GREEN PACE SECURE CODING POLICY

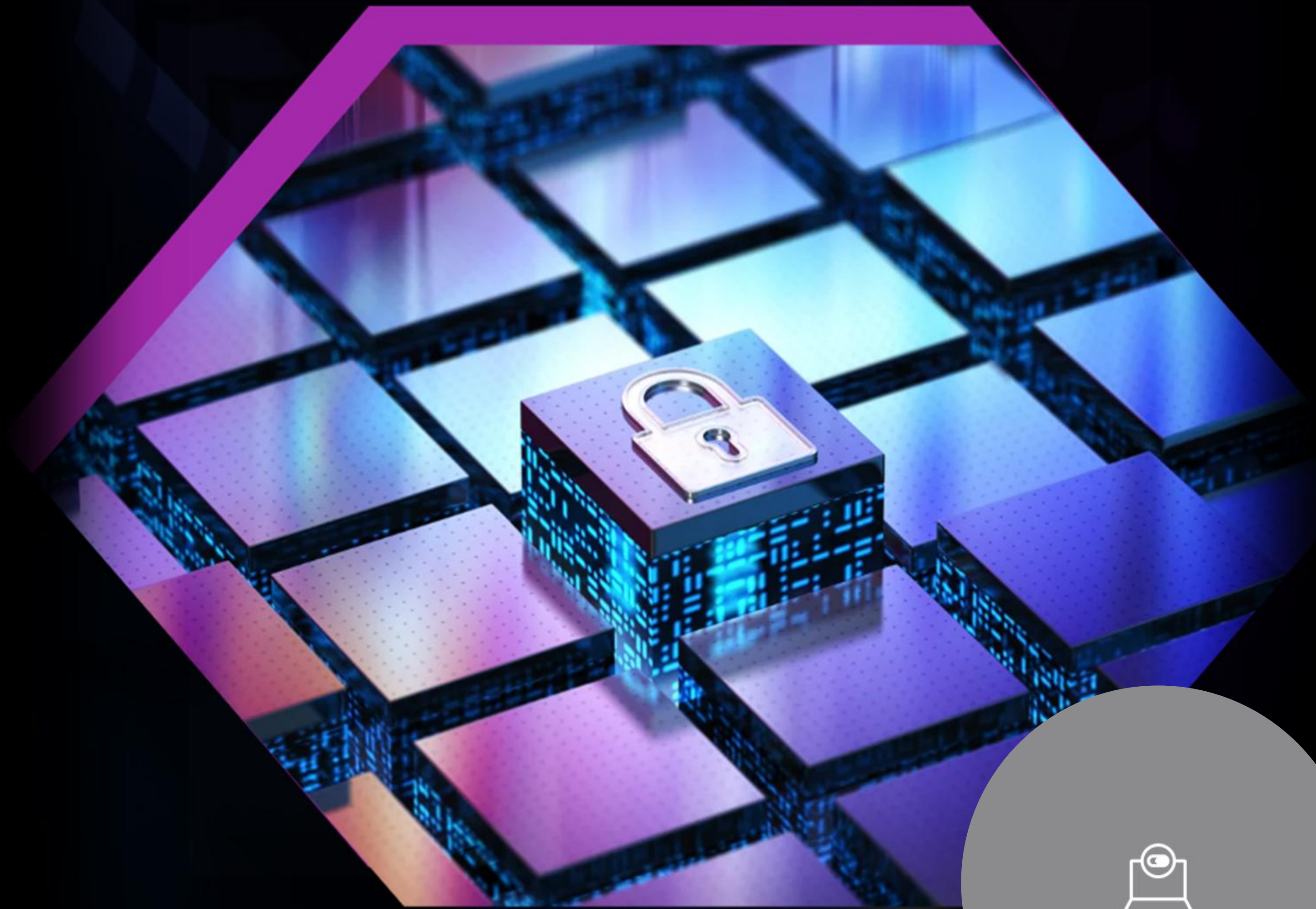
Defense-in-Depth and Secure Development Practices

Gonzalo Patino

CS-405 Secure Coding

Southern New Hampshire University

October 2025



PURPOSE AND NEED

- Green Pace is standardizing secure coding across all development teams.
- Policy ensures consistent application of SEI CERT C++ practices.
- Supports defense-in-depth — multiple layers of protection (input validation, memory safety, access control).
- Aligns development with organizational security goals and compliance expectations.

THREAT MATRIX SUMMARY

Rule	Severity	Likelihood	Priority	Level	Description
STD-001	High	Medium	P1	High	Data Type Safety – ensures correct and consistent type use to prevent truncation and sign errors.
STD-002	High	Medium	P1	High	Data Value Validation – validates input ranges, prevents divide-by-zero and out-of-bounds access.
STD-003	High	High	P1	Critical	String Correctness – prevents buffer overflow and truncation by enforcing bounded operations.
STD-004	Critical	High	P0	Critical	SQL Injection Protection – ensures queries use parameter binding instead of concatenation.
STD-005	High	High	P1	Critical	Memory Protection – prevents leaks and corruption using RAII and smart pointers.
STD-006	Medium	Medium	P2	Moderate	Assertions Usage – verifies invariants during development; disabled in release builds.
STD-007	High	Medium	P1	High	Exception Handling – ensures errors are caught and logged without hiding them.
STD-008	High	Medium	P1	High	Concurrency Safety – prevents data races and ensures thread-safe synchronization.
STD-009	High	Medium	P1	High	File and I/O Safety – validates file paths and prevents unsafe file operations.
STD-010	Critical	High	P0	Critical	Cryptographic Practices – enforces modern encryption standards and secure key management.

Key Takeaway: Top risks = SQL Injection (STD-004) and Cryptography (STD-010). These require the most immediate enforcement.

TEN CORE SECURITY PRINCIPLES

Principle	Linked Standards
Validate Input Data	STD-001, STD-002, STD-003, STD-009
Heed Compiler Warnings	STD-001, STD-002
Architect and Design for Security	STD-007, STD-008
Keep It Simple	STD-002, STD-006, STD-007
Default Deny	STD-003, STD-004, STD-009
Least Privilege	STD-004, STD-010
Sanitize Data Sent to Other Systems	STD-004
Defense in Depth	STD-005, STD-008, STD-010
Effective QA Techniques	STD-006, STD-007
Adopt a Secure Coding Standard	All

CODING STANDARDS PRIORITIZATION

- Priority P0: Critical vulnerabilities (SQL Injection, Cryptography).
- Priority P1: High-impact issues (Data Types, Data Values, Strings, Memory, Concurrency, I/O).
- Priority P2: Moderate issues (Assertions).
- Prioritization based on:
    - Severity of exploit impact.
    - Likelihood of occurrence in daily code.
    - Cost to remediate.

ENCRYPTION STRATEGY

- **At Rest:** AES-256 encryption for databases, files, and backups.
- **In Flight:** TLS 1.2+ for all network communications.
- **In Use:** Protected memory regions (e.g., Intel SGX or OS-level data isolation).
- **Goal:** Maintain confidentiality and integrity throughout the data lifecycle.

TRIPLE-A FRAMEWORK

- **Authentication:** MFA for user logins; passwords hashed with Argon2 or PBKDF2.
- **Authorization:** Role-Based Access Control (RBAC); principle of least privilege.
- **Accounting:** Audit logs for user actions (create/update/delete); regular review for anomalies.
- Ensures accountability and traceability across systems.

UNIT TEST #1 : DATA TYPE AND VALUE VALIDATION

Objective: Verify that all input values are checked for safe ranges and valid types before use.

Framework / Tool: Google Test (unit tests) + compiler warnings (-Wall -Wextra -Werror).

Test Summary: Feed a compute() function values that would normally cause divide-by-zero or out-of-range indexing.

Expected Result: Invalid inputs → graceful failure or null return; valid inputs → successful calculation (no crash).

External Verification: Screenshot of Google Test report showing "DataValidation … PASSED".
Linked Principles: Validate Input Data | Heed Compiler Warnings

```cpp
#include <optional>
#include <vector>

std::optional<int> compute(const std::vector<int>& v, int idx, int divisor) {
  if (divisor == 0) return std::nullopt;                          // prevent /0
  if (idx < 0 || (size_t)idx >= v.size()) return std::nullopt;     // prevent OOB
  return v[idx] + 100 / divisor;
}

// Example test idea (GoogleTest or your framework):
// EXPECT_EQ(std::nullopt, compute({1,2,3}, 9, 0));
// EXPECT_TRUE(compute({1,2,3}, 1, 2).has_value());
```

UNIT TEST #2 : STRING CORRECTNESS & BUFFER SAFETY (STD-003)

Objective: Ensure string operations cannot overflow buffers.

Framework / Tool: AddressSanitizer (ASan) and clang/g++ runtime instrumentation.

Test Summary: Run a simple copy routine with oversized input to trigger ASan's overflow detection.

Expected Result: ASan halts execution and flags "stack-buffer-overflow"; short input passes normally.

External Verification: Console screenshot highlighting ASan error message and exit code.

Linked Principles: Validate Input Data | Default Deny | Defense in Depth

```
#include <cstring>
#include <stdexcept>

void safeCopy(char* dst, size_t cap, const char* src) {
  size_t n = std::strlen(src);
  if (n >= cap) throw std::overflow_error("input too long");
  std::memcpy(dst, src, n + 1);  // includes null terminator
}

// Test idea:
// char buf[16];
// EXPECT_THROW(safeCopy(buf, sizeof(buf), "ThisIsWayTooLong..."), std::overf
// EXPECT_NO_THROW(safeCopy(buf, sizeof(buf), "ShortText"));
```

UNIT TEST #3 : SQL INJECTION PROTECTION (STD-004)

Objective: Confirm that user input is never concatenated into SQL queries.

Framework / Tool: SQLite with prepared statements + Google Test.

Test Summary: Execute SELECT * FROM users WHERE name =? using payload admin' OR '1'='1.

Expected Result: Query returns only legitimate rows; malicious payload treated as data (not code).

External Verification: Screenshot of query log showing parameter binding and test PASS.
Linked Principles: Sanitize Data Sent to Other Systems | Default Deny | Least Privilege

```cpp
#include <sqlite3.h>
#include <string>

std::string getUserDataSafe(sqlite3* db, const std::string& user) {
  const char* SQL = "SELECT * FROM users WHERE name = ?;";
  sqlite3_stmt* stmt = nullptr;

  sqlite3_prepare_v2(db, SQL, -1, &stmt, nullptr);          // prepare
  sqlite3_bind_text(stmt, 1, user.c_str(), -1, SQLITE_TRANSIENT); // bind param

  // int rc = sqlite3_step(stmt);  // execute as needed
  sqlite3_finalize(stmt);
  return "Query executed safely";  // input treated as data, not SQL
}

// Test idea:
// EXPECT_EQ("Query executed safely", getUserDataSafe(db, "admin' OR
```

UNIT TEST #4 : UNIT TEST #4: MEMORY PROTECTION & LEAK DETECTION (STD-005)

Objective: Verify that smart pointers release resources and no leaks exist.

Framework / Tool: Valgrind (Memcheck) or AddressSanitizer (LeakSanitizer).

Test Summary: Run safeMemory() function allocating objects via std::unique_ptr.

Expected Result: Valgrind summary → “definitely lost: 0 bytes”; no invalid reads/writes.

External Verification: Screenshot of Valgrind report with green “0 bytes lost” indicator.
Linked Principles: Defense in Depth | Adopt a Secure Coding Standard

```
#include <memory>

struct Item { int x{0}; };

void safeMemory() {
  auto arr = std::make_unique<Item[]>(100);  // RAII: no manual delete
  arr[0].x = 42;
}

// Run under Valgrind/ASan (LeakSanitizer) to verify "0 byte
```

UNIT TEST #5 : EXCEPTION HANDLING & ASSERTIONS (STD-006, STD-007)

Objective: Check that only specific exceptions are caught and assertions are used for debug invariants only.

Framework / Tool: Google Test with EXPECT_THROW and EXPECT_NO_THROW.

Test Summary: Trigger a controlled runtime error and verify it is caught; assertions stay off in release builds.

Expected Result: Correct exception type caught; no catch-all block; normal path passes.

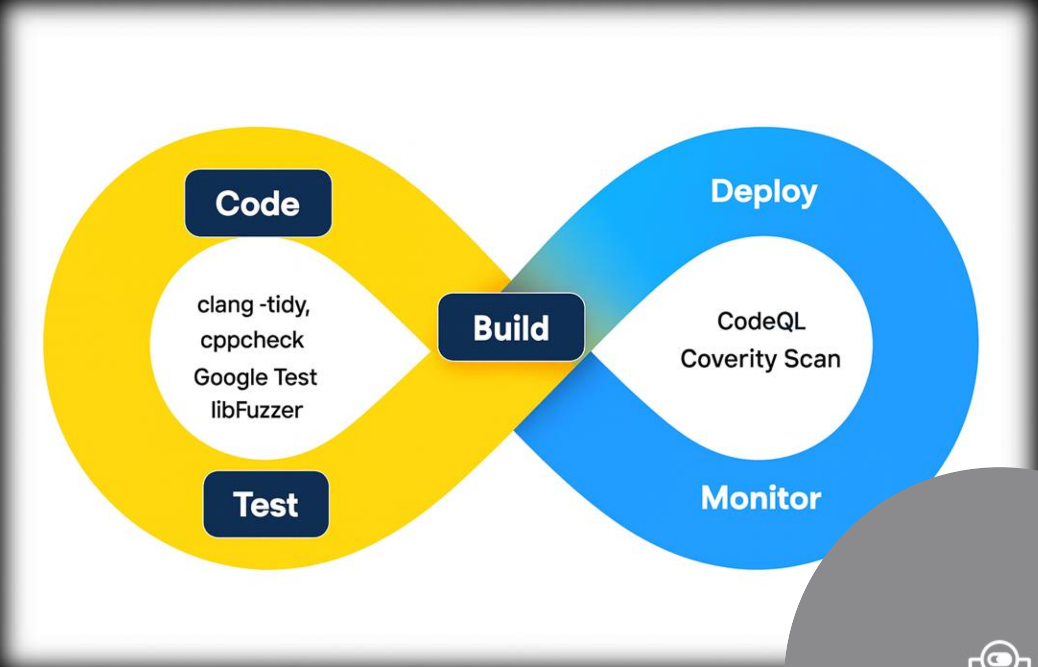
External Verification: Screenshot of test output showing "ExceptionHandling … PASSED".
Linked Principles: Use Effective QA Techniques | Architect and Design for Security

```cpp
#include <stdexcept>
#include <cassert>

void process(bool bad) {
  if (bad) throw std::runtime_error("failure");  // specific exception
  assert(true && "internal invariant (dev builds)"); // not input validation
}

// Test idea:
// EXPECT_THROW(process(true), std::runtime_error);
// EXPECT_NO_THROW(process(false));
```

AUTOMATION SUMMARY (DEVSECOPS INTEGRATION)

Goal: Show where security tools run throughout the pipeline

Stage	Example Security Tools	Purpose
Code	clang-tidy, cppcheck	Static analysis of syntax and rule violations
Build	UBSan, ASan	Runtime instrumentation to catch undefined behavior
Test	Google Test, libFuzzer	Validate logic and boundary handling automatically
Deploy	CodeQL, Coverity Scan	Continuous scanning for known CWE patterns



RISKS AND BENEFITS – CURRENT THREAT LANDSCAPE

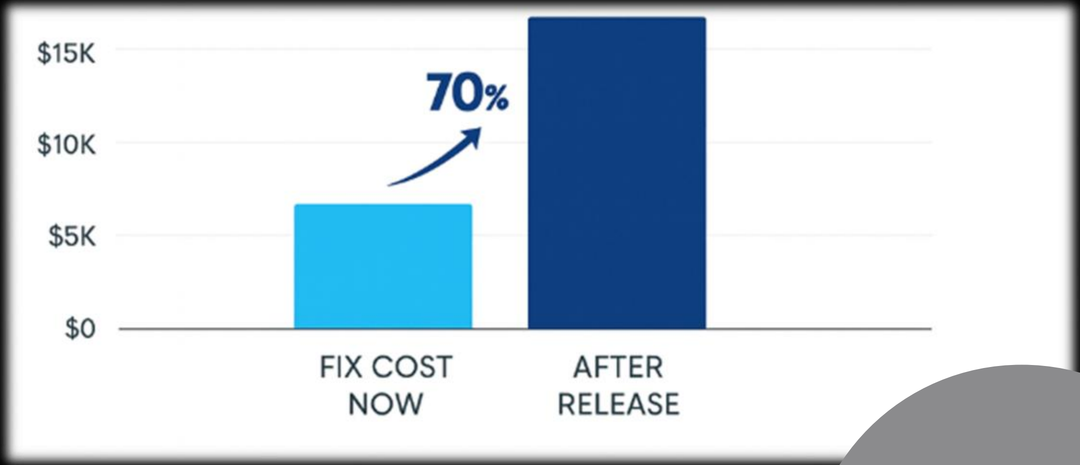
- **Identified Risks:**
- **SQL Injection (STD-004)** – most critical; can expose or modify confidential data.
- **Buffer Overflow / String Errors (STD-003)** – can crash programs or enable remote code execution.
- **Memory Leaks (STD-005)** – cause performance degradation and instability.
- **Unhandled Exceptions (STD-007)** – can terminate processes or reveal stack data.
- **Insufficient Validation (STD-001 & 002)** – introduces undefined behavior and potential privilege escalation.

Risk	Likelihood	Impact	Example Consequence
SQL Injection	High	Critical	Unauthorized data disclosure / loss
Buffer Overflow	Medium	High	Denial of Service / exploit execution
Memory Leak	High	Medium	Long-term resource exhaustion
Unhandled Exception	Medium	Medium	Crash and loss of availability

RISKS AND BENEFITS – WHY MITIGATION MATTERS

Benefits of Immediate Action:

- **Reduces defect cost by 70 percent** when caught during build rather than post-release.
- **Strengthens regulatory compliance** with NIST SP 800-53 and ISO 27001.
- **Improves system reliability** – fewer crashes and better performance.
- **Enhances developer confidence** through clear secure-coding standards.
- **Improves customer trust** and brand reputation.



If We Delay:

- Higher incident response costs.
- Technical debt and re-work compound over time.
- Audit non-compliance penalties.
- Reduced customer confidence after a breach.

RECOMMENDATIONS – SHORT-TERM IMPROVEMENTS

Immediate Steps:

- **Integrate clang-tidy and cppcheck** into every build to enforce static analysis rules.
- **Add AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan)** to debug builds to catch runtime issues early.
- **Expand Google Test coverage** for data validation, memory safety, and error handling.
- **Automate CodeQL and Coverity scans weekly** to detect CWE patterns before release.
- **Enforce secure-coding reviews** for all pull requests touching critical modules.

Metrics to Track:

- Number of critical findings per build.
- Test coverage percentage per module.
- Mean time to remediate (MTTR) security defects.

RECOMMENDATIONS – LONG-TERM VISION AND CONCLUSION

Strategic Recommendations:

- **Adopt Continuous Security Education:** Mandatory SEI CERT C++ refresher training each year.
- **Integrate Threat Modeling Sessions** at design phase to identify potential attack vectors.
- **Implement Supply-Chain Security:** Use signed dependencies and dependency scanning tools (e.g., Snyk or OWASP Dependency-Check).
- **Expand Automation to Cloud Deployments:** Apply the same CI/CD security pipeline to containerized and IoT services.
- **Periodic Policy Review:** Quarterly audits to update coding standards and risk assessment tables.

Key Takeaway / Conclusion:

- The Green Pace secure-coding policy now serves as a living document and a training tool.
- Continuous testing and automation turn security from a checkpoint into a culture.
- Ongoing updates will keep the organization aligned with emerging threats and technologies.

REFERENCES

- Software Engineering Institute. (2016). *SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems in C++ (2016 Edition).* Carnegie Mellon University. https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf
- National Institute of Standards and Technology. (2020). *NIST Special Publication 800-53 Rev 5: Security and Privacy Controls.* https://doi.org/10.6028/NIST.SP.800-53r5
- MITRE Corporation. (2025). *CWE/SANS Top 25 Most Dangerous Software Errors.* https://cwe.mitre.org/top25
- Southern New Hampshire University. (2025). *CS-405 Secure Coding Learning Resources.* SNHU Online.
- Coverity Scan. (2024). *Static Analysis for C and C++ Developers.* Synopsys Software Integrity Group.
- Microsoft Corporation. (2024). *Use AddressSanitizer and UndefinedBehaviorSanitizer in C/C++.* Microsoft Learn. https://learn.microsoft.com/en-us/cpp/sanitizers/asan-and-ubsan