ISO/IEC JTC 1/SC 22/OWGV N 0108

Date: 28 November 2007
Contributed by C H Pygott
Original filename "Proposed additions v2.doc"

Proposed additions to ISO/IEC PDTR 24772

C H Pygott, 15/11/2007 – revised after OWGV meeting Birmingham 16/11/2007

Proposed addition to 1.4 (under intended audience)

As described in the following paragraphs, developers of applications that have clear safety, security or mission criticality are usually aware of the risks associated with their code and can be expected to use this document to ensure that *all* relevant aspects of their development language have been controlled.

That should not be taken to mean that other developers can ignore this document. A flaw in an application that of itself has no direct criticality may provide the route by which an attacker gains control of a system or may otherwise disrupt co-located applications that are safety, security or mission critical.

It would be hoped that such developers would use this document to ensure that common vulnerabilities are removed from all applications.

New sub-section in section 6: Programming Language Vulnerabilities (assumed 6.17)

6.17 XYZ Order of evaluation and initialization

6.17.0 Status and history

6.17.1 Description of application vulnerability

The order of evaluation of sub-expressions within an expression, and the order of initialisation of variables across a project are often not fully specified.

This vulnerability most often manifests when code that is working due to happenstance is recompiled for use in a different environment, where the order of evaluation or initialisation changes, changing the behaviour of the code. It is particularly relevant if a 'track record' argument is used to incorporate previously developed and 'trusted' code into an application with a reduced level of testing.

However, where order of evaluation/initialisation is unspecified, there is no guarantee that behaviour will even be consistent for multiple executions of the same binary image. The order may be dependent on data values, position in memory, or any other 'environmental' factor.

This problem may also be observed where code is developed in a host/target environment, and the host and target implement these unspecified features differently.

6.17.2 Cross reference

6.17.3 Categorization

6.17.4 Mechanisms of failure

Behaviour may be unspecified where an expression modifies the state of more than one variable (or the same variable multiple times), and one of the modified variables is used more than once in the expression. For example in C:

```
array[i] = ++i;
```

which value of i is used to index the array? its initial or incremented value?

This vulnerability also arises if the modification and use of variables occurs indirectly, through functions called by the expression. Again for C, if Pop() returns the value from the top of a stack and removes that value from the stack, it is unspecified whether:

```
x = Pop() – Pop()
```

evaluates the original top of the stack minus the next member, or the second member of the stack minus the top.

Note that the use of brackets does not remove this unspecified behaviour. The use of brackets in:

```
A – B – C
```

distinguishes between (A – B) – C and A – (B – C) but the three sub-expressions, A, B and C can still be evaluated in any order.

The order of evaluation of parameters to a function may similarly be unspecified.

For initialisation, the vulnerability arises if initialisation of one variable depends on the value of another that may or may not have been initialised. For example in C++, the order of initialisation of global variables within a translation unit is specified, but not the order in which translation units are initialised, so if the initialisation of variable A in one translation unit depends on the value of B in another, then B may either have its default initialisation value (usually 0) or its fully initialised value when used to initialise A.

6.17.5 Range of language characteristics considered

The order of evaluation of sub-expression within an expression, where the expression may cause multiple side-effects

The order of evaluation of parameters to a function, where the evaluation of one parameter may modify a variable that another parameter depends upon

The order of initialisation of variables, where the initialisation of one depends on the value of another than may or may not have been initialised before use.

6.17.6 Avoiding the vulnerability or mitigating its effect

Code should be developed with the requirement that it produces consistent results under all evaluation and initialisation orders that the language definition allows. This may then be enforced, for example, by disallowing expressions with multiple side-effects.

6.17.7 Implications for standard

6.17.8 Bibliography

New sub-section in section 6: Programming Language Vulnerabilities (assumed 6.18) - or should this be in section 7 – Application Vulnerabilities?

6.18 XYZ Dead code and unspecified functionality

6.18.0 Status and history

6.18.1 Description of application vulnerability

Dead code is code that exists in an application, but which can never be executed, either because there is no call path to the code (e.g. a function that is never called) or because the execution path to the code is semantically infeasible, e.g. in

```
if (true) A; else B;
```

B is dead code

The presence of dead code is not in itself an error, but begs the question why is it there? Is its presence an indication that the developer believed it to be necessary, but some error means it will never be executed? Or is there a legitimate reason for its presence, for example:

- as defensive code, only executed as the result of a hardware failure
- as part of a library not required in this application
- as diagnostic code not executed in the operational environment

Such code may be referred to as "deactivated"

'Unspecified functionality' is code that may be executed, but whose behaviour does not contribute to the requirements of the application. In security-critical applications particularly, does the presence of unspecified functionality represent the possibility of the developer having included a 'trap-door' that could allow illegitimate access to the system on which it is eventually executed?

6.18.2 Cross reference

DO178B definitions of Dead and Deactivated code

6.18.3 Categorization

6.18.4 Mechanisms of failure

As described in 6.18.1, the presence of dead code or unspecified functionality potentially indicates an error or malicious attempt to subvert the intent of the application.

There is a secondary consideration for dead code in languages that permit overloading of functions etc. and use complex name resolution strategies. The developer may believe that some code is legitimately dead, but its existence in the program means that it appears in the namespace, and may be selected as the best match for some use that was intended to be of an overloading function. That is, although the developer believes it is never going to be used, in practice it is used in preference to the intended function.

6.18.5 Range of language characteristics considered

Code that exists in an application that can never be executed.

Code that exists in an application that was not expected to be executed, but is.

Code that exists in an application that can be executed, but whose behaviour does not contribute to the requirements of the program.

6.18.6 Avoiding the vulnerability or mitigating its effect

As a first resort, the developer should endeavour to remove, as far as is practical, dead code from an application.

Notwithstanding that, the developer should identify any dead code in the application, and provide a justification (if only to themselves) as to why it is there. At the same time, the developer should ensure that any code that was expected to be unused is actually recognised as dead.

In general a developer should be able to show that there is no unspecified functionality in their application, or that any unspecified functionality is there for a legitimate reason (e.g. diagnostics required for developer maintenance or enhancement). The customer for bespoke security-critical code in particular, should ask to see such traceability as part of their acceptance of the application

6.18.7 Implications for standard

6.18.8 Bibliography