## 6.CGX Concurrent Data Access [CGX]

## 6.CGX.0 Terminology

## 6.CGX.1 Description of Application Vulnerability

Concurrency presents a significant challenge to program correctly, and has a large number of possible ways for failures to occur, quite a few known attack vectors, and many possible but undiscovered attack vectors. In particular, data visible from more than one thread and not protected by a sequential access lock can be corrupted by out-of-order accesses. This, in turn, can lead to incorrect computation, premature program termination, livelock, or system corruption.

## 6.CGX.2 Cross References

ISO IEC 8692 Programming Language Ada, with TC 1:2001 and AM 1:2007.

Burns A. and Wellings A., Language Vulnerabilities - Let’s not forget Concurrency, IRTAW 14, 2009.

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

CWE 214 Information Exposure Through Process Environment

CWE 362 Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

CWE 366 Race Condition Within a Thread

CWE 368 Context Switching Race Conditions

CWE 413 Improper Resource Locking

CWE 764 Multiple Locks of a Critical Resource

CWE 765 Multiple Unlocks of a Critical Resource

CWE 821 Missing Synchronization

CWE 821 Incorrect Synchronization

## 6.CGX.3 Mechanism of Failure

Making data visible to multiple threads causes the data to be shared. Shared data can be monitored or updated directly by more than one thread, possibly circumventing any access lock protocol in operation. Some concurrent programs do not use access lock mechanisms but rely upon other mechanisms such as timing or other program state to determine if shared data can be read or updated by a thread. Regardless, direct visibility to shared data permits direct access to such data concurrently. Arbitrary behaviour of any kind can result.

### 6.CGX.4 Applicable Language Characteristics

The vulnerability is intended to be applicable to

* all languages with code executing concurrently and sharing data.

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

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

* Place all data in memory regions accessible to only one thread at a time.
* Use languages and those language features that provide a robust sequential protection paradigm to protect against data corruption. For example, Ada's protected objects and Java's Protected class, provide a safe paradigm when accessing objects that are exclusive to a single program.
* Use operating system primitives, such as the POSIX locking primitives for synchronization to develop a protocol equivalent to the Ada “protected” and Java “Protected” paradigm.
* Where order of access is important for correctness, implement blocking and releasing paradigms, or provide a test in the same protected region to check for correct order and generate errors if the test fails. For example, the following structure in Ada could be used to implement an enforced order.

package buffer\_pkg is

  protected Buffer is

     entry Read (Data :    out Data\_Type);

     entry Write (Data : in    Data\_Type);

  private

Buffer: …   ; -- Data needing sequential access

… and then, in the body of the package…

     entry Read (Data :    out Data\_Type) when not empty(Buffer) is … ;

-- Read waits until the buffer is not empty.

     entry Write (Data : in    Data\_Type) when not full(Buffer) is … ;

-- Write waits until the buffer is not full.

  end Buffer;

end Buffer\_Pkg

In the above example, the writer must block until there is room to write a new record, and readers must block if there are no records available.

## 6.CGX.6 Implications for Standardization

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

* Languages that do not presently consider concurrency should consider creating primitives that let applications specify regions of sequential access to data. Mechanisms such as protected regions, Hoare monitors or synchronous message passing between threads result in significantly fewer resource access mistakes in a program.
* Provide the possibility of selecting alternative concurrency models that support static analysis, such as one of the models that are known to have safe properties. For examples, see [1] and [3]. {Editor: [1] and [2] are bibliography references to Ada (for Ravenscar)\_ and to CSP.}