ISO/IEC JTC 1/SC 22/WG 23 N 0290

Proposed revision of "6.26 Dead and Deactivated Code [XYQ]"

Date	13 December 2010
Contributed by	David Keaton
Original file name	
Notes	

6.26 Dead and Deactivated Code [XYQ]

6.26.1 Description of application vulnerability

Dead and Deactivated code (the distinction is addressed in ~~6.35~~XYQ.3) is code that exists in the executable, but which can never be executed, either because there is no call path that leads to it (for example, a function that is never called), or the path is semantically infeasible (for example, its execution depends on the state of a conditional that can never be achieved).

Dead and Deactivated code ~~is~~may be undesirable because it ~~indicates~~may indicate the possibility of a coding error and because it may provide a "jump" target for an intrusion.

Also covered in this vulnerability is code which is believed to be dead, but which is inadvertently executed.

6.26.2 Cross reference

CWE:
- 561. Dead Code
- 570. Expression is Always False
- 571. Expression is Always True

JSF AV Rules: 127 and 186
MISRA C 2004: 2.4 and 14.1
MISRA C++ 2008: 0-1-1 to 0-1-10, 2-7-2, and 2-7-3
CERT C guidelines: MSC07-C and MSC12-C
DO-178B/C

6.26.3 Mechanism of failure

DO-178B defines Dead and Deactivated code as:

- Dead code – Executable object code (or data) which... cannot be executed (code) or used (data) in an operational configuration of the target computer environment and is not traceable to a system or software requirement.
- Deactivated code – Executable object code (or data) which by design is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component, or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options.

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 (for example, a function that is never called) or because the execution path to the code is semantically infeasible, as in

```
integer i = 0;
if ( i == 0)
 then fun_a();
 else fun_b();
```

`fun_b` is dead code, as only `fun_a` can ever be executed.

The presence of dead code is not in itself an error~~, but begs the question why is it there?. Is it~~Its presence may be an indication that the developer believed it to be necessary, but some

error means it will never be executed?. Or is there aThere may also be legitimate reasons for its presence, for example:

- Defensive code, only executed as the result of a hardware failure.
- Code that is part of a library not required in this application.
- Diagnostic code not executed in the operational environment.
- Code that is temporarily deactivated but may be needed soon. This may occur as a way to make sure the code is still accepted by the language translator to reduce opportunities for errors when it is reactivated.
- Code that is made available so that it can be executed manually via a debugger.

Such code may be referred to as “deactivated”. That is, dead code that is there by intent.

There is a secondary consideration for dead code in languages that permit overloading of functions and other constructs that use complex name resolution strategies. The developer may believe that some code is not going to be used (deactivated), 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 towill never be used, in practice it is used in preference to the intended function.

6.26.4 Applicable language characteristics

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

- Languages that allow code to exist in the executable that can never be executed.

6.26.5 Avoiding the vulnerability or mitigating its effects

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

- The developer should endeavour to remove, as a first resort and as far as practical, dead code from an application unless its presence serves a purpose.
- When a developer identifies code that is dead because a conditional always evaluates to the same value, this could be indicative of an earlier bug and additional testing may be needed to ascertain why the same value is occurring.
- The developer should identify any dead code in the application, and provide a justification (if only to themselves) as to why it is there.
- The developer should also ensure that any code that was expected to be unused is actually recognized as dead code.
- The developer should apply standard branch coverage measurement tools and ensure by 100% coverage that all branches are neither dead nor deactivated

6.26.6 Implications for standardization

[None]