ISO IEC JTC 1 SC 22 WG 23 N0614

9 January 2016

The following 3 vulnerabilities are contributed for consideration as additions to TR24772-1.

6.xx Redispatching [PPH]

6.xx.1 Description of application vulnerability

When very similar functionality is provided by methods or interfaces with varying parameter structures, a frequently found implementation strategy is to designate one of them as the "work horse" and have all others call on it to perform the (common) work. A prime example are constructor or initialization methods where different sets of initial values for certain components are provided and the remaining components are set to default values.

When the semantics of inner calls of dispatching methods ask for dispatching in turn, the call is said to be "redispatching". In this case, the following scenario can evolve: In class C, the implementation of method A dispatches to method B, the work horse. In a derived class CD, the implementation of B needs to be changed. The programmer finds the signature of the inherited method A matching his needs and calls A as part of the redefinition of B. The outcome of a previously correct dispatching call on B in C for a polymorphic variable of class C holding a reference to an object of class CD now causes infinite recursion between the redefined method B and the inherited method A of class CD.

6.xx.2 Cross reference

CWE: << TBD >>
JSF AV Rules: <<<TBD>>>
CERT C guidelines: <<TBD>>
Ada Quality and Style Guide: <<TBD>>

6.xx.3 Mechanism of failure

The mechanism is the intrinsic call semantics of the language. If it demands dispatching for nested method calls, the failure scenario is guaranteed. While the example above is tractable, the infinite recursion can involve multiple objects along a reference chain and, thus, it becomes quickly undecidable whether such a situation exists or not. Even for simple cases, avoidance requires knowledge about the implementation of all called methods inherited from superclasses and needs to apply this knowledge transitively. Such a requirement is diametrically opposed to fundamental software engineering axioms.

Malicious exploit of the vulnerability adds a subclass that contains this infinite recursion conditionally on some trigger value. The recursion can be sufficiently obscured so that no analysis tool or reviewer can detect it with any certainty. The system can then be caused to fault with a stack overflow anytime this trigger is used. The vulnerability can thus be used for Denial-of-Service attacks.

It has been shown that the released Java SDK contained more than 10 (about 30?) of such infinite recursions. << EP:I can provide a citation and precise numbers, if the sentence is to stay, but need to research a little to find it.>>

6.xx.4 Applicable language characteristics

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

- Languages that demand or allow dispatching for calls within dispatching operations.

6.xx.5 Avoiding the vulnerability or mitigating its effects

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

- Avoid dispatching calls in methods, although this is a highly controversial advice in stark conflict with the principles of object orientation. See upcast consequences in subclause [BKK].
- Agree on and document a redispatch hierarchy within groups of affected methods, e.g., initializers or constructors, and use it consistently throughout the class hierarchy.

6.xx.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Find a solution to the problem.

6.yy Deep vs. Shallow Copying [YAN]

6.yy.1 Description of application vulnerability

When structures containing references as data components are copied, it is of high importance to decide whether the references are to be copied (shallow copy) or, instead, the objects designated by the references are to be copied and a reference to the newly created object used as the component value of the copied structure (deep copy). Almost all languages define structure-copying operations as shallow copies, i.e., the copied structure contains the same reference which designates the same object. Deep copying is algorithmically more challenging and may be very expensive in time and memory consumption. If, however, a shallow copy is made where a deep copy was needed, serious aliasing problems can arise in the objects that are part of the graphs spanned by the copied references. Subsequent modification of such an object are visible via both the old and the new structure.

An identical problem arises when array indices are stored as component values (in lieu of pointers or references) and used to access objects in an array outside the copied data structure.

6.yy.2 Cross reference

CWE: << TBD >>
JSF AV Rule 76, 77, 80
CERT C guidelines: <<TBD>>
Ada Quality and Style Guide: <<TBD>>

6.yy.3 Mechanism of failure

Problems with shallow copying arise when values in the objects (transitively) referenced by the original or the copy are assigned to: in a deep copy, such assignments affect only the original or the copy of the graph, respectively; in a shallow copy, the value of the object is changed in both graphs, which may not have been the intention of the programmer. Consequently, the problem may manifest itself only during maintenance when, for the first time, such as assignment to a contained object is introduced, while shallow copying was originally chosen for reasons of efficiency but relying on the absence of assignments.

Knowledge of the use of shallow copying in lieu of deep copying can be exploited in attacks by causing unintended changes in data structures via the described aliasing effect.

6.yy.4 Applicable language characteristics

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

- Languages that have pointers or references as part of composite data structures.
- All languages that support arrays.

6.yy.5 Avoiding the vulnerability or mitigating its effects

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

- Use shallow copying only where the aliasing caused is intentional.
- Use deep copying if there is any possibility that the aliasing of a shallow copy would affect the application adversely.
- If in doubt, use deep copying.
- Be careful in programming a deep copy to prevent infinite recursion when the references create a cycle.
- Use abstractions to ensure deep copies where needed, e.g., by (re-)defining assignment operations, constructors, and other operations that copy component values.

6.yy.6 Implications for standardization

In future standardization activities, the following items should be considered:

- Provide means to create abstractions that guarantee deep copying where needed.

6.zz Polymorphic variables [BKK]

6.zz.1 Description of application vulnerability

Some languages allow polymorphic variables, in which values of different types or classes can be stored at different time. For example, object-oriented languages permit variables to be declared to be of some class, while the actual value may be of a more specialized subclass. (For technical reasons, this capability is usually restricted to variables that are references or pointers whose designated object is of some such subclass.) Polymorphic variables go hand in hand with method selection at run time, when the method defined for the actual subclass of the receiving object or controlling argument is invoked. This approach is safe, as method implementation and nature of the object match by construction. If, however, the language permits casting of the polymorphic reference to process the object as if it were of the class casted to, several vulnerabilities arise. We distinguish

- “upcasts”, where the cast is to a superclass
- “downcasts”, where the cast is to a subclass and a check is made that the object is indeed of the target class of the cast (or a subclass thereof)
- unsafe casts, where there is no assurance that the object is of the casted class

Distinct vulnerabilities arise for each of these cast types:

Upcasts are needed so that redefined methods can call upon the corresponding method of the parent class to achieve the respective portion of the needed functionality and then complete it for the extensions added by the subclass. For example, this is the only way by which private components of the parent class can be initialized or manipulated by methods of a subclass. If the subclass-specific part is omitted, however, the object can end up in an inconsistent state.

Downcasts carry the risk that the object is not of the correct class. If checked, as language-defined downcasts typically are, an exception will occur in this case.

Unchecked casts allow arbitrary breaches of safety and security. See subclause [HFC].

Note that some languages also have implicit upcasts and downcasts as part of the language semantics. The same issues apply as for explicit casts.

6.zz.2 Cross reference

CWE: << TBD >>
JSF AV Rules: 78, 94
CERT C guidelines: <<TBD>>
Ada Quality and Style Guide: <<TBD>>

6.zz.3 Mechanism of failure

Objects left in an inconsistent state by means of an upcast and a subsequent legitimate method call of the parent class can be exploited to cause system malfunctions.

Exceptions raised by failing downcasts allow Denial-of-Service attacks. Typical scenarios include the addition of objects of some unexpected subclasses in generic containers.

Unchecked casts to classes with the needed components allow reading and modifying arbitrary memory areas.   See subclause [HFC] for more details.

6.zz.4 Applicable language characteristics

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

- Languages that have polymorphic variables, particularly object-oriented languages.
- Languages that permit upcasts, downcasts, or unchecked casts.

6.zz.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 use unchecked casts.
- Try to avoid downcasts. Instead prefer dynamic method selection based on the actual class of the receiving object or controlling argument to the downcasting of the reference to the respective class.

The second advice is controversial. In languages with static name binding of methods, it leads to specifications of methods in superclasses merely to be able to call them for subclasses. This destroys proper class design and creates classes with hundreds of methods. In languages with dynamic name binding of methods, it trades the “inappropriate class”-exception of the downcast against the “method-not-found”-exception of the dispatching call.

6.zz.6 Implications for standardization

In future standardization activities, the following items should be considered:

Do not allow unchecked casts.