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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.gjdgxs)

[Purpose 2](#_heading=h.30j0zll)

[Scope 2](#_heading=h.1fob9te)

[Module Three Milestone 2](#_heading=h.3znysh7)

[Ten Core Security Principles 2](#_heading=h.4d34og8)

[C/C++ Ten Coding Standards 3](#_heading=h.2s8eyo1)

[Coding Standard 1 4](#_heading=h.3rdcrjn)

[Coding Standard 2 5](#_heading=h.26in1rg)

[Coding Standard 3 6](#_heading=h.lnxbz9)

[Coding Standard 4 7](#_heading=h.35nkun2)

[Coding Standard 5 8](#_heading=h.1ksv4uv)

[Coding Standard 6 9](#_heading=h.2jxsxqh)

[Coding Standard 7 10](#_heading=h.z337ya)

[Coding Standard 8 11](#_heading=h.28h4qwu)

[Coding Standard 9 13](#_heading=h.nmf14n)

[Coding Standard 10 14](#_heading=h.37m2jsg)

[Defense-in-Depth Illustration 15](#_heading=h.1mrcu09)

[Project One 15](#_heading=h.46r0co2)

[1.](#_heading=h.2lwamvv) Revise the C/C++ Standards 15

[2.](#_heading=h.111kx3o) Risk Assessment 15

[3.](#_heading=h.3l18frh) Automated Detection 15

[4.](#_heading=h.206ipza) Automation 15

[5.](#_heading=h.4k668n3) Summary of Risk Assessments 16

[6.](#_heading=h.2zbgiuw) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.1egqt2p) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.3ygebqi)

[Enforcement 18](#_heading=h.2dlolyb)

[Exceptions Process 18](#_heading=h.sqyw64)

[Distribution 19](#_heading=h.3cqmetx)

[Policy Change Control 19](#_heading=h.1rvwp1q)

[Policy Version History 19](#_heading=h.4bvk7pj)

[Appendix A Lookups 19](#_heading=h.2r0uhxc)

[Approved C/C++ Language Acronyms 19](#_heading=h.1664s55)

## 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 | Unvalidated user input is one of the most common causes of exploits and bugs in a coding project, especially in lower level languages such as C and C++ where manual memory management is possible. Input validation can prevent common exploits such as buffer/integer underflow/overflow. For any system that requires even a minimum level of security, allowable parameters should be set for every user input. Any input that is outside the allowable parameters should be rejected. |
| 1. Heed Compiler Warnings | Compiler warnings are one of the most useful tools developers have for detecting potential issues in their code. These compilers are designed by hundreds of contributors over decades of development and testing. Every compiler warning is there for a good reason and any compiler warnings generated during the build process should be handled. The general goal is of course zero compiler warnings, but if that is not possible then every compiler warning should have a detailed reason for why the warning is allowed. |
| 1. Architect and Design for Security Policies | The first step to having a secure system is to design and plan for that system’s security. This involves designing for security first rather than security being an afterthought. It is much more difficult to rewrite a system to fix a security vulnerability rather than designing the system to prevent that vulnerability in the first place. There is a 1-10-100 principle in software development that states that what took 1x effort to fix in development will take 10x effort to fix in testing, and 100x effort to fix in production. Developing with security in mind prevents future headaches and wasted time. |
| 1. Keep It Simple | Complexity is inherently harder to understand. When possible a simple and stable solution to a problem is preferred to a complex solution that might be marginally faster. Additionally wherever possible and secure, using the standard library is preferable to re-implementing standard library functionality. Vulnerabilities in the standard library are typically well-known and avoidable with millions of programmers having utilized its functions. Although the original author might be able to understand complex solutions, this makes it harder for outside programmers to understand the program and extend it securely. |
| 1. Default Deny | By default all access to a service is denied. The only allowable access is from entities and functions that are specifically whitelisted. This ensures that only trusted entities can access the service, as the default is to deny access to the service. This is in contrast to default allow in which any entity can access a service unless it is explicitly disallowed. The purpose of default deny is to reduce the available attack surface on a service by limiting the number and type of entities that can access it. |
| 1. Adhere to the Principle of Least Privilege | Adhering to the principle of least privilege is the concept that for any entity only the least level of privilege should be given to that entity to perform the tasks it needs to. This is the same principle regardless of whether that entity is an application, service, user, or other entity. This helps prevent a common issue where an entity is able to escalate its privileges and gain access to parts of the system it shouldn’t have. In the case of a rogue actor or malware this can be catastrophic. |
| 1. Sanitize Data Sent to Other Systems | When sending data to other systems the data should be sanitized and formatted in a way that is secure and acceptable by the other system. While a certain level of security could be assumed during program execution, this can not be guaranteed during the transit of data or once that data reaches an outside system. An example of this is a user’s password input. While the password has to be taken from the user in plain-text format it should not be transmitted to an authentication server in that plain-text format. There could be bad actors listening to the data being transmitted, or the authentication server could be compromised in some way. When data has left a secure system, it should be assumed to be no longer secure. |
| 1. Practice Defense in Depth | Defense in Depth is the practice of utilizing multiple layers of security to provide redundant security in the case that one layer of security fails there is another layer of security ready to take its place. For each additional layer of depth additional security is provided but at the cost of additional time, technical resources, manpower, and complexity. Where there is no depth in security, a single vulnerability could result in the security of the entire system being compromised. It is important to implement Defense in Depth to ensure the overall security of a system. |
| 1. Use Effective Quality Assurance Techniques | Quality Assurance is one of the most important steps of development to ensure that major issues don’t reach production and potentially cause major security vulnerabilities. Some effective quality assurance techniques might include: manual systems testing, static code analysis, and code reviews. It is important that code is reviewed for security vulnerabilities or functional bugs before being committed to production. The systems in place to prevent potentially dangerous code from reaching production is just as important as every coding standard implemented. |
| 1. Adopt a Secure Coding Standard | None of the above security principles are useful if they aren’t actually implemented and enforced. This is where a secure coding standard is useful: A set of guidelines to be followed when coding a project to ensure quality and security. These sets of secure coding standards are even more useful where their implementation is automated through the use of continuous integration tools. |

### 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 | [Do not use floating-point variables as loop counters](https://wiki.sei.cmu.edu/confluence/display/c/FLP30-C.+Do+not+use+floating-point+variables+as+loop+counters)  Using a floating-point variable as a loop counter can cause many unintended side effects due to the nature of floating-point numbers’ imprecision. Loops could have platform dependent number of iterations when using floating-point loop counters, or could vary from what the developer intended. |

| **Noncompliant Code** |
| --- |
| Using a floating-point variable as a loop counter can have unknown behavior where the loop loops an inconsistent number of iterations based on the platform compiled and executed on. The following example might execute 4 or 5 times. |
| for (float i = 0.2f; i <= 1.0f; i += 0.2f)  {  //do something  } |

| **Compliant Code** |
| --- |
| Replacing the floating-point variable with a fixed-size unsigned integer variable ensures that there are no possible variations in loop iterations between different platforms or compilers. |
| for (uint8\_t i = 2; i <= 10; i += 2)  {  //do something  } |

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

| **Principles(s):**  Keep It Simple  While it is possible to use floating point types as iterator values, their use is unpredictable and potentially exploitable. By keeping it simple and using the commonly agreed upon iterator types these issues can be avoided. There are commonly agreed upon standards for a good reason. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PC-lint Plus | 1.4 | 9009 | Static analysis tool that automatically checks c and cpp programs for known issues. |
| Clang | 3.9 | cert-flp30-c | Checks during compile time whether the issue is present or not. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | [Ensure that unsigned integer operations do not wrap](https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap)  Unsigned integers should never be allowed to wrap during arithmetic operation. This can result in unintended behavior where the result of an operation is smaller than its two inputs. |

| **Noncompliant Code** |
| --- |
| In the following code 20 is added to the input number. However if the input number is greater than 235 then the result will wrap around to 0-24. This could lead to unintended behavior depending on how the function is used. |
| uint8\_t add\_20(uint8\_t num)  {  return num + 20;  } |

| **Compliant Code** |
| --- |
| In the following code a check is made before adding the two numbers to see if wrapping will occur. If wrapping occurs, the error should be handled immediately. |
| uint8\_t add\_20(uint8\_t num)  {  if (numeric\_limits<uint8\_t>::max() - 20 < num)  {  //do something! Integer wrapping will occur!  }  return num + 20;  } |

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

| **Principles(s):**  Architect and Design for Security Policies  While it can be difficult for a reasonably sized unsigned integer to wrap, the consequences of such a wrap-around can be dire including undefined behavior and incorrect values shown to users. A common exploit is on systems that utilize a 32-bit ticking counter for timing events; eventually the 32-bit ticker will wrap around to 0. This can cause events to occur early or not at all. Rather than allowing this at all, the system should be designed with this in mind from the beginning to prevent the issue before it ever happens. Retrofitting a complex program to no longer allow integer wrapping can be time consuming and costly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | Very High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Static analysis tool available in a variety of programming languages intended to be included in the continuous integration process. |
| TrustInSoft Analyzer | 1.38 | unsigned overflow | Software security static analysis tool. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | [Guarantee that storage for strings has sufficient space for character data and the null terminator](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)  Failing to guarantee that storage for strings has sufficient space results in one of the most well known and exploitable vulnerabilities a program can have: a buffer overflow. This can result in attackers altering the execution of code in the program to whatever they desire under the correct circumstances. |

| **Noncompliant Code** |
| --- |
| In the following function no check is made to see if str\_buffer can contain the full contents of input\_string. In the case that input\_string has a size greater than 10, a buffer overflow will occur which can cause unintended behavior and result in security vulnerabilities. |
| char str\_buffer[10];  void move\_into\_buffer(char\* input\_string)  {  strcpy(str\_buff, input\_string);  } |

| **Compliant Code** |
| --- |
| In the following function the strcpy function is replaced with the strncpy function which prevents the possibility of a buffer overflow. |
| char str\_buffer[10];  void move\_into\_buffer(char\* input\_string)  {  strncpy(str\_buff, input\_string, sizeof(str\_buff) - 1);  } |

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

| **Principles(s):**  Architect and Design for Security Policies  A buffer overflow attack can be catastrophic, preventing it from ever happening in the first place can save the organization from expensive refactoring and damage control. Buffer overflow exploits are a common attack vector among security breaches in low-level programs.  Validate Input Data  Functions like strcpy() can still be used in programs, but only where it is absolutely certain that there is enough memory in the output buffer at compile time. Any variable-sized strings should have operations bound to their size.  Keep It Simple  It is well known that strcpy() and other buffer copying methods are susceptible to buffer overflow. Keep it simple and use the commonly accepted standard library alternatives such as strncpy() and snprintf(). |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | Static analysis tool available in a variety of programming languages intended to be included in the continuous integration process. |
| Cppcheck | 2.14 | testbufferoverrun | Automatically checks strcpy() among other buffer copying methods |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | [Prevent SQL injection](https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection)  SQL Injection can result in catastrophic security vulnerabilities for systems that handle sensitive information. It is incredibly important that user input is sanitized before being used for an SQL query. |

| **Noncompliant Code** |
| --- |
| In the following code an SQL injection is possible because no input sanitation or validation is performed on the username variable. The username variable could potentially contain “‘JIM’ or ‘2’=’2’” which would result in all USERS being returned. |
| std::ostringstream sql\_query;  sql\_query << “SELECT \* FROM USERS WHERE NAME=” << username << “;”;  query(sql\_query.str()); |

| **Compliant Code** |
| --- |
| A compliant implementation of an SQL query should implement parameterized inputs for a specific query. The following example uses the parameter username. This API is one of many that will prevent SQL injections by sanitizing the inputs. |
| sql::Connection \*con;  sql::PreparedStatement \*prep\_stmt  prep\_stmt = con->prepareStatement(“SELECT \* FROM USERS WHERE NAME=?;”;  prep\_stmt->setString(1, username);  prep\_stmt->execute(); |

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

| **Principles(s):**  Validate Input Data  SQL Injection attacks are only possible when the user is allowed to input whatever they please without its value being sanitized. SQL queries should never be sent unchecked from client to database server as it could be manipulated before reaching the server. Even if input checking and sanitation is present in client-side code, this code can be modified by savvy attackers.  Adhere to the Principle of Least Privilege  When designing a secure SQL database, preventing authorized access and attacks should be one of the first security priorities. Users should only be allowed to submit queries that are absolutely needed for their legitimate use cases. These queries should be constructed on the server-side with rigorously checked parameterized inputs. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2024.2 | SV.DATA.DB  SV.SQL  SV.SQL.DBSOURCE | Real-time static analysis tool meant to run alongside an IDE. |
| SonarQube | 9.9 | S2077  S3649 | Static analysis tool designed to comply with software security standards. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | [Do not access freed memory](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory)  Only memory that is allocated should be accessed. Accessing memory that has already been freed will have unpredictable contents and the behavior of the program could be compromised. |

| **Noncompliant Code** |
| --- |
| The following code attempts to access the integer s after it has already been deallocated, resulting in unintended behavior. This could cause catastrophic issues in a program as the contents of s will be unpredictable. |
| int\* s = new int(5);  delete(s);  \*s += 5;  std::cout << \*s << endl; |

| **Compliant Code** |
| --- |
| In the revised code below the integer s remains allocated until it is no longer needed and then the memory is freed. |
| int\* s = new int(5);  \*s += 5;  std::cout << \*s << endl;  delete(s); |

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

| **Principles(s):**  Architect and Design for Security Policies  Memory management can not be an afterthought to a program’s design. Instead it should be tackled head on as development progresses. The usage of smart pointers can simplify this process. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Code compiler that can check for some memory allocation issues at compile time. |
| CodeSonar | 8.1p0 | ALLOC.UAF | Static analysis tool available in a variety of programming languages intended to be included in the continuous integration process. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | [Do not dereference null pointers](https://wiki.sei.cmu.edu/confluence/display/c/EXP34-C.+Do+not+dereference+null+pointers)  Before dereferencing any pointer a check should be made to ensure the pointer is not a null pointer. Attempting to dereference a null-pointer could have unintended side-effects that could be exploited by attackers. |

| **Noncompliant Code** |
| --- |
| In the following code example the contents of results are accessed without checking if the variable result references a null pointer. This could result in an illegal memory access. |
| void add\_function(int\* result, int num)  {  \*result += num;  } |

| **Compliant Code** |
| --- |
| The following code example checks if the variable result references a null pointer before accessing its contents. This prevents unallowable memory accesses. |
| void add\_function(int\* result, int num)  {  if (result == nullptr)  {  //do something!  return;  }  \*result += num;  } |

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

| **Principles(s):**  Validate Input Data  One of the most common uses for pointers is to pass along data to functions without having to perform large memory copies. Instead of potentially copying structures that are multiple KB in size, a single 4-8 byte pointer can be passed along. When using pointers for this purpose among many others, it is important to ensure the pointer actually points somewhere. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | Critical | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.MEM.NPD  LANG.STRUCT.NTAD  LANG.STRUCT.UPD | Static analysis tool available in a variety of programming languages intended to be included in the continuous integration process. |
| Cppcheck | 1.66 | nullPointer,  nullPointerDefaultArg,  nullPointerRedundantCheck | C++ Static analysis tool. In this context is relevant in that it checks for when a pointer is being dereferenced without determining first if it is null. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | [Handle all exceptions](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions)  Exceptions are thrown to communicate to the program when a potentially fatal error has occurred. These exceptions should be handled to be able to determine if the program can recover from the exception and if not the program should exit gracefully. |

| **Noncompliant Code** |
| --- |
| In the following code example an exception is thrown but not caught. This could result in the program terminating in an unexpected way. |
| void throw\_exception()  {  throw std::logic\_error(“Generic Exception!”);  }  int main()  {  throw\_exception();  return 0;  }; |

| **Compliant Code** |
| --- |
| The following code example correctly catches all exceptions and can terminate in a normal fashion. |
| void throw\_exception()  {  throw std::logic\_error(“Generic Exception!”);  }  int main()  {  try  {  throw\_exception();  }  catch (...)  {  //handle the error!  }  return 0;  }; |

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

| **Principles(s):**  Use Effective Quality Assurance Techniques  Exceptions are a powerful tool to control the flow of a program when something unexpected happens. Rather than having potentially undefined behavior or an instant crash, exception handling allows a program to determine if it's possible to recover and if not then it can exit gracefully. This can be used as an effective quality assurance tool to prevent undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Static analysis tool available in a variety of programming languages intended to be included in the continuous integration process. |
| Klocwork | 2024.2 | MISRA.CATCH.ALL | Real-time static analysis tool meant to run alongside an IDE. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Passphrases** | STD-008-CPP | [Never hard code sensitive information](https://wiki.sei.cmu.edu/confluence/display/c/MSC41-C.+Never+hard+code+sensitive+information)  Hard-coding sensitive information or storing that information in plain-text format is one the most insecure forms of security. Prying eyes are able to easily view the context of the text and potentially use that information to gain access to the secure system. |

| **Noncompliant Code** |
| --- |
| In the following block of code the username and password “fred” and “football123” are hardcoded string literals. This presents a serious security vulnerability because anyone who has access to the executable can read these raw strings easily to gain unauthorized access to the system. |
| bool authenticate(std::string input\_username, std::string input\_password)  {  if ( (strcmp(input\_username, “fred”) == 0) && (strcmp(input\_password, “football123”) )  {  return true; //access granted  }  return false; //access denied  } |

| **Compliant Code** |
| --- |
| In the following block of code rather than accepting raw string inputs and comparing them to constant string literals the paradigm is changed to instead compare a hashed input to a known hash. Typically these known hashes would be retrieved from a file. This prevents would-be attackers from being able to inspect the contents of the executable to easily see authentication details. |
| bool authenticate(std::string input\_username\_hash, std::string input\_password\_hash)  {  if ( (strcmp(input\_username\_hash, username\_hash) == 0) && (strcmp(input\_password\_hash, password\_hash) )  {  return true; //access granted  }  return false; //access denied  } |

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

| **Principles(s):**  Architect and Design for Security Policies  How passwords and other sensitive information are stored is one of the most important characteristics for a secure system. Once that sensitive information is stolen, the damage is already done. It can be extremely hard for users to recover from identity theft and the damage to a company's reputation can be irreparable.  Sanitize Data Sent to Other Systems  Passwords and other sensitive information should never be transmitted across programs or networks in plain-text format. If there needs to be a comparison for authentication, the comparison should be done on two hashed/encrypted values. Although two systems might be completely secure, data can be targeted during transmission through a myriad of attack vectors. Special care should be taken to prevent any information gleaned during transmission from being useful to attackers. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | Very High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2024.2 | HCC  HCC.PWD  HCC.USER  CXX.SV.PWD.PLAIN  CXX.SV.PWD.PLAIN.LENGTH  CXX.SV.PWD.PLAIN.ZERO | Real-time static analysis tool meant to run alongside an IDE. |
| CodeSonar | 8.1p0 | HARDCODED.AUTH  HARDCODED.DNS  HARDCODED.KEY  HARDCODED.SALT  HARDCODED.SEED | Static analysis tool available in a variety of programming languages intended to be included in the continuous integration process. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Files** | STD-009-CPP | [Close files when they are no longer needed](https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed)  The life-cycle of a file should be limited to when its contents are being written and read. Immediately after these operations are complete, the file should be closed to allow other processes or threads access to the file. |

| **Noncompliant Code** |
| --- |
| The following code example opens up a file, writes “Hello World!\n” to that file, and continues code execution. The file remains open, potentially disallowing other processes from accessing that file. |
| void open\_file(char\* file\_name)  {  FILE \*fptr;  fptr = fopen(file\_name, “w”);  fprintf(fptr, “Hello world!\n”);  } |

| **Compliant Code** |
| --- |
| This revised code properly closes the file after all file operations are complete before continuing code execution. |
| void open\_file(char\* file\_name)  {  FILE \*fptr;  fptr = fopen(file\_name, “w”);  fprintf(fptr, “Hello world!\n”);  fclose(fptr);  } |

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

| **Principles(s):**  Architect and Design for Security Policies  Correctly initializing files and closing them when done is an easy yet important part of manipulating files. Ensuring this process is followed allows for other programs and threads to access the file. Additionally leaving a file open forever could leave file buffers available for viewing access when the program exits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | ALLOC.LEAK | Static analysis tool available in a variety of programming languages intended to be included in the continuous integration process. This checker will check for file allocation leaks whenever there is a file open call but no file close call. |
| Klocwork | 2024.2 | RH.LEAK | Real-time static analysis tool meant to run alongside an IDE. Similar checker functionality to the CodeSonar checker. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Iteration** | STD-010-CPP | [Use valid iterator ranges](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges)  Any time an array is iterated over, the range should always be within the bounds of the array by checking the size of the array. This ensures that only valid elements of the array are accessed. Attempting to access illegal elements of an array won’t always result in a compilation or run-time error, so special care should be taken to ensure it doesn’t happen. |

| **Noncompliant Code** |
| --- |
| The example provided assumes the vector has a size of 100, but if the vector has less than 100 elements the for loop will attempt to access an out of bounds element of the vector. If the vector has more than 100 elements then some elements won’t be accessed. |
| void vector\_increment\_all(const std::vector<uint8\_t> &v)  {  for (int i = 0; i < 100; i++)  {  v.at(i) += 1;  }  } |

| **Compliant Code** |
| --- |
| Rather than using a hardcoded value, the vector is iterated based on its size. This ensures that every element is accessed. |
| void vector\_increment\_all(const std::vector<uint8\_t> &v)  {  for (int i = 0; i < v.size(); i++)  {  v.at(i) += 1;  }  } |

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

| **Principles(s):**  Validate Input Data  Every iterator should be bounded by the data it is iterating over. If this is not done, it allows for garbage data to be accessed, with potentially undefined behavior. Validating this to be true is simple, yet should be done for every iterator. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.MEM.BO | Static analysis tool available in a variety of programming languages intended to be included in the continuous integration process. Checks for buffer overrun in iteration. |
| Astrée | 22.10 | overflow\_upon\_dereference | Static analysis tool specializing in customized runtime errors. Specializes in complex memory analysis. |

### 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.

Automated security testing and scanning is often best introduced during both the build process and verification and testing process depending on the purpose of the automation. Often static code analysis and compile time checks can be included in both the build and verify and test phases of pre-production. The developers creating a pre-production build can run compile time checks and static analysis tools as they are developing the program. Once the program is ready to be verified, it should be ran against static analysis tools again before being verified by peers. Automating this process helps ensure compliance. The most important aspect of automating the compliance of a coding and security standards policy is that its enforcement must be done *before* a program reaches production. If security testing is only done after a program reaches production, the process has failed.

### 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](#_heading=h.3rdcrjn) | High | Likely | High | Medium | 2 |
| [STD-002-CPP](#_heading=h.26in1rg) | High | Likely | High | High | 2 |
| [STD-003-CPP](#_heading=h.lnxbz9) | High | Likely | Medium | Critical | 1 |
| [STD-004-CPP](#_heading=h.35nkun2) | High | Likely | Medium | Critical | 1 |
| [STD-005-CPP](#_heading=h.1ksv4uv) | High | Likely | Medium | Critical | 1 |
| [STD-006-CPP](#_heading=h.2jxsxqh) | High | Likely | Medium | Critical | 1 |
| [STD-007-CPP](#_heading=h.z337ya) | Low | Probable | Medium | Low | 3 |
| [STD-008-CPP](#_heading=h.28h4qwu) | High | Probable | Medium | Very High | 5 |
| [STD-009-CPP](#_heading=h.nmf14n) | Medium | Unlikely | Medium | Low | 3 |
| [STD-010-CPP](#_heading=h.37m2jsg) | High | Probable | High | Medium | 2 |

### 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 | All sensitive data at rest should be encrypted in a high-security and battle-tested algorithm such as AES-256. This ensures that the data is safe from prying eyes.  Encryption at rest targets data that is not actively being used. Whether this is stored in a file system or a distributed database. This data typically has a longer life cycle of anywhere from a day to forever. This is the data that the majority of major data breaches target. Large scale database of customer information among other things. |
| Encryption in flight | All sensitive data in flight should be encrypted in a safe and secure format. Often this is a combination of RSA and AES-256. This format allows for a safe exchange of AES keys encrypted using the public/private key protocol of RSA. What is important is that information being transmitted can not be decrypted except by the two parties involved.  Encryption in flight targets data that is actively being transmitted over a network or other systems. This data’s life-cycle is only as long as it takes to transmit the data: typically a second or less. This form of data is less often targeted during data breaches as not all user information is being transmitted all the time. Rather data in flight is more often the target of man-in-the-middle attacks and attacks targeting specific individuals. |
| Encryption in use | After the data is done being initially inputted and sanitized, the data should be encrypted in a short-term high security format such as AES-128. This will prevent the contents of the program from being read during runtime, revealing sensitive information.  Encryption in use targets data that is actively being used by either the client or server. This data’s life cycle is typically the life cycle of the program. Often it can be hard to keep all data in use encrypted as data typically has to be entered in a plaintext format to begin with. This opens the user up to keyloggers or external programs reading the contents of the program during execution. Attacks of this nature will typically only affect individuals, not all users. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures that entities are who they claim to be. This is usually done through an identity and access management (IAM) login process through password protected accounts often with two-factor authentication. This helps personalize data and ensure that entities only have access to data that is relevant to them. Users typically are not allowed to see sensitive information relevant to other users. |
| Authorization | Authorization is the act of allowing authenticated entities access to specific functionality of a program. This functionality should always be limited to only the functionality that is necessary for their use-case as defined in the principle “Adhere to the Principle of Least Privilege”. This is useful for preventing authorized users from accessing parts of a system that they shouldn’t need to access. This can also be used to give different users different levels of access. Ordinary users can have different levels of access to administrators. |
| Accounting | Accounting is used to log the actions that entities perform and what information they access. Proper accounting is useful for both preventing and recovering from security breaches. All access and changes to sensitive information should generate an entry in a log somewhere. The lifecycle of this log should be determined by the security needs of the organization. In the event of a security breach, extensive logs allow for a company to determine who accessed what data and when they accessed it. Often this is used to see who accessed or made changes to a database, or what specific files were being accessed by a user. Without this information, it is impossible to tell what an attacker viewed or modified, or if there ever was an attacker. |

**\***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 | 07/28/2024 | Revision 1 | Nate Ludwig |  |
| 1.2 | 08/17/2024 | Revision 2 | Nate Ludwig |  |
| 1.21 | 08/23/2024 | Revision 3 | Nate Ludwig |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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