Green Pace

Green Pace Secure Development Policy

Jessica Giliam

February 15, 2024

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	Any system that contains data input should require input validation. By only allowing input that meets certain criteria, it becomes more difficult for an attack to occur from input that is created to cause harm. Many attacks gaining unauthorized access and theft of personal information can be prevented by having strong validation techniques. Implementation can be as simple as having a list of valid inputs and a blacklisted set of inputs.
2. Heed Compiler Warnings	Code should be compiled using the highest-level warning possible. The code should be changed to correct the warnings to avoid obvious flaws. Static and dynamic tools should also be used to detect and correct flaws prior to being deployed live.
3. Architect and Design for Security Policies	The design should be divided appropriately for optimal implementation of the security policies. If the system requires various stages of access, those stages should be separated to reflect access.
4. Keep It Simple	As a design becomes larger and more complex, so does the ability to keep it secure. By keeping the design as simple and small as it needs to be to function, security is easier to achieve.
5. Default Deny	Users or administrators with unnecessary access allow more vulnerability in the system. For this reason, all access should be denied unless otherwise given privileges. All doors remain locked unless the correct key is presented.
6. Adhere to the Principle of Least Privilege	To decrease opportunities for a proposed attacker, the code should only execute what is required to complete the task. If permission is needed to complete a task, it should only be allowed only for the time needed at that level.
7. Sanitize Data Sent to Other Systems	Data encryption is necessary to decrease the use of SQL attack, command, and other injection attacks. All data that is passed to command shells, relational databases, and commercial off the shelf components (COTS) should be sanitized.

Green Pace

Principles	Write a short paragraph explaining each of the 10 principles of security.
8. Practice Defense in Depth	Using multiple layers of protection ensures that even though no one layer is 100% secure, there are few layers that will be insecure in the same way. Even if a threat passes through one layer, it should be caught in another.
9. Use Effective Quality Assurance Techniques	Testing, audits, and independent security reviews can decrease the likelihood of persistent vulnerabilities. Even when code passes testing with no obvious signs of vulnerability, another set of eyes for review can prove to make a system more secure.
10. Adopt a Secure Coding Standard	Ensuring that a security system is well defined will determine how well it can be evaluated. Having a secure coding standard puts all contributors on the same page regarding what security means and the best ways to prevent vulnerabilities.

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

Coding Standard 1

Coding Standard	Label	Do not cast to an out-of-range enumeration value.
Data Type	[STD-001-CPP]	CPP enumerations come in two forms, scoped (fixed) and unscoped (may or may not be fixed). The range of values that can be represented by either form of enumeration may include enumerator values not specified by the enumeration itself. To avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression (SEI-CERT-C++ Coding Standard).

Noncompliant Code

Non-compliant code attempts to check whether a value (given) is inside the range of acceptable enum values. It does this after casting to the enum type, which might not be able to represent the given value (Int). On a two's complement system, the valid range of values represented by EnumType are [0..2], a value passed outside of that range would result in an unspecified type. Using within the statement will result in unspecified behavior.

```
enum EnumType {
  First,
  Second,
```

Noncompliant Code

```
  Third
};

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

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

Compliant Code

The compliant code checks that the value can be represented by the enum type first before performing the conversion. This guarantees that the result is not an unspecified value. It restricts the converted value for each specific enum value.

```
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): Input Data Validation and Data Sanitization. These principals map to the standard of not casting an out-of-range enum because serializing data out of the program or reading it into the program often does not have knowledge of the enum. It must be the correct range to prevent a user from providing/receiving undefined behavior.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool

Tool	Version	Checker	Description Tool
Astree	22.10	Cast-integer-to-enum	Partially checked
CodeSonar	8.0p0	LANG.CAST.COERCE LANG.CAST.VALUE	Coercion Alters Value Cast Alters Value
Parasoft C/C++test	2023.1	CERT_CPP-INT50-a	An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration
RuleChecker	22.10	Cast-to-integer-enum	Partially checked

Coding Standard 2

Coding Standard	Label	Ensure that operations on signed integers do not result in overflow.
Data Value	[STD-002-CPP]	Signed integer overflow is undefined behavior. It is important to ensure that operations on signed integers do not result in overflow. Implementations have variance in how they deal with signed integer overflow. An implementation that defines signed integer types as being modulo, for example, need not detect integer overflow. Implementations may also trap signed arithmetic overflows or presume that overflows will never happen. It is also possible for the same conforming implementation to emit code that behaves differently in different contexts. An implementation may determine that a signed integer loop control variable declared in a local scope cannot overflow and may emit efficient code based on that determination, while the same implementation may determine that a global variable used in a similar context will wrap.

Noncompliant Code

Noncompliant code example can result in a signed integer overflow during the addition of the signed operands si_a and si_b.

```
void func(signed int si_a, signed int si_b) {
  signed int sum = si_a + si_b;
  /* ... */
}
```

Compliant Code

Compliant code solution ensures that the addition operation cannot overflow, regardless of representation.

[Compliant code block; code should be indented using 12-point Courier New font.]

```
#include <limits.h>

void f(signed int si_a, signed int si_b) {
  signed int sum;
  if (((si_b > 0) && (si_a > (INT_MAX - si_b))) ||
      ((si_b < 0) && (si_a < (INT_MIN - si_b)))) {
    /* Handle error */
  } else {
    sum = si_a + si_b;
  }
  /* ... */
}
```

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

Principles(s): This standard maps to heed compiler warnings. With integer overflow, a user could cause intentional or unintentional security vulnerabilities. The compiler warning will help alert the developer for signed values during certain operations about possible overflow.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	P9	L2

Automation

Tool	Version	Checker	Description Tool
Astree	23.04	Integer-overflow	Fully Checked
Coverity	2017.07	TAINTED_SCALAR BAD_SHIFT	Implemented
LDRA tool suite	9.7.1	493 S, 494 S	Partially Implemented
Parasoft C/C++test	2023.1	CERT_C-INT32-a CERT_C-INT32-b CERT_C-INT32-c	Avoid signed integer overflows Integer overflow or underflow in constant expression in '+', '-', '*' operator Integer overflow or underflow in constant expression in '<<' operator.

Coding Standard 3

Coding Standard	Label	Do not confuse narrow and wide character strings and functions.
String Correctness	[STD-003-CCC]	Passing narrow string arguments to wide string functions or wide string arguments to narrow string functions can lead to unexpected and undefined behavior. Scaling problems are likely. Because wide strings are terminated by a null wide character and can contain null bytes, determining the length is also problematic. Because wchar_t and char are distinct types, many compilers will produce a warning diagnostic if an inappropriate function is used.

Noncompliant Code

Noncompliant code example incorrectly uses the strncpy() function in an attempt to copy up to 10 wide characters. Because wide characters can contain null bytes, the copy operation may end earlier than anticipated, resulting in the truncation of the wide string.

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

void func(void) {
  wchar_t wide_str1[]  = L"0123456789";
  wchar_t wide_str2[] =  L"0000000000";

  strncpy(wide_str2, wide_str1, 10);
}
```

Compliant Code

Compliant solution uses the proper-width functions. Using wcsncpy() for wide character strings and strncpy() for narrow character strings ensures that data is not truncated, and buffer overflow does not occur.

```
#include <string.h>
#include <wchar.h>

void func(void) {
  wchar_t wide_str1[] = L"0123456789";
  wchar_t wide_str2[] = L"0000000000";
  /* Use of proper-width function */
  wcsncpy(wide_str2, wide_str1, 10);

  char narrow_str1[] = "0123456789";
  char narrow_str2[] = "0000000000";
  /* Use of proper-width function */
  strncpy(narrow_str2, narrow_str1, 10);
}
```

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

Principles(s): This standard maps to adopting a coding standard. The idea of wide character strings in functions that are not anticipating them result in a bug and is not best practice. This should be a high priority for code reviewers.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Low	P27	L1

Automation

Tool	Version	Checker	Description Tool
Astree	23.04	Wide-narrow-string-cast Wide-narrow-string-cast-implicit	Partially checked
Axivion Bauhaus Suite	7.2.0	CertC-STR38	Fully implemented
CodeSonar	8.0p0	LANG.MEM.BO LANG.MEM.TBA	Buffer overrun tainted buffer access
Coverity	2017.07	PW	Implemented

Coding Standard 4

Coding Standard	Label	Sanitize untrusted data passed across a trust boundary.
SQL Injection	[STD-004-CCC]	A SQL injection vulnerability arises when the original SQL query can be altered to form an altogether different query. Execution of this altered query may result in information leaks or data modification. The primary means of preventing SQL injection are sanitizing, validating untrusted input, and parameterizing queries.

Noncompliant Code

Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. 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 whitelisting 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.

```
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 = '_';
}
```

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

Principles(s): This standard maps to sanitizing data sent to other systems. Not having sanitized data could result in SQL injection attacks.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	High	P1	L3

Automation

Tool	Version	Checker	Description Tool

Tool	Version	Checker	Description Tool
Astree	23.04	Function-argument-with-padding	Partially checked
Axivion Bauhaus Suite	7.2.0	CertC-DCL39	Detects composite structures with padding, particularly those passed to trust boundary routines.
CodeSonar	8.0p0	MISC.PADDING.POTB	Padding passed across a trust boundary.
Helix QAC	2023.4	DF4941, DF4942, DF4943	Fully implemented.

Coding Standard 5

Coding Standard	Label	Do not access freed memory.
Memory Protection	[STD-005-CPP]	Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable vulnerabilities.

Noncompliant Code

Noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems.

```
#include <new>

struct S {
  void f();
};

void g() noexcept(false) {
  S *s = new S;
  // ...
  delete s;
  // ...
  s->f();
}
```

Compliant Code

In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required.

```
#include <new>

struct S {
  void f();
};

void g() noexcept(false) {
  S *s = new S;
  // ...
  s->f();
```

Compliant Code

```
  delete s;
}
```

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

Principles(s): This standard maps to adopting a secure coding standard. Secure coding standard best practices should include what memory is freed during code review.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astree	23.04	Dangling_pointer_use	Supported. Astree reports all accesses to freed allocated memory.
Axivion Bauhaus Suite	7.2.0	CertC-MEM30	Detects memory accesses after its deallocation and double memory deallocations.
CodeSonar	8.0p0	ALLOC.UAF	Use after free.
LDRA tool suite	9.7.1	51 D, 484 S, 112 D	Partially implemented.

Coding Standard 6

Coding Standard	Label	Understand the termination behavior of assert() and abort().
Assertions	[STD-006-CCC]	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

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

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 */
  }

  /* ... */
```

Compliant Code

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

  /* ... */
}
```

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

Principles(s): This standard also maps to the adopting of a secure coding standard principle. This establishes best practices for handling program termination to ensure proper process during shut down. Nothing should be left in a vulnerable state.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Compass/ROSE	N/A	N/A	Can detect some violations of this rule. 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	Guarantee exception safety.
Exceptions	[STD-007-CPP]	Proper handling of errors and exceptional situations is essential for the continued correct operation of software. The preferred mechanism for reporting errors in a C++ program is exceptions rather than error codes. A number of core language facilities, including dynamic_cast, operator new(), and typeid, report failures by throwing exceptions. In addition, the C++ standard library makes heavy use of exceptions to report several different kinds of failures. Few C++ programs manage to avoid using some of these facilities. Most C++ programs must be prepared for exceptions to occur and must handle each appropriately.

Noncompliant Code

Noncompliant code example shows a flawed copy assignment operator. The implicit invariants of the class are that the array member is a valid (possibly null) pointer and that the nElems member stores the number of elements in the array pointed to by array. The function deallocates array and assigns the element counter, nElems, before allocating a new block of memory for the copy. As a result, if the new expression throws an exception, the function will have modified the state of both member variables in a way that violates the implicit invariants of the class. Consequently, such an object is in an indeterminate state and any operation on it, including its destruction, results in undefined behavior.

```
#include <cstring>

class IntArray {
  int *array;
  std::size_t nElems;
public:
  // ...

  ~IntArray() {
    delete[] array;
  }


  IntArray(const IntArray& that); // nontrivial copy constructor
  IntArray& operator=(const IntArray &rhs) {
    if (this != &rhs) {
      delete[] array;
      array = nullptr;
      nElems = rhs.nElems;
      if (nElems) {
        array = new int[nElems];
        std::memcpy(array, rhs.array, nElems * sizeof(*array));
      }
    }
```

Noncompliant Code

```
    return *this;
  }

  // ...
};
```

Compliant Code

The copy assignment operator provides a strong exception safety guarantee. The function allocates new storage for the copy before changing the state of the object. Only after the allocation succeeds does the function proceed to change the state of the object. In addition, by copying the array to the newly allocated storage before deallocating the existing array, the function avoids the test for self-assignment, which improves the performance

```
#include <cstring>

class IntArray {
  int *array;
  std::size_t nElems;
public:
  // ...

  ~IntArray() {
    delete[] array;
  }

  IntArray(const IntArray& that); // nontrivial copy constructor

  IntArray& operator=(const IntArray &rhs) {
    int *tmp = nullptr;
    if (rhs.nElems) {
      tmp = new int[rhs.nElems];
      std::memcpy(tmp, rhs.array, rhs.nElems * sizeof(*array));
    }
    delete[] array;
    array = tmp;
    nElems = rhs.nElems;
    return *this;
  }

  // ...
};
```

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

Principles(s): As being one of the most important principles, this standard also maps to the adopting of a secure coding standard. By coding according to these standards, proper error handling ensures there are no unexpected or unhandled errors.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	P9	L2

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.0p0	ALLOC.LEAK	Leak
LDRA tool suite	9.7.1	527 S, 56 D, 71 D	Partially implemented
Parasoft C/C++test	2023.1	CERT_CPP-ERR56-a CERT_CPP-ERR56-b	Always catch exceptions. Do not leave 'catch' blocks empty.
Polyspace Bug Finder	R2023b	CERT C++: ERR56-CPP	Checks for exceptions violating class invariant (rule fully covered).

Coding Standard 8

Coding Standard	Label	Do not invoke virtual functions from constructors or destructors.
Object Oriented Programming	[STD-008-CPP]	Do not directly or indirectly invoke a virtual function from a constructor or destructor that attempts to call into the object under construction or destruction. Because the order of construction starts with base classes and moves to more derived classes, attempting to call a derived class function from a base class under construction is dangerous. The derived class has not had the opportunity to initialize its resources, which is why calling a virtual function from a constructor does not result in a call to a function in a more derived class. Similarly, an object is destroyed in reverse order from construction, so attempting to call a function in a more derived class from a destructor may access resources that have already been released.

Noncompliant Code

The base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the B::B() constructor calls B::seize() rather than D::seize(). Likewise, the B::~B() destructor calls B::release() rather than D::release().

```
struct B {
  B() { seize(); }
  virtual ~B() { release(); }

protected:
  virtual void seize();
  virtual void release();
};

struct D : B {
  virtual ~D() = default;

protected:
  void seize() override {
    B::seize();
    // Get derived resources...
  }

  void release() override {
    // Release derived resources...
    B::release();
  }
};
```

Compliant Code

Compliant Code

The constructors and destructors call a nonvirtual, private member function (suffixed with mine) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources.

```
class B {
  void seize_mine();
  void release_mine();

public:
  B() { seize_mine(); }
  virtual ~B() { release_mine(); }

protected:
  virtual void seize() { seize_mine(); }
  virtual void release() { release_mine(); }
};

class D : public B {
  void seize_mine();
  void release_mine();

public:
  D() { seize_mine(); }
  virtual ~D() { release_mine(); }

protected:
  void seize() override {
    B::seize();
    seize_mine();
  }

  void release() override {
    release_mine();
    B::release();
  }
};
```

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

Principles(s): This standard maps to adopting of a secure coding standard. This is best practice and would result in more secure development and help prevent undefined behaviors.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	P2	L3

Automation

Tool	Version	Checker	Description Tool
Astree	22.10	Virtual-call-in-constructor Invalid_function_pointer	Fully checked
Clang	3.9	Clang-analyzer-alpha.cplusplus.VirtualCall	Checked by clang-tidy
CodeSonar	8.0p0	LANG.STRUC.VCALL_IN_CTOR LANG.STRUCT.VCALL_IN_DTOR	Virtual Call in Constructor Virtual Call in Destructor
LDRA tool suite	9.7.1	467 S, 92 D	Fully implemented

Coding Standard 9

Coding Standard	Label	Use valid references, pointers, and iterators to reference elements of a container.
Containers	[STD-009-CPP]	The C++ Standard allows references and pointers to be invalidated independently for the same operation, which may result in an invalidated reference but not an invalidated pointer. Relying on this distinction is insecure because the object pointed to by the pointer may be different than expected even if the pointer is valid. It is possible to retrieve a pointer to an element from a container, erase that element (invalidating references when destroying the underlying object), then insert a new element at the same location within the container causing the extant pointer to now point to a valid, but distinct object. Thus, any operation that invalidates a pointer or a reference should be treated as though it invalidates both pointers and references.

Noncompliant Code

In this noncompliant code example, pos is invalidated after the first call to insert(), and subsequent loop iterations have undefined behavior.

```
#include <deque>

void f(const double *items, std::size_t count) {
  std::deque<double> d;
  auto pos = d.begin();
  for (std::size_t i = 0; i < count; ++i, ++pos) {
    d.insert(pos, items[i] + 41.0);
  }
}
```

Compliant Code

In the compliant solution, pos is assigned a valid iterator on each insertion, preventing undefined behavior.

```
#include <deque>

void f(const double *items, std::size_t count) {
  std::deque<double> d;
  auto pos = d.begin();
  for (std::size_t i = 0; i < count; ++i, ++pos) {
    pos = d.insert(pos, items[i] + 41.0);
  }
}
```

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

Principles(s): Another standard that maps to adopting a secure coding standard principle that prevents memory-oriented bugs that result in further security issues.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	High	P6	L2

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.0p0	ALLOC.UAF	Use after free
Astree	22.10	Overflow_upon_dereference	N/A
Klocwork	2023.3	ITER.CONTAINER.MODIFIED	N/A
Parasoft C/C++test	2023.1	CERT_CPP-CTR51-a	Do not modify container while iterating over it.

Coding Standard 10

Coding Standard	Label	Do not dereference null pointers.
Expressions	[STD-010-CPP]	Dereferencing a null pointer is undefined behavior. On many platforms, dereferencing a null pointer results in abnormal program termination, but this is not required by the standard.

Noncompliant Code

Noncompliant code example is derived from a real-world example taken from a vulnerable version of the libpng library as deployed on a popular ARM-based cell phone [Jack 2007]. The libpng library allows applications to read, create, and manipulate PNG (Portable Network Graphics) raster image files. The libpng library implements its own wrapper to malloc() that returns a null pointer on error or on being passed a 0-byte-length argument.
This code also violates ERR33-C. Detect and handle standard library errors.

```
#include <png.h> /* From libpng */
#include <string.h>

void func(png_structp png_ptr, int length, const void *user_data) {
  png_charp chunkdata;
  chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
  /* ... */
  memcpy(chunkdata, user_data, length);
  /* ... */
 }
```

Compliant Code

This compliant solution ensures that the pointer returned by png_malloc() is not null. It also uses the unsigned type size_t to pass the length parameter, ensuring that negative values are not passed to func().
This solution also ensures that the user_data pointer is not null. Passing a null pointer to memcpy() would produce undefined behavior, even if the number of bytes to copy were 0. The user_data pointer could be invalid in other ways, such as pointing to freed memory. There is no portable way to verify that the pointer is valid, other than checking for null.

```
#include <png.h> /* From libpng */
#include <string.h>

 void func(png_structp png_ptr, size_t length, const void *user_data) {
  png_charp chunkdata;
  if (length == SIZE_MAX) {
    /* Handle error */
  }
  if (NULL == user_data) {
    /* Handle error */
  }
```

Compliant Code

```
  chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
  if (NULL == chunkdata) {
    /* Handle error */
  }
  /* ... */
  memcpy(chunkdata, user_data, length);
  /* ... */

 }
```

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

Principles(s): This also maps to secure coding practice and prevents crashing the program. This is not only a secure standard, but a regular coding best practice.

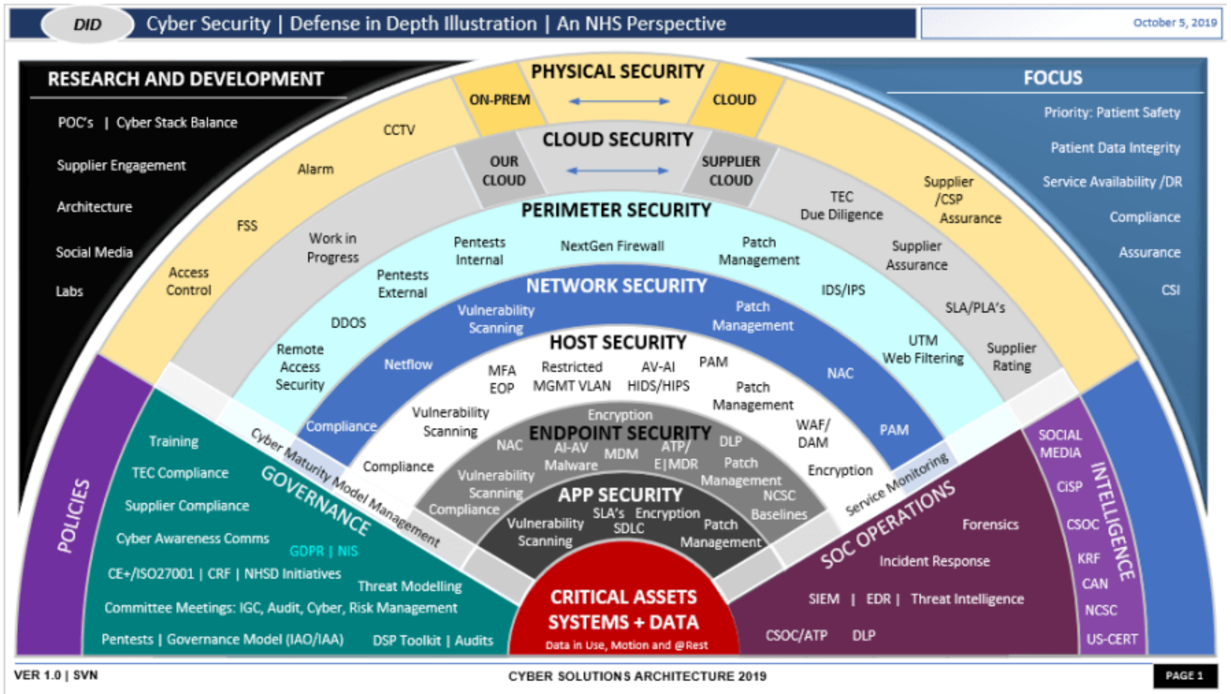
Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astree	23.04	Null-dereferencing	Fully checked
Axivion Bauhaus Suite	7.2.0	CertC-EXP34	N/A
CodeSonar	8.0p0	LANG.MEM.NPD LAND.STRUCT.NTAD LANG.STRUCT.UPD	Null pointer dereference Null test after dereference Unchecked parameter dereference
Helix QAC	2023.4	DF2810, DF2811, DF2812, DF2813	Fully implemented.

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

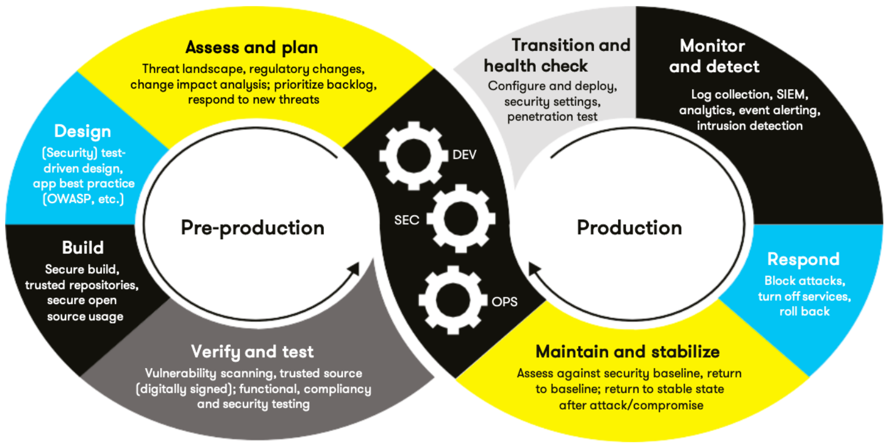
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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


Assess and plan
Threat landscape, regulatory changes, change impact analysis; prioritize backlog, respond to new threats
Design
(Security) test-driven design, app best practice (OWASP, etc.)
Build
Secure build, trusted repositories, secure open source usage
Verify and test
Vulnerability scanning, trusted source (digitally signed); functional, compliancy and security testing
Pre-production
DEV
SEC
OPS
Transition and health check
Configure and deploy, security settings, penetration test
Monitor and detect
Log collection, SIEM, analytics, event alerting, intrusion detection
Production
Respond
Block attacks, turn off services, roll back
Maintain and stabilize
Assess against security baseline, return to baseline; return to stable state after attack/compromise

Automation will be used for the enforcement of and compliance with the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

[Insert your written explanations here.]

Summary of Risk Assessments

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

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-001-CPP	Medium	Unlikely	Medium	P4	L3
[STD-002-CPP]	High	Unlikely	High	P9	L2
[STD-003-CPP]	High	Likely	Low	P27	L1
[STD-004-CPP]	Low	Unlikely	High	P1	L3
[STD-005-CPP] [Insert text.]	High	Likely	Medium	P18	L1
[STD-006-CPP]	Medium	Unlikely	Medium	P4	L3
[STD-007-CPP]	High	Likely	High	P9	L2
[STD-008-CPP]	Low	Unlikely	Medium	P2	L3
[STD-009-CPP]	High	Probable	High	P6	L2
[STD-010-	High	Likely	Medium	P18	L1

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CPP]					

Create Policies for Encryption and Triple A

Include all three types of encryptions (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 they are, how they should be applied in practice, and why they should be used.

a. Encryption	**Explain what it is and how and why the policy applies.**
Encryption in rest	Protect data stored on disk, back up media, and solid-state drives as part of a broader security strategy. Provides privacy for users, giving limited access to systems, and leaves data protected even if data is stolen. Attackers cannot understand or decrypt information.
Encryption at flight	Protect data in transit and whenever engaged with a cloud resource. TLS to transmit traffic (both symmetric and asymmetric) and HTTPS as a widely accepted and used protocol for most traffic. Initiates secure communication.
Encryption in use	The “newest” encryption technique to protect data while being used protecting data from insecure end points and other vulnerabilities. Data should never be left unsecure.

b. Triple-A Framework*	**AAA Framework controls access to resources, audits and monitors usage, and enforces policies. Tracking activities and screening users for network management ensure an additional layer of security.**
Authentication	Users must present log in credentials and confirm their ID with username, password or other two factor authentication tools such as a pin number sent to email, phone number, etc. New Users must be identified and have stored credentials.
Authorization	If “Deny by Default” is being used accordingly, authorization decides which access goes to the user. Not all users have the same privileges. Prevents changes to database by unauthorized indivuduals and determines their level of access.
Accounting	Track activity by all users while logged in. Analyze trends, time logs, etc. Essential for ensuring secure working environments, show what users are accessing what files, etc.

*****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 standard and justify the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it is time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is best practice.

NOTE: Green Pace has already successfully implemented the following:

- Operating system logs
- Firewall logs
- Anti-malware logs

All three of these logs are standards connected to the Defense in Depth principle. Logging allows monitoring at each layer (OS, Firewall, anti-malware) so that if an attack makes it through one layer it can be possibly stopped at the others. Abnormal or unexpected usage patterns such as log ins on systems that should not have log ins, established connections that should not exist, errors on firewall logs all can be caught. Another principle that connects to these is the architect and design for security policies, as these stages are separated and reflect access for optimal implementation.

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 comply with this policy.

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

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

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

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

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

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

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

Policy Version History

Version	Date	Description	Edited By	Approved By
1.0	08/05/2020	Initial Template	David Buksbaum	
2.0	02/18/2024	Security Policies Created	Jessica Giliam	
[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]	[Insert text.]

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV