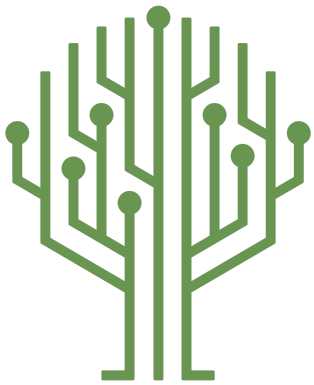


Green Pace

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	Summary
1. Validate Input Data	All input data should have a clearly defined type and format. It should then be validated against these definitions and only allowed to pass through the application flow if they meet that strict criterion. This is done to prevent any type of injection attack or exploit within our applications.
2. Heed Compiler Warnings	Compiler warnings could cause an issue with an application later on in the development process. For example, it could have unintended consequences in the application's runtime. It is for this reason that all developers should heed compiler warnings.
3. Architect and Design for Security Policies	To prevent tech debt and promote overall security of the business, all projects should be architected and designed based on our security policies. One example of this is adding logging to your application for auditability.
4. Keep It Simple	All projects should be modularized and modules should be well-documented and contain comments. Overcomplicated source code files should be avoided and broken down into smaller, digestible, files.
5. Default Deny	Default deny is similar to the network security policy of denying traffic in unused ports. All traffic which has not been expressly permitted should be blocked.
6. Adhere to the Principle of Least Privilege	From a security standpoint, users should have least privilege when onboarded to new areas and applications. This practice is to prevent users from having too much access and causing unintended damage to systems as well as to prevent hazards if their identity is compromised.
7. Sanitize Data Sent to Other Systems	Similarly to the validation of input data, data should be parsed against a model before being sent to another application to promote application integrity. Data should be formatted and meet the requirements of the receiving application.

Principles	Summary
8. Practice Defense in Depth	Layering security principles and standards, such as zero-trust (default deny and least privilege), validation of input data, the use of secrets to store sensitive credentials, firewall rules, etc. is crucial to defense in depth. Every layer of an application should implement best practice security standards and take advantage of any platform-level security implementations available to architects.
9. Use Effective Quality Assurance Techniques	Applications should all be tested but the rigor of testing should be based on the highest data sensitivity of the application pipeline. For example, all applications should be penetration tested (static and dynamic) but this process should be more involved if the data the application touches is business critical. All applications should be required to have unit and integration testing completed by the owning team.
10. Adopt a Secure Coding Standard	Use tools available based on your application architecture to enforce secure coding standards. These tools are often implemented at the enterprise level in the continuous integration and continuous delivery (CI/CD) pipeline.

C/C++ Ten Coding Standards

Coding Standard 1

Coding Standard	Label	Data Type Coding Standard
Data Type	STD-001-CPP	Never qualify a reference type with const or volatile.

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):
Heed Compiler Warnings - This particular standard in C/C++ should appear when compiling with clang

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	
Helix QAC	2022.4	C++0014	
Klocwork	2022.4	CERT.DCL.REF_TYPE.CONST_OR_VOLATILE	
Parasoft C/C++test	2022.2	CERT_CPP-DCL52-a	Never qualify a reference type with 'const' or 'volatile'
Polyspace Bug Finder	R2022b	CERT C++: DCL52-CPP	Checks for: • const-qualified reference types • Modification of const-qualified reference types Rule fully covered.
PRQA QA-C++	4.4	0014	
Clang	3.9		Clang checks for violations of this rule and produces an error without the need to specify any special flags or options.
SonarQube C/C++ Plugin	4.10	S3708	

Coding Standard 2

Coding Standard	Label	Data Value Coding Standard
Data Value	STD-002-CPP	Do not read uninitialized memory.

Noncompliant Code

In this noncompliant code example, the class member variable c is not explicitly initialized by a ctor-initializer in the default constructor. Despite the local variable s being default-initialized, the use of c within the call to S::f() results in the evaluation of an object with indeterminate value, resulting in undefined behavior.

```
class S {
  int c;

public:
  int f(int i) const { return i + c; }
};

void f() {
  S s;
  int i = s.f(10);
}
```

Compliant Code

In this compliant solution, S is given a default constructor that initializes the class member variable c.

```
class S {
  int c;

public:
  S() : c(0) {}
  int f(int i) const { return i + c; }
};

void f() {
  S s;
  int i = s.f(10);
}
```

Principles(s):
Heed Compiler Warnings - This particular standard in C/C++ may appear when compiling with clang

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Medium	P12	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	uninitialized-read	Partially checked
Clang	3.9	-Wuninitialized clang-analyzer-core.UndefinedBinaryOperatorResult	Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory.
CodeSonar	7.2p0	LANG.STRUCT.RPL LANG.MEM.UVAR	Return pointer to local Uninitialized variable
Helix QAC	2022.4	DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF978	
Klocwork	2022.4	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	2022.2	CERT_CPP-EXP53-a	Avoid use before initialization
Parasoft Insure++			Runtime detection
Polyspace Bug Finder	R2022b	CERT C++: EXP53-CPP	Checks for: • Non-initialized variable

Tool	Version	Checker	Description Tool
			• 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	7.23	V546, V573, V614, V670, V679, V730, V788, V1007, V1050	
RuleChecker	22.10	uninitialized-read	Partially checked

Coding Standard 3

Coding Standard	Label	String Correctness Coding Standard
String Correctness	STD-003-CPP	Range check element access.

Noncompliant Code

In this noncompliant code example, the value returned by the call to get_index() may be greater than the number of elements stored in the string, resulting in undefined behavior.

```
#include <string>

extern std::size_t get_index();

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

Compliant Code

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

```
#include <stdexcept>
#include <string>
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
  }
}
```

Principles(s):
Use Effective Quality Assurance Techniques - basic unit testing should enforce standards of this type
Adopt a Secure Coding Standard - Pen testing or automated SAST should be able to detect this error

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	7.2p0	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	2022.4	C++3162, C++3163, C++3164, C++3165	
Parasoft C/C++test	2022.2	CERT_CPP-STR53-a	Guarantee that container indices are within the valid range
Polyspace Bug Finder	R2022b	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	SQL Injection Coding Standard
SQL Injection	STD-004-CPP	Prevent SQL injection.

Noncompliant Code

The JDBC library provides an API for building SQL commands that sanitize untrusted data. The java.sql.PreparedStatement class properly escapes input strings, preventing SQL injection when used correctly. This code example modifies the doPrivilegedAction() method to use a PreparedStatement instead of java.sql.Statement. However, the prepared statement still permits a SQL injection attack by incorporating the unsanitized input argument username into the prepared statement.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

class Login {
  public Connection getConnection() throws SQLException {
    DriverManager.registerDriver(new
            com.microsoft.sqlserver.jdbc.SQLServerDriver());
    String dbConnection =
      PropertyManager.getProperty("db.connection");
    // Can hold some value like
    // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"
    return DriverManager.getConnection(dbConnection);
  }

  String hashPassword(char[] password) {
    // Create hash of password
  }

  public void doPrivilegedAction(
    String username, char[] password
  ) throws SQLException {
    Connection connection = getConnection();
    if (connection == null) {
      // Handle error
    }
    try {
      String pwd = hashPassword(password);
```

Noncompliant Code

```
      String sqlString = "select * from db_user where username=" +
        username + " and password =" + pwd;
      PreparedStatement stmt = connection.prepareStatement(sqlString);

      ResultSet rs = stmt.executeQuery();
      if (!rs.next()) {
        throw new SecurityException("User name or password incorrect");
      }

      // Authenticated; proceed
    } finally {
      try {
        connection.close();
      } catch (SQLException x) {
        // Forward to handler
      }
    }
  }
}
```

Compliant Code

This compliant solution uses a parametric query with a ? character as a placeholder for the argument. This code also validates the length of the username argument, preventing an attacker from submitting an arbitrarily long user name.

```
public void doPrivilegedAction(
  String username, char[] password
) throws SQLException {
  Connection connection = getConnection();
  if (connection == null) {
    // Handle error
  }
  try {
    String pwd = hashPassword(password);

    // Validate username length
    if (username.length() > 8) {
      // Handle error
    }

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

Compliant Code

```
    ResultSet rs = stmt.executeQuery();
    if (!rs.next()) {
      throw new SecurityException("User name or password incorrect");
    }

    // Authenticated; proceed
  } finally {
    try {
      connection.close();
    } catch (SQLException x) {
      // Forward to handler
    }
  }
}
```

Principles(s):
Validate Input Data - using parameterized queries and checking for standard injection patterns using regex helps enforce this standard
Sanitize Data Sent to Other Systems - data should pass through a model which maps to the database and not allow abnormal interactions
Use Effective Quality Assurance Techniques - penetration testers help to enforce this standard by flagging unsafe SQL invocation

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Medium	P12	L1

Automation

Tool	Version	Checker	Description Tool
The Checker Framework	2.1.3	Tainting Checker	Trust and security errors (see Chapter 8)
CodeSonar	7.2p0	JAVA.IO.INJ.SQL	SQL Injection (Java)
Coverity	7.5	SQLI FB.SQL_PREPARED_STATEMENT_GENERATED_ FB.SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE	Implemented

Tool	Version	Checker	Description Tool
Findbugs	1.0	SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE	Implemented
Fortify	1.0	HTTP_Response_Splitting SQL_Injection__Persistence SQL_Injection	Implemented
Klocwork		SV.DATA.BOUND SV.DATA.DB SV.HTTP_SPLIT SV.PATH SV.PATH.INJ SV.SQL	Implemented
Parasoft Jtest	2022.2	CERT.IDS00.TDSQL	Protect against SQL injection
SonarQube	6.7	S2077 S3649	Executing SQL queries is security-sensitive SQL queries should not be vulnerable to injection attacks
SpotBugs	4.6.0	SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING	Implemented

Coding Standard 5

Coding Standard	Label	Memory Protection Coding Standard
Memory Protection	STD-005-CPP	Properly deallocate dynamically allocated resources.

Noncompliant Code

In the following noncompliant code example, an array is allocated with array new[] but is deallocated with a scalar delete call instead of an array delete[] call, resulting in undefined behavior.

```
void f() {
  int *array = new int[10];
  // ...
  delete array;
}
```

Compliant Code

In the compliant solution, the code is fixed by replacing the call to delete with a call to delete [] to adhere to the correct pairing of memory allocation and deallocation functions.

```
void f() {
  int *array = new int[10];
  // ...
  delete[] array;
}
```

Principles(s):
Use Effective Quality Assurance Techniques - implementing effective unit testing locally or via CI/CD pipeline should validate the behavior of the application
Heed Compiler Warnings - Clang may help to enforce this standard

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	7.2p0	ALLOC.FNH ALLOC.DF ALLOC.TM ALLOC.LEAK	Free non-heap variable Double free Type mismatch Leak
Helix QAC	2022.4	C++2110, C++2111, C++2112, C++2113, C++2118, C++3337, C++3339, C++4262, C++4263, C++4264	
Klocwork	2022.4	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
Parasoft C/C++test	2022.2	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

Tool	Version	Checker	Description Tool
			declared for classes with a nontrivial destructor Properly deallocate dynamically allocated resources
Parasoft Insure++			Runtime detection
Polyspace Bug Finder	R2022b	CERT C++: MEM51-CPP	Checks for: • Invalid deletion of pointer • Invalid free of pointer • Deallocation of previously deallocated pointer Rule partially covered.
PRQA QA-C++	4.4	2110, 2111, 2112, 2113, 2118, 3337, 3339, 4262, 4263, 4264	
PVS-Studio	7.23	V515, V554, V611, V701, V748, V773, V1066	
SonarQube C/C++ Plugin	4.10	S1232	

Coding Standard 6

Coding Standard	Label	Assertions Coding Standard
Assertions	STD-006-CPP	Use a static assertion to test the value of a constant expression.

Noncompliant Code

This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly

```
#include <assert.h>

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

int func(void) {
  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned
int) + sizeof(unsigned int));
}
```

Compliant Code

For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution. Using #error directives allows for clear diagnostic messages. Because this approach evaluates assertions at compile time, there is no runtime penalty.

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

#if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned
int) + sizeof(unsigned int)))
  #error "Structure must not have any padding"
#endif
```

Principles(s):
Use Effective Quality Assurance Techniques - implementing effective unit testing locally or via CI/CD pipeline should validate the behavior of the application
Heed Compiler Warnings - clang may detect and enforce this standard

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	High	P1	L3

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	7.2.0	CertC-DCL03	
Clang	3.9	misc-static-assert	Checked by clang-tidy
CodeSonar	7.2p0	(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
ECLAIR	1.2	CC2.DCL03	Fully implemented
LDRA tool suite	9.7.1	44 S	Fully implemented

Coding Standard 7

Coding Standard	Label	Exceptions Coding Standard
Exceptions	STD-007-CPP	Honor exception specifications.

Noncompliant Code

In this noncompliant code example, a function is declared as nonthrowing, but it is possible for std::vector::resize() to throw an exception when the requested memory cannot be allocated.

```
#include <cstddef>
#include <vector>

void f(std::vector<int> &v, size_t s) noexcept(true) {
  v.resize(s); // May throw
}
```

Compliant Code

In this compliant solution, the function's noexcept-specification is removed, signifying that the function allows all exceptions.

```
#include <cstddef>
#include <vector>

void f(std::vector<int> &v, size_t s) {
  v.resize(s); // May throw, but that is okay
}
```

Principles(s):
Use Effective Quality Assurance Techniques - implementing effective unit testing locally or via CI/CD pipeline should validate the behavior of the application
Architect and Design for Security Policies - error handling should be a consideration when architecting applications, all errors should be handled and show minimal details on the client side
Sanitize Data Sent to Other Systems - error messages to other systems should always be handled or sensitive application details may be exposed in the error stack

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Likely	Low	P9	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	unhandled-throw-noexcept	Partially checked
Axivion Bauhaus Suite	7.2.0	CertC++-ERR55	
CodeSonar	7.2p0	LANG.STRUCT.EXCP.THROW	Use of throw
Helix QAC	2022.4	C++4035, C++4036, C++4632	
LDRA tool suite	9.7.1	56 D	Partially implemented
Parasoft C/C++Test	2022.2	CERT_CPP-ERR55-a	Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)
Polyspace Bug Finder	R2022b	CERT C++: ERR55-CPP	Checks for noexcept functions exiting with exception (rule fully covered)
PRQA QA-C++	4.4	4035, 4036, 4632	
RuleChecker	22.10	unhandled-throw-noexcept	Partially checked

Coding Standard 8

Coding Standard	Label	Name of Standard
Concurrency	STD-008-CPP	Write constructor member initializers in the canonical order.

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):
Heed Compiler Warnings - clang should be able to enforce this standard
Use Effective Quality Assurance Techniques - if the application compiles, unit testing should be in place to assure the application operates as intended

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	7.2p0	LANG.STRUCT.INIT.OOMI	Out of Order Member Initializers
Helix QAC	2022.4	C++4053	
Klocwork	2022.4	CERT.OOP.CTOR.INIT_ORDER	
LDRA tool suite	9.7.1	206 S	Fully implemented
Parasoft C/C++test	2022.2	CERT_CPP-OOP53-a	List members in an initialization list in the order in which they are declared
Polyspace Bug Finder	R2022b	CERT C++: OOP53-CPP	Checks for members not initialized in canonical order (rule fully covered)
PRQA QA-C++	4.4	4053	
RuleChecker	22.10	initializer-list-order	Fully checked
SonarQube C/C++ Plugin	4.10	S3229	

Coding Standard 9

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

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();
```

Compliant Code

```
  if (file.fail()) {
    // Handle error
  }
  std::terminate();
}
```

Principles(s):
Adopt a Secure Coding Standard - With any language, static files should be accessed minimally and closed if accessed

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.2p0	ALLOC.LEAK	Leak
Helix QAC	2022.4	DF4786, DF4787, DF4788	
Klocwork	2022.4	RH.LEAK	
Parasoft C/C++test	2022.2	CERT_CPP-FIO51-a	Ensure resources are freed
Parasoft Insure++			Runtime detection
Polyspace Bug Finder	R2022b	CERT C++: FIO51-CPP	Checks for resource leak (rule partially covered)

Coding Standard 10

Coding Standard	Label	Name of Standard
Object-Oriented Programming	STD-010-CPP	Do not delete a polymorphic object without a virtual destructor.

Noncompliant Code

In this noncompliant example, the explicit pointer operations have been replaced with a smart pointer object, demonstrating that smart pointers suffer from the same problem as other pointers.

```
#include <memory>

struct Base {
  virtual void f();
};

struct Derived : Base {};

void f() {
  std::unique_ptr<Base> b = std::make_unique<Derived()>();
}
```

Compliant Code

In this compliant solution, the destructor for Base has an explicitly declared virtual destructor, ensuring that the polymorphic delete operation results in well-defined behavior.

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

struct Derived : Base {};

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

Principles(s):
Use Effective Quality Assurance Techniques - Unit and integration testing coverage should be maximized to prevent undefined behavior during the application's runtime
Adopt a Secure Coding Standard - a full understand of OOP in the language (C/C++) in this instance is necessary to enforce this standard

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Likely	Low	P9	L2

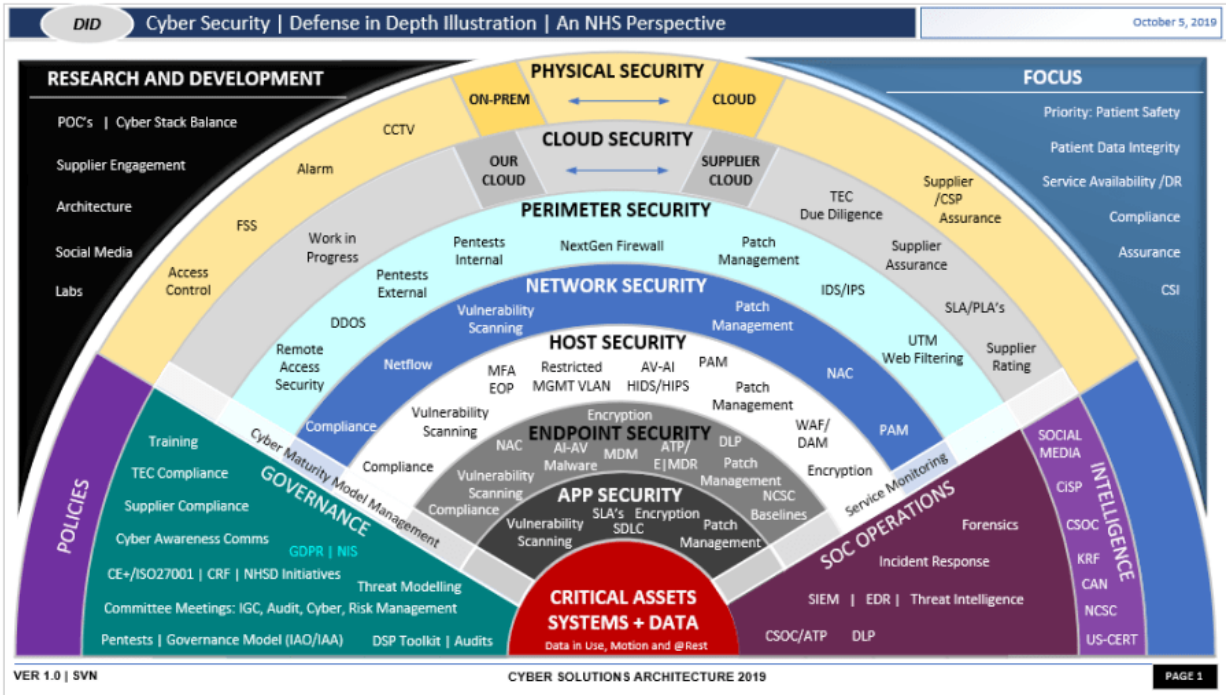
Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	non-virtual-public-destructor-in-non-final-class	Partially checked
Axivion Bauhaus Suite	7.2.0	CertC++-OOP52	
Clang	3.9	-Wdelete-non-virtual-dtor	
CodeSonar	7.2p0	LANG.STRUCT.DNVD	delete with Non-Virtual Destructor
Helix QAC	2022.4	C++3402, C++3403, C++3404	
Klocwork	2022.4	CL.MLK.VIRTUAL CWARN.DTOR.NONVIRT.DELETE	
LDRA tool suite	9.7.1	303 S	Partially implemented
Parasoft C/C++test	2022.2	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	
Polyspace Bug Finder	R2022b	CERT C++: OOP52-CPP	Checks for situations when a class has virtual functions but not a virtual destructor (rule partially covered)
PVS-Studio	7.23	V599, V689	
RuleChecker	22.10	non-virtual-public-destructor-in-non-final-class	Partially checked

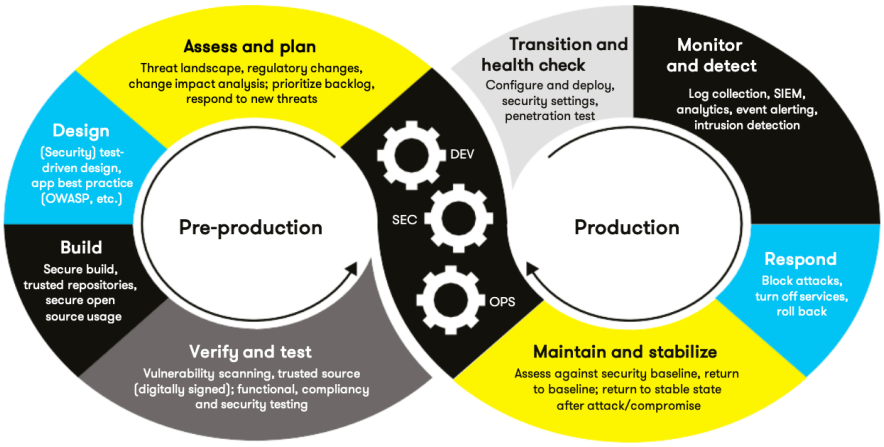
Tool	Version	Checker	Description Tool
SonarQube C/C++ Plugin	4.10	S1235	

Defense-in-Depth Illustration

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



Automation



Automation should be a part of the build and verify phases of pre-production. For the purpose of this document we will assume no SAST or unit testing were done by developers. The design process should heavily involve security and require approval to move to production. It is possible to incorporate parts of the security testing and approval process with automation tools. For example, static code scans should be required as well as validation from the development team that testing has been completed. In terms of automation within the CI/CD pipeline, GitOps can be utilized to implement automated code scanning tools when commits are made to internal repositories. This will add time to the build process; however, it is a fundamental way to assure compliance with common coding standards. These tools would scan the source code of an application and report any known malpractice and vulnerabilities. Automated gatekeeping processes should prevent applications with known vulnerabilities from being deployed in production until the findings are remediated. Once an application is in production, maintainers should be alerted if new findings are revealed. For example, the log4j vulnerability was discovered after many affected applications were deployed within production. In the case of such a post-mortem finding, teams should be given notice and a period within which to remediate the finding. If application architecture plans to be changed, teams should need to submit these changes in an automated fashion to information security for further review.

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	Low	Unlikely	Low	P3	L3
STD-002-CPP	High	Probable	Medium	P12	L1
STD-003-CPP	High	Unlikely	Medium	P6	L2
STD-004-CPP	High	Probable	Medium	P12	L1
STD-005-CPP	High	Likely	Medium	P18	L1
STD-006-CPP	Low	Unlikely	High	P1	L3
STD-007-CPP	Low	Likely	Low	P9	L2
STD-008-CPP	Medium	Unlikely	Medium	P4	L3
STD-009-CPP	Medium	Unlikely	Medium	P4	L3
STD-010-CPP	Low	Likely	Low	P9	L2

Policies for Encryption and Triple A

Encryption	Summary
Encryption in rest	Encryption in rest protects stored data ranging from local hardware (HDD, SSD,NVMe) storage to cloud assets. Encryption is a requirement for sensitive data to prevent potential attackers from seizing digital assets or private information.
Encryption at flight	Encryption at flight (or encryption in transit) pertains to protecting data transferred between systems. Data is protected in transit by scrambling the data so that, if intercepted, bad actors cannot easily consume it. This is usually implemented using SSL or TLS.
Encryption in use	Encryption in use protects data that is created, edited, or otherwise defined as in-use. Encrypting data in memory slows data processing; however, if an application is dealing with highly sensitive data, this precaution is necessary.

Triple-A Framework	Summary
Authentication	Authentication means validating a user's identity. The simplest form of authentication is using a username and password. In recent years, and depending on the sensitivity of an organization's information, multiple factors of authentication are used to validate a user's identity. Time based one-time passwords, access tokens, and biometric credentials are just a few ways of establishing known identity. Authentication is vital to ensure the security of business-critical information.
Authorization	Authorization specifies the access rights and privileges of a user, and are an important part of information security. Authorization determines what a user can and cannot access. Adopting least privilege limits possible leaks of information outside of your organization. New users should only have access to resources they need to access according to their job description or teammates' access. Users should not have access beyond what is required by their role. The same applies to service accounts and system users. Hijacked accounts become less of a threat when they are unable to access system vitals.
Accounting	Accounting or auditability means ensuring proper and accurate logging is in place. Sensitive data and systems should have a ledger of transactions and user access timestamps. In the event of an incident, the incident response team will then quickly be able to quarantine any related users or applications to prevent a more critical issue. The easiest and most common practice of this is within databases, where virtually all access and transaction can be logged. In the event of a breach, teams can determine exactly who accessed the data and when to isolate it and begin triage.

Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

- Code compliance to standards
- Well-documented access-control strategies, with sampled evidence of compliance
- Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
- Historical evidence of sustained practice (emails, logs, audits, meeting notes)

Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

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

Exceptions Process

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

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

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

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

Distribution

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

Policy Change Control

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

Policy Version History

Version	Date	Description	Edited By	Approved By
1.0	08/05/2020	Initial Template	David Buksbaum	
1.1	01/22/2023	Milestone	Brad Jackson	
1.2	02/12/2023	Project One	Brad Jackson	

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV