OO-Texts as was my homework. ….

Not yet integrated with the existing text, where decisions need to be made about retaining examples and explanatory text about the “normal case”.

6.41 Inheritance

The vulnerabilities as described in ISO/IEC TR 24772-1:2019 clause 6.41 apply to Python.

Python supports inheritance through a dynamic hierarchical search of class namespaces starting at the class of a given object and proceeding upward through its superclasses. Multiple inheritance is also supported. Here the lookup sequence is a mixture of left-most depth-first and selective breadth-first traversal; the latter ensures that all search paths back to a given parent node are explored before this parent node is visited. The rule is embedded in the Method Resolution Order (MRO) of Python. It is difficult to establish manually and its outcome differs substantially from the usual rules in other OO-languages. Additionally, Python renders certain MRO’s illegal which further complicates the understanding of the rules. For example, in a class hierarchy described by

class O: ...

class P: ...

class A(P): ...

class B(P): ...

class Z(O): ...

class Y(Z): ...

class W(O): ...

class C(Y, A, B, W) ...

c = C()

c.meth()

the MRO for resolving the method name c.meth is the linear sequence  
 C – Y – Z – A – B – P – W – O – object.   
On the other hand, Python cannot establish a consistent MRO for   
 class C(Z, Y, A, B, W),   
because Z is a superclass of Y.

While not typically shown in the standard MRO notation, notice that “object’ is always the last class in every MRO chain.

There can be unexpected outcomes from the MRO as shown in the following code. The outcome might be expected to be a=0, but in reality the result is a=2 since, as previously mentioned, methods in derived calls are always called before the method of the base class (class T).

class T():  
 a = 0  
class A(T):  
 pass  
class B(T):  
 a = 2  
class C(A,B):  
 pass  
c = C()  
print(c.a) # => 2

It is important to make sure that each class calls the \_\_init\_\_ of its superclass so that it is properly initialized.

There is no protection in Python against accidental redefinition, method capture, or accidental non-redefinition along the MRO sequence, so that these vulnerabilities apply fully.

Moreover, as the search for a binding is at run-time in dynamically established class hierarchies, a static analysis cannot predetermine the danger of these vulnerabilities to incur. Neither can a reviewer of the code without detailed analysis of the entire class hierachy determine which method is called. Hailed as a flexibility in Python literature, it is possible to add an additional sibling class into a given hierarchy, thereby redefining parent method definitions (or adding new ones), so that the elder sibling appears to have these capabilities from the viewpoint of all classes below.

There are no language mechanisms to enforce class invariants when methods are redefined, so that class invariants can be easily violated by redefinitions.

Use of getter and setter methods to access class members cannot be enforced. There is a mechanism however, to make members effectively private: the use of leading double underscores (without matching trailing underscores) for their name implies only local visiblility in Python.

Any inherited methods are subject to the same vulnerabilities that occur whenever using code that is not well understood.

<<< …. Mitigations: I cannot think of any. >>>

------------------------

6.43.

The vulnerability as described in ISO/IEC TR 24772-1:2019 exists in Python. By default, all calls in Python are redispatching and thus can result in infinite recursion between redefined and inherited methods, as described in ISO/IEC TR 24772-1:2019.

Redispatching can be prevented by the use of Super() or by prefixing a method call by the name of the desired class. See, however, section 6.44 for associated vulnerabilities.

6.44. Polymorphics:

The vulnerability as described in ISO/IEC TR 24772-1:2019 exists in Python. While there are no casting operators in Python, prefixing method calls can achieve similar effects for these calls and cause respective vulnerabilities.

Super as a prefix to a call ignores local definitions and, instead, picks the binding from the next class in the applicable MRO (often a parent class as in most OO-languages, but occasionally a sibling class, as shown in the example in section 6.41). As such, it is reasonably safe, since the classes are ancestors of the class of the object, albeit possibly not yielding the expected binding. The vulnerabilities of *upcasts*, as described in ISO/IEC TR 24772-1:2019, apply in any case.

Prefixing a call with the name of a specific class forces the binding of the method name to be taken from this class. There is, however, no check performed whether the named class is an ancestor class of the class of the self object, and thus safe to use. Any class is accepted, turning the feature into an *unsafe cast* in the terminology of ISO/IEC TR 24772-1:2019. Subsequent failures occur in Python only when the class of self does not have members named by the implementation of the chosen method, or, if it does, malfunctions arise when the user semantics of these members are different in the two classes, e.g., a member count in two unrelated classes may stand for the count of very different entities.