**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 5](#_Toc52464059)

[Coding Standard 2 7](#_Toc52464060)

[Coding Standard 3 9](#_Toc52464061)

[Coding Standard 4 11](#_Toc52464062)

[Coding Standard 5 13](#_Toc52464063)

[Coding Standard 6 15](#_Toc52464064)

[Coding Standard 7 17](#_Toc52464065)

[Coding Standard 8 19](#_Toc52464066)

[Coding Standard 9 21](#_Toc52464067)

[Coding Standard 10 23](#_Toc52464068)

[Defense-in-Depth Illustration 25](#_Toc52464069)

[Project One 25](#_Toc52464070)

[1. Revise the C/C++ Standards 25](#_Toc52464071)

[2. Risk Assessment 25](#_Toc52464072)

[3. Automated Detection 25](#_Toc52464073)

[4. Automation 25](#_Toc52464074)

[5. Summary of Risk Assessments 26](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 26](#_Toc52464076)

[7. Map the Principles 27](#_Toc52464077)

[Audit Controls and Management 28](#_Toc52464078)

[Enforcement 28](#_Toc52464079)

[Exceptions Process 28](#_Toc52464080)

[Distribution 29](#_Toc52464081)

[Policy Change Control 29](#_Toc52464082)

[Policy Version History 29](#_Toc52464083)

[Appendix A Lookups 29](#_Toc52464084)

[Approved C/C++ Language Acronyms 29](#_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 | Any input data from an untrusted source is a potential security issue. User input can be malformed (by accident) or maliciously designed. Additionally, unvalidated checks on things like command line arguments, APIs and other input not generated within your software can be a potential attack vector. Examples would include the usage of a buffer overflow on a size limited data type (like a character array), or a publicly accessible function call accepting a path through an unvalidated argument. |
| 1. Heed Compiler Warnings | Compiler warnings exist for a reason. Often compiler warnings can flag potential issues and vulnerabilities in code. These vulnerabilities if not adequately corrected can cause unintended or undefined behavior that can leave the software vulnerable. These warnings should not be ignored and should only be suppressed in the case when a developer fully understands the risk associated with the warning but for whatever reason must use the code in this way, otherwise code should be corrected. An example of this would be avoiding type casting when there is an alternative secure method that is feasible to use. |
| 1. Architect and Design for Security Policies | Design software with security policy in mind. Privilege escalation needs should be encapsulated to the need of the escalation and no longer. Systems that need separate privileges should be separated to only have the appropriate privilege sets and function as intercommunicated systems rather than having one system that regularly uses large privilege sets. An example here would be only given database privileges only when needed for a task, then removing those privileges as soon as the task is handled, this should be located in a separate system from the main software whenever possible. |
| 1. Keep It Simple | Design and code should be as simple and concise as is feasible. Well designed, simple, clear, concise code is less likely to cause an error when it can be easily understood during regular code maintenance. As software gets more complex the likelihood that a mistake compromises security or leads to undefined behavior increases dramatically. An example would be writing a number of bitwise/bit shift operations to change an integer instead of using more basic math that clearly relay intent to developer ( x << 1 rather than x \* 2) |
| 1. Default Deny | All permissions and access should be denied by default. Systems should be designed to grant a permission when specific criteria are granted, rather than explicitly removing those privileges when criteria are not met. An example would be granting no database rights by default to a user. When a user meets the criteria for having read rights in the database the user is then given those rights, rather than given the rights be default then having them explicitly removed. I.E, explicitly grant permissions rather than explicitly removing them. |
| 1. Adhere to the Principle of Least Privilege | An extension of principle 3, no process should execute with more rights/permissions than required to complete a task. A task that is only required to check the value of a database entry should not be given read writes, or system specific privileges other than what is immediately required. As soon as a task no longer requires said permission that permission should be revoked until needed again. This reduces the likelihood a vulnerability in one part of the system can lead to code execution with elevated rights, even if those elevated rights are used elsewhere in the code. |
| 1. Sanitize Data Sent to Other Systems | All data passed to subsystems should be sanitized before being passed. This needs to be done in the calling process as unused functionality or bad context can lead to otherwise undesirable results. Passing un-sanitized data can leave vulnerabilities as a subsystem does not understand the context of the call made to it. A system may receive an otherwise valid call where an attacker has exploited un-sanitized data to insert additional command line arguments or invoked unintended code in the context of the subsystem, thereby gaining a vector of attack against the system or called subsystem. |
| 1. Practice Defense in Depth | Defense in depth is using multiple redundant lines of defense (fire walls, IP checks, user privileges, etc.) to reduce the likelihood that a security breach of a specific defensive layer exposes the system. By using an in-depth defense an attacker may have to defeat multiple layers of defense to gain access to the desired system. In this way an exploit used on one level of the defense may only grant the user a small amount of access (limiting the effectiveness of the exploit) or no access at all as the system may require multiple levels of defense being breached before an attack can yield any value to an attacker. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance can help to identify and correct vulnerabilities. The usage of penetration testing, fuzz testing and audits of source code are needed to improve the effectiveness of a quality assurance program. Third party evaluations and security reviews can help to secure systems and bring in unbiased perspectives when looking for vulnerabilities in a system. |
| 1. Adopt a Secure Coding Standard | Design your own or apply a known secure coding standard for all development in your target languages and platforms to ensure that quality coding approaches are taken during the development process. |

### 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 create incompatible declarations of the same function or object. This means that variables (For example) should not be assigned different or inconsistent types or be used interchangeable with other types. |

| **Noncompliant Code** |
| --- |
| In the case of function declarations, the below code example shown below the external prototype of a function is defined to accept and return an int, then in a separate file is defined to accept and return a long. This leads to undefined behavior if the code is compiled in the first place (Visual Studio fails to compile by default with this example and flags 3 warnings) |
| //File A.c  extern int func(int a);  // In B.c  long func(long a) {  return a \* 2;  }  //While a compiler will likely flag this and fail to compile this can //lead to undefined behavior |

| **Compliant Code** |
| --- |
| Below the function prototype and the definition are consistent leading to expected behavior. |
| //File A.c  extern int f(int a);    // In B.c  int f(int a) {  return a \* 2;  } |

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

| **Principles(s):**  2. Heed Compiler Warnings: While not all compilers will recognize this as an issue most modern compilers will and may refuse to compile the code until this is fixed, do not try to circumvent this.  4. Keep it simple: Code should be simple and concise wherever possible, having conflicting type returns can cause truncations and other issues. Its simpler to have a something like a template than to try to override a declaration. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | **Low** | **1** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 22.04 | **type-compatibility**  **type-compatibility-link**  **distinct-extern** | Fully checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL40** | Fully implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.0p0 | **LANG.STRUCT.DECL.IF LANG.STRUCT.DECL.IO** | Inconsistent function declarations Inconsistent object declarations |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **MISRA C 2012 Rule 8.4** | Implemented |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **18, 621, 793, 4376** | Fully supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2022a | [CERT C: Rule DCL40-C](https://www.mathworks.com/help/bugfinder/ref/certcruledcl40c.html) | Checks for declaration mismatch (rule fully covered) |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 8.5.4 | **1 X, 17 D** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.1 | **CERT\_C-DCL40-a** **CERT\_C-DCL40-b** | All declarations of an object or function shall have compatible types If objects or functions are declared more than once their types shall be compatible |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Ensure that integer conversions do not result in lost or misinterpreted data. Explicitly and implicitly converted integers should be guaranteed to not lose information in the process of conversion. |

| **Noncompliant Code** |
| --- |
| The below code attempts to put the max value of unsigned int into a signed short via an explicit cast resulting in truncated bits. |
| int main()  {  unsigned int useful\_int = UINT32\_MAX;  signed short bad\_conv\_short = (signed short)useful\_int;  std::cout<< “Int: “ << useful\_int << “Short: “ << bad\_conv\_short;  };  // This results in a truncated value of -1 for the short while useful\_int is 4294967295 |

| **Compliant Code** |
| --- |
| The below code checks to see if the conversion can be made before attempting to cast integer to short, if possible the conversion is made, if not the error is handled. Additionally signed short is initialized to 0 to remove potential uninitialized error (which could be useful if a value will not be used if a successful operation is completed) |
| int main()  {  unsigned int useful\_int = UINT32\_MAX;  signed short bad\_conv\_short =0;  if (useful\_int <= SHRT\_MAX && useful\_int >= SHRT\_MIN) {  bad\_conv\_short = (signed short)useful\_int;  }  else {  //Error handling code  }    std::cout << "Int: " << useful\_int << "Short : " << bad\_conv\_short;  }  //Results in known value for short (0) and truncated conversion. |

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

| **Principles(s):**  2. Heed Compiler Warnings: Most compilers (with the proper settings) will notify you if there is a potential that you will truncate or lose data during a conversion, if that is the case ensure you are properly checking to eliminate the chance of losing data.  4. Keep it simple: If you explicitly design your code, it simplifies issues and maintenance. Explicitly defining expected ranges before casting a variable makes it clear what the intended value of that variable is. For example, you may want to use a long to ensure that some series of mathematical operations do not overflow, but at the end you expect the value to always be a short, explicitly checking this makes your code simple and clear and reduces unintended consequences.  7. Sanitize Data Sent to Other Systems: This loosely applies due the possibility that a developer may need to pass an integer to another system but the data is explicitly (or implicitly cast) prior to sending, this could result in unexpected results if not handled appropriately so the data should be sanitized before passing, especially when the input is smaller than the output or of a different signed/unsigned value. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | **High** | **5** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity)\* | 2017.07 | **NEGATIVE\_RETURNS**  **REVERSE\_NEGATIVE**  **MISRA\_CAST** | Can find array accesses, loop bounds, and other expressions that may contain dangerous implied integer conversions that would result in unexpected behavior  Can find instances where a negativity check occurs after the negative value has been used for something else  Can find instances where an integer expression is implicitly converted to a narrower integer type, where the signedness of an integer value is implicitly converted, or where the type of a complex expression is implicitly converted  \* Coverity Prevent cannot discover all violations of this rule, so further [verification](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-verification) is necessary. |
| [Cppcheck](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck) | 1.66 | **memsetValueOutOfRange** | The second argument to memset() cannot be represented as unsigned char |
| [PRQA QA-C](https://wiki.sei.cmu.edu/confluence/display/c/PRQA+QA-C) | 9.7 | **2850, 2851, 2852, 2853, 2855, 2856, 2857, 2858,**  **2890, 2891, 2892, 2893, 2895, 2896, 2897, 2898**  **2900, 2901, 2902, 2903, 2905, 2906, 2907, 2908** | Partially implemented |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | **signed\_downcast** | Exhaustively verified. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Do not pass a non-null-terminated character sequence to a library function that expects a string. This can result in accessing out of bounds memory. |

| **Noncompliant Code** |
| --- |
| Specifying the bounds of a character sequence results in a non-null-terminated string leading to undefined behavior when a string is expected. |
| Void func(void){  Char c\_seq[6] = “String”;  Printf(“%s\n”, c\_seq);  } |

| **Compliant Code** |
| --- |
| In this compliant version of the above function the char array does not specify a bound allowing the compiler to allocate storage and include a null character after the string. |
| Void func(void){  Char c\_seq[] = “String”;  Printf(“%s\n”, c\_seq);  } |

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

| **Principles(s):**  1. Validate Input Data: This principle applies due to the fact that improper validation of a string can result in massive security vulnerabilities where data could potential exceed the limit of a character array.  2. Heed Compiler Warnings: Some compilers will warn you about this, some will not, if they do pay attention as these warnings are made to help you.  4. Keep it simple: There is very little reason to use primitive arrays for string input (especially for user input, or input of an undefined size), use Strings to reduce the likelihood of an issue as the code is significantly more straight forward to read and easier to maintain. If you must use primitive arrays (like in an embedded system), ensure that the code explicitly tells other developers its intended use to at minimum mitigate risk.  9. Use Effective Quality Assurance Techniques: This issue can be caught pre deployment with proper quality assurance techniques, if you do not use them code like the Noncompliant Code above could easily slip through the cracks and cause damage to the system or worse. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-STR32** | Partially implemented: can detect some violation of the rule |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.0p0 | **MISC.MEM.NTERM.CSTRING** | Unterminated C String |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **STRING\_NULL** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.1 | **CERT\_C-STR32-a** | Avoid overflow due to reading a not zero terminated string |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Ensure all sql statements are prepared statements where the parameters are filled in later after having been checked. |

| **Noncompliant Code** |
| --- |
| This code example shows an SQL query that is susceptible to SQL injection with an attack like name = “yes or 1=1” |
| void sql\_query(std::string inputName) {  std::string sql\_query = "SELECT \* FROM USERS WHERE NAME = '" +  inputName + "'";  //SQL Query code  } |

| **Compliant Code** |
| --- |
| In this example the inputName variable fills in one parameter and should generally not be susceptible to a 1=1 style attack as the entire input is taken as 1 string. |
| void sql\_query(std::string inputName) {  std::string sql\_query = "SELECT \* FROM USERS WHERE NAME = (?)";  //Ommited code for sql prepared statements  Sqlite3\_bind\_text(  sql\_query,  1,  inputName.c\_str(),  inputName.length(),  SQLITE\_STATIC);  )  //SQL Query Code  } |

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

| **Principles(s):**  1. Validate Input Data: By using prepared sql statements you are more effectively validating the data simply by reducing the likelihood bad data makes it through. While this should not be your only check it is a could final validation for sql execution.  3. Architect And Design for Security Policy: Ensuring that code is built in a way that generally protects from attack (or reduces its ability to inflict damage) falls under this category. Allowing arbitrary code execution (like 1=1) in your database clearly circumvents the intended privilege requirements to access that data.  7. Sanitize Data Sent to Other Systems: An SQL database is a separate system in its own right that generally should run outside of your main application logic. Sanitizing data through numerous means ensures that bad or malicious data is not sent to the system/service and then used to gain access to data that should otherwise not be accessible, prepared statements are not the only way to do this, but they can be very effective as the last check before calling another system.  9. Use Effective Quality Assurance Techniques: With effective techniques this type of issue can be checked and corrected, either by static code analysis or by fuzz testing inputs to ensure something does not get by the parameterized statements. While one mistake may not seem like a huge deal in testing, on a publicly accessible system this will likely result in a data breach or loss. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | **High** | **5** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/java/Coverity) | 7.5 | **SQLI FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_** **FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** | Implemented |
| [Findbugs](https://wiki.sei.cmu.edu/confluence/display/java/Findbugs) | 1.0 | **SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** | Implemented |
| [Fortify](https://wiki.sei.cmu.edu/confluence/display/java/Fortify) | 1.0 | **HTTP\_Response\_Splitting** **SQL\_Injection\_\_Persistence** **SQL\_Injection** | Implemented |
| [SpotBugs](https://wiki.sei.cmu.edu/confluence/display/java/SpotBugs) | 4.6.0 | **SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** **SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING** | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Explicitly construct and destruct objects when manually managing object lifetime. Failing to explicitly call the constructor and destructor can result in undefined behavior when the object is called outside of its lifetime. |

| **Noncompliant Code** |
| --- |
| In this example the constructor of the object S is never called leading to undefined behavior. This is |
| struct S {  S(); //Constructor user defined  void f();  };    void g() {  S \*s = static\_cast<S \*>(std::malloc(sizeof(S)));  s->f(); //Undefined behavior due to constructor    std::free(s);  } |

| **Compliant Code** |
| --- |
| In this compliant code the constructor is explicitly called with the usage of new, the storage location and the object are separated through usage of a pointer allocation. |
| struct S {  S(); //Constructor user defined  void f();  };    void g() {  void \*ptr = std::malloc(sizeof(S));  S \*s = new (ptr) S; //Calling with new calls constructor  //Also separates object from storage location    s->f();    s->~S();//Explicit Destructor call  std::free(ptr);  } |

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

| **Principles(s):**  2. Heed Compiler Warnings: While not every compiler will warn you about this, they generally will or will notify you of an issue during runtime (during testing). These issues need to be taken seriously as this could allow access to bad memory locations and can ultimately be exploited to run malicious code, the best case is a program that crashes regularly, the worst is exposing every user to an exploitable vulnerability.  4. Keep It Simple: By following a strict pattern of memory allocation and deallocation you simplify the overall process of development. This standardized approach may not be the most creative, but it also enhances security as constructors and destructors can be expected and reviewed in every step of testing and verification. It’s easier to identify when something that is always there is missing than if memory is properly deallocated during runtime.  9. Use Effective Quality Assurance Techniques: This issue can be easily checked and audited if followed. Following the Keep It Simple Principle following this standard helps catch an issue before it happens. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.1 | **CERT\_CPP-MEM53-a** | Do not invoke malloc/realloc for objects having constructors |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2022.2 | **C++4761, C++4762, C++4766, C++4767** | No noted description. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.19 | [**V630**](https://pvs-studio.com/en/docs/warnings/v630/)**,**[**V749**](https://pvs-studio.com/en/docs/warnings/v749/) | No noted description. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assertions to test assumptions when writing code to ensure assumptions are/remain accurate. When building new code, it is imperative that the code functionality is well understood, assertions enable developers to ensure code is performing as expected. Static Assertions should be used for compile time checking. |

| **Noncompliant Code** |
| --- |
| In this example an assumption is made about the outcome of the code but is not checked for validity during runtime. This assumption may have been right at the time of development but recently a constant has been changed. |
| Int main(){  const int IMPORTANT\_NUM\_1 = 15; //A constant that is used for  //Several critical program functions, it was recently changed from  // 16  //A bunch of code and logic  int important\_int = IMPORTANT\_NUM\_1 / 4;  //code dependent on important\_int  } |

| **Compliant Code** |
| --- |
| In this assertion code, the assertion is placed just after the assignment of a critical integer, allowing the integer to be checked during runtime. Upon running the code this assertion would fail, the program would abort, and a message about the failure would show to the developer allowing them to correct the code. This would likely stop the change to the const as well as the developer that did so would have caught this unrelated area upon testing. |
| Int main(){  const int IMPORTANT\_NUM\_1 = 15; //A constant that is used for  //Several critical program functions, it was recently changed from  // 16  //A bunch of code and logic  int important\_int = IMPORTANT\_NUM\_1 / 4;  assert(important\_int == 4);  //code dependent on important\_int  } |

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

| **Principles(s):**  1. Validate Input Data: Assertions can be used to validate that data is correctly formatted and of the right value. Assertions can also be used to ensure that Validation techniques (like a technique for checking user input) throws proper exceptions or returns the proper input in any test. While not useful in terms of production code, these validations can be used to ensure that validation is taking place and that the input is validated in the first place.  4. Keep it Simple: If code is dependent on some value, it makes sense to ensure that value is present and correct. While again not very valuable in production, in development the presence of an assertion can signal to another developer (or yourself) that there is a need for a certain result to be true. Additionally, as assertions will likely be in use for early testing, these will warn a developer/tester immediately when a change they have made negatively influences another piece of code. A great example is asserting a constant has a certain value before it is used, that assertation can signal a developer to ensure that changing that constant will not cause additional problems down the line.  9. Use Effective Quality Assurance Techniques: Assertions in critical code makes quality testing and checking much easier to do, during final testing they will be removed but in initial testing these assertions could show a problem given certain states of the system or inputs that were not expected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.0p0 | **(customization)** | Users can implement a custom check that reports uses of the assert() macro |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.DCL03** | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions. When exceptions are not properly handled it can result in a termination of the software with an abnormal state. This behavior can be exploit in denial-of-service attacks. |

| **Noncompliant Code** |
| --- |
| In this example there is no exception handled for a after a function throws an exception so std::terminate() is called potentially not deallocating resources reserved for the software. |
| Void function\_throws\_exception() noexcept(false);  Void function\_caller(){  Function\_throws\_exception();  }  Int main(){  Function\_caller();  } |

| **Compliant Code** |
| --- |
| To make the above code compliant the exception must be caught somewhere at or above where the original exception was thrown. In this case the error is caught with a catch all statement, to ensure stack unwinding. |
| Void function\_throws\_exception() noexcept(false);  Void function\_caller(){  Function\_throws\_exception();  }  Int main(){  Try{  Function\_caller();  }catch(…){  //Error is handled or at least developer can explicitly deallocate  //resources that may still be allocated before crash  }  } |

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

| **Principles(s):**  2.Heed Compiler Warnings: Not handling explicitly thrown exceptions will often show compiler warnings, these issues should be rectified prior to running the code. Proper exception handling can be the difference between a crash and proper termination.  4. Keep it Simple: If you throw and exception or may throw an exception that means there is likely an issue that you have determined could come up. If that is the case keep it simple and handle the exception at the lowest appropriate level. Keeping it simple here just means if something throws and exception (or has the potential to) it should be caught.  9. Use Effective Quality Assurance Techniques: It is pretty simple how this is applied here, if it throws and exception, but does not catch one it is an issue. While some exceptions may be handled at a different level than where the exception is thrown this needs to be thoroughly checked to ensure that the exception is always caught. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 20.10 | **main-function-catch-all early-catch-all** | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.0p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 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](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output/ Memory Protection | [STD-008-CPP] | Close files when they are no longer needed. This properly deallocates resources and reduces the risk of exhausting system resources or the potential of file-buffers not being flushed from memory. |

| **Noncompliant Code** |
| --- |
| In this example, a destructor for file is never called and the basic\_filebuf object is not closed. |
| void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| In this example, the file.close() function is called and a chance to handle any errors with this call is implemented. In this way a developer can be assured the file is removed from memory. |
| void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  file.close();  if (file.fail()) {  // Handle error  }  std::terminate();  } |

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

| **Principles(s):**  2. Heed Compiler Warnings: I have yet to see a compiler that does not warn you about leaving a file open. This warning should always be dealt with as this is a clear and obvious issue, and it is likely very easy to rectify.  3. Architect and Design for Security Policies: In a sense, even having an open file can be considered a privilege, regardless of if the file is open in a read only or write only capacity. Leaving a file open longer than necessary increases the chance that a vulnerability can be used to access this file and take data or write to it.  4. Keep it Simple: Most file operations involve opening, doing some operations, and closing the file. Keep it that simple, there is no reason for a file to be open after its usage has passed. Code should be open, used, then closed immediately following the usage, and anything outside of this direct functionality should be handled outside of this area. There is almost no reason to take user input, or output data to the user while a file is still open, do that before or after.  6. Adhere to the Principle of Least Privilege: Just like above, no file should be accessible outside of its usage. Leaving a file open while taking input, or running other operations just increase the chance for that file to be accessed maliciously. Closing it as soon as feasible is the only solution that makes sense. Think of it this way, a user may only be entitled to one line of a large text file, if that user can access the file that user may be able to get every line of that file, which is clearly them exceeding their intended privileges.  8. Practice Defense in Depth: This loosely falls in this category because an open file can circumvent it. If you were to write to a file that holds privileges information in it then leave that file open a user could potentially exploit this later on in the code because that file is likely still open, with write privileges. You, granting the user the privilege of logging in could easily become the privilege to read from a database, or access other accounts, or review traffic, or literally any other privilege that exists in your application.  9. Use Effective Quality Assurance Techniques: Code that all follows the same pattern is easier to review and audit than code that follows a myriad of different patterns. Open, Use, Close, preferably in a separate function makes this much simpler to check. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.0p0 | **ALLOC.LEAK** | Leak |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 22.04 | No Explicit checker | Supported, but no explicit checker |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **RESOURCE\_LEAK (partial)** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.1 | **CERT\_C-FIO42-a** | Ensure resources are freed |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Protection | [STD-009-CPP] | Do not store an already-owned pointer value in an unrelated smart pointer. This behavior reduces the likelihood of a double-free vulnerability. |

| **Noncompliant Code** |
| --- |
| In the below example a function constructs two unrelated pointers from the same value, these shared pointers then deallocate the same pointer value twice. |
| Void pointer\_function(){  Int \*I = new int; //Creates a pointer object  Std::shared\_ptr<int> smart\_point1(i); //points to I pointer value  Std::shared\_ptr<int> smart\_point2(i); //points to I pointer value as well  }//Both pointers go out of scope and call their destructors |

| **Compliant Code** |
| --- |
| In this example the shared pointers are related to one another leading to a pointer count changing when one is destroyed, but not destroying the pointer until the last reference has been destroyed. |
| Void pointer\_function(){  Std::shared\_ptr<int> smart\_point1 = std::make\_shared<int>();  Std::shared\_ptr<int> smart\_point2(smart\_point1);  }//In this case the shared pointer values decrement to 0 and the object //is deallocated one time |

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

| **Principles(s):**  4. Keep it simple: Use the correct pointer for the job, and make sure not to go outside of its scope. If you need multiple pointer references use a shared pointer (correctly). If not, why are you making multiple copies of a pointer in the first place. While the mistake above is easy to make, keeping it simple is the easiest way to correct it.  9. Use Effective Quality Assurance Techniques: This should absolutely be something that is checked. Whether by a tool or by manual review, using proper techniques to ensure dangling pointers do not exist is an absolute must. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 20.10 | **dangling\_pointer\_use** | No description listed. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-MEM56** | No description listed. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.1 | **CERT\_CPP-MEM56-a** | Do not store an already-owned pointer value in an unrelated smart pointer |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022a | [CERT C++: MEM56-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem56cpp.html) | Checks for use of already-owned pointers (rule fully covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| String Management | [STD-010-CPP] | Range check element access. Do not use C style access operations and instead use the at() function. |

| **Noncompliant Code** |
| --- |
| In this example a string and integer are passed to a function and the function returns the character at the integer index. If the integer is out of range the function will access out of bounds memory and have undefined behavior. |
| Char getCharInString(std::string inputString, int index){  Return inputString[index];  }  //In this case imagine the string is the name John and the index is 5  //It is clear this will access memory outside of the strings characters  //while debug in VS may catch this, in release build this will throw no  //error and could go undetected causing abnormal behavior. |

| **Compliant Code** |
| --- |
| As above we will attempt to access an out of bounds character but after using the at function and exception handling this code will throw an exception and continue without issue (if the exception is handled effectively) |
| Char getCharInString(std::string inputString, int index){  Char returnChar;  Try{  returnChar = inputString.at(index);  }catch(…){  //exception handling (maybe calls self with index-1?)  }  Return returnChar;  }  //In this case if the index is out of bounds an exception will be thrown // and caught and the operation can be handled. |

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

| **Principles(s):**  2. Heed Compiler Warnings: While not usual, a compiler warning can actually show up if a constant is used in to access in a c style way. If you see this it needs to be fixed, but do not depend on the compiler to warn you about this.  4. Keep it simple: Unless you need the direct memory access of C (Like in open GL graphics processing) there is no reason to use them when you can use the at function. The at function throws an exception and allows you to determine what went wrong, you might not even notice an error with the C style accessor, or when you do it will be extremely difficult to track down. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 20.10 | **assert\_failure** | No description listed. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.0p0 | **LANG.MEM.BO** **LANG.MEM.BU** **LANG.MEM.TBA** **LANG.MEM.TO** **LANG.MEM.TU** | Buffer overrun Buffer underrun Tainted buffer access Type overrun Type underrun |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.1 | **CERT\_CPP-STR53-a** | Guarantee that container indices are within the valid range |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022a | [CERT C++: STR53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr53cpp.html) | Checks for:   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered. |

### Defense-in-Depth Illustration

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



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

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

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



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 procedures will become a large part of the DevSecOps enforcement and compliance policy at Green Pace. In that regard, a large portion of the changes will be focused in the “Build” and “Verify and test” portions of the Diagram above. We will be using automated static code analysis to test any saved code and flag the code with the results. The event that triggers this will be a save to the repository (even on separate branches) that will launch a static code analysis using several different analysis tools listed in this document like CppCheck and [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724). This will be built to automatically run and then save those results into a file in the workspace of the project. There will be future work to determine if we can implement these tools directly into the compiler to show up along with normal compiler warnings in each of the majority IDEs (VS, VS Code, Eclipse, InelliJ, etc.). Additionally, dependencies will be checked during this case to ensure vulnerabilities are known and properly mitigated. When code is committed/pushed to a branch that code will be required to be checked by unit tests that will be designed in the design phase and build along side the code (though the two need not be built by the same developer, and in fact may be better to separate the development from the test development). During the Verify and Test phase, these tests will be verified once again and additional automated testing for vulnerabilities will completed such as building the code into the larger application (test version) and Penetration, Input Fuzzing, etc. These tests should also be automated so that all the testers need to do is build the code and the vast majority of these type of tests will run automatically where possible to do so, allowing the testers to do code audits and other work that cannot be fully automated reliably.

These techniques are just the first steps in the Process of Defense in Depth as they are used to reduce cumulative threat levels and mitigate vulnerabilities. In that regard, regular audits of files and other critical warnings will be automated. For example, files will be mapped daily on the services and any changes will be evaluated against a whitelist. In this case, if an attacker managed to put malicious code on the machine in our directories, we will be able to detect those files, narrow down the timeline of when this attack occurred, and reduce the amount of time spent finding and rectifying any vulnerabilities. Additionally, extensive logging should be used to ensure that the likelihood of an attack or successful penetration occurring and going unnoticed will be reduced to all but the most careful of attackers. These logs will need to be backed up (automatically) during each change, and they should be compared to highlight any discrepancies in the two, further narrowing attack vectors. Finally, all dependencies and known vulnerabilities will be tracked and checked regularly (Partially automatically). If there is a known, and unmitigated vulnerabilities (for example awaiting a patch from a third party with no known mitigation at this time) then these issues should be tracked, and the moment a mitigation strategy or patch is available plans should be made and executed for this change if feasible. LDAP and AD will be used wherever it is possible and appropriate to do so.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | Medium | **Low** | **1** |
| STD-002-CPP | High | Probable | High | **High** | **5** |
| STD-003-CPP | High | Probable | Medium | **High** | **4** |
| STD-004-CPP | High | Probable | Medium | **High** | **5** |
| STD-005-CPP | High | Likely | Medium | **High** | **5** |
| STD-006-CPP | Low | Unlikely | High | **Medium** | **2** |
| STD-007-CPP | Low | Probable | Medium | **Low** | **3** |
| STD-008-CPP | Medium | Unlikely | Medium | **Low** | **2** |
| STD-009-CPP | High | Likely | Medium | **High** | **4** |
| STD-010-CPP | High | Unlikely | Medium | **Medium** | **3** |

### 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 in rest | Encryption at rest refers to how data is stored and held on servers or in repositories. Examples of how we will implement this would be file-level encryption, database encryption, full disk encryption and Protection with Digital rights management. This applies to the Defense in Depth principles as this is effectively a last line of defense against Data Breaches. If an attacker managed to get through every level of defense in the system, it is still entirely plausible that the data they recover would be encrypted and would be indecipherable to them. This also can ensure that a user that gains access to some data due to a bug or error rather than malicious intent would not be able to read and decipher any data. An example of this would be a file accidently (due to code or a system error) ending up in a user’s directory (think the cloud or a work system with a combined but separated directory). This user would still be unable to access the file, or unable to read its contents without the appropriate key pairing (which hopefully they would not have access to). |
| Encryption at flight | Encryption at flight refers to data in transit, or rather data being sent from one computer or device to another. There is always a potential that this data could be intercept by another person, especially in the case of public networks. This data could contain sensitive details about the user, a business, security credentials, or a host of other information. As such, all of this data needs to be fully encrypted and preferable sent in a way that is either impossible or unlikely to be decipherable. The way this is done is with private and public keys in what is called a Diffe-Hellman key exchange. Without going into specifics about how this key exchange works, data that is transmitted between systems or from system to user will be encrypted so that any intercepted data will effectively be indecipherable to the person that intercepts them (without the appropriate keys). This prevents an avenue of attack that could be used to intercept key data transmitted and then steal or use that data to retrieve more data. For example, if credentials for a login were sent without this encryption on a public network, the information if intercepted could then be directly used by the interceptor. |
| Encryption in use | Encryption in use is refers to when data is directly in use. This policy is enforced by Identity and roll-based management. Here a user is verified and authenticated (often with two factor authentication) to ensure the accessor of the data is who they say they are. Additionally, user information is checked against known information, the user lives in Cleveland Ohio and uses an IP address that reflects Shanghai China, for the first time, it is likely that this user would be challenged and authenticated through other means, maybe this user would require a VPN to access the data. Finally, a user will only have access to data the user is allowed to have access to. This user would be able to view data, or if they could it would remain encrypted if the user attempts to access it. This means that even if a user has complete control over the file (such as a downloaded copy) this user will only be able to see what is decrypted for them, even if additional data is contained, that data will not be completely or even partially accessible without the appropriate user credentials. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of identifying and verifying a user is who they say they are. This will be handled with user credentials and two factor authentication. Additionally, in the event that user data (say location, browser configuration, etc.) are not the expected values then a user may be challenged to provide more verification such as answering secret questions. If these credentials are verified a user will be granted access, if not the user will not be granted access. |
| Authorization | Authorization is the process of ensuring a user can do a specified task. After authentication a user will have a set of authorizations such as the ability to view certain documents, write to portions of a database, grant, or remove user privileges, and access specific directories. Each time an action is taken (such as opening a file or making a change to user roles) authorization come into play as the system will ensure that the user is allowed to do the task. Without authorization, any user can access anything, but with it, a user will only be able to access files and perform tasks within the scope of their authorization. It is important to note that these authorizations do not come from the user, and rather from the credentialing system, in this way a user can not send an authorization they do not have, they can only request to do something and have their authorization checked. |
| Accounting | Accounting is the practice of logging user statistics and data. After a user is authenticated and authorized to do something, it is important to note and track what the user is doing. It may be usual for a user to download several files, upload one and leave the system, but is it usual for a user to use up 80% of total system resources, or to download thousands of files? Accounting is done not only to keep track of modifications and usage statistic but to monitor and detect potential intrusions or attacks. A user that is operating in an unusual manner may warrant an investigation into activities, from there it may be determined that account credential were stolen or that a user is attempting to do something nefarious. |

**\***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 | 08/07/2022 | Updated Standards and Guidance | Joshua Kovacevich | Joshua Kovacevich |
| [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 |