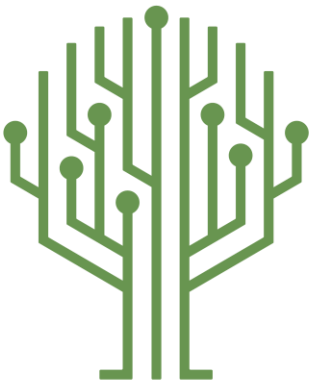


Green Pace Secure Development Policy

Contents

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.

Scope

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

Module Three Milestone

Ten Core Security Principles

Reference website: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Principles	Write a short paragraph explaining each of the 10 principles of security.
1. Validate Input Data	Validation of input data is critical to eliminate the most of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user-controlled files.
2. Heed Compiler Warnings	Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code. Use static and dynamic analysis tools to detect and eliminate additional security flaws.
3. Architect and Design for Security Policies	Security policies should be considered at the early phases of software life cycle such as architecture and design. Ensure that architecture and design enforce the security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.
4. Keep It Simple	Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.
5. Default Deny	Base the access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted.
6. Adhere to the Principle of Least Privilege	Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should only be given for the least amount of time required to complete the privileged task. This approach reduces the opportunities for attacker to execute arbitrary code with elevated privileges.

Principles	Write a short paragraph explaining each of the 10 principles of security.
7. Sanitize Data Sent to Other Systems	Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.
8. Practice Defense in Depth	Manage risk with multiple defensive strategies, so that if one layer of defense 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. For example, 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.
9. Use Effective Quality Assurance Techniques	Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration 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.
10. Adopt a Secure Coding Standard	Each language has its own coding standards. 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.

Reference website: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Coding Standard 1

Coding Standard	Label	Name of Standard
Data Type	STD-001-CPP	Do not pass a nonstandard-layout type object across execution boundaries. An *execution boundary* is the delimitation between code compiled by differing compilers, including different versions of a compiler produced by the same vendor.

Noncompliant Code

This noncompliant code example assumes that there is a library whose header is library.h, an application (represented by application.cpp), and that the library and application are not ABI (application binary interface) compatible. Therefore, the contents of library.h constitute an execution boundary. A nonstandard-layout type object S is passed across this execution boundary. Because the layout is not guaranteed to be compatible across the boundary, this results in unexpected behavior.

```
// library.h
struct S {
  virtual void f() { /* ... */ }
};

void func(S &s); // Implemented by the library, calls S::f()

// application.cpp
#include "library.h"

void g() {
  S s;
  func(s);
}
```

Compliant Code

Because the library and application do not conform to the same ABI, this compliant solution modifies the library and application to work with a standard-layout type. Furthermore, it also adds a static_assert() to help guard against future code changes that accidentally modify S to no longer be a standard-layout type.

```
// library.h
#include <type_traits>

struct S {
  void f() { /* ... */ } // No longer virtual
```

Compliant Code

```
};
static_assert(std::is_standard_layout<S>::value, "S is required to be a
standard layout type");

void func(S &s); // Implemented by the library, calls S::f()

// application.cpp
#include "library.h"

void g() {
  S s;
  func(s);
}
```

Principles(s): The principle “Validate Input Data” include two things – type of the data (value) and scope of the variable. In the example of “non-compliant” code given above, the object passed across the execution boundary of the application which could result unexpected behavior.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Medium	**Medium**	**L1**

Automation

Tool	Version	Checker	Description Tool
Clang	3.9	-Wdynamic-class-memaccess	Catches instances where the vtable pointer will be overwritten
Parasoft C/C++test	2020.2	**CERT_CPP-EXP60-a**	Do not pass a nonstandard-layout type object across execution boundaries
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	STD-002-CPP	A value-returning function must return a value from all code paths; otherwise, it will result in undefined behavior

Noncompliant Code

The programmer forgot to return the input value for positive input, so not all code paths return a value.

```
int absolute_value(int a) {
  if (a < 0) {
    return -a;
  }
}
```

Compliant Code

Add a return statement for the positive code path

```
int absolute_value(int a) {
  if (a < 0) {
    return -a;
  }
  return a;
}
```

Principles(s): The function should return a value in all paths otherwise it will result an unexpected behavior. This coding standard maps with ”Validate Input Data” principle as well

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	**Medium**	**L2**

Automation

Tool	Version	Checker	Description Tool
Astrée	20.10	**return-implicit**	Fully checked

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	6.9.0	**CertC++-MSC52**	
Clang	3.9	**-Wreturn-type**	Does not catch all instances of this rule, such as *function-try-blocks*
CodeSonar	6.0p0	**LANG.STRUCT.MRS**	Missing return statement

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	STD-003-CPP	Detect errors when converting a string to a number. Always explicitly check the error state of a conversion from string to a numeric value (or handle the related exception, if applicable) instead of assuming the conversion results in a valid value

Noncompliant Code

If the text received from the standard input stream cannot be converted into a numeric value that can be represented by an int, the resulting value stored into the variables i and j may be unexpected.

```
#include <iostream>

void f() {
  int i, j;
  std::cin >> i >> j;
  // ...
}
```

Compliant Code

In this compliant solution, exceptions are enabled so that any conversion failure results in an exception being thrown. However, this approach cannot distinguish between which values are valid and which values are invalid and must assume that all values are invalid. Both the badbit and failbit flags are set to ensure that conversion errors as well as loss of integrity with the stream are treated as exceptions.

```
#include <iostream>

void f() {
  int i, j;

  std::cin.exceptions(std::istream::failbit | std::istream::badbit);
  try {
    std::cin >> i >> j;
    // ...
  } catch (std::istream::failure &E) {
    // Handle error
  }
}
```

Principles(s): This coding standard maps to “Defense in Depth” principle. If there is any error with conversion of string into integer, the try/catch layer will catch the error and handles it.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	**Low**	**L3**

Automation

Tool	Version	Checker	Description Tool

Tool	Version	Checker	Description
Axivion Bauhaus Suite	6.9.0	**CertC-ERR34**	
Clang	3.9	`cert-err34-c`	Checked by `clang-tidy`
CodeSonar	6.0p0	**BADFUNC.ATOF** **BADFUNC.ATOI** **BADFUNC.ATOL** **BADFUNC.ATOLL** **(customization)**	Use of atof Use of atoi Use of atol Use of atoll Users can add custom checks for uses of other undesirable conversion functions.
Compass/ROSE			Can detect violations of this recommendation by flagging invocations of the following functions: • • `atoi()` • `scanf()`, `fscanf()`, `sscanf()` • Others?

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	STD-004-CPP	SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the untrusted data may maliciously alter the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are sanitization and validation, which are typically implemented as parameterized queries and stored procedures.

Noncompliant Code

If an SQL statement is passed as “SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='Fred' OR 1 = 1; “. Where clause always be true which returns all the row by causing 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(" \
```

Noncompliant Code

```
    "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
```

Noncompliant Code

```
  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;");
```

Noncompliant Code

```
      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);
  }
```

Noncompliant Code

```
}

// 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);
  }
```

Noncompliant Code

```
  return return_code;
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu
```

Compliant Code

Verify the SQL statement to detect if any of the elements could cause SQL injection and remove the same.

```
// 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 <regex>

#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);
```

Compliant Code

```
        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;
}
```

Compliant Code

```
int find_match_position(std::string sql)
{
    std::smatch m;
    std::regex expression1("or [1-9]+=[1-9]+;");
    std::regex expression2("or '[a-zA-Z]+'='[a-zA-Z']+';");

    std::regex_search(sql, m, expression1);

    if (m.position(0) == 0)
        std::regex_search(sql, m, expression2);

    return m.position(0);
}

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;
    // check if there is any part of the sql statement could cause 'sql
injection'
    int pos = find_match_position(sql.c_str());
    std::string sql1 = sql.c_str();
    char temp[300];
    if (pos == 0)
        pos = sql1.length();

    if (pos > 0)
    {
        int counter = 0;
        for (counter = 0; counter< pos; counter++)
           temp[counter] = sql.c_str()[counter];
        temp[counter] = ';';
        temp[counter+1] = '\0';
    }
    if (sqlite3_exec(db, temp, callback, &records, &error_message) !=
SQLITE_OK)
    {
```

Compliant Code

```
        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);
}
```

Compliant Code

```
// 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
```

Compliant Code

```
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
```

Principles(s): The SQL Injection coding standard is mapped to “Sanitize Data Sent to Other Systems”. Ensure all data sent as input is sanitized and cleaned up, otherwise it will cause unexpected behaviors.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	High	**High**	**L2**

Automation

Tool	Version	Checker	Description Tool
Astrée	20.10	**bad-function**	Fully checked
LDRA tool suite	9.7.1	**44 S, 593 S, 594 S**	Partially implemented
Parasoft C/C++test	2020.2	**CERT_C-ERR07-a** **CERT_C-ERR07-b**	The library functions atof, atoi and atol from library stdlib.h shall not be used The Standard Library input/output functions shall not be used
PC-lint Plus	1.4	**586**	Fully supported

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	STD-005-CPP	Do not access freed memory. Pointers to memory that has been deallocated are called *dangling pointers*. Accessing a dangling pointer can result in exploitable vulnerabilities.

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.

```
#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.

```
#include <new>

struct S {
  void f();
};
void g() noexcept(false) {
  S *s = new S;
  // ...
  s->f();
  delete s;
}
```

Principles(s): The best principles that I could think to map would be “Keep it simple” and “Default Deny” for this memory protection coding standard. Usage of pointers not only make programs complicated and if not handled carefully, it results accessing unauthorized memory. Having some authorization [default deny] could provide better protection to memory.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	**Medium**	**L1**

Automation

Tool	Version	Checker	Description Tool
Coverity	v7.5.0	**USE_AFTER_FREE**	Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer
Axivion Bauhaus Suite	6.9.0	**CertC++-MEM50**	Pointer access out of bounds Deallocation of previously deallocated pointer Use of previously freed pointer
Clang	3.9	clang-analyzer-cplusplus.NewDelete clang-analyzer-alpha.security.ArrayBoundV2	Checked by clang-tidy, but does not catch all violations of this rule.
CodeSonar	6.0p0	**ALLOC.UAF**	Use after free

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	STD-006-CPP	Use a static assertion to test the value of a constant expression. Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities.

Noncompliant Code

This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly:

```
#include <assert.h>

struct timer {
  unsigned char MODE;
  unsigned int DATA;
  unsigned int COUNT;
};

int func(void) {
  assert(sizeof(struct timer) == sizeof(unsigned char)
+ sizeof(unsigned int) + sizeof(unsigned int));
}
```

Compliant Code

For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution:

```
struct timer {
  unsigned char MODE;
  unsigned int DATA;
  unsigned int COUNT;
};

#if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned
int) + sizeof(unsigned int)))
  #error "Structure must not have any padding"
#endif
```

Principles(s): Assertion coding standard maps to “Use Effective Quality Assurance Techniques” principle. It will help to improve software quality by uncovering defects in the software.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	High	**High**	**L3**

Automation

Tool	Version	Checker	Description Tool
Clang	3.9	misc-static-assert	Checked by clang-tidy
CodeSonar	6.0p0	**(customization)**	Users can implement a custom check that reports uses of the assert() macro
Compass/ROSE			Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation
ECLAIR	1.2	**CC2.DCL03**	Fully implemented

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	STD-007-CPP	Handle all exceptions. 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.

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
  }
}
```

Principles(s): The coding standard of handling exceptions and having catch all handler in case no matching handler found, is something maps to defense in depth principle in my view. Because, you are kind of putting layers of security. If individual handler can't match then catch all will defend it.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	**High**	**L2**

Automation

Tool	Version	Checker	Description Tool
Parasoft C/C++test	2020.2	**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
Polyspace Bug Finder	R2020a	CERT C++: ERR51-CPP	Checks for unhandled exceptions (rule partially covered)
RuleChecker	20.10	**main-function-catch-all** **early-catch-all**	Partially check
LDRA tool suite	9.7.1	**527 S**	Partially implemented

Coding Standard 8

Coding Standard	Label	Name of Standard
Containers	STD-008-CPP	Use valid iterator ranges. 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.

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; });
}
```

Principles(s): This coding standard of “containers – valid iterator ranges” maps to the “Use Effective Quality Assurance Techniques” principle. With effective testing and code audits, this vulnerability can be uncovered and eliminated.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	High	**High**	**L2**

Automation

Tool	Version	Checker	Description Tool
Astrée	20.10	**overflow_upon_dereference**	
Parasoft C/C++test	2020.2	**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**	
PVS-Studio	7.07	**V539**, **V662**, V789	

Coding Standard 9

Coding Standard	Label	Name of Standard
File Input Output	STD-009-CPP	Reset strings on fgets() or fgetws() failure. It is necessary to reset the string to a known value to avoid errors on subsequent string manipulation functions.

Noncompliant Code

In this noncompliant code example, an error flag is set if fgets() fails. However, buf is not reset and has indeterminate contents:

```
#include <stdio.h>

enum { BUFFER_SIZE = 1024 };
void func(FILE *file) {
  char buf[BUFFER_SIZE];

  if (fgets(buf, sizeof(buf), file) == NULL) {
    /* Set error flag and continue */
  }
}
```

Compliant Code

In this compliant solution, buf is set to an empty string if fgets() fails. The equivalent solution for fgetws() would set buf to an empty wide string.

```
#include <stdio.h>

enum { BUFFER_SIZE = 1024 };

void func(FILE *file) {
  char buf[BUFFER_SIZE];

  if (fgets(buf, sizeof(buf), file) == NULL) {
    /* Set error flag and continue */
    *buf = '\0';
  }
}
```

Principles(s): This coding standard "File Input and Output – reset string on the failure of fgets" maps to "Adopt a Secure Coding Standard" principle. By resetting the strings, unexpected behavior and content can be eliminated.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	**Medium**	**L3**

Automation

Tool	Version	Checker	Description Tool
LDRA tool suite	9.7.1	**44 S**	Enhanced enforcement
Parasoft C/C++test	2020.2	**CERT_C-FIO40-a**	Reset strings on fgets() or fgetws() failure
Polyspace Bug Finder	R2020a	CERT C: Rule FIO40-C	Checks for use of indeterminate string (rule partially covered)
PRQA QA-C++	4.4	**2956**	

Coding Standard 10

Coding Standard	Label	Name of Standard
Integers	STD-010-CPP	Do not cast to an out-of-range enumeration value. To avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression

Noncompliant Code

This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two's complement system, the valid range of values that can be represented by EnumType are [0..3], so if a value outside of that range were passed to f(), the cast to EnumType would result in an unspecified value, and using that value within the if statement results in unspecified behavior.

```
enum EnumType {
  First,
  Second,
  Third
};

void f(int intVar) {
  EnumType enumVar = static_cast<EnumType>(intVar);

  if (enumVar < First || enumVar > Third) {
    // Handle error
  }
}
```

Compliant Code

This compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value.

```
enum EnumType {
  First,
  Second,
  Third
};

void f(int intVar) {
  if (intVar < First || intVar > Third) {
    // Handle error
  }
```

Compliant Code

```
  EnumType enumVar = static_cast<EnumType>(intVar);
}
```

Principles(s): The coding standard “Integer - Do not cast to an out-of-range enumeration value” will map to “Adopt a Secure Coding Standard “ and “Validate Input Data” principles. Going out of range, will cause vulnerabilities.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	**Medium**	**L3**

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	6.9.0	**CertC++-INT50**	
Parasoft C/C++test	2020.2	**CERT_CPP-INT50-a**	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration
PRQA QA-C++	4.4	**3013**	
PVS-Studio	7.07	V1016	

Coding Standard 11

Coding Standard	Label	Name of Standard
Data Structures - Pointers	STD-011-CPP	Always initialize pointers values to nullptr

Noncompliant Code

Below noncompliant example shows that two allocations are attempted within the same try block, and if either fails, the catch handler attempts to free resources that have been allocated. However, because the pointer variables have not been initialized to a known value, a failure to allocate memory for i1 may result in passing ::operator delete() a value (in i2) that was not previously returned by a call to ::operator new(), resulting in undefined behavior.

```
#include <new>

void f() {
  int *i1, *i2;
  try {
    i1 = new int;
    i2 = new int;
  } catch (std::bad_alloc &) {
    delete i1;
    delete i2;
  }
}
```

Compliant Code

This compliant solution initializes both pointer values to nullptr, which is a valid value to pass to ::operator delete()

```
#include <new>

void f() {
  int *i1 = nullptr, *i2 = nullptr;
  try {
    i1 = new int;
    i2 = new int;
  } catch (std::bad_alloc &) {
    delete i1;
    delete i2;
  }
}
```

Principles(s): The coding standard “Always initialize pointers values to nullptr” will map to “Adopt a Secure Coding Standard” principle. Not initializing pointers will cause vulnerabilities.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	**High**	**L1**

Automation

Tool	Version	Checker	Description Tool
Polyspace Bug Finder	R2020a		Checks for: • Invalid deletion of pointer • Invalid free of pointer • Deallocation of previously deallocated pointer
CodeSonar	6.0p0	**ALLOC.FNH** **ALLOC.DF** **ALLOC.TM**	Free non-heap variable Double free Type mismatch
SonarQube C/C++ Plugin	4.10	**S1232**	

Coding Standard 12

Coding Standard	Label	Name of Standard
Data Structures - Arrays	STD-012-CPP	Do not use pointer arithmetic on polymorphic objects. Pointer arithmetic does not account for polymorphic object sizes, and attempting to perform pointer arithmetic on a polymorphic object value results in undefined behavior.

Noncompliant Code

In this noncompliant code example, the for loop uses array subscripting. Since array subscripts are computed using pointer arithmetic, this code results in undefined behavior.

```
#include <iostream>

// ... definitions for S, T, globI, globD ...

void f(const S *someSes, std::size_t count) {
  for (std::size_t i = 0; i < count; ++i) {
    std::cout << someSes[i].i << std::endl;
  }
}

int main() {
  T test[5];
  f(test, 5);
}
```

Compliant Code

Instead of having an array of objects, an array of pointers solves the problem of the objects being of different sizes, as in this compliant solution.

```
#include <iostream>

// ... definitions for S, T, globI, globD ...

void f(const S * const *someSes, std::size_t count) {
  for (const S * const *end = someSes + count; someSes != end; ++someSes) {
    std::cout << (*someSes)->i << std::endl;
  }
}
```

Compliant Code

```
int main() {
  S *test[] = {new T, new T, new T, new T, new T};
  f(test, 5);
  for (auto v : test) {
    delete v;
  }
}
```

Principles(s): This coding standard “Do not use pointer arithmetic on polymorphic objects” maps to “Adopt a Secure Coding Standard” principle. By not using arithmetic on polymorphic objects, unexpected behavior and vulnerabilities can be eliminated.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	**Medium**	**L2**

Automation

Tool	Version	Checker	Description Tool
LDRA tool suite	9.7.1	**44 S**	Enhanced enforcement
Parasoft C/C++test	2020.2	**CERT_C-FIO40-a**	Reset strings on fgets() or fgetws() failure
Polyspace Bug Finder	R2020a	CERT C: Rule FIO40-C	Checks for use of indeterminate string (rule partially covered)
PRQA QA-C++	4.4	**2956**	

Coding Standard 13

Coding Standard	Label	Name of Standard
Data Structures	STD-013-CPP	Pair the memory allocate and deallocate functions correctly

Noncompliant Code

In the following noncompliant code example, an array is allocated with array new[] but is deallocated with a scalar delete call instead of an array delete[] call, resulting in undefined behavior.

```
void f() {
  int *array = new int[10];
  // ...
  delete array;
}
```

Compliant Code

In the compliant solution, the code is fixed by replacing the call to delete with a call to delete [] to adhere to the correct pairing of memory allocation and deallocation functions.

```
void f() {
  int *array = new int[10];
  // ...
  delete[] array;
}
```

Principles(s): The coding standard “Pair the memory allocate and deallocate functions correctly” will map to “Adopt a Secure Coding Standard “principle. The scalar delete will cause vulnerabilities.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	**Medium**	**L1**

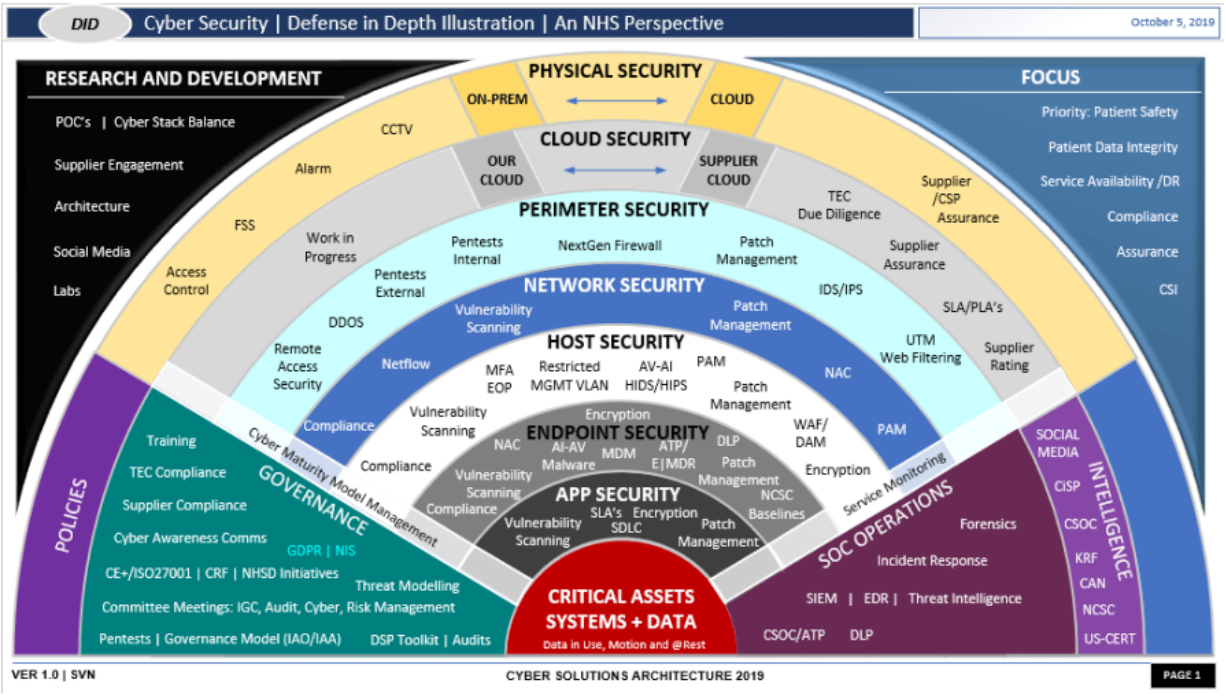
Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	6.9.0	**CertC++-MEM51**	

Tool	Version	Checker	Description Tool
Polyspace Bug Finder	R2020a	CERT C++: MEM51-CPP	Checks for: • Invalid deletion of pointer • Invalid free of pointer • Deallocation of previously deallocated pointer Rule partially covered.
PRQA QA-C++	4.4	3013	
Clang	3.9	clang-analyzer-cplusplus.NewDeleteLeaks -Wmismatched-new-delete clang-analyzer-unix.MismatchedDeallocator	Checked by clang-tidy, but does not catch all violations of this rule

Defense-in-Depth Illustration

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



Project One

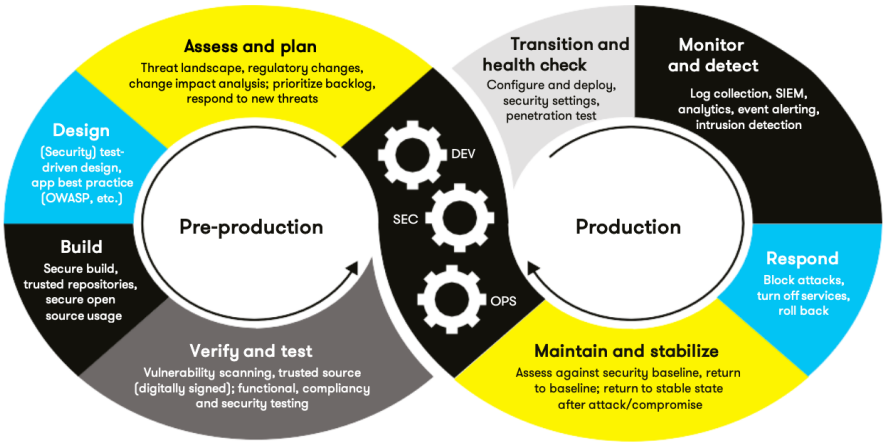
There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

1. **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.

2. **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.

3. **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.

4. **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.

[Insert your written explanations here.]

5. **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	Probable	Medium	**Medium**	**L1**
STD-002-CPP	Medium	Probable	Medium	**Medium**	**L2**
STD-003-CPP	Medium	Unlikely	Medium	**Low**	**L3**
STD-004-CPP	High	Probable	High	**High**	**L2**
STD-005-CPP	High	Likely	Medium	**Medium**	**L1**
STD-006-CPP	Low	Unlikely	High	**High**	**L3**
STD-007-CPP	Medium	Probable	Medium	**High**	**L2**
STD-008-CPP	High	Probable	High	**High**	**L2**
STD-009-CPP	Medium	Probable	Medium	**Medium**	**L3**
STD-010-CPP	Medium	Unlikely	Medium	**Medium**	**L3**
STD-011-CPP	High	Likely	Medium	**High**	**L1**
STD-012-CPP	High	Likely	High	**Medium**	**L2**
STD-013-CPP	High	Likely	Medium	**Medium**	**L1**
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

6. **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**.**
   a. Explain each type of encryption, how it is used, and why and when the policy applies.
   b. 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.
Note: Refered the website: sealpath.com/protecting-the-three-states-of-data/

a. Encryption	**It is one of the key aspects of the IT security. Without, encryption, sensitive business information would be vulnerable to anyone who could steal or interpret it. Data can be encrypted in one of the three states: at rest, in use and in transit.**
Encryption in rest	Protects your data where it's stored—on your computer, in your phone, on your data database, or in the cloud. For example, you saved a copy of a paid invoice on your server with a customer's credit card information. You definitely don't want that to fall into the wrong hands. By encrypting data at rest, you're essentially converting your customer's sensitive data into another form of data. This usually happens through an algorithm that can't be understood by a user who does not have an encryption key to decode it. Only authorized personnel will have access to these files, thus ensuring that your data stays secure
Encryption at flight	Protects your data as it moves from one location to another, as when you send an email, browse the Internet, or upload documents to the cloud. For example, an app on a mobile phone connects to a banking service to request a transaction. Such data is typically encrypted using protocols such as HTTPs.
Encryption in use	Protects your data when it is opened by one or more applications for its treatment or and consumed or accessed by users. Normally, behind the application there is a user who wants to access the data to view it, change it, etc. In this state, the data is more vulnerable, in the sense that in order to see it, the user must have been able to access the content decrypted (in the case that it was encrypted). To protect the data in use, controls should normally be put in place "before" accessing the content. For example, through h: Identity management tools: To check that the user trying to access the data is who he says he is and there has been no identity theft. In these cases it is increasingly important to protect access to the data through a two-factor authentication.

Referred website: https://www.techopedia.com/definition/24130/authentication-authorization-and-accounting-aaa

b. Triple-A Framework*	**Explain what it is and how and why the policy applies.**
Authentication	Authentication refers to unique identifying information from each system user, generally in the form of a username and password. System administrators monitor and add or delete authorized users from the system.
Authorization	Authorization refers to the process of adding or denying individual user access to a computer network and its resources. Users may be given different authorization levels that limit their access to the network and associated resources. Authorization determination may be based on geographical location restrictions, date or time-of-day restrictions, frequency of logins or

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
	multiple logins by single individuals or entities. Other associated types of authorization service include route assignments, IP address filtering, bandwidth traffic management and encryption.
Accounting	Accounting refers to the record-keeping and tracking of user activities on a computer network. For a given time period this may include, but is not limited to, real-time accounting of time spent accessing the network, the network services employed or accessed, capacity and trend analysis, network cost allocations, billing data, login data for user authentication and authorization, and the data or data amount accessed or transferred.

*****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

7. **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

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	Aparna Pyneni	
2.0	03/10/2020	Project One – updated threats and automation	Aparna Pyneni	
3.0	05/29/2021	Addition of secure coding standards for data sturcures	Aparna Pyneni	

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV