6.36 Returning Error Status or Raising Exceptions[NZN]

6.36.1 Description of application vulnerability

Unpredicted error conditions—perhaps from hardware (such as an I/O device error), perhaps from software (such as heap exhaustion)—sometimes arise during the execution of code. Programming languages provide a surprisingly wide variety of mechanisms to deal with such errors. The choice of a mechanism that doesn't match the programming language can lead to errors in the execution of the software or unexpected termination of the program. This could lead to a simple decrease in the robustness of a program or it could be exploited in a denial of service attack.

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

Even in the best-written programs, error conditions sometimes arise. Some errors occur because of defects in the software itself, but some result from external conditions in hardware, such as errors in I/O devices, or in the software system, such as exhaustion of heap space. If left untreated, the effect of the error might result in termination of the program or continuation of the program with incorrect results. To deal with the situation, designers of programming languages have equipped their languages with different mechanisms to detect and treat such errors. These mechanisms are typically intended to be used in specific programming idioms. However, the mechanisms differ among languages. A programmer expert in one language might mistakenly use an inappropriate idiom when programming in a different language with the result that some errors are left untreated, leading to termination or incorrect results. Attackers can exploit such weaknesses in denial of service attacks.

In general, languages make no distinction between dealing with programming errors (like an access to protected memory), unexpected hardware errors (like device error), expected but unusual conditions (like end of file), and even usual conditions that fail to provide the typical result (like an unsuccessful search). This description will use the term "error" to apply to all of the above. The description applies equally to error conditions that are detected via hardware mechanisms and error conditions that are detected via software during execution of a subprogram (such as an inappropriate parameter value).

6.36.4 Applicable language characteristics

Different programming languages provide remarkably different mechanisms for treating errors. In languages that provide a number of error detection and treatment mechanisms, it becomes a design issue to match the mechanism to the condition. This clause will describe the mechanisms that are provided in widely used languages.

The simplest case is the set of languages that provide no special mechanism for the notification and treatment of unusual conditions. In such languages, error conditions are signaled by the value of an auxiliary status variable, sometimes a subprogram parameter. The programming language C standard library functions use a variant of this approach; the error status is provided as the return value and sometimes in an additional global error value. Obviously, in such languages, it is imperative to check and act upon the status variable after every call to a subprogram that might provide an error indication. If error conditions can occur in an asynchronous manner, it is necessary to provide means to check for errors in a systematic and periodic manner.

Some languages permit the passing of a label parameter. If an error is encountered, the subprogram returns to the indicated label rather than to the point at which it was called. Similarly some languages accept the name of a subprogram to be used to handle errors. In either case, it is imperative to provide labeled code or a subprogram to deal with all possible error situations.

The approaches described above have the disadvantage that error checking must be provided at every call to a subprogram. This can clutter the code immensely to deal with situations that may occur rarely. For this reason, some languages provide an exception mechanism that automatically transfers control when an error is encountered. This has the potential advantage of allowing error treatment to be factored into distinct error handlers, leaving the main execution path to deal with the usual results. The disadvantages, of course, are that the language design is complicated and the programmer must deal with the conceptually more complex problem of providing error handlers that are removed from the immediate context of a specific call to a subprogram. Furthermore, different languages provide exception-handling mechanisms that differ in the manner in which various design issues are treated:

- How is the occurrence of an exception bound to a particular handler?
- What happens when no handler is local to an exception occurrence? Is the exception propagated in some manner or is it lost?
- What happens after an exception handler executes? Is control returned to the point before the call or after the call, or is the calling routine terminated in some way? If the calling routine is terminated, is there some provision for finalization, such as closing files or releasing resources?
- Are programmers permitted to define additional exceptions?
- Does the language provide default handlers for some exceptions or must the programmer explicitly provide for all of them?
- Can predefined exceptions be raised explicitly?
- Under what circumstances can error checking be disabled?

Common to all these mechanisms of error reporting is the principle mechanism of failure, namely the omission of handling the error in the right place. For status variables, checks are often omitted at the call site so that the error is ignored, execution continues despite the preceeding fault and causes further faults. For exceptions, the necessary handler may be omitted at an approriate enclosing context. While execution does not simply continue despite the fault as for an unchecked error status variable, the exception may be handled by an inappropriate handler or by none at all, leading to the unexpected termination of a program or program component.

Formatted: Bullets and Numbering

6.36.5 Avoiding the vulnerability or mitigating its effects

Given the variety of error handling mechanisms, it is difficult to write general guidelines. However, dealing with exception handlers can stress the capability of many static analysis tools and can, in some cases, reduce the effectiveness of their analysis. Therefore, for situations where the highest of reliability is required, the application should be designed so that exception handling is not used at all. In the more general case, exception-handling mechanisms should be reserved for truly unexpected situations and other situations (possibly hardware arithmetic overflow) where no other mechanism is available. Situations which are merely unusual, like end of file, should be treated by explicit testing—either prior to the call which might raise the error or immediately afterward.

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.
- In dealing with languages where untreated exceptions can be lost (for example, an exception that goes untreated within an Ada task), it is mandatory to deal with the exception in the local context before it is lost.

• When execution within a particular context is abandoned due to an exception, 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 condition and retry the operation. In such cases, one often treats a symptom but not the underlying problem. It is usually a better solution to finalize and terminate the current context and retreat to a context where the situation is known.

• 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 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 (for example, label parameters, auxiliary status variables), but each of the mechanisms should be standardized.