Enabling declarative security through the use of Java Data Objects

Matthias Merz

Department of Information Systems III, University of Mannheim, Germany

Received 1 December 2006; received in revised form 1 February 2007; accepted 11 July 2007
Available online 22 October 2007

Abstract

The concept of declarative security allows the separation of security concerns from business logic and enables the development of highly flexible and secure applications. Whereas Hibernate and the Enterprise Java Beans specification provide sufficient authentication and authorization functionalities in the context of object persistence, the Java Data Objects (JDO) specification designed as a lightweight persistence approach doesn’t provide any declarative security capabilities.

The novel security approach, JDOSecure, introduces a role-based permission system to the JDO persistence layer, which is based on the Java Authentication and Authorization Service (JAAS). JDOSecure is based on the dynamic proxy approach and ensures the collaboration with any JDO implementation. It comprises a management solution for users, roles, and permissions and allows storing the authentication and authorization information in any arbitrary JDO resource. Furthermore, a Java-based administration utility with a graphical user interface simplifies the maintenance of security privileges and permissions.

Keywords: Java Data Objects; Declarative security; Persistence

1. Introduction

The development of distributed multi-tiered enterprise applications requires an adequate solution in order to handle the persistence of business data. Numerous approaches have been suggested like the widespread Enterprise Java Beans (EJB) component model that 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 a useful way of mapping the object-oriented application layer to the conceptually different data stores. However, using proprietary, platform or database dependent solutions often locks developers into a particular vendor, and consequently, limits the portability of applications.

With the specification of Java Data Objects (JDO) [1,2], a new object persistence standard has been established in order to improve this situation. JDO proposes a transparent and database independent persistence abstraction layer for Java. It enables application developers to deal with persistent objects in a transparent fashion. Furthermore, JDO as a data store independent abstraction layer enables the mapping of domain object architectures to any arbitrary type of data store.

However, since security issues have become more and more important, developers call for authentication and authorization functionalities with regard to persistence solutions more often. Since JDO is designed as a lightweight...
persistence approach, it does not provide access control capabilities restricting 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 neutralize the major advantage of JDO.

In order to remedy this situation, JDOSecure has been developed to introduce a declarative security mechanism to the JDO persistence layer [3,4]. Its objective is preventing unauthorized access to the data store while using the JDO-API. It introduces a fine-grained access control mechanism to the JDO persistence layer and allows the definition of role-based permissions. Based on the dynamic proxy approach, JDOSecure is able to collaborate with any JDO implementation without extensive source code modification or recompilation. JDOSecure comprises a management solution for users, roles, and permissions and allows storing the authentication and authorization information in any arbitrary JDO resource. Furthermore, a Java-based administration utility with a graphical user interface simplifies the maintenance of security privileges and permissions.

This paper attempts to examine 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 introduces the architecture of JDOSecure. In Section 4, a simple web application for a car rental company is presented as proof-of-concept for JDOSecure. The last section gives a critical review and addresses areas for future research.

2. Persistence and security issues

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

2.1. The Java Data Object specification

The Java Data Objects 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 was introduced in April 2002 and is intended for the usage 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 recent version of JDO (JDO 2.0) was finally approved in May 2006 [2]. Its objective is simplifying the usage of JDO e.g. by providing a standardized object/relational mapping format to allow a higher degree of application portability. It also introduces an attach/detach mechanism for persistent objects in order to facilitate middle tier architectures. One of its most beneficial features is the extension of the JDO Query language (JDOQL) to support e.g. projections, aggregates, or named queries.

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.

The interfaces and classes of the JDO-API are located in the package javax.jdo [1,2]. The Transaction interface provides methods for initiation and management of transactions under user control. The Query interface allows obtaining persistent instances from the data store by providing a Java-oriented query language called JDO Query Language (JDOQL). The PersistenceManager serves as primary application interface and provides methods to control the life cycle of persistent objects. An instance implementing this interface could be constructed by calling the getPersistenceManager() method of a PersistenceManagerFactory instance. Since PersistenceManagerFactory itself is just another interface, constructing an instance prior to this type becomes necessary. It usually could be constructed by calling the static JDOHelper method getPersistenceManagerFactory(Properties props). The class JDOHelper is also part of the JDO-API and enables an easy replacement of the currently preferred JDO implementation without too extensive source code modifications. The information about the currently used JDO implementation and data store specific parameters has to be passed to this method by a Properties object. The user identification and a password in order to access the underlying data store are also part of the Properties object. In order to prevent 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 JDO resource without further restrictions.

