# Security and Meta Programming in Java

Julien Vayssière
Julien.Vayssiere@sophia.inria.fr

OASIS Team, INRIA - CNRS - I3S
Univ. Nice Sophia-Antipolis

**Abstract**

This article investigates the security problems that may appear with the use of meta-programming extensions to the Java language and also how meta-programming may help in expressing and implementing security policies. Depending on the moment when the shift from the base-level to the meta-level is performed, we present different security problems and their consequences. We also raise a number of issues related to security and metaprogramming that we hope will result in fruitful discussions within the meta-programming community.

## 1 Introduction

In this article we present some early thoughts and remarks about how meta-programming in Java and the new security architecture of Java [10] relate to each other.

It is actually quite remarkable that Java is the first mainstream programming language to take security into account from scratch and at the same time gave birth to such a large number of meta-programming extensions. This is why we think it is interesting to have a closer look at how these two seamingly uncorelated concerns interact with each other.

The organisation of the paper is as follows. In section 2 we explain why Java is an attractive platform for developing metacomputing extensions and present the standard reflective features available in the language. In section 3, we investigate what it means for a meta-object protocol (MOP) [13] to be secure by reviewing a few MOPs for Java and describing their strengths and weaknesses with respect to security. In section 4 we look at the problem from an opposite angle: assuming that MOPs are secure, how can we put them to good use in order to implement security policies ? We conclude with a few hints at what we think may be valuable research issues in the field of meta-programming and security.

## 2 Java and Meta-Programming: a good match

Since its first public release in 1995, Java[1] has become an implementation platform of choice for researchers in the field of meta-programming with object-oriented languages. A significant number of systems have been proposed and implemented, which cover a broad range of implementation techniques, from compile-time and load-time MOPs to run-time MOPs. Let us now quickly present what we think are the three main reasons why Java became such an appealing platform for implementing meta-programming systems.

First, Java is an interpreted language and interpreters are a popular model for thinking about and implementing reflective features into programming languages [7]. Interpreters provide a natural separation between an application written in a given language (like Java) and the description of how this language is executed (how the JVM is implemented in C, for example). Interpreters allow to alter the execution of a program simply by modifying the interpreter without having to modify the program itself. It is the approach taken by such runtime-MOPs as MetaXa[2] and Guarana.

Second, the Java language itself contains a set of limited reflective features which designers of meta-programming systems can use as basic building blocks. The role of these features is to make things that belong to the execution environment available to Java programs as standard Java objects. Classes such as `ClassLoader` [9], `Reference`[3], `SecurityManager` and `AccessController` clearly are reflected elements of the runtime environment, which a Java programmer can customise using standard Java code. It is always arguable which features should be reflected and which should not. Java does not provide, for example, reflected views of the garbage collection algorithm or the thread scheduler, which a Java programmer may like to customise as well.

The best-known reflective feature of Java is the *Reflection API* [17]. It was introduced with JDK 1.1 and provides *introspection* features: a Java program can discover at runtime what are the types (primitive types, arrays, classes and interfaces) that exist inside the JVM and enquire about their constructors, methods and variables. The Refection API also provides a limited possibility to dynamically invoke those reflected members but never comes close to *behavioural reflection* [7], which is a completely different story.

A third reason why Java is an interesting platform for implementing meta-programming systems is that Java classes are loaded and linked on demand at runtime. The class-loading mechanism is reflected through objects of type `ClassLoader`, which provides the indispensable hook for implementing load-time MOPs. This category of MOPs usually modify the bytecode representation of a class at load-

[1]by *Java*, we mean the whole *Java Platform*, which encompasses the *Java Virtual Machine* [15] (JVM), the *Java Language* [11] itself and all the *Java Core APIs*.

[2]formerly known as MetaJava

[3]The abstract class java.lang.ref.Reference was introduced with Java 2 and is the superclass of a number of classes that provide a limited degree of interaction with the garbage-collector.

time so as to add hooks to the bytecode that implement shifts from the base-level to the meta-level on the occurence of specific events.

Now that we've seen why Java is an attractive platform for meta-programming, let us have a closer look at some of the existing metaprogramming extensions for Java and more specifically at how they do or do not fit with the security architecture of Java 2.

## 3 MOPs and the Security Architecture of Java

The question we address in this section is the following: does using a MOP for writing Java programs weaken or perhaps completely invalidate the security architecture of Java ?

The answer to the question depends heavily on when and how the shifts from the base-level to the meta-level (known as *reification points*) are introduced in the program. With respect to when the shift to the meta-level happens, MOPs can be broadly sorted out into three categories.

*Compile-time MOPs* reflect language constructs available at compile-time such as classes, methods, loops, statements,... and are usually implemented through some kind of source code preprocessor. *Load-time MOPs* reflect on the bytecode and make use of a modified class loaded in order to modify the bytecode at the moment it is loaded into the JVM. *Run-time MOPs* often make use of a modified version of the JVM in order to intercept things that only exist at runtime such as method invocations and field accesses. It is important to note that both compile-time and load-time mops work on a per-class basis while run-time MOPs have the ability to work an a per-object basis.

A somehow corelated issue is to decide *what* to make available at the meta-level, i.e. what to reify. Reifying method invocation is the most popular feature among MOPs, with reification points at the moment a method is entered of returned from. Reifying method invocations is a very desirable feature because many non-functional aspects can be efficiently implemented this way, like transparent distribution, persistence, access control or pre- and post-conditions on method calls. Reifying access to variables is also found, although it is less common than reifying method invocation, both because directly accessing other objects' variables somehow goes against sound object-oriented software engineering practice and because it is harder to implement without relying on a modified virtual machine.

Let us now present a number of examples in order to identify the issues raised by implementing MOPs in Java. We will see a compile-time MOP (*OpenJava*), a load-time MOP (*Dalang* which later evolved into *Kava*) and two run-time MOPs (*MetaXa* and *Guarana*).

### 3.1 OpenJava

OpenJava [5, 21] is a compile-time MOP for Java that inherits most of the design philosophy of its direct ancestor Open C++ Version 2 [4]. It can be seen as an "advanced macro processor" that performs a source-to-source translation of a set of classes written in a possibly extended version of Java into a set of classes written in standard Java.

The translation to be applied to a base class is described in a metaclass associated with the base class. The metaclass is written in standard Java using a class library that extends the Java Reflection API with new classes that reflect language constructs such as assignments, conditional expressions, field accesses, method calls, variables, type casts, etc... As a result, writing a translation is quite easy and natural because of the object-oriented design of the library, which contrasts with the approach taken in Open C++ where the sole abstraction made available to the meta-level programmer is bare abstract syntax trees.

The use of OpenJava does not break the security model of Java in any way: OpenJava outputs standard Java classes that compile and run within the standard Java environment and are subject to the same security restrictions as any Java class. Moreover, since OpenJava requires access to the source code of the classes it translates, we can assume that the translation is performed by the same person or organisation (in security terms, the same *principal*[4]) who wrote the source code for the base class, as opposed to load-time or run-time MOPs which can be used, for example, to add a distributed or persistent behavior to a Commercial-Off-The-Shelf (COTS) component [22], without the original implementor of the componenent having a word to say.

Nevertheless, there is still a little concern that does not introduce any breach of security as such but weakens the protection one might expect from Java 2 security architecture because it goes against the *principle of least privilege* [12]. The principle states that a piece of code "should operate using the least set of privileges necessary to complete the job". This principle is important for both computer security and software engineering since it limits the damage that can result from a security attack conducted by a malevolent attacker and it also protects a program from the consequences of a bug unwantingly introduced by a benevolent programmer.

In Java 2's security architecture, classes are grouped together into *protection domains*, which can be considered as a set of classes (usually a package) to which some permissions are granted through a *policy* file. The problem is that OpenJava allows a translation associated with a given base class to affect other classes that may belong to different protection domains than the protection domain of the base class, which blurs the fine-grained protection policy available in Java 2.

OpenJava restricts the scope of the translation expressed in the metaclass associated with a base class according to the following rule: a translation can only affect the base class itself (*callee-side* translation) and the classes that perform method calls to the base class (*caller-side* translation)[5].

As a consequence, a caller-side translation may introduce into all the client classes code that may require additional permissions in order to run. The user will then have to modify the *policy* file in order to grant new permissions to the protection domains that contain the client classes, because it is quite likely that the client classes belong to different protection domains than the class the translation if performed on. This clearly goes against the principle of least privilege.

OpenJava does not seem to raise any major security problem but is not perfectly in line with the security architecture of Java 2 though. It is difficult to say whether this results from OpenJava being too permisive or the security architecture of Java being too restrictive.

[4]A principal is a person or organisation on behalf of whom an operation is executed and who is responsible and accountable for the consequences of the operation.

[5]Performing caller-side translation implies that all the client classes of the base class on which the translation is performed are known at the time of the translation.

## 3.2 Dalang and Kava

Dalang [24] is a load-time MOP that makes use of class wrappers (also known as *proxy* objects [8]) in order to intercept method invocations. The idea behind load-time MOPs is to modify the bytecode of a class at load-time in order to introduce hooks that implement a shift from the base-level to the meta-level at some specific reification points in the code, usually on entering or leaving a method or on reading or writing a field.

Dalang evolved into Kava [23], which solved a lot of the security problems identified in Dalang. Nevertheless, it is interesting to present these problems here since they are certainly common to many load-time MOPs.

Since load-time MOPs work on the bytecode representation of a Java class (i.e. what is contained in a `.class` file), it is possible that the result of the translation of a class is a piece of bytecode that could not have been produced as the result of the compilation of a valid Java class.

Apart from permissions and protection domains, a large part of the security architecture of Java relies on the type system and access modifiers. Name space separation, for example, entirely relies on the soundness of the type system and is a crucial issue for Java applets. Allowing a class to modify its position in the inheritance tree by means of bytecode modification at load-time may completely undermine security. Even without any modification of the inheritance relationship between classes, simply changing the access modifier of an object variable is a serious security threat.

For example, changing the access modifier of the object variable that points to the private key of a cryptographic key pair from `protected` to `public` may allow a rogue class to leak the private key to an external party.

Kava's answer to this problem is to let the user have control over which transformation is applied to which class. This is done through a meta configuration file whose usage is similar to the *policy* file for security. Nevertheless, this is an all-or-nothing solution: the user cannot express fine-grained constraints on what the meta-level class is allowed to do. We would like, for example, to allow a metaclass to only perform translations of bytecode that respect the structure of a class (its name, position in the inheritance tree and method signatures), because, for example, this is all we need for intercepting method invocations.

Although load-time MOPs work on a per-class basis, it is possible to transform the bytecode in such a way that the reification mechanisms can be switched on and off on demand on a per-object basis without the resulting object suffering heavy runtime penalties.

It seems unlikely that load-time MOPs suffer from the same problem as compile-time MOPs (when a translation performed on a class also affect its client classes), since load-time MOPs work on a class-by-class basis. This means that the problem with permissions and caller-side translations presented in 3.1 does not exist here but a new problem related to the *policy* file appears.

The problem has to do with digital signatures. Java's security architecture relies on digital signatures to ensure the integrity of classes and the authentication of their authors. Of course, a secure class loader can be used to check the signature of the base class, but how do we handle the compound class that is the result of the translation of the base class according to its meta-level class ? If the user's *policy* file grants some permissions to the base class, should we grant the same permission to the compound class ? And even if both the base class and the meta class are trusted, how do we know that the resulting class won't perform harmful actions because of an unforeseen consequence of the translation ?

From a more general point of view, the problem raised here is one of *compoung principal*: who is to be held responsible for the actions taken by a class that is created as the composition of a base-level behaviour and a meta-level behaviour, each of which is under the responsibility of a different principal ? We can at least identify three principals: the author of the base-level code, the author or the meta-level code and the author of the MOP that binds that base-level and meta-level code together. Further research is needed in order to understand the security implications of developing programs with the use of MOPs.

Nevertheless, we think that load-time MOPs are probably the best compromise because they do not go against the load-time binding model of Java (as opposed to compile-time MOPs) and do not require a modified version of the virtual machine (as opposed to run-time MOPs) and introduce a reasonable performance penalty.

## 3.3 MetaXa and Guarana

MetaXa [14] and Guaraná [18] are two examples of run-time MOPs. They both rely on a modified version of the Java virtual machine. Guaraná is implemented using a modified version of the freely-available Kaffe virtual machine and MetaXa extends the virtual machine with a collection of native methods put together in a dynamic library.

The very fact that these MOPs rely on a modified version of the JVM is not necessarily a security problem. It is not the implementation of the MOP that creates a security threat (the modified virtual machines can be trusted because their implementors are clearly identified persons) but what the MOP enables a user class to do.

The problem is actually with controlling the enormous power unleashed by letting metaclasses access the inner workings of the Java Virtual Machine. For example, what happens if it becomes possible to mofify the state of the execution stack of a thread, which is central to the decision-making algorith of Java 2's security model ?

A possible solution is to control the power of run-time MOPs through a trusted kernel that would restrict access to potentially dangerous meta-level operations to a reduced set of classes.

## 3.4 Other MOPs in Java

The MOPs for Java that we've presented so far are the best-known and most representative meta-programming architectures for Java. Nevertheless, there exist other MOPs for Java, such as *RJava* [6], a wrapper-based runtime-MOP, *Reflective Java* [26], a compile-time MOP for intercepting method invocation, or ProActive [3], a run-time MOP that generates type-compatible wrappers at runtime for reifying method invocation without relying on a modified virtual machine.

## 4 Using MOPs for implementing security policies

The idea of using MOPs for expressing and implementing security policies is not a new one. The security aspect of an application has long been recognized as fairly orthogonal to functional code.

A metaobject that intercepts method invocations for an object that represents a resource to be secured is the ideal place for implementing an access control checks without having to mix functional code with security-related code. In a model based on capabilities, a metaobject attached to a reference can control the propagation of the capability across protection domains. Riechmann [19], for example, proposes a model in which metaobjects attached to the boundary of a component control how references to objects that live inside the component are transmitted to other components, dynamically attaching security metaobjects to these references according to the level of trust of the component the reference is transmitted to.

A similar idea was developed with the concept of *Channel Reification* [16, 1]. This model enhances the message reification model with the notion of history (or state). The model was implemented in Java as part of a history-dependent access control mechanism [2] that goes beyond the well known access matrix model, which is essentially a stateless access control mechanism. The Channel Reification model is also claimed to be superior to the meta-object model where a single metaobject monitors all access to a resource because it works with method-level granularity and can be used for implementing *role-based access models* [20] which are particularly well-suited to distributed object-oriented computing.

Another instance of using MOPs for implementing security policy is presented in [25]. The idea here is to use the load-time MOP Kava (see 3.2) in order to adapt third-party components to meet real-world security requirements. The authors contrast their approach with the wrapper-based approach adopted by the Enterprise Java Beans framework and argue that load-time MOPs provide a cleaner implementation of meta-level security policies. In addition, having a separate meta-level for the security policy attached to a component eases the expression of any kind of high-level security mechanisms, while the wrapper-based approach seems less expressive and is in fact only appropriate for enforcing access control on resources.

These experiments proved the feasability of using MOPs for implementing security policies. Another issue is to know if this approach is worthwhile, i.e. if the expression of a security policy at the metalevel is orthogonal enough to functional code for this approach to become really workable in real-world applications.

In the context of Java, the very fact that the declaration of which permissions are granted to which piece of code (the *policy* file) is separated from the source code might be interpreted as a proof that functional code and the declaration of security policies are orthogonal. However, there is at least one hint that functional code and security are not that orthogonal. In practice, the security policy as described in the *policy* file is unworkable if the code does not make use of the `doPrivileged` call for bypassing part of the security mechanism.

## 5 Future Work

We now would like to list a number of points that we think are interesting research issues in the field of security and meta-programming.

- Is the expression of security policies really orthogonal to functional code so as to allow all security-related code to belong to the meta level ? Or, as security is a growing concern in today's applications, will it become more and more difficult to draw a line between functionnal code and security-related code ?

- Is the use of MOPs for security really a new idea ? Or is it only the reformulation of decades-old security techniques in terms of meta-programming ?

- Which security policy should we apply to compound classes (like the classes generated by load-time MOPs such as Kava) that mix base-level and meta-level code ? What happens when the principals for the base-level and the meta-level code have different and possibly conflicting interests ? How is this issue related to the problem of security in open systems and mobile code systems ?

- How do we handle meta-level translations of base-leve classes whose effects cross protection domains (like for OpenJava) ?

## 6 Conclusion

This paper presents our remarks and thoughts on security issues with metaprogramming in Java and highlights possible weaknesses in today's MOPs. We also identify possible uses of MOPs for expressing security policies.

We hope that the few issues we provide as hints for future research will provide a useful starting point for fruitful discussions within the meta-programming community.

## References

[1] M. Ancona, W. Cazzola, and E. B. Fernandez. Reflective authorization systems: Possibilities, benefits, and drawbacks. *Lecture Notes in Computer Science*, 1603:35–50, 1999.

[2] Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. A history-dependent access control mechanism using reflection. In *Proceedings of the 5th ECOOP Workshop on Mobile Object Systems (MOS'99)*, Lisbon, Portugal, June 1999.

[3] D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.

[4] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 285–299. ACM Press, 1995.

[5] Shigeru Chiba and Michiaki Tatsubori. Yet another java.lang.class. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems.*

[6] José de Oliveira Guimarães. Reflection for statically typed languages. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 440–461. Springer, 1998.

[7] J. Ferber. Computational reflection in class based object-oriented languages. *ACM SIGPLAN Notices*, 24(10):317–326, October 1989.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison Wesley, Reading, 1996.

[9] Li Gong. Secure Java class loading. *IEEE Internet Computing*, 2(5):56–61, 1998.

[10] Li Gong. *Inside Java 2 platform security: architecture, API design, and implementation.* Addison-Wesley, Reading, MA, USA, 1999.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, ? 1997.

[12] Michael D. Schroeder Jerome H. Saltzer. The protection of information in computer systems. In *Proceedings of the IEEE.*

[13] Gregor Kiczales and Jim des Rivieres. *The art of the metaobject protocol.* MIT Press, Cambridge, MA, USA, 1991.

[14] Juergen Kleinoeder and Michael Golm. Metajava: An efficient run-time meta architecture for java. Techn. Report TR-I4-96-03, Univ. of Erlangen-Nuernberg, IMMD IV, 1996. english.

[15] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, USA, 1997.

[16] Gabriella Dodero Massimo Ancona, Walter Cazzola and Vittoria Gianuzzi. Channel reification: A reflective model for distributed computation. In *Proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC'98).*

[17] Sun Microsystems. Java core reflection, 1998. http://java.sun.com/products/jdk/1.2/docs/guide/reflection/index.html.

[18] Alexandre Oliva and Luiz Eduardo 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.

[19] T. Riechmann and J. Kleinoeder. Meta objects for access control: Role-based principals. *Lecture Notes in Computer Science*, 1438:296–??, 1998.

[20] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[21] Michiaki Tatsubori. An extension mechanism for the java language. Master's thesis, Graduate School of Engineering, University of Tsukuba, 1999.

[22] I. Welch and R. Stroud. Adaptation of connectors in software architectures. *Lecture Notes in Computer Science*, 1543:145–146, 1998.

[23] I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. *Lecture Notes in Computer Science*, 1616, 1999.

[24] Ian Welch and Robert Stroud. Dalang - a reflective java extension. In *OOPSLA'98 Workshop on Reflective Programming in C++ and Java.*

[25] Ian Welch and Robert Stroud. Supporting real world security models in java. In *7th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'99).*

[26] Zhixue Wu and Scarlet Schwiderski. Reflective java: Making java even more flexible. Technical report, ANSA, 1997.