

# Leveraging Oracle Database Security with J2EE Container Managed Persistence

*An Oracle White Paper*  
*November 2003*

# Leveraging Oracle Database Security with J2EE Container Managed Persistence

Introduction .....	3
Mixing J2EE Applications with the Oracle database.....	3
Identity-based Security .....	4
VPD .....	4
J2EE CMP .....	5
Connection Pools.....	5
The Problem.....	6
Solution – identity propagation.....	6
Database Security.....	6
Application Contexts .....	7
Default Context USERENV .....	8
Client Identifier.....	8
Securing Client Identifiers.....	9
VPD Example .....	10
Setting Up VPD.....	10
VPD Code.....	11
VPD Test.....	12
VPD Performance.....	14
Auditing.....	16
Defense in Depth.....	18
J2EE .....	18
Setting the Client Identifier .....	18
Controlling Database Sessions with JDBC Connection Pools .....	19
The J2EE Transaction Trick .....	20
J2EE Transactions Primer, Example #1.....	20
Auto Commit.....	22
Transaction Boundaries with CMP Entity Beans, Example #2 .....	22
findAll().....	24
Testing example #2 – Transactions and entity bean access.....	26
Test Results .....	27
Using Session Beans to Manage Transactions and Entity Bean Access, Example #3 .....	27
Bridging the User Identity to the EJB Container .....	28
EJB Session example #3 sample code .....	28
Conclusion.....	30

# Leveraging Oracle Database Security with J2EE Container Managed Persistence

## INTRODUCTION

Oracle Corporation has invested over 25 years in the development of security technologies for its flagship product – the Oracle Database. Many of these technologies are unparalleled in the industry, and all were developed to meet real world customer requirements. The motivation for these technologies was two fold. The first and primary motivator was to ensure that the security of information in the database was strictly and consistently enforced. Second, but certainly not least, was the goal of easing application development efforts in the areas of security.

As we look at J2EE application development, Container Managed Persistence (CMP) has been met with much popularity. This model for ensuring data is stored and retrieved accurately and efficiently greatly simplifies and shortens the development time for J2EE applications. The CMP model attempts to hide the database complexities from developers. Unfortunately, this model can also render ineffective the abilities of the database to employ its security features.

In this paper we will explore the reasons for this. More importantly, we will show how to use the CMP model for J2EE application development while also utilizing database security. We will focus on two specific database security technologies: Virtual Private Database (VPD), and Auditing. These technologies are simply illustrative and do not exclude the use of other database security technologies. The objective is to show that application developers can take advantage of both CMP and the proven database security features.

## MIXING J2EE APPLICATIONS WITH THE ORACLE DATABASE

Our problem, as outlined above, is simply that CMP, by its database independence nature, does not allow an Oracle database to exploit its security capabilities. While this is inconvenient, it is not insurmountable.

We will identify the problem, and then show how CMP can be coerced into assisting the database in implementing its security functions. We will employ a simple VPD policy and exercise fine-grained auditing to illustrate our solution.

The first part of this paper reviews the relevant database security capabilities. The second part of the paper explains how CMP can be used in conjunction with the security capabilities just discussed.

## Identity-based Security

Much of security is based on one simple principle: Users have to be identified and distinguished from each other. This is true for applications and for databases. This security tenet is exemplified in our world every day. Email applications show only the emails for the user who has logged in. Financial reports are delivered only to those authorized to receive them. For these applications to work effectively, you have to know who the person is before you can know if they are supposed to access an email or read a financial report.

The database is no exception to this rule. Many of the Oracle database security capabilities, such as VPD, allow or prevent users from accessing specific tables, data records, and even columns within the database. Most often, the decision to grant or deny access to information is based on contextual information about the user. For example, the organization a user belongs to, their level in the organization such as individual or manager, or at the very least, the identity of the person all help to determine their individual access rights and privileges.

In this paper we will build a simple VPD policy. For simplicity, the policy will be based only on the user's identity. We'll see later how the application will provide the information necessary for our CMP applications to utilize the VPD policy.

### VPD

VPD is a database feature that provides row-level security. It does this by applying a security policy to tables or views. The tables or views are registered as VPD enhanced tables via the DBMS\_RLS package. When a DML operation occurs on one of these registered tables, the database will invoke the PL/SQL function defined as part of the registration process in the DBMS\_RLS package. The DML types – SELECT, INSERT, UPDATE, DELETE – are specified in the DBMS\_RLS package. Two practical virtues of VPD are a different policy can be used for each DML type, and multiple policies can be applied to the same table or view.

The PL/SQL program's job is to return a predicate or a *where* clause. In effect, the original query is modified, the predicate attached, and then the query is executed. For example, a simple query `select * from EMP` might be augmented by a VPD policy that returns the predicate `ename = USER`. The effective SQL that is now executed, due to the existence of this VPD, is `select * from EMP where ename = USER`.

The security derives from the fact that the predicates are used to restrict records returned by the original query. This whole process is transparent to the application originally issuing the query. This record filtering provides consistent row-level security.

**One of the strongest arguments for VPD is that the security is tightly fixed to the data it protects - it is consistent, centrally managed, and it cannot be by-passed.**

To understand why this is desirable, let's look at an alternative which requires the applications to implement the record filtering. A particular challenge arises when the same data is required by multiple applications. In this case, the security about the data has to be replicated to all the applications. Varying programming languages, COTS applications, and design models often make this an arduous job at best. By using features like VPD, we can have the database itself control the security policies and thus any application using the data would have the security policies automatically applied. There are many advantages to VPD including the following:

- There's no way to get around it, as the database itself controls the security,
- Security is consistent and predictable. As programming paradigms change, the security remains the same. Next week when you add wireless access to the data, no reimplementations are required.
- Security is centrally managed. If there is an update to the policy, a single change can be done in a single place.

The actual PL/SQL implementation which enforces the VPD can be based on anything – IP address, time of day, or whatever is relevant. One consistent element commonly found is the user's identity.

### **J2EE CMP**

Within J2EE application development, the CMP model allows the J2EE container (application server) to manage all database access. Using the CMP model, a J2EE developer will create entities, and the container will automatically manage instances of these entities. The container persists the data by manipulating the actual database tables. Subsequent access to the data is done by the J2EE methods, such as the `findBy<attribute>` method, which the J2EE container translates into SQL queries. The container management includes not only all read access but also the ability to modify entities.

The container maintains the mappings between Java entities and the database records. Each entity usually maps to one or more rows in relational tables. Overall, the database schema design and associated complexities are hidden from the developer.

There usually are many different users simultaneously accessing these entities. As such, the container manages all access; it caches common entries that would be shared across users as well as mediates access to the entities themselves.

### **Connection Pools**

Another critical element in the CMP-database dynamic is understanding how CMP uses connection pools. The CMP model typically creates a connection pool to the database using a generic database account. The connection pool, literally a pool of pre-established database connections, links the application to a database schema. The application will open multiple connections to the same database schema. This

is done to enhance performance in a multi-user environment. Concurrent requests can be handled simultaneously by granting each request one of the connections from the connection pool. We'll return to more on the specifics of connection pools in the J2EE sections below.

