6.CGM Protocol Lock Errors [CGM]

6.CGX.0 Terminology

Thread: A piece of code that can execute independently in time and memory space from other parts of the code. A Thread may be a separate program, an asynchronous artificat such as an interrupt, or a creation from the operating system or the language runtime.

6.CGM.1 Description of Application Vulnerability

All concurrent programs, whether they are co-operating processes across networks, parallel threads, or a main program with event handlers or interrupt handlers, must use protocols to control

- the way that threads interact with other,
- how the schedule the relative rates of progress,
- how they participate in the generation and consumption and 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., Asplund, L., and Michell, S., “A Formal Model of the Ada Ravenscar Tasking Profile; Protected Objects”, In Proc. Reliable Software Technologies, Ada-Europe 1999

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,

6.CGM.3 Mechanism of Failure

All threads must 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 the expected rules for co-operation are not followed, or when the order of lock acquisitions and release causes the coroutines to quit working together. It is possible for programming errors to not consider all aspects of the the protocol or for an agent to send or set erroneous protocol states to some of the participants.

In such situations, there are a number of possible consequences. One common consequence is “deadlock”, where every coroutine eventually quits computing as it waits for results from another coroutine. A second common consequence is “livelock”, where one coroutine or portion of the system commandeers all of the computing resource and effectively locks out the other portions. In each case, no further progress in the system is made. A third common consequence is that portions of the system keep executing but produce erroneous data, or cause erroneous behaviour. A fourth common consequence is that one or more threads detect an error associated with the protocol and terminate prematurely, leaving the protocol in unrecoverable state.

Concurrent protocols are the most difficult for humans to correctly design and implement. Completely synchronous protocols, such as defined by CSP, by Petrinets or by the simple Ada rendezvous protocol can be statically shown to be free from protocol errors such as deadlock and livelock, and considerable progress has been made to show that the complete

system (data and concurrency protocols) is correct. Simple asynchronous protocols that exclusively use concurrent threads and protected regions can also be shown statically to have correct behaviour using model checking technologies, as shown by [LA 2000]. More complex and asynchronous protocols cannot be shown statically to have correct behaviours. Dynamic verification (testing) of all protocols can show that the system generally behaves correctly, but cannot show that it is free of such vulnerabilities.

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 coroutine), or coroutines 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 potential damage from attacks on protocols depends upon the nature of the 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. In fact, many 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 the application to communicate with other applications (on the same system or networked) and use blocking services such as file IO.

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

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

- Use high level paradigms such as CSP, Ada's rendezvous model, Java's Protected or Ada's protected objects. Careful design of the architecture of the application should be done to ensure that some threads or tasks never block, and can be available for detection of concurrency conditions and to initiate recovery;
- 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 cannot be reversed.
- Use techniques such as digital signing (effective use of encryption and hashing) to protect data being exchanged using open protocols.
- Implement simple detection and recovery protocols when verification of the complete protocol is not possible.

6.CGM.6 Implications for Standardization

In future standardisation 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 failures such as deadlock.