6.37 Fault Tolerance and Failure Strategies [REU]

6.37.1 Description of application vulnerability

Check that the current writeup works now.

AI - to Erhard to rework this vulnerability.

Expectations that a system will be dependable are based on the confidence that the system will operate as expected and not fail in normal use. The dependability of a system and its fault tolerance are determined by the component part's reliability, availability, safety and security. Reliability is the ability of a system or component to perform its required functions under stated conditions for a specified period of time [IEEE 1990 glossary]. Availability is how timely and reliable the system is to its intended users. Both of these factors matter highly in systems used for safety and security. In spite of the best intentions, systems may encounter a failure, either from internally poorly written software or external forces such as power outages/variations, floods, or other natural disasters. The reaction to a fault can affect the performance of a system and in particular, the safety and security of the system and its users.

When the software unexpectedly fails to render a requested service or terminates in an unspecified way, safety or security may be compromised. In safety-related systems the results can be catastrophic: for other systems the result can mean failure of the complete system. Failures need not necessarily cause the termination of the failing service; delivering an incorrectly computed result is a failure that, when not discovered, can have even more catastrophic consequences than a termination of the failing service.

For termination issues associated with multiple threads, multiple processors or interrupts also see ***Fehler! Verweisquelle konnte nicht gefunden werden.~~Error! Reference source not found.~~*** **Fehler! Verweisquelle konnte nicht gefunden werden.**~~6.61 Concurrency – Directed termination [CGT]~~ and **Fehler! Verweisquelle konnte nicht gefunden werden.**~~6.63 Concurrency – Premature Termination [CGS]~~***Fehler! Verweisquelle konnte nicht gefunden werden.~~Error! Reference source not found.~~***. Situations that cause an application to terminate unexpectedly or that cause an application to not terminate because of other vulnerabilities are covered in those vulnerabilities. The vulnerability at hand discusses the overall fault treatment strategy applicable to single-threaded or multi-threaded programs.

The first defense against failures is fault detection. While failures manifesting in service termination are easily detected, failures to compute correct results are more difficult to discover. Numerous checks on values can and should be made (value range, plausibility within history, reversal checks, checksums, structural checks, etc.) to establish the validity of computed results or input received. Similarly, crucial timing failures should be detected by “Watch-dog timers” and similar mechanisms that can be used to stop rogue tasks.

When a fault is detected in a component, there are many ways in which the component can react. The quickest and most noticeable way is to fail hard, also known as fail fast or fail stop. The reaction to a detected fault is then to halt the affected service (or entire system). Alternatively, the reaction to a detected fault could be to fail soft. The system would keep working with the fault present, but the performance of the system would be degraded. Systems used in a high availability environment

such as telephone switching centers, e-commerce, or other "always available" applications would likely use such a fail-soft approach, also termed “graceful degradation”. Full fault tolerance is achieved when the fault is all but indistinguishable from the normal behavior of the component, e. g. through the use of redundancy. What is actually done in a fail-soft approach can vary depending on whether the system is used for safety-critical or security-critical purposes. For fail-safe systems, such as flight controllers, traffic signals, or medical monitoring systems, there would be no effort to meet normal operational requirements, but rather to limit the damage or danger caused by the fault. A system that fails securely, such as cryptologic systems, would maintain maximum security when a fault is detected, possibly through a denial of service.

Whatever the failure or termination process, the termination of an application should not result in damage to system elements that rely upon it. Thus, it should perform “last wishes” to minimize the effects of the failure on enclosing components (e .g., release software locks) and the real world (e. g. close valves).

The reaction to a detected fault in a system can depend on the criticality of the portion in which the fault originates. When a program consists of several tasks, each task may be critical, or not. If a task is critical, it may or may not be restartable by the rest of the program as a fault handling measure. A task that detects a fault within itself but must leave the fault handling to a higher authority, should be able to halt leaving its resources available for use by the rest of the program, halt clearing away its resources, or halt the entire program. The latency of task termination and whether tasks can ignore termination signals should be clearly specified.

6.37.2 Cross reference

JSF AV Rule: 24
MISRA C 2012: 4.1
MISRA C++ 2008: 0-3-2, 15-5-2, 15-5-3, and 18-0-3
CERT C guidelines: ERR04-C, ERR06-C and ENV32-C
Ada Quality and Style Guide: 5.8 and 7.5

6.37.3 Mechanism of failure

Reasons for failures are plentiful and varied, stemming from both hard- and software. Hence the mechanisms of failure can be described only in very general terms:

- omission failures: a service is asked for but never rendered. The client might wait forever or be notified about the failure (termination) of the service.
- commission failures: a service initiates unexpected actions, e. g., communication that is unexpected by the receiver. The service might wait forever, causing omission failures for subsequent calls by clients. At a minimum, it consumes resources possibly needed by others.
- timing failures: a service is not rendered before an imposed deadline. System responses will be (too) late, causing corresponding damages to the real world affected by the system.

- Value failures: a service delivers incorrect or tainted results. The client continues computations with these corrupted values, causing a spread of consequential application errors.

Faults are the points in execution, where a failure manifests by processing going wrong. If unnoticed or unhandled, they turn into failures at the boundaries of enclosing control units or components. Failures of services are faults to their clients and, if not handled, lead to a failure of the client and consequently to faults and failures in its clients, possibly until the entire system fails. Detection and handling of faults constitutes the fault tolerance code of the system. As such, it is itself a potential source of failures. Fault-handling code is particularly difficult to design and program, since it needs to survive in an already damaged environment. Handler code is also difficult to test, since it is executed only when primary failures have occurred.

Considerable latency and processor use can arise from finalization and garbage collection caused by the termination of a task. Thus, termination must be designed carefully to avoid causing timing failures of other tasks. The termination of tasks can be maliciously used to prevent on-time performance of other active tasks.

Having inconsistent approaches to detecting and handling a fault or a lack of overall design for the fault tolerance code can potentially be a vulnerability, as faults might escape the necessary attention.

6.37.4 Applicable language characteristics

This vulnerability description is intended to be applicable to all languages.

6.37.5 Avoiding the vulnerability or mitigating its effects

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

- Decide on a strategy for fault handling. Consistency in fault handling should be the same with respect to critically similar parts.
- Use a multi-tiered approach of fault prevention, fault detection and fault reaction.
- Unambiguously describe the failure modes of each possibly failing task as fail-stop, fail-safe, fail-secure, or fail-soft as explained in 6.37.1.
- Always validate incoming data. Validate computed results at strategic points to discover value failures. See also pre- and postconditions in << reference to BLP, Liskov>>.
- Use environment- or language-provided means to stop tasks that substantially exceed deadlines.
- Always prepare for the possibility that a service does not return with a requested result in due time.
- Keep fault handling simple. If in doubt, decide for a lesser level of fault tolerance.
- In the case of continued execution, make sure that any corrupted variables of the program state have been corrected to an actual and correct or at least safe value.
- System-defined components that assist in uniformity of fault handling should be used when available. For one example, designing a "runtime constraint handler" (as described in Annex K of 9899:2012 [4]) permits the application to intercept various erroneous situations and

perform one consistent response, such as flushing a previous transaction and re-starting at the next one.

- When there are multiple tasks, a fault-handling policy should be specified whereby a task, in the absence of simple full fault tolerance or graceful degradation, may
    - Halt, and keep its resources available for other tasks (perhaps permitting restarting of the faulting task).
    - Halt, and release its resources (perhaps to allow other tasks to use the resources so freed, or to allow a recreation of the task).
    - Halt, and signal the rest of the program to likewise halt.

    <<< I consider this last advice a bit too specific to one particular model of execution. In fact, I disagreed with the original, since it excluded full fault tolerance altogether. simplify to “kill everything or do the right thing about resources” ?>>>

6.37.6 Implications for standardization

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

- Languages should consider providing a means to perform fault handling. Terminology and the means should be coordinated with other languages.