**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## 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](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## 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. ValidateInput Data | Always validate and sanitize any data coming into your system, especially if it's coming from users. I've seen how skipping this step can open the door to all kinds of problems like SQL injection, buffer overflows, and even crashes. Just taking a few extra seconds to double-check that input matches what you expect can save a ton of headaches down the line. It's like checking someone's ID before letting them into a secure building. |
| 1. Heed Compiler Warnings | Compiler warnings might seem annoying at first, but they’re actually doing us a favor. They flag stuff like uninitialized variables or type mismatches that might seem harmless… until they’re not. I’ve learned it’s way better to deal with these early than to chase down bugs later. Treat warnings like smoke, they might not be fire yet, but ignoring them is never a good idea. |
| 1. Architect and Design for Security Policies | Security isn’t something you tack on after the code’s written. It needs to be part of the design process from day one. That means thinking about what data needs to be protected, how it flows, and where things could go wrong. If you build security into your design, it saves a lot of retrofitting later, and makes your system stronger overall. |
| 1. Keep It Simple | The simpler the code, the easier it is to secure and maintain. Complexity introduces more chances for bugs, misconfigurations, and just general confusion. I try to write code that’s not just functional, but understandable, even if I have to revisit it six months later. Clean, straightforward logic reduces risk and makes life easier for everyone on the team. |
| 1. Default Deny | If someone doesn’t need access, don’t give it to them. Period. It’s better to start with no access and open things up only when necessary, rather than the other way around. I’ve seen how default allow policies can lead to things slipping through the cracks. "Default deny" keeps the system tight and forces intentional decisions about who gets what. |
| 1. Adhere to the Principle of Least Privilege | Kind of tied to “default deny,” this one’s all about giving people and systems only what they need, nothing more. Whether it’s file access, admin rights, or API permissions, less is more when it comes to privilege. It limits what damage can be done if something goes wrong, and it makes it easier to keep things under control. |
| 1. Sanitize Data Sent to Other Systems | Just like you need to validate incoming data, you also need to clean up what you're sending out. If your system passes dirty data to another one, it could cause just as much damage, whether that's SQL injection, broken parsing, or worse. It’s like cleaning up before handing something off: basic courtesy and good security. |
| 1. Practice Defense in Depth | No single security measure is perfect. That’s why we layer them, encryption, firewalls, authentication, monitoring, etc. If one thing fails, something else is there to catch it. I think of it like locking your doors, setting an alarm, and having a dog. It's not about paranoia; it’s about being smart and covering your bases. |
| 1. Use Effective Quality Assurance Techniques | QA isn’t just about making sure the app works; it’s about making sure it’s secure too. That means regular code reviews, static analysis, checking logs, running security tests… all of it. Catching issues early is always cheaper and safer than fixing them after something bad happens. A strong QA game builds confidence in your code. |
| 1. Adopt a Secure Coding Standard | Having a shared coding standard means everyone’s on the same page when it comes to writing secure, maintainable code. It keeps things consistent, helps new team members onboard faster, and makes code reviews a whole lot smoother. Plus, when everyone follows the same security rules, there’s less room for mistakes to sneak in. |

### 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 non-compliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Avoid type mismatches |

| **Noncompliant Code** |
| --- |
| Overflow |
| int x = 2147483647 + 1; // Overflow |

| **Compliant Code** |
| --- |
| Introduce limits |
| #include <limits>  long long x = static\_cast<long long>(2147483647) + 1; |

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

| **Principles(s):** Principles 1 (Validate Input Data) and 2 (Heed Compiler Warnings) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Cpp | Detects type conversion issues and overflow risks |
| Cppcheck | 2.10 | truncLongCastAssignment | Identifies truncation in type casting operations |
| Clang-Tidy | 15.0 | Bugprone-signed-char-misuse | Detects misuse of signed char leading to integer issues |
|  |  |  |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Prevents assumptions about variables leading to logic errors. |

| **Noncompliant Code** |
| --- |
| No check for b == 0 |
| int divide(int a, int b) { return a / b; } |

| **Compliant Code** |
| --- |
| Catching invalid arguments before they happen |
| int divide(int a, int b) {  if (b == 0) throw std::invalid\_argument("Division by zero");  return a / b;  } |

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

| **Principles(s):** Principles 1 (Validate Input Data) and 4 (Keep It Simple) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Cpp | Find variables used without validation |
| Cppcheck | 2.10 | nullPointer | Identifies potential null pointer dereference issues |
| Clang-tidy | 15.0 | Clang-analyzer-core.DivideZero | Detects division by zero |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Prevents buffer overflow |

| **Noncompliant Code** |
| --- |
| Not safe string handling |
| char buffer[10];  strcpy(buffer, "Too long"); |

| **Compliant Code** |
| --- |
| Prevents buffer overflow |
| char buffer[10];  strncpy(buffer, "Safe", sizeof(buffer) - 1);  buffer[sizeof(buffer) - 1] = '\0'; |

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

| **Principles(s):** Principles 1 (Validate Input Data) and 7 (Sanitize Data Sent to Other Systems) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Medium | Critical | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Cpp | Detects string buffer overflow |
| Cppcheck | 21.0 | BufferAccessOutOfBounds | Identifies buffer accesses outside allocated memory |
| Clang-tidy | 15.0 | Clang-analyzer-security.insecureAPI.strcpy | Flags unsafe string copy functions |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevents SQL injection by using prepared statements. |

| **Noncompliant Code** |
| --- |
|  |
| std::string query = "SELECT \* FROM users WHERE name = '" + userInput + "'"; |

| **Compliant Code** |
| --- |
| Uses prepared statements to prevent SQL injection |
| stmt = con->prepareStatement("SELECT \* FROM users WHERE name = ?");  stmt->setString(1, userInput); |

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

| **Principles(s):** Principles 1 (Validate Input Data) and 7 (Sanitize Data Sent to Other Systems) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Medium | Critical | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Cpp | Detects potential SQL injection vulnerabilities |
| Veracode | 22.1 | CWE-89 | Identifies improper neutralization of SQL commands |
| Checkmarx | 9.4 | SQL\_Injection | Scans for SQL injection patterns in code |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Pointers that cause memory leaks |

| **Noncompliant Code** |
| --- |
| Using pointers incorrectly |
| int\* ptr = new int(10);  delete ptr;  delete ptr; |

| **Compliant Code** |
| --- |
| Correct use of pointers |
| std::unique\_ptr<int> ptr = std::make\_unique<int>(10); |

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

| **Principles(s):** Principles 4 (Keep It Simple) and 10 (Adopt a Secure Coding Standard) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Cpp | Detects memory leaks from improper pointer management |
| Cppcheck | 2.10 | Memleak | Identifies memory leaks in c++ code |
| Valgrind | 3.19 | Memcheck | Detects memory leaks and memory management issues |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Ensuring assumptions about the system stay true |

| **Noncompliant Code** |
| --- |
| Dereferencing nullptrr |
| int\* ptr = nullptr;  \*ptr = 5; |

| **Compliant Code** |
| --- |
| Using an assert function accomplishes this |
| assert(ptr != nullptr); |

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

| **Principles(s):** Principles 2 (Heed Compiler Warnings) and 9 (Use Effective Quality Assurance Techniques) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Possible | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Cpp | Identifies code where conditions are always true or false |
| Cppcheck | 2.10 | assertWithSideEffect | Detects assertions with side effects |
| Clang-tidy | 15.0 | Bugprone-assert-side-effect | Finds assertions containing side effects |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exception handling to prevent undefined behavior |

| **Noncompliant Code** |
| --- |
| No error handling |
| int divide(int a, int b) { return a / b; } |

| **Compliant Code** |
| --- |
| Error handling |
| try {  int result = divide(10, 0);  } catch (const std::exception& e) {  std::cerr << "Error: " << e.what() << std::endl;  } |

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

| **Principles(s):** Principles 3 (Architect and Design for Security Policies) and 9 (Use Effective Quality Assurance Techniques) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Cpp | Detects insufficient exception handling |
| Cppcheck | 2.10 | exceptThrowInDestructor | Identifies exceptions thrown in destructor functions |
| Clang-tidy | 15.0 | Bugprone-exception-escape | Finds exceptions that escape destructors |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Random Number Generation | [STD-008-CPP] | Prevent randomness from being used in cryptographic applications. |

| **Noncompliant Code** |
| --- |
| Not cryptographically secure |
| int randNum = rand(); |

| **Compliant Code** |
| --- |
| Cryptographically secure |
| std::random\_device rd;  std::mt19937 gen(rd()); |

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

| **Principles(s):** Principles 3 (Architect and Design for Security Policies) and 8 (Practice Defense in Depth) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Possible | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Cpp | Detects use of pseudorandom generators in security contexts |
| Veracode | 22.1 | CWE-338 | Identifies use of weak PRNG for cryptographic operations |
| Fortify | 21.2 | Insecure Randomness | Finds instances of insecure random number generation |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Race Conditions | [STD-009-CPP] | Prevents race conditions in multithreaded environments |

| **Noncompliant Code** |
| --- |
| Not thread safe |
| int counter = 0;  void increment() { counter++; } |

| **Compliant Code** |
| --- |
| Thread safe |
| std::atomic<int> counter = 0;  void increment() { counter.fetch\_add(1, std::memory\_order\_relaxed); } |

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

| **Principles(s):** Principles 4 (Keep It Simple) and 8 (Practice Defense in Depth) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | High | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Cpp | Detects potential thread safety issues |
| ThreadSanitizer | 12.0 | Data-race | Dynamic analysis tool to detect data races |
| Clang-tidy | 15.0 | Bugprone-thread-safety-attributes | Checks thread safety attributes in multithreaded code |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Handling | [STD-010-CPP] | Prevents leaks and unauthorized access to files |

| **Noncompliant Code** |
| --- |
| Did not close file |
| FILE\* file = fopen("data.txt", "r");  if (!file) return; |

| **Compliant Code** |
| --- |
| Proper handling |
| std::ifstream file("data.txt");  if (!file) { std::cerr << "File not found"; } |

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

| **Principles(s):** Principles 5 (Default Deny) and 6 (Adhere to the Principle of Least Privilege) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | Cpp | Detects unclosed file handlers |
| Cppcheck | 21.0 | resourceLeak | Identifies resource leaks including file handlers |
| Clang-tidy | 15.0 | Bugprone-unchecked-optional-access | Finds unchecked file access operations |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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.

At Green Pace, we're going all-in on security automation throughout our DevSecOps pipeline. The idea is to weave security right into our existing DevOps setup, not just slap it on as an afterthought. That means using the tools we already rely on, while bringing in new security-focused ones to keep us on track with our standards.

**Plan Phase**

Before any code is written, we’ll bring in automated risk assessment tools to check our plans against security standards early on. We're also adding threat modeling tools so we can catch potential issues right from the start. Better to catch it now than scramble to fix it later, right?

**Create Phase**

As devs are writing code, we’ll set them up with security-focused IDE plugins to give real-time feedback. Static analysis will be baked in at the IDE level, flagging anything that doesn’t line up with our standards as the code’s being written, not after the fact.

**Verify Phase**

This is where things get serious. We'll use tools like SonarQube, Cppcheck, and Clang-Tidy for static code analysis (SAST), making sure our coding standards are actually being followed. On top of that, we’ll bring in DAST tools to test for runtime vulnerabilities and use SCA tools to flag any risky dependencies.

**Pre-Production Phase**

Before code hits production, we’ll run automated fuzzing tests to shake out any weird edge-case bugs or inputs we might’ve missed. Security-focused integration testing will also be part of the mix, because if it’s going to break, we want to know before users do.

**Prevent/Detect Phases**

In production, we’re adding Runtime Application Self-Protection (RASP) to actively monitor and stop exploitation attempts as they happen. Network monitoring tools will be in place too, keeping an eye out for anything sketchy in real time.

**Continuous Improvement**

This isn’t a “set it and forget it” kind of thing. We're tracking every coding standard violation, generating monthly reports to find patterns, and updating our tools regularly so we’re not blindsided by new types of threats. Our standards will evolve too, as the tools give us better insights, we’ll adapt.

### 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 | High | 2 |
| STD-002-CPP | High | Likely | Low | High | 4 |
| STD-003-CPP | Critical | Likely | Medium | Critical | 5 |
| STD-004-CPP | Critical | Likely | Medium | Critical | 5 |
| STD-005-CPP | High | Likely | Medium | High | 4 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | High | Likely | Medium | High | 4 |
| STD-008-CPP | Critical | Possible | Medium | High | 4 |
| STD-009-CPP | High | Possible | High | High | 4 |
| STD-010-CPP | High | Likely | Low | High | 4 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. 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.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | **What it is**: Basically, this means locking down any data that’s sitting somewhere, like on a drive, in a database, backups, etc. so that if someone gets physical access to it, it’s still unreadable without the key.  **Why we care**: We’re talking source code, customer info, passwords, config files, stuff we *really* don’t want exposed. Everything stored needs to be encrypted. We’ll use AES-256 across the board, encrypt specific database fields, and even encrypt dev containers. Also, key management can’t be an afterthought. Least privilege access and key rotation are a must so we’re not leaving the front door open. |
| Encryption in flight | **What it is**: This covers data moving between systems, like APIs, services, or apps talking over the network.  **Why we care**: It’s super easy for attackers to snoop on unencrypted traffic. That’s why everything needs to move through secure channels. TLS 1.3 for websites, SFTP instead of old-school FTP, SSH over Telnet, and encrypted API tokens all the way. We’ll rotate keys and certificates regularly, and for mobile apps, we’ll even use certificate pinning to make sure we’re talking to the right server. |
| Encryption in use | **What it is**: This is the trickiest one, protecting data while it’s actually being used in memory.  **Why we care**: If sensitive data is hanging around in RAM longer than it should, it can be exposed through memory dumps or other attacks. So we’ll use secure memory allocation, wipe memory after use, and where it makes sense, maybe even look at homomorphic encryption so we can compute on encrypted data without decrypting it. We’ll also keep encryption keys out of memory unless absolutely necessary and only for as long as they’re needed. This is especially critical for anything dealing with personal info, payments, or login credentials. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | |  | | --- | |  |  |  | | --- | | **What it is**: Proving someone or something is who they say they are.  **How we’re handling it**: MFA is non-negotiable. We’ll use secure API keys or certs for service-to-service auth, and password-based systems will have complexity rules, lockouts, and all that good stuff. Failed logins? Logged and monitored. Ideally, we’ll use centralized solutions like SSO to streamline and secure the whole process. | |
| Authorization | **What it is**: Controlling what people can actually do once they’re in.  **How we’re handling it**: RBAC (Role-Based Access Control) is our go-to. Least privilege is the name of the game, no one gets more access than they need. These checks should exist at both the UI and API levels. All permission changes will be logged, and anything high-risk, like messing with user roles or changing DB settings, should require an extra layer of approval. |
| Accounting | **What it is**: Logging and auditing activity for security, compliance, and “what just happened” situations.  **How we’re handling it**: Everything from logins to data access to permission changes will be logged. We’ll make sure logs have enough detail, timestamps, user info, IP addresses, outcomes, the works. Sensitive stuff (like user creation or DB structure changes) will get enhanced logging. Logs will live in a secure, centralized, tamper-proof place, with retention based on our compliance needs. |

**\***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 |  |
| 2.0 | 04/20/2025 | Added risk assessments, automation controls, and mapped principles to coding standards | Greg Monti |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |