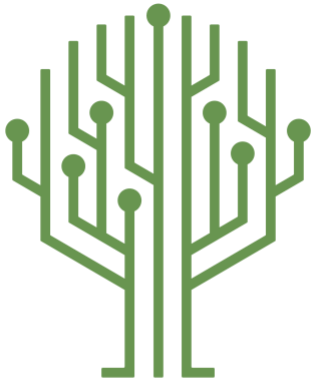


Green Pace Secure Development Policy

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.

Ten Core Security Principles

Principles	Write a short paragraph explaining each of the 10 principles of security.
1. Validate Input Data	Ensure all input being entered into the program is checked to be valid input. Also, ensure the input is sanitized to try to prevent threats such as SQL injection.
2. Heed Compiler Warnings	Compiler warnings will not keep your program from running. However, these warnings can indicate a potential issue with the code that can cause the program to react unexpectedly and could indicate a potential security issue.
3. Architect and Design for Security Policies	Security policy should be considered at the start of the development process. Any policy should be reflected in the overall architecture and design of the application. This will help to ensure that the policies are followed throughout the program.
4. Keep It Simple	In simple terms, this means favor simple code over complicated solutions. The more complicated a program is, the more chances there are for something to go wrong, or a flaw be introduced that could be exploited.
5. Default Deny	Default deny means that access and privileges are automatically denied and only granted when explicitly required. This helps to limit the reach of potential issues or security breaches by containing the problem to a single part.
6. Adhere to the Principle of Least Privilege	The principle of least privilege states that the lowest level of privilege needed to complete a task is what should be given. This helps to keep the reach of any potential issue confined to a smaller area.
7. Sanitize Data Sent to Other Systems	Before sending data to a different system, ensure the data is valid and formatted in a way that the other system is expecting. The benefit of this is that it minimizes the risk of causing an issue if the other system were to not properly validate its input.
8. Practice Defense in Depth	Depth in defense is the practice of using many different layers of security. This add redundancy into the system so that if one layer fails, there should be another that still keeps that part of the system secure.
9. Use Effective Quality Assurance Techniques	The use of thorough testing using effective quality assurance techniques help to ensure that any issues are found and fixed before they can be exploited. This results in a much more secure program.

Principles	Write a short paragraph explaining each of the 10 principles of security.
10. Adopt a Secure Coding Standard	The adoption of a secure coding standard helps to deliver properly secured code consistently and that best practices are followed. A good security policies is still no good if it is not followed, so it is important to create enforceable requirements that the policies are followed.

C/C++ Ten Coding Standards

All coding standards come from the SEI CERT C++ Coding Standard.

Coding Standard 1

Coding Standard	Label	Never qualify a reference type with const or volatile
Data Type	STD-001-CPP	Do not attempt to cv-qualify a reference type because it results in undefined behavior. A conforming compiler is required to issue a diagnostic message. However, if the compiler does not emit a fatal diagnostic, the program may produce surprising results, such as allowing the character referenced by p to be mutated.

Noncompliant Code

This noncompliant code example correctly declares p to be a reference to a const-qualified char. The subsequent modification of p makes the program ill-formed.

```
#include <iostream>

void f(char c) {
  const char &p = c;
  p = 'p'; // Error: read-only variable is not assignable
  std::cout << c << std::endl;
}
```

Compliant Code

This compliant solution removes the const qualifier.

```
#include <iostream>

void f(char c) {
  char &p = c;
  p = 'p';
  std::cout << c << std::endl;
}
```

Principles(s): 2 – Heed Complier Warnings: Warnings like this in the IDE should not be ignored. Follow the remediation steps given by the warning to resolve the issue.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Low	P3	L3

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	7.2.0	CertC++-DCL52	
Clang	3.9		Clang checks for violations of this rule and produces an error without the need to specify any special flags or options.
Helix QAC	2023.1	C++0014	
Klocwork	2023.1	CERT.DCL.REF_TYPE.CONST_OR_VOLATILE	
Parasoft C/C++test	2023.1	CERT_CPP-DCL52-a	Never qualify a reference type with 'const' or 'volatile'
Polyspace Bug Finder	R2023a	CERT C++: DCL52-CPP	Checks for: • const-qualified reference types • Modification of const-qualified reference types Rule fully covered.

Coding Standard 2

Coding Standard	Label	Value-returning functions must return a value from all exit paths
Data Value	STD-002-CPP	A value-returning function must return a value from all code paths; otherwise, it will result in undefined behavior. This includes returning through less-common code paths, such as from a function-try-block.

Noncompliant Code

In this compliant solution, all code paths now return a value.

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

Compliant Code

In this compliant solution, all code paths now return a value.

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

Principles(s): 2 – Heed Complier Warnings: Warnings like this in the IDE should not be ignored. Follow the remediation steps given by the warning to resolve the issue.
4 – Keep It Simple: Do not overcomplicate the code, make it easy to understand so issues like this can be caught easier.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	P8	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	return-implicit	Fully checked
Axivion Bauhaus Suite	7.2.0	CertC++-MSC52	

Tool	Version	Checker	Description Tool
Clang	3.9	-Wreturn-type	Does not catch all instances of this rule, such as function-try-blocks
CodeSonar	7.4p0	LANG.STRUCT.MRS	Missing return statement
Helix QAC	2023.1	DF2888	
Klocwork	2023.1	FUNCRET.GEN FUNCRET.IMPLICIT	
LDRA tool suite	9.7.1	2 D, 36 S	Fully implemented
Parasoft C/C++test	2023.1	CERT_CPP-MSC52-a	All exit paths from a function, except main(), with non-void return type shall have an explicit return statement with an expression
Polyspace Bug Finder	R2023a	CERT C++: MSC52-CPP	Checks for missing return statements (rule partially covered)
PVS-Studio	7.25	V591	
RuleChecker	22.10	return-implicit	Fully checked
SonarQube C/C++ Plugin	4.10	S935	

Coding Standard 3

Coding Standard	Label	Range check element access
String Correctness	STD-003-CPP	The std::string index operators const_reference operator[](size_type) const and reference operator[](size_type) return the character stored at the specified position, pos. When pos >= size(), a reference to an object of type charT with value charT() is returned. The index operators are unchecked (no exceptions are thrown for range errors), and attempting to modify the resulting out-of-range object results in undefined behavior.

Noncompliant Code

This noncompliant code example attempts to replace the initial character in the string with a capitalized equivalent. However, if the given string is empty, the behavior is undefined.

```
#include <string>
#include <locale>

void capitalize(std::string &s) {
  std::locale loc;
  s.front() = std::use_facet<std::ctype<char>>(loc).toupper(s.front());
}
```

Compliant Code

In this compliant solution, the call to std::string::front() is made only if the string is not empty.

```
#include <string>
#include <locale>

void capitalize(std::string &s) {
  if (s.empty()) {
    return;
  }

  std::locale loc;
  s.front() = std::use_facet<std::ctype<char>>(loc).toupper(s.front());
}
```

Principles(s): 9 - Use Effective Quality Assurance Techniques: This will help ensure the code is properly tested to catch any introduction of this issue to the code.
10 – Adopt a Secure Coding Standard: Adopting and following a secure coding standard will help to ensure that code is written in such a way that this type of issue is never introduced, or if it is, then the code should be written so that any unexpected behavior is caught.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Unlikely	Medium	P6	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	assert_failure	
CodeSonar	7.4p0	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
Helix QAC	2023.1	C++3162, C++3163, C++3164, C++3165	
Parasoft C/C++test	2023.1	CERT_CPP-STR53-a	Guarantee that container indices are within the valid range
Polyspace Bug Finder	R2023a	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 4

Coding Standard	Label	Sanitize data passed to complex subsystems
SQL Injection	STD-004-CPP	String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result, it is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted.

Noncompliant Code

Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. This example shows an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system().

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

Compliant Code

The allow-listing approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the tcp_wrappers package, shows the allow-listing approach.

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

Principles(s): 1 – Validate Input Data: Any input is validated and sanitized to prevent any unexpected behavior.
4 – Keep Is Simple: Code that is easier to understand makes it easier to predict & defend against any issues from input.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	23.04		Supported by stubbing/taint analysis

Tool	Version	Checker	Description Tool
CodeSonar	7.4p0	IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC	Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation
Coverity	6.5	TAINTED_STRING	Fully implemented
Klocwork	2023.2	NNTS.TAINTED SV.TAINTED.INJECTION	
LDRA tool suite	9.7.1	108 D, 109 D	Partially implemented
Parasoft C/C++test	2023.1	CERT_C-STR02-a CERT_C-STR02-b CERT_C-STR02-c	Protect against command injection Protect against file name injection Protect against SQL injection
Polyspace Bug Finder	R2023a	CERT C: Rec. STR02-C	Checks for: • Execution of externally controlled command • Command executed from externally controlled path • Library loaded from externally controlled path Rec. partially covered.

Coding Standard 5

Coding Standard	Label	Properly deallocate dynamically allocated resources
Memory Protection	STD-005-CPP	The C++ programming language adds additional ways to allocate memory, such as the operators new, new[], and placement new, and allocator objects. Unlike C, C++ provides multiple ways to free dynamically allocated memory, such as the operators delete, delete[](), and deallocation functions on allocator objects.

Noncompliant Code

In this noncompliant code example, the local variable space is passed as the expression to the placement new operator. The resulting pointer of that call is then passed to ::operator delete(), resulting in undefined behavior due to ::operator delete() attempting to free memory that was not returned by ::operator new().

```
#include <iostream>

struct S {
  S() { std::cout << "S::S()" << std::endl; }
  ~S() { std::cout << "S::~S()" << std::endl; }
};

void f() {
  alignas(struct S) char space[sizeof(struct S)];
  S *s1 = new (&space) S;

  // ...

  delete s1;
}
```

Compliant Code

This compliant solution removes the call to ::operator delete(), instead explicitly calling s1's destructor. This is one of the few times when explicitly invoking a destructor is warranted.

```
#include <iostream>

struct S {
  S() { std::cout << "S::S()" << std::endl; }
  ~S() { std::cout << "S::~S()" << std::endl; }
};

void f() {
  alignas(struct S) char space[sizeof(struct S)];
  S *s1 = new (&space) S;
  // ...
  s1->~S();
}
```

Principles(s): 8 – Practice Defense in Depth: Having multiple layers of defense will help to catch or prevent any breach as a result of a memory allocation issue.
9 – Use Effective Quality Assurance Techniques: Proper testing and analysis should help you catch and memory issues.

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	
Axivion Bauhaus Suite	7.2.0	CertC++-MEM51	
Clang	3.9	clang-analyzer-cplusplus.NewDeleteLeaks -Wmismatched-new-delete clang-analyzer-unix.MismatchedDeallocator	Checked by clang-tidy, but does not catch all violations of this rule
CodeSonar	7.4p0	ALLOC.FNH ALLOC.DF ALLOC.TM ALLOC.LEAK	Free non-heap variable Double free Type mismatch Leak
Helix QAC	2023.1	C++2110, C++2111, C++2112, C++2113, C++2118, C++3337, C++3339, C++4262, C++4263, C++4264	
Klocwork	2023.1	CL.FFM.ASSIGN CL.FFM.COPY CL.FMM CL.SHALLOW.ASSIGN CL.SHALLOW.COPY FMM.MIGHT FMM.MUST FNH.MIGHT FNH.MUST FUM.GEN.MIGHT FUM.GEN.MUST UNINIT.CTOR.MIGHT UNINIT.CTOR.MUST UNINIT.HEAP.MIGHT UNINIT.HEAP.MUST	
LDRA tool suite	9.7.1	232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D	Partially implemented

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
Parasoft Insure++			Runtime detection
Polyspace Bug Finder	R2023a	CERT C++: MEM51-CPP	Checks for: Invalid deletion of pointer Invalid free of pointer Deallocation of previously deallocated pointer Rule partially covered.
PVS-Studio	7.25	V515, V554, V611, V701, V748, V773, V1066	
SonarQube C/C++ Plugin	4.10	S1232	

Coding Standard 6

Coding Standard	Label	Understand the termination behavior of assert() and abort()
Assertions	STD-006-CPP	Because assert() calls abort(), cleanup functions registered with atexit() are not called. If the intention of the programmer is to properly clean up in the case of a failed assertion, then runtime assertions should be replaced with static assertions where possible. When the assertion is based on runtime data, the assert should be replaced with a runtime check that implements the adopted error strategy.

Noncompliant Code

This noncompliant code example defines a function that is called before the program exits to clean up:

```
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

In this compliant solution, the call to assert() is replaced with an if statement that calls exit() to ensure that the proper termination routines are run:

```
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);
  }
  /* ... */
}
```

Principles(s): 3 – Architect and Design for Security Policies: Thinking about the plan and design during development will help you to know when to use assert() or abort().

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Compass/ROSE			Can detect some violations of this rule. However, it can only detect violations involving abort() because assert() is implemented as a macro
LDRA tool suite	9.7.1	44 S	Enhanced enforcement
Parasoft C/C++test	2023.1	CERT_C-ERR06-a	Do not use assertions
PC-lint Plus	1.4	586	Fully supported

Coding Standard 7

Coding Standard	Label	Handle all exceptions
Exceptions	STD-007-CPP	All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process.

Noncompliant Code

In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called.

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

void f() {
  throwing_func();
}

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

Compliant Code

In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources.

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

void f() {
  throwing_func();
}

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

Principles(s): 3 – Architect and Design for Security Policies: Following this would dictate that if an exception could be thrown, you should be trying to catch it.
9 – Use Effective Quality Assurance Techniques: The implementation of thorough testing, including edge cases, should cause any uncaught exceptions to be found, and the protected against.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probable	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	main-function-catch-all early-catch-all	Partially checked
Axivion Bauhaus Suite	7.2.0	CertC++-ERR51	
CodeSonar	7.4p0	LANG.STRUCT.UCTCH	Unreachable Catch
Helix QAC	2023.1	C++4035, C++4036, C++4037	
Klocwork	2023.1	MISRA.CATCH.ALL	
LDRA tool suite	9.7.1	527 S	Partially implemented
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	R2023a	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	Do not modify the standard namespaces
Declarations and Initialization	STD-008-CPP	Do not add declarations or definitions to the standard namespaces std or posix, or to a namespace contained therein, except for a template specialization that depends on a user-defined type that meets the standard library requirements for the original template.

Noncompliant Code

In this noncompliant code example, the declaration of x is added to the namespace std, resulting in undefined behavior.

```
namespace std {
int x;
}
```

Compliant Code

This compliant solution assumes the intention of the programmer was to place the declaration of x into a namespace to prevent collisions with other global identifiers. Instead of placing the declaration into the namespace std, the declaration is placed into a namespace without a reserved name.

```
namespace nonstd {
int x;
}
```

Principles(s): 3 – Architect and Design for Security Policies: The policies should not allow for this type of behavior.
4 – Keep It Simple: Modifying the standard namespaces is the opposite of keeping it simple.
10 – Adopt a Secure Coding Standard: A secure coding standard should not allow for this type of behavior.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Unlikely	Medium	P6	L2

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	7.2.0	CertC++-DCL58	
CodeSonar	7.4p0	LANG.STRUCT.DECL.SNM	Modification of Standard Namespaces
Helix QAC	2023.1	C++3180, C++3181, C++3182	

Tool	Version	Checker	Description Tool
Klocwork	2023.1	CERT.DCL.STD_NS_MODIFIED	
Parasoft C/C++test	2023.1	CERT_CPP-DCL58-a	Do not modify the standard namespaces 'std' and 'posix'
Polyspace Bug Finder	R2023a	CERT C++: DCL58-CPP	Checks for modification of standard namespaces (rule fully covered)
PVS-Studio	7.25	V1061	
SonarQube C/C++ Plugin	4.10	S3470	

Coding Standard 9

Coding Standard	Label	Close files when they are no longer needed
Input/Output	STD-009-CPP	A call to the std::basic_filebuf<T>::open() function must be matched with a call to std::basic_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first.

Noncompliant Code

In this noncompliant code example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic_filebuf<T>::open(), and the default std::terminate_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic_filebuf<T> object maintained by the object is not properly closed.

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

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

Compliant Code

In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed.

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

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

Compliant Code

```
  std::terminate();
}
```

Principles(s): 5 – Default Deny: The file should be closed after use to default deny access to it by anything else. 6 – Adhere to the Principle of Least Privilege: Closing the file after use limits access to the file to the code that is currently using the file.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	23.04		Supported, but no explicit checker
CodeSonar	7.4p0	ALLOC.LEAK	Leak
Compass/ROSE			
Coverity	2017.07	RESOURCE_LEAK (partial)	Partially implemented
Helix QAC	2023.2	DF2701, DF2702, DF2703	
Klocwork	2023.2	RH.LEAK	
LDRA tool suite	9.7.1	49 D	Partially implemented
Parasoft C/C++test	2023.1	CERT_C-FIO42-a	Ensure resources are freed
PC-lint Plus	1.4	429	Partially supported
Polyspace Bug Finder	R2023a	CERT C: Rule FIO42-C	Checks for resource leak (rule partially covered)
SonarQube C/C++ Plugin	3.11	S2095	

Coding Standard 10

Coding Standard	Label	Write constructor member initializers in the canonical order
Object Oriented Programming	STD-010-CPP	Always write member initializers in a constructor in the canonical order: first, direct base classes in the order in which they appear in the base-specifier-list for the class, then non-static data members in the order in which they are declared in the class definition.

Noncompliant Code

In this noncompliant code example, the member initializer list for C::C() attempts to initialize someVal first and then to initialize dependsOnSomeVal to a value dependent on someVal. Because the declaration order of the member variables does not match the member initializer order, attempting to read the value of someVal results in an unspecified value being stored into dependsOnSomeVal.

```
class C {
  int dependsOnSomeVal;
  int someVal;

public:
  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};
```

Compliant Code

This compliant solution changes the declaration order of the class member variables so that the dependency can be ordered properly in the constructor's member initializer list.

```
class C {
  int someVal;
  int dependsOnSomeVal;

public:
  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}
};
```

Principles(s): 4 – Keep Is Simple: Keeps the code consistent and clear so anyone can take a look and understand.
10 – Adopt a Secure Coding Standard: This will help ensure everyone is following the same standard.

Threat Level

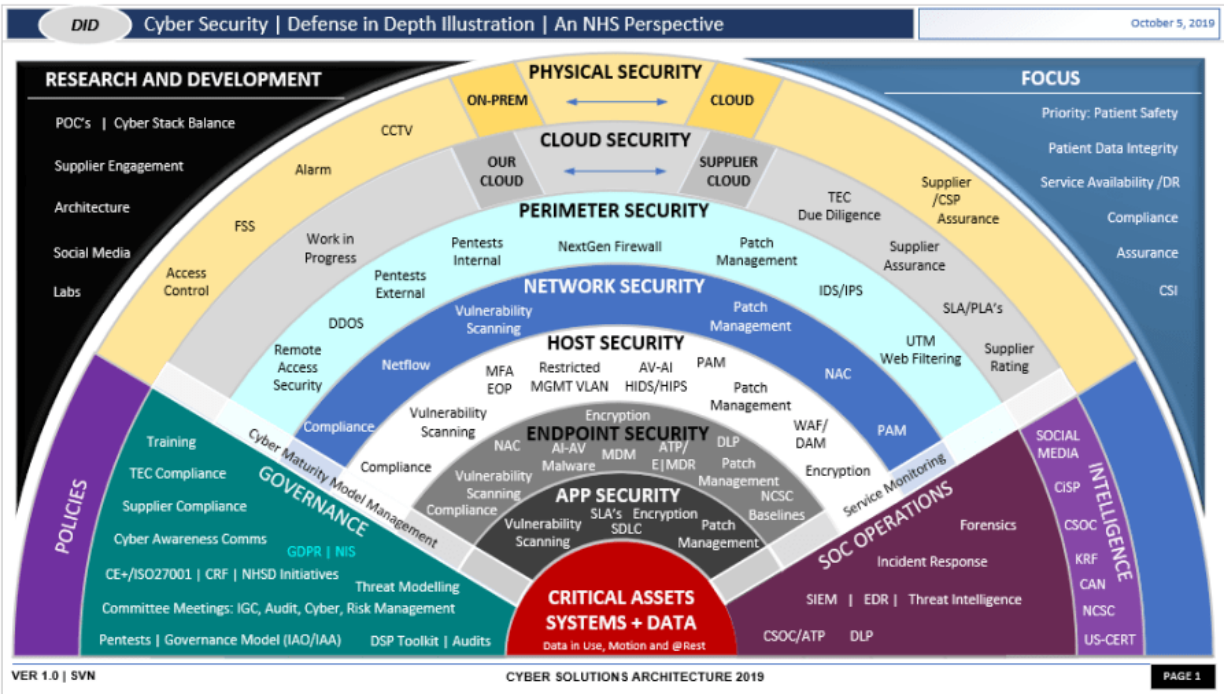
Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	initializer-list-order	Fully checked
Axivion Bauhaus Suite	7.2.0	CertC++-OOP53	
Clang	3.9	-Wreorder	
CodeSonar	7.4p0	LANG.STRUCT.INIT.OOMI	Out of Order Member Initializers
Helix QAC	2023.1	C++4053	
Klocwork	2023.1	CERT.OOP.CTOR.INIT_ORDER	
LDRA tool suite	9.7.1	206 S	Fully implemented
Parasoft C/C++test	2023.1	CERT_CPP-OOP53-a	List members in an initialization list in the order in which they are declared
Polyspace Bug Finder	R2023a	CERT C++: OOP53-CPP	Checks for members not initialized in canonical order (rule fully covered)
RuleChecker	22.10	initializer-list-order	Fully checked
SonarQube C/C++ Plugin	4.10	S3229	

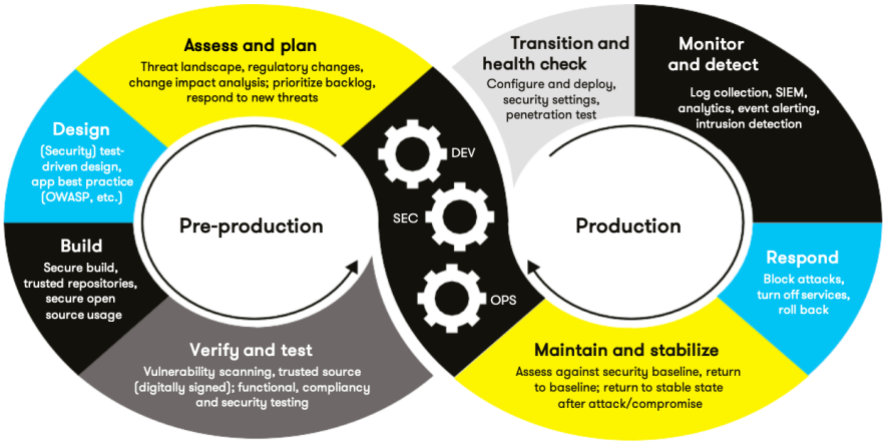
Defense-in-Depth Illustration

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



Automation

This illustration provides a visual representation of automation practices.



The use of DevSecOps can help to ensure you are creating the most secure application possible and automation can help with this process. The polices and standards set within this security policy document should be utilized during the DevSecOps process. The build and verify and test steps of the process are there first area where the use of automation can be beneficial. CI/CD pipelines can be used to automate the building and testing of the application. This helps with security to ensure consistent steps are followed when deploying and allows the security of the pipeline to be checked.

Automated tools within the CI/CD pipeline can then run and report on any unit tests, integration tests, and front-end UI tests, and end to end testing created for the application. Additionally, these tools can run static analysis looking for common coding issues or security threats, as well as check for dependency vulnerabilities.

Once building and testing is complete, the transition and health check step can utilize the CI/CD pipeline to automate the secure configuration and deployment of the application. Once deployed, tools can be used to automatically run penetration testing on the application, running through and checking for a range of common exploits. This includes threats listed in the coding standards.

Additional automated tools can gather logs output from the application and monitor for any events. If issues are detected in the logs or events alerts can be triggered notifying the necessary people and started other automated responses. These other responses can include tools in the respond step that respond to intrusions and cut off or limit access.

Lastly, during the maintain and stabilize step, more automated tools can be used to check the integrity of the system and go back to monitoring for any more issues. This can include the ongoing monitoring for any dependency vulnerabilities, which if found, can be remediated with an automated process to update the dependency to a secure version.

Summary of Risk Assessments

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-001-CPP	Low	Unlikely	Low	P3	L3
STD-002-CPP	Medium	Probable	Medium	P8	L2
STD-003-CPP	High	Unlikely	Medium	P6	L2
STD-004-CPP	High	Likely	Medium	P18	L1
STD-005-CPP	High	Likely	Medium	P18	L1
STD-006-CPP	Medium	Unlikely	Medium	P4	L3
STD-007-CPP	Low	Probable	Medium	P4	L3
STD-008-CPP	High	Unlikely	Medium	P6	L2
STD-009-CPP	Medium	Unlikely	Medium	P4	L3
STD-010-CPP	Medium	Unlikely	Medium	P4	L3

Encryption and Triple A Policies

a. Encryption	**Explain what it is and how and why the policy applies.**
Encryption in rest	Encryption at rest refers to data being in an encrypted state while it is in storage. This means that even if access to the data is gained, the data itself is unreadable unless you have the key to decrypt the data. This policy is important and should be used so that if there were a breach and the data was taken, the information within the data would still be safe because it could not be read without the key.
Encryption at flight	Encryption at flight refers to data being in an encrypted state while it is in transit from one place to another. While data is in route form one place to another, it must be encrypted. This ensures that if the data is intercepted, the information within the data would still be safe because it could not be read without the key.
Encryption in use	Encryption in use refers to the encryption of data as it is being used by the system. It is used to keep data secure at all times, even as it is changing. This is important because as more data is secured with encryption at rest and in transit, then that leave as data is being used as the weakest link to be exploited. For that reason, it is important to figure out how to keep the data encrypted as it is being used.

b. Triple-A Framework	**Explain what it is and how and why the policy applies.**
Authentication	Authentication is the process of confirming that a user is who they say they are. It is typically used when adding new users or when a user attempts to login. This policy applies whenever a user is attempting to gain access to a system or something within the system. You need to be sure that the person within the system is who you expect it to be. This is typically accomplished by verify user identification information such as a username and password.
Authorization	Authorization is the process of confirming that a user is supposed to have access to what they are requesting. It is what determines the level of access that a user has with the system. This policy applies whenever a user is attempted to access a system or something within the system. Once you have authenticated that a user is who they say they are, authorization will determine if they can gain entry to the system or an item within the system. This helps to keep users out of places they shouldn't be and also keep out those who should not be there at all.
Accounting	Accounting is the process of tracking and logging all requests and transactions made by a user. Examples of that would be changes made to a database or files accessed by a user. This policy is important so that you can know what is going on with a system. For example, if the system start behaving in an unexpected manor, you can review the logs to track down the issue and then hopefully be able to figure out a solution.

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 comply with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

- Business or technical rationale
- Risk impact analysis
- Risk mitigation analysis
- Plan to come into compliance
- Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

Policy Version History

Version	Date	Description	Edited By	Approved By
1.0	08/05/2020	Initial Template	David Buksbaum	
1.1	07/15/2023	Completed Ten Core Security Principles and Defense in Depth, and started Coding Standards	Eric Slutz	Eric Slutz
1.2	08/05/2034	Added to Coding Standards and completed Summary of Risk Assessments table	Eric Slutz	Eric Slutz
1.3	08/06/2023	Completed Coding Standards, Automation section, and Encryption and Triple A Policies	Eric Slutz	Eric Slutz

References

SEI CERT C++ Coding Standard - SEI CERT C++ Coding Standard - Confluence.

(n.d.). https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV