6.CGX Concurrency Data Access and Corruption [CGX]

6.CGX.0 Terminology

Stateless Protocol: A communication or cooperation between threads where no state is preserved in the protocol itself (example HTTP or direct access to a shared resource). Since most interaction between threads require that state be preserved, the cooperating threads must use values of the resources(s) themselves or add additional communication exchanges to maintain state.

Stateless protocols require that the application provide explicit resource protection and locking mechanisms to guarantee the correct creation, view, access to, modification of, and destruction of the resource – i.e. the state needed for correct handling of the resource).

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 artifact such as an interrupt, or a creation from the operating system or the language runtime.

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, system corruption, or arbitrary execution of code.

Many vulnerabilities that are associated with concurrent access to files, shared memory or shared network resources fall under this vulnerability, including resources accessed via stateless protocols such as HTTP and remote file protocols.

Explicit mechanisms require that the resource be visible and available to all threads that might wish access to the resource. A rogue coroutine therefore has direct access to

- Monitor the resource to extract information from the resource, or to extract information about how the resource is used
- Monitor the resource to determine the appropriate time take corrosive action
- Monitor locking mechanisms to extract information about how the resource or resource locks are used
- Access the resource without participating in cooperative locking mechanisms
- Modify the resource without participating in cooperative locking mechanisms to corrupt the resource
- Modify the resource (without participating in cooperative locking mechanisms) at times determined to be safe to effect a full update to pass preplanned wrong values to other threads.
- Modify the resource using the cooperative locking mechanisms to update the resource to present wrong information to other threads, to cause uncontrolled errors in other threads, up to and including arbitrary execution.
- Modify, access and restore a resource in ways that are undetected by the application but meaningful to attacking threads working in cooperation.

6.CGX.2 Cross References

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

6.CGX.3 Mechanism of Failure

Any time that a shared resource is open to inspection by a thread, the resource can be monitored to determine usage patterns, timing patterns, and access patterns to determine ways that a planned attack can succeed. Such monitoring could be, but are not limited to:

- Read resource values to obtain information of value to the applications

- Monitoring access time and access thread to determine when a resource can be accessed undetected by other threads (for example, Time-of-Check-Time-Of-Use attacks rely upon a determinable amount of time passage between the check on a resource and the use of the resource when the resource could be modified to bypass the check);
- Monitor a resource and modification patterns to help determine the protocols in use.
- Monitor access times and patterns to determine quiet times in the access to a resource that could be used to find successful attack vectors

Any time that a shared resource is open to shared update by a thread, the resource can be changed to determine

- how such changes affect usage patterns, timing patterns, access patterns to determine ways that a planned attack can succeed.
- Determine how well application threads detect and respond to changed values

Any time that a shared resource is open to shared update by a thread, the resource can be changed in ways to further an attack once it is initiated.

- Add the attacker threads to sets of approved participants

Any of the above occurrence can result in inconsistent, wrong, undecipherable or corrupted values, and can cause one or more of the threads to fail in unpredictable ways. Such errors rely upon the individual speeds and timing of the coroutines in question, the relative speed and rate of access to the resource in question, and other loading factors from portions of the system. Consequently it is difficult, to convert such erroneous accesses into arbitrary code execution attacks, but denial of service and loss of confidence attacks are very possible. When coupled with other vulnerabilities, however, the arbitrary code execution becomes 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 mechanisms to create shared data areas that may be accessed from outside the executing program that created the object. These objects can include shared memory allocated by OS calls, or files opened in shared mode.
- 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 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 and Java “protected” paradigm, as discussed below.
- Use a data access paradigm where a sequential lock is acquired, the object is accessed and the lock released within a single subprogram and in a single path in the subprogram.
- 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
         Buf_Data : Data_Type;
      end Buffer;
   end Buffer_Pkg.
```

In this case, 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.