6.<*> Modifying Constants [UJO]

6. <*>.1 Description of application vulnerability

Many programming languages allow the user to specify some declared entity to be "constant". However, some of these languages allow alteration of the value of this entity in some cases after all. The semantics then range from legitimate and deterministic behavior to implementation-defined or undefined behavior. Often, the alteration are performed by means of indirection.

6. <*>.2 Cross reference

CWE: <<none? I did not find any, but lots of "make const"-advice>>
CERT C guidelines: DCL52-CPP , EXP 40-C, EXP55-CPP <<incomplete?>>

MISRA C++: 7-1-1, 9-3-3 <<<somebody please check; not avail. to me>>>

6. <*>.3 Mechanism of failure

In code reviews and manual code inspections, users tend to rely on the belief that an entity declared to be constant does not change its value during the execution of the program (regardless of the exact semantics of the language). The initializing value is taken to be its value throughout the execution. For example, the upper bound of a ring buffer array might be declared as a constant. If, however, the value can be changed during the execution, the belief in immutability can be falsified. In the example, after changing the upper-bound constant, insufficiently large buffer allocations or out-of-bounds buffer accesses, seemingly checked against the "constant" upper bound, may occur.

Even the well-meant alteration of constants is very risky if the language permits optimizations based on the known initial value of the constant entity. The optimization "constant propagation" may replace uses of the constant by its initializing value. The alteration of the value at run-time then has no effect on this use of the constant, while it changes other uses of the constant where constant propagation did not take place. Moreover, different compilers or even the same compiler under different switch setting can optimize different uses of the constant differently, leading to non-deterministic executions that often result in dangerous malfunctions.

The vulnerability can be exploited if the modification of constants is known to the attacker and the code that modifies the constant can be triggered by the attacker.

The vulnerability may be difficult to detect if levels of indirection are involved in the modification of the constant.

6. <*>.4 Applicable language characteristics

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

- Languages that allow the specification of an entity to be “constant” and, at the same time, legitimize or tolerate changes of its value.

6. <*>.5 Avoiding the vulnerability or mitigating its effects

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

- Do not change the value of entities declared to be constant.
- Do not create references or pointers to entities declared to be constant. This includes passing constants as actual parameters by reference, unless immutability of the formal parameter is ensured.
- Use static analysis tools that detect the alteration of constant entities.

6. <*>.6 Implications for language design and evolution

In future language design and evolution activities, the following items should be considered:

- Avoid language constructs that allow the modification of constant entities.
- Ensure that the property to be immutable cannot be changed by language operations such as assignment or conversion.