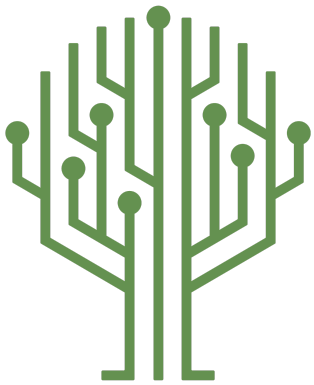


Green Pace

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 preventive option to being attack by malious software, is to test and validate all inputs from untrusted data sources. Should always be aware of external data sources, such as command line arguments, network interfaces, environmental variables, and user controlled files.
2. Heed Compiler Warnings	Best practice is to utilze the highest warning level when compiling codes. Such warnings exist to notify a developer of a potential error or issue in code. Errors prevents code from compiling while warnings do not, but both are equally important to consider code modification in preventing potential security risks.
3. Architect and Design for Security Policies	Contemplate software architecture and design to ensure that security policies are being enforced and implemented accordingly.
4. Keep It Simple	Keeping design simple and small reduces likelihood of errors in both coding and use. Making a complicated one can increase risks of failed security mechanisms and errors.
5. Default Deny	Access is denied, and access is permitted through the conditions of the protection scheme used by permission only.
6. Adhere to the Principle of Least Privilege	To lower the chance that an attacker has to execute code with elevated privileges, processes should execute with minimal required privileges needed to complete the job, and elevated privileges should be used as minimally as possible and with as little time as needed.
7. Sanitize Data Sent to Other Systems	Sanitizing data before passing the data to other systems checks for potential issues prior to invoking these systems. This will eliminate unused functions, or calls made out of context, which may pass and cause damage, such as SQL injection attacks.

Principles	Write a short paragraph explaining each of the 10 principles of security.
8. Practice Defense in Depth	Ensuring that multiple layers to defense is in place, can mitigate possible exploits or damage should one layer of defense be made vulnerable.
9. Use Effective Quality Assurance Techniques	Proper testing, such as fuzz and penetration testing, as well as audits to code, can be part of an effective QA program. Security reviews, both internal and external, can help identify and correct possible issues.
10. Adopt a Secure Coding Standard	Coding standards should be applied in your language and platform of choice to be secure from the start.

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]	When choosing a data type , unsigned integer values should represent values that can't become negative, signed integer values should be used for values that can become negative. In general, use the smallest signed or unsigned type that can fully represent the range of possible values for the given variable, as this conserves memory.

Noncompliant Cod

Truncation can occur when a value is too small to represent the result, and conversions can result in values out of range in the resulting type.

```
1 unsigned long int ul = ULONG_MAX;
2 signed char sc;
3 sc = (signed char)ul; /* cast eliminates warning */
```

Compliant Code

Validate ranges when converting from an unsigned type to a signed type.

```
1 unsigned long int ul = ULONG_MAX;
2 signed char sc;
3 if (ul <= SCHAR_MAX) {
4 sc = (signed char)ul; /* use cast to eliminate warning */
5 }
6 else {
7 /* handle error condition */
8 }
```

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

Principles(s): 1) Validate Input Data, 2) Heed compiler warnings, 9) Effective Quality assurance techniques. By properly validating we can prevent out of bound memory errors, by heeding compiler warnings we can catch before production, and by using QA techniques we can usee analsysis in the pipeline to catch this kind of error. 3) Architect for design specifically education on this issue would prevent it and ensure the team are using types correctly.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	P9	L2

Automation

Tool	Version	Checker	Description Tool
Coverity	2017.07	TAINTED_SCALAR BAD_SHIFT	Implemented
Helix QAC	2021.2	C2800, C2801, C2802, C2803, C2860, C2861, C2862, C2863 C++2800, C++2801, C++2802, C++2803, C++2860, C++2861, C++2862, C++2863	Implemented
Parasoft C/C++test	2021.1	CERT_C-INT32-a CERT_C-INT32-b CERT_C-INT32-c	Avoid integer overflows Integer overflow or underflow in constant expression in '+', '-', '*' operator Integer overflow or underflow in constant expression in '<<' operator
TrustInSoft Analyzer	1.38	Signed_overflow	Exhaustively verified

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	[STD-002-CPP]	When choosing a data type, unsigned integer values should represent values that can't become negative, signed integer values should be used for values that can become negative. use the smallest signed or unsigned type that can fully represent the range of possible values for the given variable, as this conserves memory.

Noncompliant Code

This code fails to consider that the unsigned integer value will wrap around. (infinite loop)

```
1 char a[MAX_ARRAY_SIZE] = /* initialize */;
2 size_t cnt = /* initialize */;
3
4 for (unsigned int i = cnt-2; i >= 0; i--) {
5 a[i] += a[i+1];
6 }
```

Compliant Code

As size_t is an unsigned type, this behavior is well defined by the standard to be modulo.

```
1 char a[MAX_ARRAY_SIZE] = /* initialize */;
2 size_t cnt = /* initialize */;
3
4 for (size_t i = cnt-2; i != SIZE_MAX; i--) {
5 a[i] += a[i+1];
6 }
```

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

Principles(s): 1) Validate Input Data to catch these mistakes from the onset, and 2) Use Effective Quality Assurance Techniques to mitigate these problems via analysis tools.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	P9	L2

Automation

Tool	Version	Checker	Description Tool
Coverity	2017.07	INTEGER_OVERFLOW	Implemented
CodeSonar	6.1p0	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
Parasoft C/C++test	2021.1	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	R2021a	CERT C: Rule INT30-C	Checks for: Unsigned integer overflow Unsigned integer constant overflow Rule partially covered.

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	[STD-003-CPP]	Incorrect string sizes and neglected buffer boundaries can lead to buffer overflows and runtime errors. Never copy data from some unbounded source such as stdin to a fixed-length array.

Noncompliant Code

C++: If a user inputs more than 11 characters, it will result in an out-of-bounds write

```
1 #include <iostream>
2
3 int main(void) {
4 char buf[12];
5
6 std::cin >> buf;
7 std::cout << "echo: " << buf << '\n';
8 }
```

Compliant Code

Eliminates the overflow in the previous example by setting the field width member to the size of the character array buf.

```
1 #include <iostream>
2
3 int main(void) {
4 char buf[12];
5
6 std::cin.width(12);
7 std::cin >> buf;
8 std::cout << "echo: " << buf << '\n';
9 }
06 string str;
07 string::iterator i;
08 for (i = str.begin(); i != str.end(); ++i) {
09    cout << *i
10 }
```

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

Principles(s): 1) Validate input with appropriate string function, 2) Heed compiler warnings of overflow, 3) Sanitize data to prevent string attacks, and 4) QA Techniques to catch these errors on build in the CI/CD pipeline.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
CodeSonar	6.1p0	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
Coverity	2017.07	STRING_OVERFLOW BUFFER_SIZE OVERRUN STRING_SIZE	Fully implemented
Parasoft C/C++test	2021.1	CERT_C-STR31-a CERT_C-STR31-b CERT_C-STR31-c CERT_C-STR31-d CERT_C-STR31-e	Avoid accessing arrays out of bounds Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Avoid using unsafe string functions which may cause buffer overflows
TrustInSoft Analyzer	1.38	Mem_access	Exhaustively verified

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	[STD-004-CPP]	Attack vectors for strings include command-line arguments, environmental variables, console input, text files, and network connections. These are all vectors in which an attacker can launch strategic SQL injection-based attacks in order to cause overflow. According to Secure Coding in C and C++, “String concatenation is the primary point of entry for script injection.” Test the size / data type of input and enforce appropriate limits, rejecting any entries that contain binary data, escape sequences, and comment characters. Review all code that calls EXECUTE, EXEC, or sp_executesql. Always use Parameterized Queries in SQL.

Noncompliant Code

Unfiltered code is vulnerable to SQL injection through user input

```
SqlDataAdapter myCommand =
new SqlDataAdapter("LoginStoredProcedure '" +
                Login.Text + "'", conn);
```

Compliant Code

Using the Parameters collection with Dynamic SQL:

```
SqlDataAdapter myCommand = new SqlDataAdapter(
"SELECT au_lname, au_fname FROM Authors WHERE au_id = @au_id", conn);
SQLParameter parm = myCommand.SelectCommand.Parameters.Add("@au_id",
                                SqlDbType.VarChar, 11);
Parm.Value = Login.Text;
```

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

Principles(s): 1) Input validation, ensure to use query parameterization, 2) Architect and design for security policies meaning code with knowledge of possible SQL injection attacks in mind, 3) Practice defense in depth, using a layered defensive strategy to prevent such attacks.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
CodeSonar	6.1p0	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
Coverity	2017.07	STRING_OVERFLOW BUFFER_SIZE OVERRUN STRING_SIZE	Fully implemented
Parasoft C/C++test	2021.1	CERT_C-STR31-a CERT_C-STR31-b CERT_C-STR31-c CERT_C-STR31-d CERT_C-STR31-e	Avoid accessing arrays out of bounds Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Avoid using unsafe string functions which may cause buffer overflows
TrustInSoft Analyzer	1.38	Mem_access	Exhaustively verified

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	[STD-005-CPP]	When not using the new operator to allocate sufficient memory, memory is thought to be manually managed (the lifetime of the object) and should be deallocated and destroyed in the same manual manner. An object used outside of its lifespan is undefined behavior and can lead to errors.

Noncompliant Code

A manual management of memory is occurring, due to the user provided construction, with a call to std::malloc(). The constructor for the object is never called, and this results in undefined behavior, when the class is accessed later by s->f().

```
#include <cstdlib>

struct S {
  S();

  void f();
};

void g() {
  S *s = static_cast<S *>(std::malloc(sizeof(S)));

  s->f();

  std::free(s);
}
```

Compliant Code

The constructor and destructor are both explicitly called.

```
#include <cstdlib>
#include <new>

struct S {
  S();

  void f();
};

void g() {
  void *ptr = std::malloc(sizeof(S));
  S *s = new (ptr) S;

  s->f();

  s->~S();
  std::free(ptr);
}
```

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

Principles(s): Architect for security involves trainig the team in writing secure memory allocation routines, Adopt a Secure Coding standard to ensure this is consistent across the project, Quality Assurance Techniques to test for these types of vulnerabilities on each scan in the CI/CD pipeline.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Coverity	7.5	CHECKED_RETURN	Finds inconsistencies in how function call return values are handled
Parasoft C/C++test	2021.1	CERT_CPP-MEM52-a CERT_CPP-MEM52-b	Check the return value of new Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined
Polyspace Bug Finder	R2021a	CERT C++: MEM52-CPP	Checks for unprotected dynamic memory allocation (rule partially covered)
Helix QAC	2021.2	C++3225, C++3226, C++3227, C++3228, C++3229, C++4632	Helps to target the most critical defects using filters, suppressions, and baselines. It delivers accurate diagnostics and actionable results — enabling the most important issues first.

Coding Standard 6

Coding Standard	Label	Name of Standard
Assertions	[STD-006-CPP]	Assert calls abort(), cleanup functions register with atexit() are not called. This can lead to errors in correct termination of the program leading to errors.

Noncompliant Code

A function that is called before the program exits to clean up. The assert function if failed will exit before cleanup.

```
void cleanup(void) {
  /* Delete temporary files, restore consistent state, etc. */
}

int main(void) {
  if (atexit(cleanup) != 0) {
    /* Handle error */
  }

  /* ... */

  assert(/* Something bad didn't happen */);

  /* ... */
}
```

Compliant Code

If statements are used in place of assert to allow the cleanup to occur and ensure proper termination routines.

```
void cleanup(void) {
  /* Delete temporary files, restore consistent state, etc. */
}

int main(void) {
  if (atexit(cleanup) != 0) {
    /* Handle error */
  }

  /* ... */

  if (/* Something bad happened */) {
    exit(EXIT_FAILURE);
  }
```

Compliant Code

```
  /* ... */
}
```

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

Principles(s): Architect for security involves trainig the team in writing secure memory allocation routines, Adopt a Secure Coding standard to ensure this is consistent across the project, Quality Assurance Techniques to test for these types of vulnerabilities on each scan in the CI/CD pipeline.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	High	P1	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	6.1p0	LANG.FUNCS.ASSERTS	Not enough assertions
Coverity	2017.07	ASSERT_SIDE_EFFECT	Can detect the specific instance where assertion contains an operation/function call that may have a side effect
Parasoft C/C++test	2021.1	CERT_C-MSC11-a	CERT_C-MSC11-a

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	[STD-007-CPP]	All exceptions thrown must be caught by a matching exception handler, or the stack may not unwind correctly due to std::abort() being called so destructors may not be called.

Noncompliant Code

Neither f() nor main() catch exceptions thrown by throwing_func().

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

void f() {
  throwing_func();
}

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

Compliant Code

Handles all exceptions, making sure the stack is unwound.

```
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): Architect for security involves training the team is using assertions in development codebase, Adopt a Secure Coding standard to ensure this is consistent across the project, Quality Assurance Techniques to run such assertions on each scan in the CI/CD pipeline, thereby regression testing securely.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probable	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Helix QAC	2021.2	C++4035, C++4036, C++4037	Implemented
Parasoft C/C++test	2021.1	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	R2021a	CERT C++: ERR51-CPP	Checks for unhandled exceptions (rule partially covered)
RuleChecker	20.10	main-function-catch-all early-catch-all	Partially checked

Coding Standard 8

Coding Standard	Label	Name of Standard
Expressions	[STD-008-CPP]	Of particular note are concerns pertaining to reading unintialized memory or relying on the value of a moved-from object. As local, automatic variables assume unexpected values if they are read before they are initialized, which can lead to undefined behavior.

Noncompliant Code

An uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior.

```
#include <iostream>

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

Compliant Code

The object is initialized prior to printing its value.

```
#include <iostream>

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): sequences of operators and operands that are used for one or more of these purposes

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	High	P6	L2

Automation

Tool	Version	Checker	Description Tool
Clang	3.9	-Wdangling-initializer-list	Catches some lifetime issues related to incorrect use of std::initializer_list<>
CodeSonar	6.1p0	IO.UAC ALLOC.UAF	Use after close Use after free
Parasoft C/C++test	2021.1	CERT_CPP-EXP54-a CERT_CPP-EXP54-b CERT_CPP-EXP54-c	CERT_CPP-EXP54-c Do not use resources that have been freed The address of an object with automatic storage shall not be returned from a function The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist
Polyspace Bug Finder	R2021a	CERT C++: EXP54-CPP	Checks for: Non-initialized variable or pointer Use of previously freed pointer Pointer or reference to stack variable leaving scope Accessing object with temporary lifetime Rule partially covered.

Coding Standard 9

Coding Standard	Label	Name of Standard
Polymorphic Object	[STD-009-CPP]	Deleting a object through pointer to a type without virtual destructor results in undefined behavior.

Noncompliant Code

The implicitly declared destructor is not declared as virtual in the presence of other virtual functions.

```
struct Base {
  virtual void f();
};

struct Derived : Base {};

void f() {
  Base *b = new Derived();
  // ...
  delete b;
}
```

Compliant Code

The destructor for base is declared explicitly as a virtual destructor. This guarantees that the polymorphic delete operation will have well-defined behavior.

```
struct Base {
  virtual ~Base() = default;
  virtual void f();
};

struct Derived : Base {};

void f() {
  Base *b = new Derived();
  // ...
  delete b;
}
```

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

Principles(s): Adopt a Secure Coding Standard – Using the coding standards will help avoid errors like, not calling a virtual destructor when dealing with polymorphic types.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Likely	Low	P9	L2

Automation

Tool	Version	Checker	Description Tool
LDRA tool suite	9.7.1	303 S	Partially implemented
Parasoft C/C++test	2021.1	CERT_CPP-OOP52-a	Define a virtual destructor in classes used as base classes which have virtual functions
PRQA QA-C++	4.4	3402, 3403, 3404	Do not use resources that have been freed
Polyspace Bug Finder	R2020a	CERT C++: OOP52-CPP	Checks for situations when a class has virtual functions but not a virtual destructor (rule partially covered)

Coding Standard 10

Coding Standard	Label	Name of Standard
Integer Precision	[STD-010-CPP]	An integer's size is contributed by the padding bits, but this does not inherently carry over to the precision. This can lead to incorrect assumptions about the numeric range of these types.

Noncompliant Code

If this code runs on a platform where unsigned int has one or more padding bits, it can result in a value for exp that are too large.

```
#include <limits.h>

unsigned int pow2(unsigned int exp) {
  if (exp >= sizeof(unsigned int) * CHAR_BIT) {
    /* Handle error */
  }
  return 1 << exp;
}
```

Compliant Code

Using popcount() function allows the code to determine the precision of any integer type, signed or unsigned. This function will do this by counting the number of bits set on any unsigned integer.

```
#include <stddef.h>
#include <stdint.h>

/* Returns the number of set bits */
size_t popcount(uintmax_t num) {
  size_t precision = 0;
  while (num != 0) {
    if (num % 2 == 1) {
      precision++;
    }
    num >>= 1;
  }
  return precision;
}
#define PRECISION(umax_value) popcount(umax_value)
```

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

Principles(s): Validate Input Data – Assumptions with input data can lead to errors. This principle ties in as the focus is on using less assumptions with data handling, and in this case integer precision.

Threat Level

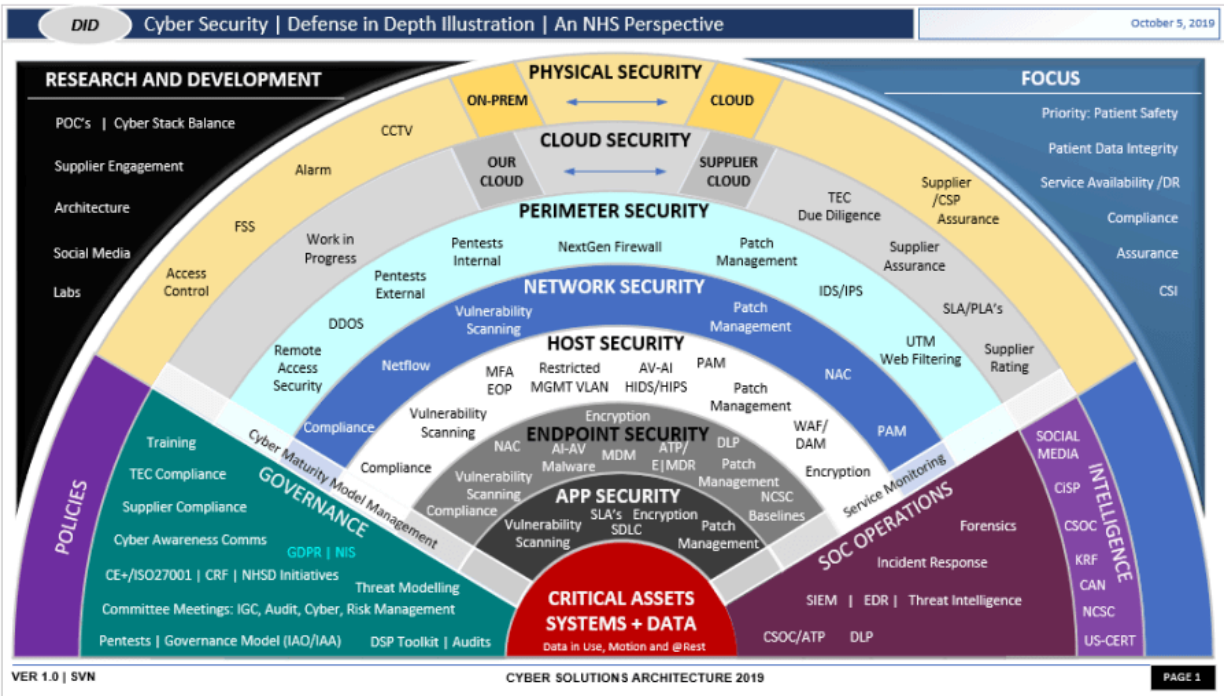
Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	P2	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	20.10		Supported: Astrée reports overflows due to insufficient precision.
Helix QAC	2021.1	C0582 C++3115	Implemented
Parasoft C/C++test	2021.1	CERT_C-INT35-a	Use correct integer precisions when checking the right hand operand of the shift operator
Polyspace Bug Finder	R2021a	CERT C: Rule INT35-C	Checks for situations when integer precisions are exceeded (rule fully covered)

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

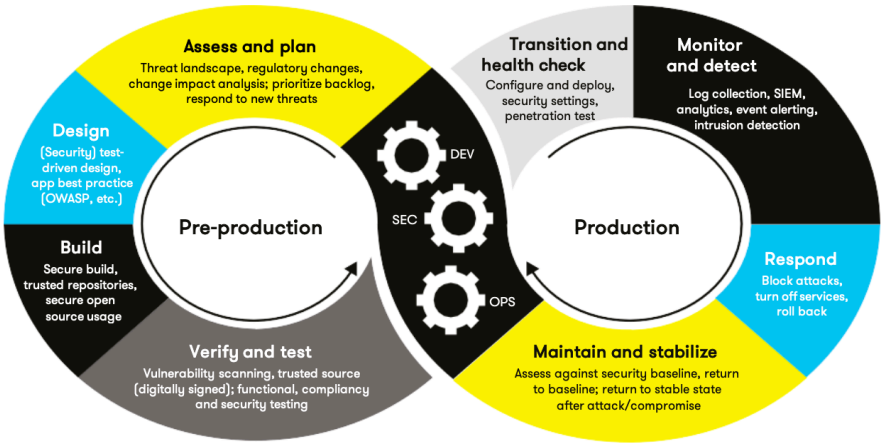
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, 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	Likely	High	P9	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]	Low	Unlikely	High	P1	L3
[STD-007-CPP]	Low	Probable	Medium	P4	L3
[STD-008-CPP]	High	Probable	Medium	P4	L3
[STD-009-CPP]	Low	Likely	Low	P9	L2
[STD-010-CPP]	Low	Unlikely	Medium	P2	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 in rest	We shall encrypt data at rest using full-disk encryption at the server level, as well as database encryption using MySQL Server, and provide a backup strategy.
Encryption at flight	Use up to date, secure libraries, use Public Key infrastructure for end-to-end protection on message bodies or attachments, use Managed File Transfer or SSH with expiration date on the link, password access, utilize Data leak prevention mechanisms built into cloud services,
Encryption in use	Utilize identity management mechanisms to confirm user roles and identity, allow conditional access to the tools functionality based on the user roles and other parameters. Use IRM digital rights management, to apply persistent protection to documentation.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	Process of identifying a user, using valid credentials of user and password. Control how a user is authenticated using a secured local database or external AWS server, prefer to use tried and trusted protocol.
Authorization	After the user has been authenticated, authorization shall be used to determine which resources and functionality the user is allowed to access and which operations can be performed.
Accounting	Monitor and log any user events while they are logging in/out or utilizing the resources, as well as user uptime or any other configured parameter.

*****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	
1.5	10/09/2021	Updated All Policies	Victor Feight	Professor Alhweiti
Version	**Date**	**Description**	**Edited By**	**Approved By**

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV