JDOSecure: A Security Architecture for the Java Data Objects-Specification

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

Abstract

Java Data Objects (JDO) is a specification for object persistence that enables application developers to deal with persistent objects in a transparent fashion. Since JDO is designed as a lightweight persistence approach, it neither supports user authentication nor role-based authorization. In order to remedy this situation, JDOSecure[1] has been developed in order to introduce a fine-grained access control mechanism to the JDO persistence layer. By implementing the Java Authentication and Authorization Service, JDOSecure allows the definition of role-based permissions. Moreover, since JDOSecure is based on the dynamic proxy approach, the collaboration with any JDO implementation is ensured.

1 Introduction

One key aspect of developing distributed multi-tiered enterprise application systems is how to provide an appropriate architecture for persisting business data. Numerous requirements have to be met, e.g. with regard to performance, concurrency, transactions, interoperability, or maintainability. Therefore, many different persistence solutions had been developed since Java was introduced in 1995. For example, the widespread Enterprise JavaBeans (EJB) component model covers a reasonable persistence solution for the J2EE environment. Besides EJB, proprietary object/relational mapping tools like Hibernate or TopLink are becoming increasingly popular. They provide an adequate way of mapping between the object-oriented application layer on the one hand and the conceptually different data stores on the other. However, using proprietary or platform- dependent solutions often locks developers into a particular vendor, and consequently, limits the portability of an application.

With the specification of Java Data Objects (JDO) [1], a new object persistence standard has been established in order to improve this situation. JDO introduces a standardized persistence abstraction layer and, moreover, it enables the usage of various data store types from different vendors, like relational databases, file systems, and object-oriented databases. However, since security issues have become more and more important when developing distributed enterprise application systems, developers more often call for authentication and authorization functionalities with regard to persistence solutions. Since JDO is designed as a lightweight persistence approach, it does not provide access control capabilities to restrict user access to persistent objects. In case access control is still required within the usage of JDO, individual user identifications and appropriate permissions have to be defined inside the data store. This would lead to the loss of portability and neutralizes the major advantage of JDO.

In order to remedy this inappropriate situation, JDOSecure has been developed to introduce a role-based permission system to the JDO persistence layer. By providing a fine-grained access control mechanism, JDOSecure prevents unauthorized access to the data store while using the JDO API. Based on the dynamic proxy approach, JDOSecure is also able to collaborate with any JDO implementation.

This paper attempts to examine the security issues within the JDO persistence specification. Hence, section 2 begins with an introduction to the JDO specification and outlines some of its shortcomings. Section 3 gives a brief overview on whether and how issues relevant to security are dealt with within JDO and other persistence frameworks. Section 4 presents the JDOSecure system architecture (for a more technical exposition on JDOSecure, we refer to [2]). The last section gives a critical review and addresses areas for future research.

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

2 Java Data Objects Specification

In this section, the Java Data Objects Specification will be introduced and afterwards, some JDO shortcomings will be mentioned.

2.1 JDO Overview

The JDO specification is an industry standard for object persistence developed by an initiative of Sun Microsystems under the auspices of the Java Community Process [1]. JDO 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 arbitrary type of data store.

