Specification of Test Bench

Toyota InfoTechnology Center, U.S.A.

【 Table of Contents 】

[1 Scope 4](#_Toc384046358)

[2 Overview of Test Suites 4](#_Toc384046359)

[3 Detailed Specification 8](#_Toc384046360)

[3.1 Bit Shift Bigger Than Integral Type or Negative 8](#_Toc384046361)

[3.2 Dynamic Buffer Overrun 8](#_Toc384046362)

[3.3 Dynamic Buffer Underrun 8](#_Toc384046363)

[3.4 Comparison NULL with Function Pointer 8](#_Toc384046364)

[3.5 Contradict Conditions 8](#_Toc384046365)

[3.6 Integer Precision Lost Because of Cast 8](#_Toc384046366)

[3.7 Data Overflow 8](#_Toc384046367)

[3.8 Data Underflow 8](#_Toc384046368)

[3.9 Deadcode 9](#_Toc384046369)

[3.10 Deadlock 9](#_Toc384046370)

[3.11 Deletion of Data Structure Sentinel 9](#_Toc384046371)

[3.12 Double Free 9](#_Toc384046372)

[3.13 Double Lock 9](#_Toc384046373)

[3.14 Double Release 9](#_Toc384046374)

[3.15 Unintentional Endless Loop 9](#_Toc384046375)

[3.16 Free Non Dynamically Allocated Memory 10](#_Toc384046376)

[3.17 Free NULL Pointer 10](#_Toc384046377)

[3.18 Bad Cast of a Function Pointer 10](#_Toc384046378)

[3.19 Return Value of Function Never Checked 10](#_Toc384046379)

[3.20 Improper Error Handling 10](#_Toc384046380)

[3.21 Improper Termination of Block 10](#_Toc384046381)

[3.22 Useless Assignment 10](#_Toc384046382)

[3.23 Bad Extern Type for Global Variable 10](#_Toc384046383)

[3.24 Invalid Memory Access to Already Freed Area 10](#_Toc384046384)

[3.25 Assign Small Buffer for Structure 10](#_Toc384046385)

[3.26 Live Lock 11](#_Toc384046386)

[3.27 Locked but Never Unlock 11](#_Toc384046387)

[3.28 Memory Allocation Failure 11](#_Toc384046388)

[3.29 Memory Leakage 11](#_Toc384046389)

[3.30 Non Void Function does not Return Value 11](#_Toc384046390)

[3.31 Dereferencing a NULL pointer 11](#_Toc384046391)

[3.32 Static Buffer Overrun 11](#_Toc384046392)

[3.33 Memory Copy at Overlapping Areas 11](#_Toc384046393)

[3.34 Power Related Errors 11](#_Toc384046394)

[3.35 Incorrect Pointer Arithmetic 12](#_Toc384046395)

[3.36 Race Condition 12](#_Toc384046396)

[3.37 Redundant Condition 12](#_Toc384046397)

[3.38 Return of a Pointer to a Local Variable 12](#_Toc384046398)

[3.39 Integer Sign Lost Because of Unsigned Cast 13](#_Toc384046399)

[3.40 Long Lock 13](#_Toc384046400)

[3.41 Cross Thread Stack Access 13](#_Toc384046401)

[3.42 Stack Overflow 13](#_Toc384046402)

[3.43 Stack Underrun 13](#_Toc384046403)

[3.44 Static Buffer Underrun 13](#_Toc384046404)

[3.45 Uninitialized Memory Access 13](#_Toc384046405)

[3.46 Uninitialized Pointer 13](#_Toc384046406)

[3.47 Uninitialized Variable 13](#_Toc384046407)

[3.48 Unlock Without Lock 13](#_Toc384046408)

[3.49 Unused Variable 14](#_Toc384046409)

[3.50 Wrong Arguments Passed to a Function Pointer 14](#_Toc384046410)

[3.51 Division by Zero 14](#_Toc384046411)

[4 Bibliography 15](#_Toc384046412)

【 Tables 】

[Table 1 Defect Types 4](#_Toc384046413)

[Table 2 Specification of Test Bench 4](#_Toc384046414)

[Table 3 Relation between Defect Sub-Type and Test Suite Files 6](#_Toc384046415)

[Table 4 Corner Cases of Pow (x,y) 12](#_Toc384046416)

# Scope

This document discusses the test suite specifications. It provides examples and variations of different types of defects considered for generating the test suites.

# Overview of Test Suites

Based on [1] Annex A (Source Code Weaknesses), we prepare a set of defect types. From [1] Annex A, we select the defects related to embedded systems. Moreover, we add some defect types related to dynamic memory allocation, error handling, multithreading, which can be detected by tools but not mentioned in [1] Annex A.

Table 1 depicts the splitting of the 9 different defect types into specific number of defect sub-types. These defect variations are shortlisted to make the test bench used for evaluation of various tools.

Table 1 Defect Types

|  |  |  |  |
| --- | --- | --- | --- |
| # | Defect Type | Subtype samples | # of Defect Subtypes |
| 1 | Static memory defects | Static buffer overrun, etc. | 2 |
| 2 | Dynamic memory defects | Dynamic buffer overrun, etc. | 5 |
| 3 | Stack related defects | Stack overflow, etc. | 3 |
| 4 | Numerical defects | Division by zero, etc. | 7 |
| 5 | Resource management defects | Invalid memory access to already freed area, etc. | 7 |
| 6 | Pointer related defects | Dereferencing a NULL pointer, etc. | 7 |
| 7 | Concurrency defects | Dead lock, etc. | 8 |
| 8 | Inappropriate code | Dead code, etc. | 7 |
| 9 | Misc defects | Uninitialized variables, etc. | 5 |
| Total | | | **51** |

Table 2 gives a detailed specification of the test bench in terms of the defect sub-type classification. This depicts the fact that the test bench not only supports defects, but also supports defect-free variation to evaluate the false-positive scenarios.

Table 2 Specification of Test Bench

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Defect Subtype | Defect Type | w/ Defect | w/o Defect |
| 1 | Bit shift bigger than integral type or negative | Numerical | 17 | 17 |
| 2 | Dynamic buffer overrun | Dynamic memory | 32 | 32 |
| 3 | Dynamic buffer underrun | Dynamic memory | 39 | 39 |
| 4 | Comparison NULL with function pointer | Pointer related | 2 | 2 |
| 5 | Contradict conditions | Inappropriate code | 10 | 10 |
| 6 | Integer precision lost because of cast | Numerical | 19 | 19 |
| 7 | Data overflow | Numerical | 25 | 25 |
| 8 | Data underflow | Numerical | 12 | 12 |
| 9 | Dead code | Inappropriate code | 13 | 13 |
| 10 | Dead lock | Concurrency | 5 | 5 |
| 11 | Deletion of data structure sentinel | Dynamic memory | 3 | 3 |
| 12 | Double free | Resource management | 12 | 12 |
| 13 | Double lock | Concurrency | 4 | 4 |
| 14 | Double release | Concurrency | 6 | 6 |
| 15 | Unintentional endless loop | Misc | 9 | 9 |
| 16 | Free non dynamically allocated memory | Resource management | 16 | 16 |
| 17 | Free NULL pointer | Pointer related | 14 | 14 |
| 18 | Bad cast of a function pointer | Pointer related | 15 | 15 |
| 19 | Return value of function never checked | Inappropriate code | 16 | 16 |
| 20 | Improper error handling | Inappropriate code | 4 | 4 |
| 21 | Improper termination of block | Inappropriate code | 4 | 4 |
| 22 | Useless assignment | Misc | 1 | 1 |
| 23 | Bad extern type for global variable | Misc | 6 | 6 |
| 24 | Invalid memory access to already freed area | Resource management | 17 | 17 |
| 25 | Assign small buffer for structure | Dynamic memory | 11 | 11 |
| 26 | Live lock | Concurrency | 1 | 1 |
| 27 | Locked but never unlock | Concurrency | 9 | 9 |
| 28 | Memory allocation failure | Resource management | 16 | 16 |
| 29 | Memory leakage | Resource management | 18 | 18 |
| 30 | Non void function does not return value | Misc | 4 | 4 |
| 31 | Dereferencing a NULL pointer | Pointer related | 17 | 17 |
| 32 | Static buffer overrun | Static memory | 54 | 54 |
| 33 | Memory copy at overlapping areas | Dynamic memory | 2 | 2 |
| 34 | Power related errors | Numerical | 29 | 29 |
| 35 | Incorrect pointer arithmetic | Pointer related | 2 | 2 |
| 36 | Race condition | Concurrency | 8 | 8 |
| 37 | Redundant conditions | Inappropriate code | 14 | 14 |
| 38 | Return of a pointer to a local variable | Resource management | 2 | 2 |
| 39 | Integer sign lost because of unsigned cast | Numerical | 19 | 19 |
| 40 | Long lock | Concurrency | 3 | 3 |
| 41 | Cross thread stack access | Stack related | 6 | 6 |
| 42 | Stack overflow | Stack related | 7 | 7 |
| 43 | Stack underrun | Stack related | 7 | 7 |
| 44 | Static buffer underrun | Static memory | 13 | 13 |
| 45 | Uninitialized memory access | Resource management | 15 | 15 |
| 46 | Uninitialized pointer | Pointer related | 16 | 16 |
| 47 | Uninitialized variable | Misc | 15 | 15 |
| 48 | Unlock without lock | Concurrency | 8 | 8 |
| 49 | Unused variable | Inappropriate code | 7 | 7 |
| 50 | Wrong arguments passed to a function pointer | Pointer related | 18 | 18 |
| 51 | Division by zero | Numerical | 16 | 16 |
| Subtotal | | | **638** | **638** |
| Total | | | **1,276** | |

Table 3 shows the relation between the various defect sub-types used for analysis the file names as specified in the test suite.

Table 3 Relation between Defect Sub-Type and Test Suite Files

|  |  |  |  |
| --- | --- | --- | --- |
| # | Defect Sub-type | Defect Type | File Name |
| 1 | Bit shift bigger than integral type or negative | Numerical defects | bit\_shift |
| 2 | Dynamic buffer overrun | Dynamic memory defects | buffer\_overrun\_dynamic |
| 3 | Dynamic buffer underrun | Dynamic memory defects | buffer\_underrun\_dynamic |
| 4 | Comparison NULL with function pointer | Pointer related defects | cmp\_funcadr |
| 5 | Contradict conditions | Inappropriate code | conflicting\_cond |
| 6 | Integer precision lost because of cast | Numerical defects | data\_lost |
| 7 | Data overflow | Numerical defects | data\_overflow |
| 8 | Data underflow | Numerical defects | data\_underflow |
| 9 | Dead code | Inappropriate code | dead\_code |
| 10 | Dead lock | Concurrency defects | dead\_lock |
| 11 | Deletion of data structure sentinel | Dynamic memory defects | deletion\_of\_data\_structure\_sentinel |
| 12 | Double free | Resource management defects | double\_free |
| 13 | Double lock | Concurrency defects | double\_lock |
| 14 | Double release | Concurrency defects | double\_release |
| 15 | Unintentional endless loop | Misc defects | endless\_loop |
| 16 | Free non dynamically allocated memory | Resource management defects | free\_nondynamic\_allocated\_memory |
| 17 | Free NULL pointer | Pointer related defects | free\_null\_pointer |
| 18 | Bad cast of a function pointer | Pointer related defects | func\_pointer |
| 19 | Return value of function never checked | Inappropriate code | function\_return\_value\_unchecked |
| 20 | Improper error handling | Inappropriate code | improper\_error\_handling |
| 21 | Improper termination of block | Inappropriate code | improper\_termination\_of\_block |
| 22 | Useless assignment | Misc defects | insign\_code |
| 23 | Bad extern type for global variable | Misc defects | invalid\_extern |
| 24 | Invalid memory access to already freed area | Resource management defects | invalid\_memory\_access |
| 25 | Assign small buffer for structure | Dynamic memory defects | littlemem\_st |
| 26 | Live lock | Concurrency defects | livelock |
| 27 | Locked but never unlock | Concurrency defects | lock\_never\_unlock |
| 28 | Memory allocation failure | Resource management defects | memory\_allocation\_failure |
| 29 | Memory leakage | Resource management defects | memory\_leak |
| 30 | Non void function does not return value | Misc defects | not\_return |
| 31 | Dereferencing a NULL pointer | Pointer related defects | null\_pointer |
| 32 | Static buffer overrun | Static memory defects | overrun\_st |
| 33 | Memory copy at overlapping areas | Dynamic memory defects | ow\_memcpy |
| 34 | Power related errors | Numerical defects | pow\_related\_errors |
| 35 | Incorrect pointer arithmetic | Pointer related defects | ptr\_subtraction |
| 36 | Race condition | Concurrency defects | race\_condition |
| 37 | Redundant conditions | Inappropriate code | redundant\_cond |
| 38 | Return of a pointer to a local variable | Resource management defects | return\_local |
| 39 | Integer sign lost because of unsigned cast | Numerical defects | sign\_conv |
| 40 | Long lock | Concurrency defects | sleep\_lock |
| 41 | Cross thread stack access | Stack related defects | st\_cross\_thread\_access |
| 42 | Stack overflow | Stack related defects | st\_overflow |
| 43 | Stack underrun | Stack related defects | st\_underrun |
| 44 | Static buffer underrun | Static memory defects | underrun\_st |
| 45 | Uninitialized memory access | Resource management defects | uninit\_memory\_access |
| 46 | Uninitialized pointer | Pointer related defects | uninit\_pointer |
| 47 | Uninitialized variable | Misc defects | uninit\_var |
| 48 | Unlock without lock | Concurrency defects | unlock\_without\_lock |
| 49 | Unused variable | Inappropriate code | unused\_var |
| 50 | Wrong arguments passed to a function pointer | Pointer related defects | wrong\_arguments\_func\_pointer |
| 51 | Division by zero | Numerical defects | zero\_division |

# Detailed Specification

## Bit Shift Bigger Than Integral Type or Negative

This defect variation helps in identifying *Numerical defects*, which are related to the size of bit shift operation being larger than the size of operands.

## Dynamic Buffer Overrun

This defect variation identifies *Dynamic Memory defects*, which are related to memory access outside dynamically allocated memory space.

## Dynamic Buffer Underrun

This defect variation identifies *Dynamic Memory defects*, which are related to memory access lower than the bounds of the dynamic allocated memory space.

## Comparison NULL with Function Pointer

This defect variation identifies *Pointer related defects*, which are related to a function address comparison with NULL. Usually, this operation is a misuse case that is supposed to be a coding error of a function call with a missing ().

## Contradict Conditions

This defect variation identifies *Inappropriate code*, which are related to validating the conditions that conflict each other.

## Integer Precision Lost Because of Cast

This defect variation identifies *Numerical defects*, which assign a variable of a larger variable type to another variable with smaller data size. The result of which is some information of assigned variable is lost.

## Data Overflow

This defect variation identifies *Numerical defects*, which are illustrated in brief below.

* Integer - the result value of an operation exceed the maximum value of operands. As a result, the result value is incorrect.
  + Signed: Maximum Value+1→Minimum Value (Alternate Sign)
  + Unsigned: Maximum Value+1→0
* Floating Point - the result value of an operation exceed the maximum value of operands. As a result, the result value is infinite.
* Positive Overflow: Maximum Value + Positive Value -> Infinite Positive
* Negative Overflow: Minimum Value – Positive Value -> Infinite Negative

## Data Underflow

This defect variation identifies *Numerical defects*, which are illustrated in brief below.

* Integer…the result value is under the minimum value of operands. As a result, the result value is incorrect.
  + Signed: Minimum Value-1→Maximum Value (Alternate Sign)
  + Unsigned：0-1→Maximum Value
* Floating point…the result value is under the minimum value of operands. As a result, the result value is incorrect.
  + Positive Underflow: Maximum Value/2→0
  + Negative Underflow: Minimum Value2→0

## Deadcode

This defect variation identifies *Inappropriate code*, which validates the existing code that is never executed.

