# CS 405 Project Two Script Template

Complete this template by replacing the bracketed text with the relevant information.

| **Slide Number** | **Narrative** |
| --- | --- |
| **SLIDE 1**  **Title Page** | This is Andrew Emilio Di Stefano and today we’ll be talking about our new security policy designed to help us adhere to a few guidelines which will enhance the security of our systems. |
| **SLIDE 2**  **Overview: Defense in Depth** | Our policy is centered around the concept of Defense in Depth, which refers to a combination of multiple defense strategies of different types and natures to address and mitigate a wide variety of potential attacks (National Institute of Standards and Technology, n. d.). This approach results in a redundant system, or a system which has multiple different ways of protecting against a single threat or vulnerability. |
| **SLIDE 3**  **Threat Matrix** | This threat matrix shows the priority level and the level of likelihood of a security vulnerability resulting from non-adherence to each of our security standards.  Each of the ten principles defined in the ‘Ten Core Security Principles’ section applies directly to one or several of the ten standards defined in this document. Principle 1 (Validate Input Data) relates to Standard STD-001-CPP (Use Appropriate Data Types) in that the use of appropriate data types prevents potentially dangerous input from entering the system (Scheila, 2024). The same principle applies directly to Standard STD-003-CPP (Always Validate Strings to Prevent Buffer Overflows) because the validation of input data is an effective mitigation strategy to prevent successful buffer overflow attack attempts, and to Standard STD-004-CPP since the prevention of successful SQL injection attempts is an important reason to validate input data. Principle 1 also applies directly to Standard STD-009-CPP since validating input data is an important mitigation strategy for avoiding vulnerabilities and cyberattacks involving user input.    Principle 2 (Heed Compiler Warnings) applies directly to Standard STD-002-CPP (Always Initialize Variables) since variables which are not initialized properly within scope are likely to set off compiler warnings, which gives engineers and developers an additional way to assure that this standard is adhered to. The same principle also directly applies to Standard STD-008-CPP (Do Not Invoke Virtual Functions From Constructors or Destructors) since the calling of virtual functions from constructors or destructors is also likely to set off a compiler warning, which gives engineers and developers an additional way to assure that this standard is adhered to.  Principle 3 (Architect and Design for Security Policies) applies directly to Standard STD-001-CPP (Use Appropriate Data Types) since it is important to assure the use of correct data types as early as possible in the production process so that early-phase technical design documents help engineers and developers to adhere to best practices (such as the use of appropriate data types) in later stages. This is the only principle which explicitly describes what should take place at the earliest stages of each iteration of the software development lifecycle (when software is architected and designed). The principle is directly related only to Standard STD-001-CPP of this policy because the drafting of technical design documents (such as UML class diagrams) often defines the data types which will be used for each class.  Principle 4 (Keep it Simple) applies directly to Standard STD-001-CPP (Use Appropriate Data Types) since the use of more intuitive and less convoluted data types is often a better strategy for avoiding vulnerabilities, as is the case in the code examples included within the section of this policy which describes Standard STD-001-CPP.  Principle 5 (Default Deny) applies directly to Standard STD-003-CPP (Always Validate Strings to Prevent Buffer Overflows), STD-004-CPP (Always Validate Input to Prevent SQL Injections), and STD-009-CPP (Prevent User Input Format String Vulnerabilities) because upon receiving input data with potentially malicious quantities, digits, operators, digits, strings, or other relevant characters or character combinations, the user’s attempt to have their input used by the system will be denied by default.  Principle 6 (Adhere to the Principle of Least Privilege) applies directly to Standard STD-004 (Always Validate Input to Prevent SQL Injections) and Principle STD-009-CPP (Prevent User Input Format String Vulnerabilities) since not all users will have the ability to execute SQL commands to create, read, update or delete data from our databases.  Principle 7 (Sanitize Data Sent to Other Systems) applies directly to Standard STD-003-CPP (Always Validate Strings to Prevent Buffer Overflows), Standard STD-004-CPP (Sanitize data sent to other systems), and Standard STD-009-CPP (Prevent User Input Format String Vulnerabilities) because user input is a type of data which could certainly be sent to other systems, and because the sanitization of user data could refer to measures meant to ensure adherence to any of these three standards.  Principle 8 (Practice Defense in Depth) applies directly to all ten of the standards defined in this policy because adherence to all of these standards will adhere to the concept of Defense-in-Depth and contribute to our comprehensive and redundant DevSecOps pipeline.  Principle 9 (Use Effective Quality Assurance Techniques) applies directly to all ten of the standards defined in this policy since the use of effective quality assurance measures will help to enforce this policy by identifying areas where the standards and principles described herein are not adhered to.  Principle 10 (Adopt a Secure Coding Standard) applies directly to all ten of the standards defined in this policy since each coding standard described within this policy works together to assure the security of our systems. |
| **SLIDE 4**  **10 Principles** | Here are the 10 principles of our new security policy.  Principle 1 describes the need for the validation of input data.  Principle 2 describes the need to heed compiler warnings.  Principle 3 describes the importance of designing systems and system components for secure policies.  Principle 4 relates to the idea of making code as simple as possible for ease of interpretation and editing.  Principle 5 involves the idea of default denial, which essentially means that the system should be programmed to deny access if there is any doubt that the system is being used as intended.  Principle 6 describes the importance of adhering to the principle of least privilege, which means that users should only be given the minimum necessary privileges to accomplish their intended goals within the system.  Principle 7 describes the need for the sanitization of data before it is sent to other systems.  Principle 8 reminds us to always maintain security practices which align with defense-in-depth principles.  Principle 9 involves the use of effective quality assurance measures,  and Principle 10 reminds us to constantly consider the security standard as we develop systems for the organization and for our clients and partners.  You may notice some connections between the standards we covered in the previous slide and the principles on this slide. Underneath each principle is a list of standards which directly relate to each principle. |
| **SLIDE 5**  **CODING STANDARDS** | **Standard 4 ( or Code STD-004-CPP)** involves validating user input to avoid SQL injections. This standard has a high priority level because of the wide range of common vulnerabilities associated with ineffective user input. The standard has a medium likelihood level because while most C++ developers and engineers at a professional level will know to validate user input, the wide range of potential validation specifications makes this process comprehensive. We have ranked this as the standard with the highest priority due to the many different ways that a lack of input validation can be taken advantage of by attackers.  **Standard 9 ( or Code STD-009-CPP)** involves preventing vulnerabilities associated with user input being used as the values of format strings. This standard has a high priority level because allowing users to enter unvalidated data into user input fields can leave systems vulnerable to a wide variety of attack types, as is the case with Standard 4. This standard has a high level of priority because it allows users to effectively format a system and/or data store through the use of format strings. This could lead to the successful implementation of severe and damaging system attacks. The standard has a medium level of likelihood because, while it requires adding extra code to assure that user input is not directly used for format strings, the implementation of input validation on user input before it is passed to another variable for use as the value of format strings mitigates against this risk. We have ranked this as a high priority standard, but lower than Standard 4 due to the fact that adhering to Standard 4 would greatly reduce the possibility of a user being able to use malicious input to influence the values of format strings.  **Standard 1 ( or Code STD-001-CPP)** involves selecting the appropriate data type for storing values. This standard has a high priority level due to the many security risks associated with the inappropriate use of data types (notably buffer overflow and SQL injection attacks). For example, if you want to save a username or other non-numerical data entered into user input fields, the use of a string data type would usually be a better option than a char array. The standard has a medium likelihood level since most C++ programmers at the professional level will know how to select the correct data types for their variables in order to avoid the insecure writing of code. We have ranked this standard at a higher priority level than the following standards due to the wide variety of potential vulnerabilities which could result from non-adherence.  **Standard 3 ( or Code STD-003-CPP)** involves ensuring string correctness to prevent buffer overflows. This standard has a high priority level since allowing users to enter unvalidated data into user input fields can leave systems vulnerable to buffer overflow attacks. This standard has a medium level of likelihood because, while adding input validation does add another level of complexity to a program, the implementation of input validation effectively mitigates against this risk quite effectively.  **Standard 6 ( or Code STD-006-CPP)** involves the use of assertions to diagnose errors. This standard has a medium priority level because non-conformity does not directly lead to vulnerabilities, but conformity makes it much easier to identify potential security issues. The standard has a medium level of likelihood because review by several different individuals may be necessary in order to assure enough appropriate use of assertions in C++ code.  **Standard 7 ( or Code STD-007-CPP)** involves the use of ‘try,’ ‘catch,’ and ‘throw’ statements to catch exceptions. The standard has a medium likelihood level and a medium priority level for the same reasons that Standard 6 is in the same position within the matrix.  **Standard 8 ( or Code STD-008-CPP)** involves refraining from invoking Virtual Functions From Constructors or Destructors. This standard has a medium priority level and a medium likelihood level because, while non-adherence would be difficult for malicious actors to take advantage of directly, non-adherence could cause the system to engage in unexpected and undesired behavior.  **Standard 2 ( or Code STD-002-CPP)** involves the initialization of variables used to store data values. This standard has a medium priority level since while this error is usually caught within a code editor or IDE, it is somewhat easy for developers to forget to initialize variables before they are used. The standard has a low likelihood level since, as is the case with Standard 1, C++ programmers at the professional level are unlikely to make this mistake enough for it to become an issue.  **Standard 5 ( or Code STD-005-CPP)** involves refraining from accessing dangling pointers. Accessing a dangling pointer (or a pointer which has been deallocated by a memory management function) could result in undefined behavior (Britton & Pincar, 2023). This standard has a medium priority level because it would be hard for an attacker to exploit purposefully, but could result in the divulging of sensitive data involuntarily. The standard has a low level of likelihood because the issue is relatively easy to notice and a wide range of mainstream static testing tools check for it.  **Standard 10 ( or Code STD-010-CPP)** involves making sure not to access objects outside of their lifetime. This standard has a medium level of priority because nonconformity could lead to unexpected and undesired behavior. The standard has a low level of likelihood since developers who know when and how to set lifetimes for objects are unlikely to then have the object called outside of its lifetime (though this is of course a possibility nonetheless). |
| **SLIDE 6**  **ENCRYPTION POLICIES** | Encrypting data while it is at rest within a system adds a layer of security by making stored data uninterpretable without a decryption key (IBM, 2025). Our policy for encryption at rest will involve implementing encryption when it is generated or input (Raza, 2023), and storing the encrypted data so that it will only be decrypted when it needs to be accessed. This will make it much harder for our data to be compromised even if an attacker is able to gain some level of access to the system.  Encrypting data while it is in transit helps to ensure secure communication between system components or with external systems and devices. Our policy for encryption in flight will involve "encrypting data that is transferred between two nodes of the network” (Raza, 2023) or that is sent to or received from external systems or devices. The policy will involve the use of SSL(TLS) certificates, and the use of HTTPS (secure hypertext transfer protocol) for communication across the internet (Google Cloud, n. d.).  The encryption of data while it is in memory during runtime helps to further reduce the possibility of data being compromised. Our encryption in use policy will consist of hardware-based encryption and partial or full homomorphic encryption, “which allows data to be processed while it’s still encrypted” (SentinelOne, 2024). Since full homomorphic encryption is more computationally intensive (Velimirovic, 2023), partial homomorphic encryption may be used for less sensitive data in use while full homomorphic encryption is be used for more sensitive data while it is in use by a system component. |
| **SLIDE 7**  **Triple-A Policies** | Triple a is “a security framework that controls access to computer resources, enforces policies, and audits usage…its combined processes play a major role in…cybersecurity by screening users and keeping in track of their activity while they are connected” (Fortinet, 2025). The three As in Triple-A stand for authentication, authorization, and accounting.  Authentication involves prompting system users for information which proves that they are who they say that they are (Fortinet, 2025). User login credentials, for example, tell the system that a specific user is logging in. The system trusts that the person logging in is authorized to use that account to access the system due to the fact that the user has entered the correct password which, ideally, only said user has. Our authentication policy will involve user authentication via the usual method of giving every user a username and password (the user is able to change their password later if they choose), and using multi-factor authentication (MFA) as an additional layer of security. MFA will also be used for access to any third-party services used by our organization such as cloud providers for example. Additionally, user authentication consisting of password re-entry should take place when a user attempts to access or make changes to our databases. The addition of new users should also involve authentication via one of the new user’s personal devices such as a smartphone in order to prevent the addition of users through unauthorized system manipulation. The strength of our authentication policy for user logins will help to prevent unauthorized access to our systems and will allow us to implement access control in accordance with Principle 6 described above (‘Adhere to the Principle of Least Privilege’).  “Authorization follows authentication…a user can be granted privileges to access certain areas of a network or system…authorization dictates what a user is allowed to do” (Fortinet, 2025). After user authentication takes place, the implementation of our authorization policy will prevent users from accessing data and methods which are irrelevant to their function within the system. Only administrative users (admin) will be granted access to the entire system. Other types of users, whether institutional or external, will only be able to access parts of the system which they need to access in order to achieve their goals within the system. Each user should also only be able to access their own data, and there should be no way for users to access data related to other users (unless said data is required for them to achieve their goals within the system). The strength of our Authorization policy will prevent unauthorized access to data and functions which could result in compromised data or even damage to system components.  “Accounting keeps track of user activity…such as how long they were logged in, the data they sent or received, their…(IP) address, the…(URI) they used, and the different services they accessed” (Fortinet, 2025). Effective accounting enhances the ability for administrators to prevent unauthorized access, and can be used to block users suspected of repetitively attacking the system purposefully. Input which contains data indicating a potential attack should be documented, and the system should be automated to respond accordingly. For example, a system or system component must check for and document certain combinations of the ‘-,’ ‘(,’ ‘>,’ or ‘=’ characters within an input string, since certain arrangements or uses of these characters could indicate that the user is likely attempting a SQL injection attack. The repeated entry of numerical data types (int, float, etc) which are far too large for their use-case must also be documented. User attempts to repeatedly access files with sensitive data will also be accounted for, and the action taken in this scenario will depend on the type of user, on the type of data being accessed, and on the specific situation in general. Effective accounting to track user interaction with the system mitigates against certain types of attacks by blocking suspected attackers from continuing to interact with the system. |
| **SLIDES 8, 9, 10, 11,**  **12, 13, and 14**  **Unit Testing / Unit Tests** | **SLIDE 8**  “**Unit Testing** is a software testing technique in which individual units or components of a software application are tested in isolation” (GeeksForGeeks, 2024). By comprehensively unit testing small components of our system, we can assure a constant adherence to our security policy which will help us to increase the security of our systems. This is especially important when writing in C++ for optimized memory management.  **SLIDE 9**  **TEST 1:** This positive unit test checks for adherence to Standard 1: Use Appropriate Data Types (STD-001-CPP) by declaring a variable called ‘number’ as a string data type when the name of the variable implies that it was meant to be used as an int data type. We use a boolean variable called ‘is\_not\_an\_int’ which is set to ‘true’ if the ‘number’ variable is not of int data type. We then assert that ‘is\_not\_an\_int’ has been set to true. In practice, the value of this boolean variable could be used to determine whether or not user input is allowed to be set as the value of a variable.  **SLIDE 10**  **TEST 2:**  This positive unit test checks for adherence to Standard 4: Always Validate Input to Prevent SQL Injections (STD-004-CPP) by declaring a variable called ‘user\_input’ which contains a SQL injection attempt. We use a boolean variable called ‘possible\_injection\_attempt’ which is set to ‘true’ if the variable contains characters or character combinations which indicate a possible SQL injection attack. We then assert that ‘possible\_injection\_attempt’ has been set to true given the suspicious value of the ‘user\_input’ string.  **SLIDE 11**  **TEST 3:** This positive unit test checks for adherence to Standard 3: Always Validate Strings to Prevent Buffer Overflows (STD-003-CPP) by declaring a variable called ‘user\_input\_string’ which holds a string with 21 characters. We use a boolean variable called ‘too\_big’ which is set to ‘true’ if the ‘user\_input\_string’ variable is longer than 20 characters. We then assert that, given the size of the string value of the ‘input\_string’ variable, the value of the boolean ‘too\_big’ variable is set to ‘true’.  **SLIDE 12**  **TEST 4:** This negative unit test checks for adherence to Standard 3: Always Validate Strings to Prevent Buffer Overflows (STD-003-CPP) by declaring a variable called ‘user\_input\_string’ which holds a string with 21 characters.   We use a boolean variable called ‘correct\_size’ which is initialized with a value of ‘true’. We then assert that, given the size of the string value of the ‘input\_string’ variable, the value of the boolean ‘correct\_size’ variable is set to ‘false’.  You may notice the similarities between this test and our VerifyStringValidationPreventsOverflowPositive test. The addition of a parallel negative test covers this test case from an additional angle makes our testing efforts more comprehensive for this test case.  **SLIDE 13**  **TEST 5:** This negative unit test checks for adherence to Standard 1: Use Appropriate Data Types (STD-001-CPP) by declaring a variable called ‘number’ as a string data type when the name of the variable implies that it was meant to be used as an int data type. We use a boolean variable called ‘is\_an\_int’ which is set to ‘false’ if the ‘number’ variable is not of int data type. We then assert that ‘is\_an\_int’ has been set to false.  **SLIDE 14**  **TEST 6:**  This negative unit test checks for adherence to Standard 4: Always Validate Input to Prevent SQL Injections (STD-004-CPP) by declaring a variable called ‘user\_input’ which contains an obvious SQL injection attempt.   We use a boolean variable called ‘safe\_input’ which is set to ‘false’ if the variable contains characters or character combinations which indicate a possible SQL injection attack. We initialize this variable with the value ‘true’.   We then assert that ‘safe\_input’ has been set to false given the suspicious substrings found in the value of the ‘user\_input’ string. |
| **SLIDE 15**  **Automation Summary** | In order to automate the enforcement of compliance to the standards and principles described in this presentation, the static testing tools which will be described in the next section must be integrated into the organization’s existing DevOps processes and infrastructure. Static code testing will be a major aspect of our automation and DevSecOps infrastructure, and the use of each of these static testing tools will ensure the enforcement of each of the standards and principles described above among our organization’s teams. Quality Assurance professionals will be responsible for periodically using these tools to review and test code as it is written. The frequency of the use of these tools on developed code will depend on many situation-specific variables.  These static testing tools are used during the ‘Build’ and the ‘Verify and Test’ stages depicted on screen. While the second of these phases is focused exclusively on verifying and testing code, static testing tools should be used throughout the development (Build) process to ensure exhaustive and redundant testing measures in adherence to Defense-in-Depth principles (National Institute of Standards and Technology, n. d.). Comprehensive and redundant testing will help to ensure that the code which makes up our systems is secure and bug-free, and that sufficient measures have been taken to enforce adherence to this security policy.  The tools and techniques described in the following slides will be integrated into the organization’s existing processes. Static code testing will be a major aspect of our DevSecOps infrastructure, and the use of each of these static testing tools will ensure the enforcement of each of the standards and principles described above among our organization’s teams. Quality Assurance professionals will be responsible for periodically using these tools to review and test code as it is written. |
| **SLIDE 16 and 17**  **Tools** | **SLIDE 16**  The tools which will be used for automation throughout the DevSecOps process are Astrée, Axivion Bauhaus Suite, CodeSonar, Coverity, CppCheck, and Parasoft.  “Astrée is not only a reliable debugging aid, it is a rigorous formal veriﬁer which can be included in a stringent certiﬁcation process…Astrée can contribute to the production of high-quality software in a cost-effective, timely, and  reproducible manner” (Cousot et al., 2007).  Axivion Bauhaus Suite is a highly robust static testing solution which is made to handle large software projects with complex system architectures (QT Group, 2025).  CodeSonar is known for focusing on critical defects such as memory leaks, system crashes, and security vulnerabilities (GrammaTech, n. d.).  **SLIDE 17**  Coverity has a highly comprehensive variety of static code analysis capabilities and is known for its low false-positive rate (PeerSpot, 2025).  “Cppcheck is a popular, open-source, free, cross-platform static code analysis tool dedicated to C and C++. It is known for being easy to use and its simplicity” (Sibony, 2021).  Parasoft is known for its comprehensive testing tool known for its integration of static testing with unit testing, security testing, and advanced analytics (Parasoft, 2025). |
| **SLIDE 18**  **Risks and Benefits** | It is critical that we all remain in sync with principles and best practices. The solutions discussed in this presentation and in our new Security Policy will not only help to ensure the quality of our code, but will also condition our teams to rapidly adjust to the levels of company growth which we expect to see in the near future.  If we act now, we can assure that company growth does not cause us to fall behind in terms of assuring the quality of our systems. The costs associated with making these changes is much less than the costs which could be incurred by not taking these issues seriously.  The more we wait, the further our current security and quality assurance measures will lag behind the trajectory of our organization’s growth. There is no benefit to putting these changes on hold and allowing our organization to become a producer of sub-standard systems and system components. |
