**Green Pace Developer: Security Policy**



# 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 | All input should be checked for things like type, under/overflow, and should be sterilized to prevent crashes and breaches due to bad input. Bad input should be handled, and the user prompted for new input/the problem reported to admins. |
| 1. Heed Compiler Warnings | All compiler warnings should be addressed and resolved. Compiler warnings should never be ignored or dismissed outside the development stage. |
| 1. Architect and Design for Security Policies | When designing programs, classes, functions, or any standalone part of the overarching project—all security measures should be considered, and any potential vulnerabilities should be addressed. If at any point in development new potential vulnerabilities are found, they should be addressed, and the design should be re-assessed to take the newfound vulnerabilities into account. |
| 1. Keep It Simple | The simplest solution should be considered the best solution—barring dangerous vulnerabilities. |
| 1. Default Deny | The default case for inbound and outbound traffic should be to deny said traffic. |
| 1. Adhere to the Principle of Least Privilege | Users should *only* have privileges that meet the minimum requirements for them to complete their tasks. |
| 1. Sanitize Data Sent to Other Systems | All outbound data should be sanitized to prevent any unintended information from being leaked. |
| 1. Practice Defense in Depth | All security measures should be built up in layers so that no one vulnerability can cause a breach of the system as a whole. |
| 1. Use Effective Quality Assurance Techniques | Quality Assurance should not only account for user experience, but also for malicious users attempting to break/breach the system. |
| 1. Adopt a Secure Coding Standard | Make security a primary goal in all facets of the development process. Pride yourself in developing well written, secure code by default. |

### 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] | All data should be assigned the **most applicable** type. |

| **Noncompliant Code** |
| --- |
| Use of non-applicable type for storing Boolean value. |
| Std::String factChecker = “True”;  if(factChecker == “True”)  {  doSomething(); } |

| **Compliant Code** |
| --- |
| Use of most-applicable type for storing Boolean value. |
| bool factChecker = true;  if(factChecker)  {  doSomething();  } |

| **Principles(s):**  4) Keep It Simple: Using a more complex type such as a string when a Boolean would suffice is makes the program unnecessarily more complex than it needs to be.  10) Adopt a Secure Coding Standard: Adopting a secure coding mindset includes sticking to best practices. Using most applicable types would be considered a best practice. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Low | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Custom Tool | -- | -- | The tool could check for things like strings holding values like “True” and “False”. It could also check for things like a double being used when the value could be accurately held by a float. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | When assigning data values, care should be taken to ensure the target variable meets the needs of the data being set. |

| **Noncompliant Code** |
| --- |
| Assigns a floating-point value to a variable of type integer resulting in loss of data |
| int result;  result = 10/7; |

| **Compliant Code** |
| --- |
| Assigns the floating-point value to a variable that can handle floating-point values with negligible loss |
| float result;  result = 10/7; |

| **Principles(s):**  9) Use Effective Quality Assurance Techniques: Loss of accuracy may lead to incorrect output, affecting the program’s reliability. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2022b | CERT C: Rule INT35-C | Checks for situations when integer precisions are exceeded (rule fully covered) |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to modify string literals |

| **Noncompliant Code** |
| --- |
| Attempting to modify a pointer to an address of a string literal results in undefined behavior. |
| char \*str = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| Str can now be modified safely because a copy of the string literal has been made in the location of the character string. |
| char str[] = "string literal";  str[0] = 'S'; |

| **Principles(s):**  3) Architect and Design for Security Policies: using string literals in places where the value needs to be modified should be avoided. The design should take into account if a future function will need to modify the value.  10) Adopt a Secure Coding Standard: understanding the difference between string literals and character arrays is important when complying with standards of best practice. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | string-literal-modfication  write-to-string-literal | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR30 | Fully implemented |
| Helix QAC | 2022.3 | C0556, C0752, C0753, C0754  C++3063, C++3064, C++3605, C++3606, C++3607 |  |
| Parasoft C/C++test | 2022.1 | CERT\_C-STR30-a  CERT\_C-STR30-b | A string literal shall not be modified  Do not modify string literals |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | All queries should be pre-compiled/prepared when possible. |

| **Noncompliant Code** |
| --- |
| SQL query is raw and subject to SQL injection. |
| String query = “SELECT sensitive\_rows FROM table WHERE name = “Joe””;  runQuery(query); |

| **Compliant Code** |
| --- |
| SQL query is prepared, preventing SQL injection |
| String query = “SELECT sensitive\_rows FROM table WHERE name = ?”;  //prepare statement  Sqlite3\_stmt\* stmt;  Sqlite3\_prepare\_v2(database, query.c\_str(), query.length(), &stmt, nullprt);  //bind parameter  String name = “Joe”;  Sqlite3\_bind\_text(stmt, 1, name.c\_str(), name.length(), SQLITE\_STATIC);  //call query  Sqlite3\_step(stmt); |

| **Principles(s):**  1) ValidateInput Data: input data should be sanitized before being passed into statements.  3) Architect and Design for Security Policies: functions executing SQL should use prepared statements by design |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported via stubbing/taint analysis |
| PRQA QA-C++ | 4.4 | 4916, 4917, 4918 |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Any time new memory is allocated, it should also be deallocated using a delete() function or destructor method. |

| **Noncompliant Code** |
| --- |
| New is called but not cleaned up |
| Int\* pointer = new int(10);  …  …  …  Return 0; |

| **Compliant Code** |
| --- |
| New is called and the data memory is freed at the end of the program |
| Int\* pointer = new int(10);  …  …  …  Delete(pointer);  Return 0; |

| **Principles(s):**  9) Use Effective Quality Assurance Techniques: Quality assurance would ensure the program contained no memory leaks for performance as well as security reasons.  10) Adopt a Secure Coding Standard: It is best practice to deallocate memory when memory is manually allocated |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2022.3 | C++4761, C++4762, C++4766, C++4767 |  |
| PVS-Studio | 7.21 | V630, V749 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions should be used throughout development in order to catch and resolve as many bugs as possible prior to release. |

| **Noncompliant Code** |
| --- |
| Code assumes that our “changeString” method from coding standard 3 doesn’t change the original string. |
| String s = “This string is fun”;  String s2 = changeString(s);  Cout << s << s2 << endl; |

| **Compliant Code** |
| --- |
| Code ASSERTS that our “changeString” method doesn’t change the original string |
| String s = “This string is fun”;  String s2 = changeString(s);  Assert(s == “This string is fun”);  Cout << s << s2 << endl; |

| **Principles(s):**  9) Use Effective Quality Assurance Techniques: use assertions throughout development to ensure requirements are met and bugs are caught |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2020.2 | ERR56-CPP | Parasoft C/C++test |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exceptions should be used whenever an error is expected to be possible to keep the program in a stable state. |

| **Noncompliant Code** |
| --- |
| Code doesn’t handle divide by 0 exception |
| Double divide(int a, int b)  {  Return (a/b);  }  c = divide(x, y); |

| **Compliant Code** |
| --- |
| Code properly handles exceptions within function and during function calling |
| Double divide(int a, int b)  {  If(b == 0)  Throw “Divide by zero exception”;  Return (a/b);  }  try {  C = Divide(x/y); }  Catch(exception e)  {  cout << e.what();  } |

| **Principles(s):**  3) Architect and Design for Security Policies: Designs should consider all possible exceptions to ensure the program maintains a stable state  9) Use Effective Quality Assurance Techniques: To ensure the system doesn’t completely crash errors should be handled |
| --- |

**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 |
| Parasoft C/C++test | 2022.1 | 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 | R2022b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Comments** | [STD-008-CPP] | Header comments for functions, classes, and methods should always be included. Inline comments should be included for any unintuitive code. This is very important for maintainability and reproducibility. |

| **Noncompliant Code** |
| --- |
| Lack of purposeful commenting |
| int fact(int a)  {  If(a == 1)  Return a;  Else  Return(a \* fact(a – 1));  } |

| **Compliant Code** |
| --- |
| Commenting explains the function |
| //this function returns the factorial of the number given  int fact(int a)  {  If(a == 1)  Return a;    //recurse to final result  Else  Return(a \* fact(a – 1));  } |

| **Principles(s):**  10) Adopt a Secure Coding Standard: In accordance with the standards of best practice, commenting throughout code of the developer’s mindset and methods will promote maintainability in code as well as communicate reasoning for things—possibly preventing future vulnerabilities due to misunderstanding how previously written code functions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Low | Low | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Custom Tool | -- | -- | The tool could simply check if functions are preceded by header comments. It could also check for an arbitrary amount of comments per # of lines of code |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Buffer overflow Prevention | [STD-009-CPP] | Buffer overflow always be protected against when taking input. |

| **Noncompliant Code** |
| --- |
| Doesn’t regulate input passed in to length-limited type |
| Char input[20];  Cout << “enter name: “ << endl;  Cin >> input; |

| **Compliant Code** |
| --- |
| Regulates input |
| Char input[20];  Cout << “enter name: “ << endl;  //limit input to 20 characters  Cin >> setw(20) >> input;  //clear input stream  Cin.ignore (std::numeric\_limits<std::streamsize>::max(), '\n'); |

| **Principles(s):**  1) ValidateInput Data: All input should be considered malicious  8) Practice Defense in Depth: Buffer overflow is one vulnerability that leads to data breaches and should be protected against as a first line of defense  10) Adopt a Secure Coding Standard: The standard should be to always protect from buffer overflow |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stream-input-char-array | Partially checked + soundly supported |
| Parasoft C/C++test | 2022.1 | 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 | R2022b | 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. |
| RuleChecker | 22.10 | stream-input-char-array | Partially checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Type overflow and underflow prevention | [STD-010-CPP] | Type overflow and underflow should always be protected against when using input as parameters |

| **Noncompliant Code** |
| --- |
| Code uses input as parameter without any consideration for type limits |
| //T is some *type*  T sequentialAdd(T start, T increment, int iterations)  {  T localStart = start;  For(int i = 0; i < iterations; i++)  {  localStart = localStart + increment;  }  Return localStart;  } |

| **Compliant Code** |
| --- |
| Code throws an exception If overflow / underflow happens |
| //T is some *type*  T sequentialAdd(T start, T increment, int iterations)  {  T localStart = start;  For(int i = 0; i < iterations; i++)  {  //check numeric limit for underflow  if(decrement > 0 && result < (std::numeric\_limits<T>::min() + decrement))  {  throw std::underflow\_error("UNDERFLOW OCCURRED");  }    // check for overflow)  if (decrement < 0 && result > (std::numeric\_limits<T>::max() + decrement))  {  throw std::overflow\_error("OVERFLOW OCCURRED”);  }  else  {  localStart = localStart + increment  }  }  Return localStart;  } |

| **Principles(s):**  8) Practice Defense in Depth: In the case that malicious input somehow makes it through to be used as an iterator—the code should defend against type overflow/underflow  10) Adopt a Secure Coding Standard: The default standard should be to prevent type overflow/underflow |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | integer-overflow | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT30 | Implemented |
| CodeSonar | 7.1p0 | 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 | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| Compass/ROSE |  |  | Can detect violations of this rule by ensuring that operations are checked for overflow before being performed (Be mindful of exception INT30-EX2 because it excuses many operations from requiring validation, including all the operations that would validate a potentially dangerous operation. For instance, adding two unsigned ints together requires validation involving subtracting one of the numbers from UINT\_MAX, which itself requires no validation because it cannot wrap.) |

### Defense-in-Depth Illustration

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





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.

**Automation should be implemented *at the latest* during the “Build” phase. This is not to say that it should be contained completely in the “Build” phase, however—testing automation can be a constant in the lifecycle of the software, even behind the scenes in releases.**

**Beginning automation in the build phase will help ensure that security requirements and principles are being considered and implemented throughout the entire process. If a program is built upon a foundation that conforms to security standards, there is a much smaller chance for vulnerabilities to slip through the cracks down the line incurring large remediation costs.**

**Continuing automation throughout the testing phase will continue to ensure all elements of the program conform to the set standards. Vulnerabilities being caught late are tremendously better than vulnerabilities not caught at all. This philosophy continues throughout released versions where performance allows.**

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Probable | Low | Low | 3 |
| STD-002-CPP | Medium | Probable | Low | Low | 3 |
| STD-003-CPP | Low | Likely | Low | Medium | 2 |
| STD-004-CPP | High | Probable | Medium | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | High | Likely | High | Medium | 2 |
| STD-007-CPP | Low | Probable | Medium | Low | 3 |
| STD-008-CPP | Low | Low | Low | Low | 3 |
| STD-009-CPP | High | Likely | Medium | Low | 1 |
| STD-010-CPP | High | Likely | High | Medium | 2 |

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Data at rest should be encrypted using the Advanced Encryption Standard (AES). Considered by the National Institute of Standards and Technology to be the advanced standard, AES is a very robust encryption method. This makes it a good method for data at rest. This will ensure any resting data is properly protected by industry standard means and will not be at risk for data breaches. |
| Encryption at flight | AES is sometimes used for data in flight as well, but with the amount of data being transferred the encryption method should be a combination of Twofish and the AES methods. Twofish should be used to encrypt data in flight as it is quite fast, and to practice DiD, Twofish encryption/decryption keys should be encrypted using the much slower AES method. This simply adds an additional layer of protection for our users’ data while not sacrificing performance. The encryption of keys in transit also prevents any data breaches via data interception. |
| Encryption in use | AES should also be used to encrypt data in use. As it is the standard, we also consider it our standard. As data is being read, created, modified—it should be considered no different from data in other states due to our default deny and least privilege principles. This will protect data as it is being modified, hopefully closing all possible doors to data breach. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Two Factor Authentication (2FA) should be required of all users as a DiD practice, ensuring if ever a user’s password is somehow compromised—their account will not be able to be accessed by that password alone. |
| Authorization | Following with the principle of least privilege, and authentication. Users should only be authorized access to data that is pertinent to them and their uses of the system. This includes customer users and company users. |
| Accounting | User information and behavior within the system should be monitored for suspicious activity but should only be logged if suspicious activity is detected. This will enhance performance as well as ensuring sensitive information is not logged without purpose. Logging suspicious activity will alert the owners to potential breaches and allow for the notification to users that their passwords should be changed. This enhances our DiD approach by adding another layer of security beyond our 2FA. |

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 11/13/22 | Milestone | Zachary Mohler |  |
| 2.0 | 12/4/22 | Project 1 | Zachary Mohler | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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