**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Any time software must handle string inputs, we must be careful to validate and sanitize the input. Without doing so may allow actors to input malicious code leading to unintended behavior in our software. Whether it is out of range input or an SQL injection it is important to be mindful to scrub string input. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user-controlled files [Seacord 05]. |
| 1. Heed Compiler Warnings | We receive these warnings for a reason! Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [C MSC00-A, C++ MSC00-A]. Use static and dynamic analysis tools to detect and eliminate additional security flaws. |
| 1. Architect and Design for Security Policies | Our software should implement and enforce security policies such as requiring different privileges at different times. Our system should be divided into distinct intercommunicating subsystems, with their own appropriate privilege set. |
| 1. Keep It Simple | Keep the design as simple and small as possible [Saltzer 74, Saltzer 75]. As systems grow in complexity, so do the chances of introducing errors in implementation, configuration, and use. As security mechanisms become more complex, we require additional effort to achieve an appropriate level of assurance. |
| 1. Default Deny | All access decisions should be based on permission as opposed to exclusion. We should deny access by default and identify conditions under which access is permitted. |
| 1. Adhere to the Principle of Least Privilege | Each process should execute with the least privileges necessary to complete the job. Elevated permission granted should be for the least amount of time it takes to complete the privileged task. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges [Saltzer 74, Saltzer 75]. |
| 1. Sanitize Data Sent to Other Systems | All data passed to complex subsystems [C STR02-A] such as command shells, relational databases, and commercial off-the-shelf (COTS) components must be sanitized. SQL and other types of injection attacks may invoke unused functionality or result in unintended behavior. A complex subsystem does not understand the context in which a call is made. For this reason, the calling process sanitizes the data because it understands the context of the call. |
| 1. Practice Defense in Depth | Multiple defensive strategies offer us better leverage to manage risk. In the case that one layer turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. Combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment [Seacord 05]. |
| 1. Use Effective Quality Assurance Techniques | The better our quality assurance practices, the more effective we are at identifying and eliminating vulnerabilities. Fuzz/Pen testing and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions [Seacord 05]. |
| 1. Adopt a Secure Coding Standard | Develop and/or apply a secure coding standard for your target development language and platform. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | INT31-C | Ensure that integer conversions do not result in lost or misinterpreted data.  Integer conversions, both implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data. This rule is particularly true for integer values that originate from untrusted sources and are used in any of the following ways:   * Integer operands of any pointer arithmetic, including array indexing * The assignment expression for the declaration of a variable length array * The postfix expression preceding square brackets [] or the expression in square brackets [] of a subscripted designation of an element of an array object * Function arguments of type size\_t or rsize\_t (for example, an argument to a memory allocation function)   This rule also applies to arguments passed to the following library functions that are converted to unsigned char:   * memset() * memset\_s() * fprintf() and related functions (For the length modifier c, if no l length modifier is present, the int argument is converted to an unsigned char, and the resulting character is written.) * fputc() * ungetc() * memchr()   and to arguments to the following library functions that are converted to char:   * strchr() * strrchr() * All of the functions listed in <ctype.h>   The only integer type conversions that are guaranteed to be safe for all data values and all possible conforming implementations are conversions of an integral value to a wider type of the same signedness. The C Standard, subclause 6.3.1.3 [[ISO/IEC 9899:2011](https://wiki.sei.cmu.edu/confluence/display/c/AA.+Bibliography#AA.Bibliography-ISO-IECTR24731-2-2010)], says  When a value with integer type is converted to another integer type other than \_Bool, if the value can be represented by the new type, it is unchanged.  Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.  Otherwise, the new type is signed and the value cannot be represented in it; either the result is [implementation-defined](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-implementation-definedbehavior) or an implementation-defined signal is raised.  Typically, converting an integer to a smaller type results in truncation of the high-order bits. |

| **Noncompliant Code** |
| --- |
| Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of an unsigned integer type to a value of a signed integer type. This noncompliant code example results in a truncation error on most implementations: |
| #include <limits.h>    **void** func(**void**) {    unsigned **long** **int** u\_a = ULONG\_MAX;  **signed** **char** sc;    sc = (**signed** **char**)u\_a; /\* Cast eliminates warning \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Validate ranges when converting from an unsigned type to a signed type. This compliant solution can be used to convert a value of unsigned long int type to a value of signed char type: |
| #include <limits.h>    **void** func(**void**) {    unsigned **long** **int** u\_a = ULONG\_MAX;  **signed** **char** sc;  **if** (u\_a <= SCHAR\_MAX) {      sc = (**signed** **char**)u\_a;  /\* Cast eliminates warning \*/    } **else** {      /\* Handle error \*/    }  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** ValidateInput Data (While this deals with integers, we must validate input data especially if our system converts to strings)/Sanitize Data Sent to Other Systems (This deals with the casting in the compliant sample. I considered this sanitizing and chances are that this data may be sent somewhere) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | **P6** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 |  | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7 |
| Cppcheck | 1.66 | memsetValueOutOfRange | The second argument to memset() cannot be represented as unsigned char |
| RuleChecker | 20.10 |  | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7 |
| TrustInSoft Analyzer | 1.38 | signed\_downcast | Exhaustively verified. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | DCL30-C | Declare objects with appropriate storage durations  Do not attempt to access an object outside of its lifetime. Attempting to do so is undefined behavior and can lead to an exploitable vulnerability. (See also undefined behavior 9 in the C Standard, Annex J.) |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the address of the variable c\_str with automatic storage duration is assigned to the variable p, which has static storage duration. The assignment itself is valid, but it is invalid for c\_str to go out of scope while p holds its address, as happens at the end of dont\_do\_this(). |
| #include <stdio.h>    **const** **char** \*p;  **void** dont\_do\_this(**void**) {  **const** **char** c\_str[] = "This will change";    p = c\_str; /\* Dangerous \*/  }    **void** innocuous(**void**) {  **printf**("%s\n", p);  }    **int** main(**void**) {    dont\_do\_this();    innocuous();  **return** 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, p is declared with the same storage duration as c\_str, preventing p from taking on an indeterminate value outside of this\_is\_OK(): |
| **void** this\_is\_OK(**void**) {  **const** **char** c\_str[] = "Everything OK";  **const** **char** \*p = c\_str;    /\* ... \*/  }  /\* p is inaccessible outside the scope of string c\_str \*/ |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** Architect and Design for Security Policies (This is arguably a rule concerning design. It is important to understand object lifetimes are exploitable)/Practice Defense in Depth(This crosses over with the previous explanation) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | **P6** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | **pointered-deallocation**  **return-reference-local** | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-DCL30** | Fully implemented |
| CodeSonar | 6.1p0 | **LANG.STRUCT.RPL** | Returns pointer to local |
| LDRA tool suite | 9.7.1 | **42 D, 77 D, 71 S, 565 S** | Enhanced Enforcement |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR31-C | Guarantee that storage for strings has sufficient space for character data and the null terminator  Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013b]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character. (See STR03-C. Do not inadvertently truncate a string.)  When strings live on the heap, this rule is a specific instance of MEM35-C. Allocate sufficient memory for an object. Because strings are represented as arrays of characters, this rule is related to both ARR30-C. Do not form or use out-of-bounds pointers or array subscripts and ARR38-C. Guarantee that library functions do not form invalid pointers. |

| **Noncompliant Code** |
| --- |
| Noncompliant Code Example (Off-by-One Error)  This noncompliant code example demonstrates an off-by-one error [Dowd 2006]. The loop copies data from src to dest. However, because the loop does not account for the null-termination character, it may be incorrectly written 1 byte past the end of dest. |
| #include <stddef.h>    **void** copy(**size\_t** n, **char** src[n], **char** dest[n]) {  **size\_t** i;    **for** (i = 0; src[i] && (i < n); ++i) {       dest[i] = src[i];     }     dest[i] = '\0';  } |

| **Compliant Code** |
| --- |
| Compliant Solution (Off-by-One Error)  In this compliant solution, the loop termination condition is modified to account for the null-termination character that is appended to dest: |
| #include <stddef.h>    **void** copy(**size\_t** n, **char** src[n], **char** dest[n]) {  **size\_t** i;    **for** (i = 0; src[i] && (i < n - 1); ++i) {       dest[i] = src[i];     }     dest[i] = '\0';  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** ValidateInput Data (All strings must be validated)/Architect and Design for Security Policies (Understand that string use is exploitable)/Practice Defense in Depth (Address buffer overflows) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | **P18** | **L1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | **STRING\_OVERFLOW**  **BUFFER\_SIZE**  **OVERRUN**  **STRING\_SIZE** | Fully implemented |
| Parasoft C/C++test | 2021.2 | **CERT\_C-STR31-a**  **CERT\_C-STR31-b**  **CERT\_C-STR31-c**  **CERT\_C-STR31-d**  **CERT\_C-STR31-e** | Avoid accessing arrays out of bounds  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Avoid using unsafe string functions which may cause buffer overflows |
| Splint | 3.1.1 |  |  |
| TrustInSoft Analyzer | 1.38 | **mem\_access** | Exhaustively verified |

#### 

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | IDS00-J  (I had difficulty locating this standard for C/C++) | SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the [untrusted data](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-untrusteda) may maliciously alter the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are [sanitization](https://wiki.sei.cmu.edu/confluence/display/java/Rule+BB.+Glossary#RuleBB.Glossary-sa) and validation, which are typically implemented as parameterized queries and stored procedures.  Suppose a system authenticates users by issuing the following query to a SQL database. If the query returns any results, authentication succeeds; otherwise, authentication fails.   |  | | --- | | SELECT \* FROM db\_user WHERE username='<USERNAME>' AND                              password='<PASSWORD>' |   Suppose an attacker can substitute arbitrary strings for <USERNAME> and <PASSWORD>. In that case, the authentication mechanism can be bypassed by supplying the following <USERNAME> with an arbitrary password:   |  | | --- | | validuser' OR '1'='1 |   The authentication routine dynamically constructs the following query:   |  | | --- | | SELECT \* FROM db\_user WHERE username='validuser' OR '1'='1' AND password='<PASSWORD>' |   If validuser is a valid user name, this SELECT statement yields the validuser record in the table. The password is never checked because username='validuser' is true; consequently, the items after the OR are not tested. As long as the components after the OR generate a syntactically correct SQL expression, the attacker is granted the access of validuser.  Similarly, an attacker could supply the following string for <PASSWORD> with an arbitrary username:   |  | | --- | | ' OR '1'='1 |   producing the following query:   |  | | --- | | SELECT \* FROM db\_user WHERE username='<USERNAME>' AND password='' OR '1'='1' |   '1'='1' always evaluates to true, causing the query to yield every row in the database. In this scenario, the attacker would be authenticated without needing a valid username or password. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example is vulnerable to an “OR value=value;” type SQL Injection. |
| // SQLInjection.cpp : This file contains the 'main' function. Program execution begins and ends there.  //  #include <algorithm>  #include <iostream>  #include <locale>  #include <tuple>  #include <vector>  #include "sqlite3.h"  // DO NOT CHANGE  typedef std::tuple<std::string, std::string, std::string> user\_record;  const std::string str\_where = " where ";  // DO NOT CHANGE  static int callback(void\* possible\_vector, int argc, char\*\* argv, char\*\* azColName)  {  if (possible\_vector == NULL)  { // no vector passed in, so just display the results  for (int i = 0; i < argc; i++)  {  std::cout << azColName[i] << " = " << (argv[i] ? argv[i] : "NULL") << std::endl;  }  std::cout << std::endl;  }  else  {  std::vector< user\_record >\* rows =  static\_cast<std::vector< user\_record > \*>(possible\_vector);  rows->push\_back(std::make\_tuple(argv[0], argv[1], argv[2]));  }  return 0;  }  // DO NOT CHANGE  bool initialize\_database(sqlite3\* db)  {  char\* error\_message = NULL;  std::string sql = "CREATE TABLE USERS(" \  "ID INT PRIMARY KEY NOT NULL," \  "NAME TEXT NOT NULL," \  "PASSWORD TEXT NOT NULL);";  int result = sqlite3\_exec(db, sql.c\_str(), callback, NULL, &error\_message);  if (result != SQLITE\_OK)  {  std::cout << "Failed to create USERS table. ERROR = " << error\_message << std::endl;  sqlite3\_free(error\_message);  return false;  }  std::cout << "USERS table created." << std::endl;  // insert some dummy data  sql = "INSERT INTO USERS (ID, NAME, PASSWORD)" \  "VALUES (1, 'Fred', 'Flinstone');" \  "INSERT INTO USERS (ID, NAME, PASSWORD)" \  "VALUES (2, 'Barney', 'Rubble');" \  "INSERT INTO USERS (ID, NAME, PASSWORD)" \  "VALUES (3, 'Wilma', 'Flinstone');" \  "INSERT INTO USERS (ID, NAME, PASSWORD)" \  "VALUES (4, 'Betty', 'Rubble');";  result = sqlite3\_exec(db, sql.c\_str(), callback, NULL, &error\_message);  if (result != SQLITE\_OK)  {  std::cout << "Data failed to insert to USERS table. ERROR = " << error\_message << std::endl;  sqlite3\_free(error\_message);  return false;  }  return true;  }  bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  // TODO: Fix this method to fail and display an error if there is a suspected SQL Injection  // NOTE: You cannot just flag 1=1 as an error, since 2=2 will work just as well. You need  // something more generic  // clear any prior results  records.clear();  char\* error\_message;  if(sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message) != SQLITE\_OK)  {  std::cout << "Data failed to be queried from USERS table. ERROR = " << error\_message << std::endl;  sqlite3\_free(error\_message);  return false;  }  return true;  }  // DO NOT CHANGE  bool run\_query\_injection(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  std::string injectedSQL(sql);  std::string localCopy(sql);  // we work on the local copy because of the const  std::transform(localCopy.begin(), localCopy.end(), localCopy.begin(), ::tolower);  if(localCopy.find\_last\_of(str\_where) >= 0)  { // this sql has a where clause  if(localCopy.back() == ';')  { // smart SQL developer terminated with a semicolon - we can fix that!  injectedSQL.pop\_back();  }  switch (rand() % 4)  {  case 1:  injectedSQL.append(" or 2=2;");  break;  case 2:  injectedSQL.append(" or 'hi'='hi';");  break;  case 3:  injectedSQL.append(" or 'hack'='hack';");  break;  case 0:  default:  injectedSQL.append(" or 1=1;");  break;  }  }    return run\_query(db, injectedSQL, records);  }  // DO NOT CHANGE  void dump\_results(const std::string& sql, const std::vector< user\_record >& records)  {  std::cout << std::endl << "SQL: " << sql << " ==> " << records.size() << " records found." << std::endl;  for (auto record : records)  {  std::cout << "User: " << std::get<1>(record) << " [UID=" << std::get<0>(record) << " PWD=" << std::get<2>(record) << "]" << std::endl;  }  }  // DO NOT CHANGE  void run\_queries(sqlite3\* db)  {  char\* error\_message = NULL;  std::vector< user\_record > records;  // query all  std::string sql = "SELECT \* from USERS";  if (!run\_query(db, sql, records)) return;  dump\_results(sql, records);  // query 1  sql = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='Fred'";  if (!run\_query(db, sql, records)) return;  dump\_results(sql, records);  // run query 1 with injection 5 times  for (auto i = 0; i < 5; ++i)  {  if (!run\_query\_injection(db, sql, records)) continue;  dump\_results(sql, records);  }  }  // You can change main by adding stuff to it, but all of the existing code must remain, and be in the  // in the order called, and with none of this existing code placed into conditional statements  int main()  {  // initialize random seed:  srand(time(nullptr));  int return\_code = 0;  std::cout << "SQL Injection Example" << std::endl;  // the database handle  sqlite3\* db = NULL;  char\* error\_message = NULL;  // open the database connection  int result = sqlite3\_open(":memory:", &db);  if(result != SQLITE\_OK)  {  std::cout << "Failed to connect to the database and terminating. ERROR=" << sqlite3\_errmsg(db) << std::endl;  return -1;  }  std::cout << "Connected to the database." << std::endl;  // initialize our database  if(!initialize\_database(db))  {  std::cout << "Database Initialization Failed. Terminating." << std::endl;  return\_code = -1;  }  else  {  run\_queries(db);  }  // close the connection if opened  if(db != NULL)  {  sqlite3\_close(db);  }  return return\_code;  } |

| **Compliant Code** |
| --- |
| This implements handling of the SQL injection using conditional logic and replace\_substring() function to scrub input data. |
| // SQLInjection.cpp : This file contains the 'main' function. Program execution begins and ends there.  //  #include <algorithm>  #include <iostream>  #include <iomanip>  #include <sstream>  #include <locale>  #include <tuple>  #include <vector>  #include <string>  #include "sqlite3.h"  // DO NOT CHANGE  typedef std::tuple<std::string, std::string, std::string> user\_record;  const std::string str\_where = " where ";  // DO NOT CHANGE  static int callback(void\* possible\_vector, int argc, char\*\* argv, char\*\* azColName)  {  if (possible\_vector == NULL)  { // no vector passed in, so just display the results  for (int i = 0; i < argc; i++)  {  std::cout << azColName[i] << " = " << (argv[i] ? argv[i] : "NULL") << std::endl;  }  std::cout << std::endl;  }  else  {  std::vector< user\_record >\* rows =  static\_cast<std::vector< user\_record > \*>(possible\_vector);  rows->push\_back(std::make\_tuple(argv[0], argv[1], argv[2]));  }  return 0;  }  // DO NOT CHANGE  bool initialize\_database(sqlite3\* db)  {  char\* error\_message = NULL;  std::string sql = "CREATE TABLE USERS(" \  "ID INT PRIMARY KEY NOT NULL," \  "NAME TEXT NOT NULL," \  "PASSWORD TEXT NOT NULL);";  int result = sqlite3\_exec(db, sql.c\_str(), callback, NULL, &error\_message);  if (result != SQLITE\_OK)  {  std::cout << "Failed to create USERS table. ERROR = " << error\_message << std::endl;  sqlite3\_free(error\_message);  return false;  }  std::cout << "USERS table created." << std::endl;  // insert some dummy data  sql = "INSERT INTO USERS (ID, NAME, PASSWORD)" \  "VALUES (1, 'Fred', 'Flinstone');" \  "INSERT INTO USERS (ID, NAME, PASSWORD)" \  "VALUES (2, 'Barney', 'Rubble');" \  "INSERT INTO USERS (ID, NAME, PASSWORD)" \  "VALUES (3, 'Wilma', 'Flinstone');" \  "INSERT INTO USERS (ID, NAME, PASSWORD)" \  "VALUES (4, 'Betty', 'Rubble');";  result = sqlite3\_exec(db, sql.c\_str(), callback, NULL, &error\_message);  if (result != SQLITE\_OK)  {  std::cout << "Data failed to insert to USERS table. ERROR = " << error\_message << std::endl;  sqlite3\_free(error\_message);  return false;  }  return true;  }  //replace\_substring is required to check input for an sql injection  //The size\_t type used in the for loop is the unsigned integer type that is the result of the sizeof operator (and the offsetof operator)  //so it is guaranteed to be big enough to contain the size of the biggest object your system can handle (e.g., a static array of 8Gb).  std::string replace\_substring(std::string s, const std::string s\_replaced, const std::string s\_replace\_new) {  for (size\_t position = 0; ; position += s\_replace\_new.length()) {  position = s.find(s\_replaced, position);  if (position == std::string::npos || s.empty()) break;  s.erase(position, s\_replaced.length());  s.insert(position, s\_replace\_new);  }  return s;  }  bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  // TODO: Fix this method to fail and display an error if there is a suspected SQL Injection  // NOTE: You cannot just flag 1=1 as an error, since 2=2 will work just as well. You need  // something more generic  // clear any prior results  records.clear();  //Determining if input contains an "or" command. This serves as a warning.  //This would trigger false positives with usernames containing "or" which could be ignored as a threat.  if (sql.find("or") != std::string::npos) {  std::string clean = sql.c\_str();  clean = replace\_substring(clean, "\'", "\"");  std::cout << "Query contains 'OR'. SQL Injection possible." << std::endl;  std::cout << "Stopped Query: " << sql.c\_str() << std::endl;  std::cout << "Clean query: " << clean << std::endl;  return false;    }  char\* error\_message;  if (sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message) != SQLITE\_OK)  {  std::cout << "Data failed to be queried from USERS table. ERROR = " << error\_message << std::endl;  sqlite3\_free(error\_message);  return false;  }  return true;  }  // DO NOT CHANGE  bool run\_query\_injection(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  std::string injectedSQL(sql);  std::string localCopy(sql);  // we work on the local copy because of the const  std::transform(localCopy.begin(), localCopy.end(), localCopy.begin(), ::tolower);  if (localCopy.find\_last\_of(str\_where) >= 0)  { // this sql has a where clause  if (localCopy.back() == ';')  { // smart SQL developer terminated with a semicolon - we can fix that!  injectedSQL.pop\_back();  }  switch (rand() % 4)  {  case 1:  injectedSQL.append(" or 2=2;");  break;  case 2:  injectedSQL.append(" or 'hi'='hi';");  break;  case 3:  injectedSQL.append(" or 'hack'='hack';");  break;  case 0:  default:  injectedSQL.append(" or 1=1;");  break;  }  }  return run\_query(db, injectedSQL, records);  }  // DO NOT CHANGE  void dump\_results(const std::string& sql, const std::vector< user\_record >& records)  {  std::cout << std::endl << "SQL: " << sql << " ==> " << records.size() << " records found." << std::endl;  for (auto record : records)  {  std::cout << "User: " << std::get<1>(record) << " [UID=" << std::get<0>(record) << " PWD=" << std::get<2>(record) << "]" << std::endl;  }  }  // DO NOT CHANGE  void run\_queries(sqlite3\* db)  {  char\* error\_message = NULL;  std::vector< user\_record > records;  // query all  std::string sql = "SELECT \* from USERS";  if (!run\_query(db, sql, records)) return;  dump\_results(sql, records);  // query 1  sql = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='Fred'";  if (!run\_query(db, sql, records)) return;  dump\_results(sql, records);  // run query 1 with injection 5 times  for (auto i = 0; i < 5; ++i)  {  if (!run\_query\_injection(db, sql, records)) continue;  dump\_results(sql, records);  }  }  // You can change main by adding stuff to it, but all of the existing code must remain, and be in the  // in the order called, and with none of this existing code placed into conditional statements  int main()  {  // initialize random seed:  srand(time(nullptr));  int return\_code = 0;  std::cout << "SQL Injection Example" << std::endl;  // the database handle  sqlite3\* db = NULL;  char\* error\_message = NULL;  // open the database connection  int result = sqlite3\_open(":memory:", &db);  if (result != SQLITE\_OK)  {  std::cout << "Failed to connect to the database and terminating. ERROR=" << sqlite3\_errmsg(db) << std::endl;  return -1;  }  std::cout << "Connected to the database." << std::endl;  // initialize our database  if (!initialize\_database(db))  {  std::cout << "Database Initialization Failed. Terminating." << std::endl;  return\_code = -1;  }  else  {  run\_queries(db);  }  // close the connection if opened  if (db != NULL)  {  sqlite3\_close(db);  }  return return\_code;  }  // Run program: Ctrl + F5 or Debug > Start Without Debugging menu  // Debug program: F5 or Debug > Start Debugging menu |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** Validate Input Data (String use is how people utilize SQL Injection)/Architect and Design for Security Policies (SQL Injection is a common exploitable vulnerability that should be designed for)/Sanitize Data Sent to Other Systems (Sending string data can be problematic regarding SQL Injection)/ Practice Defense in Depth (This provides an extra layer of defense) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | **P12** | **L1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | **JAVA.IO.INJ.SQL** | SQL Injection (Java) |
| Coverity | 7.5 | **SQLI**  **FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_**  **FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** | Implemented |
| Fortify | 1.0 | **HTTP\_Response\_Splitting**  **SQL\_Injection\_\_Persistence**  **SQL\_Injection** | Implemented |
| Parasoft Jtest | 2021.2 | **CERT.IDS00.TDSQL** | Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM50-CPP. | Do not access freed memory  Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable vulnerabilities.  It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required.  In the second example we see automatic storage duration. When possible, use automatic storage duration instead of dynamic storage duration. Since s is not required to live beyond the scope of g(), this compliant solution uses automatic storage duration to limit the lifetime of s to the scope of g(). |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...    s->f();  **delete** s;  }  **struct** S {  **void** f();  };    **void** g() {    S s;    // ...    s.f();  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** Architect and Design for Security Policies (This is protection for data in use)/Keep It Simple (Compliance with this rule is a matter of paying attention to your variables/pointers)/Practice Defense in Depth (Memory must be protected) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | **P18** | **L1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | dangling\_pointer\_use |  |
| Helix QAC | 2021.2 | C++4303, C++4304 |  |
| Polyspace Bug Finder | R2021b | CERT C++: MEM50-CPP | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |
| Splint | 5.0 |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | ERR50-CPP | Do not abruptly terminate the program  The std::abort(), std::quick\_exit(), and std::\_Exit() functions are used to terminate the program in an immediate fashion. They do so without calling exit handlers registered with std::atexit() and without executing destructors for objects with automatic, thread, or static storage duration. How a system manages open streams when a program ends is [implementation-defined](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-implementation-definedbehavior) [[ISO/IEC 9899:1999](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO-IEC9899-1999)]. Open streams with unwritten buffered data may or may not be flushed, open streams may or may not be closed, and temporary files may or may not be removed. Because these functions can leave external resources, such as files and network communications, in an indeterminate state, they should be called explicitly only in direct response to a critical error in the application. (See ERR50-CPP-EX1 for more information.)  The std::terminate() function calls the current terminate\_handler function, which defaults to calling std::abort().  The C++ Standard defines several ways in which std::terminate() may be called implicitly by an [implementation](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-implementation) [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)]:   1. When the exception handling mechanism, after completing the initialization of the exception object but before activation of a handler for the exception, calls a function that exits via an exception ([except.throw], paragraph 7)    * See [ERR60-CPP. Exception objects must be nothrow copy constructible](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR60-CPP.+Exception+objects+must+be+nothrow+copy+constructible) for more information. 2. When a throw-expression with no operand attempts to rethrow an exception and no exception is being handled ([except.throw], paragraph 9) 3. When the exception handling mechanism cannot find a handler for a thrown exception ([except.handle], paragraph 9)    * See [ERR51-CPP. Handle all exceptions](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions) for more information. 4. When the search for a handler encounters the outermost block of a function with a noexcept-specification that does not allow the exception ([except.spec], paragraph 9)    * See [ERR55-CPP. Honor exception specifications](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR55-CPP.+Honor+exception+specifications) for more information. 5. When the destruction of an object during stack unwinding terminates by throwing an exception ([except.ctor], paragraph 3)    * See [DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL57-CPP.+Do+not+let+exceptions+escape+from+destructors+or+deallocation+functions) for more information. 6. When initialization of a nonlocal variable with static or thread storage duration exits via an exception ([basic.start.init], paragraph 6)    * See [ERR58-CPP. Handle all exceptions thrown before main() begins executing](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR58-CPP.+Handle+all+exceptions+thrown+before+main%28%29+begins+executing) for more information. 7. When destruction of an object with static or thread storage duration exits via an exception ([basic.start.term], paragraph 1)    * See [DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL57-CPP.+Do+not+let+exceptions+escape+from+destructors+or+deallocation+functions) for more information. 8. When execution of a function registered with std::atexit()or std::at\_quick\_exit() exits via an exception ([support.start.term], paragraphs 8 and 12) 9. When the implementation’s default unexpected exception handler is called ([except.unexpected], paragraph 2) Note that std::unexpected() is currently deprecated. 10. When std::unexpected() throws an exception that is not allowed by the previously violated dynamic-exception-specification, and std::bad\_exception() is not included in that dynamic-exception-specification ([except.unexpected], paragraph 3) 11. When the function std::nested\_exception::rethrow\_nested() is called for an object that has captured no exception ([except.nested], paragraph 4) 12. When execution of the initial function of a thread exits via an exception ([thread.thread.constr], paragraph 5)     * See [ERR51-CPP. Handle all exceptions](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions) for more information. 13. When the destructor is invoked on an object of type std::thread that refers to a joinable thread ([thread.thread.destr], paragraph 1) 14. When the copy assignment operator is invoked on an object of type std::thread that refers to a joinable thread ([thread.thread.assign], paragraph 1) 15. When calling condition\_variable::wait(), condition\_variable::wait\_until(), or condition\_variable::wait\_for() results in a failure to meet the postcondition: lock.owns\_lock() == true or lock.mutex() is not locked by the calling thread ([thread.condition.condvar], paragraphs 11, 16, 21, 28, 33, and 40) 16. When calling condition\_variable\_any::wait(), condition\_variable\_any::wait\_until(), or condition\_variable\_any::wait\_for() results in a failure to meet the postcondition: lock is not locked by the calling thread ([thread.condition.condvarany], paragraphs 11, 16, and 22)   In many circumstances, the call stack will not be unwound in response to an implicit call to std::terminate(), and in a few cases, it is implementation-defined whether or not stack unwinding will occur. The C++ Standard, [except.terminate], paragraph 2 [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)], in part, states the following:  In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before std::terminate() is called. In the situation where the search for a handler encounters the outermost block of a function with a noexcept-specification that does not allow the exception, it is implementation-defined whether the stack is unwound, unwound partially, or not unwound at all before std::terminate() is called. In all other situations, the stack shall not be unwound before std::terminate() is called.  Do not explicitly or implicitly call std::quick\_exit(),  std::abort(), or std::\_Exit(). When the default terminate\_handler is installed or the current terminate\_handler responds by calling std::abort() or std::\_Exit(), do not explicitly or implicitly call std::terminate(). [Abnormal process termination](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-abnormaltermination) is the typical vector for [denial-of-service attacks](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-denial-of-service).  The std::exit() function is more complex. The C++ Standard, [basic.start.main], paragraph 4, states:  Terminating the program without leaving the current block (e.g., by calling the function std::exit(int) (17.5)) does not destroy any objects with automatic storage duration (11.4.6). If std::exit is called to end a program during the destruction of an object with static or thread storage duration, the program has undefined behavior.  You may call std::exit() only in a program that has not yet initialized any objects with automatic storage duration. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the call to f(), which was registered as an exit handler with std::at\_exit(), may result in a call to std::terminate() because throwing\_func() may throw an exception. |
| #include <cstdlib>    **void** throwing\_func() noexcept(**false**);    **void** f() { // Not invoked by the program except as an exit handler.    throwing\_func();  }    **int** main() {  **if** (0 != std::**atexit**(f)) {      // Handle error    }    // ...  } |

| **Compliant Code** |
| --- |
| Solution 1- In this compliant solution, f() handles all exceptions thrown by throwing\_func() and does not rethrow.  Solution 2- ERR50-CPP-EX1: It is acceptable, after indicating the nature of the problem to the operator, to explicitly call std::abort(), std::\_Exit(), or std::terminate() in response to a critical program error for which no recovery is possible, as in this example. |
| Solution 1  #include <cstdlib>    **void** throwing\_func() noexcept(**false**);    **void** f() { // Not invoked by the program except as an exit handler.  **try** {      throwing\_func();    } **catch** (...) {      // Handle error    }  }    **int** main() {  **if** (0 != std::**atexit**(f)) {      // Handle error    }    // ...  }  Solution 2  #include <exception>    **void** report(**const** **char** \*msg) noexcept;  [[**noreturn**]] **void** fast\_fail(**const** **char** \*msg) {    // Report error message to operator    report(msg);      // Terminate    std::terminate();  }    **void** critical\_function\_that\_fails() noexcept(**false**);    **void** f() {  **try** {      critical\_function\_that\_fails();    } **catch** (...) {      fast\_fail("Critical function failure");    }  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** Use Effective Quality Assurance Techniques (All efforts should be made to avoid a programs unintended termination) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | **P4** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | **C++5014** |  |
| PVS-Studio | 7.15 | **V667, V2014** |  |
| LDRA tool suite | 9.7.1 | **122 S** | Enhanced Enforcement |
| Polyspace Bug Finder | R2021b | **CERT C++: ERR50-CPP** | Checks for implicit call to terminate() function (rule partially covered) |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR51-CPP. | Handle all exceptions.  When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. The C++ Standard, [except.handle], paragraph 9 [ISO/IEC 14882-2014], states the following:  If no matching handler is found, the function std::terminate() is called; whether or not the stack is unwound before this call to std::terminate() is implementation-defined.  The default terminate handler called by std::terminate() calls std::abort(), which abnormally terminates the process. When std::abort() is called, or if the implementation does not unwind the stack prior to calling std::terminate(), destructors for objects may not be called and external resources can be left in an indeterminate state. Abnormal process termination is the typical vector for denial-of-service attacks. For more information on implicitly calling std::terminate(), see ERR50-CPP. Do not abruptly terminate the program.  All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process.  As per ERR50-CPP-EX1, a program that encounters an unrecoverable exception may explicitly catch the exception and terminate, but it may not allow the exception to remain uncaught. One possible solution to comply with this rule, as well as with ERR50-CPP, is for the main() function to catch all exceptions. While this does not generally allow the application to recover from the exception gracefully, it does allow the application to terminate in a controlled fashion. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** Heed Compiler Warnings(Exceptions will throw warnings! Exceptions that aren’t caught can do terrible things to external resources if each is not caught.) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | **P4** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-ERR51** |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| PRQA QA-C++ | 4.4 | 4035, 4036, 4037 |  |
| Helix QAC | 2021.2 | C++4035, C++4036, C++4037 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Use valid iterator ranges** | CTR53-CPP. | When iterating over elements of a container, the iterators used must iterate over a valid range. An iterator range is a pair of iterators that refer to the first and past-the-end elements of the range respectively.  A valid iterator range has all of the following characteristics:   * Both iterators refer into the same container. * The iterator representing the start of the range precedes the iterator representing the end of the range. * The iterators are not invalidated, in conformance with [CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR51-CPP.+Use+valid+references%2C+pointers%2C+and+iterators+to+reference+elements+of+a+container).   An empty iterator range (where the two iterators are valid and equivalent) is considered to be valid.  Using a range of two iterators that are invalidated or do not refer into the same container results in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior). |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the two iterators that delimit the range point into the same container, but the first iterator does not precede the second. On each iteration of its internal loop, std::for\_each() compares the first iterator (after incrementing it) with the second for equality; as long as they are not equal, it will continue to increment the first iterator. Incrementing the iterator representing the past-the-end element of the range results in undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {    std::for\_each(c.end(), c.begin(), [](**int** i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the iterator values passed to std::for\_each() are passed in the proper order. |
| #include <algorithm>  #include <iostream>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {    std::for\_each(c.begin(), c.end(), [](**int** i) { std::cout << i; });  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** Heed Compiler Warnings(The compiler or testing tools will warn about this, this must be handled or risk unintended behavior with a high remediation cost.)/Keep It Simple(Pass iterators in the proper order) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | **P6** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | **overflow\_upon\_dereference** |  |
| Helix QAC | 2021.2 | **C++3802** |  |
| Parasoft C/C++test | 2021.2 | **CERT\_CPP-CTR53-a**  **CERT\_CPP-CTR53-b** | Do not use an iterator range that isn't really a range  Do not compare iterators from different containers |
| PRQA QA-C++ | 4.4 | **3802** |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Use valid references, pointers, and iterators to reference elements of a basic\_string** | STR52-CPP. | Since std::basic\_string is a container of characters, this rule is a specific instance of [CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR51-CPP.+Use+valid+references%2C+pointers%2C+and+iterators+to+reference+elements+of+a+container). As a container, it supports iterators just like other containers in the Standard Template Library. However, the std::basic\_string template class has unusual invalidation semantics. The C++ Standard, [string.require], paragraph 5 [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)], states the following:  References, pointers, and iterators referring to the elements of a basic\_string sequence may be invalidated by the following uses of that basic\_string object:   * As an argument to any standard library function taking a reference to non-const basic\_string as an argument. * Calling non-const member functions, except operator[], at, front, back, begin, rbegin, end, and rend.   Examples of standard library functions taking a reference to non-const std::basic\_string are std::swap(), ::operator>>(basic\_istream &, string &), and std::getline().  Do not use an invalidated reference, pointer, or iterator because doing so results in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior). |

| **Noncompliant Code** |
| --- |
| This noncompliant code example copies input into a std::string, replacing semicolon (;) characters with spaces. This example is noncompliant because the iterator loc is invalidated after the first call to insert(). The behavior of subsequent calls to insert() is undefined. |
| #include <string>    **void** f(**const** std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the value of the iterator loc is updated as a result of each call to insert() so that the invalidated iterator is never accessed. The updated iterator is then incremented at the end of the loop. |
| #include <string>    **void** f(**const** std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      loc = email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** ValidateInput Data (Do not use an invalidated reference, pointer, or iterator because doing so results in undefined behavior) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | **P6** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | **C++4746, C++4747, C++4748, C++4749** |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Detect and handle memory allocation errors** | MEM52-CPP | The default memory allocation operator, ::operator new(std::size\_t), throws a std::bad\_alloc exception if the allocation fails. Therefore, you need not check whether calling ::operator new(std::size\_t) results in nullptr. The nonthrowing form, ::operator new(std::size\_t, const std::nothrow\_t &), does not throw an exception if the allocation fails but instead returns nullptr. The same behaviors apply for the operator new[] versions of both allocation functions. Additionally, the default allocator object (std::allocator) uses ::operator new(std::size\_t) to perform allocations and should be treated similarly.  Furthermore, operator new[] can throw an error of type std::bad\_array\_new\_length, a subclass of std::bad\_alloc, if the size argument passed to new is negative or excessively large.  When using the nonthrowing form, it is imperative to check that the return value is not nullptr before accessing the resulting pointer. When using either form, be sure to comply with ERR50-CPP. Do not abruptly terminate the program. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. The function is marked as noexcept, so the caller assumes this function does not throw any exceptions. Because ::operator new[](std::size\_t) can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| #include <cstring>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** **int**[size];    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

| **Compliant Code** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| #include <cstring>  #include <new>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** (std::**nothrow**) **int**[size];  **if** (!copy) {      // Handle error  **return**;    }    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| **Principles(s):** Practice Defense in Depth (As mentioned in a previous rule, abnormal termination must be avoided at all cost and all exceptions must be caught. Referencing a null pointer by accident can have disastrous results.) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | **P18** | **L1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | **CHECKED\_RETURN** |  |
| Parasoft C/C++test | 2021.2 | **CERT\_CPP-MEM52-a**  **CERT\_CPP-MEM52-b** | Check the return value of new  Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| PRQA QA-C++ | 4.4 | **3225, 3226, 3227, 3228, 3229, 4632** |  |
| Helix QAC | 2021.2 | **C++3225, C++3226, C++3227, C++3228, C++3229, C++4632** |  |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

When considering the legacy system in place, we must understand that it may not be capable of delivering new software at the speed of operations. In a previous system security may not have been a priority, and we must keep in mind that security measures may not have been implemented until later in the lifecycle of an application or the underlying infrastructure. The goal of DevSecOps is to unify software development, security, and operations. The main characteristic of this culture/practice is to automate, monitor, and apply security to every phase of the software lifecycle: Plan, Design, Build, Verify/test, transition, and health check, Monitor/Detect, Respond, Maintain & Stabilize. DevSecOps is unique in that security and functional capabilities are tested and built simultaneously.

Benefits of adopting DevSecOps:

* Reduced mean-time to production.
* Increased deployment frequency
* Risk characterization, monitoring, and mitigation is fully automated across the application lifecycle
* Updates and patches released at “the speed of operations”.

The image provided illustrates the importance of prioritizing security in every phase of the development lifecycle and provides information about where to focus efforts.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| CTR53-CPP | High | Probable | High | **P6** | **L2** |
| DCL30-C | High | Probable | High | **P6** | **L2** |
| ERR50-CPP | Low | Probable | Medium | **P4** | **L3** |
| ERR51-CPP | Low | Probable | Medium | **P4** | **L3** |
| IDS00-J | High | Probable | Medium | **P12** | **L1** |
| INT31-C | High | Probable | High | **P6** | **L2** |
| MEM50-CPP | High | Likely | Medium | **P18** | **L1** |
| MEM52-CPP | High | Likely | Medium | **P18** | **L1** |
| STR31-C | High | Likely | Medium | **P18** | **L1** |
| STR52-CPP | High | Probable | High | **P6** | **L2** |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Protecting data that is stored on devices. This is an important aspect of any defense as all stored data must be protected. |
| Encryption at flight | Protecting data that is being sent. Encrypting in-flight data to secure communications across fiber-optic cables can help eliminate gaps within your data security strategy. |
| Encryption in use | Like encryption in rest, however the data is in memory instead of on disk. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication provides a method of identifying a user, typically by having the user enter a valid username and password before access to the network is granted. Authentication is based on each user having a unique set of login credentials for gaining network access.  The AAA server compares a user's authentication credentials with other user credentials stored in a database; in this case, that database is Active Directory. If the user's login credentials match, the user is granted access to the network. If the credentials don't match, authentication fails, and network access is denied. |
| Authorization | Following authentication, a user must gain authorization for doing certain tasks. After logging in to a system, for instance, the user may try to issue commands. The authorization process determines whether the user has the authority to issue such commands.  Simply put, authorization is the process of enforcing policies—determining what types or qualities of activities, resources, or services a user is permitted. Usually, authorization occurs within the context of authentication. After you have authenticated a user, they may be authorized for different types of access or activity.  As it relates to network authentication via RADIUS and 802.1x, authorization can be used to determine what VLAN, Access Control List (ACL), or user role that the user belongs to. |
| Accounting | The final piece in the AAA framework is accounting, which monitors the resources a user consumes during network access. This can include the amount of system time or the amount of data sent and received during a session.  Accounting is carried out by logging session statistics and usage information. It is used for authorization control, billing, trend analysis, resource utilization, and planning for the data capacity required for business operations.  ClearPass Policy Manager functions as the accounting server and receives accounting information about the user from the Network Access Server (NAS). The NAS must be configured to use ClearPass Policy Manager as an accounting server, and it is up to the NAS to provide accurate accounting information to ClearPass Policy Manager. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 12/7/2021 | Project One Update | Dylan Jeffery | Trevor Hodde |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |