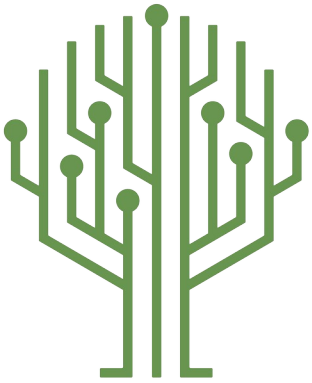


Green Pace

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	This is used to prevent malicious attacks like SQL injection by checking and cleaning any externally received data. Incoming data should be cleaned of any malicious characters and verified to meet the expected parameters for the input stream including data type and format. Thoroughly validate any input regardless of source to maximize the use of this important security principle.
2. Heed Compiler Warnings	Compiler warnings exist for a reason, and that reason is to help you write code that better adheres to the coding standard. These warnings do not mean your code does not run and seems to work the way you expect, but addressing the generated warnings helps create strong, secure code. It is often small exploits that get overlooked and then get turned into a big problem, use this security principle to minimize that possibility.
3. Architect and Design for Security Policies	Security should exist as an integral part of every stage of the software development life cycle. At the birth of the project idea security should be brainstormed as the project ideas are fleshed out. Then when the code base is started it will already be full of secure principles that make elaboration easy. It should be remembered however that security is not a one time event, new vulnerabilities must be monitored for and addressed as necessary to remain in full compliance with this principle.
4. Keep It Simple	Easy to read codebases are easy to understand and manipulate even far down the road when the company may have had a full rotation of employees or changed hands or simply wanted to update some legacy code. Simple, easy to maintain code means that it is easier to update when new vulnerabilities need to be addressed This means this principle is used to promote security protocols by utilizing the programming principles of readability and maintainability.

Principles	Write a short paragraph explaining each of the 10 principles of security.
5. Default Deny	Default Deny protects a system from unauthorized users when configured appropriately. Accessing system wide resources and controls should not be allowed for the majority of users, not even the ability to view certain settings. By default, ideally, a guest can access almost nothing, and certainly should not be able to make edits, meanwhile, a standard user should have access to their personal settings, but not sitewide options, an admin should have access to all relevant settings, but only the root user should have access to everything. External tools, such as firewalls, can be used to assist the effectiveness of this principle.
6. Adhere to the Principle of Least Privilege	Controlling user permissions is an easy way to add a layer of security to any system. Adhering to the principle of least privilege means ensuring that users only have access to absolutely necessary privileges within the system to complete the actions they need. Admin privileges should only belong to the users specifically designated as administrators and even then when providing modify permissions only the areas relevant to the admin's job are necessary. For example, the lead software engineer is unlikely to need to be able to manage and edit user accounts but will need to be able to manipulate various deployment settings. Use this security principle to minimize both intentional and accidental malicious activity.
7. Sanitize Data Sent to Other Systems	Cleaning data being transferred is as equally important to maintaining security as validating input is. Taking the time to ensure any outgoing data matches the expected data types on both sides and does not contain characters that could create a vulnerability minimizes the possibility of XSS or SQL Injection. Creating a white list of valid characters in combination with input validation is a good start to data safety, an additional layer of encoding should also be added to maximize the benefits of this security principle.
8. Practice Defense in Depth	Defense in Depth (DiD) involves using various layers of secure options to protect against malicious intent. Layering the concept of secure policy principles is a good way to practice DiD, options like using both firewalls and antivirus decrease the chances of a data break more effectively than only one. Practicing appropriate data management, encryption, system/permission access, and other security best practices in combination fulfills the expectations of a DiD adherent system.
9. Use Effective Quality Assurance Techniques	Testing and QA are other techniques that should be practiced throughout the entire lifecycle of any project or system. This includes testing the security of code and the associated system, penetration testing should occur as well. Ideally, vulnerabilities or potential vulnerabilities will be neutralized before deployment. Post deployment the system should be subject to regular review and security audits to address any newly discovered possible avenues of attack or exploitation.
10. Adopt a Secure Coding Standard	Actively creating secure code from the start can decrease the possibility of common vulnerabilities, practicing this habit increases the programming skills of a developer and reduces maintenance workload. Code should adhere to security

Principles	Write a short paragraph explaining each of the 10 principles of security.
	best practices in addition to being simple to understand to minimize issues in future reviews and a developed dedicated to secure development should stay up to date on the current standards.

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

Source: https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow

Coding Standard	Label	Ensure that operations on signed integers do not result in overflow
Data Type	[STD-001-CPP]	Practicing type safety is an important factor in preventing unauthorized access and unexpected behavior which can lead to vulnerabilities. Selecting the appropriate data type is always important, SEI CERT INT32-C states that signed integer overflow notes that performing an operation on an integer can cause unexpected behavior.

Noncompliant Code

This code demonstrates a multiplication operation between two integers without checking for overflow.

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

Compliant Code

This code handles overflow by using assert to check if the resulting product can be represented as a 32bit integer and returns an error if not.

```
#include <stddef.h>
#include <assert.h>
#include <limits.h>
#include <inttypes.h>

extern size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)

void func(signed int si_a, signed int si_b) {
  signed int result;
  signed long long tmp;
  assert(PRECISION(ULLONG_MAX) >= 2 * PRECISION(UINT_MAX));
  tmp = (signed long long)si_a * (signed long long)si_b;

 /*
  * If the product cannot be represented as a 32-bit integer,
  * handle as an error condition.
  */
 if ((tmp > INT_MAX) || (tmp < INT_MIN)) {
```

Compliant Code

```
  /* Handle error */
 } else {
  result = (int)tmp;
 }
 /* ... */
}
```

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

Principles(s): 1 - *Validate Input Data* – Ensuring that received input meets all the requirements and limitations of the input specification, including size/length limitations for the assigned data type (and any expected data type resulting from arithmetic operations as well) will prevent this type of overflow.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	P9	L2

Automation

Tool	Version	Checker	Description Tool
Astree	24.04	integer-overflow	Fully checked
CodeSonar	8.3p0	ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW	Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size
Coverity	2017.07	TAINTED_SCALAR BAD_SHIFT	Implemented
Cppcheck Premium	24.11.0	premium-cert-int32-c	

Coding Standard 2

Source: https://wiki.sei.cmu.edu/confluence/display/c/DCL40-C.+Do+not+create+incompatible+declarations+of+the+same+function+or+object

Coding Standard	Label	Do not create incompatible declarations of the same function or object
Data Value	[STD-002-CPP]	Data should be handled correctly throughout different parts of code, this prevents undefined behavior caused by multiple declarations of an object with different data types, this is addressed by SEI CERT DCL40-C.

Noncompliant Code

In the following code, a variable is declared first as a pointer and then in a later file used as an array, the two are incompatible for manipulating together.

```
/* In a.c */
extern int *a;   /* UB 14 */

int f(unsigned int i, int x) {
  int tmp = a[i];   /* UB 36: read access */
  a[i] = x;         /* UB 36: write access */
  return tmp;
}

/* In b.c */
int a[] = { 1, 2, 3, 4 };   /* UB 14 */
```

Compliant Code

This code declares the variable as an array in the original file so as to prevent undefined behavior later when implemented.

```
/* In a.c */
extern int a[];

int f(unsigned int i, int x) {
  int tmp = a[i];
  a[i] = x;
  return tmp;
}

/* In b.c */
int a[] = { 1, 2, 3, 4 };
```

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

Principles(s): 4 - *Keep it Simple* – Keeping code small and simple increases maintainability and decreases the possibility of even accidentally switching around data types of the same function or object.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Medium	P2	L3

Automation

Tool	Version	Checker	Description Tool
Astree	24.04	type-compatibility type-compatibility-link distinct-extern	Fully checked
Axivion Bauhaus Suite	7.2.0	CertC-DCL40	Fully implemented
CodeSonar	8.3p0	LANG.STRUCT.DECL.IF LANG.STRUCT.DECL.IO	Inconsistent function declarations Inconsistent object declarations
Cppcheck Premium	24.11.0	premium-cert-dcl40-c	

Coding Standard 3

Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator

Coding Standard	Label	Guarantee that storage for strings has sufficient space for character data and the null terminator
String Correctness	[STD-003-CPP]	Buffer overflow is a frequent issue when manipulating strings, SEI CERT STR50-CPP recommends preventing this by ensuring that any place a string is being copied to has enough space to hold the string being copied to it.

Noncompliant Code

The following code takes an unbounded input on a bounded char array which will overflow if the input is longer than the array limit.

```
#include <iostream>

void f() {
  char buf[12];
  std::cin >> buf;
}
```

Compliant Code

Utilizing std::string instead of bounded char array will prevent buffer overflows in this instance.

```
#include <iostream>
#include <string>

void f() {
  std::string input;
  std::string stringOne, stringTwo;
  std::cin >> stringOne >> stringTwo;
}
```

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

Principles(s): 1 - *Validate Input Data* – Minimize the possibility of buffer overflow by validating any accepted input meets the necessary requirements and limitations of the data types involved with any part of the input transaction.
10 – *Adopt a Secure Coding Standard* – String manipulation is a common weakness point and it should be a core concern from the start of any project.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astree	22.10	stream-input-char-array	Partially checked + soundly supported
CodeSonar	8.3p0	MISC.MEM.NTERM LANG.MEM.BO LANG.MEM.TO	No space for null terminator Buffer overrun Type overrun
Helix QAC	2024.4	C++5216 DF2835, DF2836, DF2839,	[Insert text.]
Klocwork	2024.4	NNTS.MIGHT NNTS.TAINTED NNTS.MUST SV.UNBOUND_STRING_INPUT.CIN	[Insert text.]

Coding Standard 4

Source: https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings

Coding Standard	Label	Exclude user input from format strings
SQL Injection	[STD-004-CPP]	SQL injection happens when an attacker is able to insert commands into a field that performs operations on a database, such as signing in to an account. Following the best practice of excluding direct user input from formatted strings helps prevent SQL Injection.

Noncompliant Code

The function in this example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by user, which is untrusted data.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void incorrect_password(const char *user) {
  int ret;
  /* User names are restricted to 256 or fewer characters */
  static const char msg_format[] = "%s cannot be authenticated.\n";
  size_t len = strlen(user) + sizeof(msg_format);
  char *msg = (char *)malloc(len);
  if (msg == NULL) {
    /* Handle error */
  }
  ret = snprintf(msg, len, msg_format, user);
  if (ret < 0) {
    /* Handle error */
  } else if (ret >= len) {
    /* Handle truncated output */
  }
  fprintf(stderr, msg);
  free(msg);
}
```

Compliant Code

This code instead passes the user input as an argument to fprintf() instead of part of the format string to eliminate the possibility of causing that vulnerability.

Compliant Code

```
#include <stdio.h>

void incorrect_password(const char *user) {
  static const char msg_format[] = "%s cannot be authenticated.\n";
  fprintf(stderr, msg_format, user);
}
```

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

Principles(s): 1 - *Validate Input Data* – Validate all data input by users to only take acceptable characters and be sure not to pass it directly into anything such as format strings prior to sanitization.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Cppcheck Premium	24.11.0	Premium-cert-fio30-c	Can detect violations of this rule when the -Wformat-security flag is used
Axivion Bauhaus Suite	7.2.0	CertC-FIO30	Partially implemented
CodeSonar	8.3p0	IO.INJ.FMT MISC.FMT	Format string injection Format string
Coverity	2017.07	TAINTED_STRING	Implemented

Coding Standard 5

Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory

Coding Standard	Label	Do not access freed memory
Memory Protection	[STD-005-CPP]	Dangling pointers are pointers that point to memory that has been deallocated already can result in undefined behavior that leads to vulnerabilities. Memory should not be written to or accessed once it has been freed.

Noncompliant Code

The following code shows dynamically allocated memory being accessed after deallocation.

```
#include <iostream>
#include <memory>
#include <cstring>

int main(int argc, const char *argv[]) {
  const char *s = "";
  if (argc > 1) {
    enum { BufferSize = 32 };
    try {
      std::unique_ptr<char[]> buff(new char[BufferSize]);
      std::memset(buff.get(), 0, BufferSize);
      // ...
      s = std::strncpy(buff.get(), argv[1], BufferSize - 1);
    } catch (std::bad_alloc &) {
      // Handle error
    }
  }

  std::cout << s << std::endl;
}
```

Compliant Code

The following code shows how std::string could be used instead to improve the security, and provide a simpler implementation.

```
#include <iostream>
#include <string>

int main(int argc, const char *argv[]) {
```

Compliant Code

```
 std::string str;

 if (argc > 1) {
   str = argv[1];
 }

 std::cout << str << std::endl;
}
```

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

Principles(s): 10 - *Adopt a Secure Coding Standard* – Make it a point to program securely, including handling pointers appropriately from the beginning of the project, this helps prevent unexpected behavior and denial of service attacks.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Clang	3.9	clang-analyzer-cplusplus.NewDelete clang-analyzer-alpha.security.ArrayBoundV2	Checked by clang-tidy, but does not catch all violations of this rule.
CodeSonar	8.3p0	ALLOC.UAF	Use after free
Coverity	V7.5.0	USE_AFTER_FREE	Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer
LDRA tool suite	9.7.1	483 S, 484 S	Partially implemented

Coding Standard 6

Source: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152296

Coding Standard	Label	Understand the termination behavior of assert() and abort()
Assertions	[STD-006-CPP]	Since assert() calls abort(), any cleanup function using atexit() will not run, in the event a failed assertion needs to be cleaned up static assertions should be used where possible.

Noncompliant Code

This code shows an assert without appropriate cleanup handling if the assert were to fail.

```
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

This code uses an if statement to call cleanup instead if something goes wrong.

```
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): 9 - *Use Effective Quality Assurance Techniques* – Assertions are for testing and being familiar with the expected behavior and appropriate use of termination calls for assertions is vital to quality assurance.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astree	24.04	bad-function bad-macro-use	Supported
LDRA tool suite	9.7.1	44 S	Enhanced enforcement
Parasoft C/C++ test	2024.2	CERT_C-ERR06-a	Do not use assertions
PC-lint Plus	1.4	586	Fully Supported

Coding Standard 7

Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions

Coding Standard	Label	Handle all exceptions
Exceptions	[STD-007-CPP]	If an exception is thrown and there are no assigned matching handlers then the program expands the search into sounding code within the thread in an attempt to find one. To prevent this all exceptions should be caught by an appropriate handler, even if the exception cannot be recovered from.

Noncompliant Code

In this code the thread entry point does not catch any exceptions throwing.

```
#include <thread>

void throwing_func() noexcept(false);

void thread_start() {
  throwing_func();
}

void f() {
  std::thread t(thread_start);
  t.join();
}
```

Compliant Code

Using a try...catch block eliminates this issue.

```
#include <thread>

void throwing_func() noexcept(false);

void thread_start(void) {
  try {
    throwing_func();
  } catch (...) {
    // Handle error
  }
}

void f() {
```

Compliant Code

```
 std::thread t(thread_start);
 t.join();
}
```

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

Principles(s): 9 - *Use Effective Quality Assurance Techniques* – Ensuring that exceptions are handled gracefully is an important QA technique for preventing unexpected behavior or DoS attacks, increasing code maintainability, and simplifying debugging.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probable	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astree	22.10	main-function-catch-all early-catch-all	Partially checked
CodeSonar	8.3p0	LANG.STRUCT.UCTCH	Unreachable Catch
LDRA tool suite	9.7.1	527 S	Partially implemented
Parasoft C/C++ test	2024.2	CERT_CPP-ERR51-a CERT_CPP-ERR51-b	Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point

Coding Standard 8

Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/MSC51-CPP.+Ensure+your+random+number+generator+is+properly+seeded

Coding Standard	Label	Ensure your random number generator is properly seeded
Data - RNG	[STD-008-CPP]	If random number generators always use the same seed they will always produce the same set of numbers. An attacker can use this knowledge to predict future numbers and gain access to information or processes for malicious intent. Properly seeding a random number generator is important unless a true random number generator is used.

Noncompliant Code

This code shows an RNG being used with the default seed, this code will always get the same sequence of numbers.

```
#include <random>
#include <iostream>

void f() {
  std::mt19937 engine;

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

Compliant Code

This code uses random_device to generate a random seed every, this way ever time the code is ran a different sequence will be returned

```
#include <random>
#include <iostream>

void f() {
  std::random_device dev;
  std::mt19937 engine(dev());

  for (int i = 0; i < 10; ++i) {
    std::cout << engine() << ", ";
  }
}
```

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

Principles(s): 3 - *Architect and Design for Security Policies* – Any system utilizing a random number generator should consider the appropriate security methods necessary from the start to prevent security vulnerabilities caused by an attacker being able to guess the generated number.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Likely	Low	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astree	22.10	Default-construction	Partially checked
CodeSonar	8.3p0	HARDCODED.SEED MISC.CRYPTO.TIMESEED	Hardcoded Seed in PRNG Predictable Seed in PRNG
Polyspace Bug Finder	R2024a	CERT C++: MSC51-CPP	Checks for: Deterministic random output from constant seed Predictable random output from predictable seed Rule partially covered.
Parasoft C/C++ test	2024.2	CERT_CPP-MSC51-a	Properly seed pseudorandom number generators

Coding Standard 9

Source: https://wiki.sei.cmu.edu/confluence/display/c/MEM31-C.+Free+dynamically+allocated+memory+when+no+longer+needed

Coding Standard	Label	Free dynamically allocated memory when no longer needed
Memory Allocation	[STD-009-CPP]	Unless allocated memory is assigned to a pointer that terminates with the program, free() should always be called to deallocate pointer memory. This prevents DoS attacks via resource exhaustion.

Noncompliant Code

This code does not free an object prior to the end of the lifetime of the last pointer referring to it.

```
#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void) {
  char *text_buffer = (char *)malloc(BUFFER_SIZE);
  if (text_buffer == NULL) {
    return -1;
  }
  return 0;
}
```

Compliant Code

This code demonstrates the pointer being properly deallocated with free()

```
#include <stdlib.h>

enum { BUFFER_SIZE = 32 };

int f(void) {
  char *text_buffer = (char *)malloc(BUFFER_SIZE);
  if (text_buffer == NULL) {
    return -1;
  }

  free(text_buffer);
  return 0;
}
```

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

Principles(s): 3 - *Architect and Design for Security Policies* – Memory management is a primary security concern, handling memory allocation/deallocation appropriately can help prevent resource exhaustion which could lead to things like DoS attacks.
9 - *Use Effective Quality Assurance Techniques* – Testing for appropriate memory handling during the QA phase can help prevent this issue from making it to the deployment stage.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Probable	Medium	P8	L2

Automation

Tool	Version	Checker	Description Tool
Axivion Bauhaus Suite	7.2.0	CertC-MEM31	Can detect dynamically allocated resources that are not freed
CodeSonar	8.3p0	ALLOC.LEAK	Leak
Coverity	2017.07	RESOURCE_LEAK ALLOC_FREE_MISMATCH	Finds resource leaks from variables that go out of scope while owning a resource
LDRA tool suite	9.7.1	CERT_C-MEM31-a	Ensure resources are freed

Coding Standard 10

Source: https://wiki.sei.cmu.edu/confluence/display/c/EXP34-C.+Do+not+dereference+null+pointers

Coding Standard	Label	Do not dereference null pointers
Data	[STD-010-CPP]	Dereferencing a null pointer often results in program termination, however in the event it does not it turns into undefined behavior. This behavior can lead to security vulnerabilities on certain platforms.

Noncompliant Code

In this example, input_str is copied into dynamically allocated memory referenced by c_str. If malloc() fails, it returns a null pointer that is assigned to c_str. When c_str is dereferenced in memcpy(), or if input_str is a null pointer, the call to strlen() dereferences a null pointer, the program exhibits undefined behavior.

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

void f(const char *input_str) {
  size_t size = strlen(input_str) + 1;
  char *c_str = (char *)malloc(size);
  memcpy(c_str, input_str, size);
  /* ... */
  free(c_str);
  c_str = NULL;
  /* ... */
}
```

Compliant Code

This improved code adds error handling if the pointer is null.

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

void f(const char *input_str) {
  size_t size;
  char *c_str;

  if (NULL == input_str) {
    /* Handle error */
  }

  size = strlen(input_str) + 1;
```

Compliant Code

```
 c_str = (char *)malloc(size);
 if (NULL == c_str) {
   /* Handle error */
 }
 memcpy(c_str, input_str, size);
 /* ... */
 free(c_str);
 c_str = NULL;
 /* ... */
}
```

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

Principles(s): 10 - *Adopt a Secure Coding Standard* – Pointers should be handled appropriately to prevent abnormal program shutdown, or possible exploits.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

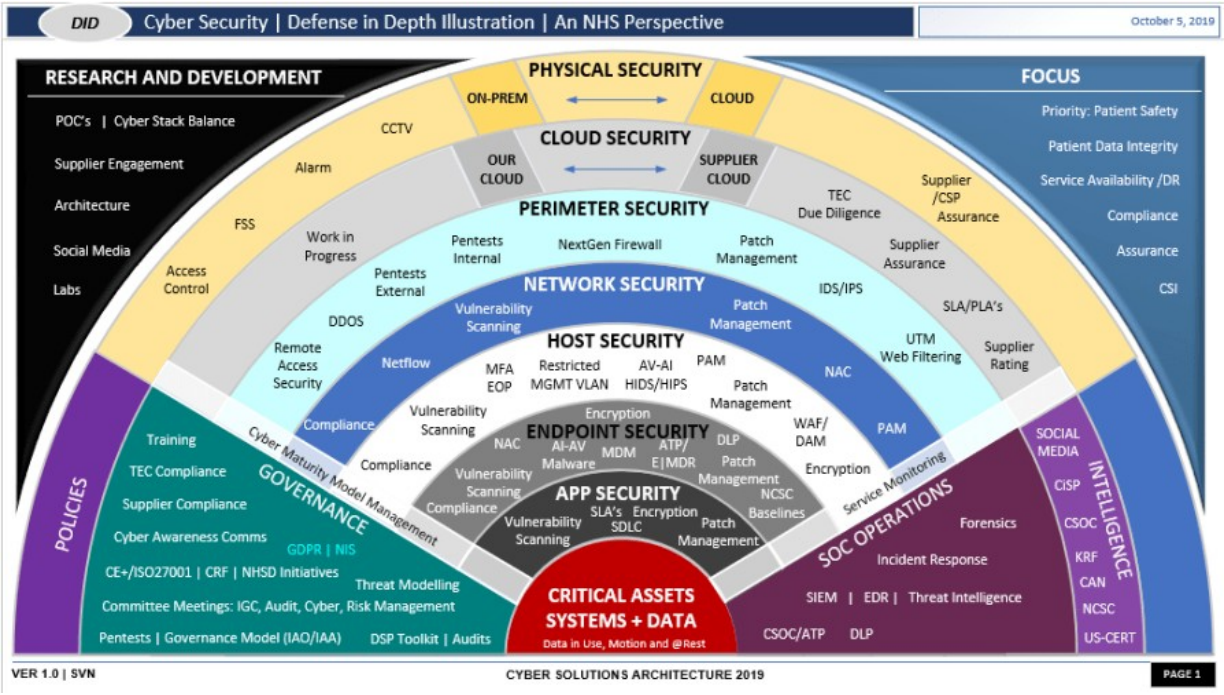
Automation

Tool	Version	Checker	Description Tool
Astree	24.04	null-dereferencing	Fully checked
CodeSonar	8.3p0	LANG.MEM.NPD LANG.STRUCT.NTAD LANG.STRUCT.UPD	Null pointer dereference Null test after dereference Unchecked parameter dereference
Coverity	2017.07	CHECKED_RETURN NULL_RETURNS REVERSE_INULL FORWARD_NULL	Finds instances where a pointer is checked against NULL and then later dereferenced Identifies functions that can return a null pointer but are not checked Identifies code that dereferences a pointer and then checks the pointer against NULL Can find the instances where NULL is explicitly dereferenced or a pointer is checked against NULL

Tool	Version	Checker	Description Tool
			but then dereferenced anyway. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary
Helix QAC	2024.4	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

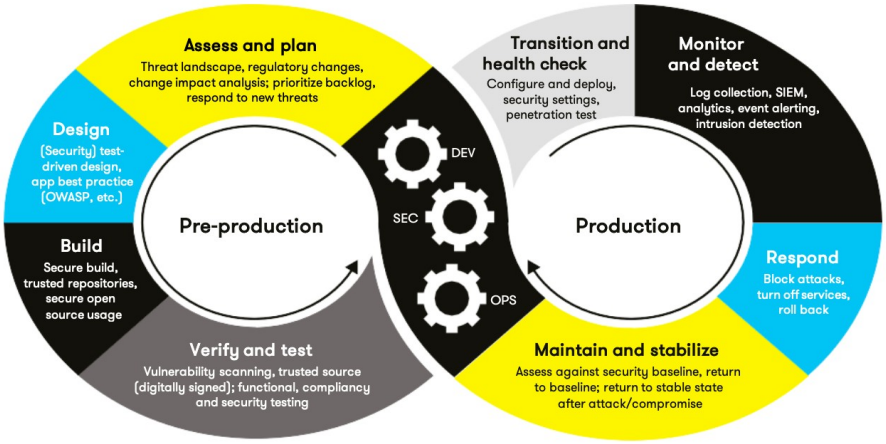
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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



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.

The plan for security automation should start as soon as the planning process is started, however, the automation process itself should be inserted into the build stage where options like static testing can be implemented to start identifying and preventing security vulnerabilities as soon as programming starts. Next, the verify and test step can start automating unit testing and more dynamic security testing and analysis tools to check the software components being used in a runtime type environment prior to actual release. Once this is complete the next stage to implement automation can be the transition and health check stage, there are tools that can be used to automate some factors of penetration testing, such as intruder, and this is where they could be finetuned for the Green Pace system as well as any other automated security tools for continuous/repeated use. Finally, implementing automated logging and alerts for suspicious activity, such as unauthorized access attempts or failed authentication, should be added to the monitor and detect stage. Altogether adding security automation to these four steps can provide increased vulnerability protection for the Green Pace system.

Summary of Risk Assessments

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

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-001-CPP	High	Unlikely	Medium	P9	2
STD-002-CPP	Low	Unlikely	Medium	P2	3
STD-003-CPP	High	Likely	Medium	P18	1
STD-004-CPP	High	Likely	Medium	P18	1

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-005-CPP	High	Likely	Medium	P18	1
STD-006-CPP	Medium	Unlikely	Medium	P4	3
STD-007-CPP	Low	Probable	Medium	P4	3
STD-008-CPP	Medium	Likely	Low	P18	1
STD-009-CPP	Medium	Probable	Medium	P8	2
STD-010-CPP	High	Likely	Medium	P18	1

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 at rest	Encryption at rest means that any data in storage should be encrypted. Even if this data is not used/accessed frequently storing it in plain text can still be a liability should a malicious user gain access. If utilized appropriately, any data collected in a breach must still be decrypted.
Encryption in flight	Encryption in flight indicates that any data being transferred from one place to another should still be encrypted. This helps prevent man-in-the-middle attacks because the decryption key is still necessary should the data get intercepted.
Encryption in use	Any data being used by the system should remain encrypted at all times, even as it is updated and modified. This helps keep sensitive data secure in the final place it could be accessed by a malicious user.

b. Triple-A Framework *	**Explain what it is and how and why the policy applies.**
Authentication	Authentication is used to confirm a user is who they say they are, most commonly

b. Triple-A Framework *	Explain what it is and how and why the policy applies.
	upon login, and may include multiple authentication factors. This policy is used anytime a user needs to access something in a system, be it accessing sensitive data or performing system modifications of any sort. Not everyone should have access to everything, and therefore authentication should always be used to verify someone is who they say they are.
Authorization	Authorization is what a user is allowed to do, and the level of authorization will vary by user type. This should be used in compliance with the coding standards of default deny and the principle of least privilege, meaning users should not have access to any system actions/services that are not necessary to their role. For example, a user logged in to their bank account should be able to see their balance and shift their money between say a checking and savings account, but have no reason to be able to modify user permissions.
Accounting	Accounting is keeping detailed logs of the activities performed/requests made by any user/visitor to a system. Having access to these logs can help identify where and who an attack came from quickly in the event it happens. Combining accounting with automated monitoring can provide early warning of an attempted attack as well.

*Use this checklist for the Triple A to be sure you include these elements in your policy:

- User logins
- Changes to the database
- Addition of new users
- User level of access
- Files accessed by users

Map the Principles

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

NOTE: Green Pace has already successfully implemented the following:

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

The only item you must complete beyond this point is the Policy Version History table.

Audit Controls and Management

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

Evidence will include the following:

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

Enforcement

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

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

Exceptions Process

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

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

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

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

Distribution

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

Policy Change Control

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

Policy Version History

Version	Date	Description	Edited By	Approved By
1.0	08/05/2020	Initial Template	David Buksbaum	
1.1	01/23/2025	Added ten core security principles descriptions and started filling out information about coding standards	Tabitha Tallent	[Insert text.]
1.2	02/16/2025	Completed coding standards, risk summary assessment, and Triple A security policies	Tabitha Tallent	[Insert text.]

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV