7.26 Fault Tolerance and Failure Strategies [REU]

7.26.1 Description of application vulnerability

Stephen Michell 2017-1-23 4:55 PM
Deleted: 6.XX

Check that the current writeup works now.

AI - to Erhard to rework this vulnerability to focus not on fault tolerance itself, but on vulnerabilities caused by it.

In spite of the best intentions, system components may fail, either from internally poorly written software or external forces such as power outages/variations, radiation or inadmissible user input. Systems are often designed with fault tolerance to detect and deal with such failures. Fault tolerance is itself a potential source of vulnerabilities, particularly when inappropriate or incomplete strategies are implemented.

Fault-handling code is difficult to design and program, since it needs to execute in an already damaged environment. Handler code is also difficult to test, since it is executed only when primary failures have occurred. These failures, e.g. radiation damage, may be impossible to recreate with sufficient coverage in a testing environment. Moreover, it is not easy to determine the right kind of fault tolerance for a given fault. For security, termination of the malfunctioning system may be the best action; for safety, termination may be more catastrophic than any other fault tolerance mechanism.

Reasons for failures are plentiful and varied, stemming from both hard- and software. Hence the mechanisms of primary 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. The receiver might be hindered to do its legitimate actions in time. At a minimum, resources are consumed that are 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. The mechanisms of fault tolerance are manifold, corresponding to the nature of the failure and the needs of the application, and range from recovery with subsequent normal continuation of the system (“full fault tolerance”) or restricted continuation (“graceful degradation”, “fail soft”) to termination of the system (“fail stop”, “fail safe”, “fail-secure”), possibly combined with a subsequent restart.

Arising vulnerabilities are, for example:

- The fault is not recognized and the system malfunctions or terminates as a consequence
- The fault is recognized but the damage already done is incompletely repaired, with the same consequences as in the first bullet
- A value fault is recognized too late, allowing the incorrect value to be used in the computations of other, thus corrupted, values (which, if not repaired, can cause vulnerabilities such as buffer overflows)
- The fault tolerance processing takes too long to meet timing demands
- Recovery is prevented by the cause of a permanent fault, e.g., a programming error, leading to an infinite series of recovery attempts
- The fault tolerance mechanism causes itself new faults

For vulnerabilities caused by termination issues associated with multiple threads, multiple processors or interrupts also see ***Error! Reference source not found.*** Error! Reference source not found. and Error! Reference source not found. ***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.

Stephen Michell 2017-2-10 1:18 AM
Deleted: ***Error! Reference source not found.***

Stephen Michell 2017-2-10 1:18 AM
Deleted: 6.61 Concurrency – Directed termination [CGT]

Stephen Michell 2017-2-10 1:18 AM
Deleted: 6.63 Concurrency – Premature Termination [CGS]

Stephen Michell 2017-2-10 1:18 AM
Deleted: ***Error! Reference source not found.***

Triggering known fault detection mechanisms can be used to initiate or aggravate Denial-of-Service attacks. Knowledge of a lack of fault detection, particularly of value faults, can be used to initiate system intrusions through mechanisms explained elsewhere in this document.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).

7.26.2 Cross reference

Stephen Michell 2017-1-23 4:56 PM
Deleted: xx

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

7.26.3 Mechanism of failure

Stephen Michell 2017-1-23 4:56 PM
Deleted: xx

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

- Fault tolerance code, in particular fault checking code, may interfere with the timeliness of the components to meet their deadlines
- An inappropriate fault tolerance mechanism or strategy may lead to failures in fault detection and other secondary failures

- Considerable latency and processor use can arise from finalization and garbage collection caused by the termination of a service. Thus, termination must be designed carefully to avoid causing timing failures of other services. The termination of services can be maliciously used to prevent on-time performance of other active services.
- 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.
- If faults are not detected in time and repaired completely, the following failures arise:
    - omission failures: a service is asked for but never rendered. The client might wait forever or be notified too late 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, or the actions might interfere with the regular processing going on in the meantime. At a minimum, it consumes resources possibly needed by others to meet deadlines.
    - 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. If not the client continues computations with these corrupted values, causing a spread of consequential application errors and implementation vulnerabilities caused by corrupted values as discussed elsewhere in this document.

7.26.5 Avoiding the vulnerability or mitigating its effects

Stephen Michell 2017-1-23 4:56 PM
Deleted: 6.37.4 Applicable language characteristics ... [1]

Stephen Michell 2017-1-23 4:56 PM
Deleted: 6.37

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 service.
- Check early for any faults, particularly value faults. 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.
- Validate incoming data and computed results at strategic points to discover value failures. See also pre- and postconditions in clause 6.43.
- Detect timing failures by watch-dog timers or similar mechanisms.

- Use environment- or language-provided means to stop services 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.

- Use system-defined components that assist in uniformity of fault handling when available.
- Prior to any abnormal termination of a component, perform “last wishes” to minimize the effects of the failure on enclosing components (e .g., release software locks held locally) and the real world (e. g. close valves opened by the component).
- Specify a fault-handling policy whereby a service, in the absence of full fault tolerance or graceful degradation, will halt safely and securely respectively.