ISO/IEC JTC 1/SC 22/WG 23 N 0329

Revised proposed rewrite of NZN

Date	2011-03-25
Contributed by	Erhard Ploedereder
Original file name	
Notes	Replaces N0312. Also see N0328.

6.36 Ignored Error Status and Unhandled Exceptions [NZN]

6.36.1 Description of application vulnerability

Unpredicted faults and exceptional situations arise during the execution of code, preventing the intended functioning of the code. They are detected and reported by the language implementation or by explicit code written by the user. Different strategies and language constructs are used to report such errors and to take remedial action. Serious vulnerabilities arise when detected errors are reported but ignored or not properly handled.

6.36.2 Cross reference

CWE:
754: Improper Check for Unusual or Exceptional Conditions
JSF AV Rules: 115 and 208 MISRA C 2004: 16.10
MISRA C++ 2008: 15-3-2 and 19-3-1
CERT C guidelines: DCL09-C, ERR00-C, and ERR02-C

6.36.3 Mechanism of failure

The fundamental mechanism of failure is that the program does not react to a detected error or reacts inappropriately to it. Execution may continue outside the envelope provided by its specification, making additional errors or serious malfunction of the software likely. Alternatively, execution may terminate. The mechanism can be easily exploited to perform denial-of-service attacks.

The specific mechanism of failure depends on the error reporting and handling scheme provided by a language or applied idiomatically by its users.

In languages that expect routines to report errors via status variables, return codes, or thread-local error indicators, the error indications need to be checked after each call. As these frequent checks cost execution time and clutter the code immensely to deal with situations that may occur rarely, programmers are reluctant to apply the scheme systematically and consistently. Failure to check for and handle arising error condition continues execution as if the error never occurred. In most cases, this continued execution in an ill-defined program state will sooner or later fail, possibly catastrophically.

The raising and handling of exceptions was introduced into languages to address these problems. They bundle the exceptional code in exception handlers, they need not cost execution time if no error is present, and they will not allow the program to continue execution by default when an error occurs, since upon raising the exception, control of execution is automatically transferred to a handler for the exception found on the call stack. The risk and the failure mechanism is that there is no such handler (unless the language enforces restrictions that guarantees its existence), resulting in the termination of the current thread of control. Also, a handler that is found might not be geared to handle the multitude of error situations that are vectored to it. Exception handling is therefore in practice more complex for the programmer than, e.g., the use of status parameters. Furthermore, different languages provide exception-handling mechanisms that differ in details of their design, which in turn may lead to misunderstandings by the programmer.

The cause for the failure might be simply laziness or ignorance on the part of the programmer, or, more commonly, a mismatch in the expectations of where fault detection and fault recovery is to be done. Particularly when components meet that employ different fault detection and reporting strategies, the opportunity for mishandling recognized errors increases and creates vulnerabilities.

Another cause of the failure is the scant attention that many library providers pay to describe all error situations that calls on their routines might encounter and report. In this case, the caller cannot possibly react sensibly to all error situations that might arise. As yet another cause, the error information provided when the error occurs may be insufficiently complete to allow recovery from the error.

6.36.4 Applicable language characteristics

Whether supported by the language or not, error reporting and handling is idiomatically present in all languages. Of course, vulnerabilities caused by exceptions require a language that supports exceptions.

6.36.5 Avoiding the vulnerability or mitigating its effects

Given the variety of error handling mechanisms, it is difficult to provide general guidelines. However, dealing with exception handling in some languages can stress the capabilities of static analysis tools and can, in some cases, reduce the effectiveness of their analysis. Inversely, the use of error status variables can lead to confusingly complicated control structures, particularly when recovery is not possible locally. Therefore, for situations where the highest of reliability is required, the decision for or against exception handling deserves careful thought. In any case, exception-handling mechanisms should be reserved for truly unexpected situations and other situations where no local recovery is possible. Situations which are merely unusual, like the end of file condition, should be treated by explicit testing—either prior to the call which might raise the error or immediately afterward. In general, error detection, reporting, correction, and recovery should not be a late opportunistic add-on, but should be an integral part of a system design.

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Checking error return values or auxiliary status variables following a call to a subprogram is mandatory unless it can be demonstrated that the error condition is impossible.
- Equally, exceptions need to be handled by the exception handlers of an enclosing construct as close as possible to the origin of the exception but as far out as necessary to be able to deal with the error.
- For each routine, all error conditions need to be documented, and matching error detection and reporting needs to be implemented, providing sufficient information for handling the error situation.
- When execution within a particular context is abandoned due to an exception or error condition, it is important to finalize the context by closing open files, releasing resources and restoring any invariants associated with the context.
- It is often not appropriate to repair an error situation and retry the operation. It is usually a better solution to finalize and terminate the current context and retreat to a context where the fault can be handled completely.
- Error checking provided by the language, the software system, or the hardware should never be disabled in the absence of a conclusive analysis that the error condition is rendered impossible.

- Because of the complexity of error handling, careful review of all error handling mechanisms is appropriate.
- In applications with the highest requirements for reliability, defense-in-depth approaches are often appropriate, for example, checking and handling errors even if thought to be impossible.

6.36.6 Implications for standardization

In future standardization activities, the following items should be considered:

- A standardized set of mechanisms for detecting and treating error conditions should be developed so that all languages to the extent possible could use them. This does not mean that all languages should use the same mechanisms as there should be a variety, but each of the mechanisms should be standardized.