**CS2S562**

**Secure Software Development**

**Validation Report Guide**

This guide is a suggestion only to give you an idea how to structure your report for the second coursework. Feel free to deviate from it.

# Title Page

**Minimum information here:**

* Your name
* Enrolment number
* Date
* 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 006.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 realised by classes ..., ... and ...

1.2.2 Authorization pattern realised by classes ..., ... and ...

1.3 Summary and Overall Verdict

2 Report on Program 032.zip

(enter the same structure as for Program1.zip here)

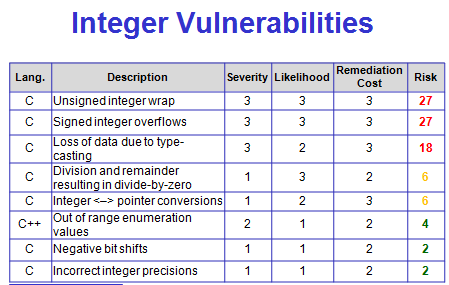
3 Report on Program 036.zip

(enter the 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 the topics we covered. Validate against the **3 top risks** in these tables. For example, in the case of integer vulnerabilities these 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 a 3 checks (where possible) 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**

You may have noticed that there is no risk table for the topic '*Formatted input/output'*. Instead you will find in the lecture slides a set of 'Best Practices' (search for this keyword in the slides). Validate against these 'Best Practices'.

### There is also no risk table for *'Automated tool usage'* but you can surely design your own that checks if compiler flags, linker flags, etc. have been used (the lecture slides show exactly what to look for).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**

Create a similar table for the Authentication Pattern.

## 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.