Green Pace

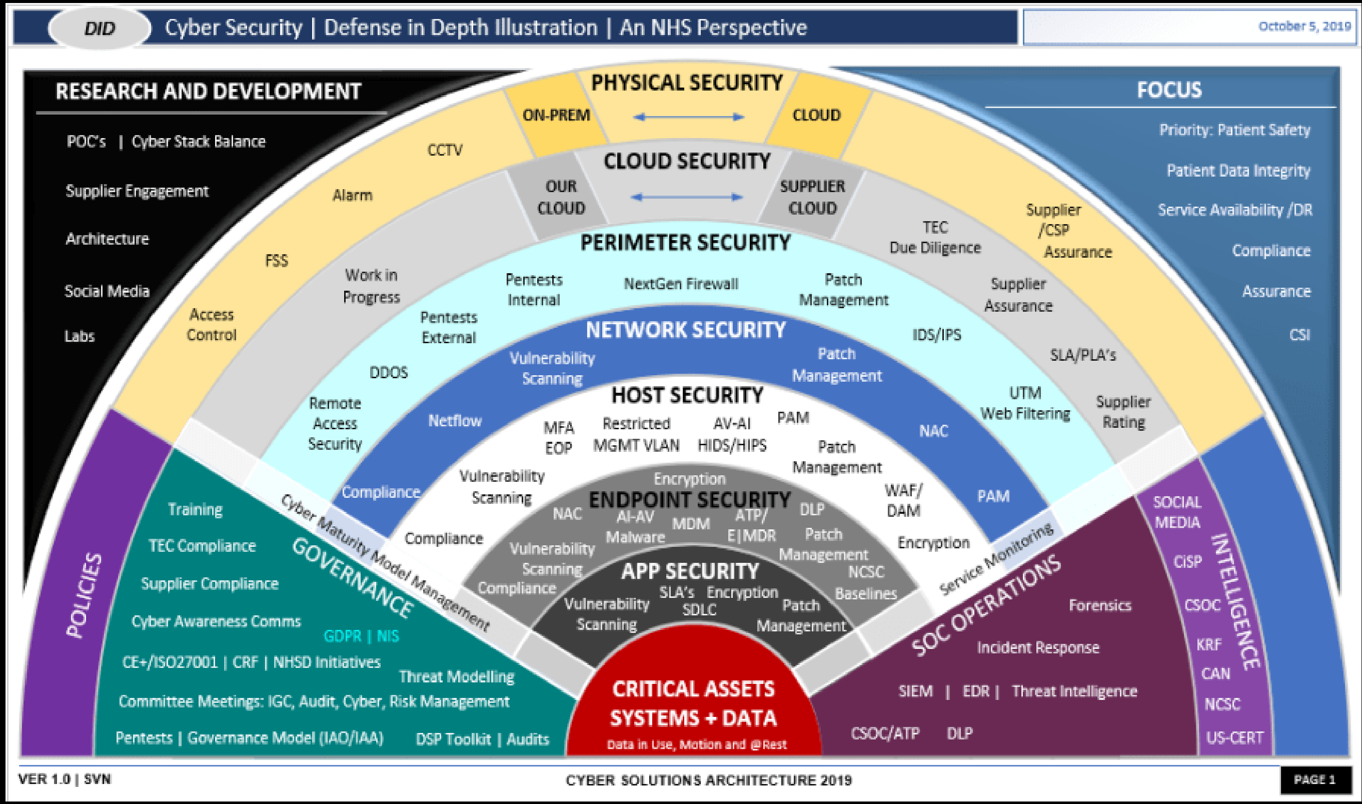
Security Policy Presentation

Developer: Joshua Shoemaker

Green Pace

OVERVIEW: DEFENSE IN DEPTH

The Green Pace Secure Development Policy standardizes secure coding and architectural practices across a growing development team.



Green Pace

THREATS MATRIX

Likely	Priority
SQL Injection Unsafe strings Integer overflow Range/Buffer Overflow validation failures Raw pointers creating leaks/use-After-free	Missing RAII leads to resource leak Unhandled exceptions causes crashes/exposure
Low priority	**Unlikely**
Oversized types Poor exception recovery Over-reliance on assertions	

Green Pace

10 PRINCIPLES

1	Validate Input Data	STD-002, STD-003, STD-004, STD-008
2	Heed Compiler Warnings	STD- 002, STD-006
3	Architect and Design for Security Policies	STD-001, STD-008
4	Keep It Simple	STD-001, STD-005, STD-007, STD-009
5	Default Deny	STD-005, STD-009
6	Adhere to the Principle of Least Privilege	STD-001
7	Sanitize Data Sent to Other Systems	STD-002, STD-003, STD-004
8	Practice Defense in Depth	STD-003, STD-004, STD-005, STD-007, STD-008, STD-010
9	Use Effective Quality Assurance Techniques	STD-005, STD-006, STD-009, STD-010
10	Adopt a Secure Coding Standard	All standards align with this core principle

Green Pace

CODING STANDARDS

1	STD-004	SQL Injection	5
2	STD-002	Range Validation	4
3	STD-003	Safe Strings	4
4	STD-005	Smart Pointers	4
5	STD-008	Arithmetic Overflow	4
6	STD-009	RAII	4
7	STD-010	Handle Exceptions	4
8	STD-001	Smallest Type	3
9	STD-007	Recoverable Exceptions	3
10	STD-006	Assertions	2

Green Pace

ENCRYPTION POLICIES

- **Encryption at rest**: AES-256 or better on databases, files, backups. Mandatory for sensitive and or persistent data to prevent unauthorized access if storage is compromised.
- **Encryption in flight**: TLS 1.3 with strong ciphers on all network communications (APIs, web, internal sensitive). Mandatory to prevent MITM, interception, tampering.
- **Encryption in use**: Homomorphic encryption or confidential computing for high-sensitivity workloads where data must remain encrypted during processing (protects against memory scraping).

Green Pace

TRIPLE-A POLICIES

- **Authentication**: Multi-factor, strong passwords, SSO for all logins/API/sessions. Mandatory to verify identity and block unauthorized access.
- **Authorization**: Role-based/least privilege/attribute-based controls on DB changes, files, user management. Mandatory to limit blast radius of compromised accounts.
- **Accounting**: Audit logs for logins, DB changes, new users, access levels, file access. Mandatory with retention and integration into OS/firewall/anti-malware logs for forensics, anomaly detection, compliance.

Green Pace

Unit Testing

(STD-002-CPP & STD-003-CPP)

Does vector at() method throw out of range exception when used on invalid index?

```cpp
// Negative: expects exception (secure behavior)
TEST_F(CollectionTest, Does_vector_at_throw_out_of_range_on_invalid_index)
{
    add_entries(5);  // fill with 5 elements (indices 0-4)
    size_t invalid_index = collection->size();  // == 5 -> out of bounds

    ASSERT_THROW(collection->at(invalid_index), std::out_of_range);
    // Policy benefit: at() enforces bounds -> prevents buffer overflow / invalid read
}
```

Run | Debug

Test Detail Summary

CollectionTest.Does_vector_at_throw_out_of_range_on_invalid_index

Source: test.cpp line 322

Duration: 1 ms

Taking it further

- Add negative tests for at() with negative indices (if using signed type)
- Fuzz index values in CI to increase coverage
- Measure how often [] vs at() is used via static analysis

Green Pace

Does vector at() method allow valid access without throwing exceptions?

```cpp
// Positive: valid access succeeds
TEST_F(CollectionTest, Does_vector_at_allow_valid_access_without_throwing)
{
    add_entries(3);  // indices 0-2
    ASSERT_NO_THROW(collection->at(1));            // valid
    EXPECT_EQ(collection->at(1), collection->operator[](1));  // same value
}
```

Run | Debug

Test Detail Summary

CollectionTest.Does_vector_at_allow_valid_access_without_throwing

Source: test.cpp line 332

Duration: < 1 ms

Taking it further

- Add performance comparison test: [] vs at() in tight loop (document acceptable overhead)
- Use Clang-Wunsafe-buffer-usage or Cppcheck to flag remaining [] usages
- Use at() in untrusted/high-risk paths; [] only when index proven valid

Can operator [] access elements within bounds without throwing exception?

```
// Positive: operator[] works when in bounds
TEST_F(CollectionTest, Can_operator_access_elements_within_bounds_without_issues)
{
    add_entries(4);
    EXPECT_NO_FATAL_FAILURE(collection->operator[](2));   // no crash expected
    EXPECT_EQ(collection->operator[](0), collection->at(0));   // consistent
}
```

Test Detail Summary

CollectionTest.Can_operator_access_elements_within_bounds_without_issues

Source: test.cpp line 340

Duration: < 1 ms

Taking it further

- Pair with static analysis rule: allow [] only after prior size() check in same scope
- Add test coverage goal: ≥ 85% branch coverage on bounds-checking paths
- Policy: Use at() → untrusted [] → proven safe"

Does operator[] lead to undefined behavior when index is out of bounds?

```cpp
// Negative: demonstrate risk of operator[] (UB warning)
TEST_F(CollectionTest, Does_operator_on_out_of_bounds_index_lead_to_undefined_behavior)
{
    add_entries(1);  // only index 0 valid
    size_t invalid = 10;

    // WARNING: This is deliberately unsafe -> UB in real run
    // In practice: run with -fsanitize=address to catch
    // Here we run in Debugger because it performs a safety check
    EXPECT_DEBUG_DEATH(collection->operator[](invalid), ".*");  // optional: if using death test
    // Or comment: "Unsafe - may crash, read garbage, or allow exploits (violates STD-003)"
}
```

Test Detail Summary

CollectionTest.Does_operator_on_out_of_bounds_index_lead_to_undefined_behavior

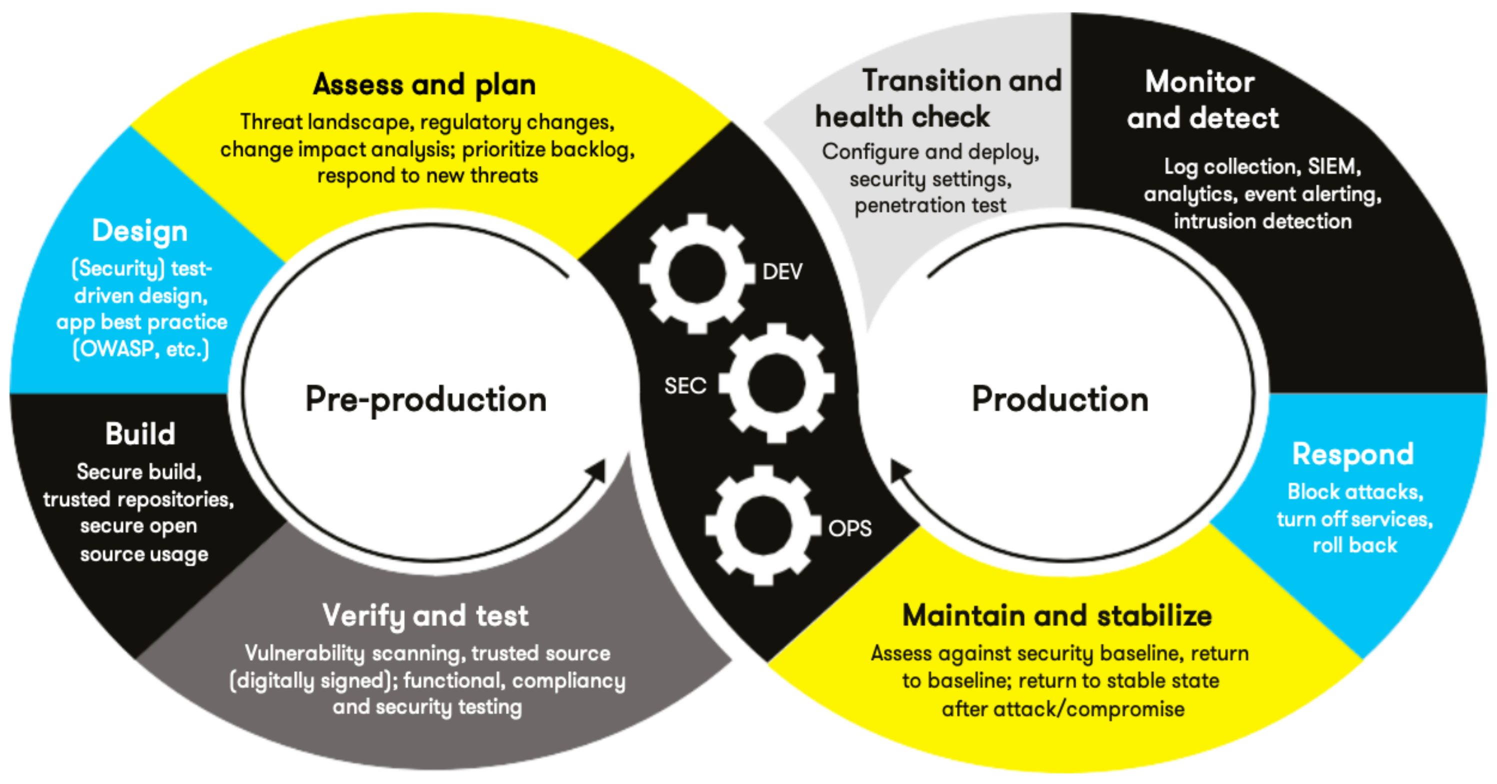
Source: test.cpp line 348

Duration: 3.5 sec

Taking it further

- Run this test under AddressSanitizer (-fsanitize=address) in CI → should crash reliably
- Use death tests (EXPECT_DEBUG_DEATH) only in debug builds

AUTOMATION SUMMARY



Green Pace

TOOLS

- Integrate Clang Static Analyzer, Cppcheck, SonarQube, Coverity into Build/Verify phases of DevSecOps pipeline.
- Run on commit and CI for early detection. Use -fsanitize in Pre-production.
- Feed results to Monitoring/Analytics for dashboards and feedback loops.
- This shifts security left while enforcing defense-in-depth across Pre-production and production pipeline states.

Green Pace

RISKS AND BENEFITS

- Problems: Inconsistent practices incubate security holes for injection, overflows, and leaks.
- Solutions: Enforce policy via standards, automation, Triple-A, encryption.
- Act now: Immediate reduction in attack surface, fewer incidents, compliance readiness.
- Wait: Increased breach risk, higher remediation cost, potential regulatory fines.
- Lacking: Mobile/embedded-specific rules, zero-trust deeper integration.
- Risks: Initial slowdown from automation; false positives. Steps: Prioritize Level 5/4 fixes first, expand automation, annual review.

Green Pace

RECOMMENDATIONS

- Current gaps: No dedicated rules for concurrency/race conditions, secure random number generation, or third-party dependency scanning (SCA).
- Future focus: Adopt SEI CERT rules for uncovered areas, integrate SCA tools (OWASP Dependency-Check), add fuzzing in CI, conduct regular threat modeling.

Green Pace

CONCLUSIONS

While this policy establishes a strong baseline, it is not exhaustive. To further enhance our security posture, we recommend adopting the full SEI CERT C++ rules, OWASP Secure Coding Practices, regular penetration testing, and mandatory code reviews. These additions will drive proactive threat prevention, continuous improvement, and alignment with industry best practices.

Green Pace

REFERENCES

- SEI CERT C++ Coding StandardSoftware Engineering Institute. (2023). SEI CERT C++ Coding Standard. Retrieved February 18, 2026, from https://wiki.sei.cmu.edu/confluence/display/cplusplus2.

- OWASP Secure Coding Practices Quick Reference GuideOWASP Foundation. (2022). OWASP Secure Coding Practices Quick Reference Guide. Retrieved February 18, 2026, from https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/

Green Pace