## Deadlock

This defect variation identifies *Concurrency defects*. Deadlock conditions occur when resources with exclusive control are implemented inappropriately i.e. if different orders of locks are sequenced among tasks.

## Deletion of Data Structure Sentinel

This defect variation identifies *Dynamic memory defects*. The accidental deletion of a data-structure sentinel can cause serious programming logic problems. Often data-structure sentinels are used to mark structure of data. A common example of this is the null character at the end of strings. Another common example is linked lists, which may contain a sentinel to mark the end of the list. It is dangerous to allow this type of control data to be easily accessible. Therefore, it is important to protect from the deletion or modification outside of some wrapper interface, which provides safety.

## Double Free

This defect variation identifies *Resource Management defects*, which validates an attempt to free heap memory, which is already “free”.

## Double Lock

This defect variation identifies *Concurrency defects*, which validates if in the same task, the same resource is locked twice, or is never unlocked.

## Double Release

This defect variation identifies *Concurrency defects*, which validates if in the same task, the same resource is unlocked twice.

## Unintentional Endless Loop

This defect variation identifies some *Miscellaneous defects*, which validates no termination of a program due to an infinite loop.

## Free Non Dynamically Allocated Memory

This defect variation identifies *Resource Management defects*, which validates an attempt to free a memory location that was not allocated dynamically.

## Free NULL Pointer

This defect variation identifies *Pointer Related defects*, which validates an attempt to free a pointer which is NULL.

## Bad Cast of a Function Pointer

This defect variation identifies *Pointer Related defects*, which validates an attempt to assign a function pointer to other function pointer that has different arguments and/or return value.

## Return Value of Function Never Checked

This defect variation identifies *Inappropriate code*, which validates if a return value from a function is never evaluated.

## Improper Error Handling

This defect variation identifies *Inappropriate code* in C++ programing, which validates if detected errors are handled properly. For instance, a try block without a catch or throw.

## Improper Termination of Block

This defect variation identifies *Inappropriate code* as in the improper termination of a block of code.

## Useless Assignment

This defect variation identifies *Miscellaneous defects*, which validates if within a certain block of statements, there exists some meaningless assignment.

## Bad Extern Type for Global Variable

This defect variation identifies *Miscellaneous defects*, which validates a global variable being used as a different type in a different file.

## Invalid Memory Access to Already Freed Area

This defect variation identifies *Resource Management defects*, which validates access to a memory location, which is already freed.

## Assign Small Buffer for Structure

This defect variation identifies *Dynamic memory defects*. These defects are used to validate an allocation of a smaller array buffer to a larger structure pointer for data access. When such a structure pointer is accessed, it could result in invalid data read or a buffer overrun.

## Live Lock

This defect variation identifies *Concurrency defects*. Live lock is a condition that occurs when two or more [processes](http://www.webopedia.com/TERM/P/process.html) continually changes their state in response to changes in the other processes. The result is that none of the processes will complete. An analogy is when two people meet in a hallway and each tries to step around the other but they end up swaying from side to side getting in each other's way as they try to get out of the way.

## Locked but Never Unlock

This defect variation identifies *Concurrency defects*, which validates if a resource is locked from access but never unlocked for access.

## Memory Allocation Failure

This defect variation identifies *Resource Management defects*, which validates a memory allocation failure due to insufficient memory. Typically a memory allocation failure would happen when there is a heap overflow condition.

## Memory Leakage

This defect variation identifies *Resource Management defects*, which validates a memory allocation on the heap and but a failure to “free” it, resulting in memory holes.

## Non Void Function does not Return Value

This defect variation identifies *Miscellaneous defects*, which evaluates if a function does not return a value even though its return type is non-void.

## Dereferencing a NULL pointer

This defect variation identifies *Pointer Related defects*, which validates access to an address pointed by a NULL pointer.

## Static Buffer Overrun

This defect variation identifies *Static Memory defects*, which validates an access to the memory area that is not reserved statically, e.g., an oversized index for an array

## Memory Copy at Overlapping Areas

This defect variation identifies *Dynamic Memory defects*, which validates data overwriting while copying of array elements continuously.

## Power Related Errors

This defect variation identifies *Numerical defects*, which validate use of large data as a result of “pow” operations. Double values, which are given as an input to “pow” functions, may not be able to store those big values. The result can be sometimes larger than the maximum value double can hold. Table 4 depicts a tabulation of the corner cases while using “pow” functionality.

* Positive Overflow: Maximum Value ^ Positive Value = Positive Overflow
* Negative Overflow: Minimum Value ^ Negative Value = Negative overflow or underflow

Table 4 Corner Cases of Pow (x,y)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Base  Exponent | Positive Large Number > 1 | Negative Large Number < -1 | Small Absolute Number 0 < x < 1 | Small Absolute Number -1 < x < 0 |
| Positive Large and Even Number > 1 | Overflow | Overflow | Losing precision | Losing precision |
| Positive Large and Odd Number > 1 | Same as above | Underflow | Same as above | Same as above |
| Negative Large and Even Number > 0 | Overflow | Overflow | Loosing Precision | Loosing Precision |
| Negative Large and Odd Number > 0 | Same as above | Same as above | Same as above | Underflow |
| Positive Small Number 1 > y > 0 | Same as above | Same as above | Losing precision | Complex Number (Not Real Number) |
| Negative Small Number -1 < y < 0 | Same as above | Same as above | Same as above | Same as above |

## Incorrect Pointer Arithmetic

This defect variation identifies *Pointer Related defects*, which validates pointer arithmetic resulting to access different kinds of memory areas.

## Race Condition

This defect variation identifies *Concurrency defects*, which validates a race condition. A race condition occurs when 2 or more threads are attempting to access shared data and write into the shared location at the same time. A scheduler can swap between threads at any point and hence the order at which the threads will attempt to access the shared data is unpredictable. Therefore, the result of the change in data is dependent on the thread-scheduling algorithm, i.e. both threads are 'racing' to access/change the data.

## Redundant Condition

This defect variation identifies *Inappropriate code*, which validates any redundant conditions if they exist.

## Return of a Pointer to a Local Variable

This defect variation identifies *Resource Management defects*, which validates if a returned pointer from a function call points to a local variable, and if the caller access the memory via the pointer.

## Integer Sign Lost Because of Unsigned Cast

This defect variation identifies *Numerical defects*, which validates if the sign information is lost while assigning a signed variable to unsigned variable; If while an unsigned variable is assigned to a signed variable, and then the sign is alternated.

## Long Lock

This defect variation identifies *Concurrency defects*, which validates the time-consuming steps between lock and unlock.

## Cross Thread Stack Access

This defect variation identifies *Stack Related defects*, which validates a thread accessing a different thread's stack

## Stack Overflow

This defect variation identifies *Stack Related defects*, which validates the size of used stack exceeding the size of the prepared stack memory.

## Stack Underrun

This defect variation identifies *Stack Related defects*, which validates a memory access lower than the bounds of a declared stack.

## Static Buffer Underrun

This defect variation identifies *Static Memory defects*, which validates a memory access to lower address allocated statically (arrays).

## Uninitialized Memory Access

This defect variation identifies *Resource Management defects*, which validates a memory access/ read of a memory location without initialization.

## Uninitialized Pointer

This defect variation identifies *Pointer Related defects*, which validates if a pointer is accessed without initialization.

## Uninitialized Variable

This defect variation identifies *Miscellaneous defects*, which validates an access to an uninitialized variable.

## Unlock Without Lock

This defect variation identifies *Concurrency defects*, which validates if resource is being unlocked prior to being locked.

## Unused Variable

This defect variation identifies *Inappropriate code*, which validates the existence of any unused variables in the code.

## Wrong Arguments Passed to a Function Pointer

This defect variation identifies *Pointer Related defects*, which validates the different number of arguments or argument types passed to the function pointer.

## Division by Zero

This defect variation identifies *Numerical defects*, which validates if an arithmetic expression is divided by zero or module operation is performed with zero.

# Bibliography

|  |  |
| --- | --- |
| [1] | P. E. Black, M. Kass, M. Koo and E. Fong, "Source Code Security Analysis Tool Functional Specification Version 1.1," National Institute of Standards and Technology, Gaithersburg, 2011. |