### **The Problem**

In the process of fulfilling its container responsibilities, the container has not provided any end-user identifying information to the database. That is, from the database end, there appears to be a lot of requests coming from a single user; albeit that user has several open sessions at any one time.

As stated above, the database needs information about the user for it to provide security. The J2EE application or container must therefore pass this user identifying information along to the database.

### **SOLUTION – IDENTITY PROPAGATION**

Our solution is multi-faceted. We will see how to pass user information from the application to the database while still leveraging J2EE CMP and connection pools. Once there, the database will utilize this information to restrict user access and seed the audit trail with the actual end-user's identity. We'll unravel this solution from the inside-out. That is, we will explain how the database security works first, and then discuss the CMP-database dynamic.

### **Database Security**

For many database applications, especially client/server database applications, the user's identity can be easily obtained by the database. This is based on the assumption that each user has a distinct database account. When the user logs in, they provide a unique username (and some sort of authenticator such as a password) and are subsequently connected to their private account or database schema. The schemas are not shared amongst users. User identity is not a problem.

For connection pooled applications, the user identity, as seen by the database, is a problem because the identity is always the same. Oracle recognized this some time ago and developed a capability to allow applications to tell the database the identity of the real end user even though all end users are actually operating in the same schema. The feature, sometimes referred to as either Application User Proxy or Client Identifiers, is built upon a technology framework called application contexts. We will see how the Client Identifiers (Application User Proxy) feature works after we first look at application contexts.

**Application Contexts are quick and efficient ways to tell the database information about a user.**

### Application Contexts

The Oracle database allows users to create application contexts. These user defined application contexts are name-value pairs, held in memory that can be set for individual database sessions. The application context can only be manipulated by a single PL/SQL program which is specified when you create the application context. For example, to create a context called “CMP\_EX”, one would issue the following:

```
create context CMP_EX using APP_USER.CTX_MGR;
```

This creates an application context with a namespace of “CMP\_EX”. The only program authorized to manipulate values within this application context is the program CTX\_MGR which resides in the APP\_USER schema. Access to the PL/SQL program provides the first level of security. The second level of security comes from the implementation code itself which generally performs checks and validations prior to setting the application context. Note, you need the privilege CREATE ANY CONTEXT to execute the above SQL.

The values in the application context are set by invoking the DBMS\_SESSION.SET\_CONTEXT procedure while inside the PL/SQL program registered to maintain the specific application context. Within an application context, developers can create the name-value pairs relevant to their application(s). A simple example of how to do this is illustrated here. The CTX\_MGR procedure takes a parameter which it will use to set an attribute called “userInfo”.

```
app_user@KNOX10g> CREATE OR REPLACE PROCEDURE CTX_MGR (p_user
2 AS
3 BEGIN
4     DBMS_SESSION.set_context (namespace => 'CMP_EX',
5                               attribute => 'userInfo',
6                               value      => p_user);
7 END;
8 /
```

Referencing values after they have been set is easily done via the SYS\_CONTEXT database function. The following example shows how to set and retrieve values for the CMP\_EX application context. We first have to execute the procedure as the initial value for the attribute is null. We will set the user information to be the username of “BLAKE”;

```
app_user@KNOX10g> execute CTX_MGR('BLAKE')
```

PL/SQL procedure successfully completed.

```
app_user@KNOX10g> select sys_context('CMP_EX','userInfo') from
dual;
```

```
SYS_CONTEXT('CMP_EX','USERINFO')
```

```
-----
BLAKE
```

Typically, application contexts hold several attributes such as name, organization, role, title, etc. It is the combination of all of these things which sets the security context for the user. That is, the VPD policy will most likely be based on something other than just the user's name.

Application contexts are ideal for performing security functions. They are secure because they are private for the session. They are fast because the values are stored in memory. They are flexible because they can be set to any user defined string.

### **Default Context USERENV**

Oracle provides a default application context. It has the namespace of "USERENV". Most of the attributes are predefined by the database. These attributes are very useful in providing information about the client connections. For example, the SESSION\_USER attribute indicates who logged in to the database (similar to the USER function).

There are many other useful attributes that help provide security related information such as the client's IP Address, protocol used to connect to the database, and even how the user authenticated. One attribute not directly set by the database is also one of significance to this discussion – the Client Identifier. This attribute is the basis of the Application User Proxy model referenced earlier.

### **Client Identifier**

The Client Identifier is an attribute in the USERENV namespace that can be set by calling the DBMS\_SESSION.SET\_IDENTIFIER procedure. Its value to us is that it is the only application context attribute that developers can manipulate that will also be picked up in the database audit trails. We'll show this in our auditing example.

Setting and retrieving the Client Identifier is quite simple. We simply pass a varchar2 string to the SET\_IDENTIFIER procedure. The string can be anything, but is usually some useful identifying piece of information about the user such as their username.

In the following example, we'll slightly augment our application context scenario used earlier by adding an additional piece of information. The Client Identifier will be set to the name of a user concatenated with their IP address. By default, IP Addresses is not audited. The IP address is provided to us from the USERENV application context. The point here is that any information about the user can be used. The more information you provide from the application, the more information the database has when applying security. Concatenating several attributes in the Client Identifier is one way to achieve this.

```
scott@KNOX10g> begin
  2  DBMS_SESSION.SET_IDENTIFIER( USER || ':' ||
  3      sys_context('userenv', 'ip_address') );
  4  end;
  5  /
```

**Client Identifiers allow applications to convey additional information for auditing purposes. User defined application contexts are not audited.**

Note in the example above, the SET\_IDENTIFIER procedure is being called by SCOTT. Any database user can set their Client Identifier by executing the SET\_IDENTIFIER procedure. We'll address the security ramifications of this below.

Just as in user defined application contexts, retrieving the value of the Client Identifier is done by querying the SYS\_CONTEXT function.

```
select sys_context('userenv','client_identifier')
       2      from dual;
```

```
SYS_CONTEXT('USERENV','CLIENT_IDENTIFIER')
```

```
-----  
SCOTT:138.1.114.230
```

The above example, while illustrative, does not solve the problem presented by J2EE applications. These applications maintain a pool of connections. Each connection is to the same database schema. The above example would therefore add little value as the user's identity is being derived by the database. Since we can't rely on the current user as seen by the database, the application itself needs to propagate the identity of the actual user to the database for this particular database session. We'll see how to correctly populate the Client Identifier in the J2EE code example.

We will use the Client Identifier as part of our VPD implementation, but we cannot rely on just this value alone for our record-level security.

### Securing Client Identifiers

One particular challenge with Client Identifiers is that they can be set by anyone to anything. Recall that setting the Client Identifier is done by invoking a procedure in the DBMS\_SESSION PL/SQL package. The privilege to execute the DBMS\_SESSION package has been granted directly to PUBLIC making it accessible to anyone in the database. Revoking that privilege is generally considered unsupported as it is considered "tampering" with the Oracle internal mechanisms. Therefore, we do not and should not base security on this value alone.

One option is to not use the Client Identifier, and simply create our own application context. In an application context, the values can only be set by a specific PL/SQL program. Only the user(s) with execute privileges on that specific PL/SQL program will be able to set the values. Another advantage is the PL/SQL program can do additional checks prior to setting any application context values. For example, checking for valid IP Addresses, the authentication type, and the time of day are often good additional security precautions. As such, the value returned by the application context is considered secure.

There is a drawback to this approach. While this is sufficient for VPD, the application context values are not audited; only the Client Identifier is audited. One work around for this, and a preferred solution, is to use both a Client Identifier and a user defined application context. The solution would entail setting the Client

**Client Identifiers can be set by anyone to anything. As such, security should not be based on the Client Identifier alone.**

Identifier to the same (or a similar) value as one in the user defined application context. The VPD procedure would then verify that these values are congruent. The actual VPD policy and rows returned would be based on the application context values. This design would then allow the benefits of using application contexts while simultaneously allowing for effective auditing.

As an alternative (and for simplification reasons), we will combine the Client Identifier with something else that cannot be manipulated by users. In the following VPD example, we will use the schema name used by our connection pool. The basis for this choice is that the schema is protected by, at the very least, a strong password. Since access to the schema is controlled and secure, the combination of schema name and Client Identifier is more secure than just using the Client Identifier alone, and subsequently will form the basis of our VPD policy.

### VPD Example

We now have the background to understand our VPD example. We will create a simple VPD policy on SCOTT's EMP table. The policy enforces a requirement that users can see all records except in the case they ask for the salary (i.e., the SAL column). If a user queries the salary, the VPD policy will be enforced, and the users will only be able to see their record. That is, users can only see their salary.

This is a new Oracle Database 10g feature. It is called Column Sensitive VPD. However, this example is still relevant to pre-Oracle Database 10g. The only difference is prior to Oracle Database 10g, the policy would always be enforced regardless of what columns are selected. That is, the policy would say that users can only see their record.

### Setting Up VPD

To create a VPD policy, we have to execute the DBMS\_RLS.ADD\_POLICY procedure. We add the policy as the SYSTEM user because the privilege to execute the DBMS\_RLS package has not been granted to PUBLIC. The policy will be in effect on all queries and updates. After running the following statement, the database will call the PRED\_FUNCTION in the SCOTT schema whenever someone queries or updates the SAL column of the SCOTT.EMP table.

```
system@KNOX10g> begin
 2   DBMS_RLS.ADD_POLICY ( OBJECT_SCHEMA => 'SCOTT',
 3                       OBJECT_NAME   => 'EMP',
 4                       POLICY_NAME   => 'CMP_EX',
 5                       FUNCTION_SCHEMA => 'SCOTT',
 6                       POLICY_FUNCTION => 'PRED_FUNCTION',
 7                       SEC_RELEVANT_COLS => 'SAL',
 8                       STATEMENT_TYPES => 'SELECT,UPDATE' );
 9   end;
10  /
```

## VPD Code

For our VPD implementation code, we will base the identity of the user on the Client Identifier. Since the Client Identifier can be modified by anyone, we need to do an additional check. Our implementation will verify that one of the following two things is true:

1. The query is being submitted from the APP\_USER schema or,
2. The user executing the query is the same user defined by the Client Identifier.

The APP\_USER schema is the schema to which our J2EE CMP connection pool will attach. Since it is secured by a strong password, we feel confident that the Client Identifier value is accurate. Only the J2EE application can connect to the APP\_USER schema, so, only the J2EE application can set the Client Identifier for each APP\_USER session.

We allow the second condition for the situations where users are directly connected to the database. Users may or may not have a direct connection to the database. If they do not (or should not) have direct database access, this second condition may not be necessary.

(An additional check or alternative would be to verify that the Client Identifier value is congruent with a secure application context. For brevity and simplicity, this is not shown.)

Note this design implies a shared trust between the database and the application. This is absolutely necessary in almost all situations. It is not 100% trust, but it will require some trust. If you don't trust your applications, at least a little, then don't let them connect to your database! Otherwise, you will need to use a limited trust model similar to the one illustrated here.

In our sample VPD code, we create two local variables. The first is set to the Client Identifier. If the value has not been set or is otherwise NULL, we give it a value. We do this because NULL comparisons are undefined. In efforts to keep the code more readable, the logged in user is also set in a local variable. The logged in user is determined by the value returned from the function

`sys_context('userenv', 'session_user')`. Here is our VPD code:

```
scott@KNOX10g> CREATE OR REPLACE function pred_function
2         ( p_schema in varchar2 default NULL,
3         p_object in varchar2 default NULL)
4 RETURN varchar2
5 AS
6 l_client_id varchar2(30) :=
7   nvl(sys_context('userenv', 'client_identifier'),
8       ' NULL');
9 l_user varchar2(30)      :=
10    sys_context('userenv', 'session_user');
11 BEGIN
12   IF ( (l_user = 'APP_USER') OR
13       (l_user = upper(l_client_id))) THEN
14     return
```

```

15         'ename = ' ||
16         'sys_context(''userenv'', 'client_identifier')';
17     ELSE
18         return '1=0'; -- no rows returned
19     END IF;
20 END;
21 /

```

Function created.

This function returns one of two strings. It returns the “ename = sys\_context ..” string if our security checks out i.e., either of the two conditions described above is true. This will restrict rows in the result set to be only those where the ENAME value is equal to the value stored in the Client Identifier. As a result, the following SQL:

```
select sal from emp
```

will effectively transformed into

```
select * from emp
  where ename = sys_context('userenv', 'client_identifier')
```

The other possible return string is “1=0”. This is returned when the security requirements have not been met. This will eliminate all rows from the result set i.e., no rows will be returned.

#### VPD Test

A couple of quick tests are helpful in showing the effect of our VPD policy. The PRED\_FUNCTION (PL/SQL) function’s parameters are defaulted to NULL intentionally. This trick allows us to view the output from our logic without actually invoking the VPD engine. The string returned is ultimately is appended to the SQL statements. An initial query of the function shows that no rows are returned.

```
scott@KNOX10g> select pred_function from dual;
PRED_FUNCTION
-----
1=0

```

This is because the Client Identifier was not set. A query now would result in no rows returned. We set our Client Identifier, and then verify the function works as desired.

```
scott@KNOX10g> exec DBMS_SESSION.SET_IDENTIFIER(USER);
PL/SQL procedure successfully completed.
scott@KNOX10g> select pred_function from dual;
PRED_FUNCTION
-----
ename = sys_context('userenv', 'client_identifier')

```

Next, we query the EMP table to determine if VPD does in fact do what we want it to do. A simple count of the records shows that we can see all records when not querying on the SAL column.

```
scott@KNOX10g> select count(ename) from emp;

COUNT(ENAME)
-----
              14
```

When we include the SAL column, the VPD policy will be enforced. We see from the results that the policy works per our requirements. SCOTT can only see his salary. Note that there is no predicate in the SQL we submitted. The VPD adds the SQL which in turn limits the result set.

```
scott@KNOX10g> select ename, sal from emp;

ENAME          SAL
-----
SCOTT          3000
```

To really test the policy, we want to try to spoof another user. SCOTT resets his Client Identifier to BLAKE.

```
scott@KNOX10g> exec DBMS_SESSION.SET_IDENTIFIER('BLAKE');

PL/SQL procedure successfully completed.
```

SCOTT then reissues the query trying to see BLAKE's salary.

```
scott@KNOX10g> select ename, sal from emp;

no rows selected
```

Finally, connecting to the APP\_USER schema, we can set the Client Identifier to BLAKE. One important and interesting feature is that failure to set the Client Identifier or setting to something other than a valid name (i.e., a valid value in the ENAME column) results in no rows returned. This is an extra security safeguard as it helps to ensure that the applications or users are following proper procedures. If they forget to set their Client Identifier, then auditing will not be able to capture who they are; that's OK as they will not get any rows returned.

```
app_user@KNOX10g> exec DBMS_SESSION.SET_IDENTIFIER('BLAKE');

PL/SQL procedure successfully completed.
```

Now when we query, the VPD policy will be enforced and will show the records for the user specified in the Client Identifier – BLAKE in this example.

```
app_user@KNOX10g> select ename, sal from SCOTT.EMP;

ENAME          SAL
-----
BLAKE          2850
```

Note that this is the intent of our application design. The application will be connected to the APP\_USER schema, it will set the Client Identifiers to the respective end user, and then issue queries.

The database security (VPD) prevents users connected outside the application from gaining access to private salary data. It also ensures that the application does not have to provide the query modification when the salaries are queried. In other words, we have a consistent, easy to manage, non-bypassable security mechanism.

### **VPD Performance**

A clear and obvious concern when implementing any type of security is performance. Rest assured this architecture is highly performant.

### **Bind Variables**

**Bind variables help to ensure high performance by allowing the database to save valuable computing resources when queries only differ by variable values.**

The first area to investigate is the performance of the returned predicate. Since the actual SQL to be executed includes not only the original SQL, but also the SQL returned from the predicate, we have to ensure that this SQL string performs well. In the above example, we return (assuming everything in nominal) the string “ename = sys\_context ...” The key is the use of sys\_context. The performance comes from the fact that the sys\_context is treated as a bind variable.

Bind variables are the staple of performance in an Oracle database. Bind variables allow the database to reuse SQL between database sessions. That is, the database can share a single parsed plan for multiple open cursors. The performance is achieved because the database does not have to re-parse the SQL. As Oracle database guru, Tom Kyte put it, “Not using bind variables in SQL is like not compiling java code into bytecode every time. Imagine having to compile each java class time you needed to execute the class. Having to parse the SQL each time is exactly the same.”

In our connection pooled architecture, this is invaluable. If we did not use bind variables, but rather returned the actual resolved value, e.g., “ename = 'SCOTT'”, the database would be spending all its time re-parsing the SQL statements.

Note performance as measure here is not based on how the SQL is generated. In our example, the VPD policy is invoked, and the predicate is produced. However, the application could have produced the same or similar SQL. The point is it does not matter how the SQL was generated – if you want to achieve stellar performance, you have to produce good SQL; bind variables are generally a good way to go.

### **Code Location**

Another question on the design revolves around whether the SQL should be modified at the database or at the application. Essentially, from a performance perspective, it does not matter. The same process will have to occur regardless of

where it occurs. That is, some procedural logic will fire, check some things, and then determine how to reform the SQL query thus securing the data for the user.

From a security perspective, the database implementation is much better. It guarantees that the SQL, and thus security, will always be enforced. This has value not only when the data may be needed by other applications, but also helps provide defense in depth in the case that the Web application is successfully attacked. In the later case, the security of the application itself has been compromised, and it is only the database security that will now ensure that an attacker does not get access to unauthorized data.

### **Policy Invocation**

Lastly, a question arises on the performance regarding the time required for the database to invoke VPD a.k.a. “overhead”. Since VPD invokes a function each time a statement or cursor is issued, performance could be a concern.

To help ensure things are running extremely fast, the database allows you to cache the VPD policy. This can be done by setting the `STATIC_POLICY` parameter to `TRUE` in the `DBMS_RLS.ADD_POLICY` procedure. If you declare your policy to be static, the database will cache, on the first execution of the VPD policy, the results from your policy function. This can result in significant performance improvements as the PL/SQL code implementing your VPD policy will not be called in further queries. A simple test illustrates this point.

In this example, we will introduce an artificial latency into our VPD policy. Our policy function will simply sleep for two seconds. (The sleep procedure below is really a simple call to the privileged `DBMS_LOCK.SLEEP` procedure.)

```
scott@KNOX10g> CREATE OR REPLACE function pred_function
 2          ( p_schema in varchar2 default NULL,
 3            p_object in varchar2 default NULL)
 4 RETURN varchar2
 5 AS
 6 BEGIN
 7     sleep(2);
 8     return null;
 9 END;
10 /
```

We update the policy by setting the `STATIC_POLICY` parameter to true.

```
system@KNOX10g> begin
 2     DBMS_RLS.ADD_POLICY ( OBJECT_SCHEMA      => 'SCOTT' ,
 3                           OBJECT_NAME       => 'EMP' ,
 4                           POLICY_NAME       => 'CMP_EX' ,
 5                           FUNCTION_SCHEMA   => 'SCOTT' ,
 6                           POLICY_FUNCTION   => 'PRED_FUNCTION' ,
 7                           STATEMENT_TYPES  => 'SELECT,UPDATE' ,
 8                           STATIC_POLICY     => TRUE);
 9 end;
10 /
```

PL/SQL procedure successfully completed.

Then we set the timing on and issue to queries back to back. By retuning NULL, the policy will allow the user to see all records.

```
scott@KNOX10g> set timing on
scott@KNOX10g> select ename, sal from emp where ename = USER;
```

ENAME	SAL
SCOTT	3000

**Elapsed: 00:00:02.05**

The two seconds was introduced by our PRED\_FUNCTION. But there is only a one time hit. Any subsequent execution by any user will use the cached policy. To illustrate this, we simply re-run the query. Note: the results of this are independent of the user sessions. The policy is cached in the Oracle Shared Global Area.

```
scott@KNOX10g> select ename, sal from emp where ename = USER;
```

ENAME	SAL
SCOTT	3000

**Elapsed: 00:00:00.01**

### **Caching Caution**

A word of caution: static (cached) policies may not prove effective in all situations. The predicate is the same for all sessions of the same user. Once cached, the PL/SQL function implementing the VPD will never be re-executed. There are some situations when this is undesirable. The most obvious is when the predicate should change among sessions of the same user. Note that in our example, the predicate is constant while the value returned by the application context changes. This is very desirable and furthermore allows for a cached VPD policy.

An example of a situation when caching is not desirable is seen with a security policy that is time based. For example, a policy which states records should only be accessed from 9 a.m. to 5 p.m. would not be an effective use of the caching feature.

In most cases, caching should be considered. If you are unsure as to whether caching should be used, enable it and test to ensure that security is working as desired.

### **Auditing**

This exercise is being conducted to illustrate not just how to leverage VPD, but also Oracle's auditing capabilities. Auditing is an important element in security. Unlike VPD and most other security techniques, auditing cannot be used as a preventive tool. It is a detection tool. Its value is nonetheless important.

For this exercise, we will simply enable Oracle's Fine-Grained Auditing (FGA) mechanism. FGA was new in Oracle 9i. It differs from standard auditing in several

**Fine Grained Auditing allows a higher fidelity in the auditing process than what is possible in standard auditing. However, the two can be used in conjunction.**

ways. FGA is usually enabled on events that should not occur. Standard auditing may well be conducted on actions that should occur.

Our audit policy is configured to detect if a user queries another user's salary. This is a departure from standard auditing which does not allow this degree of fidelity. Standard auditing will only allow us to audit SELECT's on the table. We are not interested in this as it will most likely fill up the audit logs with records of people doing what they are supposed to be doing. FGA helps to separate auditing activities from capturing all actions to capturing all unauthorized actions.

Similar to VPD, FGA is enabled by invoking a PL/SQL procedure. The following policy is placed on the EMP table. The AUDIT\_COLUMN is specified as we are only interested in auditing when someone queries salaries. We add the condition to audit only when the user invoking the query is not the one identified by the Client Identifier.

```
system@KNOX10g> begin
 2     DBMS_FGA.ADD_POLICY(object_schema => 'SCOTT',
 3                         object_name   => 'EMP',
 4                         policy_name    => 'CMP_FGA_EX',
 5                         audit_condition => 'USER !=
sys_context('userenv','client_identifier')',
 6                         audit_column   => 'SAL',
 7                         enable        => TRUE);
 8 end;
 9 /
```

To see if this is working, we simply execute our query from the application schema.

```
app_user@KNOX10g> exec DBMS_SESSION.SET_IDENTIFIER('BLAKE');
```

PL/SQL procedure successfully completed.

```
app_user@KNOX10g> select ename, sal from SCOTT.EMP;
```

ENAME	SAL
BLAKE	2850

This will generate an audit as our audit condition was met. The details are located in SYS.DBA\_FGA\_AUDIT\_TRAIL.

```
system@KNOX10g> select db_user, client_id,
 2     sql_text
 3     from sys.dba_fga_audit_trail;
```

DB_USER	CLIENT_ID	SQL_TEXT
APP_USER	BLAKE	select ename, sal from SCOTT.EMP

While this is useful, we probably want to amend the policy to be congruent with our VPD policy. If we don't do this, we end up with audits for every authorized application query. That was not our intention. Here is the amended policy.

```

system@KNOX10g> begin
2     DBMS_FGA.ADD_POLICY(object_schema => 'SCOTT',
3                         object_name   => 'EMP',
4                         policy_name   => 'CMP_FGA_EX',
5                         audit_condition => 'USER !=
''APP_USER'' AND ename != sys_context(''userenv
'', ''client_identifier'')',
6                         audit_column  => 'SAL',
7                         enable       => TRUE);
8 end;
9 /

```

FGA only fires when at least one row is returned. If our preventive security measures are working properly, then nothing will be in the audit trail. This makes it easy to check on the security status of the application. If there are no records in the audit trail, then everything should be working.

### Defense in Depth

Note the relationship between the condition above and the VPD function. The VPD function is supposed to limit the records returned to be those of the users as specified by the Client Identifier. The audit policy is set to fire if for some reason that VPD policy does not do this.

The examples show how two of the many security capabilities can be exercised. An important outcome is that the two technologies can be used independently of each other. One does not require nor obsolete the other.

### J2EE

We're ready to link our Oracle database security with our J2EE applications. In the first part, we will see how to set the Client Identifier for authenticated users. Then, we will see how to have everything work when using CMP.

### Setting the Client Identifier

Our first task will be to set the user's identity. The application server can set this value to anything, but it is normally set to the user who authenticated to the application server. We can retrieve the current user in a J2EE application by using the standard J2EE method `getUserPrincipal()` available in either the `HttpServletRequest` object or the `EJB Context` object.

The following code snippet shows how a servlet would convey the user's identity to the database. Remember that this needs to be done *before* any other database access, either through JDBC or CMP entity beans, to ensure the database security can be employed.

```

String currentUser =
    request.getUserPrincipal().getName();

InitialContext ctx = new InitialContext();
DataSource ds =
    (DataSource)ctx.lookup("java:comp/env/OracleDataBase");

```

```

Connection conn = ds.getConnection();
PreparedStatement pstmt =
    conn.prepareStatement("begin dbms_session.set_identifier(?)
end;");
pstmt.setString(1,currentUser);
pstmt.execute();

pstmt.close();
conn.close();

```

The above code takes a connection from the connection pool, sets the Client Identifier to the user who has authenticated to the application server, then returns the connection to the connection pool. After executing this code, the Client Identifier for this database session is set. The database can now apply security based on this information.

### **Controlling Database Sessions with JDBC Connection Pools**

A small problem exists with JDBC connections and the connection pools. The above example set the Client Identifier for a JDBC connection, and then returned the connection back to the pool thus making it available for the next request (another thread.)

The purpose of a connection pool is to minimize the number of physical database connections. It is generally undesirable if not impossible to establish a private and physical database connection for each end user. Therefore a connection pool is used to share a set of pre-established physical connections amongst many end users.

There is one important challenge to overcome when using connection pools with security. Since many users are sharing connections, we need to ensure exclusive access to a JDBC connection for the life of the “request”. A request generally consists of several operations. Processing any one operation has three basic steps:

1. Getting a JDBC connection from the pool;
2. Performing the operation;
3. Returning the connection to the pool.

This three step process may happen several times during a single request. (When you access an entity bean, this same three step process is performed for you automatically.)

Ensuring that the application gets the same connection for the multiple operations of a request is the problem to solve as it is necessary in ensuring proper user identification and security throughout the entire request.

One common solution to this problem is to hold the JDBC connection until all database processing has been completed for the request, and only then returning it to the pool. This is a workable solution but not when using CMP. When using CMP, the container is responsible for all database access, including obtaining and

managing JDBC connections. There is no obvious and direct way to control the JDBC connection the container uses for CMP operations.

### **The J2EE Transaction Trick**

The answer to this problem involves using J2EE transactions. If we start a transaction before we set the Client Identifier (and before we perform any operations), the J2EE container will ensure that the same JDBC connection will be used for the duration of the transaction. Thus, we can freely open and close connections from the pool, always knowing that the same JDBC database connection will be used.

Since the EJB container uses the same process for retrieving JDBC connections as a manual process, this holds true for all CMP operations. The transaction boundary ensures that we always get the same connection and that no other thread will acquire it while in our transaction. If the application starts a transaction, sets the Client Identifier, manipulates the database (via CMP entity beans or JDBC), and then commits the transaction, it can be guaranteed that the database will know who is doing the work.

There are many ways to use transactions in a J2EE container. The following three examples will show how to use transactions and CMP entity bean access in a J2EE environment. We will see three variations on using J2EE transactions as follows:

1. Utilizing the UserTransaction object when using straight JDBC
2. Using the UserTransaction object with CMP entity beans
3. Starting and ending a transaction declaratively within a session bean which then calls entity beans.

### **J2EE Transactions Primer, Example #1**

We'll start by first understanding how to begin and end a J2EE transactions. The first example illustrates how this is done using pure JDBC. The objective is to prove that utilizing J2EE transactions guarantees re-use of the same JDBC connections across multiple operations. With this, we can re-use database sessions and reap the benefits of the database provided security mechanisms.

According to the J2EE specification, all J2EE containers must publish a UserTransaction object which clients can use to manually control transactions. The example servlet code below shows that within a transaction, two (separate) JDBC connections will actually use the same physical connection. This conclusion is drawn because both JDBC connections can alter the same row. If these were two separate physical connections, the second connection would not be able to update the row until the first connection released the lock via a commit or rollback.

Our first step will be to acquire the UserTransaction object. This code must run in the Web container as a servlet, JSP, bean, etc. It is not applicable when run as a standalone java client.

**The database locks updated rows until the transaction issues a commit or rollback. Different database sessions are unable to update these locked rows.**

```
InitialContext context = new InitialContext();
UserTransaction ut = (UserTransaction)
    context.lookup("java:comp/UserTransaction");
```

To start a transaction, we execute the begin method of the UserTransaction object.

```
ut.begin();
```

We next establish our DataSource. Once we retrieve this, we can obtain the actual JDBC connection from the connection pool. Note that all subsequent JDBC lookups from this same DataSource (connection pool) will return the same JDBC physical connection, because we got the connection while in the transaction. If we did not start the transaction, then subsequent lookups may return a different JDBC connection resulting in a different database session.

```
DataSource ds = (DataSource) context.lookup("jdbc/ScottDS");
Connection conn = ds.getConnection();
```

Next, we execute a simple update statement to the EMP table. The update will cause the database to lock the row. The row will remain locked until the transaction has issued a commit or rollback.

```
PreparedStatement cstmt =
    conn.prepareStatement("update emp set ename='MATT' " +
        "where empno=7369");
cstmt.execute();
```

Note the use of bind variables in SQL statements is strongly suggested. It is not done here for code simplicity and user readability.

We'll next perform a second update. By utilizing the same DataSource identified by the "jdbc/ScottDS", we get a second connection that will in reality be our already established, and identified, database session. We then use the second connection to issue an update to the same row (where ename = 7369) that our first connection updated.

```
DataSource ds2 = (DataSource) context.lookup("jdbc/ScottDS");
Connection conn2 = ds2.getConnection();
PreparedStatement cstmt2 =
    conn2.prepareStatement("update emp set JOB='ANALYST' " +
        "where empno=7369");
cstmt2.execute();
```

For the second update to work without being locked, both connections will have to be using the same physical database connection. Finally, we close our connections and commit our transaction which signals the end of our transaction statements.

```
conn2.close();
conn.close();
ut.commit();
```

### **Auto Commit**

To examine what would happen if there were no transaction boundaries defined, let's remove the `UserTransaction` begin and commit statements. By default, JDBC connections are set to auto commit, so the above example would be OK because after the first update, the JDBC drivers would automatically commit, thus releasing the lock on the row (and invalidating our example proof.) One might question then if the example above works because it is the same JDBC connection, or if the example works because the first update has auto-committed and released the row lock. It is the former, and the following explains why.

If we were using two separate connections and modified the code to not automatically commit, by adding the lines `conn.setAutoCommit(false)` and `conn2.setAutoCommit(false)`, the second update would stall waiting for the first update to either commit or roll back (remember the `UserTransaction` begin and end are removed).

By utilizing the `UserTransaction` object, the J2EE container automatically sets the JDBC connections' auto commit to false, and manages the commit and rollbacks. In fact, you will receive an error if you try to enable auto commit while in the middle of a transaction. Subsequently, we would expect the second transaction to stall if we were actually using two separate database connections.

When this servlet code is executed, it does not lock. We can thus conclude that we are using the same database session for both JDBC connections. Even though the application has two JDBC connections, in reality, we are only using one physical connection to the database i.e., one database session. That is the key to ensuring we can use the database security features described above.

### **Transaction Boundaries with CMP Entity Beans, Example #2**

Now that we know how to manage transactions using the `UserTransaction` object, let's expand our example to include accessing (finds, gets and sets) an entity bean. Assume we have created an entity bean on the EMP table. Details on how to do this are available from the Oracle 9iAS product documentation, the Oracle JDeveloper documentation or the J2EE 1.3 specification.

The steps that a client will perform to access entity beans will be as follows:

- Start the transaction
- Set the Client Identifier
- Select entities using a finder method
- Get and Set entity attributes
- Close the transaction

The `findAll()`, `get<attribute>` and `set<attribute>` are being used here to illustrate the use of multiple CMP operations as part of the same request.

The following servlet code snippet illustrates how we can apply what we have just learned to access entity beans in a transactional setting. We will start by retrieving the user's identity and pass it to the database via the Client Identifier. Before we do that, we have to start a transaction so as to guarantee that we will be bound to the same database session. Java comments are added in line to aid in understanding.

```
// get the user's identity
String userName = request.getUserPrincipal().getName();

// Lookup the UserTransaction object and start our transaction
InitialContext context = new InitialContext();
UserTransaction ut = (UserTransaction)
    context.lookup("java:comp/UserTransaction");
ut.begin();

// get JDBC connection from the pooled datasource
DataSource ds = (DataSource) context.lookup("jdbc/scottDS");
Connection conn = ds.getConnection();

// set the Client Identifier in the database session
PreparedStatement cstmt =
    conn.prepareCall("begin dbms_session.set_identifier(?); end;");
cstmt.setString(1, userName);
cstmt.execute();
conn.close();
```

In this example, we are setting the Client Identifier directly. As indicated earlier, we would probably not be just doing this. We would probably be setting several things in the database simultaneously. As such, we would be calling a custom PL/SQL procedure which would then call the DBMS\_SESSION.SET\_IDENTIFIER procedure, as well as probably initialize an application context.

Our next step will be to retrieve the Entity bean's home interface from the JNDI tree.

```
EmpHome empHome = (EmpHome)
    PortableRemoteObject.narrow(context.lookup("Emp"), EmpHome.class);
```

We next iterate through all the entities counting as we go. We'll print the name and salary for each record. We will give each employee a 10% raise, and then print out the new salary.

```
Collection c = empHome.findAll();
int x=0;
Emp employee=null;
//now we have all the Employee entities.
for (Iterator i = c.iterator(); i.hasNext();)
{
    x++;
    employee = (Emp) i.next();
    out.println("<BR>" + employee.getEname() + ": Current
        sal: " + employee.getSal());
    Double newSal = new Double(employee.getSal().longValue() * 1.1);
    employee.setSal( new Long( newSal.longValue() ));
    out.println(" New sal: " + employee.getSal());
}
```

We commit the transaction thus marking the JDBC connection as “free”, which can then be used by another user (thread.) The final statement indicates how many records the user was able to view and update. If the VPD policy states a user can only see their records, the count should be one. The actual row returned will be different for each user.

```
ut.commit();
out.println("There were "+x+" employee's returned from findAll());
```

If our VPD policy were not in effect, this code would give everybody a 10% raise. Since VPD is restricting the data returned (only the record of the authenticated user is returned), there is only one person getting the raise - i.e., the same person who authenticated. Additionally, if database auditing were enabled, these entity bean actions could be recorded. The audit records would show, among other things, the information set in the Client Identifier.

### **findAll()**

The above example is not yet fully functional. Even though the entity bean's `findAll()` and `set<attribute>()` methods are within the transaction boundaries, these methods will not actually know they are part of a transaction until we explicitly configure the container to allow this.

To understand what is really happening, we'll follow the actions performed by the J2EE container. Let's look at the `findAll()` method, which returns all entities to the caller.

First, the EJB container gets a JDBC connection from the pool. Next it creates a SQL statement based on the primary key of the entity. For example, if the EMP table has a primary key on the EMPNO column, then the SQL statement would be, “select empno from emp”. It executes the SQL getting a `ResultSet` object. The container will then create an entity object for each row in the result set and initialize the values by calling `ejbLoad()`. This populates the entity attributes with the remaining column values for each record. If an entity object already exists in the cache, a new entity will not be created.

**Regardless of entity caching, the SQL will always be executed. This ensures that all result sets are accurate and up-to-date with respect to security.**

After all entities have been created, each entity is added into a `Collection` to be returned. Regardless, an important thing to know is that the container will *always* execute the SQL query to determine which entities, either newly created or cached, are to be put into the returned `Collection`. This is important because our VPD policy might return a different set of rows for the same SQL query. If the container did not always query the database and just assumed the same rows are always returned for the same query, the VPD policies would be useless. Finally, the container will close the JDBC connection and return the `Collection` to the caller.

Note OC4J has an option called “exclusive-write-access” which can be optionally enabled for each CMP entity bean. With this option set to true, the default is false,

the container will not always re-issue queries for cached entities thus preventing the database from exercising security<sup>1</sup>.

For our database security implementation to work properly, we have to guarantee the JDBC connection used by the container for the `findAll()` and `setSal()` methods will be the same JDBC connection (and database session) we used to set the Client Identifier. By modifying the EJB deployment descriptor to include the EJB methods (`findAll`, `setSal`) as part of the transaction, the JDBC connection will be the same one that was used to initially setup the connection.

By default, EJB methods will not use the current transaction unless otherwise configured. To mark these methods to use the current transaction, we need to modify the J2EE `ejb-jar.xml` file to tell the container that these methods must be part of a transaction. Here is the relevant part of the `ejb-jar.xml` file which shows how to mark an entity bean as requiring a transaction:

```
...
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>Emp</ejb-name>
      <method-intf>Home</method-intf>
      <method-name>findAll</method-name>
    </method>
    <trans-attribute>Mandatory</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
      <ejb-name>Emp</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>setSal</method-name>
      <method-params>
        <method-param>java.lang.String</method-param>
      </method-params>
    </method>
    <trans-attribute>Mandatory</trans-attribute>
  </container-transaction>
</assembly-descriptor>
...
```

The two methods, `findAll()` and `setSal()` have been marked as having a mandatory transaction attribute (bolded above). This means that there must be an existing transaction in place before the container can call these two methods. Since we have included these two methods calls in between the `ut.begin()` and `ut.commit()` calls, there is an existing transaction in place. Because of this existing transaction, the J2EE container will use the same JDBC connection to access the database.

This is same behavior as we saw in the earlier example when we had two separate JDBC connections, but in reality, it was a single physical connection. For this to work across a J2EE application, all other relevant methods in the entity bean would have to be marked the same way, i.e., with the mandatory transaction attribute.

---

See the [Oracle9i Application Server Performance Guide](#) for details.

### Testing example #2 – Transactions and entity bean access

Now that we can guarantee database session affinity for CMP, we can test our implementation by using a VPD policy which is based on the user’s identity. The first requirement for testing will be to ensure we know the user’s identity at the application server tier.

### Configuring Web Authentication

We need to configure the web container to authenticate the servlet. By adding the following lines to the web.xml file, we secure the URL context “/” with basic authentication. Therefore, any user accessing our application will be forced to authenticate.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>main</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>users</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>test</realm-name>
</login-config>

<security-role>
  <role-name>my_app_users</role-name>
</security-role>
```

This restricts access to just those users who are members of the role named “my\_app\_users”. There are several ways to assign users to roles in OC4J. We’ll base our example on the default JAZN implementation. To add users to the “my\_app\_users” role, we need to modify the jazn-data.xml file. To execute a test against the database using multiple users, we would have to configure all the end users for the appropriate role membership. The following lines only show the user Matt as a member of the “my\_app\_users” role.

```
<users>
  <user>
    <name>Matt</name>
    <description>the user named Matt</description>
    <credentials>!mypass</credentials>
  </user>
</users>

<roles>
  <role>
    <name>my_app_users</name>
    <members>
      <member>
        <type>user</type>
        <name>Matt</name>
      </member>
    </members>
  </role>
</roles>
```

```
</members>
</role>
</roles>
```

### Test Results

The tests confirm that the database security is in effect when using our transaction trick. The VPD predicate will return records where the ENAME value is equal to the Client Identifier as set by the application. When logging in as SCOTT, the results from the servlet were:

```
SCOTT,42432
There were 1 employee's returned from findAll()
```

When logging in as BLAKE, the results were consistent for what BLAKE should see.

```
BLAKE,42432
There were 1 employee's returned from findAll()
```

And when logging in as GUEST, the results were a bit different.

```
There were 0 employee's returned from findAll()
```

Since there is no record with an ename of "GUEST" in the EMP table, the database returned zero rows.

### Using Session Beans to Manage Transactions and Entity Bean Access, Example #3

This last example shows how you can use a stateless session bean to facilitate all access to the entity beans/database. This is commonly known as a façade pattern. To enable database security, VPD and auditing, we found we had to perform a consistent set of actions. For every database access as part of a request, the client had to perform the following:

- Start a transaction
- Setup the JDBC connection – set Client Identifier to username
- Access the entities (finds, gets and sets)
- Close the transaction (commit or rollback).

By using an EJB session bean to access the entity beans, we can consolidate these operations into a central location. There will be many clients (JSP's, servlets, javabeans, etc.) that will need to access the entities. Instead of each client accessing the entity directly, and performing the four steps above, they will delegate entity access to the session bean. This makes the process more efficient by centralizing entity access to one location, thus making the process easier to code, manage and debug.

Another advantage of using a session bean is the ability to use declarative transactions versus explicitly programming the UserTransaction object. A session bean method can be declaratively marked as part of a transaction in the ejb-jar.xml

file. By creating a method in an EJB session bean to “get all Employees” and marking that method as requiring a transaction, the J2EE container will automatically start a new transaction upon calling that method, and automatically commit the transaction when that method ends. If the session bean throws an exception, the transaction will be rolled back. This means that we will not have to manually use the UserTransaction object and will not have to manually issue the begin(), commit() or rollback() statements.

To mark the session bean method as requiring a transaction, include the following in the ejb-jar.xml file:

```
<container-transaction>
  <method>
    <ejb-name>MySessionBean</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>getEmployees</method-name>
  </method>
  <trans-attribute>RequiresNew</trans-attribute>
</container-transaction>
```

The above lines mark the method “getEmployees” in the session bean “MySessionBean” as requiring a new transaction when invoked. This session bean method could then call the entity bean methods, which all have been marked with transaction mandatory (which we saw in the previous section).

### **Bridging the User Identity to the EJB Container**

The Web container knows the user’s identity because we configured our servlet for authentication. We want to ensure the EJB container also has access to the end user’s identity. The user identity is passed from the servlet/JSP (Web container) to the session bean (EJB container) by including these lines in the ejb-jar.xml for the session bean.

```
<security-identity>
  <use-caller-identity/>
</security-identity>
```

The session bean now has access to the user’s identity via the bean’s context object.

### **EJB Session example #3 sample code**

The following is a sample EJB session bean implementation which demonstrates all the steps required to use a session bean to automatically manage the transaction and interact with the entity beans. In the example code below, we have included helper functions to setup and break down the JDBC connections before and after each use.

```
public Collection getEmployees() throws RemoteException {
    try {
        //before we can do anything,
        //we must first identify this jdbc connection
```

```

setDBContext();

//get the home interface of the Entity bean.
InitialContext ctx = new InitialContext();
EmpHome empHome =
    (EmpHome)PortableRemoteObject.narrow(ctx.lookup("Emp"),
        EmpHome.class);

Collection c = empHome.findAll();

java.util.LinkedList newColl =
    new java.util.LinkedList();

// Iterate over the real entities
// and convert/create value objects.
// We could use BC4J for this or some other
// library which would actually cache them, etc.
for (java.util.Iterator i = c.iterator(); i.hasNext();)
{
    Emp e = (Emp) i.next();
    ValueEmp ve = e.getAllFields();
    newColl.add(ve);
}

unsetDBContext();
return newColl;
} catch (NamingException ne) {
    // ...
}

private void setDBContext() throws SQLException, NamingException
{
    // Get the JDBC connection. Since getEmployees is marked
    // with NewTXRequired, this will grab the JDBC
    // connection and mark it as "in transaction".
    // Subsequent calls to the cmp entity beans
    // (since they are marked w/tx mandatory) will
    // use the same jdbc connection

    InitialContext ctx = new InitialContext();
    DataSource ds =
        (DataSource)ctx.lookup("java:comp/env/OracleDataBase");
    Connection conn = ds.getConnection();

    // Make the call to identify this jdbc connection
    PreparedStatement cstmt =
        conn.prepareStatement("begin dbms_session.set_identifier(?); end;");

    // Note the user is now Identified in the EJB Context
    cstmt.setString(1,context.getCallerPrincipal().getName());
    cstmt.execute();
    cstmt.close();

    // Don't forget to put the jdbc connection back in
    // the pool. Since we are sill in a tx,
    // It won't get used by any other user/thread until our
    // tx is either
    // rolled back or committed (or timed-out).
    conn.close();
}

private void unsetDBContext() throws SQLException, NamingException
{

```

```

// Since we call a constructor before all database
// access, it is good programming practice to call a
// destructor after all DB access. Here you could
// break down the database session (unset the client
// identifier, or do any other db processing. An example
// might be custom application logging, where a custom
// audit record might be written after regular DB
// processing. Although not necessary, we will un-set
// the client_identifier here, just to be sure!

InitialContext ctx = new InitialContext();
DataSource ds =
    (DataSource)ctx.lookup("java:comp/env/OracleDataBase");
Connection conn = ds.getConnection();

//Make the call to reset the connection

PreparedStatement cstmt =
    conn.prepareStatement("begin " +
        "dbms_session.set_identifier(null); " +
        "end;");

cstmt.execute();
cstmt.close();
conn.close();
}

```

## CONCLUSION

The ability to use the built in Oracle database security features greatly increases overall security and adheres to the best practice of defense in depth. There are many ways to secure data within the database. We explored the use of VPD for row level security, Client Identifiers for user identity propagation, and fine grained auditing for conclusive end user accountability.

This paper only discussed the use of a limited set of technologies to demonstrate basic security functionality. Many database security implementations may also take advantage of additional Oracle database technologies available such as application contexts, secure application roles, Oracle Label Security, enterprise users, and data encryption – just to name a few.

The use of J2EE CMP greatly increases developer productivity. By understanding the actions and interactions that take place, we can manipulate the CMP container to allow us to utilize the Oracle database security features. The solution presented involved identifying the user, and then preserving that identity across multiple CMP operations. We preserved the identity by exploiting J2EE transactional capabilities. We offered three successive examples which illustrated various ways to use J2EE transactions to ensure the security integrity of our application and database.

By exploiting the J2EE CMP transactional capabilities and the Oracle database security features, we truly have a solution that is optimal. The database security cannot be circumvented, it's layered, efficient, and can be easily managed. The application uses standard methods and processes which helps to ensure portability and scalability. This framework can serve as a model for building and deploying secure, scalable, and standards-based applications with no compromises.



Leveraging Oracle Database Security with J2EE Container Managed Persistence  
October 2003  
Authors: Matt Piermarini, David C. Knox

Oracle Corporation  
World Headquarters  
500 Oracle Parkway  
Redwood Shores, CA 94065  
U.S.A.

Worldwide Inquiries:  
Phone: +1.650.506.7000  
Fax: +1.650.506.7200  
[www.oracle.com](http://www.oracle.com)

Copyright © 2003, Oracle. All rights reserved.

This document is provided for information purposes only  
and the contents hereof are subject to change without notice.

This document is not warranted to be error-free, nor subject to  
any other warranties or conditions, whether expressed orally  
or implied in law, including implied warranties and conditions of  
merchantability or fitness for a particular purpose. We specifically  
disclaim any liability with respect to this document and no  
contractual obligations are formed either directly or indirectly  
by this document. This document may not be reproduced or  
transmitted in any form or by any means, electronic or mechanical,  
for any purpose, without our prior written permission.

Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective owners.