**CS2S562**

**Secure Software Development**

**Validation Report Guide**

Your name **Jessica Clara Fealy**

Enrolment number **18024092**

Date **26th March 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

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.) | Not been used |
| Search for C99 size constants such as ‘UINT16\_MAX’ for wrap checking. | Line 195 Sensors.cpp, UINT32\_MAX has been used to ensure no wrapping has been caused.  Yet this function does not actually do what is it intended to do |
| Check for C99 data types such as uint16\_t instead of just int. | Mostly C99 types have been used, yet in the WaterTypeFactor unsigned (which is not C99 has been used) |
| Signed integer overflow | Check if wrap checks are made when vulnerable operators are used (+, -,\*,++, etc.) | Not been used |
| Search for C99 size constants such as INT16\_MAX for wrap check. | Been used to determine if temp is outside bounds of INT16\_MAX |
| Search for C99 data types such ‘int16\_t’ instead of just int. | Been used in some situations, yet in other places non C99 types are used |
| Loss of data due to type-casting | Check for any instances of type casting. (e.g. (int)double) | Implicit cast with temp \* 1.8, but no explicit casting |

**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\*) | For input on line 116 there is no need to use char\* for input, as string would be more secure. |
| When creating a char array or allocating space for a string, ensure that there is space for the null terminator. | Malloc(100) is being used to allocate a block of memory to input, but if the user enters more that 100 chars this will overflow, using a string would prevent this from happening |
| 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 ) ) | Line 120, Sensors.cpp, this line checks that input is not a nullptr |
| 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. | Cannot check, as no relevant code |
| Check for valid references to reference elements of a string. | Cannot check as no relevant code |

**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 | No attempts to access deallocated memory |
| Only free memory that was allocated dynamically. | Check if an appropriate deallocator was used for dynamically allocated memory  (e.g. delete() [], free etc) | Free has been used to deallocate a char\* create from malloc – this is correct  And all instances of new have been correctly deleted using Delete |
| Check if the destructors of objects are used correctly to prevent memory leaks | No destructors used, yet none were needed |
| Check if smart pointers are used correctly to provide automatic memory management | No smart pointers were used |
| Allocate sufficient memory | Check if sufficient amount of memory has been allocated for a given task (e.g. in arrays, vectors etc) | The char\* input was allocated sufficient memory for input requirements of Yes or No |
| Check if memory allocation errors are handled correctly (e.g. using std::bad\_alloc exception or checking nulllptr) | Char\* input is checked for nullptr before use. factory and wType are not checked for nullptr before use |

**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 | Scan and print are never used |
| Check if parametric manipulator functions were used correctly (e.g. setprecision etc) | Set precision has been used, but no other functions have been used |
| Unformatted I/O | Is ‘user input’ from the keyboard canonicalised, normalised, sanitised, if so how? | When selection 0 in the menu causes a system crash. The user input is not being correctly checked |
| Check if the data fall within the expected domain of valid input (e.g. uint16\_t overflow) | Overflow is not being checked on the only user input |
| Has the programmer used conversion to integers or float from a string? if so has this been done securely? | No such conversations have been preformed |

**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 securely? | Triple lock is in place to check for TOCTOU 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. | Gcount() has been used to check for EOF |
| 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. | Read is used to get the data from file, which is defined and has built in error checkers |
| 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() | They have used read and write which check for undefined behaviour |
| 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. | No exception catching |
| Close files when they are no longer needed | Check if user has closed all files that are opened. | File closing is only preformed in conditional statements which may not always be called upon |

**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. | Wherever a pointer variable has been used, all are initialised to nullptrs.  In one instance where nullptr has not been correctly checked |
| 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. | There is no confusion between which pointers to use |
| 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. | Correct pointer has been used throughout |
| Smart pointers | Check if smart pointers are used correctly and in preference to “naked” pointers | Raw pointers are being used throughout, no smart pointers |

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

**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) | Only default, no additional |
| Check use of secure linker flags /DYNAMICBASE, /NXCOMPAT ect) | Only dynamicbase and NXcompat set, no additional linkers |
| Code analysis | Check for Evidence of static code analysis use. | Code analysis on build is switched off, this would prevent warnings and errors on each build. Mircosoft rules set to : Native recommended rules. Not set correctly to all rules  This is evidence against static code analysis |
| Check (significant) warnings from built in Visual Studio code analysis. | No significant warnings present |
| Check Intellisense errors and warnings. | With correct warning level and complier flags set, 60 intellisense warnings are discovered. No errors. |

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

**1.2 PATTERN VALIDATION**

|  |  |  |
| --- | --- | --- |
| **Design Pattern** | **Checks** | **Result** |
| Authenticator | Check if the pattern separates the subject for the authentication method and the authentication database | User and authentication are separate |
| Check if the solution securely implements access to different user accounts (e.g using encryption) | ID and Passport has not been encrypted |
| Check if the proofofid is implemented correctly | This has not been properly used, they are only checking if proofofID has been set not if the user and proofofid is the same |
| **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 been used |
| Check if the solution correctly supports different formatting options (e.g HTML, encryption etc) using decorator pattern. | Not been used |
| Check if the solution uses factory method for different loggers | Not been used |
| **Overall pattern implementation verdict** (0=poor, 5=excellent): 0 | | |

## 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 | No wrapping / overflow checks have been performed for either signed or unsigned ints, and not consistent uses of C99 types | 2 |
| String | This uses a char\* where using a STL string would be more suitable and secure, the char\* has been correctly protected | 4 |
| Memory | Overall memory management security is done very well. But not all allocated memory was properly checked for null pointers before use. Using smart pointers would additionally grant a more secure system. | 4 |
| Formatted I/O | Formatted I/O has some serious security flaws, user inputs are not being correctly checked with program crashes happening with the wrong input. And there is only one time a parametric manipulator has been used at all. | 3 |
| File I/O | File IO has been done very well, with adding exception checks full marks would’ve been given | 4 |
| Pointers | Pointers are mostly used correctly with minor issues, smart pointers being used would’ve made the program more secure | 3 |
| Tool Usage | Flags have not been set correctly, and a lot of warnings are still present | 0 |
| Pattern 1 | While the authorisation patterns works, there’s no encryption or saving to file of users, and proofofID is not being used correctly | 2 |
| Pattern 2 | No secure pattern | 0 |
| **OVERALL** | This program excels in some topic but has significant draw back in others. Not correctly setting flags and other tools dropped points, and the implementations of secure patterns also dropped points. However, string and memory security was done very well. | 2.5 |

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.) | No wrap checks on unsigned int operations. |
| Search for C99 size constants such as ‘UINT16\_MAX’ for wrap checking. | No C99 constants used |
| Check for C99 data types such as uint16\_t instead of just int. | No uses of C99 types |
| Signed integer overflow | Check if wrap checks are made when vulnerable operators are used (+, -,\*,++, etc.) | No overflow checks on int operations, example: line 25 Status.cpp |
| Search for C99 size constants such as INT16\_MAX for wrap check. | No C99 constants used as no overflow wraps were used |
| Search for C99 data types such ‘int16\_t’ instead of just int. | No C99 int types were used |
| Loss of data due to type-casting | Check for any instances of type casting. (e.g. (int)double) | Static casting from an unsigned long to unsigned int. This will have some loss of data, yet the intention is to create a random seed so the loss of data will not create any security issues |

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

* + 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\*) | Char\* has not been used  In some instances such as Interface::optionSelect char is used where string could also be easily used.  Char has also been used in Authenticator for the “key” this is done correctly  In all other instances String has been used |
| When creating a char array or allocating space for a string, ensure that there is space for the null terminator. | Strings and chars have not been initialised or allocated memory on creation.  When using cin in Interface::Print, the data is read directly to Char option, option is not initialised on creation and would benefit from being a string rather than char |
| 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 ) ) | No checks are preformed |
| 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. | Strings elements are only access inside for loops and done so securely |
| Check for valid references to reference elements of a string. | No use case of this in the program |

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

* + 1. **MEMORY VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Do not access freed memory | Check if memory has been accessed after it was deallocated | Memory has not been access after deallocation |
| Only free memory that was allocated dynamically. | Check if an appropriate deallocator was used for dynamically allocated memory  (e.g. delete() [], free etc) | Both factory and logger were deallocated correctly using Delete. Yet proofofID was incorrectly deallocated using Free. |
| Check if the destructors of objects are used correctly to prevent memory leaks | No destructor for LoggerDecorator is delete the ownership of Logger\* Contents |
| Check if smart pointers are used correctly to provide automatic memory management | Smart pointers were not used |
| Allocate sufficient memory | Check if sufficient amount of memory has been allocated for a given task (e.g. in arrays, vectors etc) | Sufficient space was allocated |
| Check if memory allocation errors are handled correctly (e.g. using std::bad\_alloc exception or checking nulllptr) | No allocation errors are being handled |

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

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

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

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

* + 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? | No TOCTOU race conditions implemented when writing to a file |
| 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 instances of a file being read from |
| 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. | No instances of a file being read from |
| 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() | A file is not being written or read from in the same function. Each instance of file manipulation has the file being opened and closed before another operations is attempted |
| 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. | Only myfile.fail() is being used, no additional exceptions, and no try and catch is being used |
| Close files when they are no longer needed | Check if user has closed all files that are opened. | Files are being closed correctly |

**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. | Logger\* Contents is not being initialised  No checks if factory or logger are null pointers before use.  proofofID is initialised to null pointer upon creation |
| 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. | All pointers are being used correctly |
| 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. | Pointers are being used correctly |
| Smart pointers | Check if smart pointers are used correctly and in preference to “naked” pointers | Smart pointers are not being 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) | Complier flags are not set  Warning levels are set to Level3 not Allwarnings |
| Check use of secure linker flags /DYNAMICBASE, /NXCOMPAT ect) | No linker flags are set |
| Code analysis | Check for Evidence of static code analysis use. | Under the project’s properties, submenu Code Analysis all options under general are turned on, and the Microsoft rule set is se to All Rules |
| Check (significant) warnings from built in Visual Studio code analysis. | Code analysis does not generate any significant warnings |
| Check Intellisense errors and warnings. | No errors presents  Intellisense detects 89 warnings |

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

* 1. **PATTERN VALIDATION**

|  |  |  |
| --- | --- | --- |
| **Design Pattern** | **Checks** | **Result** |
| Authenticator | Check if the pattern separates the subject for the authentication method and the authentication database | AuthenticatiorInfo separates the authentication database and methods |
| Check if the solution securely implements access to different user accounts (e.g using encryption) | Users are hard coded and no encryption is used |
| Check if the proofofid is implemented correctly | ProofofID has not been used correctly, only being used to check if the system proofofID has been set, not that the proofofID matches the user |
| **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) | Different logging methods have been implemented correctly |
| Check if the solution correctly supports different formatting options (e.g HTML, encryption etc) using decorator pattern. | Different formats and encryption have been implemented correctly |
| Check if the solution uses factory method for different loggers | Factor method used |
| **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 | No attempts at unsigned wrapping or signed in overflow protections. C99 types were not used. However, correct uses of static casting | 1 |
| String | In this case, points would’ve been gained by just using strings and not Chars, as no instance of Char was needed. Additionally, in the future correct initialisation and memory allocation for strings and chars should be observed. However, strings were correctly accessed with indexes. | 2 |
| Memory | Overall memory management was okay, good attempts are correctly deleting allocated memory and not access memory after deallocation. Improvements could be made on destructors and including correct error handling. | 2 |
| Formatted I/O | Poor attempt. No attempts to check or validate user input in any way. With some inputs crashing the program. Correct uses or Cin and Cout. | 2 |
| File I/O | Without evidence of reading from a file a lot of points can’t be spoken about. When writing to a file there is evidence of closing files correctly and no instance has a file needed a flush or position call. TOCTOU race conditions have not been taken into account. Overall good attempt, but adding more secure methods will grant higher mark | 3 |
| Pointers | Pointers have been used correct most of the time. However, proper initialisation to nullptr of pointers and checking to ensure that defined pointers are not nullptrs should be implemented into the code. Otherwise use smart pointers | 3 |
| Tool Usage | Showed evidence of visual studio code analysis working, but no flags were set. And still 89 warnings present | 2 |
| Pattern 1 | Separating Authenticator and AuthenticatorInfo is very secure, but not saving user details to a file, and not using encryption is not flexible. ProofofID was not correctly used. With easy some chances to the code would fix all these issues | 2 |
| Pattern 2 | Logger has been very well implemented, with only notable flaw being no deconstructor for LoggerDectorator | 5 |
| OVERALL | Overall this project done okay with security. With most points being lost with Interger security, and most marks being gained with the secure logger. Simple improvements would drastically improve these points. | 2.5 |

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.) | Excellent wrapping protections |
| Search for C99 size constants such as ‘UINT16\_MAX’ for wrap checking. | Uses UINT16\_MAX through UINT64\_MAX to check wrapping |
| Check for C99 data types such as uint16\_t instead of just int. | No instances of unsigned int, with a typedef called uint16\_s through uint64\_s which is using uint16\_t though uint64\_t |
| Signed integer overflow | Check if wrap checks are made when vulnerable operators are used (+, -,\*,++, etc.) | Excellent overflow protections |
| Search for C99 size constants such as INT16\_MAX for wrap check. | Uses INT\*\*\_MIN and INT\*\*\_MAX through 16 – 32 to overflow protect ints |
| Search for C99 data types such ‘int16\_t’ instead of just int. | No instances of int being used insecurely. A typedef called int\*\*\_s (16-64) has been created using int\*\*\_t (16-32) |
| Loss of data due to type-casting | Check for any instances of type casting. (e.g. (int)double) | Correct static casting has been observed throughout the program |

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

* + 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\*) | Correct uses of char\* and strings |
| When creating a char array or allocating space for a string, ensure that there is space for the null terminator. | No relevant |
| 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 ) ) | Has done null checks before creating strings |
| 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. | Strings are never iterated through |
| Check for valid references to reference elements of a string. | Strings characters are never referenced |

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

* + 1. **MEMORY VULNERABILITIES**

|  |  |  |
| --- | --- | --- |
| **Vulnerability Type** | **Checks** | **Result** |
| Do not access freed memory | Check if memory has been accessed after it was deallocated | Memory has 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) | Used delete correctly for deallocating memory |
| Check if the destructors of objects are used correctly to prevent memory leaks | No memory leaks |
| Check if smart pointers are used correctly to provide automatic memory management | Smart pointers have been used |
| Allocate sufficient memory | Check if sufficient amount of memory has been allocated for a given task (e.g. in arrays, vectors etc) | Arrays of static size are not used, vectors have been used where required |
| Check if memory allocation errors are handled correctly (e.g. using std::bad\_alloc exception or checking nulllptr) | Pointers are not null checked before doing operations |

**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 | Scan and print are not used |
| Check if parametric manipulator functions were used correctly (e.g. setprecision etc) | Some parametric functions such as setw are used correctly in the program |
| Unformatted I/O | Is ‘user input’ from the keyboard canonicalised, normalised, sanitised, if so how? | Text can only be accepted in one section – TextInput.cpp. This has abstracted and used by many different parts of the program.  Checks preformed is the fail bit, to ensure that the text input is in the sizes expected. Text input has been sanitised. If an error is found, then the user is redirected for additional inputs. No explicit error messages are shown to the user |
| Check if the data fall within the expected domain of valid input (e.g. uint16\_t overflow) | Data is being checked that it falls within boundaries. |
| Has the programmer used conversion to integers or float from a string? if so has this been done securely? | No conversions |

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

* + 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? | As files are only written to and read from then this isn’t needed. Yet no attempts have been implemented |
| 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 relevant as a file has not been read from |
| 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 relevant as a file has not been read from |
| 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() | Files are being flushed after being written too, this can be found line 43 FileLogger.cpp. However, as a file is never read from this isn’t strictly necessary |
| 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. | Errors are handled well when writing from a files, errors are being checked when opening a file, writing to a file and closing the file |
| Close files when they are no longer needed | Check if user has closed all files that are opened. | The file is being closes as long as no fail has occurred inside the ~FileLogger |

**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. | Pointer are checked for being null points before use |
| 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. | Pointers have not been confused when doing operations |
| 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. | Pointers have not been misused |
| Smart pointers | Check if smart pointers are used correctly and in preference to “naked” pointers | Smart pointers have been used, yet a raw pointer has been used unnecessarily where a smart pointer would’ve been more appropriate |

**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) | Warning level is set to 4 not all warnings  Some complier flags such as Gs are missing, where other flags such as RTC is present |
| Check use of secure linker flags /DYNAMICBASE, /NXCOMPAT ect) | Secure complier flags are set |
| Code analysis | Check for Evidence of static code analysis use. | Code analysis on build has been turned off, this is evidence against static code analysis use. Additionally, Microsoft native rules are on rather than Microsoft all rules |
| Check (significant) warnings from built in Visual Studio code analysis. | No significant warnings |
| Check Intellisense errors and warnings. | No intellisense warnings present |

**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 | User repository contains the hard-coded log in credentials which is separate from Authenticator |
| Check if the solution securely implements access to different user accounts (e.g using encryption) | Login details are hard coded, with no encryption or getting from a file |
| Check if the proofofid is implemented correctly | ProofofID is not used. No token systems are implemented, so consist login security cannot be maintained. |
| **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) | Supports 2 different logging methods, console, and file |
| Check if the solution correctly supports different formatting options (e.g HTML, encryption etc) using decorator pattern. | Different formatting types are available such as XML and CVS by using a decorator pattern. Yet this pattern hasn’t been correctly implemented. With a logger decorator you can do any formatting via one function inside the decorator, instead this person has used multiple files and functions to do the same job |
| Check if the solution uses factory method for different loggers | The factory method has been created and works, but another cpp file has been used to do this, where the same functionality can be done via internal logger decorator cpp |
| **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 | Excellent integer protection, no issues could be found. If anything, this approach seemed over engineered, yet very effective | 5 |
| String | Excellent string protections. All instances of strings and char\* used appropriately | 5 |
| Memory | Memory managed has mostly bee handled excellently. Minor hiccup on null pointer checks for some instances, but in general very good. | 4 |
| Formatted I/O | No issues found with formatted I/O, input is taken very sensibly from the user | 5 |
| File I/O | A TOCTOU approach has not been implemented as no attempts to read from a file has been attempted. Correct implementation of reading from a file would’ve granted the full 5 points | 4 |
| Pointers | Pointer protections has been excellently done, only one instance of using a raw pointer where a smart pointer would’ve been more appropriate. | 4 |
| Tool Usage | Complier flags are missing, and evidence of not preforming static code analysis. However, no intellisense warnings detected. Overall satisfactory | 3 |
| Pattern 1 | Authenticator pattern poorly implemented, no proofofID used, and user details not saved to a file nor encrypted. This pattern more reflects a Role Base authentication. | 2 |
| Pattern 2 | The secure logger done satisfactory. The decorator pattern nor the factory pattern have been correctly used. Both decorator and factory are overcomplicated and use excess cpp files, these can easily be condensed and still retain the same functionality. | 3 |
| **OVERALL** | Overall this project has managed security very well, primirally excelling in integer and string security. To get 5 points further work would be require in the secure patterns. | 4 |