Discussion 24 May 2021 on this topic. We will attempt to write this up for clause 7, application vulnerabilities.

Priority Inversion  
  
.1 Description of application vulnerability  
  
Priorities are used to schedule program resources such as threads, physical devices and  
critical data. Higher priority tasks that are ready to execute will be given precedence over lower  
priority tasks by the scheduler. It is a fundamental mistake in a priority-based system if a lower priority thread is executing while a higher priority thread is ready to execute but cannot. This effect is called a priority inversion.

When accessing resources that are shared between a high priority thread and a low priority thread, there are times when the lower priority thread is accessing the resource and the higher priority thread is also ready to execute, but cannot. This is only a priority inversion if the higher priority thread was ready to execute before the lower priority thread accessed the shared resource; the lower priority thread effectively borrows the higher priority to access the shared resource.

.2 Cross references  
  
https://en.wikipedia.org/wiki/Priority\_inversion  
  
Lui Sha, Rangunathan Rajkumar, John P Lehoczky, "Priority Inheritance  
Protocols: An Approach to Real-Time Synchronization"  
IEEE Transactions on Computers, September 1990,  
https://doi.org/10.1109/12.57058  
  
[Ada2012] ISO/IEC 8652:2012, Information technology - Programming  
languages - Ada  
  
Burns, Wellings, ??? (check this)  
  
...  
  
.3 Mechanism of failure  
  
Consider three tasks H, M and L, of high, medium and low priority respectively executing on a system with a single processor. Here assume h = priority(H) > m = priority(M) > l = priority(L). H and L share access to a shared resource R while M executes unrelated code that can consume significant time.

In a poorly designed system, R is guarded with a lock that blocks a thread attempting to access R if another thread is already accessing R, for example L access r and H blocks until L releases its lock. We further assume that the priority of L remains at l while it accesses R. In this scenario, if M becomes ready to execute, m > l and M executes. H is therefore blocked until M completes and l completes and releases R.

In an inversion-free system, the priority of L is raised to h while it accesses R. In this scenario, L completes accessing R, releases H to access R and drops its priority to L. H and M are ready to execute, but h > m, so H executes and no priority inversion occurs.

Another scenario of priority inversion can occur if the routines that access R are longer than H can afford to wait for L to access R.

On multiprocessor systems, the simple priority boosting scenario described above is insufficient to prevent priority inversion.

either of which can acquire exclusive use of a shared resource R.  
If H attempts to acquire R after L has acquired it, then H becomes blocked until L relinquishes the resource. Sharing an exclusive-use resource (R in this case) in a well-designed system typically involves L relinquishing R promptly so that H (a higher priority task) does not stay blocked for excessive periods of time.  
It is, however, possible that a third task M of medium priority (p(L) <  
p(M) < p(H), where p(x) represents the priority for task (x)) becomes  
runnable during L's use of R.  
At this point, M being higher in priority than L, preempts L (since M  
does not depend on R), causing L to not be able to relinquish R  
promptly, in turn causing H - the highest priority process - to be  
unable to run  
(that is, H suffers unexpected blockage indirectly caused by lower  
priority tasks like M).  
  
.4 Applicable language characteristics  
  
The vulnerability is intended to be applicable to languages with the  
following characteristics:  
    • Languages that support concurrency and priorities directly.  
    • Languages that permit calls to operating system primitives to  
obtain concurrent behaviours and priorites.  
    • Languages that support interrupt handling directly or indirectly  
(via the operating system).  
  
.5 Avoiding the vulnerability or mitigating its effect  
  
Software developers can avoid the vulnerability or mitigate its effects  
in the following ways:  
    • Consider the use of synchronous protocols, such as defined by Ada,  
that support some sort of priority inheritance [Ada2012].  
    • Consider the use of external tools to prove that priority  
inversion does not occur.  
  
.6 Implications for language design and evolution  
  
In future language design and evolution activities, the following items  
should be considered:  
    • If priorities are supported by the scheduler, ensure that a scheme  
like priority inheritance is supported by the language.  
    • Providing services or mechanisms to detect and recover from  
priority inversion.