Returns true if expires_at is set and the current system time is past it. The verifier uses this internally; you can call it directly if you want the raw boolean without the grace-period semantics that LicenseStatus::InGracePeriod provides.

5.5 is_in_grace_period()

```
bool is_in_grace_period() const
```

Returns true if the license has expired but is still within the grace_period_days window. Equivalent to (is_expired() && grace_days_remaining() > 0).

5.6 grace_days_remaining()

```
int grace_days_remaining() const
```

Returns the number of grace days remaining after expiry, clamped at zero. Returns 0 if the license has not expired or if the grace period has already elapsed.

5.7 has_feature()

```
bool has_feature(const std::string& feature) const
```

Returns true if the named feature is in the features vector (exact match, case-sensitive). The verifier's check_feature() wraps this and returns it as a LicenseResult so it composes with the rest of the verification API.

5.8 validate_for_issue() (v1.3+)

```
bool validate_for_issue(std::string* error = nullptr) const

static constexpr int kMaxGraceDays = 36500;
```

Returns true if the license is fit to be signed and issued. Rejects an empty license_id, product, or licensee; a grace_period_days outside [0, kMaxGraceDays]; or an expires_at that is neither "permanent"/empty nor a parseable ISO 8601 date. On failure, *error (if non-null) receives a human-readable reason for the first failing check; on success it is cleared. LicenseGenerator::generate_string() and the license_create CLI call this and refuse to mint when it fails (v1.3 made issuance enforcing rather than warning-only). It does NOT run during verification -- already-signed licenses, including older ones with empty ids, still verify.

5.9 is_evaluation_mint() (v1.3.2+)

```
bool is_evaluation_mint() const

static constexpr int kEvaluationMintDays = 7;
```

Returns true if this license expires within kEvaluationMintDays of the current time -- which deliberately INCLUDES an expiry already in the past, since a pre-expired license is the normal way to exercise the expiry path. Returns false for "permanent", for an empty expires_at, and for an expires_at that does not parse.

This exists for the generation side and is unlikely to be interesting to a verifying application, but it sits on the public License struct so it is documented here. LicenseGenerator meters licenses that answer true against a separate counter, so the short-lived licenses an integrator mints while wiring RockyGuard up do not consume the vendor license's max_end_user_licenses budget. Verification never consults it: a short-dated license verifies and expires exactly like any other, and nothing about a license's runtime behaviour changes because it was classified this way at issuance.

6. LICENSEVERIFIER CLASS

Defined in: <rockyguard/license_verifier.h>

Verifies end-user licenses.

Order of operations:
The expected lifecycle is: construct, load(), then any number of check_*() calls and license() accesses. Calling check_node_locked(), check_feature(), check_version(), or check_expiry() before a successful load() is defined behavior, not a crash: each returns {LicenseStatus::MalformedFile, "No license loaded"}. license() called before load() returns a default-constructed License (all fields empty / zero / NodeLocked); is_loaded() (§6.9) is the recommended way to test whether load() has succeeded. The constructor itself never reads a license -- it only loads the public key.

Note that as of v1.3 license() returns a License by value rather than a const reference. See "Thread safety" in the section 6 introduction for the change and its rationale; existing code using License l = verifier.license(); or const License& l = verifier.license(); is unaffected.

What each check method re-evaluates:
The check methods are not all equivalent. Some re-run the full expiry + anti-tampering + clock-manipulation pipeline; others are pure data lookups against the license that load() already parsed. Choose based on whether you need a fresh time check or just a "what does this license say" answer:

Method	What it re-evaluates	Side effects and threading
load() / load_from_string()	Full pipeline: signature, integrity self-check, clock-manipulation check via time anchors, expiry. Implicitly calls check_expiry() before returning.	Writes time-anchor files (§9.2). Must not race with anything else on the same verifier instance; see "Thread safety" below.
check_expiry()	Full pipeline: integrity self-check, clock-manipulation check, expiry / grace-period evaluation.	Writes time-anchor files (§9.2). Single-writer; see "Thread safety" below.
check_node_locked()	Hardware-fingerprint match against the license's hardware_fingerprint. On a hardware pass, tail-calls check_expiry() so the full integrity + clock + expiry pipeline runs too. A license that passed load() but expires between load() and check_node_locked() will surface here.	On the hardware-pass path only: writes time-anchor and generation-counter state to disk/registry AND may block on a synchronous outbound HTTPS time check (both inherited from check_expiry). Takes the WRITER lock, so it serialises against all other verifier calls. The failing path skips all of that but still collects a fresh hardware fingerprint (§7.2) -- the success path is the expensive one. See 6.4 and "Thread safety" below.
check_feature(name)	Pure features-array lookup. Does NOT re-check expiry, clock, or integrity.	None. If you need a fresh time check before calling check_feature(), call check_expiry() or check_node_locked() first and gate on its result. Safe to call concurrently from multiple threads after load().
check_version(current)	Pure version-range match against the license's version_range string. Does NOT re-check expiry, clock, or integrity.	None. Same pre-gate guidance as check_feature(). Safe to call concurrently from multiple threads after load().
is_loaded(), license()	Pure accessors; no checks of any kind. license() returns a by-value snapshot (v1.3+).	None. Safe to call concurrently from multiple threads after load(), including concurrently with a load().

Practical pattern for a long-running application: call load() at startup, then re-call check_expiry() or check_node_locked() periodically (say, hourly) to catch clock manipulation or expiry that happened mid-session; in between, check_feature() and check_version() are cheap read-only paths.

Thread safety (v1.3+):
LicenseVerifier owns an internal std::shared_mutex that serializes writers against readers automatically. v1.2.1 documented this contract but left enforcement to the caller; v1.3 implements it inside the library. The classification used by the internal lock:

```
unique_lock (writers):
  load()
  load_from_string()
  check_expiry()         -- has time-anchor + integrity-check
                            side effects; concurrent invocations
                            would race the time-anchor file writes
                            if both ran simultaneously.
  check_node_locked()    -- writer because it transitively calls
                            check_expiry() on the success path
                            (§6.4). The alternative -- treating it
                            as a reader and letting it nest a
                            writer -- would deadlock std::shared_mutex,
                            which is not recursive.

shared_lock (readers):
  check_feature()
  check_version()
  license()
  is_loaded()
```

Practical consequences for caller-side code:

- Concurrent readers are now first-class. Multiple threads calling check_feature() / check_version() / license() / is_loaded() on the same verifier instance run concurrently with no caller-side coordination. Throughput is bounded only by the work done inside each method (a few hash lookups and string comparisons -- microseconds per call).
- A writer call blocks readers and vice versa, the standard shared_mutex contract. While load() (or check_expiry() / check_node_locked()) is running, concurrent reader calls block until it returns. Reader calls block subsequent writer calls only if the readers are still in flight when the writer arrives. No starvation in practice given how brief both sides are.
- Construction is still your responsibility. The mutex protects the verifier AFTER construction; the constructor itself runs once on one thread and the standard "publish after the constructor returns" pattern still applies. Constructing on thread A and immediately handing the verifier to thread B without a release/acquire pair is undefined behavior independent of the mutex.
- license() returns a License by value (changed in v1.3). Up to v1.2.1 it returned a const License& pointing into the verifier's internal state, and this section told you to copy the result before storing it across a reload. That guidance is obsolete: the copy is now made for you, while the internal reader lock is held, so what you get back is a consistent snapshot that no later load() can disturb. No caller-side discipline is required.

```
The v1.2.1 hazard, for anyone auditing older integration code: License holds seven strings, a vector and a
    map. A concurrent load() reassigns those members and frees the heap buffers a reader may still be walking,
    so reading through a retained reference during a reload was undefined behavior. (The reference itself
    never dangled -- the License is a member object of the verifier, not a pointer -- which is why the failure
```

```
    showed up as a torn or garbled read rather than a crash on the object itself.)
```

Source compatibility: both of the common forms still compile and are now both safe.

```
rockyguard::License snap = verifier.license();       // fine
const rockyguard::License& r = verifier.license();   // fine: the temporary's lifetime is extended
```

Only a non-const reference binding stops compiling, which is deliberate -- a compile error is a better signal than a silent behavior change:

```
auto& bad = verifier.license();                      // no longer compiles

Cost is one License copy per call. license() is a metadata accessor rather than a hot path, so this is not
    expected to matter; if it ever does, tell us -- the zero-copy answer is an additive accessor returning
    std::shared_ptr<const License>, which needs no further change to this signature.
```

- Two distinct LicenseVerifier instances are completely independent. No synchronization between them is needed.

Migration note for v1.2.x callers: code that previously wrapped check_*() in its own mutex still works correctly (the locks are independent). You can remove the caller-side mutex when you're sure all your call sites are running against v1.3+. If you must support a mixed v1.2 / v1.3 binary deployment for a transition period, keep the caller-side mutex -- it's redundant but harmless.

Summary: in v1.3+, load once, publish, then read freely from many threads. The library guarantees write-vs-read serialization.

6.1 Constructor

```
explicit LicenseVerifier(const std::string& public_key_pem,
                         SignatureAlgorithm algo = SignatureAlgorithm::AutoDetect)
```

Pass your public key as a PEM string (not a file path).

The default algo is SignatureAlgorithm::AutoDetect (v1.2.1+): the library inspects the loaded public key and picks the matching signature algorithm automatically -- Ed25519 keys are verified with Ed25519, RSA keys with RSA-SHA256. The customer no longer has to track which algorithm they used at keypair creation. Existing callers that pass an explicit Ed25519 / RSA_SHA256 value continue to work; the explicit value overrides auto-detection. (v1.2.0 callers passing the previous default Ed25519 keep working unchanged; nothing breaks for them.)

If you want your binary to ACCEPT only one algorithm (e.g., a security-policy decision to reject RSA licenses signed by a stale RSA key after migrating to Ed25519), pass that algorithm explicitly. AutoDetect intentionally trusts the key; an explicit value asserts your expectation.

If the loaded key is neither Ed25519 nor RSA, the constructor throws std::runtime_error (the customer hit a key type the library does not support; this is the same surface as a malformed PEM and the existing recommendation to wrap construction in try/catch when loading from untrusted sources applies).

Exception behavior: the constructor throws std::runtime_error if public_key_pem cannot be parsed as a valid PEM-encoded public key. Once construction succeeds the verifier never throws -- every subsequent failure mode (bad license file, signature mismatch, hardware mismatch, etc.) is reported via LicenseStatus from load() / check_node_locked() / check_expiry(). The constructor is the only point in the verifier path that throws, and it does so only on malformed input you control -- the public key string you embedded into your binary at build time. "You control it" does not mean "it parses", though: see the guidance below on why the embedded-literal case still deserves a try/catch.

If your public key string is sourced from outside your build (e.g., loaded from a file or fetched at runtime), wrap

construction in try/catch:

```
try {
    rockyguard::LicenseVerifier verifier(public_key_pem);
    auto result = verifier.load("license.json");
    // ... LicenseStatus-based handling continues from here
} catch (const std::runtime_error& e) {
    std::cerr << "Bad public key: " << e.what() << "\n";
    return 1;
}
```

We recommend the try/catch for the embedded-literal case too. Earlier revisions of this section said the throw "cannot fire" when the PEM is a const char* literal compiled into your binary. That is too strong: what the compiler guarantees is that the string is present and unmodified at runtime, not that its contents parse as a key. A literal that was truncated, line-wrapped incorrectly, or pasted with a missing BEGIN/END armor line compiles cleanly and throws at construction -- and pasting public.pem into a source file is the most common way the standard pattern goes wrong, particularly when an AI assistant performs the paste (see AI_INTEGRATION_GUIDE.md section 4).

The cost of guarding is a few lines; the cost of not guarding is std::terminate() on the end user's machine, which surfaces as a hard crash with no actionable message. Note that LicenseVerifier is neither copyable nor movable, so if the verifier must outlive the try block, hold it in a std::optional and construct with emplace() rather than assigning:

```
#include <optional>

std::optional<rockyguard::LicenseVerifier> maybe_verifier;
try {
    maybe_verifier.emplace(ROCKYGUARD_PUBLIC_KEY);
} catch (const std::runtime_error& e) {
    std::cerr << "License system error: " << e.what() << "\n";
    return 1;   // see the host-type list below before copying this line
}
rockyguard::LicenseVerifier& verifier = *maybe_verifier;
```

This also avoids re-indenting an existing main() body into a try block.

