A simple security-aware MOP for Java

Denis Caromel, Fabrice Huet and Julien Vayssière

INRIA - CNRS - I3S
Université de Nice Sophia-Antipolis
First.Last@sophia.inria.fr

Abstract. This article investigates the security problems raised by the use of proxy-based runtime meta-object protocols (MOPs) for Java and provides an approach for making meta-level code transparent to base-level code, security-wise. We prove that, but giving all permissions only to the kernel of the MOP and by using Java's built-in mechanism for propagating security contexts, the permissions required by base-level and meta-level code do not interfere. We illustrate this result in the context of a simple proxy-based runtime MOP that we wrote.

1 Introduction

Different authors have suggested using Meta-Object Protocols (MOPs) as an elegant solution for implementing security mechanisms. However, studying the security issues *raised by* MOPs has never been adressed as a problem on its own.

A previous work [2] provided a set of rules for combining together the permissions associated with the different protection domains of a typical reflective component-based application. One of the main results was that a proxy-based runtime MOP implied fewer constraints than other types of MOPs.

This article presents an approach that makes it possible to have, security-wise, a completely transparent meta-level in the context of a simple proxy-based runtime MOP that we designed and implemented for Java.

The paper is organised as follows. Section 2 introduces related work. Our simple proxy-based runtime MOP is described in section 3, while section 4 proposed a technique for making this MOP transparent with respect to security, and section 5 concludes.

2 Related Work

In this section we first present an overview of the different kinds of MOPs that exist for Java, then review some related work on meta-programming and security.

2.1 Different kinds of MOPs

MOPs come in multiple flavours. Depending on the criterion used, many different classification of MOPs are possible. Ferber [4], for example, first proposed

to separate between *structural* and *computational* reflection. The former is concerned with the structure of a program, that is its classes, inheritance relations and so forth, while the later deals with elements that only exist at runtime, such as method invocation or object creation. Alternatively, it is possible to make a distinction between *implicit* and *explicit* reflection, which indicates whether the shifts to the meta-level are visible from the base level or not.
However, we have chosen to follow here a third classification of MOPs that uses as a criterion the period in the life of a program when meta-objects are actually in use. The three periods of time we consider are *compile-time*, *load-time* and *runtime*. A specific MOP may use meta-objects at more than one period in the life of a program. This is why we end up with six different cases (see Fig. 1), which range from the most static compile-time MOPs to the most dynamic run-time MOPs.

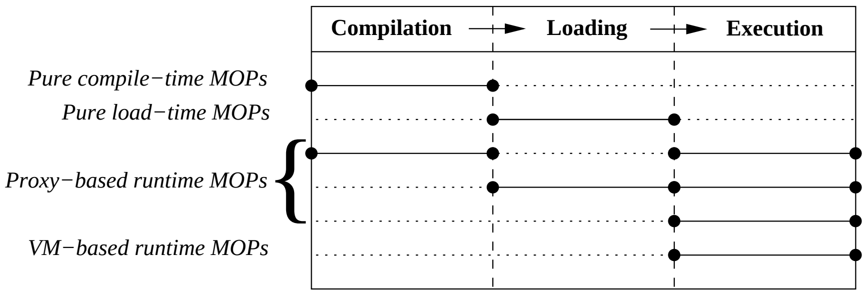

Fig. 1. A classification of MOPs based on when meta-objects are actually in use

Pure compile-time MOPs perform source-to-source transformations. The difference with usual pre-processors is that the translation itself is expressed as a Java program which handles meta-objects that represent classes, methods, loops, statements, etc. OpenJava [3] is an example of such a MOP.

Pure load-time MOPs perform bytecode-to-bytecode transformations. Like for pure compile-time MOPs, the translation uses meta-objects for providing an object view of the bytecode representation of a class. Most of the time, load-time MOPs are implemented through a specialised classloader object.

Proxy-based MOPs introduce hooks into the program in order to reify runtime events such as method invocation or access to fields. The hooks are introduced either at compile-time, for example by using a compile-time MOP, or at load-time by modifying the bytecode for a class (see Kava [10]) or even at runtime [7] by using objects that implement the *proxy* pattern (also called *wrapper* objects). These MOPs do not require any modification to the JVM, which explains why some low-level events cannot be reified.

VM-based Runtime MOPs rely on a modified JVM in order to intercept things that only exist at runtime, such as method invocations. On occurrence of such events, control is transferred to meta-level objects that are standard Java objects. MetaXa [6], Guaraná [7], and Iguana/J [9] are such MOPs.

2.2 Security and MOPs

The idea of using MOPs for implementing security mechanisms has been explored in a number of different works [1, 11]. However, the problem of studying the security problems raised by MOPs received little attention so far.

On several occasions, MOP implementors addressed the issue of security in their work, but mostly as a side note. Oliva and Buzato in [8] present some ideas on how to discipline the interaction between base-level and meta-level objects in their VM-based runtime MOP Guaraná, especially with respect to dynamic reconfiguration of the binding between the two levels. Welch and Stroud in [10] also present some security issues raised by their proxy-based run-time MOP Kava. They introduce the idea of making a clear separation between classes of the kernel of the MOP that are trusted and untrusted meta-object classes developed by third parties.

3 A simple proxy-based MOP

The purpose of our MOP is to reify two elements of the execution environment of a Java program: method invocations on objects[1] and calls to constructors.

3.1 Mapping base-level objects to meta-objects

Deciding which objects are reified and which are not is done on a per-object basis: for a given class, non-reified instances are created, as usual, using the `new` operator of the Java language, while reified instances are created using a call to the static method `MOP.newInstance` of our library.

One of the originalities of our MOP lies in the way the programmer declares which meta-level class is to be used when a reified instance of a given base-level class is created. Unlike other MOPs, which use a separate file for declaring such mappings, we have chosen to use the interface construct of the Java language instead. More specifically, we have an open-ended set of marker interfaces[2] which all inherit, either directly or indirectly, from `Reflect` (see Fig. 2). Each interface has a static `String` field that contains the name of a meta-level class and that can be overridden by sub-interfaces.

A base-level class implements one of these interfaces in order to declare which meta-object should be associated with a reified instance of this class. When it is not possible to modify the base-level class, the programmer can subclass the

[1] as opposed to the invocation of `static` methods.

[2] Marker interfaces are interfaces that do not declare any methods but 'flag' a class with a specific property. The use of marker interfaces has become an idiom of the Java language, as exemplified by the `Serializable` or `Cloneable` interfaces.

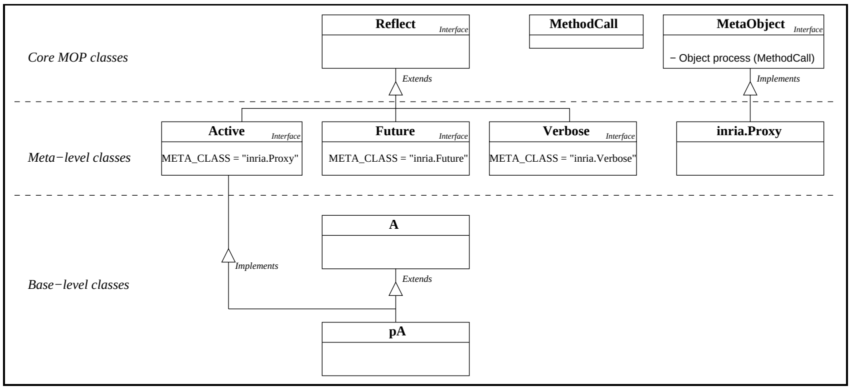

Fig. 2. Class diagram of the core MOP classes, meta-level classes and base-level classes.

base-level class in order to implement the marker interface in the subclass. This is the case shown in figure 2 where the base-level class `pA` inherits from class `A` and also implements the marker interface `Active`. If this, in turn, is not possible, the programmer can still specify which meta-level class to use by passing its name as a parameter of the call to `newInstance`.

3.2 Meta-level classes

Meta-level behaviours are expressed in meta-level classes that all implement the interface `MetaObject`. Each time a call to a reified object is intercepted by the MOP, it is reified into an instance of class `MethodCall` and passed to the meta-object associated with the reified object as a parameter to the call to the method `process` declared in interface `MetaObject`.

Our MOP does not impose any constraint on the organisation of the meta-level. It is up to the meta-level objects to handle the creation and organisation of the computations that take place at the meta-level.

3.3 Implementation

We implemented our MOP with two main design goals in mind: to provide a non-intrusive reification mechanism and to induce as few modifications of the source code as possible when retro-fitting existing code with meta-level behaviour.

Transparent interception of method calls is achieved through the use of *stub objects*. Stub objects are created and returned by the MOP as the result of the creation of a reified object and represent the reified object for its clients. Stub objects are instantiated from stub classes that are type-compatible with the class of the reified object. Then, it becomes possible to use the stub object wherever the original non-reified object is expected. What a stub class actually does is simply to redefine all the methods it inherits from its superclass[3] in order to, within the body of each such method, build an object that represents the call and pass this object to the meta-object associated with the stub. If needed, stub classes can be generated on-the-fly by our MOP: the source code is generated and written to the local file system, compiled and loaded just like any other locally-available Java class.

4 Security and MOP in a single address space

In this section we first present a quick overview of the security architecture of Java 2 and then show how, within a single virtual machine and under certain conditions, we obtain an interesting result: using our MOP for adapting base-level components is perfectly transparent with respect to security.

4.1 The Security Architecture of Java

The security architecture of Java 2 [5] is mostly concerned with *access control*, i.e. protecting access to critical local resources such as files, sockets, or the windowing system. An application is composed of a number of *protection domain*, which correspond to a URL where classes can be downloaded from or to a set of certificates that can be used for signing classes. Each protection domain has an associated set of *permissions*, each permission consists of a resource (say, a file) that we want to protect access to, and an access mode (such as read, write, ...).

At runtime, each of the classes loaded inside the virtual machine is associated with a specific protection domain. Whenever a thread performs a call that requires a specific permission, the security manager computes the *intersection* of the permission sets of all the protection domains on the execution stack of the thread. If the resulting set of permissions contains the permission needed for accessing the resource, the access is granted, otherwise an exception is thrown.

The security architecture of Java also provides the `doPrivileged` construct that limits the computation of the intersection of the permissions of the protection domains where `doPrivileged` is invoked, and those invoked from it.

4.2 A security-aware MOP

In the context of an component-based application that runs within a single virtual machine, we want the following property to hold:

In a reified call that crosses the boundary between two protection domains, the set of permissions under which the reified call is actually executed in the second protection domain after meta-level processing should be exactly the same as if the call had not been reified at all.

[3] Of course, `final` classes and `final` methods of non-final classes cannot be reified, which is a limitation of our MOP.

We will now prove that this property holds with our MOP, given that reasonable permissions are given to the different components.

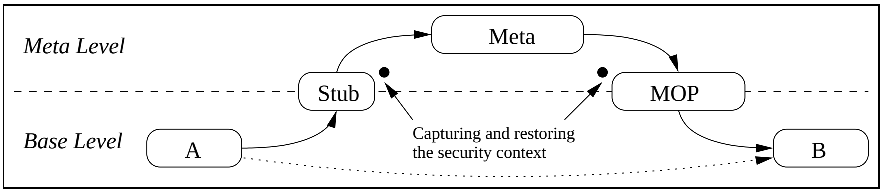

Fig. 3. Chain of calls in a reified call and handling of the security context

Let us consider that we have a component A that calls a method on a component B. This method performs an operation that requires an access check, therefore the set P_{std} of the permissions that are needed for the call to succeed in the standard case is the intersection of the permissions of A and the permissions of B. If the call from A to B is reified with a proxy-based runtime MOP like ours, we obtain the chain of calls described in figure 3, which lead to the permission set P_{ref}. If we call P the function that maps protection domains to their set of permissions, we have

$$P_{std} = P(A) \cap P(B) \text{ and } P_{ref} = P(A) \cap P(Stub) \cap P(Meta) \cap P(MOP) \cap P(B)$$

We will now explain step by step how, by making sensible choices with respect to which permissions are granted to each of the protection domain in the above equation, we can build a MOP for which $P_{std} = P_{ref}$.

Capturing and restoring a security context. With the security architecture of Java 2, it is possible to take a snapshot of the set of permissions associated with the current thread with a call to `AccessController.getContext()`. This call returns a `AccessControlContext` object that represents the permissions available to the current thread at the moment of the call. This object can be used later in a call to `doPrivileged` in order to perform a call under the security context captured within the `AccessControlContext` object.

We use this feature for capturing the security context when the call is reified (in the stub object), and restoring it later when the reified call is actually performed on the reified object, that is when the `execute` method is called on the `MethodCall` object that belongs to the *MOP* protection domain (see Fig. 3).

The security context $P_{captured}$ at the moment of the reification of the call is re-injected inside the component MOP, which leads to

$$P_{captured} = P(A) \cap P(Stub) \text{ and } P_{ref} = P_{captured} \cap P(MOP) \cap P(B)$$

$$\text{and hence } P_{ref} = P(A) \cap P(Stub) \cap P(MOP) \cap P(B)$$

Which is already an improvement over the previous value of P_{ref} because we have managed to remove the term $P(Meta)$ that corresponds to the permission set of the meta-object. Capturing and restoring a security context does not require any modification to the base-level classes or to the meta-object component $Meta$, but only to the classes of the MOP.

In the MOP we trust By nature, the classes of the MOP perform security-sensitive operations. This includes such different things as, for example, reading standard user properties, writing the source file of stub classes to the local file system and calling the compiler, or using the Reflection API for invoking non-public methods, which also requires a specific permission.

For all those reasons, we advocate that the protection domain that contains the base classes for the MOP should be granted all permissions. This means that, when using a MOP in a component-based application, the MOP must be fully trusted, which, by the very nature of a MOP, is very much needed anyway. This does not mean at all that the meta-level classes that implement the meta-level behaviour ($Meta$ in our example) will all be granted all permissions because those classes actually belong to a different protection domain. As the *MethodCall* class belongs to the protection domain of the MOP, we have

$$P(MOP) = \infty$$

Granting permissions to stub classes With our MOP, stub classes can be generated either statically or dynamically. If stub classes are generated statically, it can be assumed that those classes will be bundled with the components that use them. In our case, this means that stub classes will belong to the same protection domain as A, and hence

$$P(A) \cap P(Stub) = P(A) \text{ if stubs are generated statically}$$

If, on the other hand, stubs are generated dynamically, they all go into the same directory on the local file system, which then becomes a protection domain of its own. As the stub classes are generated by the MOP itself, we believe they should be granted all permissions. It is the MOP that controls what goes into the code of stub classes, and granting all permissions to stub classes should arise from granting all permissions to the MOP classes. As a matter of fact, stub classes never perform any action that may require a permission, all they do is build meta-objects for reifying calls and forwarding them to the meta-level. As a consequence, granting all permissions to the protection domain of the stubs is not a dangerous thing to do, and in terms of permission sets we have:

$$P(Stub) = \infty \text{ and hence } P(A) \cap P(Stub) = P(A)$$

which means that in both cases, either static or dynamic, we end up with the same relation. The term P_{ref} then rewrites to:

$$P_{ref} = P(A) \cap P(B) \text{ which is enough for stating that } P_{std} = P_{ref}$$

and we have proved that using our MOP with the strategy and permissions we have implemented does not add or remove permissions to base classes.

5 Conclusion

Within the context of a proxy-based runtime MOP for Java, we proved that it is possible to prevent the security constraints of the base-level and these of the meta-level from interfering, given that the classes for the MOP (but not the classes that implement meta-level behaviors) are given all permissions. In the near future, we plan to widen the scope of the results presented in this paper and work on designing a general security model for reflective applications.

References

1. M. Ancona, W. Cazzola, and E. B. Fernandez. Reflective authorization systems: Possibilities, benefits, and drawbacks. *LNCS*, 1603:35–50, 1999.
2. Denis Caromel and Julien Vayssière. Reflections on MOPs, Components, and Java Security. In J. Lindskov Knudsen, editor, *Proceedings of ECOOP 2001*, volume 2072 of *LNCS*, pages 256–274, Budapest, Hungary, June 2001. Springer-Verlag.
3. Shigeru Chiba and Michiaki Tatsubori. Yet another java.lang.class. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.
4. J. Ferber. Computational reflection in class based object-oriented languages. *ACM SIGPLAN Notices*, 24(10):317–326, October 1989.
5. Li Gong. *Inside Java 2 platform security: architecture, API design, and implementation.* Addison-Wesley, Reading, MA, USA, june 1999.
6. J. Kleinoeder and M. Golm. Metajava: An efficient run-time meta architecture for java. Techn. Report TR-I4-96-03, Univ. of Erlangen-Nuernberg, IMMD IV, 1996.
7. A. Oliva and L. E. Buzato. The design and implementation of Guaraná. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 203–216. The USENIX Association, 1999.
8. Alexandre Oliva and Luiz Eduardo Buzato. Designing a secure and reconfigurable meta-object protocol. Technical Report IC-99-08, icunicamp, February 1999.
9. B. Redmond and V. Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for Java. In *ECOOP 2000 Workshop on Reflection and Metalevel Architectures*, June 2000.
10. I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. In Pierre Cointe, editor, *Proceedings of the second international conference Reflection'99*, number 1616 in LNCS, pages 2 – 21. Springer, July 1999.
11. I. Welch and R. J. Stroud. Using reflection as a mechanism for enforcing security policies in mobile code. In *Proceedings of ESORICS'2000*, number 1895 in Lecture Notes in Computer Science, pages 309–323, October 2000.