| **SLIDES 19 and 20**  **Recommendations** | The issues which need to be addressed first are STD-004-CPP (Always Validate Input to Prevent SQL Injections) and STD-009-CPP (Prevent User Input Format String Vulnerabilities). The validation of input to prevent SQL injection attacks should be addressed immediately due to the level of control which attackers could gain over our SQL databases and the data stored within. The Foxton Group’s 2020 data breach, which resulted in “over 16,000 customer records, including sensitive financial data” (Radware, 2025) serves as a sobering example of what can happen when user input validation methods are insufficient.  Adherence to Standard 9 (Prevent User Input Format String Vulnerabilities) should also be addressed immediately due to the level of control which attackers could gain over entire systems through the passing of malicious user input into format strings. Since Tymm Twillman’s identification of this vulnerability in one of his clients’ systems back in 1999, the addressing of this potentially devastating vulnerability has been a top priority for organizations (Beschokov, n. d.). This vulnerability could lead to “unexpected crashing of code…unauthorized access to stack data…execution of an arbitrary code for an application…[and]...successful Denial of Service (DoS) (Beschokov, n. d.).  Other issues which need to be addressed immediately are the implementation of static testing and unit testing. The use of a unit testing framework such as GTest to test each component of our code as it is developed will help us to make sure that our C++ projects are made up of sustainable and manageable code, and to get rid of “stupid bugs” (NDC Conferences, 2018) before they become bigger issues later along in development process.  Throughout this presentation we have explored the power of static testing in the development of secure and high-quality systems. The implementation of a robust combination of static testing tools will help us to ensure the quality of our code and to enforce the standards and principles of this Security Policy. For these reasons, we believe that the implementation of static testing is one of the issues which we should address first.  Since Axivion Bauhaus Suite, Coverity, and CppCheck cover the majority of the standards described in this document with significant redundancy (these three tools check for a wide variety of common issues related to each standard with significant overlap), developers and engineers should be trained to use these tools so that they can perform static testing on their own code during the ‘Build’ phase of development process. These same three tools will also be used by testers and quality assurance professionals during the ‘Verify and Test’ phase, along with CodeSonar, Parasoft and Astrée for further redundancy and for the detection of specific cases less comprehensively covered by the first three tools. |
| **SLIDES 21 and 22**  **Conclusions** | **What current gaps in the security policy still need to be addressed?**  While this security policy contains a comprehensive series of important security principles and ten very important security standards, the addition of more principles and standards as our organization continues to grow will even further improve our ability to write secure code, ensuring that our systems minimize the possibility of successful attacks as much as possible. As we add new standards to our security policy, the addition of more static testing tools will likely become necessary. Additional static testing tools would enhance our ability to write system-specific tests (Kienle et al., 2012) which meet the current needs of our organization, and the addition of other static testing tools as any gaps in our security policy reveal themselves.  Another potential gap in our security policy is a lack of threat intelligence and threat response automation. The use of threat response automation and threat intelligence opens up a wide variety of different security capabilities such as the ability to set up automatic alerts when system penetration attempts are suspected, (Chuvakin, 2021), and to set up response playbooks so that the system can automatically defend itself or even attempt to gain information which could lead to the apprehension of the attackers by law enforcement (Winston, 2021). These measures relate to Accounting, the third A of the Triple-A framework, since both types of tools rely on the comprehensive accounting of data from known and unknown users and their interactions with the system.  **What standards should be adopted to prevent future problems?**  Principle 3 (Architect and Design for Security Policies) highlights the importance of taking secure coding practices into consideration during early stages of the development process (or of an iteration of the process). Although code is written during the ‘Build’ and the ‘Verify and Test’ phases of development, decisions which are made during the ‘Assess and Plan’ and the ‘Design’ stages of iterative development must take secure coding practices into consideration. For example, design documents such as UML class diagrams should adhere to best practices related to data type selection in adherence to standard STD-001-CPP (Use Appropriate Data Types).  Additional standards which could be added in the future include making sure that files are closed when they are not being used, such as the use of ‘std::ifstream::close’ after performing operations on a file with ifstream (GeeksForGeeks, 2023) in order to protect the contents of the file. Assuring that all iterator ranges are valid is another standard which could be added to our policy in the interest of avoiding (Britton & Long, 2023). |

**Sources:**

Beschokov, M. (n. d.). *Format String Vulnerability.* Wallarm.

<https://www.wallarm.com/what/format-string-vulnerability>

Britton, J., & Dewhurst, S. C. (2023, April 20). *OOP50-CPP.* Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046570>

Britton, J., & Mohindra, D. (2024, June 11). *IDS06-J*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487836>

Britton, J., & Long, F. (2023, April 20). [CTR53-CPP. Use valid iterator ranges](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges).

Carnegie Mellon University.

[https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges)

[valid+iterator+ranges](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges)

Britton, J., & Pincar, J. (2023, April 20). *Do Not Access Freed Memory.* Carnegie

Mellon University.

[https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory)

[access+freed+memory](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory)

Britton, J., & Seacord, R. C. (2018,December 11). *DCL03-C*. Carnegie Mellon

University. [https://wiki.sei.cmu.edu/confluence/pages/viewpage.action](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152090)

[?pageId=87152090](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152090)

Chuvakin, A. (2021, March 19). *2021 Threat Intelligence Use Cases*. Anton on Security.

[https://medium.com/anton-on-security/2021-threat-intelligence-use-cases-](https://medium.com/anton-on-security/2021-threat-intelligence-use-cases-8f4423e250c5)

[8f4423e250c5](https://medium.com/anton-on-security/2021-threat-intelligence-use-cases-8f4423e250c5)

Cousot, P., Cousot, R., Ferret, J., Mauborgne, L., Monniaux, D., Mine, A., & Rival, X.

(2007). Varieties of Static Analyzers: A Comparison with ASTREE.

*Conference: Theoretical Aspects of Software Engineering, 2007*.

<http://dx.doi.org/10.1109/TASE.2007.55>

Fortinet. (2025). *IEC-62443 Standard.*

<https://www.fortinet.com/resources/cyberglossary/iec-62443>

Fortinet. (2025). *What Is Authentication, Authorization, And Accounting*

*(AAA) Security?*

<https://www.fortinet.com/resources/cyberglossary/aaa-security>

GeeksForGeeks. (2023, November 2). *std::fstream::close() in C++.*

<https://www.geeksforgeeks.org/std-fstream-close-in-cpp/>

Google Cloud. (n. d.). *Confidential Computing*.

<https://cloud.google.com/security/products/confidential-computing?hl=en>

Google Cloud. (n. d.). *Encryption in Transit.*

<https://cloud.google.com/docs/security/encryption-in-transit>

GrammaTech. (n. d.). *Codesonar*.

<https://www.verifysoft.com/GrammaTechCodeSonarDatasheet.pdf>

Kienle, H. M., Kraft, J., & Nolte, T. (2012). System-specific static code analyses:

a case study in the complex embedded systems domain.

*Software Quality Journal 20*, 337–367.

<https://doi.org/10.1007/s11219-011-9138-7>

National Institute of Standards and Technology. (n. d.). *Defense-in-Depth.*

<https://csrc.nist.gov/glossary/term/defense_in_depth>

NDC Conferences. (2018, October 2). *C++ Unit testing - the*

*good, the bad & the ugly - Dror Helper* [Video]. YouTube.

<https://www.youtube.com/watch?v=gCQDBz-TMIE>

Parasoft. (2025). *Static Code Analysis Solutions & Tools for Compliance.*

[https://www.parasoft.com/solutions/static-code-analysis](https://www.parasoft.com/solutions/static-code-analysis/#:~:text=with%20security%20standards.-,How%20It%20Works,recommendations%20that%20accelerate%20remediation%20steps)

PeerSpot. (2025). *What is Coverity?*

<https://www.peerspot.com/products/coverity-reviews>

QT Group. (2025). *Axivion Static Code Analysis*.

[https://www.qt.io/product/quality-assurance/axivion-static-code-analysis](https://www.qt.io/product/quality-assurance/axivion-static-code-analysis#:~:text=Axivion-,Static%20Code%20Analysis,code%20%2D%20right%20from%20the%20start)

Radware. (2025). SQL Injection: Examples, Real Life Attacks & 9 Defensive Measures

<https://www.radware.com/cyberpedia/application-security/sql-injection/>

Raza, M. (2023, January 31). *Encryption Explained: At Rest, In Transit &*

*End-To-End Encryption.* Splunk.

<https://www.splunk.com/en_us/blog/learn/end-to-end-encryption.html>

Sentinel One. (2024, August 27). *What is Encryption? Types, Use Cases, and Benefits*.

<https://www.sentinelone.com/cybersecurity-101/cybersecurity/what-is-encryption/>

Sibony, J, (2021, June 30). *Top 9 C++ Static Code Analysis Tools.* IncredibleBuild.

<https://www.incredibuild.com/blog/top-9-c-static-code-analysis-tools>

TutorialsPoint. (2025). *Difference Between LPAD and Active Directory.*

<https://www.tutorialspoint.com/difference-between-ldap-and-active-directory>

Velimirovic, A. (2023 November 16). *Data Encryption In Use Explained.* PhoenixNap.

<https://phoenixnap.com/blog/encryption-in-use>

Winston, T. (2021, September 9). [*Cyber Threat Intelligence in ICS Sectors:*](https://gca.isa.org/blog/cyber-threat-intelligence-in-ics-sectors-context-is-everything)

[*Context is Everything*](https://gca.isa.org/blog/cyber-threat-intelligence-in-ics-sectors-context-is-everything)*.* Global Cybersecurity Alliance.

[https://gca.isa.org/blog/cyber-threat-intelligence-in-ics-sectors-context-](https://gca.isa.org/blog/cyber-threat-intelligence-in-ics-sectors-context-is-everything)

[is-everything](https://gca.isa.org/blog/cyber-threat-intelligence-in-ics-sectors-context-is-everything)