**CS2S562**

**Secure Software Development**

**Validation Report Guide**

Your name **Hossam Farfour**

Enrolment number **30008722**

Date **25/03/2021**

Module code (CS2S562)

Coursework title (Assess security principles in a software application)

# Table of Content

When you compare the structure below to the marking grid you will find that it lists all the aspects expected in the coursework.

1 Report on Program 1.zip

1.1 Code Validation

1.1.1 Integer vulnerabilities

1.1.2 String vulnerabilities

1.1.3 Memory vulnerabilities

1.1.4 Formatted input/output vulnerabilities

1.1.5 File I/O vulnerabilities

1.1.6 Pointer vulnerabilities

1.1.7 Automated quality tool usage

1.2 Pattern & Principle Validation

1.2.1 Authenticator pattern

1.2.2 Logger pattern

1.3 Summary and Overall Verdict

2 Report on Program 2.zip

(same structure as for Program1.zip here)

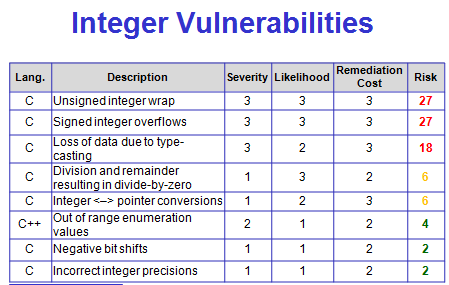
3 Report on Program 3.zip

(same structure as for Program1.zip here)

# Content of the Various Sections

## Code Validation

In the lecture slides you will usually find a table with the top-risk vulnerabilities for each of the topics we covered. For example, in the case of integer vulnerabilities the 3 top risks are:



For each of these 3 top risks the lecture slides and/or the CERT website will specify a number of checks that can be made to minimise or avoid these vulnerabilities. **Run the specified in the template below checks** for each risk as shown in the **example** table and underneath provide an overall verdict as shown:

**Table 1.1: Integer Vulnerabilities**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks Made** | **Result** |
| Unsigned integer wrap | Check 1: When vulnerable operators (+,-,\*,++,--, +=, -=, \*=, <<) are used wrap checks are made or wrapping otherwise avoided | Arithmetic operations were all made using dedicated methods which all performed pre-operator wrap checks. << was not used. |
| Check 2: Use of C99 size constants such as UINT16\_MAX for wrap checking | Not used at all! |
| Check 3: Use of C99 defined length types such as the ones defined in <inttypes.h>, e.g. uint16\_t instead of just 'int' | Occasionally used (ca. 50% of all unsigned ints) |
| Signed integer overflow | Check 1: When vulnerable operators (+,-,\*,/,%,++,- -, +=, -=, \*=, <<, .....) are used wrap checks are made or wrapping otherwise avoided | Arithmetic operations were all made using dedicated methods which all performed pre-operator wrap checks. << and >> were not used. |
|  | Check 2: Use of C99 size constants such as INT16\_MAX for wrap checking | Not used at all |
| Loss of data due to type-casting | Check 1: ..... | Type casting to an int or unsigned int not used / required in the program, therefore no problem. |

**Overall integer security performance verdict** (0=poor, 5=excellent): **3**

### Secure Design Principles and Pattern Validation

**An example** for the validation of the Authorization Pattern is shown below:

**Table 2.1: Secure Design Principles and Pattern Validation**

|  |  |  |
| --- | --- | --- |
| **Design Principle or Pattern** | **Checks Made** | **Result** |
| Authorization Pattern | Check 1: Resource independence | Yes. This generic approach has been used which makes the solution independent of the type of resource. |
| Check 2: Different types of users | Partly fulfilled. Only one type of user has been implemented. The subjects could be of different types including users, administrators or even processes. |
| Check 3: Authorization rules | Yes. The authorization rules can easily be added, modified or removed. This allows flexible management of the authorization rules. |

