

RockyGuard Library - API Reference

Version 1.2.0

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

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

1. QUICK START

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

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

int main() {
    rockyguard::LicenseVerifier verifier(PUBLIC_KEY);
    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 section 9.

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/hardware_fingerprint.h>  // Machine fingerprinting
#include <rockyguard/floating_client.h>       // Floating client (Premium)
```

Linking with CMake (static, Windows; full link line):

```
add_executable(your_app main.cpp)
target_include_directories(your_app PRIVATE
    ${ROCKYGUARD}/include
    ${ROCKYGUARD}/deps/include)
target_link_libraries(your_app PRIVATE
    ${ROCKYGUARD}/lib/static/rockyguard.lib
    ${ROCKYGUARD}/deps/lib/libssl.lib
    ${ROCKYGUARD}/deps/lib/libcrypto.lib
    ws2_32 crypt32 iphlapi ole32 oleaut32 wbemuuid)
```

Where `${ROCKYGUARD}` is the path to the extracted package directory. See `Customer_Documentation.txt` section 3.1 for the Linux equivalent and the CRT-variant caveat (the shipped library is built with the Release CRT /MD; consumer projects must build in Release mode or request a Debug-CRT library).

IMPORTANT on Windows: the shipped `rockyguard.lib` and the bundled `libssl.lib` / `libcrypto.lib` are Release-CRT (/MD) builds. Building your application in Debug mode against them produces LNK2038 / LNK1319 errors on `_ITERATOR_DEBUG_LEVEL` and `RuntimeLibrary`. Build your consumer project in Release mode, or contact the library vendor for a Debug-CRT build (planned for v1.2.1).

Coming in v1.3: a shipped `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. v1.2.0 ships the explicit link line above; v1.3 will keep it working alongside the imported-target form.

For shared (DLL) linking, see `Customer_Documentation.txt` section 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/macOS
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 on the v1.3 roadmap. The warnings remain on stderr until then.

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

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
ServerUnreachable	Floating: cannot reach server
LibraryNotInitialized	init() not called (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

4.3 enum class SignatureAlgorithm

Ed25519	Ed25519 (default, recommended)
RSA_SHA256	RSA with SHA-256

4.4 struct LicenseResult

Member	Type	Description
status	LicenseStatus	Result status
message	std::string	Human-readable description
grace_days_remaining	int	Grace days left (0 if N/A)

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

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

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 Section 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.

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.

6. LICENSEVERIFIER CLASS

Defined in: <rockyguard/license_verifier.h>

Verifies end-user licenses.

6.1 Constructor

```
explicit LicenseVerifier(const std::string& public_key_pem,  
                        SignatureAlgorithm algo = SignatureAlgorithm::Ed25519)
```

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

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).

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()` (Section 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 socket timeout. Worst-case wall-clock latency is bounded by $3 * (\text{system DNS timeout} + 3 \text{ seconds})$. 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 Section 6.6 for the full state matrix).

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.

Coming in v1.3: a non-blocking `load_async()` returning `std::future<LicenseResult>` with the same semantics, so GUI hosts do not have to roll their own async wrapper.

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

6.3 load_from_string()

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

Same as `load()` but from a JSON string.

6.4 check_node_locked()

```
LicenseResult check_node_locked()
```

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

Returns: Valid, InGracePeriod, HardwareMismatch.

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.

Example:

```
if (verifier.check_feature("export_pdf")) {  
    menu.enable_export();  
}
```

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.
Anchors MISSING, online	Online time fetched and validated; drift > 1h => ClockManipulated; else fresh anchors written and check returns Valid.

State	Behavior
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.

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 license()

```
const License& license() const
```

Access the parsed license after successful load():

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

6.8 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 (returned MalformedFile or SignatureInvalid). 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(), or check_expiry().

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).

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, IOKit on macOS). Returns a struct with one `std::string` per slot; any slot the OS could not provide is left empty.

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.

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.

8. FLOATINGLICENSECLIENT CLASS (PREMIUM)

Defined in: <rockyguard/floating_client.h>

Checks out floating licenses from a server.

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
};
```

The last three fields combine into layered defenses. They protect different things and are independent of each other.

`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.

`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. Empty path = TLS encrypts but does not verify the peer; an active MITM with any cert can intercept and decrypt.

Recommended configurations:

Configuration	Settings	When to use
Production	<code>use_tls = true</code> , <code>tls_ca_cert_path</code> set, <code>server_public_key_pem</code> set	All three layers active; no single defense failure exposes the deployment.
Internal / pinned	<code>use_tls = false</code> , <code>server_public_key_pem</code> 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.
Development only	<code>use_tls = false</code> , <code>server_public_key_pem</code> 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(). Generates a unique client_id (UUID) for this session.

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.

Returns: Valid, NoLicensesAvailable, ServerUnreachable, SignatureInvalid, MalformedFile.

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.

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.

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 floating_server_example

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

```
floating_server_example server_config.yaml
```

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

9.6 floating_client_example

Test floating license checkout/checkin against a running server.

```
floating_client_example [host] [port]
```

A minimal CLI client useful for smoke-testing the floating server before integrating FloatingLicenseClient into your real application.

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 section 2.4 of the main documentation for build instructions.

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

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

static constexpr char PUBLIC_KEY[] = R"(-----BEGIN PUBLIC KEY-----
MCoWBQYDK2VwAyEAxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-----END PUBLIC KEY-----)";

int main() {
    rockyguard::LicenseVerifier verifier(PUBLIC_KEY);

    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;
    }

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

    return 0;
}
```

10.2 Floating Client Application (examples/floating_client_example.cpp)

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

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

int main() {
    // No init() needed

    rockyguard::FloatingClientConfig config;
    config.server_host = "license-server.internal";
    config.server_port = 8080;
    config.server_public_key_pem = PUBLIC_KEY; // Verify server
```

```
// config.use_tls = true;           // If server uses TLS
// config.tls_ca_cert_path = "server.crt"; // Verify server cert

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:

```
floating_server_example 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	Optional: enable TLS encryption
# tls_key: server.key	Optional: enable TLS encryption

End of API Reference