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	Data obtained from either the user or an external source has the potential to contain harmful information such as an SQL injection that could compromise private data of the application. On top of this, input data could be of the wrong input type. It is important to check that input data is both valid and safe before using it to prevent common security vulnerabilities such as buffer overflow and injection. Never assume that incoming data is going to be 100% safe.
2. Heed Compiler Warnings	While a compiler warning will not always prevent code from compiling and running, it is still wise to pay attention to these warnings. Warnings can alert developers to security vulnerabilities and deprecated code. Using deprecated code often results in security vulnerabilities and looking into those warnings can help the developer find a way to rewrite the code without the deprecated features. These warnings may also alert developers to bugs in their code that may be difficult to find at first. They are warnings for a reason.
3. Architect and Design for Security Policies	Setting up a robust security system through architect and design is essential. When the team has these design guidelines to refer to, the number of potential errors during development will hopefully decrease.
4. Keep It Simple	While it is important to have strong security, it is also important to make sure the application is still usable. Also, keeping the code simple makes it easier to debug by decreasing the amount of code one must sift through. Secure the most important aspects of the application and don't overcomplicate the design by adding too much.
5. Default Deny	Rather than create a blacklist of entities or roles to deny access to, create a whitelist with the default behavior being denial of access. Manually add access to those that absolutely need it, and deny everyone else, including new users. This prevents situations where a new user ends up with permission to access content they should otherwise be unable to

Principles	Write a short paragraph explaining each of the 10 principles of security.
	access. It also is just safer to err on the side of caution and completely limit who or what is able to access the network or application.
6. Adhere to the Principle of Least Privilege	Similar to the principle of Default Deny, adhering to the Principle of Least Privilege means giving the minimum amount of permission needed to an entity or role. One should only have the permissions needed to accomplish their task, and not anything else. Giving more than is needed only opens the door to potential risks.
7. Sanitize Data Sent to Other Systems	Sanitizing data sent to other systems helps keep your own data safe while also building trust with that system. Any data sent over a medium to another network or system has the possibility of being intercepted, so securing and sanitizing that data becomes more important. Also, just as it is important to validate input received from others, making sure the data you send is secure and sanitized builds trust in users and others using that data.
8. Practice Defense in Depth	Defense in Depth involves adding layers of security measures. Having multiple layers is important in case one of those security measures fails. In fact, it is wise to implement security measures with the assumption that they may fail and incorporate both backup systems and system logs to get information on what caused the security measure to fail. When implementing Defense in Depth, it is also important not to add too many security layers that do the same thing. If one is breached, then it is likely others of the same type will be breached as well.
9. Use Effective Quality Assurance Techniques	Quality Assurance assures that code and software are secure and functional. Keeping up with QA techniques keeps code from becoming full of bugs and other security issues. Code should be reviewed often before deployment to ensure that production is going smoothly, but not so often that it cuts into development time. Finding the right balance of QA checks can help production flow smoother and catch vulnerabilities before they become more difficult to find. Also, performing static code analysis will help keep code clean and secure.
10. Adopt a Secure Coding Standard	Create or find a coding standard to apply to the project so that all developers can follow it. When there is a standard to follow, it helps make sure that all code, no matter who worked on it, adheres to the same design and security principles. This can also allow for easier debugging and code review.

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	Name of Standard
Data Type	[STD-001-CPP]	Never qualify a reference type with const or volatile. This can result in undefined behavior such as allowing the data to be mutated when it otherwise should not be. Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP.+Never+qualify+a+reference+type+with+const+or+volatile

Noncompliant Code

A simple error that may arise while programming is that the programmer accidentally writes the following when trying to const-qualify a type that uses reference type:

```
char &const p;
```

Compliant Code

This is how to correctly const-qualify a type using reference type:

```
char const &p;
```

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

Principles(s): 10. Secure Coding Standard. Following the standard that has been laid out for developers will ensure that simple errors like these do not occur often.
2. Heed Compiler Warnings, would help detect and prevent this vulnerability.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Low	Low	L3

Automation

Tool	Version	Checker	Description Tool
Parasoft C/C++ test	2024.2	CERT_CPP-DCL52-a	Never qualify a reference type with 'const' or 'volatile'
Polyspace Bug Finder	R2025b	CERT C++:DCL52-CPP	Checks for const-qualified reference types and modification of const-qualified reference types

Tool	Version	Checker	Description Tool
Clang	3.9		Checks for violations of this rule and produces an error without the need to specify any special flags or options
SonarQube C/C++ Plugin	4.10	S3708	

Coding Standard 2

Coding Standard	Label	Name of Standard
Data Value	[STD-002-CPP]	Ensure that unsigned integer operations do not wrap. Wrapping can lead to undefined and unexpected behavior. Source: https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap

Noncompliant Code

When ui_a and ui_b are added, it may result in unexpected behavior depending on their values. Unsigned integers wrap around when they reach the max value, and this code does not check to see if that value will be reached.

```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  /* ... */
}
```

Compliant Code

Using UINT_MAX – ui_a < ui_b, you can check to make sure that adding the two values together will not go over the max allowed value.

```
#include <limits.h>

void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum;
  if (UINT_MAX - ui_a < ui_b) {
    /* Handle error */
  } else {
    usum = ui_a + ui_b;
  }
  /* ... */
}
```

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

Principles(s): 1. Validate Input Data. Ensuring that the input you are using will not cause overflow or underflow or other unexpected behavior is crucial as it can lead to extreme vulnerabilities such as injection.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	Medium	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	24.04	Integer-overflow	Fully checked
Axivion Bauhaus Suite	7.2.0	CertC-INT30	Implemented
CodeSonar	9.1p0	ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW 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	INTEGER_OVERFLOW	Implemented

Coding Standard 3

Coding Standard	Label	Name of Standard
String Correctness	[STD-003-CPP]	Do not attempt to modify string literals as it can lead to undefined behavior. Source: https://wiki.sei.cmu.edu/confluence/display/c/STR30-C.+Do+not+attempt+to+modify+string+literals

Noncompliant Code

Attempting to modify a string literal can lead to undefined behavior and should be avoided.

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

Compliant Code

This code creates a copy of the string literal, allowing for modification.

```
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. Ensure that incoming data is not attempting to modify string literals through buffer overflow.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Likely	Medium	Low	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	24.04	String-literal-modfication Write-to-string-literal	Fully checked
Axivion Bauhaus Suite	7.2.0	CertC-STR30	Fully implemented
Coverity	2017.07	PW	Deprecates conversion from a string literal to “char *”
LDRA tool suite	9.7.1	157 S	Partially implemented

Coding Standard 4

Coding Standard	Label	Name of Standard
SQL Injection	[STD-004-CPP]	Guarantee that storage for strings has sufficient space for character data and the null terminator. Not doing so can allow users to cause buffer overflow and potentially SQL injection through manipulating SQL queries. 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

Noncompliant Code

Running an SQL query without checking for SQL Injection can lead to errors not caught by simply checking for an access error. The code below would fail if an improper query was passed, but would not catch SQL injection techniques such as adding “OR 1=1” to the end of the query, since it would be considered proper.

```
char* error_message;

if(sqlite3_exec(db, sql.c_str(), callback, &records, &error_message) !=
SQLITE_OK)

{

    std::cout << "Data failed to be queried from USERS table. ERROR = " <<

    error_message << std::endl; sqlite3_free(error_message);

    return false;

}
```

Compliant Code

With C++, it is best to use a regular expression to search SQL queries for bad code. The following code detects common SQL injection techniques, but the regular expression can be adjusted.

```
std::regex injectionTest("\\b([o|O][r|R] (.)=\\2;+.|[o|O][r|R] (.*)=\\3);?");

if (std::regex_search(sql, injectionTest)) {
    return false;
}
```

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

Principles(s): 5. Default Deny. While not exactly denying access, it still follows the line of thinking that something is bad by default. Assume all SQL queries are bad and only allow what is known to be good rather than attempting to blacklist all kinds of injection code.
6. Adhere to the Principle of Least Privilege. This also suits because you should only allow users to access the data they specifically need. There is no reason to allow a default user to access all elements of the data.
1. Validate Input Data. It is necessary to validate any data received from unknown entities whether they are users or third parties.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	High	L3

Automation

Tool	Version	Checker	Description Tool
sqlmap	1.9	sqlmap	Open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws (https://github.com/sqlmapproject/sqlmap)
Fuzzdb	1.0	Sql-injection	Fully implemented
sqlbftools	1.2	Http-sql	Automatic SQL injection tool
MySqloit	[Insert text.]	mysqloit	Injection takeover tool

Coding Standard 5

Coding Standard	Label	Name of Standard
Memory Protection	[STD-005-CPP]	Do not access freed memory. Attempting to do so can result in exploitable vulnerabilities. Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory

Noncompliant Code

The struct, s, is created and then subsequently deleted. The program then tries to access this pointer after its deletion.

```
struct S {
  void f();
};

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

Compliant Code

Do not attempt to access or use the data after it is deleted. Make sure that any code attempting to do so is before the deletion.

```
struct S {
  void f();
};

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

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

Principles(s): 10. Secure Coding Standard. Closing up and freeing memory after it is no longer needed is an essential step in secure coding. Following this standard can reduce the likelihood of accessing data that is no longer available in memory.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	High	Medium	L2

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	9.1p0	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

Coding Standard	Label	Name of Standard
Assertions	[STD-006-CPP]	Incorporate diagnostic tests using assertions. Assertions should be used during debugging and turned off before deployment. Do not use them for runtime error checking. Source: https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions

Noncompliant Code

This example uses assert to check for proper memory allocation. But because memory availability can change throughout the lifetime of a program, the program can terminate early and open up the possibility of a denial-of-service attack.

```
char *dupstring(const char *c_str) {
  size_t len;
  char *dup;

  len = strlen(c_str);
  dup = (char *)malloc(len + 1);
  assert(NULL != dup);

  memcpy(dup, c_str, len + 1);
  return dup;
}
```

Compliant Code

In this example, an assertion is not used for production code and memory allocation is still properly checked.

```
char *dupstring(const char *c_str) {
  size_t len;
  char *dup;

  len = strlen(c_str);
  dup = (char*)malloc(len + 1);
  /* Detect and handle memory allocation error */
  if (NULL == dup) {
      return NULL;
  }

  memcpy(dup, c_str, len + 1);
```

Compliant Code

```
  return dup;
}
```

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

Principles(s): 3. Effective Quality Assurance Techniques. Assertions are done in development and are removed when the product is released. They are useful for testing code and should be done throughout the development process.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Unlikely	Very High	Low	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	9.1p0	LANG.FUNCS.ASSERTS	Not enough assertions
Coverity	2017.07	ASSERT_SIDE_EFFECT	Can detect the specific instance where assertion contains an operation/function call that may have a side effect
Parasoft C/C++ test	2024.2	CERT_C-MSC11-a	Assert liberally to document internal assumptions and invariants
Security Reviewer – Static Reviewer	6.02	CPPPBE	Fully implemented

Coding Standard 7

Coding Standard	Label	Name of Standard
Exceptions	[STD-007-CPP]	Handle all exceptions. Not doing so can lead to denial-of-service attacks through abnormal process termination. Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions

Noncompliant Code

In this code, the exceptions thrown by throwing_func are not caught. When there is no matching handler for an exception, std::terminate() is called and abnormal process termination occurs.

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

void f() {
  throwing_func();
}

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

Compliant Code

Here, the exceptions thrown by throwing_func are handled in the try/catch block within main.

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

void f() {
  throwing_func();
}

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

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

Principles(s): 2. Heed Compiler Warnings. The compiler can alert developers to potential exceptions that may not necessarily cause a compiler error. Paying attention to both errors and warnings is essential to ensure a product that is as secure as possible.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probable	Low	Low	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	Main-function-catch-all Early-catch-all	Partially checked
CodeSonar	7.2.0	LANG.STRUCT.UCTCH PARSE.MBDH	Masked by handler Masked by default handler
LDRA tool suite	9.7.1	527 S	Partially implemented
Security Reviewer – Static Reviewer	6.02	C35	Fully implemnted

Coding Standard 8

Coding Standard	Label	Name of Standard
File Input Output	[STD-008-CPP]	Close files when they are no longer needed and before the end of the program. This ensures that dynamically allocated memory is properly deallocated. Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed

Noncompliant Code

In this code block, the file is opened but never closed before the process terminates. This could lead to memory issues.

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

Compliant Code

Here, the file is closed before the end of the program and there is additional checking for close failure.

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

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

Principles(s): 10. Secure Coding Standard. The compiler may not give warnings about files not being closed, especially if they don't lead to unexpected behavior. Therefore, adhering to the coding standard set up should make sure that developers take care to close files.
9. Effective Quality Assurance Techniques. Using assertions and static code analysis can also potentially detect this vulnerability and prevent memory issues from leaving files open when they are no longer needed.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Very High	Low	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	9.1p0	ALLOC.LEAK	Leak
Parasoft C/C++ test	2024.2	CERT_CPP-FIO51-a	Ensure resources are freed
Polyspace Bug Finder	R2025b	CERT C++:FIO51-CPP	Checks for resource leak (rule partially covered)
Security Reviewer – Static Reviewer	6.02	C80	Fully implemented

Coding Standard 9

Coding Standard	Label	Name of Standard
Containers	[STD-009-CPP]	Use valid iterator ranges. When iterating through elements of a container, use valid iterator ranges to avoid undefined behavior caused by attempting to access an element that does not exist. Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges

Noncompliant Code

The first iterator in this example, c.end(), does not precede the second, c.begin(). The first iterator will continue to increment, leading to undefined behavior.

```
void f(const std::vector<int> &c) {
  std::for_each(c.end(), c.begin(), [](int i) { std::cout << i; });
}
```

Compliant Code

Checking the iterators and being sure that the first precedes the second will ensure proper iterating. In this example, the iteration starts at the beginning and goes until the end instead of the other way around.

```
void f(const std::vector<int> &c) {
  std::for_each(c.begin(), c.end(), [](int i) { std::cout << i; });
}
```

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

Principles(s): 1. Validate Input Data. Be sure that the iterators are not set up by user input, and if so, then validate the input to prevent undefined behavior.
10. Secure Coding Standard. Preventing issues from accessing data that may not exist is a basic standard that should be implemented in all coding standards.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Very High	Medium	L2

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	Overflow_upon_dereference	
CodeSonar	9.1p0	LANG.MEM.BO	Buffer Overrun
Parasoft C/C++ test	2024.2	CERT_CPP-CTR53-a CERT_CPP-CTR53-b	Do not use an iterator range that isn’t really a range Do not compare iterators from different containers
Polyspace Bug Finder	R2025b	CERT C++: CTR53-CPP	Checks for invalid iterator range (rule partially covered).

Coding Standard 10

Coding Standard	Label	Name of Standard
Memory	[STD-010-CPP]	Detect and handle memory allocation errors. Since operator new(size_t) throws an exception but operator new(size_t, no_throw t&) does not, it is important to check for memory allocation errors on the latter before accessing the resulting pointer. Source: https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM52-CPP.+Detect+and+handle+memory+allocation+errors

Noncompliant Code

Here, a pointer is created using new(size_t) and the results are not checked. This can lead to abnormal termination of the program if an exception is thrown due to allocation fail.

```
void f(const int *array, std::size_t size) noexcept {
  int *copy = new int[size];
  std::memcpy(copy, array, size * sizeof(*copy));
  // ...
  delete [] copy;
}
```

Compliant Code

This pointer using std::nothrow, which will return a null pointer if allocation fails. This code checks for nullptr before continuing.

```
void f(const int *array, std::size_t size) noexcept {
  int *copy = new (std::nothrow) int[size];
  if (!copy) {
    // Handle error
    return;
  }
  std::memcpy(copy, array, size * sizeof(*copy));
  // ...
  delete [] copy;
}
```

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

Principles(s): 4. Keep It Simple. Using the more complex version of a function or variable definition is not always the best option. Choosing the correct version can avoid errors and vulnerabilities.

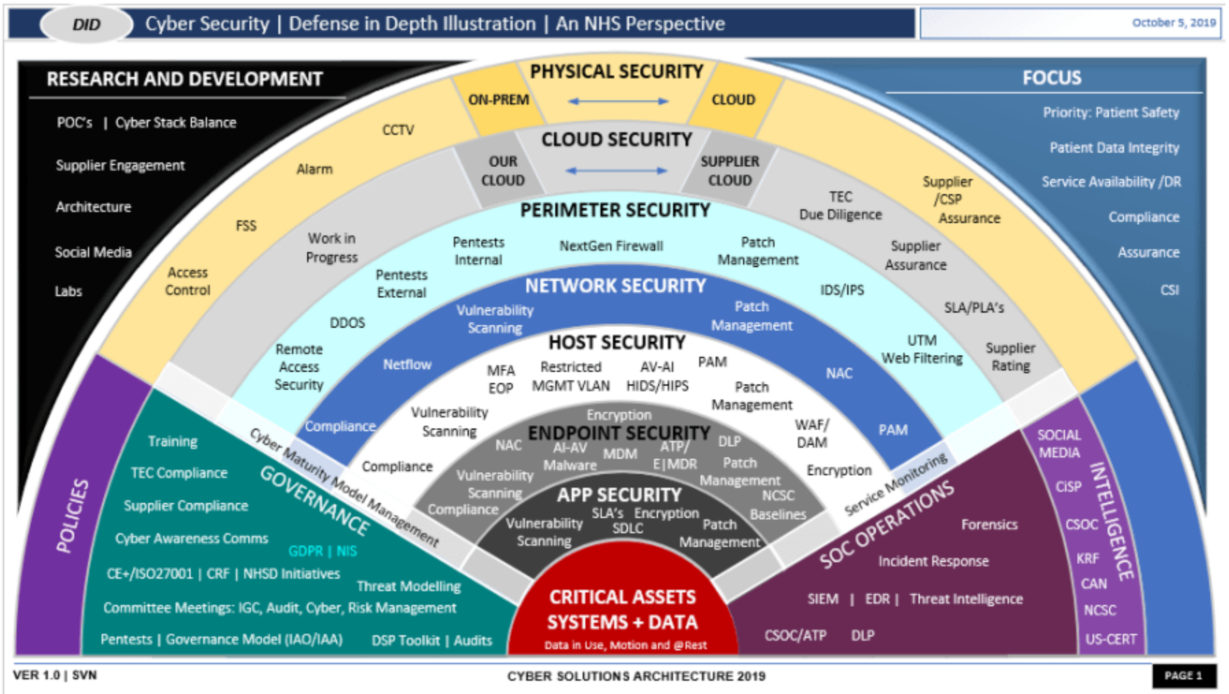
Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Low	Very High	L1

Automation

Tool	Version	Checker	Description Tool
Coverity	7.5	CHECKED_RETURN	Finds inconsistencies in how function call return values are handled
LDRA tool suite	9.7.1	45 D	Partially implemented
Parasoft C/C++ test	2024.2	CERT_CPP-MEM52-a CERT_CPP-MEM52-b	Check the return value of new Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined
Polyspace Bug Finder	R2025b	CERT C++: MEM52-CPP	Checks for unprotected dynamic memory allocation (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

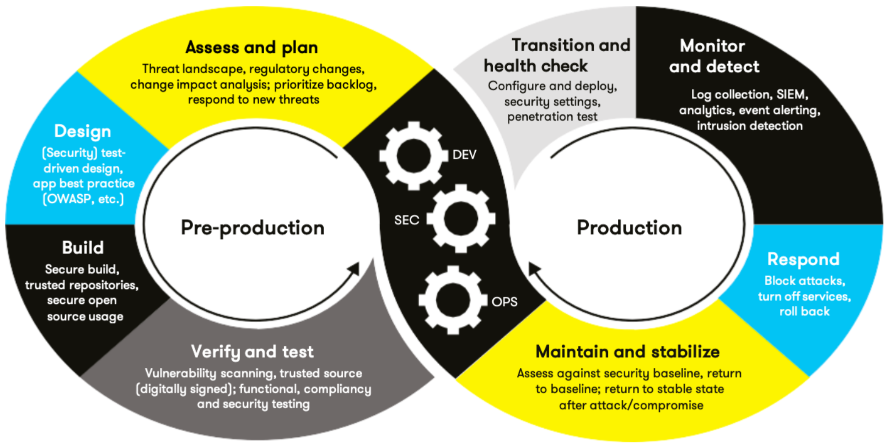
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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


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 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 existing DevOps process could be modified to include more emphasis on the design, build, and test process. Currently, the diagram implies that this process happens once before production, but it should be a continuous process throughout the development life cycle. To accomplish this, adding quality assurance techniques such as static code analysis will make the flow of the life cycle smoother. Automating testing and code analysis will save development time so that developers can focus on writing code. To make this process better, adopting a secure coding standard that all developers follow is ideal. This way, even if a product is worked on by multiple developers, the format and standard of the code will match throughout.

It is also important to include a session before pre-production begins where the team discusses what is truly necessary in the product. What data should be gathered, how it is used, and how long it is stored are all important metrics to discuss and solidify before development begins. Starting the process already thinking about security will help set the tone for the entire duration of the product's life cycle. Collecting more data than is needed sets it up for being potentially targeted and stolen, so deciding what exactly is needed will cut down on the security needed as well.

Summary of Risk Assessments

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

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

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-004-CPP	High	Likely	High	High	3
STD-005-CPP	High	Likely	High	High	2
STD-006-CPP	Low	Unlikely	Very High	Low	3
STD-007-CPP	Low	Probable	Low	Low	2
STD-008-CPP	Medium	Unlikely	Very High	Low	3
STD-009-CPP	High	Probable	Very High	Medium	2
STD-010-CPP	High	Likely	Low	Very High	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	Data is at rest when it is within some kind of storage device, like a hard drive, the cloud, or a database, and not currently in use or in the process of being sent or received. Encrypting data at rest will protect it from attackers that manage to get into the storage device. Setting up defense in depth via encryption, authorization, and authentication can help slow down and stop attackers. Data at rest is often most in need of protection, as it is usually sensitive data. This means that the encryption needs to be solid. Applying the principle of least privilege is also important as it ensures that only those who absolutely need access to the data are authorized.
Encryption in flight	Data is in flight when it is in the process of being sent or received via something like email or other messaging or communication service. Attackers can intercept data in flight to access it, so encryption at this stage is also important. Data in flight may not be as important as data at rest, but in this state, it is more vulnerable than if it were at rest. Strong encryption is needed at this stage because there is a lack of other defensive options. Reference: https://jatheon.com/blog/data-at-rest-data-in-motion-data-in-use/

a. Encryption	Explain what it is and how and why the policy applies.
Encryption in use	Data in use is data that is being accessed and used by a user or third party. It is most vulnerable in this stage and requires strong encryption. The principle of least privilege is also important to apply here so that only those who need access can. It is also important to apply the principle of Deny by Default to stop unauthorized access before the data becomes in use. Once data is in use, encryption is the best option to secure it, so it must be strongest at this stage.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authentication	At the authentication step, the user is identified through a login process, and access is given based on the user's credentials and access level. Giving certain access levels to different users keeps data safe from those who don't need access to it. Each user will have their own level of access, and this is determined upon login. Authentication also makes sure that anyone without permission cannot access specific sections of the product such as sensitive data.
Authorization	Authorization is similar to authentication, but it determines what the user can do. For example, a regular user should not be able to make significant changes, like deletions, to the database. Applying different permissions to different users helps to secure the system. By default, a new kind of user should start with zero permissions, and these permissions should be added on an as-needed basis to comply with the Principle of Least Privilege.
Accounting	With accounting, the history of what users do while logged into the system will be recorded. This allows administrators or other users with high-level permissions to view what others are doing on the system and make sure that someone does not have authorization to do what they should not be able to do, such as a regular user accessing sensitive files.

*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	09/21/2025	Security Policy Update and Coding Standards Addition	Kelsey Wanderlingh	Kelsey Wanderlingh
2.0	10/10/2025	Completed Security Policy	Kelsey Wanderlingh	Kelsey Wanderlingh

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV