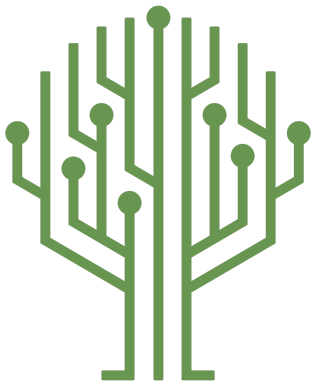


Green Pace

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	The core idea of this principle is to ensure that you always validate input data that you receive. By doing this, you can avoid general vulnerabilities from validation.
2. Heed Compiler Warnings	Make sure to always use the highest warning level you can choose in your code editor, since it is very important to finish without errors. You can fix compiler warnings on your own and/or with the help of static and dynamic analysis tubes. A simple error can result in a security flaw, so it is always important to make sure you head compiler warnings.
3. Architect and Design for Security Policies	It is important to ensure that you design your code with security principles in mind in order to keep it safe.
4. Keep It Simple	When you code, make sure that you keep the mechanisms simple, since complex algorithms can result in more errors and failed security.
5. Default Deny	Make sure that by default you deny access to everyone. This makes the code access be based on permission instead of exclusion. Special access should be the only way to permit access.
6. Adhere to the Principle of Least Privilege	The execution of anything in your code should only be permitted the permissions that are required for the job. In other words, only give things the least privilege necessary for completion. If you give elevated permissions to something that doesn't need it, it is a security vulnerability that someone might come along and try to exploit.
7. Sanitize Data Sent to Other Systems	Always make sure that when data is passed throughout systems, particularly complex ones, you sanitize the data beforehand. This ensures that someone cannot use a simple injection attack in order to access and perform commands outside of the code's intended purpose.

Principles	Write a short paragraph explaining each of the 10 principles of security.
8. Practice Defense in Depth	Defense in Depth is a wonderful principle of security that understands that each line of defense has vulnerabilities. By using multiple lines of defense, one can ensure that one of the lines of defense will stop the potential attack.
9. Use Effective Quality Assurance Techniques	Using Effective Quality Assurance techniques will increase the chance that you are able to identify and then later exterminate bugs in your code that could potentially be, or already are, security flaws. By utilizing multiple testing phases, having colleague's look over your code, and sending it to third parties to look over, you can ensure that your system is as safe as can be.
10. Adopt a Secure Coding Standard	Adopting a secure coding standard ensures that you have a baseline level of security in your code no matter how you write your code.

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.

61

Coding Standard 1

Coding Standard	Label	Name of Standard
Data Type	[STD-001-CPP]	Do not cast to an out-of-range enumeration value

Noncompliant Code

This code attempts to check if a value is within a given range of accepted enumeration values. This issue with the code is that it does this after casting to the enumeration type, which could potentially not support a given integer value. If something outside of the range was casted, then the if statement would result in unspecified behavior.

```
enum EnumType {
  First,
  Second,
  Third
};

void f(int intVar) {
  EnumType enumVar = static_cast<EnumType>(intVar);

  if (enumVar < First || enumVar > Third) {
    // Handle error
  }
}
```

Compliant Code

This solution to make the noncompliant code compliant is to first check whether the value can be represented but the enumeration type before the conversion. Thus, it restricts it to only values that have specific enumeration representations.

```
enum EnumType {
  First,
  Second,
  Third
};

void f(int intVar) {
  if (intVar < First || intVar > Third) {
    // Handle error
  }
  EnumType enumVar = static_cast<EnumType>(intVar);
}
```

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

Principles(s):
1. Validate Input Data - Ensuring valid input
2. Heed Compiler Warnings - If a compiler gives you a warning for your code, treat it seriously
3. Architect and Design for Security Policies - Ensuring proper design
9. Use Effective Quality Assurance Techniques - Quality assurance (example: tests)
10. Adopt a Secure Coding Standard - Ensuring secure code by writing it right the first time

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	7.2.0	CertC++-INT50	
CodeSonar	7.0p0	LANG.CAST.COERCE LANG.CAST.VALUE	Coercion Alters Value Cast Alters Value
Helix QAC	2022.2	C++3013	
Parasoft C/C++test	2022.1	CERT_CPP-INT50-a	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration
PRQA QA-C++	4.4	3013	
PVS-Studio	7.20	V1016	

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	[STD-002-CPP]	Do not attempt to modify string literals

Noncompliant Code

This noncompliant code has a char pointer called str that is initialized to the address of a string literal. It then attempts to modify the string literal, but it results in undefined behavior

```
char *str  = "string literal";
str[0] = 'S';
```

Compliant Code

This code uses an array initializer with a string literal. Since the code creates a copy of the strong literal in the space allocated to the character array str instead of initializing the char pointer to the address of a string literal, the string stored in str can be modified safely.

```
char str[] = "string literal";
str[0] = 'S';
```

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

Principles(s):

1. Validate Input Data - Ensuring valid input
2. Heed Compiler Warnings - If a compiler gives you a warning for your code, treat it seriously
3. Architect and Design for Security Policies - Ensuring proper design
9. Use Effective Quality Assurance Techniques - Quality assurance (example: tests)
10. Adopt a Secure Coding Standard - Ensuring secure code by writing it right the first time

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Likely	Low	P9	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	22.04	string-literal-modfication write-to-string-literal	Fully checked
Axivion Bauhaus Suite	7.2.0	CertC-STR30	Fully implemented

Tool	Version	Checker	Description Tool
Compass/ROSE			Can detect simple violations of this rule
Coverity	2017.07	PW	Deprecates conversion from a string literal to "char *"
Helix QAC	2022.2	C0556, C0752, C0753, C0754 C++3063, C++3064, C++3605, C++3606, C++3607	
Klocwork	2022.2	CERT.STR.ARG.CONST_TO_NONCONST CERT.STR.ASSIGN.CONST_TO_NONCONST	
LDRA tool suite	9.7.1	157 S	Partially implemented
Parasoft C/C++test	2022.1	CERT_C-STR30-a CERT_C-STR30-b	A string literal shall not be modified Do not modify string literals
PC-lint Plus	1.4	489, 1776	Partially supported
Polyspace Bug Finder	R2022a	CERT C: Rule STR30-C	Checks for writing to const qualified object (rule fully covered)
PRQA QA-C	9.7	0556, 0752, 0753, 0754	Partially implemented
PRQA QA-C++	4.4	3063, 3064, 3605, 3606, 3607, 3842	
PVS-Studio	7.20	V675	
RuleChecker	22.04	string-literal-modfication	Partially checked
Splint	3.1.1		
TrustInSoft Analyzer	1.38	mem_access	Exhaustively verified (see one compliant and one non-compliant example).

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	[STD-003-CPP]	Do not attempt to create a std::string from a null pointer

Noncompliant Code

This code uses std::getenv() which returns a null pointer on failure, but a string is constructed from the results of it. Thus, if there is an error, the string would be formed with the null pointer and would cause undefined behavior, which makes this noncompliant.

```
#include <cstdlib>
#include <string>

void f() {
  std::string tmp(std::getenv("TMP"));
  if (!tmp.empty()) {
    // ...
  }
}
```

Compliant Code

In order to make the above noncompliant code into compliant code, simply the results from std::getenv() are checked for the null pointer before the string object is constructed.

```
#include <cstdlib>
#include <string>

void f() {
  const char *tmpPtrVal = std::getenv("TMP");
  std::string tmp(tmpPtrVal ? tmpPtrVal : "");
  if (!tmp.empty()) {
    // ...
  }
}
```

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

Principles(s):
1. Validate Input Data - Ensuring valid input
2. Heed Compiler Warnings - If a compiler gives you a warning for your code, treat it seriously
3. Architect and Design for Security Policies - Ensuring proper design
9. Use Effective Quality Assurance Techniques - Quality assurance (example: tests)
10. Adopt a Secure Coding Standard - Ensuring secure code by writing it right the first time

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	20.10	assert_failure	
CodeSonar	7.0p0	LANG.MEM.NPD	Null Pointer Dereference
Helix QAC	2022.2	C++4770, C++4771, C++4772, C++4773, C++4774	
Klocwork	2022.2	NPD.CHECK.CALL.MIGHT NPD.CHECK.CALL.MUST NPD.CHECK.MIGHT NPD.CHECK.MUST NPD.CONST.CALL NPD.CONST.DEREF NPD.FUNC.CALL.MIGHT NPD.FUNC.CALL.MUST NPD.FUNC.MIGHT NPD.FUNC.MUST NPD.GEN.CALL.MIGHT NPD.GEN.CALL.MUST NPD.GEN.MIGHT NPD.GEN.MUST RNPD.CALL RNPD.DEREF	
Parasoft C/C++test	2022.1	CERT_CPP-STR51-a	Avoid null pointer dereferencing

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	[STD-004-CPP]	Avoid information leakage when passing a class object across a trust boundary

Noncompliant Code

This noncompliant code runs in kernel space and copies data from arg to user space. Since there are potentially padding bits within the object, copying data to the user space could reveal sensitive information within the padding bits, no matter how the data is copied.

```
#include <cstddef>

struct test {
  int a;
  char b;
  int c;
};

// Safely copy bytes to user space
extern int copy_to_user(void *dest, void *src, std::size_t size);

void do_stuff(void *usr_buf) {
  test arg{1, 2, 3};
  copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

Compliant Code

In order to make it compliant, the structure data is serialized before it is copied to anywhere else. This ensures that no uninitialized padding bits are copied to users that are not supposed to have that data. This solution packs the data, so the copy function would have to pack it again on the receiving end in order to have the original, padded structure of the data.

```
#include <cstddef>
#include <cstring>

struct test {
  int a;
  char b;
  int c;
};

// Safely copy bytes to user space.
```

Compliant Code

```
extern int copy_to_user(void *dest, void *src, std::size_t size);

void do_stuff(void *usr_buf) {
  test arg{1, 2, 3};
  // May be larger than strictly needed.
  unsigned char buf[sizeof(arg)];
  std::size_t offset = 0;

  std::memcpy(buf + offset, &arg.a, sizeof(arg.a));
  offset += sizeof(arg.a);
  std::memcpy(buf + offset, &arg.b, sizeof(arg.b));
  offset += sizeof(arg.b);
  std::memcpy(buf + offset, &arg.c, sizeof(arg.c));
  offset += sizeof(arg.c);

  copy_to_user(usr_buf, buf, offset /* size of info copied */);
}
```

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

Principles(s):

1. Validate Input Data - Ensuring valid input
3. Architect and Design for Security Policies - Ensuring proper design
5. Default Deny - Don’t give authorizations to people that don’t require it in the moment, it could lead to leaks
6. Adhere to the Principle of Least Privilege - Give least privilege to ensure a user can only access the right data
7. Sanitize Data Sent to Other Systems - Ensure that the data sent elsewhere is cleaned and authorized first
2. Heed Compiler Warnings - If a compiler gives you a warning for your code, treat it seriously
9. Use Effective Quality Assurance Techniques - Quality assurance (example: tests)
10. Adopt a Secure Coding Standard - Ensuring secure code by writing it right the first time

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++-DCL55	
CodeSonar	7.0p0	MISC.PADDING.POTB	Padding Passed Across a Trust Boundary

Tool	Version	Checker	Description Tool
Helix QAC	2022.2	C++4941, C++4942, C++4943	
Parasoft C/C++test	2022.1	CERT_CPP-DCL55-a	A pointer to a structure should not be passed to a function that can copy data to the user space

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	[STD-005-CPP]	Properly deallocate dynamically allocated resources

Noncompliant Code

In this example, a variable is passed as an expression to the “new” operator. The result of this is a pointer that is passed to ::operator delete(), which results in an 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

The solution to the noncompliant code above is to remove the call to ::operator delete(). By explicitly calling the destructor, you make the code compliant and remove the call to ::operator delete(). This is one of the few times that explicitly invoking a destructor is necessary and 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;
```

Compliant Code

```
 // ...

 s1->~S();
}
```

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

Principles(s):

2. Heed Compiler Warnings - If a compiler gives you a warning for your code, treat it seriously
3. Architect and Design for Security Policies - Ensuring proper design
9. Use Effective Quality Assurance Techniques - Quality assurance (example: tests)
10. Adopt a Secure Coding Standard - Ensuring secure code by writing it right the first time

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astrée	20.10	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	7.0p0	ALLOC.FNH ALLOC.DF ALLOC.TM ALLOC.LEAK	Free non-heap variable Double free Type mismatch Leak
Helix QAC	2022.2	C++2110, C++2111, C++2112, C++2113, C++2118, C++3337, C++3339, C++4262, C++4263, C++4264	
Klocwork	2022.2	CL.FFM.ASSIGN CL.FFM.COPY CL.FMM CL.SHALLOW.ASSIGN CL.SHALLOW.COPY FMM.MIGHT FMM.MUST FNH.MIGHT	

Tool	Version	Checker	Description Tool
		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	2022.1	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	R2022a	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	7.20	V515, V554, V611, V701, V748, V773, V1066	
SonarQube C/C++ Plugin	4.10	S1232	

Coding Standard 6

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

Noncompliant Code

This code uses assert() 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

Static assertions allow incorrect assumptions to be diagnosed at compile time instead of resulting in a silent malfunction or runtime error. Since this assertion is done at compile and not at runtime, no runtime cost in space or time is incurred. It can be used at file or block scope, and failure results in a meaningful and informative diagnostic error message.

```
#include <assert.h>

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

static_assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int),
          "Structure must not have any padding");
```

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

Principles(s):
1. Validate Input Data - Ensuring valid input
9. Use Effective Quality Assurance Techniques - Quality assurance (example: tests)
10. Adopt a Secure Coding Standard - Ensuring secure code by writing it right the first time

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	7.0p0	(customization)	Users can implement a custom check that reports uses of the assert() macro
Compass/ROSE			Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation
ECLAIR	1.2	CC2.DCL03	Fully implemented
LDRA tool suite	9.7.1	44 S	Fully implemented

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	[STD-007-CPP]	Handle all exceptions thrown before main() begins executing

Noncompliant Code

In this noncompliant code example, the S constructor may throw an exception that is not caught when globalS is constructed during program startup.

```
struct S {
  S() noexcept(false);
};

static S globalS;
```

Compliant Code

To fix this noncompliant code and turn it into compliant code, globalS is turned into a local variable with static storage duration, which allows exceptions thrown during object construction to be caught since the constructor for S will be executed for the first time when the function globalS() is called rather than at the program startup. To use this solution, the programmer would have to modify the source code so that the previous uses of globalS are replaced by a function call to globalS() instead.

```
struct S {
  S() noexcept(false);
};

S &globalS() {
  try {
    static S s;
    return s;
  } catch (...) {
    // Handle error, perhaps by logging it and gracefully terminating the application.
  }
  // Unreachable.
}
```

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

Principles(s):
2. Heed Compiler Warnings - If a compiler gives you a warning for your code, treat it seriously
3. Architect and Design for Security Policies - Ensuring proper design
8. Practice Defense in Depth - Best practice is to ensure that you leave no room for misinterpretation
9. Use Effective Quality Assurance Techniques - Quality assurance (example: tests)
10. Adopt a Secure Coding Standard - Ensuring secure code by writing it right the first time

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Likely	Low	P9	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	20.10	potentially-throwing-static-initialization	Partially checked
Axivion Bauhaus Suite	7.2.0	CertC++-ERR58	
Clang	3.9	cert-err58-cpp	Checked by clang-tidy
Helix QAC	2022.2	C++4634, C++4636, C++4637, C++4639	
Parasoft C/C++test	2022.1	CERT_CPP-ERR58-a	Exceptions shall be raised only after start-up and before termination of the program
Polyspace Bug Finder	R2022a	CERT C++: ERR58-CPP	Checks for exceptions raised during program startup (rule fully covered)
PRQA QA-C++	4.4	4634, 4636, 4637, 4639	
RuleChecker	20.10	potentially-throwing-static-initialization	Partially checked

Coding Standard 8

Coding Standard	Label	Name of Standard
Exceptions	[STD-008-CPP]	Do not let exceptions escape from destructors or deallocation functions

Noncompliant Code

In this example, a global deallocation is declared noexcept(false) and throws an exception if some conditions are not properly met. However, throwing from a deallocation function results in undefined behavior, hence the noncompliance.

```
#include <stdexcept>

bool perform_dealloc(void *);

void operator delete(void *ptr) noexcept(false) {
  if (perform_dealloc(ptr)) {
    throw std::logic_error("Something bad");
  }
}
```

Compliant Code

This compliant solution does not throw exceptions upon deallocation failure but instead just fails without throwing an exception.

```
#include <cstdlib>
#include <stdexcept>

bool perform_dealloc(void *);
void log_failure(const char *);

void operator delete(void *ptr) noexcept(true) {
  if (perform_dealloc(ptr)) {
    log_failure("Deallocation of pointer failed");
    std::exit(1); // Fail, but still call destructors
  }
}
```

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

Principles(s):
2. Heed Compiler Warnings - If a compiler gives you a warning for your code, treat it seriously
3. Architect and Design for Security Policies - Ensuring proper design
9. Use Effective Quality Assurance Techniques - Quality assurance (example: tests)
10. Adopt a Secure Coding Standard - Ensuring secure code by writing it right the first time

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Likely	Medium	P6	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	20.10	destructor-without-noexcept delete-without-noexcept	Fully checked
Axivion Bauhaus Suite	7.2.0	CertC++-DCL57	
Helix QAC	2022.2	C++2045, C++2047, C++4032, C++4631	
Klocwork	2022.2	MISRA.DTOR.THROW	
LDRA tool suite	9.7.1	453 S	Partially implemented
Parasoft C/C++test	2022.1	CERT_CPP-DCL57-a CERT_CPP-DCL57-b	Never allow an exception to be thrown from a destructor, deallocation, and swap Always catch exceptions
Polyspace Bug Finder	R2022a	CERT C++: DCL57-CPP	Checks for class destructors exiting with an exception (rule partially covered)
PVS-Studio	7.20	V509, V1045	
RuleChecker	20.10	destructor-without-noexcept delete-without-noexcept	Fully checked

Coding Standard 9

Coding Standard	Label	Name of Standard
Namespaces	[STD-009-CPP]	Do not modify the standard namespaces

Noncompliant Code

The declaration of x in namespace std results in undefined behavior.

namespace std {
int x;
}

Compliant Code

Instead of using the std namespace (since it is a reserved name), this solution places the declaration of x into a nonstandard namespace nonstd. This is assuming that the programmer's intentions were to place x into a namespace to prevent collisions with global identifiers.

namespace nonstd {
int x;
}

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

Principles(s):
3. Architect and Design for Security Policies - Ensuring proper design
4. Keep It Simple - Ensure that your code is as simple as can be, since complex code is convoluted and bug-full
10. Adopt a Secure Coding Standard - Ensuring secure code by writing it right the first time

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	
Helix QAC	2022.2	C++3180, C++3181, C++3182	

Tool	Version	Checker	Description Tool
Klocwork	2022.2	CERT.DCL.STD_NS_MODIFIED	
Parasoft C/C++test	2022.1	CERT_CPP-DCL58-a	Do not modify the standard namespaces 'std' and 'posix'
Polyspace Bug Finder	R2022a	CERT C++: DCL58-CPP	Checks for modification of standard namespaces (rule fully covered)
PRQA QA-C++	4.4	4032, 4035, 4631	
PVS-Studio	7.20	V1061	
SonarQube C/C++ Plugin	4.10	S3470	

Coding Standard 10

Coding Standard	Label	Name of Standard
Input/Output	[STD-010-CPP]	Close files when they are no longer needed

Noncompliant Code

A file object is constructed in the beginning and the constructor for std::fstream calls std::basic_filebuf<T>::open(). The default handler called by std::terminate does not call destructors, which results in the object that was opened never being 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 solution, std::fstream::close() is called before std::terminate() is called, which ensures 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()) {
```

Compliant Code

```
  // Handle error
 }
 std::terminate();
}
```

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

Principles(s):

2. Heed Compiler Warnings - If a compiler gives you a warning for your code, treat it seriously
3. Architect and Design for Security Policies - Ensuring proper design
8. Practice Defense in Depth - Best practice is to ensure that you leave no room for misinterpretation
9. Use Effective Quality Assurance Techniques - Quality assurance (example: tests)
10. Adopt a Secure Coding Standard - Ensuring secure code by writing it right the first time

Threat Level

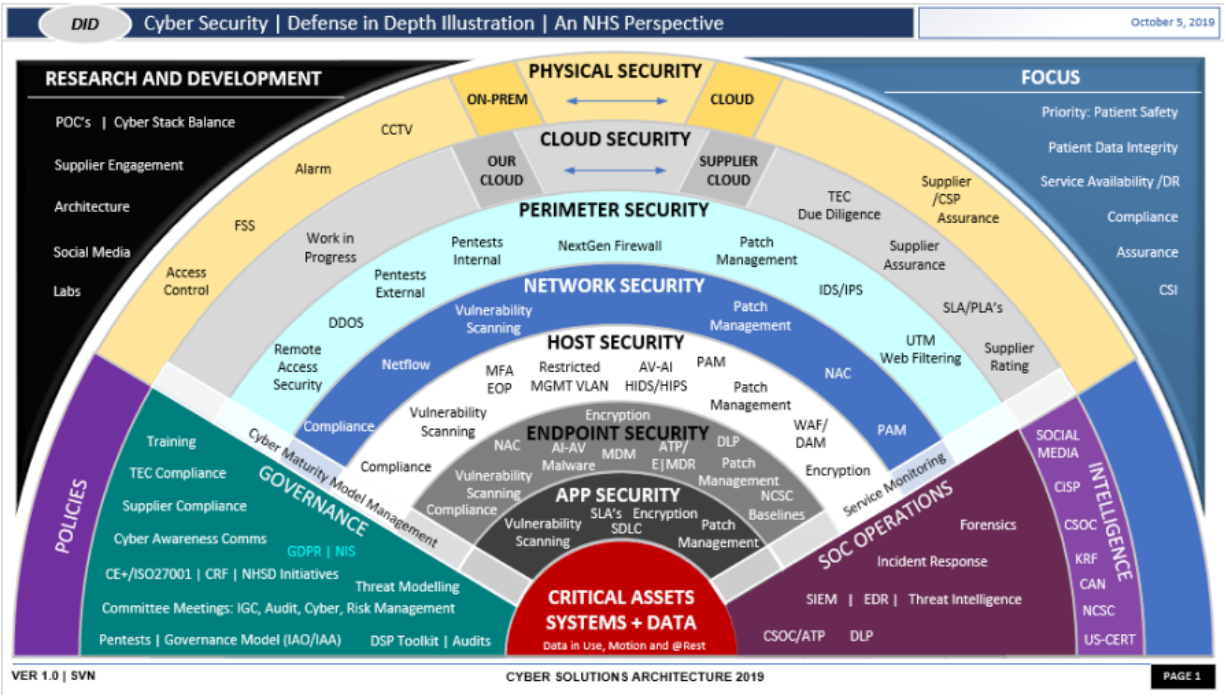
Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	7.0p0	ALLOC.LEAK	Leak
Helix QAC	2022.2	C++4786, C++4787, C++4788	
Klocwork	2022.2	RH.LEAK	
Parasoft C/C++test	2022.1	CERT_CPP-FIO51-a	Ensure resources are freed
Parasoft Insure++			Runtime detection
Polyspace Bug Finder	R2022a	CERT C++: FIO51-CPP	Checks for resource leak (rule partially 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

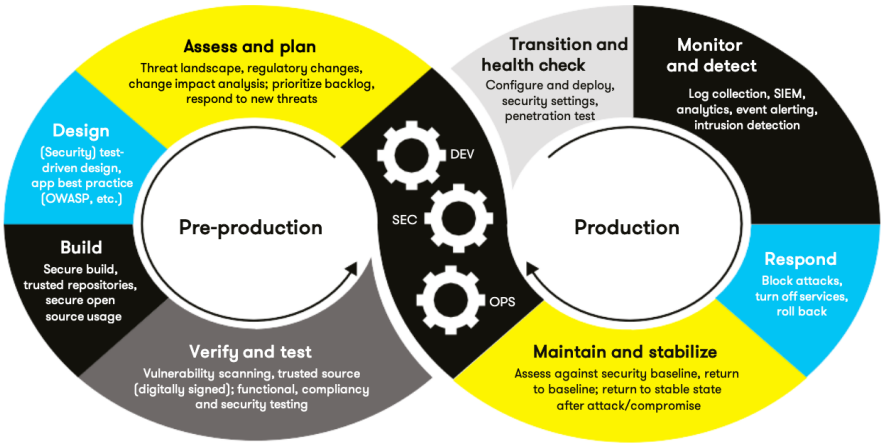
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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

Automation should be used to update and improve the existing DevOps process throughout the entirety of the process and the infrastructure, but it is just as important to include automation after the release of the product as well, since it would cut down on work loads. There is no reason that a product shouldn't be continually updating if it is feasible and as improvements are found.

Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-001-CPP	Medium	Unlikely	Medium	P4	L3
STD-002-CPP	Low	Likely	Low	P9	L2
STD-003-CPP	High	Likely	Medium	P18	L1
STD-004-CPP	Low	Unlikely	High	P1	L3
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	Low	Likely	Medium	P6	L2
STD-009-CPP	High	Unlikely	Medium	P6	L2
STD-010-CPP	Medium	Unlikely	Medium	P4	L3

Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided.

a. Explain each type of encryption, how it is used, and why and when the policy applies.
b. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

a. Encryption	**Explain what it is and how and why the policy applies.**
Encryption in rest	Encryption in rest is when you encrypt your data as it is stable in storage, then decrypting it when you need it. It is important because a hacker wouldn't be able to make sense of the data even if they had access to it because it would be encrypted.
Encryption at flight	Encryption at flight is the idea of encrypting and decrypting data as it is transmitted from one place to the next. It is important because a hacker could potentially intercept the data. By encrypting it while it is being transferred somewhere else, the hacker wouldn't be able to make sense of the data.
Encryption in use	Encryption in use is the idea of encrypting data as it is being used within the memory. This is typically done by using password protected profiles because that would protect the memory of the user as he/she is using it.

b. Triple-A Framework *	**Explain what it is and how and why the policy applies.**
Authentication	Authentication is the process that proves that someone is who they say they are, whether that be through matching usernames, passwords, tokens, IPs, or whatever else you might want to check to ensure they are who they say they are.
Authorization	After authentication, a user is given authorizations according to the permissions that are required for them to perform whatever tasks they are meant to perform. Folders, functions, and other things will be locked behind an authorization wall so that only those who have the proper authorizations will be able to access those locked files/functions/etc. You can only change the database to the degree that you are authorized to do. Depending on the level of authorization, they could also create new users and give them varying degrees of authorization.
Accounting	You should always take notes of how the authentication and authorization process went. This is the idea behind accounting. By recording who is authenticated, what they are authorized for, and then what they do with those authorizations, you can have a much safer environment for coding.

***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

This section is completed via the "Principles" section in the middle of each of the coding standard sections

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	
1.1	08/21/2022	Project One	Ayden Hooke	Ayden Hooke

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV