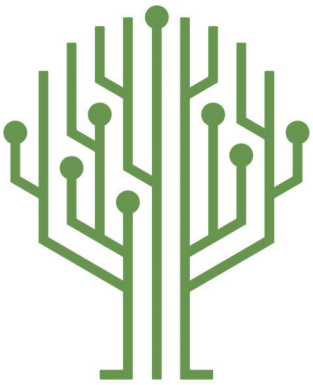


Green Pace Secure Development Policy

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	Accepting raw input, especially from users, can both allow malicious activity and instigate undefined behavior of applications. Therefore, input should be considered untrusted until it can be validated. Input should at least be checked for appropriate typing, value ranges, and unnecessary special characters. Input should "makes sense" for the intended use.
2. Heed Compiler Warnings	Warnings should be addressed, not suppressed. Warnings do not prevent code from compiling but may indicate problems that could lead to insecure code. For example, a warning about an uninitialized variable may be an indicator that the program will produce incorrect or indeterminate results leading to undefined behavior, or even leak information.
3. Architect and Design for Security Policies	Code with security policies in mind. Bake them into the program rather than relying on even a trusted user to comply. A password policy should be enforced programmatically, perhaps using regular expressions, in a trusted environment, i.e.: server-side instead of client-side.
4. Keep It Simple	Overly complex code is difficult to read and maintain. Even if it is secure when written, misunderstandings can introduce security errors upon updating or refactoring. More components increase the attack surface and amount of hardening required to secure an application.
5. Default Deny	Rather than specify specific instances that should not be allowed, only allow access to resources by users or applications that have express permission. This is akin to the concept of whitelisting; anything not included is excluded.
6. Adhere to the Principle of Least Privilege	Do not use higher privileges than necessary to complete tasks. If elevated privilege is required, only allow it for the duration of the task, then switch back to lower privileges. This is a key part of a good defense in depth strategy- if a security flaw allows a user to break out of an application and run arbitrary code, they will have less access as a lower-privileged user.

Principles	Write a short paragraph explaining each of the 10 principles of security.
7. Sanitize Data Sent to Other Systems	Much like input validation, output sanitization can protect downstream systems from malicious or unexpected data. Sanitized data can also prevent information leaks; overly verbose error messages may reveal back-end application information that can be used to hone attacks.
8. Practice Defense in Depth	Do not rely solely on one method of security. Using the principle of least privilege may not prevent a user from exploiting a buffer overflow, but it can reduce the amount of harm done if a user does gain unauthorized access to the underlying system. By layering security measures on top of each other, an attacker can be stopped or slowed down to the point of detection.
9. Use Effective Quality Assurance Techniques	Thorough code review and testing can identify many common programming and security flaws. Separating QA from development roles allows testers to focus on what a program can do, rather than what it should do.
10. Adopt a Secure Coding Standard	Standards can make programming decisions easier to make (if not follow). By establishing rules and methodologies for solving common security problems, a programmer can apply proven solutions to security challenges. Secure coding standards provide benchmarks against which to measure coding practices.

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	Use explicit, C++ casting conventions when converting data types
Data Type	[STD-001-CPP]	C-style casting uses parentheses, which may cause them to be overlooked or confused with the call operator. Information or signed-ness may be lost during implicit conversion. Explicit casting tells the compiler and the reviewer that the cast is intentional and is less likely to result in unexpected values.

Noncompliant Code

This code uses an implicit cast to convert a double to an int, then uses c-style casting to explicitly convert the double to an int. The implicit cast generates a warning while the explicit cast does not. The explicit cast looks similar to a function call, which may lead to misunderstood code thus unexpected values.

```
void main(void) {

     double d = 3.141519;

     int i = d;

     int j = int(d);

}
```

Compliant Code

This code does not generate an error, but it is much clearer that the casting is intentional

```
void main(void) {

     double d = 3.141519;

     int i = static_cast<int>(d);

}
```

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

Principles(s): 2. Heed compiler warnings. A compiler will issue a warning when it detects an unsafe conversion. To ensure conversions are safe, use explicit casting and address any warnings the compiler issues.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	P8	L2

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	BUILD.WALL BUILD.WERROR	Not all warnings are enabled Warnings not treated as errors
PVS-Studio	7.33	V665	[Insert text.]
SonarQube C/C++ Plugin	3.11	S1762 S973	Warns when the default specifier is used with #pragma warning Requires documentation of #pragma uses

Coding Standard 2

Coding Standard	Label	Ensure data values do not wrap around
Data Value	[STD-002-CPP]	Numerical data types and their operations must ensure that their expected values do not "wrap-around". For example, when approaching an unsigned int's max value, adding a sufficiently large value to it will cause the result to "wrap around" and represent a value lower than the expected sum.

Noncompliant Code

Adding one to the max value of an integer causes the result to wrap around and become the minimum value.

```
int a = std::numeric_limits<int>::max();
int b = a + 1;  // b == std::numeric_limits<int>::min()
```

Compliant Code

This code ensures that the difference between the max value of the data type and the first operand is greater than or equal to the value of the second operand which will prevent wraparound.

```
int i1 = std::numeric_limits<int>::max();
int i2 = 1;
int sum;
if (std::numeric_limits<int>::max() - i1 < i2) {
      // handle error
}
else {
      sum = i1 + i2;
}
```

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

Principles(s): 1. Validate Input Data. Validating input data can ensure that variable types are large enough to hold expected inputs and handle conditions where data may be truncated or misrepresented.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	P9	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	24.04	**integer-overflow**	Fully checked

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	7.2.0	**CertC-INT30**	Implemented
CodeSonar	8.1p0	**ALLOC.SIZE.ADDOFLOW** **ALLOC.SIZE.IOFLOW** **ALLOC.SIZE.MULOFLOW** **ALLOC.SIZE.SUBUFLOW** **MISC.MEM.SIZE.ADDOFLOW** **MISC.MEM.SIZE.BAD** **MISC.MEM.SIZE.MULOFLOW** **MISC.MEM.SIZE.SUBUFLOW**	Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size
Compass/ROSE			Can detect violations of this rule by ensuring that operations are checked for overflow before being performed (Be mindful of exception INT30-EX2 because it excuses many operations from requiring validation, including all the operations that would validate a potentially dangerous operation. For instance, adding two `unsigned int`s together requires validation involving subtracting one of the numbers from `UINT_MAX`, which itself requires no validation because it cannot wrap.)

Coding Standard 3

Coding Standard	Label	Ensure access values are within the range of the string length
String Correctness	[STD-003-CPP]	Accessing out-of-range values of a string can lead to memory leaks and unexpected behavior. Check indices from untrusted sources before using.

Noncompliant Code

Because foo does not control the value of pos, which is out of range, this code throws an error.

```
void foo(int pos) {

      std::string s("Hello World");
      s[pos] = 'X';

      std::cout << s << std::endl;
}

void main(void) {
      foo(11);
}
```

Compliant Code

The value of pos is checked against the size of the string to ensure it is within bounds.

```
void foo(int pos) {

      std::string s("Hello World");

      if (pos >= 0 && pos <= s.size() - 1) {
            s[pos] = 'X';
      }
      else {
            //handle exception
      }
      std::cout << s << std::endl;
}

void main(void) {
      foo(11);
}
```

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

Principles(s): 1. Validate input data. Validating string access values are within range prevents the unauthorized disclosure of information and unexpected program behavior.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	24.04		Supported Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data.
CodeSonar	8.1p0	**LANG.MEM.BO** **LANG.MEM.TO** **MISC.MEM.NTERM** **BADFUNC.BO.***	Buffer overrun Type overrun No space for null terminator A collection of warning classes that report uses of library functions prone to internal buffer overflows
Compass/ROSE			Can detect violations of the rule. However, it is unable to handle cases involving `strcpy_s()` or manual string copies such as the one in the first example
Polyspace Bug Finder	R2024a	CERT C: Rule STR31-C	Checks for: • Use of dangerous standard function • Missing null in string array • Buffer overflow from incorrect string format specifier • Destination buffer overflow in string manipulation • Insufficient destination buffer size Rule partially covered.

Coding Standard 4

Coding Standard	Label	Parameterize user input using prepared statements
SQL Injection	[STD-004-CPP]	Parameterized queries separate SQL command logic from conditional values. This prevents the query from executing arbitrary commands and instead treats user input as a separate value.

Noncompliant Code

In this example, the user has control over the username variable. It is directly concatenated onto the sql query, and is subsequently vulnerable to SQL injection. For example, if the value passed to username was "'Fred' or 1=1" the resulting SQL query would become SELECT * FROM USERS WHERE NAME = 'Fred' or 1=1, which would evaluate to True and return every column for each user in the database.

```
bool query_user(sqlite3* db, std::string& username) {
     char* error;
     std::string sql = "SELECT * FROM USERS WHERE NAME = ";
     sql += username;
     if(sqlite3_exec(db, sql.c_str(), callback, &data, &error) !=
     SQLITE_OK)
     {
          std::cout << "Query failed.  ERROR: " << error <<
          std::endl;
          sqlite3_free(error);
          return false;
     }
     return true;
}
```

Compliant Code

The following code uses a prepared statement to ensure untrusted data username is entered into the query as a complete value. If the username input contained an attempted SQL injection, the resulting SQL query would treat it as a conditional value: SELECT * FROM USERS WHERE NAME = 'Fred or 1=1' and not interpret the malicious code as an SQL command.

```
bool query_user(sqlite3* db, std::string& username) {
     int rc;
     sqlite3_stmt* stmt;
     std::string query = "SELECT * FROM USERS WHERE NAME = ?";

     if (sqlite3_prepare_v2(db, query.c_str(), -1 &stmt, nullptr) !=
     SQLITE_OK) {
          std::cerr << "SQL error: " << sqlite3_errmsg(db) <<
          std::endl;
          return false;
```

Compliant Code

```
}
if (sqlite3_bind_text(stmt, 1, username.c_str(), -1,
SQLITE_STATIC) != SQLITE_OK) {
     std::cerr << "Cannot bind parameter: " <<
     sqlite3_errmsg(db) << std::endl;
     return false;
}
while (sqlite3_step(stmt) != SQLITE_DONE) {
// do something with record
}
return true;
```

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

Principles(s): 1. Validate Input Data; 5. Default Deny. Validating input ensures database commands cannot be executed through query terms. Denying this behavior by default helps make sure only authorized queries are executed.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)
Coverity	7.5	**SQLI** **FB.SQL_PREPARED_STATEMENT_GENERATED_** **FB.SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE**	Implemented
SonarQube	9.9	**S2077** **S3649**	Executing SQL queries is security-sensitive SQL queries should not be vulnerable to injection attacks
SpotBugs	4.6.0	**SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE** **SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING**	Implemented

Coding Standard 5

Coding Standard	Label	Do not use C standard free() to deallocate objects allocated with new
Memory Protection	[STD-005-CPP]	free() does not invoke an object's destructor. This may result in memory leaks or resource locks. Use delete instead.

Noncompliant Code

Pointer is freed but destructor is not called. Any memory allocated by the class is leaked.

```
MyClass* myCls = new MyClass[5];
free(myCls);
```

Compliant Code

The destructor of the class is called, and any memory allocated by the class is deleted.

```
MyClass* myCls = new MyClass[5];
delete[](myCls);
```

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

Principles(s): 2. Heed Compiler Warnings; 3. Architect and Design for Security Policies. Compiler warnings that catch this sort of violation are important to note because addressing them can avoid problems like freeing a null pointer or a double free error. By designing code with these policies in mind, these types of errors can be avoided.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	**invalid_dynamic_memory_allocation** **dangling_pointer_use**	[Insert text.]
Axivion Bauhaus Suite	7.2.0	**CertC++-MEM51**	[Insert text.]
CodeSonar	8.1p0	**ALLOC.FNH** **ALLOC.DF** **ALLOC.TM** **ALLOC.LEAK**	Free non-heap variable Double free Type mismatch Leak

Tool	Version	Checker	Description Tool
Parasoft C/C++test	2023.1	**CERT_CPP-MEM51-a** **CERT_CPP-MEM51-b** **CERT_CPP-MEM51-c** **CERT_CPP-MEM51-d**	Use the same form in corresponding calls to new/malloc and delete/free Always provide empty brackets ([]) for delete when deallocating arrays Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor Properly deallocate dynamically allocated resources

Coding Standard 6

Coding Standard	Label	Do not use assertions to validate input
Assertions	[STD-006-CPP]	Assertions should primarily be used for debugging as they can be turned off for release builds. An assertion that fails noticeably during debugging may not be obvious upon release.

Noncompliant Code

This code will fail the assertion if the user does not enter any arguments at runtime. However, if assertions are turned off, the code will pass the assertion line and attempt to assign argv[1] to s, potentially causing an access violation.

```
int main(int argc, char* argv[])
{
      assert(argc > 1);
      std::string s = argv[1];
      return 0;
}
```

Compliant Code

This code does not rely on the assertion to verify an argument was passed at runtime.

```
int main(int argc, char* argv[])
{
      assert(argc > 1);
      if (argc > 1) {
            std::string s = argv[1];
      }
      return 0;
}
```

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

Principles(s): 1. Validate input data. By not relying on assertions for data validation, release code is more robust.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	P8	L2

Automation

Tool	Version	Checker	Description Tool
Parasoft C/C++test	2024.1	**CERT.MSC60.ASSERT**	Do not use assertions in production code

Coding Standard 7

Coding Standard	Label	Handle all exceptions
Exceptions	[STD-007-CPP]	By default, if no exception handler is found, std::terminate is called, which calls std::abort(). This cause an abnormal termination of the program, which can be leveraged into a Denial of Service attack. Handling exceptions allows the program to gracefully continue or end.

Noncompliant Code

The following code throws an exception back to the main function where it is not handled, so the program calls std::terminate().

```
void checkLessThan(int i) {

      if (i < std::numeric_limits<unsigned char>::min()) {
            throw std::out_of_range("Value too low");
      }
}

int main(int argc, char* argv[])
{
      checkLessThan(-1);
      return 0;
}
```

Compliant Code

This code handles the exception thrown by the checkLessThan function, and allows the program to continue gracefully.

```
void checkLessThan(int i) {

      if (i < std::numeric_limits<unsigned char>::min()) {
            throw std::out_of_range("Value too low");
      }
}
void main(void)
{
      try {
            checkLessThan(-1);
      }
      catch (std::exception& e) {
            std::cout << "Exception: " << e.what() << std::endl;
            // handle exception
      }
}
```

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

Principles(s): 2. Heed Compiler Warnings; 3. Architect and Design for Security Policies. Unhandled exceptions may cause the program to end in an unknown state. This can cause anything from memory leaks to unauthorized information access.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probabel	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	**LANG.STRUCT.UCTCH**	Unreachable Catch
Parasoft C/C++test	2023.1	**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	R2024a	CERT C++: ERR51-CPP	Checks for unhandled exceptions (rule partially covered)
RuleChecker	22.10	**main-function-catch-all** **early-catch-all**	Partially checked

Coding Standard 8

Coding Standard	Label	Make declarations unambiguous
Declarations	[STD-008-CPP]	Sometimes a statement can be interpreted as either a variable definition or function declaration. When this ambiguity occurs, the compiler will treat it as a function declaration.

Noncompliant Code

The intention of this code is to declare an int i and initialize it with the value of d cast to an int. However, the compiler treats this as the declaration of a function i that takes an int d and returns an integer value.

```
double d = 1.23;
int i(int(d));
```

Compliant Code

Using a c-style cast ensures this creates a variable i that is initialized to d cast as an integer.

```
int i((int)d);
```

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

Principles(s): 3. Architect and Design for Secuirty Policies; 4. Keep it Simple. The simplest declaration is easier to maintain and understand when code is reviewed or refactored.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	**P2**	**L3**

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	**LANG.STRUCT.DECL.FNEST**	Nested Function Declaration
Parasoft C/C++test	2023.1	**CERT_CPP-DCL53-a** **CERT_CPP-DCL53-b** **CERT_CPP-DCL53-c**	Parameter names in function declarations should not be enclosed in parentheses Local variable names in variable declarations should not be enclosed in parentheses

Tool	Version	Checker	Description Tool
			Avoid function declarations that are syntactically ambiguous
Polyspace Bug Finder	R2024a	CERT C++: DCL53-CPP	Checks for declarations that can be confused between: • Function and object declaration • Unnamed object or function parameter declaration Rule fully covered.
SonarQube C/C++ Plugin	4.10	**S3468**	

Coding Standard 9

Coding Standard	Label	Do not assume order of evaluation will ensure side effects are executed in that order
Side Effects	[STD-009-CPP]	When a variable is modified by side effects during the course of one half of an operation, the behavior is undefined when side effects of the variable exist in the other half of the operation.

Noncompliant Code

i is modified by the ++ operator during the course of the assignment. Behavior is undefined.

```
void main (void) {

      int arr[2];
      int i = 0;

      arr[i++] = i;
}
```

Compliant Code

Here it is clear that the ++ operator occurs before the assignment of the value to arr[i].

```
void main (void) {

      int arr[2];
      int i = 0;

      i++;
      arr[i] = i;
}
```

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

Principles(s) 4. Keep it simple. Making explicit the order of operations through the use of separate statements or parenthesis makes code more readable and easier to maintain.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Medium	Probable	Medium	P8

Automation

Tool	Version	Checker	Description Tool
Astrée	24.04	**evaluation-order** **multiple-volatile-accesses**	Fully checked
CodeSonar	8.1p0	**LANG.STRUCT.SE.DEC** **LANG.STRUCT.SE.INC** **LANG.STRUCT.SE.INIT**	Side Effects in Expression with Decrement Side Effects in Expression with Increment Side Effects in Initializer List
Compass/ROSE			Can detect simple violations of this rule. It needs to examine each expression and make sure that no variable is modified twice in the expression. It also must check that no variable is modified once, then read elsewhere, with the single exception that a variable may appear on both the left and right of an assignment operator
GCC	4.3.5		Can detect violations of this rule when the `-Wsequence-point` flag is used

Coding Standard 10

Coding Standard	Label	Close files when they are not needed.
Input Output	[STD-010-CPP]	Open files use resources or may be accessed by untrusted processes.

Noncompliant Code

The file is never closed. Another process may not be able to use the file.

```
#include <fstream>

int main (void) {
      std::fstream file("file.txt");
      //Do something with file

      return 0;
}
```

Compliant Code

Calling close() ensures the resource is properly closed.

```
#include <fstream>

int main (void) {

      std::fstream file("file.txt");
      //Do something with file
      file.close();

      return 0;
}
```

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

Principles(s): 3. Architect and Design for Security Policies; 7. Sanitize Data Sent to other systems. Open files may allow unauthorized processes to access them, or deny access to authorized processes.

Threat Level

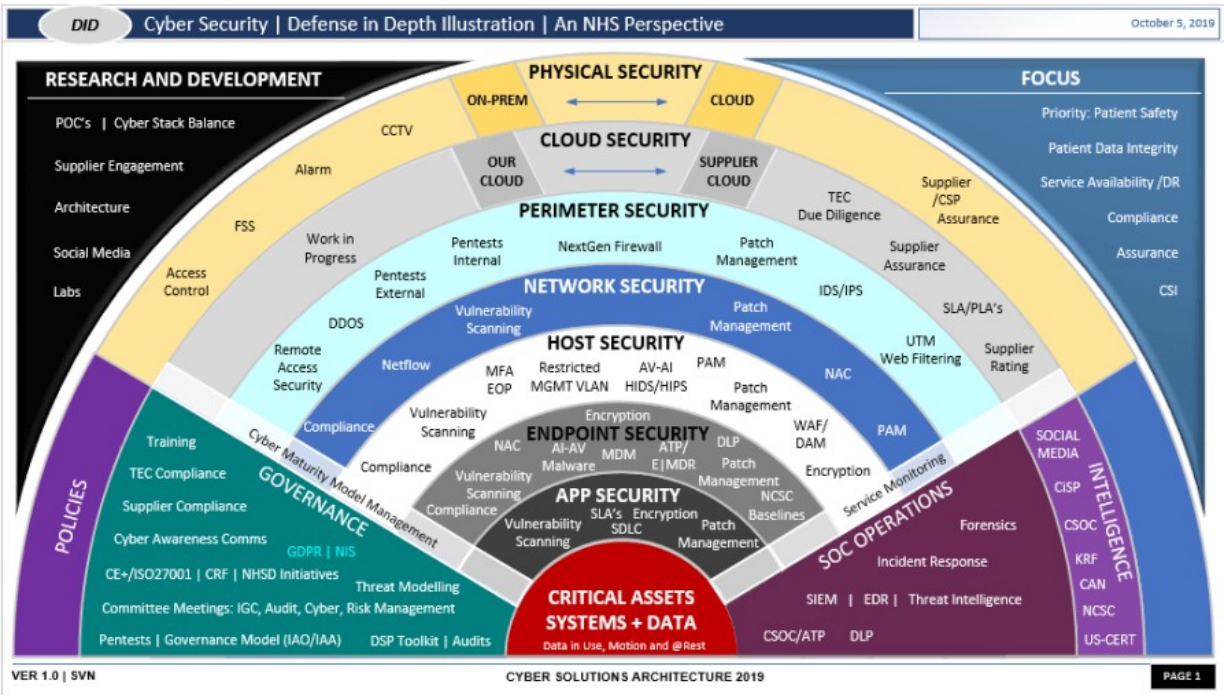
Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	**ALLOC.LEAK**	Leak
Parasoft C/C++test	2023.1	**CERT_C-FIO42-a**	Ensure resources are freed
Polyspace Bug Finder	R2024a	CERT C: Rule FIO42-C	Checks for resource leak (rule partially covered)
SonarQube C/C++ Plugin	3.11	**S2095**	

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

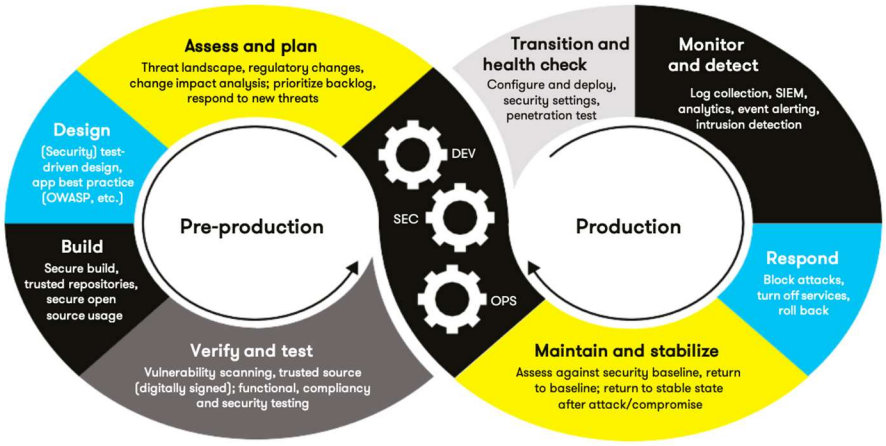
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

Automation can be used at several stages of the current DevOps process both in pre-production and production. During the Assess and Plan and Design portions, automation should be considered alongside the design of the program or system. Researching and evaluating the types of tools available to help ensure security requirements are met should be done as soon as the requirements are established. During the Build and Verify and Test portions of the DevOps process, automated tests can be written with each portion of code as necessary. This also helps ensure that changes to the code still result in correct behavior.

The push from pre-production to production environments can be enhanced by Continuous Integration (CI) and Continuous Deployment (CD) methods. CI can automate building, testing and regression testing of products and provide critiques to developers. CD can automatically push changes once they've passed testing. In the production environment, log collection and event alerting are best automated because of the sheer volume of data that can be gathered. Some responses, like cutting off access to an IP address that is trying to login with many different usernames can be automated as well.

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]	Medium	Probable	Medium	P8	L2
[STD-002-CPP]	High	Likely	High	P9	L2
[STD-003-CPP]	High	Likely	Medium	P18	L1
[STD-004-CPP]	High	Likely	Medium	P18	L1
[STD-005-CPP]	High	Likely	Medium	P18	L1
[STD-006-CPP]	Medium	Probable	Medium	P8	L2
[STD-007-CPP]	Low	Probable	Medium	P4	L3
[STD-008-CPP]	Low	Unlikely	Medium	**P2**	**L3**
[STD-009-CPP]	Medium	Probable	Medium	**P8**	**L2**

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
[STD-010-CPP]	Medium	Unlikely	Medium	P4	L3

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	Data that is not currently being used or transmitted is considered “at rest”. Because data could be at rest for a prolonged period of time, a means of encryption that is not susceptible to brute force is best. For mobile devices, full-disk encryption is recommended in case the device is lost or stolen.
Encryption in flight	In flight data is moving from one location to another, be it through an email, file transfer protocol, web upload/download or the like. The best way to secure data in transit is through shared key encryption, most often asymmetrical, which solves the key distribution problem.
Encryption in use	When data is opened by an application or user, it is considered in use. Data in use can be protected by access controls and permissions.

b. Triple-A Framework*	**Explain what it is and how and why the policy applies.**
Authentication	Authentication is identifying a user. Logins with passwords and unique usernames are a form of authentication. The in use policy can be applied here because the user will be accessing various forms of data while they are authenticated.
Authorization	These are the tasks and data the authenticated user is allowed to perform or access. Once the user is authenticated, their authorization policy is applied. Database changes must only be done by authorized users. Levels of access and files accessed are governed by authorization policies.
Accounting	Accounting logs information about a user’s session that can be used to verify authentication and authorization policies are being properly applied. It can also be used for analysis of resource usage to compare against a baseline to look for abnormalities or for design planning. New user creation should be logged and reviewed regularly. By monitoring levels of access and files accessed by users, authorization policies can be checked for compliance and enforcement.

*****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	12/07/2024	Initial Draft	Matt Jackson	[Insert text.]
[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