ISO/IEC JTC 1/SC 22/OWGV N 0203

Meeting #11 markup of proposed new vulnerability description, Overloading and Overriding

Date	15 July 2009
Contributed by	Erhard Ploedereder
Original file name	
Notes	Markup of N 0201

6.X Overloading and Overriding [OAO]

6.X.0 Status and History

2009-07-10 Proposed by Erhard Ploedereder

6.X.1 Description of application vulnerability

Overloading is a feature present in many languages. It describes the notion that more than one specification of a subprogram (or other entity) exists, where all specifications share the same simple name (e.g., “open”) for the subprogram, but differ in their so-called signature. In most such languages, the signature consists of the types of the parameters; in some languages the result type is also part of the signature. Upon calling the subprogram, the compiler statically selects the subprogram with the signature that matches the types of the actual parameters (and the expected result type) of the call.

Overriding is a feature present in object-oriented languages. It describes the notion that a subclass may replace the implementation of an inherited method. Overriding is therefore also referred to as “redefining”. Much of the power attributed to object oriented programming is the ability to have polymorphic variables, that is, variables that denote instances of different types, usually constrained to be subclasses of the declared class of the variables. Upon calling a method on a polymorphic variable, the implementation of the method is chosen at run-time to be the one for the actual class (type) of the instance. This selection is often referred to as “dispatching”, the call is a dispatching call.

Overloading and overriding combine in most object-oriented languages. While over-loading introduces a new method for the given class, overriding alters the implementation of an existing inherited method. Moreover, in most languages, overloading is statically resolved based on the declared class of the polymorphic variable only, and overriding is subsequently resolved by dispatching at run-time among the many implementations of the statically resolved method.

Many object-oriented languages make no syntactic difference between the introduction of a new method and the redefinition of an existing method. Serious application vulnerabilities are the result.

6.X.2 Cross reference
<<< to be filled in by MISRA etc. experts >>>

6.X.3 Mechanism of failure

The mechanisms of failure are demonstrated with the following example. It is intentionally formulated in pseudo-code.

```
class C is
   method M(AnotherClass Y)  -- (1)

class CNew is new C
   method M(AnotherClass Y)  -- (2)

C  Instance;
if … then Instance := new C; else Instance = new CNew;
Instance.M(X);
```

Here method M is redefined by class CNew. Any call on M, e.g., “Instance.M(X)” will invoke the implementation (1), if the method is called on an Instance of actual type C, and implementation (2), if the method is called on an Instance of actual type Cnew.

A first, very obvious mechanism of failure arises, if the writer of CNew is unaware of the existence of method M in class C. While intending to introduce a new method, he or she actually redefines the method C.M. The probability of meeting the contract of C.M is then very low, hence the program is likely to malfunction when M is called for a polymorphic variable of class C that happens to hold an instance of class CNew. This issue is often referred to as “accidental redefinition”.

Obviously this fundamental property can be maliciously exploited even by novice programmers. All it takes is to add a new subclass (e.g., by dynamic loading), redefine a method to include malicious payload code, and cause an instance of the new subclass to be created and used in the attacked system.

Yet another obvious mechanism of failure is the following, where we simply have extended our example with a name of the method that is more mnemonic:

```
class C is
   method Memorize(AnotherClass Y)  -- (1)

class CNew is new C
   method Memorise(AnotherClass Y)  -- (2)
```

While the intention (and possibly contractual obligation) was to redefine the Memorize method, the redefinition does not happen. Instead, a new method is created for CNew and

Instance.Memorize of course never dispatches to this implementation, as intended. The problem can be called an “accidental lack of redefinition”.

Using modern browser capabilities and careful code reviews, the above problems can be avoided.

A much more difficult failure situation arises, when class CNew exists as described, class C does not have a method M, but, during maintenance, a new method M is to be added to C, yielding the above code situation and again the “accidental redefinition” syndrome with its failure modes. This problem is known as “method capture”.

Unlike our first scenario, this problem cannot be detected by the maintainer of a library of classes, since he or she cannot possibly browse all subclasses created anywhere on Earth by users of the library. The problem arises once the revised library meets the user code.

Variations of the theme include alterations of the parameter profile. If, for example, the parameter class of (1) is changed to a parent class of AnotherClass, which by itself is clearly an upwards-compatible change, (2) turns from a redefinition into an overloading and thus an accidental lack of redefinition. Similarly, a previously new method (3) in CNew can turn into a redefinition, and thus fall victim to method capture.

```
class C is
    method Memorize(AnotherClass Y)  -- (1)

class CNew is new C
    method Memorize(AnotherClass Y)  -- (2)
    method Memorize(ParentofAnotherClass Y) – (3)
```

6.X.4 Applicable language characteristics

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

1. provide for overloading, overriding, and dispatching as part of the object-oriented paradigm.
2. do not syntactically distinguish overloading and overriding.

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

~~A standard guideline for object-oriented programming~~A programmer should ~~be to~~ browse the inheritance hierarchies for inherited methods of equal name and equal signature, before a new method of this name and signature is introduced.

For method capture, there is no practical general avoidance strategy in today’s object-oriented languages with the specified characteristics. To not create subclasses of library classes is the only advice known to work, but obviously not generally reasonable.

Comment [JWM1]: This doesn’t give any advice. Does a “Not Overrides” pragma have any use here? If it is impossible to avoid, are there things that can be done to mitigate the consequences? Are there after-the-fact forensic measure that can be considered? Maybe it should be regarded as bad practice, during maintenance, to add methods to libraries that are known to be parents of other libraries unless the contents of the other libraries are known.

Maybe the advice is that you should perform an impact analysis whenever you install a new library.

Maybe the advice is that the code should check, at compile time or at run time, that the intended version of the library is being accessed.

6.X.6 Implications for standardization

Redefinition and the definition of new methods should be syntactically distinct. While this does not solve the problem of accidental redefinition and method capture completely, the user will be told by the resulting error messages that action is needed to correct the program. The subsequent remedy of changing the name of the method in the subclass to avoid a method capture is, however, not a good solution if the subclass is in turn part of a library, since it implies an incompatible change to users of the library.

6.X.7 Bibliography

<< numerous literature pubs in the late 90ies that I am too lazy to retrieve at this time >>