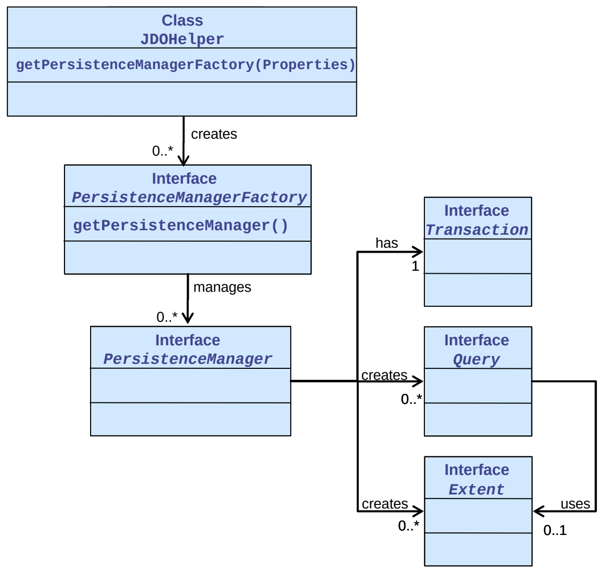

Figure 1: JDO-API

The JDO Application Programming Interface (API) allows application developers to access and manage persistent objects (cf. Figure 1). In order to enable an easy replacement of a currently preferred JDO implementation without source code modification, the JDO specification proposes the use of the static `JDOHelper` method `getPersistenceManagerFactory(Properties props)`. This method allows to construct a `PersistenceManagerFactory` instance by passing the information about the currently used JDO implementation and data store specific parameters within a `Properties` object. The `JDOHelper` class returns a `PersistenceManagerFactory` instance and allows the construction of a `PersistenceManager` instance. The `PersistenceManager` instance serves as a 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).

Each object that should be managed by a JDO implementation has to implement the public `PersistenceCapable` interface. It defines methods that provide services such as life cycle management for all persistent instances. As part of the JDO Service Providers Interface (SPI), it does not have to be implemented by an application developer. The JDO specification instead prefers a post-processor tool (*JDO-Enhancer*). Its objective is to transform plain java objects into persistent classes by adding the code to handle persistence. An XML-based *persistence descriptor* has to be configured first. 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 and data store types.

2.2 JDO Shortcomings

Although JDO provides a standardized and transparent persistence solution including tremendous benefits to application developers, the JDO specification has also been discussed in the Java community. One criticism concerns technical aspects of the JDO specification like the JDO enhancement process [3]. Other persistence solutions like db4o or Hibernate do not require the usage of a post-processor tool (cf. [4] and [5]). However, the conceptual design of JDO as a lightweight persistence approach has also been discussed controversially. Some experts even suggest shifting JDO to a more comprehensive approach including distributed access functions to the persistent objects and multi-address-space communication [6]. Finally, the substantial overlaps between the Enterprise JavaBeans specification [7] and JDO has been criticized sometimes [8]. As a result, Sun proposes to create a new persistence API for Java uniting EJB and JDO in future [9]. Nevertheless, today, JDO is still the persistence framework of choice, as long as data store independency is assumed.

Before going on by analyzing issues relevant to security in the context of JDO, the next section will give a brief overview about the availability of authentication and authorization mechanisms with regard to other persistence solutions.

3 Persistence and Security Issues

This section will give a brief overview on whether and how issues relevant to security are dealt with within different persistence solutions.

3.1 Related Work

The intention of this section is to survey and review related work in the context of JDO and security. Since Hibernate, Enterprise JavaBeans and JDO are the most commonly used solutions to implement persistence for distributed multi-tiered enterprise application, the security issues with regard to Hibernate and the EJB specification will be highlighted in the following section.

Hibernate is a popular object/relational persistence and query service that originally does not provide an access control mechanism. However, due to the need of security demands, the Hibernate community proposes the usage of a declarative permissions approach using JAAS and the `Interceptor` interface [10]. `Interceptor`s allows the registration of a custom object that gets called by Hibernate at application run-time. This approach provides declarative security functions to Hibernate applications e.g. for the use within a session facade layer. Since security aspects are getting increasingly important, the recent version of Hibernate provides the user's authentication via JAAS. Moreover, based on the Java Authorization Contract for Containers (JACC) specification, Hibernate3 allows CRUD operations for entities to be permissioned in a J2EE environment [5].

EJB is the server-side component architecture for the J2EE platform [7]. One of its most beneficial features is that it frees developers from having to deal with code that handles transactional behavior, security, connection pooling, threading, or even persistence. In the latter context, entity beans represent persistent data that can be shared across multiple simultaneous remote and local clients. In order to control access to entity beans, the EJB specification (since version 2.0) distinguishes between programmatic and declarative authorization. Programmatic authorization has to be implemented by an entity bean provider using specific methods in order to perform security checks. With declarative authorization, defining access permissions for EJB methods and security roles in a deployment descriptor becomes possible. As a result, the EJB container performs all authorization checks.