Why neither copyable nor movable, and how to store one. The verifier owns three things that do not copy sensibly: an OpenSSL key handle, the parsed License, and the std::shared_mutex that guards them (§6 "Thread safety"). Copying is deleted for that reason. Moving is not provided because a moved-from verifier would need a defined state -- either every method checks for it, or use after move is undefined -- and neither is a good trade for saving a wrapper. The consequence is only that the object must be constructed in place, wherever it lives. Two patterns cover every case:

- std::optional<rockyguard::LicenseVerifier> with emplace(), as above. It works as a class member exactly as it works as a local: declare the optional as the member, emplace() in the owning object's constructor or init routine inside the try, and use *member_ thereafter. Nothing about it is specific to main().
- std::unique_ptr<rockyguard::LicenseVerifier> with std::make_unique, when a pointer reads more naturally or the owner is itself moved around:

```
std::unique_ptr<rockyguard::LicenseVerifier> verifier;
try {
    verifier = std::make_unique<rockyguard::LicenseVerifier>(ROCKYGUARD_PUBLIC_KEY);
} catch (const std::runtime_error& e) {
    std::cerr << "License system error: " << e.what() << "\n";
    return 1;
}
```

```
This is safe with the DLL build: new and delete both run in YOUR binary, on your C runtime, and the library's
    own state sits behind an internal pointer that the library allocates in the constructor and frees in the
    destructor. The CRT-matching hazard that keeps std::future and unique_ptr out of the library's return
    types does not arise when the caller does the allocation.
```

A movable verifier is a small additive change on this layout and is being considered together with the factory described next; if the wrapper is a genuine obstacle in your integration rather than a line of boilerplate, tell us.

How to refuse on failure depends on what you are integrating into. The rule is "do not proceed into licensed functionality", which is not the same as "call exit()":

- Console app / CLI: print to stderr, return non-zero from main().
- GUI app: show a dialog carrying result.message, then quit the event loop. Exiting before the main window appears looks like a crash to the end user.
- Service / daemon: log and either stay up in a disabled state or exit deliberately. A bare exit() under a supervisor usually produces a restart loop that obscures the cause.
- Plugin / shared library: return a failure code to the host and disable your features. Never call exit() or abort() -- you have no main() to return from, and terminating takes down the host application and any unsaved user work with it.

A future release will remove the throw from this path entirely by adding a non-throwing factory that reports key-loading failure through LicenseResult (LicenseStatus::KeyMalformed) like every other verifier entry point. The throwing constructor will remain supported for source compatibility. Until then, the try/catch above is the supported approach.

6.2 load()

```
LicenseResult load(const std::string& license_file_path)
```

Load and verify a license file. Checks signature, parses payload, verifies expiry, runs anti-tampering checks (clock + integrity).

BOTH LICENSE FORMS ARE ACCEPTED (v1.3+). load() reads the file and hands the contents to the same code path as load_from_string() (§6.3), so it auto-detects which form it received. You do NOT need to read a .key file into a string yourself:

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

The file extension is irrelevant; detection is by content. Anything whose first non-whitespace characters are "RGAK" is decoded as an activation key, and anything else is parsed as the JSON envelope. Leading whitespace, the dash separators inside the key, and a trailing newline are all tolerated, so a key file written by license_create --output-key -- or one a user saved from an email -- loads as-is. A key that is detected but does not decode returns KeyMalformed or KeyChecksumMismatch rather than MalformedFile, which is what distinguishes "corrupted activation key" from "not a license at all".

The same applies in reverse: load_from_string() accepts a JSON envelope just as readily as a key string. The two entry points differ only in where the bytes come from.

Safe against malformed input: if the license file contains invalid JSON, wrong value types, or missing fields, returns MalformedFile without crashing. Never throws unhandled exceptions.

Prints a warning to stderr if license_id or product is empty in the license payload. These fields are required for clock manipulation detection to work correctly per-license.

IMPORTANT - synchronous network I/O:
load() invokes the same anti-tampering pipeline as check_expiry() (§6.6), which means it CAN perform a synchronous online time check via HTTPS. The online check fires:

- ALWAYS on first run (when no time anchors exist yet, the library has no local-only way to detect a rolled-back clock, so it consults the time-anchor host pool).
- 10% of the time on subsequent runs (random; routine verification, low traffic on the host pool).

Each online check tries up to 3 hosts from a 12-host TLS pool with a 3-second per-host budget that covers TCP connect, the TLS handshake and the HTTP exchange. Worst-case wall-clock latency is bounded by 3 * (system DNS timeout + 3 seconds). On a healthy network the typical latency is sub-second; on a flaky network or behind a captive portal, the call can block for tens of seconds. When all online attempts fail (offline), load() proceeds as if the online check were skipped (does not return an error for "offline"; see §6.6 for the full state matrix).

Address families (v1.4.0+): the pool hosts are resolved over both IPv4 and IPv6 and each address is tried until one answers, so the check works on IPv6-only networks. Before v1.4.0 it was IPv4-only there, which did not fail verification -- every host was simply unreachable, so the check reported "offline" and was skipped -- but it did mean the rollback defence was silently off on such networks.

Two notes on that bound. First, before v1.4.0 it held on Linux but not on Windows: the socket send timeout that enforced it does not apply to connect() under Winsock, so a host whose packets were silently dropped (an egress-filtered network, a blackholed route) blocked for the operating system's own retransmission budget -- about 20 seconds per host on current Windows -- rather than 3. Since v1.4.0 the connect is made with an explicit deadline on every platform and the 3-second figure is real. Second, the DNS term is the one part the library cannot bound: name resolution has no portable timeout, so against a dead resolver each host can still cost the system's DNS timeout. A machine with no network at all fails fast (no route, resolution error); it is the half-broken network that is slow.

Recommendation for GUI / desktop applications: call load() from a background thread (std::thread, std::async, a worker pool, or your application's existing IO/runtime executor) and join / .get() before unblocking the UI on the result. Calling load() on the UI thread can freeze the application's first-run startup for tens of seconds when the time-anchor pool is unreachable.

```
// Recommended for any UI application:
auto fut = std::async(std::launch::async, [&]() {
    return verifier.load("license.json");
});
// ... show splash / continue UI work ...
LicenseResult r = fut.get();
```

Recommendation for server / daemon / CLI applications: calling load() on the main thread is fine; tens-of-seconds startup is acceptable for these hosts and avoids the complexity of threading. No special handling required.

Planned for a future release, not in v1.4.0: a non-blocking load_async() returning std::future<LicenseResult>, so GUI hosts would not have to roll their own async wrapper. It has been held back for two reasons worth knowing if you are wrapping load() yourself today: returning a std::future across a DLL boundary is sensitive to the consumer's C runtime matching the library's, and load() holds an internal writer lock for the whole online check -- so an async wrapper relocates the blocking rather than removing it, and every concurrent reader still stalls until it returns. Wrapping load() in std::async on your own thread has exactly the same effect and the same caveat; keep any status polling off the critical path. When a verifier-level async API does come, it will take the shape the v1.4 diagnostics collector already uses (§11.7): an opaque, pollable handle and no std::future in the signature.

Returns: Valid, InGracePeriod, Expired, SignatureInvalid, MalformedFile, ClockManipulated, IntegrityCheckFailed.

6.3 load_from_string()

```
LicenseResult load_from_string(const std::string& input)
```

Loads a license from a string instead of a file path. Accepts either of two formats; the format is auto-detected by prefix.

(a) RAW JSON ENVELOPE. The same {"payload": "...", "signature": "..."} shape your library wrote to disk in v1.2.x. This path is unchanged from v1.2.0 -- code that already passes JSON envelopes continues to work without modification.

(b) ACTIVATION KEY (v1.3+). The same signed envelope encoded as a Base32 string with the literal "RGAK-" prefix and dash-separated 5-character body groups, e.g. RGAK-A4DGB-2P67Q-RXKMN-...-Z9V77. The decoder is tolerant of mixed case, embedded whitespace / line breaks, and stray dashes (whatever an email client may have added during transport).

Detection is by prefix: a case-insensitive "RGAK" at the start of the (leading-whitespace-stripped) input picks path (b); anything else falls through to path (a). The cryptographic guarantees are identical for both paths -- the activation-key form decodes back to the same JSON envelope that load() reads from a file, and the same Ed25519/RSA signature is the only thing that authenticates the license. The decoder adds a CRC32 trailer (4 bytes inside the decoded body) so that single-character transcription errors produce a distinct status code (KeyChecksumMismatch) BEFORE the signature verification step runs -- this is a customer-support aid, NOT a security primitive.

Possible LicenseStatus values specific to the activation-key path (in addition to all the statuses load() can return). As of v1.4 each also carries a distinct LicenseErrorCode, and that is where the useful precision now lives: KEY_MALFORMED and KEY_UNSUPPORTED_VERSION both report LicenseStatus::KeyMalformed, but they are different problems -- a malformed key needs re-copying, whereas an unsupported-version key is intact and needs a newer build of your application. Before v1.4 the two were distinguishable only by reading the message text. KEY_CHECKSUM_MISMATCH maps to LicenseStatus::KeyChecksumMismatch as before. See section 4.5.

- KeyMalformed (v1.3+): the input starts with the RGAK prefix but the body contains characters outside the Base32 alphabet, is too short to decode, or uses an unsupported format version. The .message field names the precise failure.
- KeyChecksumMismatch (v1.3+): the body decoded successfully but its trailing CRC32 disagrees with the rest -- almost always a single-character transcription error. Distinguishable from SignatureInvalid, which fires when the cryptographic signature itself fails to verify.

Generating an activation key from your end is the vendor side: license_create --output-key <file> writes the key to disk; --output (the .lic file) and --output-key may be passed in the same run. The portal exposes both formats side-by-side after a license is minted.

The thread-safety contract on load_from_string() is identical to load(): both acquire the unique writer lock on the internal shared_mutex (see §6 "Thread safety"). load_from_string() incurs no file I/O, but on the activation-key path it performs Base32 decoding + CRC32 verification before delegating to the same internal pipeline load() uses; expect roughly the same latency in practice.

6.4 check_node_locked()

```
LicenseResult check_node_locked()
```

Verify the license matches this machine's hardware. Call after load().

NOT A PASSIVE CHECK -- READ THIS BEFORE PROFILING OR CALLING IT ON A HOT PATH. The name suggests a read-only hardware hash comparison, and that is only the first half of what happens. When the hardware matches, this

call goes on to do all of the following:

- WRITES TO DISK (and on Windows, the registry): the time-anchor and generation-counter state files are updated. check_node_locked() is a mutating call, not a query.
- MAY PERFORM SYNCHRONOUS NETWORK I/O: an outbound HTTPS time check can fire, blocking the calling thread for as long as that request takes. The conditions are listed under §6.2 "IMPORTANT - synchronous network I/O"; they apply here identically.
- TAKES THE WRITER LOCK, not the reader lock. It therefore serialises against every other verifier call, including the ones that look like pure reads. Two threads calling check_node_locked() concurrently do not run concurrently.

None of this is reachable when the hardware check fails: that path returns before check_expiry() runs, so there are no disk writes, no network and no expiry evaluation. It is not free, though. Every call collects a fresh hardware fingerprint (§7.2), and on Windows that is two WMI queries: tens of milliseconds warm, seconds on a cold or busy machine. So the failing path costs one hardware collect, and the passing path costs that PLUS the anti-tampering pipeline. The expensive case is the SUCCESS case, which is the opposite of the usual intuition and makes a naive benchmark of the failing path badly misleading.

How long, concretely. The pipeline's own worst case is the one §6.2 bounds for load(): up to three time-anchor hosts at 3 seconds each, so roughly 9 seconds inside the library, plus the operating system's DNS resolution time per host, which the library cannot bound. A network that blackholes port 443 produces that worst case; a machine with no network at all fails fast. Add the hardware collect on top. Whether the online check fires at all follows the §6.2 rules: always when no anchors exist yet, otherwise on 10% of calls at random. And wherever this call blocks, everything else on the same verifier blocks with it, because the writer lock is held for the whole duration: a request handler on another thread that only wanted check_feature() waits for the HTTPS attempt to time out.

Where to call it from, by host type:

- GUI / desktop application: on a background thread, exactly as §6.2 recommends for load(). Show the window first and gate the licensed UI on the result when it arrives. Calling it on the UI thread freezes the application for the full worst case above, and does so most reliably on a new customer's first launch, when no anchors exist and the online check is mandatory.
- Server / daemon / high-concurrency host: once at startup, then on a timer thread for periodic re-validation (hourly is a reasonable default), never on the request path. A per-request check_node_locked() turns every request into a potential multi-second stall AND stalls every concurrent check_feature() behind its writer lock. Cache the LicenseResult and have the hot path read the cache.
- CLI / batch tool: on the main thread is fine; the process would be waiting for the same answer anyway.

```
// Startup: worker thread in a GUI host, main thread in a daemon.
std::atomic<bool> licensed{static_cast<bool>(verifier.check_node_locked())};

// Hot path / request handler: read the cached verdict, never re-check here.
if (!licensed.load()) return refuse_request();

// Timer thread, hourly. check_expiry() re-runs the pipeline without the
// hardware collect; use check_node_locked() instead if you also want the
// hardware re-confirmed (a licence file copied to another machine mid-run).
licensed.store(static_cast<bool>(verifier.check_expiry()));
```

If you want the hardware comparison alone -- no disk writes, no network, no writer lock -- there is no single verifier method for it, but every piece is public and the comparison itself is a string operation:

```
// Once, on a worker thread (WMI on Windows, see §7.2), then kept for
```

```
// the life of the process: hardware does not change under a running
// process, so there is nothing to gain from collecting it again.
const std::string my_fp = rockyguard::HardwareFingerprint::fingerprint();

// Any thread, any time. license() takes the reader lock only; the
// rest is string comparison. No I/O of any kind.
const rockyguard::License lic = verifier.license();
const bool hw_ok =
    rockyguard::HardwareFingerprint::match_count(lic.hardware_fingerprint, my_fp)
    >= lic.fingerprint_match_threshold;
```

That is the hardware STAGE of check_node_locked() for a node-locked licence and nothing more, and it is worth being precise about what it shares with the real call and what it does not. It shares the collector: fingerprint() is the same function check_node_locked() calls, so firmware placeholder serials are already blanked (§7.2) and match_count() skips a blank slot on either side, exactly as the library does -- the two agree on every drift and virtualisation case in Customer_Documentation §4.2. It does NOT share the licence-type check: a Floating licence passes check_node_locked() with no hardware stage at all, while this snippet reports false for it unless the threshold is 0. It yields a bool where the library yields a LicenseResult carrying a code and, on a mismatch, the matched-component count in its message. And it runs none of what follows the hardware stage: the empty-fingerprint safety described below, expiry, the clock, binary integrity. Use it as a pre-flight, benchmark or diagnostic aid ("would this licence match this box?"). check_node_locked() remains the enforcement call; do not let this snippet stand in for it. The practical pattern remains the one above: check_node_locked() once at startup, cache the LicenseResult, re-validate on a timer.

Internally this is a two-stage check: first the hardware fingerprint is matched against the license's hardware_fingerprint field; if (and only if) the hardware passes, the call tail-invokes check_expiry() to re-run the full anti-tampering pipeline (clock-manipulation detection, binary integrity self-check, expiry / grace-period evaluation -- see §6.6). The single LicenseResult returned is whichever stage decided the outcome: a hardware failure short-circuits before check_expiry() runs, so the hardware-related statuses are returned directly; otherwise the result is whatever check_expiry() produced.

Returns the union of the two stages:

- Hardware stage: HardwareMismatch (also returned with the specific message "Node-locked license has no hardware fingerprint" when the license carries an empty hardware_fingerprint AND fingerprint_match_threshold is non-zero -- a node-locked license with no fingerprint is rejected by default as a safety against issuance bugs. The opt-in to bypass this safety is fingerprint_match_threshold == 0, which is the documented "intentionally not hardware-locked" path; see Customer_Documentation §4.2).
- Expiry / anti-tampering stage (only reached when the hardware passes): Valid, InGracePeriod, Expired, ClockManipulated, IntegrityCheckFailed.
- Pre-load guard: MalformedFile (with message "No license loaded") if check_node_locked() is called before a successful load(); see the "Order of operations" notes at the top of §6.

Because check_node_locked() reaches check_expiry() on the success path, it inherits check_expiry()'s side effects: it WRITES time-anchor files (§9.2) and is therefore subject to the same single-writer threading constraint as check_expiry() itself (see "Thread safety" earlier in §6). Treat check_node_locked() as a writing call when reasoning about concurrency, not a pure read.

6.5 check_feature()

```
LicenseResult check_feature(const std::string& feature_name)
```

Check if a feature is in the license. Call after load().

Returns: Valid, FeatureNotLicensed.

DOES NOT RE-CHECK EXPIRY, THE CLOCK, OR BINARY INTEGRITY. This is a pure lookup in the features array of the License cached at load() time. It answers "was this feature licensed?", never "is the licence still valid right now?". The distinction does not matter in a tool that starts, checks, and exits. It matters a great deal in anything long-running.

The failure mode, concretely: a daemon or desktop application loads a licence on Monday and thereafter gates features on check_feature() alone. The licence expires on Wednesday. check_feature() keeps returning Valid on Thursday, Friday, and for as long as the process stays up, because nothing has re-evaluated the expiry date. The same is true of a clock rollback or a failed integrity check -- those are detected by the anti-tampering pipeline, and check_feature() does not run it.

Re-validate on a schedule and gate on THAT result:

```
// At startup, and then periodically -- hourly is a reasonable
// default; align it with your tolerance for a stale licence.
auto status = verifier.check_expiry();     // or check_node_locked()
if (!status) {
    enter_unlicensed_mode(status.message);
    return;
}

// Only meaningful once the check above has passed recently.
if (verifier.check_feature("export_pdf")) {
    menu.enable_export();
}
```

check_expiry() (§6.6) is the cheaper of the two re-validation calls; check_node_locked() (§6.4) additionally re-confirms the hardware. Both write state and may perform network I/O, which is precisely why they are not folded into check_feature() -- doing so would turn every feature-gate lookup into a disk write and a potential blocking HTTPS request.

There is currently no built-in staleness policy: the library will not force a re-check for you, and it does not track how long ago the last one happened. Scheduling is the integrator's responsibility. Tell us if you would use an opt-in maximum-age setting.

6.6 check_expiry()

```
LicenseResult check_expiry()
```

Explicitly check expiry. Also runs anti-tampering checks:

- Multi-location time anchor verification.
- Online time verification via HTTPS with TLS certificate timestamps (mandatory when stored anchors are missing; otherwise 10% random during normal operation).
- Binary integrity self-check (DLL builds).

Online-check behavior, by combination of state:

State	Behavior
Anchors PRESENT, online	Drift > 1h => ClockManipulated; otherwise proceed to evaluate stored anchors (a roll-back beyond tolerance there is also ClockManipulated).
Anchors PRESENT, offline	Online check skipped silently; stored anchors still authoritative. Roll-back beyond tolerance => ClockManipulated.

State	Behavior
Anchors MISSING, online	Online time fetched and validated; drift > 1h => ClockManipulated; else fresh anchors written and check returns Valid.
Anchors MISSING, offline	Fail-open by design: fresh anchors are written using the current system clock and check returns Valid (see note below).

The (Anchors MISSING, offline) fail-open is a deliberate trade-off so that a legitimate first run on an air-gapped or briefly-disconnected machine is not blocked. The narrow attack this exposes (rolled-back clock + all anchors deleted + permanent air-gap) requires three simultaneous adversarial conditions; any one of internet connectivity, anchor presence, or normal clock advancement closes it.

There is no switch to disable the online check or to point it at an internal host, and the reason is the same threat model: the end user owns the machine, so any runtime setting the vendor could flip, the end user can flip too (a hosts-file redirect of the pool already fails certificate verification and is treated as offline for exactly this reason). A build-time form -- a pinned internal host with its CA compiled in, or an explicit offline-only build -- is what our roadmap carries; Customer_Documentation §9.2 "Deploying on intranet-only or air-gapped networks" has the practical guidance, including how to tell a mandatory-proxy network from a genuine air gap.

One of the anchor locations is a witness file, .lck_cache_<tag>, written NEXT TO license.json on every verification. If that directory is not writable by the running user -- an application installed under %ProgramFiles% or /opt and run non-elevated is the common case -- the write fails silently: no stderr line, no log-callback message, no result code, and verification still passes on the two per-user locations, so rollback detection is weaker on that machine for good and nothing says so. Deploy license.json to a per-user directory, and confirm on a test install that the witness file appears beside it. Customer_Documentation §3.4 and §9.2.

Returns: Valid, InGracePeriod, Expired, ClockManipulated, IntegrityCheckFailed. There is no ServerUnreachable status: a missing online check is treated as "no signal", never as a verification failure, so customers behind captive portals or transient network outages are not falsely rejected.

6.7 check_version() (v1.2.1+)

```
LicenseResult check_version(const std::string& current_version) const
```

Checks whether the running application's version satisfies the license's version_range field. Pass YOUR application's version as a dotted string ("3.1.5"); the comparison is component-wise and zero-pads shorter versions, so "3.0" and "3.0.0" compare equal.

Two range syntaxes are supported, and the matcher chooses based on the first non-whitespace character of the license's version_range:

Glob form (the form shown in the license_create --version examples):

""	empty -- matches any version (default; preserves v1.2.0 behavior)
"*"	matches any version
"3.*"	matches versions whose first component is 3 (3.0, 3.1.5, 3.99.99...)
"3.1.*"	matches versions whose first two components are 3.1
"3.1.5"	exact match (component-wise, zero-padded)

Comparator form (when version_range starts with <, >, =, or !):

">=3.0"	matches versions >= 3.0
"<4.0"	matches versions strictly less than 4.0
">=3.0,<4.0"	comma-separated AND -- both clauses must hold

"!=3.5.0"	excludes an exact version

"=3.1.5", "==3.1.5" exact match (alias for the bare form) Operators: < <= > >= = == !=

Like check_feature() (§6.5), check_version() DOES NOT re-check expiry, the clock, or binary integrity -- it is a pure comparison against the version_range of the License cached at load() time. In a long-running process it will keep returning Valid indefinitely after the licence has expired. See §6.5 for the failure mode and the re-validation pattern; the same scheduling applies here.

Pre-release suffixes ("1.2-rc1", "1.2+build5") are NOT supported and surface as MalformedFile. If you need them, drop the suffix at issuance time.

Empty current_version: if the license's version_range is empty or "*" (the open range), check_version() returns Valid without inspecting current_version. For any other range, an empty current_version is unparseable and returns MalformedFile with message "current_version is not parseable: "". Guard against this in your build wiring -- if you derive APP_VERSION from a build-time substitution, assert it is non-empty before calling check_version() so a misconfigured build fails loud at startup rather than silently producing a MalformedFile under a non-empty range.

Returns:

Valid	current_version satisfies the range
VersionMismatch	current_version does NOT satisfy the range
MalformedFile	current_version or the license's version_range is unparseable

Example:

```
auto result = verifier.check_version("3.2.1");
if (!result) {
    std::cerr << result.message << "\n";
    return 1;
}
```

A note on history: License::version_range existed in v1.2.0 but was informational only -- no shipped library code consulted it. v1.2.1 makes it actionable through this API. Licenses issued under v1.2.0 with informational version strings still load correctly under v1.2.1; if you do not call check_version(), behavior is unchanged.

6.8 license()

```
License license() const
```

Access the parsed license after successful load(). Returns a snapshot BY VALUE as of v1.3 -- it returned const License& through v1.2.1; see "Thread safety" in the section 6 introduction for why that changed and what it means for existing code.

```
const License lic = verifier.license();
std::cout << "Licensed to: " << lic.licensee << "\n";
```

If called before load() has succeeded, returns a default-constructed License (every field empty / zero / NodeLocked). is_loaded() (§6.9) is the recommended pre-check if you are not sure whether load() has run yet.

6.9 is_loaded()

```
bool is_loaded() const
```

Returns true if a license has been successfully parsed via load() or load_from_string() and the verifier holds a usable

License object. Returns false before the first load() call, or after a load() call that failed to PARSE the license -- MalformedFile or SignatureInvalid. It returns TRUE after a load() that parsed a structurally valid, correctly signed license but returned a policy status such as Expired or ClockManipulated: the verifier does hold a usable License object in those cases, which is what lets you read license() and report the expiry date to the user. is_loaded() answers "is there a License object to inspect", NOT "is the license currently valid" -- the check_* methods answer the second question, and you must still call them. Use this when the verifier is held by long-lived state and you need to confirm it is ready before calling check_node_locked(), check_feature(), check_version(), or check_expiry().

6.10 unwrap_data_key() (v1.4+)

```
std::optional<std::vector<uint8_t>> unwrap_data_key() const
```

The "load-bearing license" primitive. It recovers a data key that was sealed into the license at mint time with license_create --wrap-data-key (§7 of Customer_Documentation), so you can DERIVE a key from the validated license instead of BRANCHING on the verify result. Branching decides valid/invalid on the attacker's CPU -- a single patchable branch; deriving a key that decrypts data or code the program genuinely needs means a patched-out check yields no key, so the bytes stay encrypted and the app produces garbage rather than a working unlock.

Returns the exact bytes you passed to --wrap-data-key (typically a 32-byte AES key), so you decrypt your asset with the same key you encrypted it under at build time. Returns std::nullopt -- never throws -- when: no license has loaded successfully; the loaded license carries no wrapped key; or unwrapping fails authentication (a foreign, absent, or tampered license, or one whose signature does not match the one the key was sealed under).

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: the wrong signature yields a GCM authentication failure, not a forged key. The key is therefore bound to SIGNATURE validity, NOT to expiry or hardware -- a genuinely-signed but expired or wrong-machine license still unwraps. Continue to enforce expiry and node-locking with check_expiry() / check_node_locked(); this call adds a crypto floor under the branch, it does not replace those checks. §12.4 explains why neither binding can be folded into the key.

USE IT WITHOUT BRANCHING. Feed the result straight into your decryption and use the plaintext; do NOT write if (!unwrap_data_key()) refuse;, which reintroduces the flippable branch. HONEST CEILING: this defeats the free branch-flip / keygen-less crack and forces an attacker to possess a genuine license and reverse your (ideally obfuscated) derive/decrypt path. It does NOT, for an offline scheme, force signature forgery -- one leaked genuine license carries the signature that yields the key. Worked end to end in examples/loadbearing_asset_example.cpp; see also AI_INTEGRATION_GUIDE §6.5. PREFER LicenseAsset (§12), which wraps this call and has no optional to branch on; reach for unwrap_data_key() directly only when you need the raw key for a decryption the library does not perform.

Thread safety: takes the READER lock; a pure accessor like license(). Safe to call concurrently after load().

6.11 unwrap_feature_key() (v1.4+)

```
std::optional<std::vector<uint8_t>> unwrap_feature_key(const std::string& feature_name) const
```

unwrap_data_key() applied to entitlement. Recovers the key the vendor sealed for ONE feature at mint time with license_create --wrap-feature-key <name>=<hex> (Customer_Documentation §7.3). check_feature() answers "does the signed payload list this feature" -- a boolean, and therefore a branch an attacker can flip. This answers a question with no boolean in it: "did the vendor seal a key for this feature into this license". A license the vendor did not entitle carries no blob for that name, so the lookup misses before any crypto runs; a blob sealed under a different signature, or moved here from another feature's slot, fails authentication. Patching has_feature() to return true changes the check_feature()

verdict and nothing else -- it cannot conjure a key the envelope never carried.

Returns the exact bytes passed to --wrap-feature-key, or std::nullopt -- never throws -- when nothing has loaded successfully, the license carries no key for that name (the not-entitled case), or unwrapping fails authentication. Each feature's wrap key is HKDF-SHA256 over the signature with the feature name bound into the derivation, so keys do not collide across features or with the product data key.

Per-feature keys are wrapped by the vendor, not derived from the product key. Deriving them (HKDF(data_key, feature) or similar) was evaluated and rejected: every input is public or comes from the one product-wide key, so a license of any tier would yield every tier's key offline.

USE IT WITHOUT BRANCHING, exactly as with unwrap_data_key(); if (!unwrap_feature_key("x")) disable_x(); puts back the flippable branch. Deliberately there is no has_feature_key() or available_feature_keys() enumeration API -- both read as conveniences and function as invitations to write that line. PREFER LicenseAsset(verifier, "x") (§12), which has no optional to test. Same honest ceiling as §6.10: bound to signature validity, not to expiry or hardware.

Thread safety: takes the READER lock. Safe to call concurrently after load().

7. HARDWAREFINGERPRINT CLASS

Defined in: <rockyguard/hardware_fingerprint.h>

All methods are static.

7.1 HardwareComponents Struct

```
struct HardwareComponents {
    std::string mac_address;
    std::string cpu_id;
    std::string disk_serial;
    std::string motherboard_id;
};
```

The four hardware fields the library hashes into a fingerprint. Empty strings indicate the value could not be read on this machine; the library handles that case gracefully (see match_count() below). Since v1.4.0 a recognised firmware placeholder serial ("To be filled by O.E.M.", "Default string", "System Serial Number", "None", "0", a run of one repeated character, the SMBIOS sample UUID) is returned as an empty string too, for disk_serial and motherboard_id on every platform, so it is skipped by match_count() rather than hashed as an identity shared by every board or VM clone with the same firmware default. Customer_Documentation §4.2, "Placeholder serials", has the compatibility note.

7.2 collect()

```
static HardwareComponents collect()
```

Collect hardware info from this machine. Reads MAC, CPU id, disk serial, and motherboard serial via OS-specific APIs (WMI on Windows, /sys + /proc on Linux). The library also contains an IOKit-based macOS implementation, available upon request as a separate build (see Customer_Documentation §1, Supported platforms). Returns a struct with one std::string per slot; any slot the OS could not provide is left empty.

Cost and threading. This call is not cheap and is not instant. On Windows the disk and motherboard serials come from WMI, which is an out-of-process service: a warm query takes tens of milliseconds, a cold one (first WMI use since boot, or a busy machine) can take seconds, and each of the two queries is bounded by a 5-second timeout, so the worst case is on the order of 10 seconds. Two rules follow:

- Do not call collect() or fingerprint() on a UI thread. Compute the fingerprint on a worker and hand the result to the UI.
- Compute it once. Hardware does not change while your process runs; cache the HardwareComponents or the fingerprint string and reuse it for every verification rather than collecting per call.

Since v1.4.0 the Windows implementation runs its WMI queries on a short-lived thread that the library owns, regardless of which thread calls collect(). The result therefore does not depend on the calling thread's COM apartment state, and a call made from a single-threaded-apartment (STA) thread -- every WinForms, WPF and Qt UI thread -- no longer pumps window messages while it waits, so none of your paint handlers or timers can run re-entrantly from inside collect(). If the host has already initialised COM on the calling thread, in either apartment model, that state is left exactly as it was. Before v1.4.0 the queries ran on the caller's thread; the COM handling was correct in both apartment models, but an STA caller could observe message-pump re-entrancy during the call.

7.3 fingerprint()

```
static std::string fingerprint()
```

Get the full fingerprint string for this machine: SHA-256 of each component, pipe-separated in fixed order (MAC | CPU | Disk | Motherboard). Equivalent to compute_fingerprint(collect()). This is the canonical form passed to license_create's --fingerprint-value flag.

7.4 compute_fingerprint()

```
static std::string compute_fingerprint(const HardwareComponents& hw)
```

Compute the fingerprint string from given components. Same output format as fingerprint(); use this when you have already called collect() and want to inspect the components or compute the fingerprint without re-querying the OS.

7.5 match_count()

```
static int match_count(const std::string& fp_a, const std::string& fp_b)
```

Compare two fingerprints, return matching component count (0-4). Unavailable components (empty hash) are skipped on either side - they don't count as matches or mismatches.

Edge case - all four components unavailable: if every slot on one side is empty (e.g., a sandbox with no MAC, no CPU id, no disk serial, no motherboard serial), match_count returns 0. The verifier compares this against the license's fingerprint_match_threshold (default 2): with the default, LicenseStatus::HardwareMismatch is returned and the license is rejected (fail-closed). A vendor who deliberately issues a threshold-0 license has chosen "not hardware-locked" semantics, in which case an all-empty fingerprint is accepted by design.

Policy use of the skip rule: the same empty hash may be placed deliberately in any slot of the fingerprint passed to license_create --fingerprint-value, which removes that component from the count for that licence; --threshold then applies to the remaining slots. This is how a vendor locks a cloud VM to CPU + motherboard while ignoring a fabric-assigned MAC and a volume serial. Customer_Documentation §4.2, "Deployment profiles", lists the profiles.

Diagnostics: compute_fingerprint() emits one "[RockyGuard] WARNING: Hardware component '<name>' is unavailable. Fingerprint will be weaker." line per missing component on stderr at every collect, so an operator deploying on a stripped-down host sees the weakness at runtime. Important: this warning comes from the LIBRARY function, so it surfaces only when the host application links against rockyguard.lib / librockyguard.a (e.g., license_create, your own application, or any tool that calls HardwareFingerprint::compute_fingerprint). It does NOT surface from rg_fingerprint, which is a deliberately library-free standalone tool with its own platform fingerprint code (see Customer_Documentation §7.1) and therefore does not exercise the library's warning path. To diagnose hardware-component availability when rg_fingerprint output looks suspicious, run license_create (or any rockyguard-linked tool) on the same host and read its stderr.

7.6 VmInfo struct + detect_vm() (v1.3+)

```
struct VmInfo {
    bool        detected = false;
    std::string hypervisor;
};

VmInfo detect_vm();
```

Read-only inspection of CPUID and platform sysctls -- no network, no file I/O outside Linux's /sys, no privileged operations. Returns immediately (<1 ms in typical measurements).

VmInfo.detected is true when any of these signals is present:

- x86 CPUID leaf 1 ECX bit 31 (hypervisor-present) is set (Windows, Linux, x86 macOS).
- /sys/hypervisor/type is populated (Linux).
- /sys/class/dmi/id/sys_vendor matches a known hypervisor vendor string (Linux fallback, also catches AWS / GCP / Azure Hyper-V).
- sysctl kern.hv_vmm_present is non-zero (macOS, Apple Silicon).

VmInfo.hypervisor is a friendly name when known ("VMware", "Hyper-V", "KVM", "Xen", "VirtualBox", "Parallels", "QEMU", "bhyve", "ACRN", "AWS", "GCP", "Apple Hypervisor"). For unknown hypervisors that still expose the CPUID signature, the raw 12-byte signature string is returned so a customer can identify it from their support log.

The library does NOT use this value to gate verification. Licenses are valid in VMs by default. Customers who want anti-VM policy implement it themselves -- e.g., refuse to start the application when detect_vm().detected is true, or downgrade to a trial mode, or just log the event for compliance audit.

Known caveat on modern Windows: VBS (Virtualization-Based Security), HVCI (Hypervisor-Protected Code Integrity), Credential Guard, and the WSL 2 platform all run the Windows kernel inside a Hyper-V root partition on otherwise bare-metal hosts. CPUID cannot distinguish "guest of a hypervisor" from "host with VBS enabled" -- both report detected=true with hypervisor="Hyper-V". Customers writing anti-VM policy on Windows should treat the Hyper-V signal as advisory and combine it with additional probes (e.g., the WMI Win32_ComputerSystem.HypervisorPresent + Manufacturer fields, or the rg_fingerprint Motherboard ID -- a real bare-metal motherboard reports a vendor like "Dell Inc." or "ASUSTeK" while a true Hyper-V guest reports "Microsoft Corporation") if they need to differentiate.

Example -- log VM context at startup:

```
rockyguard::VmInfo vm = rockyguard::detect_vm();
if (vm.detected) {
    std::cerr << "[my-app] starting in " << vm.hypervisor << " environment\n";
}
```

Example -- anti-VM policy (vendor's choice):

```
if (rockyguard::detect_vm().detected) {
    std::cerr << "[my-app] this product is licensed for physical machines only\n";
    return 1;
}
```

8. FLOATINGLICENSECLIENT CLASS (PREMIUM)

Defined in: <rockyguard/floating_client.h>

Checks out floating licenses from a server.

Thread safety:
FloatingLicenseClient is internally synchronized. The class owns a background heartbeat thread that runs from a successful checkout() to the matching checkin() (or to destruction); checkout() and checkin() are serialized by an internal mutex. The concrete contract is:

- Construction must complete before any other thread observes the client. The constructor itself is not thread-safe (no reasonable C++ object's constructor is). Use the standard publish-after-construction pattern: construct in one thread, then hand the client to other threads via reference, pointer, or shared_ptr.
- After construction, checkout() and checkin() may be called from any thread. Both methods take an internal mutex for their full duration, so concurrent calls serialize rather than race. Two threads racing to checkout() will not cause a double checkout: the second caller observes the first call's result via the internal checked-out flag and returns {Valid, "Already checked out"} without contacting the server. Two threads racing to checkin() likewise: the second returns {Valid, "Not checked out"}. Note that because the mutex is held across the network round-trip (potentially several seconds on a slow link), a thread blocked on this mutex waits for the in-flight call to complete; do not assume checkout() and checkin() return promptly when called concurrently from contending threads.

How long a floating call can block, and the one term the library cannot bound. Every socket operation the floating client makes -- the TCP connect, and each send and receive -- carries a 5-second deadline, on every platform since v1.4.0 (before that the connect deadline did not hold under Winsock; see §6.2 for the same fix on the time-anchor path). DNS is the exception. Resolving server_host goes through the system resolver, which has no portable timeout, so against a dead or misconfigured DNS server the resolution alone can cost the operating system's own resolver budget -- commonly 30 s, occasionally more -- before the 5-second connect deadline starts running at all. A checkout(), a checkin() or a heartbeat can therefore exceed its 5-second budget by a wide margin on a half-broken network, and NETWORK_TIMEOUT (§4.5) is not what you will see while it is happening: the call is still inside getaddrinfo(). A machine with no network at all fails fast; it is the half-broken one that is slow.

Two things follow, and the first removes the problem rather than mitigating it. SET server_host TO AN IP LITERAL WHERE YOU CAN -- FloatingClientConfig accepts IPv4 and IPv6 literals (see the IPv4/IPv6 note below), and an address skips resolution entirely, which makes the 5-second bound the real one. A licence server is usually at a fixed address inside your own network, so this is often free. Where you need a name, keep the certificate's subjectAltName in step (§ the TLS identity note above). SECOND, DO NOT CALL checkout(), checkin() OR construct the client ON A UI THREAD. This is the same rule §6.2 gives for load(), for the same reason and with the same remedy: run it on a worker and publish the result. The internal mutex makes it worse than a single slow call would be -- it is held for the whole round trip, so one thread stuck in resolution stalls every other thread's checkout() and checkin() behind it.

- is_checked_out() reads an internal atomic flag directly and does NOT take the mutex. It is safe to call from any thread concurrent with anything else, including a checkout() or checkin() in flight. The returned value is advisory: between the call returning true and the caller acting on it, a different thread can complete a checkin() and flip the value. If your application needs a stable "license held for the duration of this scope" guarantee, hold that fact at your own application layer (a member flag protected by your code's own mutex, or a token returned from your own wrapper) -- the FloatingLicenseClient API exposes the lease state but does not lock it for caller-defined critical

sections.

- The internal heartbeat thread reads only fields that are immutable after construction (config, client_id, machine_id, the TLS context) plus atomic flags (checked-out, heartbeat-running, client sequence). It does not race with caller-thread access to those fields, and you cannot interact with it directly -- it is started by checkout() and stopped by checkin() or the destructor.
- The destructor is NOT safe to run concurrently with any in-flight checkout(), checkin(), or is_checked_out() from another thread. This is the standard C++ object-lifetime rule: the owner must guarantee no other thread is using the client when it goes out of scope or is deleted. The destructor itself joins the heartbeat thread and (if still checked out) sends a final checkin() to release the seat.
- Two distinct FloatingLicenseClient instances are independent and may be used concurrently from any threads with no synchronization between them. They obtain DIFFERENT client_id values but the SAME machine_id (see §8.2), so the server treats them as two independent leases originating from the same host. The operator's per-machine cap (Customer_Documentation §5.2; max_leases_per_machine_id) bounds how many concurrent leases one host may hold.

Practical pattern: a single FloatingLicenseClient instance per logical caller is the simplest model. If you genuinely need concurrent in-progress checkouts from many threads on one host (rare), the cleanest approach is one client instance per thread or per worker, letting each instance hold its own lease, and relying on the operator-side per-machine cap to bound aggregate usage. Sharing one client across threads is supported via the internal mutex, but the cost is that one thread's blocking checkout() serializes another thread's checkin() and vice versa.

8.1 FloatingClientConfig

```
struct FloatingClientConfig {
    std::string server_host          = "127.0.0.1";
    uint16_t    server_port          = 8080;
    int         heartbeat_interval_sec = 60;

std::string server_public_key_pem;  // Verify server responses
bool        use_tls = false;        // Enable TLS encryption
std::string tls_ca_cert_path;       // Server cert for TLS verification

    // v1.3+
    int                          max_heartbeat_failures = 10;
    FloatingHeartbeatCallbacks   callbacks;
    bool                         recheckout_on_lease_lost = false;
    bool                         allow_insecure_tls  = false;
    bool                         allow_insecure_http = false;
};
```

server_public_key_pem, use_tls and tls_ca_cert_path combine into layered defenses. They protect different things and are independent of each other. allow_insecure_tls and allow_insecure_http are the two opt-outs that unlock the insecure combinations; both default false and both are enforced at construction (see §8.2).

server_public_key_pem (payload signing). Set to your server's public key PEM string. The client verifies every server response against this key, so even a successful transport-layer MITM cannot forge a checkout, heartbeat, or checkin response: an attacker without the server's private key cannot produce a valid signature and the client returns SignatureInvalid. STRONGLY RECOMMENDED for any production deployment.

use_tls. Enables TLS encryption of the wire traffic. TLS protects the session secret issued by the server at checkout

(used for HMAC on subsequent requests) from passive network observers. Without TLS, that secret is transmitted in cleartext and a sniffer can capture it and forge later HMAC-authenticated heartbeats / checkins. As of v1.3 the default (false) does not stand on its own: leaving it false requires allow_insecure_http = true, or the constructor throws.

tls_ca_cert_path. Path to the server certificate (or its issuing CA) that the client uses to verify the TLS handshake. With this set, the client refuses to talk to anyone presenting a different cert.

Verification also checks the certificate's IDENTITY, not only its chain: server_host must appear in the certificate's subjectAltName -- as a DNS entry when server_host is a name, as an IP entry when it is an address; the CN is not consulted and partial wildcards are rejected. A certificate that passes the CA check but lacks that name fails the handshake. That failure is reported as LicenseStatus::ServerUnreachable with LicenseErrorCode NETWORK_TLS_HANDSHAKE_FAILED (§4.5): the TCP connection succeeded, so do not read it as a network fault. Route on the code. Customer_Documentation §5.2 has the openssl command that prints a certificate's SAN, and rg_inspect reports the exact verification error.

allow_insecure_tls (v1.3+, default false). As of v1.3, use_tls = true with an empty tls_ca_cert_path is a hard error: the constructor throws rather than silently disable certificate verification. Set allow_insecure_tls = true to deliberately accept any certificate (TLS encrypts but does not verify the peer; an active MITM with any cert can intercept). Doing so prints a one-shot MITM warning to stderr. Use only in controlled test environments; set tls_ca_cert_path for anything else.

allow_insecure_http (v1.3+, default false). As of v1.3, use_tls = false is a hard error: the constructor throws rather than silently put license traffic on the wire in cleartext. Set allow_insecure_http = true to accept plaintext deliberately. Pre-v1.3 clients simply used HTTP with no diagnostic at all.

THESE TWO FLAGS ARE DIFFERENT AXES AND ARE EASY TO CONFUSE. allow_insecure_tls relaxes certificate VERIFICATION once TLS is already on; allow_insecure_http governs whether there is any encryption at all. Setting allow_insecure_tls does not permit cleartext, and setting allow_insecure_http has no effect once use_tls is true. The fail-closed rationale is the same for both: the checkout response carries session_secret, the HMAC key for every later request, so one sniffed or intercepted exchange gives an attacker full request-forgery capability.

Recommended configurations:

Configuration	Settings	When to use
Production	use_tls = true, tls_ca_cert_path set, server_public_key_pem set	All three layers active; no single defense failure exposes the deployment.
Internal / pinned	use_tls = false, allow_insecure_http = true, server_public_key_pem set	Acceptable on a trusted network where payload authenticity is the primary concern and transport encryption is not required; the captured-session-secret risk above remains. The explicit opt-in is mandatory in v1.3+ -- without it the constructor throws.
Development only	use_tls = false, allow_insecure_http = true, server_public_key_pem empty	No transport encryption AND no payload signature verification. An active MITM can serve fake checkouts. Do not ship this configuration to end users.

Note: these flags control the CLIENT side. The server-side counterparts (signing_private_key_pem, tls_cert_path, tls_key_path in FloatingServerConfig) must be configured to match.

8.2 Constructor

```
explicit FloatingLicenseClient(const FloatingClientConfig& config)
```

Constructs a floating-license client with the given configuration. Does NOT contact the server; the first network call happens at checkout().

THROWS std::runtime_error (v1.3+) on either insecure-transport configuration, before any network activity:

- use_tls = false and allow_insecure_http = false. The message names both the fix (use_tls = true with tls_ca_cert_path) and the opt-out, and warns that allow_insecure_tls is a different setting that does not permit cleartext.
- use_tls = true, tls_ca_cert_path empty, and allow_insecure_tls = false.

Both were silent downgrades before v1.3 (plain HTTP, and SSL_VERIFY_NONE, respectively). Failing at construction means a misconfiguration surfaces at integration time rather than as a quiet production exposure, but it also means code that compiled and ran against v1.2.x can throw on first construction after upgrading. See §8.1 for the two flags and the Customer_Documentation migration notes.

At construction time, the client populates two identifiers used on every subsequent request to the server:

client_id A fresh random UUID (cryptographically random via OpenSSL RAND_bytes), unique to this FloatingLicenseClient instance. The server uses it to recognize repeated requests from the same in-process client (e.g., heartbeat refresh of an existing lease).

machine_id The host's hardware fingerprint, computed once at construction and cached for the session (no per-request platform syscalls). Same primitive used by the node-locked path: a "|"-joined string of four SHA-256 hashes (MAC address, CPU ID, disk serial, motherboard ID). The server only ever sees the hashes, never the raw hardware values. 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. Multiple concurrent FloatingLicenseClient instances on the same host produce different client_id values but the same machine_id, which is what lets the server bound per-host concurrency without collapsing legitimate concurrent sessions into one lease.

8.3 checkout()

```
LicenseResult checkout()
```

Acquire a license. Starts background heartbeat thread. If server_public_key_pem is set, verifies the server's response signature (returns SignatureInvalid if fake server detected). Always receives a session secret from the server and uses it to HMAC-authenticate subsequent checkin / heartbeat requests; this is required regardless of whether TLS is enabled, because TLS secures the transport but does not authenticate the application-level client identity (a peer who learns another client's client_id could otherwise forge eviction or heartbeat requests over the same TLS channel). Without TLS, the session secret is sent in cleartext during the checkout response and a passive sniffer could capture it; enable TLS (and pin the server certificate via tls_ca_cert_path) to close that window.

Wire data sent on every checkout / heartbeat / checkin: client_id, machine_id (the hashed hardware fingerprint described in §8.2), a fresh random nonce, a monotonic sequence number for replay protection, and the session-secret-keyed HMAC. No raw hardware identifiers, license file content, or end-user identifiers leave the host.

Returns: Valid, NoLicensesAvailable, MachineSeatLimitReached, ServerUnreachable, SignatureInvalid, MalformedFile, ReentrantCall (v1.4+; only when called from one of this client's own heartbeat callbacks -- see §8.7.2).

Resuming a lease the server still holds (v1.4.0+). If this client gave its lease up locally -- on_give_up after an outage shorter than the server's lease_timeout_sec, or a heartbeat response that failed signature verification -- and then calls checkout() again while the server still holds the lease, the server answers "already checked out" and the client RESUMES: it keeps the session secret and sequence counter it already holds, marks itself checked out, and restarts the heartbeat, which the server accepts. Before v1.4.0 the client treated that answer as a fresh lease with no secret, its next heartbeat was rejected with HTTP 403, on_license_revoked fired, and because each checkout() retry refreshed the server-side lease, the client stayed locked out of its own seat for as long as it kept retrying. The server no longer

refreshes a lease on an unauthenticated checkout; the resumed client's first heartbeat does.

NoLicensesAvailable means the global pool is exhausted across all machines. MachineSeatLimitReached means this specific machine has hit the per-machine cap configured by the operator (max_leases_per_machine_id) even though the global pool may still have seats free for other machines; the user should close another session on the same machine before retrying.

8.4 checkin()

```
LicenseResult checkin()
```

Release the license back to the server's pool. Stops the background heartbeat thread. Safe to call multiple times; subsequent calls are no-ops.

Returns: Valid, ServerUnreachable, ReentrantCall (v1.4+; only when called from one of this client's own heartbeat callbacks -- see §8.7.2).

8.5 is_checked_out()

```
bool is_checked_out() const
```

Returns true if a license is currently held (between successful checkout() and checkin() or destruction). Useful for UI gating: show "Licensed" only when this returns true.

8.6 ~FloatingLicenseClient()

```
~FloatingLicenseClient()
```

Destructor. If a license is still checked out, automatically calls checkin() on the server to release the seat. Stops the heartbeat thread cleanly. No exceptions are propagated out of the destructor -- with one case you must design around: the destructor JOINS the heartbeat thread, so it must never run on that thread. Destroying the client from inside one of its own callbacks (delete, std::unique_ptr::reset(), or the owning scope ending inside on_give_up) is a self-join, which std::thread reports by throwing std::system_error out of the destructor, and an exception leaving a destructor is std::terminate(). The re-entrancy guard on checkout() / checkin() (§8.7.2) cannot cover this: a destructor has no status to return. Set a flag in the callback and destroy the client from the thread that owns it.

8.7 Heartbeat lifecycle callbacks (v1.3+)

In v1.2.x the host application could observe lease state only by polling is_checked_out(). This collapsed three operationally distinct events into a single observable transition: transient connectivity loss, permanent server-side revocation, and long-term server outage all manifested only as is_checked_out() flipping to false. v1.3 adds an event-driven model via four std::function callbacks on FloatingHeartbeatCallbacks (a new struct that lives on FloatingClientConfig.callbacks):

```
struct FloatingHeartbeatCallbacks {
    std::function<void()>                   on_connection_lost;
    std::function<void()>                   on_connection_restored;
    std::function<void(std::string reason)>  on_license_revoked;
    std::function<void()>                   on_give_up;
};
```

Each callback is optional; unset (default-constructed empty std::function) callbacks are silently skipped. Wire only the events your host cares about.

on_connection_lost. Fires once when the first consecutive heartbeat fails with a transient error (network unreachable, 5xx, request timeout, connection refused). Does NOT fire again on subsequent consecutive failures within the same outage. Suggested host reaction: show a "connection issues, retrying..." indicator; pause non-essential network work; trigger an autosave.

on_connection_restored. Fires after a heartbeat succeeds following one or more transient failures. Symmetric counterpart to on_connection_lost: when this fires, the host can clear the connection-issues indicator. v1.3 collapses LM-X-style separate "retry" and "success" events into this single hook because for an HTTP-based heartbeat both occur in the same successful POST.

on_license_revoked(reason). Fires when the server returns a permanent error (HTTP 403 / 404) OR when a heartbeat response fails signature verification (only when server_public_key_pem is set). The lease is forfeit; the client will need a fresh checkout() to recover. The reason string carries the server's response body when available, or a short diagnostic for signature failures (e.g. "Heartbeat response signature verification failed"). Suggested host reaction: initiate graceful shutdown of licensed features; surface the reason to the user.

on_give_up. Fires when consecutive transient failures reach the configured max_heartbeat_failures budget. The lease is forfeit (checked_out flips to false) and the heartbeat thread exits. Suggested host reaction: same as on_license_revoked plus offer the user a "retry" UI affordance that calls checkout() afresh.

8.7.1 max_heartbeat_failures

```
int max_heartbeat_failures = 10;     // default
```

Retry budget for the heartbeat loop. After this many CONSECUTIVE transient failures the client gives up, marks the lease lost, and fires callbacks.on_give_up. The default of 10 with the default heartbeat_interval_sec = 60 gives the server roughly 10 minutes of grace before the client treats it as gone.

A value of 0 means "retry forever" (the v1.2.x behavior, kept for backward compatibility). A finite value is strongly recommended in production -- without it a client whose server has been permanently shut down keeps opening TCP connections to the now-closed port indefinitely.

8.7.2 Threading model

Callbacks are invoked from the FloatingLicenseClient's internal heartbeat thread, NOT from the thread that called checkout(). Two consequences:

1) Re-entrancy: callbacks MUST NOT invoke checkout() or checkin() on the same client instance. Both of those methods stop and join the heartbeat thread -- the thread the callback is running on. Since v1.4.0 the library detects this before taking any lock and rejects the call with LicenseErrorCode::FLOATING_REENTRANT_CALL (LicenseStatus::ReentrantCall). Nothing changes: the lease stays checked out, the heartbeat keeps running, and the callback returns normally. In earlier releases the same call would self-join, which the C++ runtime reports by throwing std::system_error out of the callback and out of the heartbeat thread, terminating the process; or, if another thread happened to be inside checkin() at that moment, the two would deadlock. The guard removes both failure modes, but it is a safety net, not a feature: the correct design is still to set a flag or post to a queue from the callback and call checkout() / checkin() from a different thread. is_checked_out() is safe to call from a callback -- it reads an atomic and does not touch the mutex. Calling one client's checkout() / checkin() from ANOTHER client's callback is permitted; the guard is per instance. Destroying the client from a callback is the one re-entrant act the guard cannot catch, because a destructor cannot return a status: it self-joins the heartbeat thread and terminates the process. See §8.6.

2) Synchronization: the host is responsible for synchronizing access to any captured state from inside the callback. The heartbeat thread runs concurrently with the host's other threads. Callbacks should not block -- they delay the next

heartbeat by however long they take. For slow work (logging, UI updates), capture state and dispatch to a dedicated worker thread / queue.

3) Exceptions (v1.4.0+): callbacks must not throw. One that does is caught on the heartbeat thread, reported as a single stderr line naming the callback and the exception text, and otherwise ignored -- the heartbeat continues exactly as if the callback had returned. It cannot be propagated: the heartbeat thread belongs to the library, and there is no host thread to deliver the exception to. Before v1.4.0 an exception escaping a callback reached the thread root and terminated the host process. The same guard covers the library's own code on that thread: an internal exception stops the heartbeat, forfeits the lease and fires on_give_up, rather than terminating the process.

8.7.3 Mapping to LM-X callback functions

For readers familiar with X-Formation's LM-X library, the RockyGuard hooks correspond as follows:

LM-X callback	RockyGuard hook
HEARTBEAT_CONNECTION_LOST_FUNCTION	on_connection_lost
HEARTBEAT_RETRY_FEATURE_FUNCTION	(collapsed into restored)
HEARTBEAT_CHECKOUT_SUCCESS_FUNCTION	on_connection_restored
HEARTBEAT_CHECKOUT_FAILURE_FUNCTION	on_license_revoked
HEARTBEAT_EXIT_FUNCTION	on_give_up

LM-X's separate RETRY_FEATURE and CHECKOUT_SUCCESS events are collapsed in RockyGuard because in our HTTP-based protocol both happen in the same successful POST.

8.7.4 Server-restart auto-recovery (recheckout_on_lease_lost)

```
bool recheckout_on_lease_lost = false;
```

When a floating server restarts (planned reboot, container redeploy, crash + restart) its in-memory leases vector is wiped. Every active client's next heartbeat returns 404 ("lease not found"), which is indistinguishable on the wire from an admin revocation. The default v1.3 behavior treats this as a permanent error -- fires on_license_revoked and exits the heartbeat loop. The host application has to surface a "license revoked" message even though nothing was actually revoked; the end user reconnects manually.

When recheckout_on_lease_lost is true, a heartbeat 404 / 403 triggers an automatic /checkout against the same server BEFORE the on_license_revoked path runs:

- Re-checkout succeeds (server is back up and has free seats): the client transparently picks up a fresh lease with a new session secret and continues the heartbeat loop. The host application never observes the lease loss. If the host had previously seen on_connection_lost (e.g. the server was down for a while before restarting), on_connection_restored fires now to symmetrize. If the lease loss happened mid-flight with no preceding transient failures, no callback fires at all -- a silent recovery.
- Re-checkout fails (pool full because someone else grabbed the seat, server still unreachable, signature mismatch, ...): the original on_license_revoked path fires with a compound reason string of the form "Lease lost (<original>); auto-re-checkout failed: <recheckout failure>" so the host's log captures both why the original heartbeat failed and why the recovery attempt didn't succeed.

Defaults to false because some operators want a server restart to forcibly drop all leases (for compliance, audit, or pool-resizing reasons); default-on would defeat that. Customers running high-availability deployments where end users should not see a license blip on server reboot should set it true.

Caveats:

- The re-checkout uses the same client_id and machine_id as the original checkout. The new lease's session_secret is fresh (server-issued). The host doesn't see the secret change because it's internal to the client.
- A 403 caused by a per-machine seat limit (max_leases_per_machine_id) on the server hitting its cap will turn into "Lease lost ... auto-re-checkout failed: machine seat limit reached" -- the host can string-match the inner failure if it cares to distinguish.
- This does not retry across multiple servers. It only re-attempts the same server_host / server_port. Multi-server failover is a roadmap item for v1.4.
- IPv4 and IPv6 (v1.4.0+). server_host may be a DNS name, an IPv4 literal, or an IPv6 literal (::1, 2001:db8::5 -- no brackets). A name is resolved over both families and each returned address is tried in the order the resolver gives them until one answers; an address whose family this host cannot route fails immediately rather than consuming the timeout. The floating server listens dual-stack on one port (IPv6 natively, IPv4 as mapped addresses), falling back to IPv4-only where the host has IPv6 disabled; its start-up log line says which. Before v1.4.0 the whole floating subsystem was IPv4-only: an IPv6 literal failed to parse, a name with only AAAA records read as NETWORK_HOST_UNRESOLVED, and the server accepted IPv4 connections only.
- Signature verification failures on heartbeat responses go straight to on_license_revoked without attempting re-checkout (the trust chain itself is broken; a fresh checkout would also fail the same way).

8.7.5 Example

```
rockyguard::FloatingClientConfig cfg;
cfg.server_host = "license.example.com";
cfg.server_port = 8443;
cfg.use_tls     = true;
cfg.tls_ca_cert_path     = "server.crt";
cfg.server_public_key_pem = server_pubkey;

cfg.max_heartbeat_failures      = 5;     // give up after 5 minutes
cfg.recheckout_on_lease_lost    = true;  // survive server restarts

std::atomic<bool> degraded{false};

};

};

};

cfg.callbacks.on_connection_lost     = [&]() {
    degraded = true;
    // post a UI message via your event loop -- do NOT touch the client here
cfg.callbacks.on_connection_restored = [&]() {
    degraded = false;
cfg.callbacks.on_license_revoked     = [&](std::string reason) {
    // log + initiate graceful shutdown of licensed features
    log_error("License revoked: " + reason);
cfg.callbacks.on_give_up             = [&]() {
    // permanent server outage: tell the user to check connectivity / call support

};

rockyguard::FloatingLicenseClient client(cfg);
```

```
auto r = client.checkout();
// ... use license ...
client.checkin();
```

9. CLI TOOLS REFERENCE

The following binary tools are provided for license management.

9.1 rg_fingerprint

Print machine hardware fingerprint.

```
rg_fingerprint [-v] [-o file.txt]
```

End users run this on their own machine and send the output hash to you so you can issue a node-locked license bound to their hardware.

9.2 license_keygen

Generate Ed25519 or RSA keypair.

```
license_keygen --private private.pem --public public.pem
```

Run once, at the start of your project. Keep private.pem secret on a secure machine; ship the public.pem string embedded in your end-user application as PUBLIC_KEY.

9.3 license_create

Create signed end-user licenses (requires vendor license). Typical usage (vendor issues a node-locked license bound to a specific customer's hardware; customer runs rg_fingerprint on their machine and sends the hash to you):

```
license_create --vendor-license vendor_license.json \
       --key private.pem --id "LIC-001" --product "App" \
       --licensee "User" \
       --fingerprint-value "311867...|ff9f0f...|877f34...|f3b059..." \
       --expires "2027-12-31T23:59:59Z"
```

Fingerprint flag variants:

Flag	Behavior
--fingerprint-value <hash>	Bind to the explicit hash supplied by the customer (the typical vendor workflow shown above).
--fingerprint	Auto-detect THIS machine's fingerprint (used for self-licensing or testing; not what you want when issuing to a remote customer).
(omit both)	Issue an unbound / floating license (use --type floating).

9.4 license_verify

Verify and inspect a license file.

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

Useful for debugging: prints the parsed license payload and verification status without requiring an end-user application.

9.5 rg_floating_server

Run a floating license server (Premium, requires vendor license).

```
rg_floating_server server_config.yaml
```

The companion executable to FloatingLicenseClient on the customer side. See Customer_Documentation §5 for full server configuration.

9.6 rg_floating_client

Test floating license checkout/checkin against a running server.

```
rg_floating_client [host] [port] [public_key.pem] [--tls] [--ca-cert server.crt]
rg_floating_client --help
```

A minimal CLI client useful for smoke-testing the floating server before integrating FloatingLicenseClient into your real application. Run with --help (or -h) to print the full argument list. On a connection failure (server unreachable) or invalid port, the client prints the usage block to stderr alongside the error message so the right invocation is one paste away.

9.7 rg_inspect / rg_report_read (v1.4+)

The Licence Inspector pair. Both are thin CLIs over the API in section 11, and both ship AS SOURCE in examples/inspector/ so you can build them with your own keys compiled in -- your end users are the people with licensing trouble and your desk is the one they contact, so the report has to come back to you. rg_inspect goes to your end users; rg_report_read stays on your support desk, since it is the half that uses the private key.

Build options: ROCKYGUARD_INSPECT_RECIPIENT_KEY (RSA public key reports are sealed to), ROCKYGUARD_INSPECT_VERIFY_KEY (your licence-verification public key, so reports carry real verdicts), ROCKYGUARD_INSPECT_PRODUCT_NAME. With the keys compiled in, an end user runs rg_inspect --license their.lic and needs nothing else.

Reports are sealed to an RSA public key -- one scheme, so there is nothing to choose and no per-report secret for an end user to invent or transmit. A build with no key compiled in stops and says so; --plaintext writes an unsealed report for local reading.

rg_report_read has four modes: --header-only (provenance without decrypting, no key needed), --summary (the verdict list, byte-identical to what the end user saw and never added to), --diagnose (that list plus ranked root causes correlated across the report's sections, and --json for a machine-readable form), and the default, which prints the whole report.

Exit status is 0 for both tools whenever the file was read or written, INCLUDING when it says the licence is invalid -- diagnosing a broken licence is the tool succeeding. Branch on license_verify, or on rg_report_read --diagnose --json; never on the exit code.

See Customer_Documentation section 7.7 for the full option list, the redaction policy, and how to read a hardware-mismatch report, and section 10.1.2 for the root-cause catalog.

9.8 rg_bind_asset + rockyguard_bind_asset() (v1.4+)

The build-time half of the load-bearing licence pattern (§12). Seals a plaintext asset under a 32-byte key into the blob LicenseAsset::open() reads and, with --emit-header, writes it as a C header so your build embeds ciphertext only.

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

OpenSSL-only -- it does not link the RockyGuard library -- so it runs in a build step with no licence context; its source ships as tools/rg_bind_asset.cpp (with examples/bound_asset.h) and the examples CMakeLists builds it as target

rg_bind_asset. From CMake, include cmake/RockyGuardBindAsset.cmake and call rockyguard_bind_asset(TARGET ... FEATURE ... ASSET ... HEADER ... [SYMBOL ...] [DATA_KEY_ENV ...]) instead of invoking it by hand; the helper regenerates the header whenever the plaintext changes and adds it to the target. See Customer_Documentation §7.8 for the option list and §9.11 for which key to seal under.

9.9 rg_bind_lint (v1.4+)

```
python tools/rg_bind_lint [PATH ...] [--quiet]
```

Python 3 linter over C/C++ source for the ways a load-bearing integration regresses to a flippable boolean: a branch on unwrap_data_key() / unwrap_feature_key(), a shared bool is_licensed()-style predicate, a branch on LicenseResult.message text, and a recovered key that never opens an asset. Exit status 1 on any ERROR finding. A lint, not a proof -- the wrong-licence negative test (Customer_Documentation §9.11 step 5) is what establishes that a binding is real. See Customer_Documentation §7.9.

See the main documentation for full CLI options and examples.

10. INTEGRATION EXAMPLES

Complete working examples are provided in the examples/ folder with a CMakeLists.txt for building. See Customer_Documentation §2.5 "Building and Running the Examples" for the cmake commands; Customer_Documentation §2.4 has the Quick Start code walkthrough.

10.1 Node-Locked End-User Application (examples/node_locked_example.cpp)

Note: the shipped examples/node_locked_example.cpp reads the public key from a file path passed on the command line ("node_locked_example <public_key.pem> <license.json>") so the same binary can be re-run against different keys during testing. The listing below shows the PRODUCTION pattern -- embed the key as a string constant per §3.3.B of Customer_Documentation. Do not ship a binary that reads the public key from a runtime file; an attacker who can replace that file can issue their own licenses (Customer_Documentation §12.1).

```
#include <rockyguard/rockyguard.h>
#include <iostream>
#include <optional>

static constexpr char PUBLIC_KEY[] = R"(-----BEGIN PUBLIC KEY-----
MCowBQYDK2VwAyEAxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-----END PUBLIC KEY-----)";

int main() {
    // Guarded construction, per §6.1. This listing is labelled the
    // PRODUCTION pattern, so it carries the guard that §6.1 and
    // Customer_Documentation §12.2 both require -- an embedded
    // literal is not exempt. Without it, a mis-pasted key is
    // std::terminate() on your end user's machine: no stdout, no
    // stderr, exit 0xC0000409 on Windows.
    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() << "\n";
        return 1;
    }
    rockyguard::LicenseVerifier& verifier = *maybe_verifier;

auto result = verifier.load("license.json");
if (!result) {
    std::cerr << "License error: " << result.message << "\n";
    return 1;
}

result = verifier.check_node_locked();
if (result.status == rockyguard::LicenseStatus::InGracePeriod) {
    std::cerr << "WARNING: " << result.grace_days_remaining
              << " grace days remaining\n";
} else if (!result) {
    std::cerr << result.message << "\n";
    return 1;
}
```

```
// OPTIONAL version gate (v1.2.1+). Skip the block below if
// you do not want runtime version enforcement; version_range
// is otherwise informational. See §6.7. Pass YOUR application's
// version; empty version_range matches anything.
static constexpr char APP_VERSION[] = "3.1.0";
result = verifier.check_version(APP_VERSION);
if (!result) {
    std::cerr << "Version not allowed: " << result.message << "\n";
    return 1;
}

if (verifier.check_feature("pro")) {
    std::cout << "Pro features enabled\n";
}

    return 0;
}
```

10.2 Floating Client Application (examples/rg_floating_client.cpp)

See examples/rg_floating_client.cpp for a complete working example.

```
#include <rockyguard/rockyguard.h>
#include <iostream>

// PUBLIC_KEY is the server's signing public key, embedded as a
// string constant at file scope exactly as shown in §10.1. The
// floating server signs lease responses; the client uses this key
// to verify them. See Customer_Documentation §3.3.B for the
// embed-as-constant rationale.

int main() {
    rockyguard::FloatingClientConfig config;
    config.server_host = "license-server.internal";
    config.server_port = 8080;
    config.server_public_key_pem = PUBLIC_KEY;  // Verify server

// v1.3+: transport security is mandatory-by-default. Exactly one
// of these two blocks must be present or the constructor throws.
config.use_tls = true;                      // Encrypt the wire
config.tls_ca_cert_path = "server.crt";     // Verify server cert

// Local testing against a plaintext server instead? Drop the two
// lines above and opt in explicitly. Never ship this:
// config.allow_insecure_http = true;

rockyguard::FloatingLicenseClient client(config);
auto result = client.checkout();
if (!result) {
    std::cerr << result.message << "\n";
    return 1;
}

// ... application logic ...
```

```
    client.checkin();
    return 0;
}
```

10.3 Floating License Server

Use the provided server binary with a YAML config file:

```
rg_floating_server server_config.yaml
```

See tools/floating_server_config.yaml for a sample configuration.

The server uses a thread pool for connection handling. Key config fields:

```
thread_pool_size: 4              Worker threads (default: 4)
client_timeout: 5                Drop slow clients after N seconds (default: 5)
private_key: private.pem         Sign responses (prevents server spoofing)
tls_cert: server.crt             TLS certificate (v1.3: see below)
tls_key: server.key              TLS private key (v1.3: see below)
allow_insecure_http: false       Accept cleartext deliberately (v1.3+;
                                 default false). AS OF v1.3 THE SERVER
                                 REFUSES TO START unless either
                                 tls_cert + tls_key are both set or
                                 this is true. It is the one key whose
                                 absence stops a cert-less server from
                                 starting at all, so a config copied
                                 from an older example will not run.
                                 Note the YAML keys are tls_cert and
                                 tls_key; the FloatingServerConfig
                                 members are tls_cert_path and
                                 tls_key_path.
require_tls: false               Refuse to start without tls_cert/tls_key
                                 (v1.3+; default false). Belt-and-braces
                                 on top of the above: it closes the
                                 allow_insecure_http escape hatch.
rate_limit_rps: 50               Per-IP token-bucket refill rate, req/sec
                                 (v1.3+; default 50; 0 disables).
rate_limit_burst: 250            Per-IP burst capacity (v1.3+; default 250).
                                 Over-limit -> HTTP 429 (or drop under TLS).
                                 See Customer_Documentation §5.2.
max_leases_per_machine_id: 0     Per-machine seat cap (v1.2.1+; 0 = uncapped).
                                 Defends against ghost-checkout exhaustion.
                                 See Customer_Documentation §5.2.
log_max_bytes: 104857600         Rotate active log file at N bytes (v1.2.1+;
                                 default 100 MiB; <= 0 disables rotation).
log_keep_count: 5                Archives kept (server.log.1..N; v1.2.1+;
                                 default 5; capped at 100).
```

10.4 Load-Bearing Licence (examples/loadbearing_asset_example.cpp) (v1.4+)

Both ends of the load-bearing pattern in one file: --encrypt <key_hex> <in> <out.enc> seals an asset at build time (through examples/bound_asset.h, the same sealer rg_bind_asset uses), and --pubkey <pem> --license <lic> --asset <out.enc> [--feature <name>] opens it at runtime with LicenseAsset -- the product key without --feature, that feature's key with it -- and feeds the bytes into the feature. Run it with a genuine licence (the asset opens, exit 0) and with a foreign or absent one (the feature is inert, exit 3): that second run is the negative test in Customer_Documentation

§9.11. The runtime shape to copy is in §12.3.

End of API Reference

11. DIAGNOSTICS API (v1.4+)

Header: <rockyguard/diagnostics.h> Namespace: rockyguard::diagnostics

The Licence Inspector as a library API. This is what the shipped rg_inspect and rg_report_read CLIs are built on; call it directly if you would rather put diagnostics inside your own application, your own installer, or your own support UI than ship a separate executable.

The header itself is the reference -- it documents every field and the reasoning behind each -- so this section is a map rather than a repetition.

11.1 Collecting

```
int         schema_version();
std::string collect(const Options&, ProgressFn = nullptr, void* = nullptr);
```

Options carries what to collect. The field that matters most is public_key_pem: it is PEM CONTENT, not a path, precisely so you can compile your verification key in and let the report carry the real verdicts your application produces without your end user handling a key file. license_path, current_version and feature drive the optional verification stages; probe_network, probe_pool_hosts, network_timeout_sec, server_host/server_port/server_tls control the network probe; include_raw_hardware, measure_clock_drift, run_clock_check and note do what their names say.

collect() returns the report as a JSON document and NEVER THROWS. A diagnostic that declines to produce output when something is broken is useless exactly when it is needed, so anything unexpected lands in the report as a field.

Two contract properties:

* probe_network = false means NO sockets, including the trusted-time fetch. The library enforces that rather than trusting the caller, because the two are separate code paths internally.

* The anti-rollback inventory is taken BEFORE anything verifies, and the report records that it was. Verifying writes anchor state -- creating the witnesses when there are none, re-baselining them after a confirmed forward clock excursion -- and those two states are exactly what a field failure looks like. Do not reimplement collection by calling pieces in your own order.

ProgressFn is a plain function pointer, not std::function: it crosses the DLL boundary, and a function pointer has no allocator or vtable to disagree about.

11.2 Sealing

```
SealOutcome seal(const std::string& report_json, const SealRequest&);
std::string recipient_key_problem(const std::string& pem);
```

One scheme: a fresh random AES-256-GCM content key per report, wrapped to your RSA public key with RSA-OAEP/SHA-256, container metadata bound in as AAD. Sealing to a key needs no shared secret, which is why there is nothing else on offer -- an earlier passphrase-derived variant was removed as strictly weaker and as an obligation on whoever ran the tool to invent and transmit a per-report secret.

The recipient key must be RSA. An Ed25519 key -- which is what a RockyGuard licence-signing key is -- is a signature key with no encryption operation. Call recipient_key_problem() BEFORE collecting: collection takes seconds and touches the network, and the likeliest mistake is worth catching before all that work rather than after it. Empty return means the key is usable.

11.3 Opening and presenting

```
OpenOutcome open_container(const std::string&, const OpenRequest&);
std::string describe_container(const std::string& container);
bool        is_plaintext_report(const std::string& blob);
std::string summarize(const std::string& report_json);
```

describe_container() gives human-readable provenance WITHOUT decrypting and needs no key -- which is how a desk holding several keys works out which to fetch, and how triage identifies a suspicious attachment safely. It is written to survive a hostile file.

summarize() returns the one-screen verdict list as text. Deliberately the same rendering everywhere: what your end user reads off their terminal is byte-identical to what your engineer sees after opening the report, so the two cannot disagree mid-call.

11.6 Health check: the structured view (self-test API)

```
enum class Severity { Info, Warning, Error };
struct HealthItem  { label; value; detail; code; severity; };
struct HealthReport{ overall; headline; items; report_json; };

HealthReport health_check(const Options&, ProgressFn = nullptr, void* = nullptr);
HealthReport parse_health(const std::string& report_json);
```

This is the "self-test / health check" entry point: one call that returns a full status report, suitable for calling at startup or from a Help -> Diagnose Licence menu item.

summarize() (11.3) returns terminal text -- fixed-width, padded columns -- which is right for a console and wrong for a GUI, where a proportional font turns aligned columns into ragged ones. health_check() returns the same findings as DATA: one HealthItem per finding, each with a severity you can turn into an icon and a code you can branch on. No string parsing, and no JSON library needed on your side.

Branch on code, never on value or detail. code is the v1.4 LicenseErrorCode name (section 4.5) and is stable; the other two are prose and are not.

overall is the worst severity present, for colouring a dialog. headline is one sentence for a title bar. report_json is the full report the struct was derived from, so a dialog can offer "send diagnostic" via seal() without collecting a second time.

Severity comes from the library's own error catalog rather than being re-derived, so the grading here always agrees with what the rest of the API reports for the same condition.

parse_health() structures a report you already have -- one you just collected, or one a customer e-mailed that rg_report_read opened. Same rendering either way, which is the point: your support view and their dialog cannot disagree. It never throws; an unreadable report comes back as a single Error item saying so.

One finding is worth knowing about because nothing else reports it: "Hardware drift". When the fingerprint threshold is still met but a component has changed, verification PASSES and no error code is produced anywhere -- yet that machine is closer to a lockout than it was. health_check() raises it as a Warning.

11.7 Health check: off the calling thread

```
class AsyncHealthCheck {
    explicit AsyncHealthCheck(const Options&);   // starts immediately
    bool         ready() const;                  // non-blocking
    std::string  progress() const;               // safe from any thread
```

```
    HealthReport take();                        // blocks until ready
};
```

collect() is synchronous and can take seconds: the network stage probes several hosts with multi-second timeouts, and DNS resolution cannot be bounded portably. Do not call it on a UI thread.

You could run it on a std::thread yourself. This exists because the obvious way to do that goes wrong: a progress CALLBACK fires on the collecting thread, so every GUI toolkit then needs it marshalled back to the UI thread, and getting that wrong is a crash rather than a glitch. So this class calls nothing back at all -- you poll it, from whichever thread you like:

```
rgd::AsyncHealthCheck job(options);        // returns at once
// ...on a UI timer tick:
label.set_text(job.progress());
if (job.ready()) show_dialog(job.take());
```

ready() reads one atomic and is safe to call as often as you like. progress() returns the most recent section label under a mutex, safe from any thread. take() blocks until the work finishes, so guard it with ready() if you must not block; it is valid once, and a second call returns an empty report rather than a moved-from one.

Destroying a running job is safe and joins the thread -- the case where a user closes the dialog mid-collection. It is deliberately not detached: a detached thread would go on writing into freed memory.

No std::future in the signature, deliberately. Returning one across a DLL boundary requires the caller's C runtime to match the library's -- the same reason LicenseVerifier has no load_async() (see 6.2). An opaque handle carries no such constraint.

Worked example: examples/health_check_example.cpp, which shows both the fast startup check and the async menu pattern.

11.8 Root-cause diagnosis

```
enum class Confidence { Possible, Likely, Certain };
enum class CauseBand  { Blocking, Root, Contributing, Advisory, Informational };

struct RootCause      { cause_id, headline, explanation, remediation,
                        evidence[], codes[], explains[], severity,
                        confidence, band, rank, suppressed, support_only };
struct DiagnosisReport{ overall, headline, causes[], degraded,
                        degraded_reason, report_schema_version,
                        engine_expects_schema, rule_catalog_revision };

DiagnosisReport diagnose(const std::string& report_json);
std::string     render_diagnosis(const DiagnosisReport&);
std::string     explain(const std::string& report_json);
std::string     diagnosis_json(const DiagnosisReport&);

const char const diagnose_cause_ids(std::size_t* count);
const char* confidence_name(Confidence);
const char* cause_band_name(CauseBand);
const char* severity_name(Severity);
```

Everything above this subsection reports SYMPTOMS: parse_health() emits one item per signal, each read from one section of the report. This correlates ACROSS sections, which is where the expensive failures live -- a licence reported expired because the machine's clock is three days fast is two facts in two subtrees, and no single-section reader can join

them.

That shape is not incidental. A LicenseResult carries exactly one code, so it structurally cannot report two simultaneous conditions; a secondary-code channel would have to come out of the error catalog, and one that any return site could set by hand would re-open the status/code drift the catalog exists to prevent. But two simultaneous conditions always land in two different report subtrees, and reading both is something a diagnosis engine can do without touching the catalog at all. RootCause::codes is a vector for exactly this reason.

diagnose() takes the report JSON -- matching parse_health() and summarize(), and because rules need fields HealthReport does not carry. It never throws, and it never returns an empty causes list: nothing-wrong is a cause that says so, and unreadable-input is a cause that says that. It is a pure function of its argument and NEVER READS THE SYSTEM CLOCK, so a report diagnosed a year after collection gives the byte-identical answer it gave on the day.

Branch on cause_id. It is stable in the same sense an error code name is -- never reused, retired ids never reissued -- with one documented difference: a rule's trigger may be TIGHTENED in a patch release so that it fires on fewer reports, though never loosened onto a different condition. rule_catalog_revision tells you whether two diagnoses ran the same rules; the library version cannot, because a patch may tighten a trigger. The prose fields are not stable and must not be branched on.

Causes another cause explains are KEPT, with suppressed set and their id listed in the explainer's explains[]. They are never deleted, because an engine that silently discards what it rejected cannot be argued with by an engineer who thinks it is wrong. Render them indented or behind a disclosure control; do not hide them.

The rules do not restate the error catalog. One ships only when it correlates two or more sections, or splits one code into distinct causes with distinct remedies; where correlation changes nothing, no rule fires and the catalog's own suggested_action stands. Customer_Documentation section 10.1.2 is the catalog of what does ship, and diagnose_cause_ids() is the same vocabulary at runtime.

support_only marks causes whose reasoning you may not want an end user reading -- chiefly the anti-rollback family, where the remedy doubles as a tamper recipe. Nothing filters on it for you: rg_report_read is a support-desk tool and its operator is not the party being withheld from. The shipped rg_inspect does not diagnose at all; surfacing diagnosis to your end users is your decision.

RootCause and DiagnosisReport carry no ROCKYGUARD_API, unlike HealthItem and HealthReport beside them. That is deliberate: they have no out-of-line members and no vtable, so dllexport would export only compiler-generated special members no caller links against, while being exactly what makes MSVC warn about their std::vector members. The existing exported structs keep the attribute, because removing it would delete symbols from the DLL's export table and break anyone who linked them.

diagnose_cause_ids() returns pointer + count while diagnose() returns a value, and the discriminator is ownership rather than taste: static vocabulary is pointer + count, computed results are values. See 11.4.

11.4 Why everything is std::string

No nlohmann::json, no internal structs and no iostream appear in this header. Exposing any of them would put a third-party type in the exported ABI: you would need a matching JSON library version to compile against us, our bumping a vendored dependency would become a breaking change for you, and every internal field rename would leak outward. A JSON document passed as text costs one parse and buys complete freedom to restructure everything behind it -- and it is what lets the shipped CLIs have no JSON dependency at all, which is what makes them buildable in your tree against nothing but these public headers.

11.5 Thread safety

collect() reads machine state and performs network I/O; call it from one thread at a time. seal(), open_container(), summarize() and describe_container() are pure functions of their arguments and safe to call concurrently.

12. LICENSEASSET CLASS (v1.4+)

Header: <rockyguard/license_asset.h> Namespace: rockyguard

The load-bearing licence pattern with the branch designed out of the API. unwrap_data_key() and unwrap_feature_key() (§6.10, §6.11) return std::optional, which is honest about what happened but hands you a shape whose idiomatic use is a test: if (!verifier.unwrap_data_key()) refuse(); -- one line that reintroduces, in your own binary, exactly the flippable branch the scheme exists to delete. LicenseAsset removes the shape instead of warning about it. There is no ok(), no operator bool, no optional, and no way to ask whether the key arrived.

```
class LicenseAsset {
public:
    explicit LicenseAsset(const LicenseVerifier& verifier);                       // product data key
    LicenseAsset(const LicenseVerifier& verifier, const std::string& feature_name); // one feature's key
    ~LicenseAsset();
    LicenseAsset(LicenseAsset&&) noexcept;             // movable ...
    LicenseAsset& operator=(LicenseAsset&&) noexcept;
    LicenseAsset(const LicenseAsset&) = delete;        // ... not copyable
    LicenseAsset& operator=(const LicenseAsset&) = delete;

    std::vector<uint8_t> open(const std::vector<uint8_t>& sealed) const;
    std::vector<uint8_t> open(const uint8_t* sealed, size_t sealed_len) const;
    std::string open_string(const std::vector<uint8_t>& sealed) const;
    std::string open_string(const uint8_t* sealed, size_t sealed_len) const;
};
```

12.1 Constructors

```
LicenseAsset(const LicenseVerifier& verifier)
LicenseAsset(const LicenseVerifier& verifier, const std::string& feature_name)
```

Capture the key the licence yields -- the product data key (license_create --wrap-data-key) with the first form, the key sealed for feature_name (license_create --wrap-feature-key <name>=<hex>) with the second -- or nothing. Neither constructor can fail and neither reports what it captured; the absence of a key is the ordinary condition of running without a valid licence, not an error state to query.

CONSTRUCT AFTER load(). The key is captured once, here, from whatever the verifier holds at that moment. A LicenseAsset built at namespace scope, or before load(), captures nothing and keeps capturing nothing for the life of the process. It gives no signal when that happens, by design: the signal would be a branch.

The second form does not consult check_feature(). A licence the vendor did not entitle to feature_name carries no sealed key for it, so the constructor captures nothing regardless of what a patched has_feature() says -- the key comes from the envelope, not from the signed payload's feature list.

12.2 open() / open_string()

```
std::vector<uint8_t> open(const std::vector<uint8_t>& sealed) const
std::vector<uint8_t> open(const uint8_t* sealed, size_t sealed_len) const
std::string          open_string(const std::vector<uint8_t>& sealed) const
std::string          open_string(const uint8_t* sealed, size_t sealed_len) const
```

Decrypt an asset sealed under the captured key with AES-256-GCM in the library's blob layout, nonce(12) || ciphertext ||

tag(16) -- what tools/rg_bind_asset and the example's --encrypt write. The pointer + length forms take the byte array and _len constant that rg_bind_asset --emit-header generates, so an embedded asset is opened without copying it into a vector first; a null pointer is treated as an empty blob.

Returns the plaintext, or an EMPTY vector / string when there is no key, the blob is malformed, or authentication fails. Never throws. There is deliberately no way to distinguish those cases: feed the result into your parser or loader and let it fail on garbage. Do not write if (table.empty()) refuse; -- that is the branch this class exists to remove. tools/rg_bind_lint (§9.9) reports that guard, and the equivalent test on unwrap_data_key() / unwrap_feature_key(), as an ERROR; run it over your sources before every release.

12.3 The shape to copy

```
#include <rockyguard/license_asset.h>
#include "export_tables.enc.h"     // generated by rg_bind_asset: kExportTables, kExportTables_len

rockyguard::LicenseAsset asset(verifier, "export_pdf");            // after verifier.load()
load_export_tables(asset.open(kExportTables, kExportTables_len));  // real work; garbage/nothing without the
    key
```

Bind each paid feature to its own asset under its own feature key; there is then no shared verdict anywhere to patch. The key is OPENSSL_cleanse()d when the LicenseAsset is destroyed.

12.4 Honest ceiling

Signature-bound, not machine-bound and not time-bound: a genuinely-signed but expired or wrong-machine licence still yields its keys, so check_expiry() and check_node_locked() remain necessary for expiry and hardware enforcement -- this adds a crypto floor under the branch, it does not replace those checks. It defeats the free branch-flip / keygen-less crack and forces an attacker to possess a genuine licence and reverse your (ideally obfuscated) open path; it does not, for an offline scheme, force signature forgery, since one leaked genuine licence carries the signature that yields its keys. Use per-customer keys if you need leak traceability.

Why neither binding can live in the key. Machine: node-locking matches 2 of 4 fingerprint components by default so that a replaced disk or NIC does not strand a paying customer; a key derived from the live fingerprint would change on that same single-component drift and stop decrypting a licence that still validates. There is no fuzzy key derivation -- a key either matches or it does not -- so machine binding stays with check_node_locked(), which can tolerate. Time: the asset is a fixed ciphertext that must open on every day the licence is valid, so its key must be the same on every one of those days; a key that depended on "not yet expired" would need a trusted statement of today's date, and offline the only clock is the end user's and the anchor files are local state. Such a key would be a branch on the local clock wearing a KDF, and no harder to patch than the branch it replaced. Expiry therefore stays a verdict from check_expiry(), whose strength is detection (anchors, the occasional online confirmation), and it is treated as a verdict: kept, guarded by the multi-site advice in the AI Integration Guide, never mistaken for a key.

12.5 Thread safety

The constructors take the verifier's READER lock once, through unwrap_data_key() / unwrap_feature_key(); the object then holds its own copy of the key and open() touches no shared state, so a LicenseAsset may be used from any thread after construction, and several may be constructed concurrently.

See Customer_Documentation §9.11 for the pattern end to end (keys, sealing, minting, the mandatory negative test), §7.8 / §7.9 for the tools, and AI_INTEGRATION_GUIDE §6.5 for the agent-facing recipe.