Using the Dynamic Proxy Approach to Introduce Role-Based Security to Java Data Objects

Matthias Merz
Department of Information Systems III
University of Mannheim
L 5,5, D-68131 Mannheim, Germany

E-mail: merz@uni-mannheim.de

Abstract

In this paper we present JDOSecure[1]*, a novel security approach for the Java Data Objects specification. The objective of JDOSecure is to prevent unauthorized access to the data store while using the JDO API. Based on the dynamic proxy approach, JDOSecure is able to collaborate with any JDO implementation without source code modification or recompilation. It introduces a fine grained access control mechanism to the JDO persistence layer and allows the definition of role-based permissions. Moreover, since JDOSecure uses the Java Authentication and Authorization Service, it allows the pluggable integration of further authentication services.*

1. Introduction and Research Overview

This article presents the novel security architecture JDOSecure as add-on to the Java Data Objects (JDO) specification. JDOSecure introduces a role-based permission system to the JDO persistence layer. Based on a fine grained access control mechanism, JDOSecure prevents unauthorized access to the data store while using the JDO API.

The outline of the paper is organized as follows: Section one introduces the current JDO specification and emphasizes some shortcomings as well as the current development status of the JDO architecture. Section two outlines the security deficits of the JDO specification and recalls the design of the Java 2 Security Architecture. Section three presents the JDOSecure system architecture. The basic authentication and authorization concepts are introduced and the integration of JDOSecure with a JDO implementation is covered subsequently. The last section gives a critical review and addresses areas for future research.

[1]More information about JDOSecure can be found at `projekt-jdo.uni-mannheim.de/JDOSecure`

2. The Java Data Objects-Specification

The current JDO specification is an industry standard for object persistence developed by an initiative of Sun Microsystems under the auspices of the Java Community Process [5]. The latest 1.0.1 JDO version was introduced in May 2003 and is intended for use within the Java 2 Standard (J2SE) and Enterprise Edition (J2EE). It enables application developers to deal with persistent objects in a transparent fashion. Thus, JDO as a data store independent abstraction layer enables the mapping of domain object architectures to any type of data store.

The JDO specification defines two packages: The JDO Application Programming Interface (API) allows application developers to access and manage persistent objects. The classes and interfaces of the Service Providers Interface (SPI) are intended to be used exclusively by a JDO implementation and are located in the package `javax.jdo.spi`.

The interfaces and classes of the JDO API are located in the package `javax.jdo` [5]. Three important interfaces of the JDO API, namely `PersistenceManager`, `Transaction` and `Query` are summarized in the following: The `PersistenceManager` serves as primary application interface and provides methods to control the life cycle of persistent objects. The `Transaction` interface provides methods for initiation and management of transactions under user control. The `Query` interface allows to obtain persistent instances from the data store. Therefore, the JDO specification provides a Java oriented query language *JDO Query Language* (JDOQL).

Every instance that should be managed by a JDO implementation has to implement the `PersistenceCapable` interface. As part of the JDO SPI package, the `PersistenceCapable` interface has not to be implemented explicitly by an application developer. Instead, the JDO specification prefers a post-processor

tool (*JDO-Enhancer*) that automatically implements the PersistenceCapable interface. It transforms regular Java classes into persistent classes by adding the code to handle persistence. A XML-based *persistence descriptor* has to be configured previously. The JDO-Enhancer evaluates this information and modifies the Java bytecode of these classes adequately. The JDO specification assures the compatibility of the generated bytecode for the use within different JDO implementations. The StateManager interface as part of the JDO SPI provides the management of persistent fields and controls the object lifecycle of persistent instances.

Although JDO provides a standardized, transparent and data store independent persistence solution including tremendous benefits to Java application developers, the JDO specification has been discussed critically in the Java community. Beside technical details like the JDO enhancement process [10], the conceptual design as a lightweight persistence approach has been often criticized[2]. Some experts suggest, to shift JDO to a more comprehensive approach including distributed access functions to the persistent objects and multi-address-space communication [9]. As a result of it's lightweight nature, JDO does not provide a role-based security architecture, e.g. to restrict the access of individual users to the data store. Consequently, every user is able to query the entire data store as well as to delete any persistent object without further restriction.

In order to assist in the understanding of the JDOSecure security concept, the next section outlines the security deficits of the JDO specification and recalls the design of the Java 2 Security Architecture.

3. JDO Security Deficits

As already outlined in section 2, the JDO architecture is designed as a lightweight persistence approach without role-based security. Consequently, the JDO persistence layer does not provide any methods for user authentication or authorization. Every user has full access privileges to store, query, update and delete persistent objects without further restriction. For example using the getObjectById() method allows to receive any persistent object whereas the deletePersistent() method enables a user to delete objects from the data store.

A PersistenceManagerFactory instance will usually be constructed by calling the static JDOHelper method getPersistenceManagerFactory(Properties props). Using this method enables an easy replacement of the currently preferred JDO implementation without source code modifications. Therefore information about the currently used JDO implementation and data store specific parameters has to be passed to this method by a Properties object. Even more, the user identification and password to access the underlying data store are also part of the Properties object. In order to guard against misunderstandings the JDO persistence approach does not distinguish between different user identifications or individual permissions. With the construction of a PersistenceManager instance the connection to the data store will be established and users are able to access the resource without further restriction.

At first glance, a slight improvement could be achieved by setting up individual user identifications at the level of the data store. This would allow the construction of different and user dependent PersistenceManagerFactory instances. If, however, all users should have access to a common database, individual user identifications and appropriate permissions have to be defined inside the data store. However, configuring user permissions to restrict the access to certain objects is quite complex. For example, when using a relational database management system, the permissions would have to be configured based on the object-relational mapping scheme and the structure of the database tables. Thus, it leads to the disadvantage of causing a strong dependency between the user application and the specific data store. In addition, a later replacement of the currently preferred data store leads to a time consuming and expensive migration. It is obvious that the strong binding of security permissions to a specific data store contradicts the intention of JDO, which is to provide application programmers a data store independent persistence abstraction layer.

JDOSecure considers an alternative approach and introduces data store independent and user specific permissions. Thus, it allows to restrict the access to store, query, update and delete persistent objects. Furthermore, it focuses on the compatibility to the JDO specification and allows the collaboration with any JDO implementation. As JDOSecure is based on the Java 2 Security Architecture the following section gives a brief overview to this architecture.

4. The System Architecture of JDOSecure

This section outlines the system architecture of JDOSecure. First, the design of the Java 2 Security Architecture as a basis for JDOSecure is recalled. The basic authentication and authorization concepts are introduced and subsequently, the integration of JDOSecure with a JDO implementation is covered.

[2]the substantial overlaps between the Enterprise JavaBeans specification [4] and JDO has been discussed in [6].

4.1. The Java 2 Security Architecture as a Basis for JDOSecure

The Java security architecture is based on three components: Bytecode verifier, class loader and security manager (cf. [8] and [3]). The bytecode verifier checks the correctness of the bytecode and prevents stack overflows. The class loader locates and loads classes into the Java Virtual Machine (JVM) and defines a unique namespace for each class. The security manager or, more accurately, the `AccessController` instance checks the invocation of security relevant operations e. g. local file system access, the setup of system properties or the use of network sockets [7].

With introduction of the Java Authentication and Authorization Service (JAAS) in Java 1.4 it gets possible to restrict the access to resources depending on the currently logged on user (*user-centric authorization*). If the user identity is authenticated successfully, e. g. with the help of user identification and password, the system is able to validate the user permissions before granting access to security relevant resources. In Java this mechanism is implemented by the `SecurityManager` which delegates access-requests to the `AccessController`. This instance validates the permissions and allows or disallows the access to the resources.

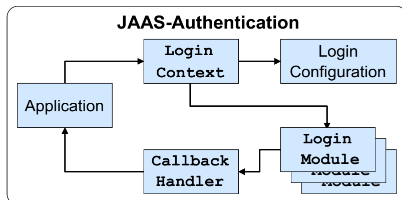

Figure 1. JAAS-Authentication

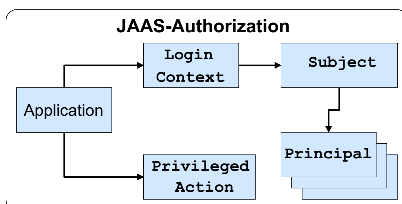

Figure 2. JAAS-Authorization

Figure 1 and 2 outlines the authentication and authorization process for JAAS. Starting with the authentication process, the application first creates a `LoginContext` and invokes the `login()` method of this instance. As defined in the login configuration file, the `LoginContext` delegates the authentication to one or several `LoginModules`. A `LoginModule` uses a `CallbackHandler` for communication with the application, the environment or the user e. g. to prompt for a password. JAAS provides the plug-in of PAM-authentication modules (Pluggable Authentication Modules) to integrate further authentication services. If an authentication attempt fails finally, a `SecurityException` will be thrown.