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 does not have 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. An 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 life cycle 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 also been criticized in the Java community. Besides technical details like the JDO enhancement process [5], the substantial overlaps between the Enterprise Java Beans specification [6] and JDO [7], as well as the conceptual design as a lightweight persistence approach has been criticized. Some experts even argue that shifting JDO to a more comprehensive approach including distributed access functions and multi-address-space communication [8] is necessary. As a result of its lightweight nature, JDO does not provide a role-based security architecture, e.g. restricting access of individual users to the data store. 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 restrictions. For example, using the `getObjectById()` method allows one 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. Whether, 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 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 to that, 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 providing application programmers a data store independent persistence abstraction layer.

### 2.2. Related work

The intention of this section is to survey and review related work in the context of JDO and security. Since Hibernate and the Enterprise Java Beans Specification also represents two popular approaches to implement object persistence in the context of distributed multi-tiered enterprise applications, the related security issues are 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 [9]. `Interceptors` 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 user authentication via JAAS. Moreover, based on the Java Authorization Contract for Containers (JACC) specification, Hibernate 3 allows CRUD operations for entities to be permissioned in a J2EE environment [10].

EJB is the server-side component architecture for the J2EE platform [11]. 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, Hibernate and the Enterprise Java Beans specification provide sufficient authentication and authorization functionalities. In contrast, the JDO specification designed as a lightweight persistence approach doesn’t provide any declarative security capabilities. In order to remedy this situation, the security architecture JDOSecure has been developed. The next sections introduce the system architecture of JDOSecure and outlines the basic authentication and authorization concepts.

3. The architecture of JDOSecure

This section outlines the system architecture of JDOSecure. As already noted, JDOSecure will introduce a role-based permission system to the JDO persistence layer based on the Java Authentication and Authorization Service. As pictured in Fig. 1, JDOSecure is intended to be interposed between an application and a JDO implementation.

In the following section, the basic authentication and authorization concepts are introduced and the collaboration of JDOSecure and a JDO implementation is covered. Afterwards, the problem of intercepting the JDO update process is discussed and an adequate solution is presented. Finally the users, roles and permissions management system of JDOSecure will be outlined and the application to maintain the appropriate information will be introduced.

3.1. The authentication process

As described in Section 2.1, 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 2.1, the JDO architecture does not distinguish among different users. Therefore, the JDOSecureHelper class 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. The intention of this replacement is to prevent a direct connection between user and JDO resource by using the JDOSecureHelper class instead of the JDOHelper 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 Fig. 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. The JDOCallbackHandler instance receives the ConnectionUserName and the ConnectionPassword property to authenticate the user. Further details about the authentication process are explained in Section 3.5. If this process completes without throwing a SecurityException, a PersistenceManagerFactory instance is constructed. In order to enable the implementation of an adequate access control mechanism, the JDOSecureHelper class does not return the PersistenceManagerFactory instance directly, but it returns a proxy of the PersistenceManagerFactory instance back to the JDO-based application as it will be described in the following section.

### 3.2. JDOSecure and the dynamic proxy approach

There are two prerequisite conditions that could affect the acceptance of JDOSecure. First, JDOSecure should be independent of a concrete JDO implementation in order to ensure an ongoing portability between different JDO implementations. And secondly, an overall approach should not contradict the JDO specification. In an attempt to meet these requirements, the presented security architecture is based on the dynamic proxy concept (cf. Fig. 3).

According to [12], a proxy provides “a surrogate or placeholder for another object to control access to it”. Therefore, a static proxy implements the original objects interfaces and delegates all method invocations to the original object. In this context, one disadvantage is the fact that the creation of a proxy has to be done at compile time. Instead, the dynamic proxy concept allows the construction of a proxy instance dynamically at run-time [13]. They could be constructed by using the static `newProxyInstance()`-method of the `java.lang.reflect.Proxy`-class.
method allows the creation of a proxy instance for a set of interfaces at run-time. A dynamic proxy instance will always be associated with an InvocationHandler and any method invocation to the proxy 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 is based on the dynamic proxy concept, as shown in Fig. 4. The basic idea is interposing a proxy between a JDO-based application on the one hand, and the JDO implementation on the other. As mentioned in Section 3.1, the JDOSecureHelper.getPersistenceManagerFactory() method returns a dynamic proxy instance of the PersistenceManagerFactory class. This would allow manipulating the getPersistenceManager() method to return a proxy instead of the original PersistenceManager instance. Finally, it becomes possible to validate permissions in the associated InvocationHandler instance, before a method call is forwarded to the PersistenceManager. The following paragraph will explain the architecture more in 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 second proxy, in this case a proxy of the PersistenceManager instance. JDOSecure uses the associated PMFInvocationHandler to manipulate method calls directed to the PersistenceManager. Thus, the PMFInvocationHandler represents the entry-point in order to implement the authorization function and to determine whether or not a user is allowed to invoke a PersistenceManager method.

### 3.3. The authorization process

JDOSecure enables the set-up of user specific permissions in order to allow or disallow the invocation of PersistenceManager methods. As already mentioned, a user receives a proxy of a PersistenceManager instance (PMProxy) by invoking the getPersistenceManager() method. Thus, JDOSecure is able to use the assigned PMFInvocationHandler to validate, if an authenticated JDOUser has the permission to make a specific method invocation. The permissions are managed by separate users, roles, and permissions management system (see Section 3.5). JDOSecure distinguishes between different permissions (Table 1) in order to restrict the access to the different PersistenceManager methods. JDOSecure also enables the limitation of user permissions to packages, classes or even objects.

As an example, a user who should be able to invoke the makePersistent() method for a package org.test.sample has to obtain the following permission: JDOMakePersistentPermission "org.test.sample.*".

In order to validate whether 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.
Table 1

<table>
<thead>
<tr>
<th>JDOSecure permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods of a PersistenceManager, that require specific permissions to be executed in the context of JDOSecure:</td>
</tr>
<tr>
<td>Necessary permission to invoke the according method for a specific class or package:</td>
</tr>
<tr>
<td>makePersistent(...) makePersistentAll(...)</td>
</tr>
<tr>
<td>deletePersistent(...) deletePersistentAll(...)</td>
</tr>
<tr>
<td>getExtent(...) Query.execute(...)</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>

as part of the Java 2 Security Architecture. If a user has the appropriate permission to invoke a specific PersistenceManager method, the method call is forwarded to the original PersistenceManager instance. If not, a Java SecurityException will be thrown and the access to the JDO resource is rejected.

JDOSecure allows the assignment of permissions not only for packages or classes but also for individual objects. Since such objects will be constructed at application run-time, JDOSecure can't manage the handling of object permissions exclusively by itself. Therefore, the collaboration between a JDO application and JDOSecure is necessary in order to benefit from object permissions. JDOSecure provides a setObjectPermission(Principal, Class, Object-ID) method that allows a JDO application to assign an object permission to a user or role. In order to prevent an unauthorized use of this method, JDOSecure checks whether an application has a SetObjectOwnerPermission for the specific class. With the help of a AllowJDOObjectPermission <Class> <Action>, the security administrator could define which permissions are being implied by holding an object permission for a specific class. Therefore it becomes possible e.g. to allow a user to update attributes of a specific object but without deleting the object itself. Finally, the PMInvocationHandler checks if a JDOObjectPermission <Class> exists, before method invocations are forwarded to the PersistenceManager.

Although the JDOSecure authorization mechanism allows to restrict the creation, query and deletion of PersistentCapable instances, it is not suitable for the JDO update process. This problem will be addressed in the next section.

3.4. 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 above described security mechanism 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 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 associatedStateManager instance will be notified. This StateManager is responsible for updating the attributes in the corresponding PersistentCapable instance as well as to propagate these updates to the database.

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

In order 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 order to improve clarity. However, JDOSecure enables the access control of the JDO update mechanism by introducing another proxy and a JDOUpdatePermission. As all other
3.5. The management of users, roles, and permissions in JDOSecure

The JDOSecure users, roles, and permissions management system allows to store the information which is necessary for authentication and authorization in a separate JDO resource. The interaction between the management component and JDOSecure is illustrated in Fig. 5. The left part of this illustration corresponds to Fig. 1. The right part describes the schematic architecture of the users, roles, and permissions management system consisting of the administration utility, the users, roles, and permission management component, and a JDO-Implementation to access a separated JDO resource in order to store the authentication and authorization data.

As pictured in Fig. 2, the JDOCallbackHandler passes the user identification and the password to the JDOLoginModule. If a Credential object in the JDO resource matches the same user identification and password, the authentication process was completed successfully. In this case, the JDOLoginModule passes all Principal objects that are associated with the Credential object to the LoginContext. Finally, JAAS adds these Principal objects to the current session (Subject).

In order to enable the mapping of the authentication and authorization information to any arbitrary JDO resource, the users, roles, and permission management system implements two interfaces: The JDOLoginModule class implements the Pluggable Authentication Modules (PAM) standard and enables the access to the authentication data like username and password. The JDOPolicy class extends the default Java Policy implementation and grants access to the JDO resource. For more details about the users, roles, and permission management system we refer to [14].

The Java-based administration utility is introduced for the purpose of simplifying the authentication and authorization information. It is intended to be used by a security administrator, from a local or even a remote computer by using a JDO connection. The administration utility provides a Role-Based Access Control (RBAC) mode (cf. [15]), which allows the assignment of individual permissions for a role or a single user. Fig. 6 shows the permissions granted to the “Internet” user. As it is described in Section 4.3, within these permissions the user is able to store every PersistentCapable object but only to query for example.Car and example.Reservation instances.

4. The car rental demonstration application

In this section, we will demonstrate the benefits of JDOSecure based on a simple web application for a car rental company. The intention of the application is not to provide a full-fledged car rental system that supports customer
or billing management. The motivation behind the development of this application is mainly to illustrate potential impacts caused by the security flaws of the JDO architecture.

The application is implemented by using the Java Servlet technology and JDO as persistence layer. The web application is available at http://projekt-jdo.uni-mannheim.de/demo/ in two different versions. The first version implements JDO directly without any security enhancements whereas the second version uses JDOSecure to prevent security and vulnerability issues. In Section 4.1, the general details about the web application are introduced. Section 4.2 identifies a major security threat in the first version of the web application. The last section will give an answer to the question, on how JDOSecure could prevent such security problems in general.

4.1. About the demo application

The car rental application is intended to serve as proof-of-concept for JDOSecure. When visiting https://projekt-jdo.uni-mannheim.de/car/ or the secure version at https://projekt-jdo.uni-mannheim.de/carsecure/, the rental car welcome site guides a customer to search for available cars. After entering pick-up date, return date and a preferred vehicle type, the web site presents an overview of all available cars matching these criteria (cf. Fig. 7). A table lists these cars including some useful information like car type, price per day, and a link to receive more details about a specific car.
When clicking on a “Car Information Detail” link a popup window appears, which displays additional information, e.g. if the equipment of the car comprises a radio, a navigation system, or air conditioning. After pressing the “next” button, the customer is asked to recheck his/her settings. By confirming the reservation, the customer is asked to enter some personal information and a valid credit card number. When completing the reservation, the customer receives a unique reservation number.

The car rental application consists of four Java classes representing the car rental domain (cf. Fig. 8): In this model, a Customer has to use his/her CreditCard to make a Reservation. A Reservation is always associated within exactly one vehicle (Car) and is initiated by one Customer.

The business logic was developed as a straightforward web application based on the Java Servlet technology. A Java Servlet receives browser requests and sends a response back to the client. In this scenario, the business logic consists of five Java Servlets, which all use a HttpSession in order to identify a customer and to store the arguments being entered by the user. Each web site a user can visit is represented by another Java Servlet class.

The persistence mechanism is implemented by using the Java Data Objects-Specification. Since JDO proposes the concept of transparent persistence, it allows persisting pure “Plain Old Java Objects” (POJOs). After preparing this classes by the usage of a JDO enhancer, the classes of the car rental domain could be stored directly by using the JDO-API.

4.2. Security issues in the first example

When analyzing the mechanism behind the “Car Information Detail” link, it turns out that the arguments (e.g. like the car-ID) are passed through the Servlet by attaching the information to the end of the URL after a question mark. For example, the URL https://projekt-jdo.uni-mannheim.de/car/showdetails?type=Car&filter=carID&value=6 is mapped to the ShowDetails Servlet and passes three arguments (type = “Car”, filter = “carID”, and value = “6”) to this class. Fig. 9 shows the Servlet results.

Even the web site appears rather inconspicuous; it turns out that the ShowDetails class is implemented in a generic way. It allows receiving detailed information not only for a car but also for different classes (e.g. to receive detailed information about a motorbike). Listing 1 presents a code snipped of the ShowDetails class. It illustrates the “generic” use of JDOQL to receive a Collection of persistent objects matching the criteria expressed by the given arguments. In the example discussed above, the Collection would consist of all persistent instances of type example.Car where the expression CarID="6" is true. In order to guard against misunderstandings, the collection in this case would only consist of one object.

Even this functionality appears as a highly flexible and sophisticated approach at first glance, it unfortunately poses a major security threat. When manipulating the arguments attached to the URL listed above e.g. by changing “Car” to “CreditCard” the security problem becomes obvious. After a few tries, someone can find the appropriate arguments to receive credit card information for maybe illegal use: https://projekt-jdo.uni-mannheim.de/car/showdetails?type=CreditCard&filter=creditCardType&value=visa. Fig. 10 shows some credit card information as a result of the presented URL.
Even if it is simply possible to rewrite the application code of the `ShowDetails` class to prevent the undesirable access to the credit card information, it demonstrates unequivocally the security flaws of the JDO architecture. In the next section, we will give an answer to the question, on how JDOSecure could prevent such security and vulnerability problems in general.

4.3. How JDOSecure prevent adequate security and vulnerability issues

JDOSecure allows controlling access to store, query, update and delete persistent objects when using the JDO-API. It introduces a fine-grained access control mechanism to the JDO persistence layer and allows the definition
of role-based permissions. For the second version of the car rental application, we have used JDOSecure to define a “webuser” role that obtains the permissions listed in Table 2.

Because we encapsulated the construction of the PersistenceManagerFactory instance in a separate method, only the modification of a single source code line is necessary to activate JDOSecure: Just the statement to construct a PersistenceManagerFactory has to be replaced in the source code by this line JDOSecureHelper.getPersistenceManagerFactory(props). Nevertheless, the adjustment of several path and library settings as well as the set-up of different JDOSecure configuration files are still necessary. For more information about how JDOSecure could be used to introduce role-based permissions in JDO-based applications, we refer to [16].

When exploring the “credit card” link as presented above in the context of the second application, the benefits of JDOSecure will become obvious. By following the https://projekt-jdo.uni-mannheim.de/carsecure/showdetails?type=CreditCard&filter=creditCardType&value=visa link, a user will receive an Access denied message, because JDOSecure prevents data store access to CreditCard instances for the authenticated webuser (see Fig. 11).

To provide a more adequate error message instead of printing the java.security.AccessControlException in case of a missing permission on the screen, the web application catches these security Exceptions by the use of simple try/catch blocks. However, the error messages are also available in the appropriate Servlet container log files. Listing 2 shows the interesting part of the catalina.out tomcat log file, that is used in this example.

```
... INFO: Server startup in 6257 ms
Initializing JDOPolicyOX...
JDOSecure: Authentication succeeded for user 'webuser'.
authentication succeeded
Authenticated user has the following Principals:
'JDOUser: webuser'
java.security.AccessControlException: access denied
(de.unimannheim.wifo.jdosecure.permissions.JDOQueryPermission example.CreditCard)
...
```

Listing 2. Tomcat Logfile Snipped.

5. Conclusion

In this article, the JDOSecure architecture 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. The permissions could be defined individually for every user/role with regards to certain operations (create, delete, update, and query) and a specific class/package or an object. JDOSecure comprises a management solution for users, roles, and permissions and allows storing the authentication and authorization information in any
Table 2
Defined JDOSecure permissions

<table>
<thead>
<tr>
<th>Permission</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDOMakePersistentPermission</td>
<td>*</td>
</tr>
<tr>
<td>JDOQueryPermission</td>
<td>example.Car</td>
</tr>
<tr>
<td>JDOQueryPermission</td>
<td>example.Reservation</td>
</tr>
</tbody>
</table>

arbitrary JDO resource. Furthermore, a Java-based administration utility with a graphical user interface simplifies the maintenance of security privileges and permissions.

In order to emphasize the benefits of JDOSecure, a simple web application for a car rental company was implemented as a proof-of-concept. The application offers the online reservation of a rental car by entering personal information and a valid credit card number. By analyzing the application, it turns out that a manipulated URL could be used to access credit card information. Even though this is the result of inadequate programming, it unequivocally demonstrates the security flaws of the JDO architecture. JDOSecure could be easily adapted for the use in JDO-based application to close this security gap. It allows restricting access to a JDO resource by defining individual permissions for users and their roles. In this example, it could contribute to preventing access to credit card information.

Even JDOSecure could improve the security of JDO applications, one potential shortcoming of JDOSecure should also be mentioned. Since JDOSecure provides a fine-grained access control mechanism, it becomes obvious that the management of permissions and the access control mechanism has negative side-effects on performance. Even worse, the dynamic proxy approach including a huge number of indirections between the constructed instances and their proxies leads to another deterioration of performance. In order to get a first impression of the performance behavior, we have suggested a test scenario that covers the measurement of CRUD operations. Each test case has been executed several times with a set of different number of persistent objects. It turns out that JDOSecure reduces the performance in this test scenario at about 15%–20% on an average. In the future, we aim to extend our performance tests and are confident of achieving a slightly better performance for JDOSecure.

References