As it turns out, the recent Hibernate implementation and the Enterprise JavaBeans specification provide sufficient authentication and authorization functionalities. The next section focuses on the analysis of Java Data Objects within security issues.

3.2 JDO Security Issues

As mentioned in section 2.2, the JDO architecture is designed as a lightweight persistence approach without distributed access functions or multi-address-space communication. As a result of its lightweight nature, the JDO persistence layer does not provide any methods for user authentication or authorization either. By invoking methods of a `PersistenceManager` instance, every user has full access privileges to store, query, update and delete persistent objects without further restrictions. 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.

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 (cf. section 2.1). 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 as persistent data store, 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 on the one hand and the specific data store on the other. In addition, a later replacement of the data store preferred currently 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 providing application programmers with a data-store-independent persistence abstraction layer.

JDOSecure contributes to improving this situation by adding a fine- grained access control mechanism to the JDO persistence layer. The JDOSecure architecture, the authentication and authorization details and the collaboration with any JDO implementation will be introduced in the following section.

4 JDOSecure: A Security Architecture for the JDO-Specification

JDOSecure introduces a role-based access control mechanism to the JDO persistence layer. There are two essential preconditions 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. However, as it becomes clear, JDOSecure attempts to meet these two serious requirements.

4.1 JDOSecure Authentication

As already mentioned in section 2.1, the JDO specification proposes the use of the static `JDOHelper` method `getPersistenceManagerFactory(Properties props)` in order to enable an easy replacement of the currently preferred JDO implementation. 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, among others, user identification and password to access a JDO resource. The `JDOSecureHelper` class analyzes the passed `Properties` object in order to authenticate a user at the level of the JDO persistence layer.

The basic idea in this context is to replacing user name and password in the `Properties` object, before the `JDOSecureHelper` class invokes the `getPersistenceManagerFactory(Properties props)` method of the original `JDOHelper` class. The intention of this replacement is preventing a direct connection between the user and the 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. Once a user has authenticated successfully, the `JDOSecureHelper` class constructs a new instance of `PersistenceManagerFactory`.

As illustrated in Figure 2, 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`.

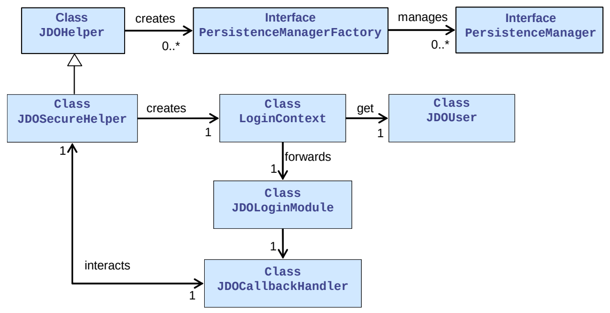

Figure 2: Context between the JAAS-Authentication and JDOSecure

The `JDOCallbackHandler` instance validates the `ConnectionUserName` and the `ConnectionPassword` property in order to authenticate the user at the level of the persistence layer. If this process is completed without throwing a `SecurityException`, the `JDOSecureHelper` class returns a `PersistenceManagerFactory` instance (or more accurately, a proxy object, as described in section 4.2) to the caller of the `getPersistenceManagerFactory()` method.

In order to understand the JDOSecure authorization process, it becomes necessary to discuss the overall architecture in the next section in a preliminary way.

4.2 The JDOSecure Architecture

As already stated above, one prerequisite condition that could affect the acceptance of JDOSecure is that the presented approach should be independent from a concrete JDO implementation. In an attempt to meet this requirement, JDOSecure implements the *dynamic proxy* pattern [11]. As it will be described, this concept enables the collaboration between JDOSecure and a standard JDO implementation, which becomes possible without an adaptation that is too extensive.

The JDOSecure architecture implements the dynamic proxy concept [11] as shown in Figure 3. The basic idea in this context is to interpose a proxy between `PersistenceManager` and a JDO user or application. Dynamic proxy instances are always associated with an `InvocationHandler` instance, which allows to intercept method calls before they are forwarded to the original object. For more technical details about how JDOSecure uses the dynamic proxy approach we refer to [2].

As mentioned above, the `JDOSecureHelper.getPersistenceManagerFactory()` method returns a dynamic proxy instance of the

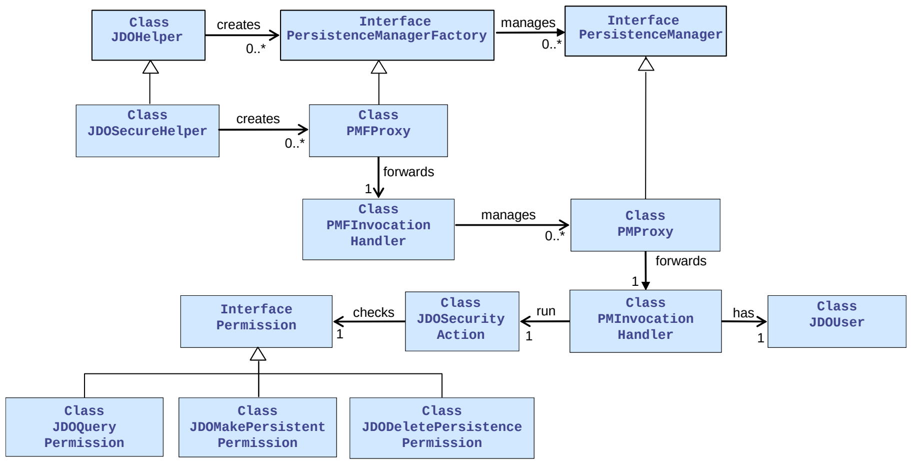
Class JDOHelper
creates
0..*
Interface PersistenceManagerFactory
manages
0..*
Interface PersistenceManager
Class JDOSecureHelper
creates
0..*
Class PMFProxy
forwards
1
Class PMFInvocation Handler
manages
0..*
Class PMProxy
forwards
1
Interface Permission
checks
1
Class JDOSecurity Action
run
1
Class PMInvocation Handler
has
1
Class JDOUser
Class JDOQuery Permission
Class JDOMakePersistent Permission
Class JDODeletePersistence Permission

Figure 3: Context between JAAS-Authorization and JDOSecure

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. By invoking the getPersistenceManager() method, the PMFInvocationHandler returns another proxy (in this case a proxy of a PersistenceManager instance). JDOSecure uses the associated InvocationHandler (PMInvocationHandler) to manipulate method calls directed to the PersistenceManager. Thus, the 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.

4.3 JDOSecure Authorization

With the introduction of the Java Authentication and Authorization Service (JAAS) in Java 1.4 restricting access to resources depending on the currently authenticated user [12] becomes possible. After the authentication process has finished successfully, the JVM is able to validate user permissions before granting access to resources relevant to security. In Java, this mechanism is implemented by the SecurityManager, which delegates access-requests to the AccessController.

Based on JAAS, JDOSecure enables the setup of user-specific permissions in order 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 individually defined for every user. Currently, JDOSecure distinguishes between different permissions (Table 1) in order to restrict 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.*";
}
```

In order to validate whether a user has permission to invoke a specific PersistenceManager method, the authorization is delegated to the AccessController as part of the Java 2 Security Architecture (for more technical details about how JDOSecure implements JAAS we refer to [2]). If a user has the appropriate permission, the method call will be forwarded to the original PersistenceManager instance. If not, a Java SecurityException will be thrown.

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

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

4.4 JDOSecure and the Update of Object Attributes

JDO introduces the concept of transparent persistence and, consequently, JDO does not provide any additional methods of updating object attributes or flushing instances to the data store. Therefore, the JDO enhancement process modifies the plain Java classes in order to implement the `PersistentCapable` interface. Additionally, all setter methods are modified, so that they do not change attributes directly. Instead, by invoking a setter method, an associated `StateManager` instance is notified. The `StateManager` instance is responsible for updating the attributes as well as for propagating updates to the data store.

In order to restrict the update of `PersistentCapable` objects by specific user permissions, JDOSecure replaces the `StateManager` by a proxy and validates the user permissions in the corresponding `InvocationHandler` instance. The implementation of this solution is quite complex, because JDOSecure is not able to construct a `StateManager` instance directly. In order to create a proxy of a `StateManager`, JDOSecure has to wait until the used JDO implementation has constructed this instance. As defined in the JDO specification, a `StateManager` instance will be created with the invocation of the `PersistenceManager` methods `makePersistent()`, `makePersistentAll()`, `getExtent()`, `getObjectById()` as well as the `execute()` method of the `Query` instance. Since the user does not interact with the `PersistenceManager` directly (see section 4.2), JDOSecure is able to replace the corresponding `StateManager` with a proxy.

Every `PersistentCapable` instance provides a private field `jdoStateManager` that references a `StateManager`. The `PMInvocationHandler` accesses this field by using methods of 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. Since a JDO implementation is able to replace the current `StateManager` by calling the `StateManager.jdoReplaceStateManager()`, intercepting this method call also becomes necessary. By invoking the `jdoReplaceStateManager()` method, the `StateManagerInvocationHandler` is responsible for exchanging the passed `StateManager` instance with a new proxy.

Thus, it becomes apparent that the presented approach is quite complex. Even worse, some details like another `InvocationHandler` or special security issues when using the `java.lang.reflection` API to access private fields are disregarded in this context 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 paper, we have discussed security issues in the context of JDO and other persistence frameworks. As it becomes apparent, Hibernate and EJB provides sufficient authentication and authorization capabilities. In contrast, JDO, as a lightweight persistence approach, does not provide an access control mechanism in order to restrict the user's access to persistent objects.

Therefore, JDOSecure has been developed to introduce a role-based permission system to the JDO persistence layer. Based on JAAS, JDOSecure enables the set-up of user specific permissions in order to allow or disallow the invocation of `PersistenceManager` methods. In the recent version, the permissions can be defined individually for every user/role with regard to certain operations (create, delete, update or query) and a specific class/package. Moreover, since JDOSecure is based on the dynamic proxy approach, the compatibility with any arbitrary JDO implementation is ensured. It becomes obvious that the implemented access control mechanism based on the dynamic proxy approach has negative effects on performance. An early performance analysis indicated a performance reduction of about 15-20% for simple CRUD operations. Nevertheless, since JDOSecure introduces a fine-grained access control mechanism to the JDO persistence layer, it will significantly increase the security of JDO persistence architectures.

References

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

[2] Merz, M.: Using the Dynamic Proxy Approach to Introduce Role-Based Security to Java Data Objects. Eighteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'06), San Francisco, USA, 5-7. July, (accepted) (2006)

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

[4] Grehan, R.: Complex Object Structures, Persistence, and db4o. Whitepaper, http://www.odbms.org/downloads.html (2005)

[5] Hibernate: Hibernate Reference Documentation, Version 3.1.1. http://www.hibernate.org/hib_docs/v3/reference/en/pdf/hibernate_reference.pdf (2006)

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

[7] Java Community Process: JSR-220: Enterprise JavaBeans 3.0, Proposed Final Draft. http://www.jcp.org/en/jsr/detail?id=220 (2005)

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

[9] DeMichiel, L., Russell, C.: A Letter to the Java Technology Community. http://java.sun.com/j2ee/letter/persistence.html (2004)

[10] Hibernate: Hibernate Security: Declarative permissions using JAAS and Interceptors. http://www.hibernate.org/140.html (2006)

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

[12] Oaks, S.: Java Security. Second edn. The Java Series. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2001)