## 6.CGX Concurrency 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, any resource that is visible from more than one thread and is not protected by a sequential access lock can be corrupted by out-of-order accesses. This corruption can lead to resource corruption, 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

All data that is openly visible to multiple threads is shared data and is open to being monitored or updated directly by a thread, whether or not that data has an 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 that data concurrently. This can permit the following errors:

* Lost data or control signals by multiple updates by one thread without corresponding reads by another thread;
* Simultaneous updates of different portions of the data by different threads, resulting in corruption of the data;
* Simultaneous updates of different portions of the data by different threads, resulting in wrong data being passed;
* Missing or corrupt data;
* Precisely written (but wrong) data that changes the behaviour of the program to undertake new objectives;
* Livelock when necessary data is missed or never correctly read.

The above scenarios usually result in corruption, livelock, or corrupted applications. Results such as arbitrary code execution are usually not achievable because threads are programmed and built into the same application, but when combined with other attacks and vulnerabilities, arbitrary code execution may be possible.

### 6.CGX.4 Applicable Language Characteristics

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

* Languages that provide explicit concurrency in the language, such as tasks, threads, co-routines, and potentially share data between threads.
* Languages that provide explicit concurrency and protected regions of sequential access and explicit concurrency control mechanism, such as suspend(thread), block(thread), resume(threads), enable(interrupt), and disable(interrupt)

### 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 or co-routine at a time.
* Use languages and those language features that provide a complete sequential protection paradigm to protect against data corruption. 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 pthread\_xxx primitives for locking and 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 would implement an enforced order.

package buffer\_pkg is

protected Buffer is

entry Read (Data : out Data\_Type);

entry Write (Data : in Data\_Type);

private

Data\_Needing\_Sequential\_Access: ...

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 coroutines 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.}