

RockyGuard Library - Documentation

Version 1.2.0

Copyright (c) 2025-2026 [Legal Entity TBD]. All rights reserved.

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

1. INTRODUCTION

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

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

Key capabilities:

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

Dependencies:

- OpenSSL 3.x (linked automatically)

Supported platforms (v1.2.0):

- Windows 10 / 11 (x64), Windows Server 2019+
- Linux x64 on glibc 2.34 or newer. The shipped Linux binary statically links libstdc++, libgcc, and OpenSSL, so glibc is the only runtime ABI dependency. It runs on:

- Ubuntu 22.04 LTS (glibc 2.35)
- Ubuntu 24.04 LTS (glibc 2.39)
- Debian 12 "Bookworm" (glibc 2.36)
- RHEL 9 / CentOS Stream 9 / Rocky Linux 9 (glibc 2.34)
- Amazon Linux 2023 (glibc 2.34)
- Any other x86_64 Linux distribution with glibc 2.34+.

It does NOT run on Debian 11, Ubuntu 20.04, RHEL 8, or Amazon Linux 2 (all have glibc 2.31 or older). A glibc-2.28 build extending coverage to those older distributions is planned for a subsequent release.

- macOS 11 (Big Sur) and newer, x64 and Apple Silicon.

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

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

2. GETTING STARTED

2.1 Package Contents

rockyguard.h	Single-include convenience header
types.h	Enums and result types
license.h	License data model
license_verifier.h	License verification
hardware_fingerprint.h	Machine hardware fingerprinting
floating_client.h	Floating license client (Premium)
export.h	DLL export macros
rockyguard.lib	Static library
rockyguard.dll	Shared library (DLL)
rockyguard.lib	Import library for DLL
rockyguard.sig	Integrity signature (ship with DLL)
libcrypto-3-x64.dll	OpenSSL runtime
license_keygen.exe	Generate your Ed25519/RSA keypair
license_create.exe	Create end-user licenses
license_verify.exe	Verify and inspect license files
floating_server_example.exe	Floating license server (Premium)
floating_server_config.yaml	Sample server config file
floating_client_example.exe	Floating license client (Premium)
rg_fingerprint.exe	Print machine hardware fingerprint
include/openssl/	Bundled OpenSSL headers
lib/libcrypto.lib	Bundled OpenSSL import library
CMakeLists.txt	Build script for examples
node_locked_example.cpp	Node-locked license verification example
floating_client_example.cpp	Floating license client example

This documentation and the API reference

The layout above shows the Windows zip (rockyguard-v1.2.0-windows-x64-customer.zip). The Linux zip (rockyguard-v1.2.0-linux-x64-customer.zip) is identical in structure with the following per-platform differences:

librockyguard.so	in place of rockyguard.dll
librockyguard.sig	in place of rockyguard.sig (same format, just matches the .so name)
(no libcrypto / libssl)	OpenSSL is statically linked into librockyguard.so on Linux, so there is no separate runtime dependency
librockyguard.a	in place of rockyguard.lib
	(license_keygen, license_create, license_verify, rg_fingerprint, floating_server_example, floating_client_example)
	(both are needed when linking the

```
static librockyguard.a; see Sec 3.1.)
```

No .lib import library is needed on Linux; the shared library is linked directly via `-lrockyguard` once `LD_LIBRARY_PATH` or `RUNPATH` points at `lib/shared/`. macOS customers: deferred to v1.3.

2.2 Your Vendor License

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

IMPORTANT:

- The vendor license is needed **ONLY** in your license generation tools and floating license server.
- Your end-user application does **NOT** need the vendor license file. It only verifies licenses using `LicenseVerifier` and `FloatingLicenseClient`, which work without initialization.

Your license tier:

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

2.3 Quick Start

Step 1: Generate your keypair (run once):

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

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

Step 2: Create an end-user license. Requires the `vendor_license.json` your library vendor provided (see section 2.2):

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

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

```
#include <rockyguard/rockyguard.h>  
  
static constexpr char PUBLIC_KEY[] = R"(-----BEGIN PUBLIC KEY-----  
MCowBQYDK2VwAyEA...your public key here...  
-----END PUBLIC KEY-----)";  
  
int main() {  
    rockyguard::LicenseVerifier verifier(PUBLIC_KEY);  
  
    auto result = verifier.load("license.json");  
    if (!result) {  
        std::cerr << result.message << std::endl;  
        return 1;  
    }  
}
```

```
result = verifier.check_node_locked();
if (!result) {
    std::cerr << result.message << std::endl;
    return 1;
}

// Application runs with valid license
return 0;
}
```

2.4 Building and Running the Examples

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

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

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

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

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

Available examples:

Demonstrates license loading, signature verification, hardware fingerprint checking, grace period handling, and feature gating.
Usage: node_locked_example <public_key.pem> <license.json>

Demonstrates floating license checkout, heartbeat, and checkin.
Usage: floating_client_example [server_host] [port]

3. LIBRARY INTEGRATION

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

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

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

3.1 Linking (Static - Recommended)

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

IMPORTANT on Windows: the shipped `rockyguard.lib` (and the bundled `libssl.lib` / `libcrypto.lib`) are Release-CRT builds (/MD, release iterator debug level). Your own executable must be built in Release mode (or with a Release-compatible CRT) or you will see LNK2038 or LNK1319 link errors about mismatched `_ITERATOR_DEBUG_LEVEL` and `RuntimeLibrary` values. If you need a Debug build of your own app, contact the library vendor for the matching Debug-CRT library build, or build your app with /MT (static CRT) so the consumer choice no longer depends on the shipped CRT variant. This restriction does not apply on Linux.

In your CMakeLists.txt (Windows):

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

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

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

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

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

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

3.2 Linking (Shared / DLL)

Windows:

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

The DLL itself (rockyguard.dll) carries libcrypto and libssl statically, so at build time you do NOT need to link them; at runtime the DLL is self-contained except for the two OpenSSL runtime DLLs listed below.

Ship with your application:

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

Linux:

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

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

Ship with your application:

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

3.3 Two Separate Integration Points

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

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

Typical invocation (Windows; Linux is the same with bare names):

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

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

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

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

int main() {
    // NO init() call needed here

    rockyguard::LicenseVerifier verifier(PUBLIC_KEY);
    auto result = verifier.load("license.json");
    if (!result) {
        std::cerr << result.message << std::endl;
        return 1;
    }

    result = verifier.check_node_locked();
    if (result.status == rockyguard::LicenseStatus::InGracePeriod) {
        std::cerr << "License expiring soon: "
                    << result.grace_days_remaining << " days left\n";
    } else if (!result) {
        std::cerr << result.message << std::endl;
        return 1;
    }

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

    return 0;
}
```


4. END-USER LICENSE MANAGEMENT

4.1 Generating Your Keypair

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

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

For RSA instead of Ed25519:

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

4.2 Getting an End User's Machine Fingerprint

Ask your end user to run:

```
rg_fingerprint -v
```

Or to save to a file they can send you:

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

The output looks like:

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

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

4.3 Creating End-User Licenses

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

Node-locked license:

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

```

```
--feature "export_pdf" \  
--feature "advanced_reports"
```

Floating license (Premium tier):

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

Permanent license:

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

4.4 Verifying a License File

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

4.5 Fuzzy Hardware Matching

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

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

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

4.6 Feature Gating

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

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

Create different license tiers by varying the feature list:

```
--feature "api_access" --feature "sso"
```

4.7 Grace Periods

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

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

5. FLOATING LICENSING (PREMIUM TIER)

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

5.1 Creating a Floating License

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

5.2 Running the License Server

The floating license server requires a vendor license (Premium tier). Deploy it on a machine accessible to all end users.

OPTION A: Using the provided server executable with a YAML config file:

Create a config file (e.g. server_config.yaml):

```
# Vendor license file (required, Premium tier)  
vendor_license: vendor_license.json  
  
# License pool size  
max_users: 10  
  
# Server settings  
port: 8080  
heartbeat_interval: 60  
lease_timeout: 120  
  
# Thread pool and client timeout  
thread_pool_size: 4  
client_timeout: 5  
  
# Response signing (recommended): server signs every response  
private_key: private.pem  
  
# TLS encryption (optional): uncomment to enable HTTPS  
# tls_cert: server.crt  
# tls_key: server.key  
  
# Server mode: release or debug  
mode: release  
  
# Log to file (uncomment to enable)  
# log_file: server.log
```

Run the server:

```
floating_server_example server_config.yaml
```

A sample config file (floating_server_config.yaml) is included in the tools/ directory.

SERVER MODES:**release (default):**

Logs basic events: server start/stop, license checkout/checkin, lease expirations, errors. Suitable for production use.

debug:

Logs everything in release mode plus: full request/response details, client connection info (IP, port), heartbeat events, periodic lease table dumps with heartbeat age, worker thread activity, client timeouts. Useful for troubleshooting.

LOG OUTPUT:

By default, all log output goes to stderr. To log to a file, set the `log_file` field in the config. Logs are appended (not overwritten).

Log format:

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

THREAD POOL AND CONNECTION HANDLING:

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

thread_pool_size (default: 4):

Number of worker threads. Each handles one connection at a time. Increase for high-concurrency environments.

client_timeout (default: 5 seconds):

If a client connects but doesn't send data within this time, the connection is dropped. Protects against denial-of-service attacks where a client connects and holds the connection open.

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

RESPONSE SIGNING AND TLS:

The server can sign every response with your private key. The client verifies the signature using the corresponding public key. This prevents server spoofing (e.g., someone redirecting traffic to a fake server that always returns "checked_out").

`private_key`: path to your Ed25519/RSA private key (same key used for license signing). The client needs the corresponding public key.

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

```
openssl req -x509 -newkey ec -pkeyopt ec_paramgen_curve:prime256v1 \
```

```
-days 3650 -nodes -keyout server.key -out server.crt \
-subj "/CN=license-server"
```

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

5.3 Client Integration (No Vendor License Needed)

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

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

rockyguard::FloatingLicenseClient client(config);

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

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

5.4 Server HTTP Endpoints

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

Leases are automatically released if heartbeat is not received within `lease_timeout_sec` (default: 120 seconds).

6. HARDWARE FINGERPRINTING

6.1 Collected Components

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

6.2 Using the Fingerprint API

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

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

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

7. CLI TOOLS REFERENCE

7.1 rg_fingerprint

Usage: rg_fingerprint [options]

Print the machine's hardware fingerprint.

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

Examples:

```
# Print fingerprint to screen
rg_fingerprint

# Show detailed components + fingerprint
rg_fingerprint -v

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

7.2 license_keygen

Usage: license_keygen [options]

--algo <ed25519|rsa> Algorithm (default: ed25519)
--rsa-bits <bits> RSA key size (default: 2048)
--private <file> Private key output (default: private.pem)
--public <file> Public key output (default: public.pem)

7.3 license_create

Usage: license_create [options]

--key <file> Private key PEM file (required)
--algo <ed25519|rsa> Signature algorithm (default: ed25519)
--output <file> Output file (default: license.json)
--id <id> License ID (required, must be unique)
--licensee <name> Licensee name
--product <name> Product name (required)
--version <range> Version range (e.g. "3.*")
--type <node_locked|floating> License type (default: node_locked)
--expires <date> Expiry (ISO 8601) or "permanent"
--grace-days <n> Grace period in days (default: 0)
--max-users <n> Max concurrent users (floating only)
--feature <name> Add a feature (repeatable)
--fingerprint Auto-detect this machine's fingerprint
--fingerprint-value <v> Use explicit fingerprint value
--threshold <n> Fingerprint match threshold (default: 2)
--show-fingerprint Print machine fingerprint and exit

7.4 license_verify

Usage: license_verify [options]

--key <file>	Public key PEM file (required)
--license <file>	License file (required)
--algo <ed25519 rsa>	Signature algorithm (default: ed25519)
--check-hardware	Verify hardware fingerprint
--check-feature <name>	Check if feature is licensed

8. LICENSE FILE FORMAT

8.1 Structure

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

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

8.2 Payload Fields

Field	Type	Description
-----	-----	-----
license_id	string	Unique license identifier (REQUIRED)
licensee	string	Licensed entity name
product	string	Product name (REQUIRED)
version_range	string	Version pattern (e.g. "3.*")
type	string	"node_locked" or "floating"
hardware_fingerprint	string	Component hashes (pipe-separated)
fingerprint_match_threshold	int	Min matching components (default: 2)
issued_at	string	ISO 8601 issue date
expires_at	string	ISO 8601 expiry or "permanent"
grace_period_days	int	Days after expiry before block
max_concurrent_users	int	Pool size (floating only)
features	string[]	Feature flag names
metadata	object	Arbitrary key-value pairs

9. ANTI-TAMPERING FEATURES

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

9.1 Digital Signatures

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

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

9.2 Clock Manipulation Detection

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

- UTC timestamps are stored in multiple hidden locations on the machine
- Anchor files are unique per license (derived from license_id, product, and licensee), so different products don't interfere with each other
- On each license check, all locations are read and cross-verified
- If the current time is more than 1 hour behind the maximum stored timestamp, clock manipulation is detected
- The stored timestamps are integrity-protected; editing them is detected
- Deleting some storage locations doesn't help - surviving ones still catch the rollback
- Online time verification: the library verifies the system clock via HTTPS (port 443) against a pool of 12 trusted internet servers, randomly selected. Uses TLS certificate timestamps (CA-signed, can't be forged) and HTTP Date headers. Hosts file redirects fail because the TLS handshake rejects invalid certificates. Mandatory when anchors are missing, 10% random during normal operation. Silently skipped if offline.

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

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

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

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

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

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

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

9.4 Floating Server Authentication

The floating license protocol includes multiple security layers:

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

IMPORTANT: Without TLS, session authentication tokens are sent in plaintext and can be sniffed by network eavesdroppers. Response signing prevents server spoofing, but a passive listener on the network can still capture the session secret and forge client requests. For full security, enable TLS. The server logs a warning at startup when signing is enabled but TLS is not.

9.5 Security Hardening

The library is compiled with OS-level security protections:

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

These are enabled automatically and require no configuration.

9.6 Threat Model

The table below summarizes the concrete attacks RockyGuard is designed to defend against, the mechanism each defense relies on, and the residual risk that remains after the defense is applied. The "Residual risk" column is what an attacker can still attempt; the column "Mitigation" describes what your application or operations team should add on top of the library to close that gap.

Attack	RockyGuard defense	Residual risk and mitigation
License file tampering (editing dates, machine fingerprint, features inside the JSON)	Ed25519 / RSA-SHA256 signature over the payload, verified at load()	None for the file itself: any byte change invalidates the signature and load() returns SignatureInvalid. Residual: an attacker who steals your private key can sign anything; mitigation = keep private.pem on a secure machine; rotate keys per the versioning policy if compromise is suspected.

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

10. TROUBLESHOOTING

10.1 Error Reference Table

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

LicenseStatus / message	Where it surfaces	Detailed entry
SignatureInvalid	LicenseVerifier::load() / load_from_string()	10.2.1
HardwareMismatch	LicenseVerifier::check_node_locked()	10.2.2
Expired	LicenseVerifier::load() / check_expiry()	10.2.3
FeatureNotLicensed	LicenseVerifier::check_feature()	10.2.4
MalformedFile	LicenseVerifier::load() (invalid JSON or missing fields)	10.2.1 (related)
LibraryNotInitialized	license_create / floating_server_example CLI	10.3.1
GenerationLimitReached	license_create CLI	10.3.2
MachineNotAuthorized	license_create / floating_server_example CLI	10.3.3
TierNotAuthorized	floating_server_example CLI	10.3.4
ClockManipulated	LicenseVerifier::check_expiry() / load()	10.4.1
IntegrityCheckFailed	LicenseVerifier::load() / check_expiry() (DLL builds)	10.4.2
NoLicensesAvailable	FloatingLicenseClient::checkout()	10.5.1
ServerUnreachable	FloatingLicenseClient::checkout() / checkin()	10.5.2
InGracePeriod	LicenseVerifier::load() / check_expiry()	Not an error: license is expired but still usable; check <code>grace_days_remaining</code> and warn the user.
Valid	Any verification call	Not an error: success.

The full enum definition and operator `bool()` semantics are in `Customer_API_Reference.txt` Sec 4.2 (enum class `LicenseStatus`) and Sec 4.4 (struct `LicenseResult`).

Each entry below is structured as:

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

10.2 License Errors

10.2.1 "License signature verification failed"

Cause: One of three things has gone wrong: (a) the license file or its signature was modified after signing, (b) the public key embedded in your application does not correspond to the private key that signed the license, or (c) the `SignatureAlgorithm` passed to `LicenseVerifier` differs from the one used at signing time.

Resolution: Replace the license with the genuine signed copy from your `license_create` run. Verify that `PUBLIC_KEY` in your application is the exact PEM string from `public.pem` produced by `license_keygen` alongside the private key used at

signing. Pass the same SignatureAlgorithm at verify time as at sign time (default is Ed25519 on both sides).

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

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

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

10.2.3 "License has expired"

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

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

10.2.4 "Feature not licensed: <name>"

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

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

10.3 Vendor License Errors

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

10.3.1 "Library not initialized. Call rockyguard::init()..."

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

Resolution: Re-run the tool with `--vendor-license vendor_license.json` (pointing at the `vendor_license.json` file your library vendor sent you).

10.3.2 "End-user license generation limit reached"

Cause: Your vendor license carries a maximum number of end-user licenses you may issue, and the persistent generation counter (preserved across CLI tool runs and reboots) has hit that cap. The counter is HMAC-protected: deleting state files does not reset it.

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

10.3.3 "This machine is not authorized for license generation"

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

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

10.3.4 "Floating licensing requires a Premium tier library license"

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

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

10.4 Anti-Tampering Errors

10.4.1 "System clock manipulation detected"

Cause: The library compared the current system clock to multiple stored time anchors and detected a backward jump beyond tolerance. Most often this means the end user genuinely set the clock back; less often, anchors were corrupted, or the user adjusted the time zone in a way that wrapped past an anchor.

Resolution: Have the end user set the system clock to the correct current time. If the rollback was unintentional (dead CMOS battery on a desktop, manual TZ fix that crossed a midnight boundary), the legitimate time correction will succeed and the next run will pass. If anchors are corrupted (rare), the end user should contact your support so you can re-issue a license; the library cannot self-recover from this state by design.

10.4.2 "Library integrity check failed"

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

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

10.5 Floating License Errors

10.5.1 "No floating licenses available"

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

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

10.5.2 "Cannot reach license server"

Cause: The FloatingLicenseClient could not establish a TCP connection (or, with use_tls, a TLS handshake) to the configured FloatingClientConfig.server_host : server_port.

Resolution: Verify the server is running and listening on the expected port. Check network connectivity (ping, traceroute) and firewall rules between client and server. Confirm server_host and server_port in FloatingClientConfig point at the right host. With use_tls = true, also confirm tls_ca_cert_path is correct and the server's certificate is valid.

End of Document