# Marked up in Meeting #19. Send to Steve for rewrite.

# 6.CGM Protocol Lock Errors [CGM]

## 6.CGM.0 Terminology

Protocol: A set of rules and supporting structures for the interaction of [threads or processes][Use consistent terminology]. Note: A protocol can be tightly embedded and rely upon data in memory and hardware to control interaction of threads or can be applied to more loosely coupled arrangements, such as message communication spanning networks and computer systems.

## 6.CGM.1 Description of Application Vulnerability

Concurrent programs use protocols to control

* The way that threads interact with each other,
* How to schedule the relative rates of progress,
* How threads participate in the generation and consumption of data
* The allocation of threads to the various roles
* The preservation of data integrity, and
* The detection and correction of incorrect operations.

When protocols are incorrect, or when a vulnerability lets an exploit destroy a protocol, then the concurrent portions fail to work co-operatively and the system behaves incorrectly.

This vulnerability is related to [CGX] Shared Data Access and Corruption, which discusses situations where the protocol to control access to resources is explicitly visible to the participating partners and makes use of visible shared resources. In comparison, this vulnerability discusses scenarios where such resources are protected by protocols, and considers ways that the protocol itself may be misused.

## 6.CGM.2 Cross References

C.A.R Hoare, A model for communicating sequential processes, 1980

Larsen, K.G., Petterssen, P, Wang, Y, UPPAAL in a nutshell, 1997

Lundqvist, K and Asplund, L., “A Formal Model of a Run-Time Kernel for Ravenscar”, The 6th International Conference on Real-Time Computing Systems and Applications – RTCSA 1999,

CWE 833 Deadlock

CWE 413 Improper Resource Locking

CWE 414 Missing Lock Check

CWE 609 Double Checked Locking

CWE 667 Improper Locking

CWE 821 Incorrect Synchronization

CWE 833 Deadlock

## 6.CGM.3 Mechanism of Failure

Threads use locks and protocols to schedule their work, control access to resources, exchange data, and to effect communication with each other. Protocol errors occur when the expected rules for co-operation are not followed, or when the order of lock acquisitions and release causes the threads to quit working together. These errors can be as a result of:

* deliberate termination of one or more threads participating in the protocol,
* disruption of messages or interactions in the protocol,
* errors or exceptions raised in threads participating in the protocol, or
* errors in the programming of one or more threads participating in the protocol.

In such situations, there are a number of possible consequences. One common consequence is “deadlock”, where every thread eventually quits computing as it waits for results from another thread. A second common consequence is “livelock”, where one or more threads commandeer all of the computing resource and effectively lock out the other portions. In each case, no further progress in the system is made. A third common consequence is that data may be corrupted or lack currency (timeliness). A fourth common consequence is that one or more threads detect an error associated with the protocol and terminate prematurely, leaving the protocol in an unrecoverable state.

When static verification is not possible, the detection and recovery from protocol errors is necessary. Watchdog timers (in hardware or in an extremely high priority thread), or threads that monitor progress can be used to detect deadlock or livelock conditions and can be used to restart the system. Such recovery techniques also constitute a protocol, but the extreme simplicity of a detection and recovery protocol can usually be verified. [The preceding two paragraphs should be reduced to bullets and placed in .5.]

The potential damage from attacks on protocols depends upon the nature of the system using the protocol and the protocol itself. Self-contained systems using private protocols can be disrupted, but it is highly unlikely that predetermined executions (including arbitrary code execution) can be obtained. On the other extreme threads communicating openly between systems using well-documented protocols can be disrupted in any arbitrary fashion with effects such as the destruction of system resources (such as a database), the generation of wrong but plausible data, or arbitrary code execution. In fact, many documented client-server based attacks consist of some abuse of a protocol such as SQL transactions.

## 6.CGM.4 Applicable Language Characteristics

The vulnerability is intended to be applicable to languages with the following characteristics:

* Languages that support concurrency directly.
* Languages that permit calls to operating system primitives to obtain concurrent behaviours.
* Languages that permit IO or other interaction with external devices or services.
* Languages that support interrupt handling directly or indirectly (via the operating system).

## 6.CGM.5 Avoiding the Vulnerability or Mitigating its Effects

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

* Consider the use of synchronous protocols, such as defined by CSP, Petri Nets or by the simple Ada rendezvous protocol since these can be statically shown to be free from protocol errors such as deadlock and livelock
* Consider the use of simple asynchronous protocols that exclusively use concurrent threads and protected regions , such as defined by the Ravenscar Tasking Profile, that can also be shown statically to have correct behaviour using model checking technologies, as shown by [LA 1999]
* When static verification is not possible, consider the use of detection and recovery techniques using simple mechanisms and protocols that can be verified independently from the main concurrency environment. Watchdog timers coupled with checkpoints constitute one such approach.
* Use high level synchronization paradigms, for example monitors, rendezvous, or critical regions.
* Design the architecture of the application to ensure that some threads or tasks never block, and can be available for detection of concurrency error conditions and for recovery initiation.
* Use model checkers to model the concurrent behaviour of the complete application and check for states where progress fails. Place all locks and releases in the same subprograms, and ensure that the order of calls and releases of multiple locks are correct.

## 6.CGM.6 Implications for Standardization

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

* Raise the level of abstraction for concurrency services.
* Provide services or mechanisms to detect and recover from protocol lock failures.
* Design concurrency services that help to avoid typical failures such as deadlock.