Mohamed Elhassan  
CS 405 - Secure Coding

Southern New Hampshire University

## **Project One: Security Policy**

[**Table of Contents**](#_3rkocbttuw1j) **2**

[**1. Coding Standards**](#_s27gsmypyo86) **3**

[**2. Risk Assessment**](#_3bkr4fu0fs7r) **5**

[**3. Automated Detection**](#_pa0t2t4d2gf1) **6**

[**4. Automation**](#_mzywu15vfhoy) **7**

[**5. Summary of Risk Assessments**](#_9mvdy4x2dv1s) **8**

[**6. Policies for Encryption and Triple-A**](#_ylvfsp9vagkm) **9**

[**7. Mapping Principles to Standards**](#_46j4kn4bxd2h) **10**

### 

### 1. Coding Standards

For each coding standard, we will define the associated principles, threat level, tools, and provide examples for better clarity.

* **Standard 1: SQL Injection Prevention**

This standard emphasizes Principle 4: *Secure Input Validation* and Principle 5: *Least Privilege*. SQL injection remains a significant security threat, with the potential for severe data breaches or unauthorized manipulation of data. Therefore, it is rated as *High* threat level. To mitigate this, tools like *SonarQube* (version 9.5) can be used to detect unsafe use of user input in SQL queries, while *CppCheck* (version 2.0) flags unsafe concatenation of user input. To prevent SQL injection, developers should always use parameterized queries instead of directly inserting user input into SQL queries.

* **Standard 2: Memory Protection**

Memory protection is vital to prevent vulnerabilities such as buffer overflows. This standard maps to Principle 7: *Minimize Buffer Overflow Risks*. The threat level is *Medium*, as buffer overflows can result in data corruption or arbitrary code execution. To detect buffer overflow risks, tools like *Clang* (version 12) and *CppCheck* (version 2.0) can be employed. Developers should avoid unsafe functions like strcpy() and replace them with safer alternatives like strncpy() to prevent such vulnerabilities.

* **Standard 3: String Correctness**

This standard refers to Principle 2: *Validate String Inputs*. Improper string handling can lead to errors or security holes, making this a *Medium* threat level. Tools like *SonarQube* (version 9.5) and *Clang Static Analyzer* can help identify issues in string handling, such as null pointer dereferencing. It's essential to validate string inputs, especially those passed to functions that expect specific formats, such as email addresses or passwords.

* **Standard 4: Assertions**

Assertions help ensure that certain conditions hold true during the execution of a program, and this standard is based on Principle 6: *Ensure Proper Use of Assertions*. The threat level here is *Low*, as incorrect or missing assertions generally lead to undetected programming errors. *CppCheck* (version 2.0) can detect invalid or missing assertions. Assertions should be used to check conditions that should always be true, such as ensuring variables remain within expected ranges.

* **Standard 5: Exception Handling**

Proper exception handling ensures a program gracefully manages unexpected events. This standard aligns with Principle 8: *Handle Exceptions Gracefully*. The threat level for this standard is *Medium*, as poor exception handling can lead to system crashes or undefined behavior. Tools like *SonarQube* (version 9.5) and *CppCheck* (version 2.0) can identify improper exception handling practices. Developers must catch exceptions and log them appropriately, while ensuring that sensitive information is not exposed in error messages.

### 

### 2. Risk Assessment

For each of the coding standards, a risk assessment is performed, evaluating the severity, likelihood, remediation cost, priority, and level of each vulnerability.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Medium | Likely | High | Medium | 3 |
| STD-003-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-004-CPP | Low | Unlikely | Low | Low | 4 |
| STD-005-CPP | Medium | Likely | Medium | High | 2 |

### 

### 3. Automated Detection

Tools such as *SonarQube* and *CppCheck* help automate the detection of security vulnerabilities in code. For SQL injection, *SonarQube* (version 9.5) detects unsafe user input in SQL queries by flagging the use of direct concatenation. This tool ensures that queries are properly parameterized, preventing SQL injection. For memory protection, *CppCheck* (version 2.0) flags potential buffer overflows, helping developers address issues with unsafe memory functions. When it comes to assertions, *CppCheck* also plays a crucial role by detecting missing or invalid assertions, which helps ensure that critical assumptions in code are verified.

### 4. Automation

Automating security checks as part of the CI/CD pipeline ensures that security is integrated continuously throughout the development lifecycle. For example, tools like *SonarQube* and *CppCheck* should be incorporated into the build process to automatically scan for issues like SQL injection, memory overflows, and improper exception handling. These tools should run every time new code is pushed to the repository, providing real-time feedback to developers. By automating these checks, we can ensure that security vulnerabilities are caught early, preventing them from reaching production.

### 

### 5. Summary of Risk Assessments

The consolidated table provides an overview of the severity, likelihood, remediation cost, priority, and level for each of the 10 coding standards:

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Medium | Likely | High | Medium | 3 |
| STD-003-CPP | Medium | Likely | Medium | Medium | 3 |
| STD-004-CPP | Low | Unlikely | Low | Low | 4 |
| STD-005-CPP | Medium | Likely | Medium | High | 2 |

### 6. Policies for Encryption and Triple-A

**Encryption Policies**

* Encryption at Rest: Encryption at rest ensures that sensitive data stored in databases or file systems is protected. By encrypting the data on disk, even if an attacker gains physical access to the storage, they will not be able to read the data without the decryption key. This policy applies whenever sensitive data is stored, such as personal information or financial records. It helps protect data in the event of a physical breach.
* Encryption in Flight: Encryption in flight protects data as it is transmitted over networks, ensuring that it remains confidential and secure during transit. This policy applies when data is exchanged over the internet or internal networks, such as sending credentials or payment information. It uses protocols like TLS to encrypt data during transmission, making it unreadable to attackers intercepting the data in transit.
* Encryption in Use: Encryption in use refers to the protection of data while it is being processed in memory. This policy applies when sensitive data is actively being used by the application. Techniques like homomorphic encryption can enable processing of encrypted data without the need to decrypt it, ensuring that sensitive data is never exposed during processing.

**Triple-A Framework Policies**

* Authentication: Authentication is the process of verifying the identity of a user or system. This policy applies when a user attempts to access secure resources. Multi-factor authentication (MFA) is an effective method for verifying the user’s identity, making it harder for attackers to gain unauthorized access.
* Authorization: Authorization ensures that users only have access to the resources they are permitted to. This policy enforces role-based access control (RBAC), ensuring that users are granted access only to the resources required for their role, minimizing the risk of unauthorized access.
* Accounting: Accounting refers to the tracking and recording of user actions to ensure accountability. This policy applies to any system where monitoring user actions is necessary for auditing or security purposes. Activity logs should be maintained, and any suspicious or unauthorized activities should be flagged for further review.

### 7. Mapping Principles to Standards

Each of the coding standards links to one or more principles that guide secure development. For example, SQL Injection Prevention aligns with Principle 4 (Secure Input Validation) to ensure user inputs are properly sanitized before being used in SQL queries. It also ties to Principle 5 (Least Privilege), as parameterized queries reduce the risk of unauthorized access to the database.

By mapping these principles to the standards, we ensure that the security policy is rooted in widely recognized best practices and follows a structured approach to secure software development.

This structure offers a security policy that addresses the standards, risk assessments, automated detection tools, encryption policies, and the Triple-A framework. Each section builds upon the other to ensure a strong, defensible security posture.