Security Gate Pattern for Second Life Viewer

Reducing CVE Surface Without Full Architectural Changes

Author: H. Overman
Contact: opsec.ee@pm.me
Date: February 2025

This document accompanies specific vulnerability findings submitted through responsible disclosure. It proposes a lightweight, non-invasive pattern for reducing the viewer's CVE surface based on the same methodology that identified those findings.

No new language. No new library. Just a pattern you can implement in your existing C++ with your existing build system.

1. Gates: What They Are

`GATE_CHECK` is a macro that compiles to nothing in release builds and full validation in debug/QA builds. Pay for what you use. You already understand `llassert` — a gate is `llassert` with a CVE number and a severity level instead of just a string.

Each gate maps to a specific vulnerability class. When a gate fires, you know exactly which CVE pattern was triggered, in which function, with what values. No guessing. No grepping logs.

2. Concrete Example

What you have now (audioengine.cpp, approximate):

```
void AudioEngine::removeChannel(Channel* ch) {
    mChannels.erase(ch->id);
    delete ch;
    // ... 200 lines later, something still holds ch
}
```

What a gate looks like (zero external dependency, pure C++):

```
void AudioEngine::removeChannel(Channel* ch) {
    if (GATE_CHECK(UAF_001, ch, ch->ref_count, 0)) {
        GATE_LOG(UAF_001, ch,
                 "removeChannel while refs held");
        return;  // don't delete, refs still active
    }
    mChannels.erase(ch->id);
    delete ch;
}
```

`UAF_001` = Use-After-Free, gate number 001. Maps to CVE-2016-6309 / CWE-416. When this fires in a debug build, you get the gate ID, the pointer, the ref count, and the call site. In a release build, the macro compiles out entirely.

3. Gate Categories

These are the vulnerability classes that gate patterns catch. Each maps to real CVEs found in production software:

Gate	Class	CVE	Description
UAF-001	Use-After-Free	CVE-2016-6309	Access to freed memory
DF-001	Double-Free	CVE-2017-9047	Freeing already-freed memory
NULL-001	Null Pointer	CVE-2009-1897	Null pointer dereference
BOF-001	Buffer Overflow	CVE-2014-0160	Heap buffer overflow (Heartbleed class)
TYPE-001	Type Confusion	CVE-2015-8651	Wrong type cast/access
RACE-001	Race Condition	CVE-2016-2853	Time-of-check to time-of-use
REF-001	Refcount Overflow	CVE-2016-0728	Reference count integer overflow

The findings submitted alongside this document map directly to entries in this table.

4. Measured Overhead

Gate checks are branch-on-condition instructions. Measured cost across security levels:

Level	Gates Active	Overhead	Use Case
NONE	0	0%	Release builds
BASIC	Refcount, Type	~5%	Development
STANDARD	+ Bounds, Overflow	~10%	QA / staging
PARANOID	All + canaries	~20%	Security audit

Release builds compile gate macros to nothing. Zero cost. The security levels exist for your debug and QA pipelines where the overhead is acceptable and the diagnostic value is high.

5. Three-Thread Sidecars

This part is about where gates could go next, not what you need to implement today. It's something to consider, especially for a team of ~210 contributors managing a codebase with this many concurrent subsystems.

The sidecar model maps naturally to what the viewer already does:

FRONT (Predict) = your interest list / LOD system. It already predicts what the viewer will need. It just doesn't feed back into security decisions. What if your interest list also predicted which capability checks to prefetch?

LEAD (Execute) = your main loop. This doesn't change. It's authoritative. It runs the simulation, renders the frame, processes events.

REAR (Verify) = what you don’t have. This is the real value. You have no verification thread. When the media plugin loads a URL, nobody checks after the fact whether that URL did something unexpected. When audioengine processes a buffer, nobody verifies the buffer wasn’t modified between check and use (TOCTOU / RACE-001). A rear thread that audits what LEAD just did catches the entire class of time-of-check-to-time-of-use bugs that plague plugin architectures.

The sidecar conversation comes later, when you’ve lived with gates long enough to want verification automation. I’m not saying I know it all — I damn well do not — but I feel like this is something worth considering.

6. Summary

Here’s the problem, here’s a pattern, here’s the cost:

- **Problem:** The viewer codebase has use-after-free, null dereference, and TOCTOU vulnerabilities in audioengine and media plugin subsystems. Specific findings submitted under responsible disclosure.
- **Pattern:** Security gates — CVE-mapped validation macros that compile out in release. No new dependencies. Pure C++ macros over your existing codebase.
- **Cost:** Zero in release builds. 5–20% in debug/QA builds depending on security level. The bugs will come back — they always do — but with gates in place, they get nailed down instantly.

I leave you with the patterns. Patterns are how I found all of the bugs. There must be something legitimate about that.

Only trying to be forward-thinking. Available for discussion:

H. Overman
opsec.ee@pm.me