**Overall performance assessment of code** (0=poor, 5=excellent): **3**

## Summary and Overall Verdict

Here you summarise your findings in a table and provide a final "mark", as before out of the range between 0 and 5:

|  |  |  |
| --- | --- | --- |
| **Check Type** | **Comment** | **Verdict** |
| Integer | Good check on arithmetic operations but no use of C99 types | 3 |
| String |  |  |
| Memory |  |  |
| Formatted I/O |  |  |
| File I/O |  |  |
| Pointers |  |  |
| Tool Usage |  |  |
| Authenticator Pattern | Good resource independence and authorization rules but some weaknesses in the types of users (only one type possible) | 3 |
| Authorisation Pattern |  |  |
| **OVERALL** | **A reasonably secure program** | **3.5** |

NOTE: More information about the specific secure design principles and patterns and their assessment criteria can be found in the lecture notes.

1. **REPORT ON PROGRAM 1.ZIP**
   1. **CODE VALIDATION**
      1. **INTEGER VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Unsigned integer wrap | Check if wrap checks are made when vulnerable operators are used (+, -,\*,++, etc.) | **Arithmetic operations were all made using dedicated methods which all performed pre-operator wrap checks. << was not used.** |
| Search for C99 size constants such as ‘UINT16\_MAX’ for wrap checking. | **Occasionally used** |
| Check for C99 data types such as uint16\_t instead of just int. | **Occasionally used** |
| Signed integer overflow | Check if wrap checks are made when vulnerable operators are used (+, -,\*,++, etc.) | **Arithmetic operations were all made using dedicated methods which all performed pre-operator wrap checks. << was not used.** |
| Search for C99 size constants such as INT16\_MAX for wrap check. | **Occasionally used** |
| Search for C99 data types such ‘int16\_t’ instead of just int. | **Occasionally used** |
| Loss of data due to type-casting | Check for any instances of type casting. (e.g. (int)double) | **Arithmetic operations were all made using dedicated methods which all performed pre-operator wrap checks. << was not used.** |

**Overall integer security performance verdict** (0=poor, 5=excellent): **3**

* + 1. **STRING VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Storage for string must have enough space for data and null terminator | Checks for use of std::string wherever possible (e.g. instead of char\*) | **Not Declaration for a string in memory** |
| When creating a char array or allocating space for a string, ensure that there is space for the null terminator. | Not used |
| Do not attempt to create a std::string from a null pointer | Check that pointers are not null before attempting to create the std::string. (e.g tmpPtr= std::getenv("TMP");  if( tmpPtr == NULL ) ) | Not used |
| Use valid references, pointers, and iterators to reference elements of a string | Check that an index/iterator is valid for the string before using it to access elements of a string. | Not used |
| Check for valid references to reference elements of a string. | **Occasionally used** |

**Overall string security performance verdict** (0=poor, 5=excellent): **1**

* + 1. **MEMORY VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Do not access freed memory | Check if memory has been accessed after it was deallocated | **Not been accessed after deallcated** |
| Only free memory that was allocated dynamically. | Check if an appropriate deallocator was used for dynamically allocated memory  (e.g. delete() [], free etc) | **There is a method called free(input); but Deallocator is not used at all** |
| Check if the destructors of objects are used correctly to prevent memory leaks | **There is a memory leak** |
| Check if smart pointers are used correctly to provide automatic memory management | **Smart pointer is not used** |
| Allocate sufficient memory | Check if sufficient amount of memory has been allocated for a given task (e.g. in arrays, vectors etc) | **Is used** |
| Check if memory allocation errors are handled correctly (e.g. using std::bad\_alloc exception or checking nulllptr) | **Memory allocation error is not checked** |

**Overall memory security performance verdict** (0=poor, 5=excellent): **1**

* + 1. **FORMATTED INPUT/OUTPUT VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Formatted I/O | Check if cin & cout were used instead of scan & print | **is used** |
| Check if parametric manipulator functions were used correctly (e.g. setprecision etc) | **no parametric of flag functions set** |
| Unformatted I/O | Is ‘user input’ from the keyboard canonicalised, normalised, sanitised, if so how? | Not sued |
| Check if the data fall within the expected domain of valid input (e.g. uint16\_t overflow) | the overflow is checked |
| Has the programmer used conversion to integers or float from a string? if so has this been done securely? | **Yes, been sued** |

**Overall formatted security performance verdict** (0=poor, 5=excellent): **3**

* + 1. **FILE I/O VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Avoid TOCTOU race conditions while accessing files | Check for race conditions, where there are two or more file operations that depend on the same resource. Has this been handled seccurly? | **There is no code to prevent an existing file from being overwritten by first opening it for reading before opening it for writing.** |
| Distinguish between characters read from a file and EOF | Check to see if user has made use of an end of file check, so that characters will only be read from a file until the end of the file is reached. | Not checked |
| Do not assume that fgets() returns a nonempty string when successful | Check that the return of a call to fgets() is not an empty string, or use functions such as strchr() that handle empty strings well. | **Not function to handle empty string** |
| Do not alternately input and output from a stream without an intervening flush or positioning call | Check for undefined behaviour if no intervening flush or positioning call between the calls to fread() and fwrite() | Is Used |
| Securely handle file opening, read, write operations | Check for secure handling of error if the file fails to open, or read and write operation fails. | Is checked |
| Close files when they are no longer needed | Check if user has closed all files that are opened. | **Yes is used** |

**Overall file security performance verdict** (0=poor, 5=excellent): **3**

* + 1. **POINTER VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks Made** | **Result** |
| nullptr pointer | Check to see if all variables containing pointers are referenced correctly. E.g. uninitialized pointers are set to null, Pointers were mostly checked if they were null before being used etc. | **Yes, been used correctly** |
| Confusing pointer and value | Check to see the user hasn’t confused the pointer with the value the pointer points too. E.g. using \*x or x instead of &x and vice versa. | **Been checked (no confusion)** |
| Pointer miscalculation | Check the user hasn’t misused the pointers so that a calculation is wrong, or memory address is returned instead of the correct value. | **Is used correctly** |
| Smart pointers | Check if smart pointers are used correctly and in preference to “naked” pointers | **No smart pointer** |

**Overall pointer security performance verdict** (0=poor, 5=excellent): **4**

**1.1.7 AUTOMATED QUALITY TOOL USAGE**

|  |  |  |
| --- | --- | --- |
| **Tools** | **Checks** | **Result** |
| Compiler and linker flags | Check use of secure compiler flags ( e.g. /RTC , warning levels etc) | **There are more errors when the warning level set to 3** |
| Check use of secure linker flags /DYNAMICBASE, /NXCOMPAT ect) | **Not used** |
| Code analysis | Check for Evidence of static code analysis use. | **Not checked at all** |
| Check (significant) warnings from built in Visual Studio code analysis. | **There are lots of errors that not been checked** |
| Check Intellisense errors and warnings. | **Not checked** |

**Overall tool usage performance verdict** (0=poor, 5=excellent): **2**

**1.2 PATTERN VALIDATION**

|  |  |  |
| --- | --- | --- |
| **Design Pattern** | **Checks** | **Result** |
| Authenticator | Check if the pattern separates the subject for the authentication method and the authentication database | Is used correctly |
| Check if the solution securely implements access to different user accounts (e.g using encryption) | **There are two type of user (user, admin) but there is no difference between them that mean both can do the same thing** |
| Check if the proofofid is implemented correctly | The proofOfid is used but very poor |
| **Overall pattern implementation verdict** (0=poor, 5=excellent): 2 | | |
| Secure logger | Check if the solution correctly supports different logging methods (e.g binary, text file, console output) | Not used at all |
| Check if the solution correctly supports different formatting options (e.g HTML, encryption etc) using decorator pattern. | Not used at all |
| Check if the solution uses factory method for different loggers | **No factory methods for different logger** |
| **Overall pattern implementation verdict** (0=poor, 5=excellent): **2** | | |

## 1.3 SUMMARY AND OVERALL VERDICT

Here you summarise your findings in a table and provide a final "mark", as before out of the range between 0 and 5:

|  |  |  |
| --- | --- | --- |
| **Check Type** | **Comment** | **Verdict** |
| Integer | **Arithmetic operations were all made using dedicated methods which all performed pre-operator wrap checks. << was not used.** | **3** |
| String | **Not Declaration for a string in memory** | 1 |
| Memory | **There is a memory leak** | 1 |
| Formatted I/O | **no parametric of flag functions set** | 3 |
| File I/O | **There is no code to prevent an existing file from being overwritten by first opening it for reading before opening it for writing.** | 3 |
| Pointers | **No smart pointer** | 1 |
| Tool Usage | **There are lots of errors that not been checked** | 2 |
| Pattern 1 | **There are two type of user (user, admin) but there is no difference between them that mean both can do the same thing** | 2 |
| Pattern 2 | **No factory methods for different logger** | 2 |
| **OVERALL** |  | 3 |

1. **REPORT ON PROGRAM 2.ZIP**
   1. **CODE VALIDATION**
      1. **INTEGER VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Unsigned integer wrap | Check if wrap checks are made when vulnerable operators are used (+, -,\*,++, etc.) | **Arithmetic operations were all made using dedicated methods which all performed pre-operator wrap checks. << was not used.** |
| Search for C99 size constants such as ‘UINT16\_MAX’ for wrap checking. | **Not used** |
| Check for C99 data types such as uint16\_t instead of just int. | **Not used** |
| Signed integer overflow | Check if wrap checks are made when vulnerable operators are used (+, -,\*,++, etc.) | **Used occasionally** |
| Search for C99 size constants such as INT16\_MAX for wrap check. | **Not used** |
| Search for C99 data types such ‘int16\_t’ instead of just int. | **Not used** |
| Loss of data due to type-casting | Check for any instances of type casting. (e.g. (int)double) | **Is used** |

**Overall integer security performance verdict** (0=poor, 5=excellent): **2**

* + 1. **STRING VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Storage for string must have enough space for data and null terminator | Checks for use of std::string wherever possible (e.g. instead of char\*) | **Used occasionally** |
| When creating a char array or allocating space for a string, ensure that there is space for the null terminator. | **Not used** |
| Do not attempt to create a std::string from a null pointer | Check that pointers are not null before attempting to create the std::string. (e.g tmpPtr= std::getenv("TMP");  if( tmpPtr == NULL ) ) | **Nullptr is used correctly** |
| Use valid references, pointers, and iterators to reference elements of a string | Check that an index/iterator is valid for the string before using it to access elements of a string. | **Not used** |
| Check for valid references to reference elements of a string. | **Is used correctly** |

**Overall string security performance verdict** (0=poor, 5=excellent): **4**

* + 1. **MEMORY VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Do not access freed memory | Check if memory has been accessed after it was deallocated | **Not been accessed after deallocated** |
| Only free memory that was allocated dynamically. | Check if an appropriate deallocator was used for dynamically allocated memory  (e.g. delete() [], free etc) | **Yes, been used and there Is no memory leak** |
| Check if the destructors of objects are used correctly to prevent memory leaks | **Yes, been used correctly** |
| Check if smart pointers are used correctly to provide automatic memory management | **Smart pointer is used correctly** |
| Allocate sufficient memory | Check if sufficient amount of memory has been allocated for a given task (e.g. in arrays, vectors etc) | **That is been allocated correctly** |
| Check if memory allocation errors are handled correctly (e.g. using std::bad\_alloc exception or checking nulllptr) | **Memory allocation is handled correctly** |

**Overall memory security performance verdict** (0=poor, 5=excellent): **4**

* + 1. **FORMATTED INPUT/OUTPUT VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Formatted I/O | Check if cin & cout were used instead of scan & print | **Is used correctly** |
| Check if parametric manipulator functions were used correctly (e.g. setprecision etc) | **Not used** |
| Unformatted I/O | Is ‘user input’ from the keyboard canonicalised, normalised, sanitised, if so how? | Have done so |
| Check if the data fall within the expected domain of valid input (e.g. uint16\_t overflow) | **Overflow is not been checked** |
| Has the programmer used conversion to integers or float from a string? if so has this been done securely? | **Yes but not been done securely** |

**Overall formatted security performance verdict** (0=poor, 5=excellent): **4**

* + 1. **FILE I/O VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Avoid TOCTOU race conditions while accessing files | Check for race conditions, where there are two or more file operations that depend on the same resource. Has this been handled seccurly? | **There are no race conditions** |
| Distinguish between characters read from a file and EOF | Check to see if user has made use of an end of file check, so that characters will only be read from a file until the end of the file is reached. | **No, the end of file is not used** |
| Do not assume that fgets() returns a nonempty string when successful | Check that the return of a call to fgets() is not an empty string, or use functions such as strchr() that handle empty strings well. | **Not function to handle empty string** |
| Do not alternately input and output from a stream without an intervening flush or positioning call | Check for undefined behaviour if no intervening flush or positioning call between the calls to fread() and fwrite() | **Is sued correctly** |
| Securely handle file opening, read, write operations | Check for secure handling of error if the file fails to open, or read and write operation fails. | **Is sued correctly** |
| Close files when they are no longer needed | Check if user has closed all files that are opened. | **Is sued correctly** |

**Overall file security performance verdict** (0=poor, 5=excellent): **4**

* + 1. **POINTER VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks Made** | **Result** |
| nullptr pointer | Check to see if all variables containing pointers are referenced correctly. E.g. uninitialized pointers are set to null, Pointers were mostly checked if they were null before being used etc. | **Is sued correctly** |
| Confusing pointer and value | Check to see the user hasn’t confused the pointer with the value the pointer points too. E.g. using \*x or x instead of &x and vice versa. | **Been checked (no confusion)** |
| Pointer miscalculation | Check the user hasn’t misused the pointers so that a calculation is wrong, or memory address is returned instead of the correct value. | **All pointer is used correctly** |
| Smart pointers | Check if smart pointers are used correctly and in preference to “naked” pointers | **Smart pointer is not used** |

**Overall pointer security performance verdict** (0=poor, 5=excellent): **3**

* + 1. **AUTOMATED QUALITY TOOL USAGE**

|  |  |  |
| --- | --- | --- |
| **Tools** | **Checks** | **Result** |
| Compiler and linker flags | Check use of secure compiler flags ( e.g. /RTC , warning levels etc) | **There are more errors when the warning level set to 3** |
| Check use of secure linker flags /DYNAMICBASE, /NXCOMPAT ect) | **Not used** |
| Code analysis | Check for Evidence of static code analysis use. | **Not used** |
| Check (significant) warnings from built in Visual Studio code analysis. | **There are lots of errors that not been checked** |
| Check Intellisense errors and warnings. | **All waring errors are left with no checking** |

**Overall Tool usage performance verdict** (0=poor, 5=excellent): **3**

* 1. **PATTERN VALIDATION**

|  |  |  |
| --- | --- | --- |
| **Design Pattern** | **Checks** | **Result** |
| Authenticator | Check if the pattern separates the subject for the authentication method and the authentication database | **Is handled correctly** |
| Check if the solution securely implements access to different user accounts (e.g using encryption) | **Is used correctly** |
| Check if the proofofid is implemented correctly | **Proof of id is not implemented correctly** |
| **Overall pattern implementation verdict** (0=poor, 5=excellent): 3 | | |
| Secure logger | Check if the solution correctly supports different logging methods (e.g binary, text file, console output) | supports different logging methods is not used (binary , text file) |
| Check if the solution correctly supports different formatting options (e.g HTML, encryption etc) using decorator pattern. | Different formatting options are sued correctly |
| Check if the solution uses factory method for different loggers | **There is factory method for different logger** |
| **Overall pattern implementation verdict** (0=poor, 5=excellent): 3 | | |

* 1. **SUMMARY AND OVERALL VERDICT**

Here you summarise your findings in a table and provide a final "mark", as before out of the range between 0 and 5:

|  |  |  |
| --- | --- | --- |
| **Check Type** | **Comment** | **Verdict** |
| Integer | C99 size constants such as INT16\_MAX for wrap check. | **2** |
| String | **Used occasionally** | 4 |
| Memory | **Memory allocation is handled correctly** | 4 |
| Formatted I/O | **Overflow is not checked** | 4 |
| File I/O | **Not function to handle empty string** | 4 |
| Pointers | **smart pointer is not used** | 3 |
| Tool Usage | **There are more errors when the warning level set to 3** | 3 |
| Pattern 1 | **Is handled correctly** | 3 |
| Pattern 2 | **There is factory method for different logger** | 3 |
| **OVERALL** |  | 3.5 |

1. **REPORT ON PROGRAM 3.ZIP**
   1. **CODE VALIDATION**
      1. **INTEGER VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| Vulnerability Type | **Checks** | **Result** |
| Unsigned integer wrap | Check if wrap checks are made when vulnerable operators are used (+, -,\*,++, etc.) | **Arithmetic operations were all made using dedicated methods which all performed pre-operator wrap checks.** |
| Search for C99 size constants such as ‘UINT16\_MAX’ for wrap checking. | **C99 used correctly** |
| Check for C99 data types such as uint16\_t instead of just int. | **C99 used correctly** |
| Signed integer overflow | Check if wrap checks are made when vulnerable operators are used (+, -,\*,++, etc.) | **Wrap checks are made when vulnerable operators are used** |
| Search for C99 size constants such as INT16\_MAX for wrap check. | **C99 used correctly** |
| Search for C99 data types such ‘int16\_t’ instead of just int. | **C99 used correctly** |
| Loss of data due to type-casting | Check for any instances of type casting. (e.g. (int)double) | **used correctly** |

**Overall integer security performance verdict** (0=poor, 5=excellent): **4**

* + 1. **STRING VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Storage for string must have enough space for data and null terminator | Checks for use of std::string wherever possible (e.g. instead of char\*) | **Strings are used correctly** |
| When creating a char array or allocating space for a string, ensure that there is space for the null terminator. | **There is no space for the null terminator** |
| Do not attempt to create a std::string from a null pointer | Check that pointers are not null before attempting to create the std::string. (e.g tmpPtr= std::getenv("TMP");  if( tmpPtr == NULL ) ) | **Not used** |
| Use valid references, pointers, and iterators to reference elements of a string | Check that an index/iterator is valid for the string before using it to access elements of a string. | **Used** |
| Check for valid references to reference elements of a string. | **Is used correctly** |

**Overall string security performance verdict** (0=poor, 5=excellent): **3**

* + 1. **MEMORY VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Do not access freed memory | Check if memory has been accessed after it was deallocated | **Not accessed after deallocated** |
| Only free memory that was allocated dynamically. | Check if an appropriate deallocator was used for dynamically allocated memory  (e.g. delete() [], free etc) | **Not used** |
| Check if the destructors of objects are used correctly to prevent memory leaks | **There is memory leak** |
| Check if smart pointers are used correctly to provide automatic memory management | **Smart pointer is used** |
| Allocate sufficient memory | Check if sufficient amount of memory has been allocated for a given task (e.g. in arrays, vectors etc) | **There is no sufficient amount of memory has been allocated for a given task** |
| Check if memory allocation errors are handled correctly (e.g. using std::bad\_alloc exception or checking nulllptr) | **Memory allocation Not been handled** |

**Overall memory security performance verdict** (0=poor, 5=excellent): **3**

* + 1. **FORMATTED INPUT/OUTPUT VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Formatted I/O | Check if cin & cout were used instead of scan & print | **Used correctly** |
| Check if parametric manipulator functions were used correctly (e.g. setprecision etc) | **Used correctly** |
| Unformatted I/O | Is ‘user input’ from the keyboard canonicalised, normalised, sanitised, if so how? | **Used correctly** |
| Check if the data fall within the expected domain of valid input (e.g. uint16\_t overflow) | **Overflow been checked** |
| Has the programmer used conversion to integers or float from a string? if so has this been done securely? | **Is used correctly** |

**Overall formatted security performance verdict** (0=poor, 5=excellent): **4**

* + 1. **FILE I/O VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Avoid TOCTOU race conditions while accessing files | Check for race conditions, where there are two or more file operations that depend on the same resource. Has this been handled seccurly? | **Been handled correctly** |
| Distinguish between characters read from a file and EOF | Check to see if user has made use of an end of file check, so that characters will only be read from a file until the end of the file is reached. | **Not used** |
| Do not assume that fgets() returns a nonempty string when successful | Check that the return of a call to fgets() is not an empty string, or use functions such as strchr() that handle empty strings well. | **Not used** |
| Do not alternately input and output from a stream without an intervening flush or positioning call | Check for undefined behaviour if no intervening flush or positioning call between the calls to fread() and fwrite() | **Used but not correctly** |
| Securely handle file opening, read, write operations | Check for secure handling of error if the file fails to open, or read and write operation fails. | **Not been used correctly** |
| Close files when they are no longer needed | Check if user has closed all files that are opened. | **Not used** |

**Overall file security performance verdict** (0=poor, 5=excellent): **4**

* + 1. **POINTER VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks Made** | **Result** |
| nullptr pointer | Check to see if all variables containing pointers are referenced correctly. E.g. uninitialized pointers are set to null, Pointers were mostly checked if they were null before being used etc. | **Is used correctly** |
| Confusing pointer and value | Check to see the user hasn’t confused the pointer with the value the pointer points too. E.g. using \*x or x instead of &x and vice versa. | **Is used correctly (no confusion)** |
| Pointer miscalculation | Check the user hasn’t misused the pointers so that a calculation is wrong, or memory address is returned instead of the correct value. | **Is used correctly** |
| Smart pointers | Check if smart pointers are used correctly and in preference to “naked” pointers | **Smart pointer Is used but not correctly** |

**Overall pointer security performance verdict** (0=poor, 5=excellent): **4**

* + 1. **AUTOMATED QUALITY TOOL USAGE**

|  |  |  |
| --- | --- | --- |
| **Tools** | **Checks** | **Result** |
| Compiler and linker flags | Check use of secure compiler flags ( e.g. /RTC , warning levels etc) | **Code is not running** |
| Check use of secure linker flags /DYNAMICBASE, /NXCOMPAT ect) | **Code is not running** |
| Code analysis | Check for Evidence of static code analysis use. | **Code is not running** |
| Check (significant) warnings from built in Visual Studio code analysis. | **Code is not running** |
| Check Intellisense errors and warnings. | **Not been checked** |

**Overall Tool usage performance verdict** (0=poor, 5=excellent): **3**

* 1. **PATTERN VALIDATION**

|  |  |  |
| --- | --- | --- |
| **Design Pattern** | **Checks** | **Result** |
| Authenticator | Check if the pattern separates the subject for the authentication method and the authentication database | **Used correctly** |
| Check if the solution securely implements access to different user accounts (e.g using encryption) | **Used correctly** |
| Check if the proofofid is implemented correctly | **Proofoid not been used correctly** |
| **Overall pattern implementation verdict** (0=poor, 5=excellent): **4** | | |
| Secure logger | Check if the solution correctly supports different logging methods (e.g binary, text file, console output) | **Used** |
| Check if the solution correctly supports different formatting options (e.g HTML, encryption etc) using decorator pattern. | **Used correctly** |
| Check if the solution uses factory method for different loggers | **Used correctly** |
| **Overall pattern implementation verdict** (0=poor, 5=excellent): **5** | | |

* 1. **SUMMARY AND OVERALL VERDICT**

Here you summarise your findings in a table and provide a final "mark", as before out of the range between 0 and 5:

|  |  |  |
| --- | --- | --- |
| **Check Type** | **Comment** | **Verdict** |
| Integer | **C99 used correctly** | **4** |
| String | **There is no space for the null terminator** | **3** |
| Memory | **There is no sufficient amount of memory has been allocated for a given task** | **3** |
| Formatted I/O | **Overflow been checked (there no overflow)** | **4** |
| File I/O | **if user has closed all files that are opened.** | **4** |
| Pointers | Smart pointer is used but still there is a memory leak | 4 |
| Tool Usage | **Code is not running** | **3** |
| Pattern 1 | **Proofoid not been used correctly** | **5** |
| Pattern 2 | **the solution uses factory method for different loggers** | **4** |
| **OVERALL** |  | **4** |