By calling the `getSubject()` method of the `LoginContext` it gets possible to refer to the `Subject` instance. This represents an authenticated user that is simultaneously associated with one or several `Principals` (a concrete user role). An application can perform a `PrivilegedAction` considering individual user permissions by invoking the static `Subject.doAs(subject, action)` method. Therefore the `AccessController` determines if a `Subject` is associated with at least one `Principal`, that provides the necessary permissions to perform this action. If a user or application has not the necessary permission to perform this `PrivilegedAction`, a SecurityException will be thrown. The relationship between permissions on the one hand and `Principals` on the other is defined in a separate *policy-file* (cf. section 4.4).

4.2. The Authentication Process

As described in section 3, a `PersistenceManagerFactory` instance can be invoked by calling the static `getPersistenceManagerFactory(Properties props)` method of the `JDOHelper` class. JDOSecure extends this concept in order to facilitate the collaboration between JDOSecure and any JDO implementation. Hence, JDOSecure provides a `JDOSecureHelper` class which is derived from `JDOHelper`. The `JDOSecureHelper` class overrides the `getPersistenceManagerFactory(Properties props)` method and serves as an entry-point for JDO applications.

The `Properties` object passed to the `JDOHelper` class contains amongst others user identification and password to access a JDO resource. As mentioned in section 3 the JDO architecture does not distinguish between different users. Therefore, the `JDOSecureHelper` analyzes the passed `Properties` object to authenticate a user at the level of the JDO persistence layer.

Once, a user has authenticated successfully, the `JDOSecureHelper` class constructs a new `PersistenceManagerFactory` instance. The basic idea in this context is to replace username and password in the `Properties` object, before the `JDOSecureHelper` class invokes the `getPersistenceManagerFactory(Properties props)` method of the original `JDOHelper` class.

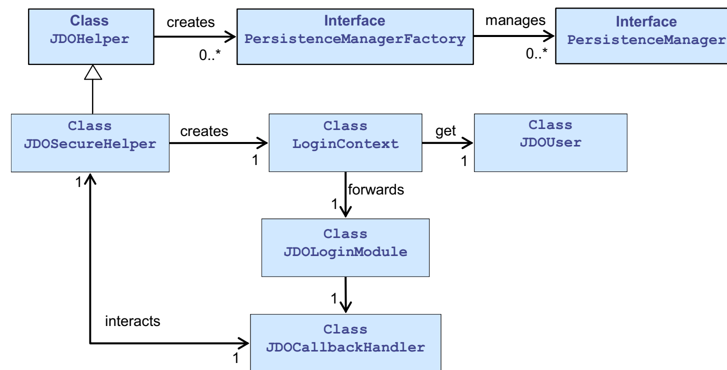

Figure 3. Context between JAAS-Authentication and JDOSecure

The intention of this replacement is to prevent a direct connection between user and JDO resource by using the `JDOHelper` class instead of the `JDOSecureHelper` class as a "workaround". The replaced password is unknown to the user and has to be configured by a security-administrator for the JDOSecure implementation and the JDO resource previously.

As illustrated in Figure 3, a `LoginContext` instance will be constructed by invoking the `getPersistenceManagerFactory()` method of the `JDOSecureHelper` class. The `LoginContext` instance forwards the authentication-request to the `JDOLoginModule` and `JDOCallbackHandler`. The `JDOCallbackHandler` instance validates the `ConnectionUserName` and the `ConnectionPassword` property to authenticate the user. If this process is completed without throwing a `SecurityException` the `LoginContext` instance is associated with a `JDOUser` instance and a `PersistenceManagerFactory` instance is constructed. As described in the next section, the `JDOSecureHelper` class returns a `PersistenceManagerFactory` instance (or more accurately a proxy object instead of the expected `PersistenceManagerFactory` instance, cf. section 4.3) to the user.

4.3. JDOSecure and the Dynamic Proxy Approach

There are two prerequisite conditions that could affect the acceptance of JDOSecure. First, JDOSecure should be independent from a concrete JDO implementation. And second, an overall approach should not contradict the JDO specification. In an attempt to meet these requirements, the presented security architecture implements the *dynamic proxy* pattern [1]. As it will be described, this concept enables the collaboration between JDOSecure and a standard JDO implementation, without an extensive adaptation.

A proxy instance implements the interfaces of a specific object and allows to control the access to it [2]. Generally, the creation of a proxy has to be done at compile time. Moreover, the dynamic proxy concept allows the construction of a proxy instance dynamically at runtime [1]. Dynamic proxy instances are created e.g. by using the static `newProxyInstance()` method of the `java.lang.reflect.Proxy` class and are always associated with an `InvocationHandler`. Any method invocation directed to proxy instance will be redirected to the `InvocationHandler.invoke()` method. The `invoke()` method allows to intercept method calls before they are forwarded to the original object.

The JDOSecure architecture implements the dynamic proxy concept as shown in Figure 4. The basic idea is to interpose a proxy between `PersistenceManager` and a JDO user or application. This would allow to validate specific user permissions at the `PMInvocationHandler` instance, before a method call is forwarded to the `PersistenceManager`. The following paragraph explains the architecture in more detail.

As mentioned above, the `JDOSecureHelper.getPersistenceManagerFactory()` method returns a dynamic proxy instance of the `PersistenceManagerFactory` class. Thus, the JDOSecure architecture avoids a direct interaction with the original `PersistenceManagerFactory`-instance and allows to manipulate method calls which are directed to the `PersistenceManagerFactory`. Invoking the `getPersistenceManager()` method, the `PMFInvocationHandler` returns a proxy of the `PersistenceManager` instance. JDOSecure uses the associated `InvocationHandler` (`PMInvocationHandler`) to manipulate method calls directed to the `PersistenceManager`. Thus, the

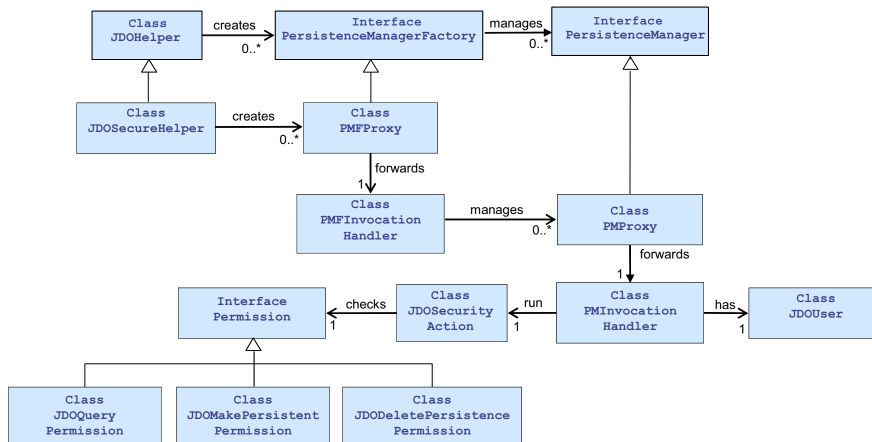

Figure 4. Context between JAAS-Authorization and JDOSecure

`PMInvocationHandler` represents the entry-point to implement the authorization function and allows to determine whether or not a user is allowed to invoke a `PersistenceManager` method.

4.4. The Authorization Process

JDOSecure enables the set-up of user specific permissions to allow or disallow the invocation of `PersistenceManager` methods. As mentioned above, a user receives a proxy of a `PersistenceManager` instance (`PMProxy`) by invoking the `getPersistenceManager()` method. Thus, JDOSecure is able to use the assigned `PMInvocationHandler` to validate, if an authenticated `JDOUser` has the permission to make a specific method invocation. The permissions are located in a separate policy-file and can be defined individually for any user. Currently, JDOSecure distinguishes between different permissions (Table 1) in order to restrict the access to the different `PersistenceManager` methods. JDOSecure enables also the limitation of user permissions to a certain package or a specific class. For example, the permission to invoke the `makePersistent()` method has to be defined for a package `org.test.sample` and a Principal ”sampleuser” as follows:

```
grant Principal JDOUser "sampleuser"{
 permission JDOMakePersistentPermission
                  "org.test.sample.*";
}
```

To validate if a user has the permission to invoke a specific `PersistenceManager` method, a `JDOSecurityAction` instance will be constructed and passed to the static `doAs(subject, action)` method of the `Subject` class. Consequently, the validation of a user permission is delegated to the `AccessController` as part of the Java 2 Security Architecture. If a user has the appropriate permission, the method call is forwarded to the original `PersistenceManager` instance. If not, a Java `SecurityException` is thrown.

Even this approach allows to restrict the creation, query and deletion of `PersistentCapable` instances, it is not suitable for the JDO update process. This problem is discussed in the following section.

4.5. JDOSecure and the Update of Object Attributes

JDO introduces the concept of transparent persistence and consequently JDO doesn't provide any additional methods to update object attributes or flushing instances to the data store. The security mechanism as described above, to verify user permissions when invoking methods of the JDO API, does not work in case of JDO updates.

As already mentioned, the JDO enhancer modifies regular Java classes to implement the `PersistentCapable` interface. Additionally, all setter methods are modified, that they do not change attributes directly. Instead, by invoking a setter method, an associated `StateManager` instance is notified. This `StateManager` is responsible to update the attributes in the corresponding `PersistentCapable` instance as well as to propagate these updates to the data base.

The idea in this context is to replace the `StateManager` by a proxy and to validate the user permissions in the corresponding `InvocationHandler` instance. As defined in the JDO specification, a `StateManager` instance

Methods of a `PersistenceManager`, that require specific permissions to be executed in the context of JDOSecure:	Necessary permission to invoke the according method for a specific class or package:
`makePersistent(..)` `makePersistentAll(..)`	`JDOMakePersistentPermission` $< Class >$
`deletePersistent(..)` `deletePersistentAll(..)`	`JDODeletePersistentPermission` $< Class >$
`getExtent(..)` `Query.execute(..)`	`JDOQueryPermission` $< Class >$

Table 1. JDOSecure Permissions

will be created by the JDO implementation with the invocation of the `PersistenceManager` methods `makePersistent()`, `makePersistentAll()`, `getExtent()`, `getObjectById()` as well as the `execute()` method of the `Query` instance. With the use of JDOSecure, the user does not interact with the `PersistenceManager` directly, but with the `PMInvocationHandler` instance. Before JDOSecure returns a `PersistentCapable` instance to the user, it gets possible to replace the corresponding `StateManager` by a proxy.

To implement this approach in JDOSecure, the `PMInvocationHandler` accesses the private `jdoStateManager` field by using the `java.lang.reflection` API to construct a dynamic proxy for the `StateManager`. In a second step, the `PMInvocationHandler` replaces the reference to the `StateManager` in the `PersistentCapable` instance with the proxy. The technical details like security issues when accessing private fields by using the `java.lang.reflection` API and other complications (e.g. the `jdoReplaceStateManager()` method of a `StateManager`) have been disregarded in this paper because of space limitations. However, JDOSecure enables the access control of the JDO update mechanism by introducing another proxy and a `JDOUpdatePermission`. As all other JDOSecure permissions, the `JDOUpdatePermission` could be specified individually for every user and a specific package or class.

5. Conclusion

In this article the novel security approach JDOSecure is introduced and the main advantages are highlighted. JDOSecure introduces a fine grained access control mechanism to the JDO persistence layer and allows the definition of role-based permissions. In the current version the permissions can be defined individually for every user/role with regards to certain operations (create, delete, update or query) and a specific class/package. Based on the dynamic proxy approach, JDOSecure is able to collaborate with any JDO implementation without source code modification or recompilation. Thus, JDOSecure will significantly contribute to the reduction of technical and conceptual JDO barriers and will consequently increase the security of JDO persistence architectures in the future. The JDOSecure software package and more information can be found at `projekt-jdo.uni-mannheim.de/JDOSecure`.

References

[1] J. Blosser. Explore the Dynamic Proxy API. http://java.sun.com/developer/technicalArticles/DataTypes/proxy/, 2000.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[3] L. Gong. Java 2 Platform Security Architecture. http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html, 2002.

[4] Java Community Process. JSR-153: Enterprise JavaBeans 2.1, 2003.

[5] Java Community Process. JSR-012: Java Data Objects (JDO) Specification, Maintenance Draft Review, 2004.

[6] A. Korthaus and M. Merz. A Critical Analysis of JDO in the Context of J2EE. In A.-A. Ban, H. R. Arabnia, and M. Youngsong, editors, *Proceedings of the 2003 International Conference on Software Engineering Research and Practice (SERP '03)*, volume I, pages p. 34–40. CSREA Press, 2003.

[7] S. Oaks. *Java Security*. The Java Series. O'Reilly & Associates, Inc., Sebastopol, CA, USA, second edition, 2001.

[8] Sun Microsystems. *The Java Language Specification*. Addison-Wesley Professional, 3rd edition, 2005.

[9] TheServerSide.COM. Craig Russell Responds to Roger Sessions' Critique of JDO. http://www.theserverside.com/articles/article.tss?l=RusselvsSessions, 2001.

[10] TheServerSide.COM. A Criticism of Java Data Objects (JDO). http://www.theserverside.com/news/thread.tss?thread_id=8571, 2003.