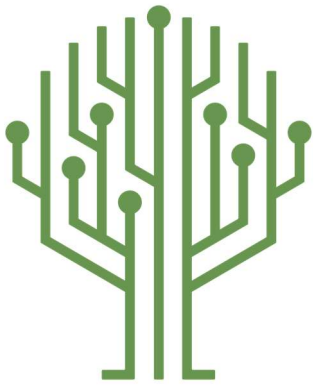


Green Pace Secure Development Policy

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	A security flaw can cause a program to be susceptible to an attack when the input has access to secure parts of the computer system while getting to the program. This can occur when an application is given greater access than the user providing input. This input could also use network services on its way to the application. One way of achieving this attack is buffer overflow. Buffer overflow must be prevented. Buffer overflow needs to be prevented by making sure input data does not exceed the size of the buffer that it is stored in. Integer data must also be checked to make sure that the data lies in the expected range without overflow or underflow.
2. Heed Compiler Warnings	Use the compiler to your advantage for checking code automatically, turn on as many compiler warnings as available. Address any warnings post compilation. Software applications must compile cleanly without any warnings.
3. Architect and Design for Security Policies	Software applications must explicitly apply to all documented security policies. Security must also be decreased by eliminating as many code defects as possible. Addressing defects increases the security of the application.
4. Keep It Simple	There should only be just enough lines of code to perform the desired task or function. Code should not be unnecessarily complex. Always be looking for a simpler way to perform the same task.
5. Default Deny	Access should be denied by default and only granted in instances where it is necessary.
6. Adhere to the Principle of Least Privilege	Applications should not be developed under administrator privileges but should be at the lowest privileges needed for the application to perform a task. Elevated privileges should be limited to the tasks that require it and then immediately revoked.
7. Sanitize Data Sent to Other Systems	If utilizing a subsystem like an off-the-shelf device or 3rd party software like a SQL database, it is the responsibility of the calling application to make sure data sent to these

Principles	Write a short paragraph explaining each of the 10 principles of security.
	subsystems is free of security risks such as SQL injections. The data should be checked pre-invocation.
8. Practice Defense in Depth	Be diverse in defensive strategies deployed in code. The key is to have many layers as part of an application strategy to deal with threats, that way if one strategy is compromised by a threat another layer of protection is in place that is not compromised.
9. Use Effective Quality Assurance Techniques	Having established techniques in place for quality helps eliminate vulnerabilities. Having unit testing in place is a good example, units tests can be performed after each iteration of code ensuring quality is upheld regardless of the author.
10. Adopt a Secure Coding Standard	Develop a standard of secure coding that can be deployed for each new coding application created. Have standards for each programming language and platform used.

C/C++ Ten Coding Standards

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

Coding Standard 1

Coding Standard	Label	Name of Standard
Data Type	[STD-001-CPP]	Do not use floating-point variables as loop counters.

Noncompliant Code

In the example below, a floating point (fp) variable is used as a loop counter index. This results in inaccurate loop counting and may result in 9 or 10 iterations.

```
void func(void) {
  for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
    /* Loop may iterate 9 or 10 times */
  }
}
```

Compliant Code

The following code shows a loop with the loop counter of an integer data type.

```
void func(void) {
  for (size_t count = 1; count <= 10; ++count) {
    float x = count / 10.0f;
    /* Loop iterates exactly 10 times */
  }
}
```

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

Principles(s):
9 — Use Effective Quality Assurance Techniques: Using integers as opposed to floating-point numbers enforces quality assurance because it eliminates unidentified behavior that is caused by using a floating-point number as a loop variable.
10 — Adopt a Secure Coding Standard: Having this standard as a part of the coding standard at the beginning of a project eliminates the use of floating-point number as loop variable, thus eliminating a source of undefined behavior.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Low	P6	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	20.1	for-loop-float	Fully checked

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	6.9.0	CertC-FLP30	Fully implemented
Clang	3.9	cert-flp30-c	Checked by clang-tidy
CodeSonar	6.0p0	LANG.STRUCT.LOOP.FPC	Float-typed loop counter
Compass/ROSE			
Coverity	2017.07	MISRA C 2004 Rule 13.4 MISRA C 2012 Rule 14.1	Implemented
ECLAIR	1.2	CC2.FLP30	Fully implemented
Klocwork	2018	MISRA.FOR.COND.FLT MISRA.FOR.COUNTER.FLT	
LDRA tool suite	9.7.1	39 S	Fully implemented
Parasoft C/C++test	2020.2	CERT_C-FLP30-a	Do not use floating point variables as loop counters
PC-lint Plus	1.4	9009	Fully supported
Polyspace Bug Finder	R2020a	CERT C: Rule FLP30-C	Checks for use of float variable as loop counter (rule fully covered)
PRQA QA-C	9.7	3339, 3340, 3342	Partially implemented
PRQA QA-C++	4.4	4234	
PVS-Studio	7.07	V1034	
RuleChecker	20.1	for-loop-float	Fully checked
SonarQube C/C++ Plugin	3.11	S2193	Fully implemented
TrustInSoft Analyzer	1.38	non-terminating	Exhaustively detects non-terminating statements (see one compliant and one non-compliant example).

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	[STD-002-CPP]	Ensure that unsigned integer operations do produce overflow and underflow.

Noncompliant Code

The code below can produce result in unsigned integer overflow. If this is not the desired effect, the value may be used to allocate memory outside what is expected, resulting in a security vulnerability.

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  /* ... */
}
```

Compliant Code

The code below checks the condition after the value usum is calculated to make sure is not less than the first operand.

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  if (usum < ui_a) {
    /* Handle error */
  }
  /* ... */
}
```

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

Principles(s):
1 — Validate Input Data: A user being able to overflow an integer input value can give access to the secure parts of a computer system.
9 — Use Effective Quality Assurance Techniques: Standardizing the way integer input is taken and tested prior to use in code, helps ensure quality regardless of the developer on the team.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Low	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	20.1	integer-overflow	Fully checked
CodeSonar	6.0p0	ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW	Addition overflow of allocation size Integer overflow of allocation size

Tool	Version	Checker	Description Tool
		ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW	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 ints together requires validation involving subtracting one of the numbers from UINT_MAX, which itself requires no validation because it cannot wrap.)
Coverity	2017.07	INTEGER_OVERFLOW	Implemented
Klocwork	2018	NUM.OVERFLOW	
		CWARN.NOEFFECT.OUTOFRANGE	
LDRA tool suite	9.7.1	493 S, 494 S	Partially implemented
Parasoft C/C++test	2020.2	CERT_C-INT30-a CERT_C-INT30-b CERT_C-INT30-c	Avoid integer overflows Integer overflow or underflow in constant expression in '+', '-', '*' operator Integer overflow or underflow in constant expression in '<<' operator
Polyspace Bug Finder	R2020a	CERT C: Rule INT30-C	Checks for: Unsigned integer overflow Unsigned integer constant overflow Rule partially covered.
PRQA QA-C	9.7	2910 [C], 2911 [D], 2912 [A],	Partially implemented
		2913 [S], 3383, 3384, 3385, 3386	
PRQA QA-C++	4.4	2910, 2911, 2912, 2913	
PVS-Studio	7.07	V658, V1028	
TrustInSoft Analyzer	1.38	unsigned overflow	Exhaustively verified.

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	[STD-003-CPP]	Make sure the storage for strings has enough space for character data and null terminator.

Noncompliant Code

The code below has a fixed buffer of size of 12 but does not limit the width of cin to 12.

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

Compliant Code

The code below utilizes the width member of cin to limit the width to the buffer size of 12.

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

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

Principles(s):
1 — Validate Input Data: A user input of string data should be limited by the width component of the buffer.
9 — Use Effective Quality Assurance Techniques: Standardizing the way string data is taken in and tested prior to use in code, helps ensure quality regardless of the developer on the team.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Low	P27	L1

Automation

Tool	Version	Checker	Description Tool
CodeSonar	6.0p0	MISC.MEM.NTERM LANG.MEM.BO LANG.MEM.TO	No space for null terminator Buffer overrun Type overrun
Klocwork	2018	NNTS.MIGHT NNTS.TAINTED	

Tool	Version	Checker	Description Tool
LDRA tool suite	9.7.1	489 S, 66 X, 70 X, 71 X	Partially implemented
Parasoft C/C++test	2020.2	CERT_CPP-STR50-b CERT_CPP-STR50-c CERT_CPP-STR50-e CERT_CPP-STR50-f CERT_CPP-STR50-g	Avoid overflow due to reading a not zero terminated string Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Do not use the 'char' buffer to store input from 'std::cin'
Polyspace Bug Finder	R2020a	CERT C++: STR50-CPP	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 Rule partially covered.
SonarQube C/C++ Plugin	4.1	S3519	

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	[STD-004-CPP]	Prevent SQL injection.

Noncompliant Code

The user could inject a code tacked onto the end of pwd, like ‘or 1=1’ that would result in a true statement for the password. This could give unauthorized access to the system.

```
String sqlString = "SELECT * FROM db_user WHERE username = '"
                         + username +
                         "' AND password = '" + pwd + "'";
```

Compliant Code

The below code snippet utilizes parameter data for ‘username’ and ‘password,’ the SQL statement is separated from user entered data, therefore injection cannot occur.

```
String sqlString =
      "select * from db_user where username=? and password=?";
    PreparedStatement stmt = connection.prepareStatement(sqlString);
    stmt.setString(1, username);
    stmt.setString(2, pwd);
```

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

Principles(s):
7 — Sanitize Data Sent to Other Systems: Parameterizing query parameters will help prevent SQL injection from uncontrolled data sources.
9 — Use Effective Quality Assurance Techniques: Standardizing the way string data is taken in and tested prior to use in code, helps ensure quality regardless of the developer on the team.
10 — Adopt a Secure Coding Standard: Having a standard for SQL injection will inform all developers of the risk and mitigate the possibility of SQL injection into the software project.

Princip

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Taint mode	n/a	Insecure dependency in parameter \d* of DBI::db=.* method call	Catches SQL injection. Requires TaintIn attribute.

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	[STD-005-CPP]	Do not read uninitialized memory.

Noncompliant Code

In the example below, an uninitialized local variable is being evaluated and printed resulting in undefined behavior.

```
void f() {
  int i;
  std::cout << i;
}
```

Compliant Code

In the safe solution below, the variable is initialized prior to use and printing.

```
void f() {
  int i = 0;
  std::cout << i;
}
```

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

Principles(s):
2 — Not initializing a integer variable leads to a compiler warning, all compiler warnings must be addressed.
9 — Use Effective Quality Assurance Techniques: Initializing integer data prevents unidentified behavior that is caused by using a variable that is not initialized and could have unknown data in that memory location.
10 — Adopt a Secure Coding Standard: Having this standard as a part of the coding standard at the beginning of a project eliminates uninitialized integer variables, thus eliminating a source of undefined behavior.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Low	P3	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	20.1	uninitialized-read	Partially checked
Clang	3.9	-Wuninitialized	Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory.

Tool	Version	Checker	Description Tool
		clang-analyzer-core.UndefinedBinaryOperatorResult	
CodeSonar	6.0p0	LANG.STRUCT.RPL	Return pointer to local
		LANG.MEM.UVAR	Uninitialized variable
Klocwork	2018	UNINIT.CTOR.MIGHT UNINIT.CTOR.MUST UNINIT.HEAP.MIGHT UNINIT.HEAP.MUST UNINIT.STACK.ARRAY.MIGHT UNINIT.STACK.ARRAY.MUST UNINIT.STACK.ARRAY.PARTIAL.MUST UNINIT.STACK.MIGHT UNINIT.STACK.MUST	
LDRA tool suite	9.7.1	53 D, 69 D, 631 S, 652 S	Partially implemented
Parasoft C/C++test	2020.2	CERT_CPP-EXP53-a	Avoid use before initialization
Parasoft Insure++			Runtime detection
Polyspace Bug Finder	R2020a	CERT C++: EXP53-CPP	Checks for: Non-initialized variable Non-initialized pointer Rule partially covered.
PRQA QA-C++	4.4	2726, 2727, 2728, 2961, 2962, 2963, 2966, 2967, 2968, 2971, 2972, 2973, 2976, 2977, 2978	
PVS-Studio	7.07	V546, V573, V614, V670, V679, V730, V788, V1007, V1050	
RuleChecker	20.1	uninitialized-read	Partially checked

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.

Noncompliant Code

The code below uses the assert macro to assert a property for a memory-mapped structure.

```
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

The code below uses the static assert.

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

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

Principles(s):
9 — Use Effective Quality Assurance Techniques: Standardizing the use of static asserts as opposed to regular asserts for testing of constants ensure proper unit testing of constants. This helps ensure quality regardless of the developer on the team.
10 — Adopt a Secure Coding Standard: Having a standard for the use of assertions in unit testing for all developers helps ensure code is being tested properly.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	P2	L3

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	6.9.0	CertC-DCL03	
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

Coding Standard 7

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

Noncompliant Code

The code below uses throwing_func() but neither f() nor main() catch exceptions thrown by throwing_func(). In the event of an exception the program will terminate with std::terminate().

```
void throwing_func() noexcept(false);

void f() {
  throwing_func();
}

int main() {
  f();
}
```

Compliant Code

In the compliant code below, main() is handling all exceptions, upon an exception the stack is unwound up to main(), this allows for gracefully managing external resources.

```
void throwing_func() noexcept(false);

void f() {
  throwing_func();
}

int main() {
  try {
    f();
  } catch (...) {
    // Handle error
  }
}
```

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

Principles(s):
3 — Architect and Design for Security Policies: Handling exceptions in the proper way contributes to the adherence to security policies.
9 — Use Effective Quality Assurance Techniques: Adopting a standard way of handling exceptions promotes quality.
10 — Adopt a Secure Coding Standard: Having a standard on how and when to deploy exception handling will ensure that every developer produces similar results when it comes to exception handling.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	P9	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	20.1	main-function-catch-all early-catch-all	Partially checked
Axivion Bauhaus Suite	6.9.0	CertC++-ERR51	
LDRA tool suite	9.7.1	527 S	Partially implemented
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)
PRQA QA-C++	4.4	4035, 4036, 4037	
RuleChecker	20.1	main-function-catch-all early-catch-all	Partially checked

Coding Standard 8

Coding Standard	Label	Name of Standard
String Correctness	[STD-008-CPP]	Range check element access.

Noncompliant Code

In the code below, the returned value by the call to get_index() could be greater than the number of elements stored in the string which triggers undefined behavior.

```
extern std::size_t get_index();

void f() {
  std::string s("01234567");
  s[get_index()] = '1';
}
```

Compliant Code

The compliant code below uses std::basic_string::at(), which behaves in a similar fashion to the index operator[] but throws the std::out_of_range exception if pos >= size().

```
extern std::size_t get_index();

void f() {
  std::string s("01234567");
  try {
    s.at(get_index()) = '1';
  } catch (std::out_of_range &) {
    // Handle error
  }
}
```

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

Principles(s):
2 — Heed Compiler Warnings: Referencing an index outside a buffer array will yield compiler warnings, all compiler warnings must be addressed
9 — Use Effective Quality Assurance Techniques: Verifying that all references to buffer arrays are inside the size of the buffer array will help with quality assurance of the project code.
10 — Adopt a Secure Coding Standard: Having a standard for referencing buffer arrays and testing the size of buffer arrays will help ensure every developer handle buffer array in the same manner.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	20.1	assert_failure	
CodeSonar	6.0p0	LANG.MEM.BO LANG.MEM.BU LANG.MEM.TBA LANG.MEM.TO LANG.MEM.TU	Buffer overrun Buffer underrun Tainted buffer access Type overrun Type underrun
Parasoft C/C++test	2020.2	CERT_CPP-STR53-a	Guarantee that container indices are within the valid range
Polyspace Bug Finder	R2020a	CERT C++: STR53-CPP	Checks for: Array access out of bounds Array access with tainted index Pointer dereference with tainted offset Rule partially covered.

Coding Standard 9

Coding Standard	Label	Name of Standard
Input Output	[STD-009-CPP]	Close files when they are no longer needed.

Noncompliant Code

In the code below, std::fstream object file is constructed. The open() function is called with the constructor. The function close() is not called before terminate(). The object is not properly closed.

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

Compliant Code

In the code below, std::fstream::close() is called before std::terminate(), this ensures the file resources are properly closed.

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

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

Principles(s):
9 — Use Effective Quality Assurance Techniques: Establishing the proper way to close a file is imperative, resources that are not disposed of properly can result in undefined behaviors.
10 — Adopt a Secure Coding Standard: Having a standard for the way files are closed ensure that every developer on the team is managing resources in the proper way.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Low	P6	L2

Automation

Tool	Version	Checker	Description Tool
CodeSonar	6.0p0	ALLOC.LEAK	Leak
Klocwork	2018	RH.LEAK	
Parasoft C/C++test	2020.2	CERT_CPP-FIO51-a	Ensure resources are freed
Parasoft Insure++			Runtime detection
Polyspace Bug Finder	R2020a	CERT C++: FIO51-CPP	Checks for resource leak (rule partially covered)

Coding Standard 10

Coding Standard	Label	Name of Standard
Miscellaneous	[STD-010-CPP]	Value-returning functions must return a value from all exit paths.

Noncompliant Code

In the code below, there is not a return value for positive values, not all paths produce a return value.

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

Compliant Code

In the code below, all code paths produce a return value.

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

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

Principles(s):
2 — Heed Compiler Warnings: Not returning values on all code paths will result in a compiler warning, all compiler warnings must be addressed.
9 — Use Effective Quality Assurance Techniques: Verifying that a function returns data on all code paths helps alleviate undefined behavior and improves quality.
10 — Adopt a Secure Coding Standard: Having a standard dictating that data must be returned from all code paths helps ensure every developer on the team avoids this error.

Threat Level

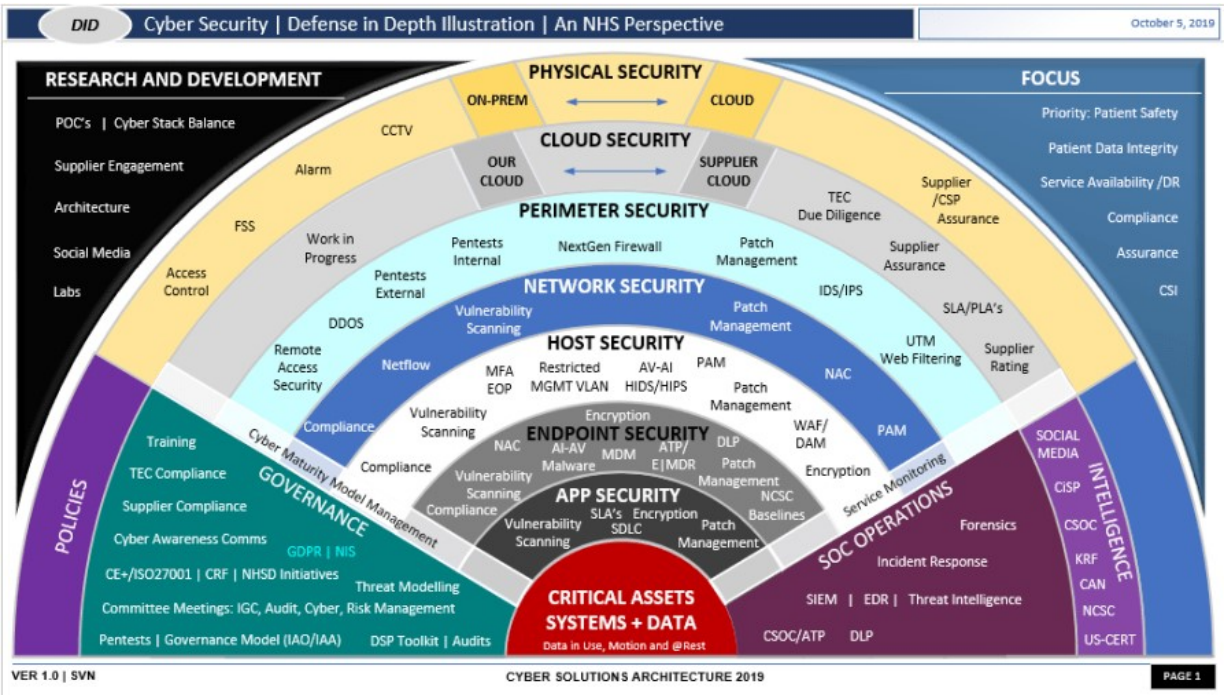
Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	20.1	return-implicit	Fully checked
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
LDRA tool suite	9.7.1	2 D, 36 S	Fully implemented
Parasoft C/C++test	2020.2	CERT_CPP-MSC52-a	All exit paths from a function with non-void return type shall have an explicit return statement with an expression
Polyspace Bug Finder	R2020a	CERT C++: MSC52-CPP	Checks for missing return statements (rule partially covered)
SonarQube C/C++ Plugin	4.1	S935	
PRQA QA-C++	4.4	1510	
PVS-Studio	7.07	V591	
RuleChecker	20.1	return-implicit	Fully checked

Defense-in-Depth Illustration

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



Project One

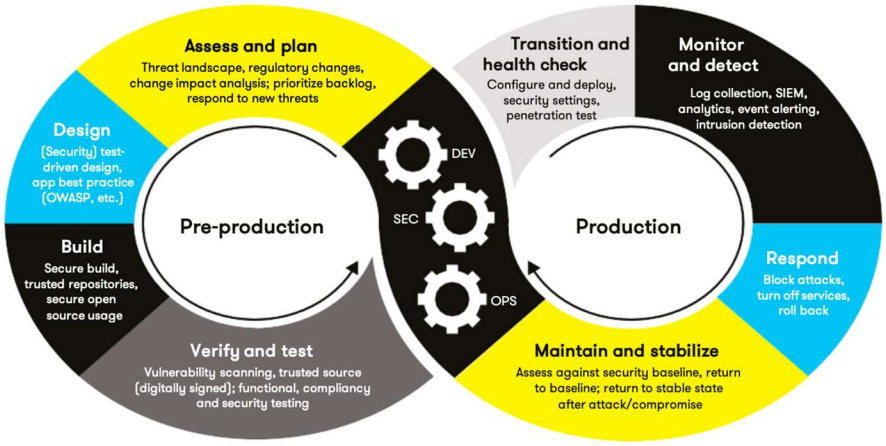
There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

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.

DevSecOps will occur the same frequency as DevOps. For a waterfall software development life cycle, this activity may occur at the end of development. In an iterative software development lifecycle should occur at the end of every iteration and for every commit if the security features are a part of a unit testing program. Automatic security testing should only be ran on code that poses a risk, not on entire project.

The additions to DevOps to support DevSecOps will be unit test written specifically to test for security vulnerability. These tests will run at every commit, nightly, and at a minimum at the end of an iteration or sprint. This is on the development side of DevSecOps. The rules stated above will need one of the forms of automatic static testing broken out in the automation table of each rule.

On the operations side DevOps will need to include integrity checks and defense-in-depth measurements for prevention. For detection, network monitoring and penetration tests will need to be implemented.

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	Medium	Unlikely	Low	P6	L2
STD-002-CPP	High	Probable	Low	P18	L1
STD-003-CPP	High	Likely	Low	P27	L1
STD-004-CPP	High	Likely	Medium	P18	L1
STD-005-CPP	Low	Unlikely	Low	P3	L3
STD-006-CPP	Low	Unlikely	Medium	P2	L3
STD-007-CPP	High	Likely	High	P9	L2
STD-008-CPP	High	Likely	Medium	P18	L1
STD-009-CPP	Medium	Unlikely	Low	P6	L2

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

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.

a. Encryption	**Explain what it is and how and why the policy applies.**
Encryption in rest	Encryption at rest refers to data that sits statically in tables. All data and encryption keys will be stored separately for each other. All sensitive data must be encrypted like passwords and any other data classified as sensitive. (Cairns & Somerfield, 2017)
Encryption at flight	Attackers can retrieve data as it is transmitted to a client, these are referred to as man-in-the-middle attacks. Sensitive data should not be transmitted in plain text, it should be encrypted. (Cairns & Somerfield, 2017)
Encryption in use	An example of encryption in use is password verification and using a hash of the original password data to do comparisons to what the user enters. (Cairns & Somerfield, 2017)

b. Triple-A Framework*	**Explain what it is and how and why the policy applies.**
Authentication	Authentication covers identifying a user and verifying who they are. Authentication ensures that only approved users gain access to a network and computer system. This is crucial to security. (O'Carroll, 2018)
Authorization	After a user is authenticated, authorization determines what parts of the computer program, system, or network the user is permitted to access. Authentication also controls what actions users are permitted to take. A user should never be given more access than they need. (O'Carroll, 2018)
Accounting	Accounting tracks the activity of users, what activity occurred when and by who in the form of a log. In the event of an attack or security event, having proper accounting will help system administrators perform a post-mortem on the security event to analyze what happened. This will determine what counter-measure activity needs to occur. (O'Carroll, 2018)

*****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 is 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 always follow this policy.

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	04/10/2021	Completed Template	Richard Schall	
[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

Reference:

Cairns, C. & Somerfield, D. (2017, January 05). The basics of web application security. Retrieved April 10, 2021, from https://martinfowler.com/articles/web-security-basics.html

CERT. (2020). Confluence. Retrieved March 19, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

O'Carroll, B. (2018, November 27). What is AAA Security? An introduction to authentication, authorisation and accounting. Retrieved April 10, 2021, from https://codebots.com/application-security/aaa-security-an-introduction-to-authentication-authorisation-accounting