HOWTO Secure and Audit Oracle 10g and 11g

Ron Ben Natan

Foreword by Pete Finnigan

Chapter 14

Database Activity Monitoring

Database activity monitoring (DAM) is a technology for monitoring and analyzing database activity that operates independently of the database and does not rely on any form of native auditing. Database activity monitoring and prevention (DAMP) is an extension to DAM that also prevents activities from happening even if these activities are allowed according to privileges defined in the database. DAMP preserves the most important characteristic of DAM—the independence from the database management system (DBMS).

DAM systems emerged when companies faced many requirements that put a focus on the need to provide more visibility into what activity occurs within production databases. DAM systems are very often used as the solution of choice to implement database auditing. But DAM systems do much more than generate audit trails—the focus of this chapter is mostly on the additional functions DAM systems can perform in addition to auditing.

When monitoring requirements first started emerging, database administrators (DBAs) tried to address them with the most appropriate tool that they had—auditing. By using AUDIT statements (and by using fine-grained auditing [FGA] policies), you can write out what the database is doing and then export these records to another system for analysis, alerting, etc. However, these implementations were lacking in two ways. The first is that the overhead of these schemes made them very expensive to implement. Requirements for activity monitoring can be very diverse and can include monitoring activities that occur very frequently. Although one very important use case for DAM is privileged user monitoring, there are other use cases including application monitoring, sensitive data monitoring, and even comprehensive monitoring. When you monitor (and audit) privileged users only, the burden in terms of the size of the audit trail will not have a huge impact on the database. But as you start monitoring and auditing more and more you need to find access anomalies in Data Manipulation Language (DML) or SELECT access, using native auditing implies a performance hit that most people cannot tolerate. This is the first type of overhead that DAM systems help resolve.

The second is the overhead associated with change management. When an auditor changes requirements, you have to modify the AUDIT statements. You cannot just apply these AUDIT statements to your production systems—you have to make sure this will not have any impact to the application. You need to try it on your test systems first and you may need to test it for a lengthy period of time to ensure that there are no issues in terms of performance and storage.

If you have to pass a yearly audit and every year the requirements change (which is not uncommon) this is a huge cost to bear. Because DAM systems do not rely on the database for monitoring and do not affect the production systems, you can apply changes to monitoring and audit policies without fearing that this will impact the applications.

Beyond the overhead, DAM systems emerged because in addition to monitoring requirements, many architectural and organizational requirements emerged. The most important of them are separation of duties and the need for the information security officers to take ownership of database security and auditing. As long as it is the database that does the auditing and as long as audit trails are written to a location where the DBA or instance owner can modify audit records, and as long as the DBA can modify the audit definitions, the implementation will not pass most audits. Additionally, because these audit trails and the definitions of the policies should be owned by the infosec group (and not by the database group), the tools used to manage the definitions and the data should not require database expertise.

Another important architectural requirement that helped DAM emerge is the fact that there are very few companies that use only Oracle. Most companies have a heterogeneous environment that may include Oracle databases but also includes other databases such as SQL Server, DB2, Sybase, Informix, MySQL, etc. Each database has its own implementation of native auditing. The functionality and the usage is different in each platform. For companies that have such heterogeneous environments DAM provides an easy way out—all functions implemented by DAM systems are equivalent no matter what the monitored database type.

DAM systems also implement real-time alerting. Good DAM systems can alert you of inappropriate access at the moment that it occurs. This allows you to launch a remediation process immediately and for most requirements this is enough. Even if someone grabs sensitive data that they are not entitled to have, if you know about it soon enough and if you have enough information about who it was and where the access was made from, then you can usually contain the issue. DAM is considered to be a very effective technology for combating data breaches (using real-time alerting and precise information about the access). However, it is still a reactive technology—the breach already occurred and the data has already leaked. DAMP is a technology that evaluates policies before letting the queries and transactions reach the database. Rather than alerting when inappropriate access occurs, it simply prevents that access from happening. Both DAM real-time alerting and DAMP prevention are important data-security technologies that are at the core of the DAM value proposition.

DAM systems are different from Security Incident Event Managers (SIEM) systems. SIEM systems read the audit records produced by the database's audit trails and populate an event repository that includes data from other logs such as firewall logs, routers, operating system logs, etc. The biggest differences between DAM and SIEM are in nonintrusiveness and in monitoring functions:

- DAM systems usually produce the monitoring data independently and nonintrusively (to the database), whereas SIEM systems rely on database auditing.
- DAM systems provide far more advanced functions in terms of database monitoring whereas SIEM systems are "generalists" in that database events are merely another type of event.
- SIEM systems can correlate and analyze events from multiple sources whereas DAM systems usually focus on database access (including application access).

Oracle Audit Vault is closer to a SIEM system than to a DAM system both in terms of architecture and in terms of functionality, but it is still an early-stage SIEM in that it can only get logs from Oracle databases and SQL server databases and as of version 10.2.3.1 DB2 UDB and Sybase as well.

Common DAM and DAMP Architectures—How They Work?

Although there are many functions that a DAM system provides, the most important function is that of being able to show you what Structured Query Language (SQL) statements were executed on the database. There are three main architectures that DAM systems used:

1. Interception-based architectures (also called inspection-based architectures): Most modern DAM systems collect what the database is doing by being able to "see" the communications between the database client and the databases server. Any database session involves a client that connects to the database server. The client and the server can reside on the same host or can be on different hosts. If the client is running on a remote host the session is a Transparent Network Substrate (TNS) session that usually runs over Transmission Control Protocol/Internet Protocol (TCP/IP). If the client resides on the same host as the server then the session can be of a variety of types—local TCP/IP session or non-TCP/IP sessions (such as those that occur when a client connects using a Bequeath protocol). In any case, there is always a client/server relationship. What DAM systems do is find places where they can view this communication stream and get the requests and responses without requiring participation from the database. The interception itself can be done at multiple points such as the network itself (using a network TAP or a SPAN port—if the communication is not encrypted using advanced security option [ASO]), at the operating system level, or even at the level of the database libraries. As Figure 14.1 shows, if there is unencrypted network traffic then packet sniffing can be used. The advantage is that nothing is done on the host and thus there is no impact on performance whatsoever. To capture local access a probe runs on the host. This probe intercepts all local access and can also intercept all networked access in case you do not want to use network gear or in case the database communications are encrypted. The probe does not do all the processing—it

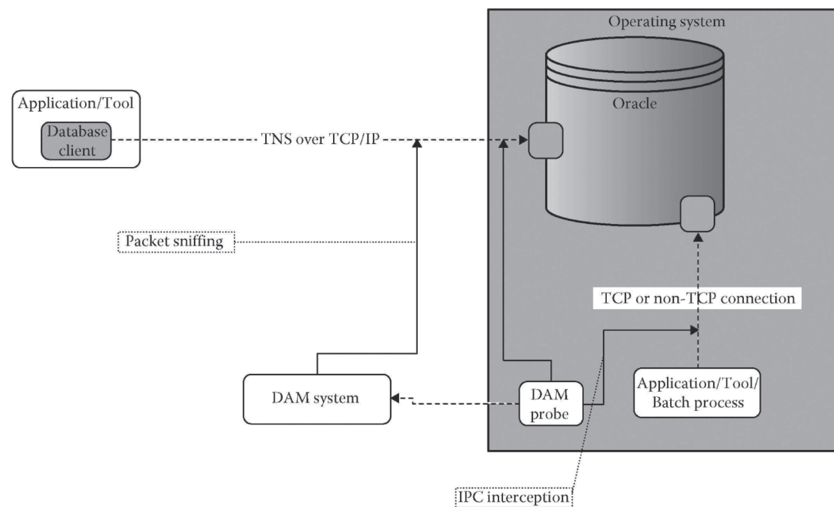

Figure 14.1 Inspection-based DAM architecture.

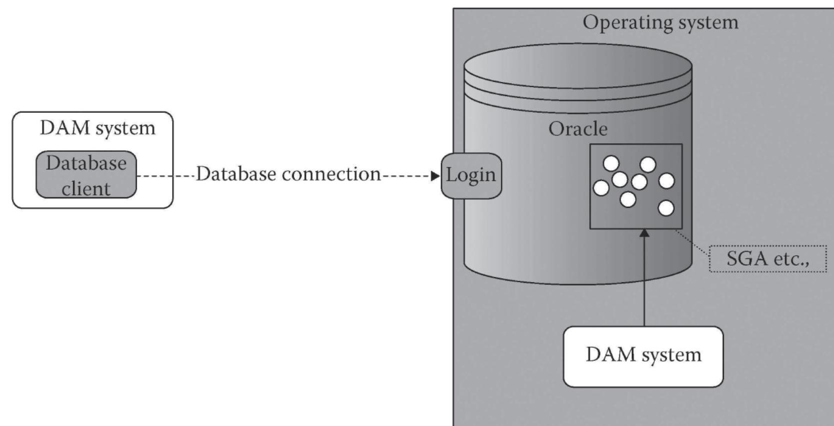

Figure 14.2 Query-based DAM architecture.

relays the data to the DAM server where all the processing occurs. This ensures that the impact on the database is negligible.

2. Query-based architectures: Some DAM systems connect to Oracle and continuously poll the system global area (SGA) to collect what SQL statements are being performed as shown in Figure 14.2. This is a carryover architecture from some of the performance products that used the SGA and other shared data structure. There are two variants of this approach—one that makes a real connection to Oracle and makes valid queries to the database and one that runs on the host and attaches to the process at the OS level to inspect private data structures. This architecture is no longer being used in the mainstream—at least not for DAM and auditing. The main problem with this architecture is that if you don't poll the SGA fast enough you can easily miss many of the statements run against the database and if you poll too often you will impact the performance of the database. Another problem with this architecture is that the DAM system needs a high-privilege connection into each of the database systems it monitors. This architecture has all but gone away and most vendors that started with this architecture have re-architected their product to use an interception-based architecture.
3. Log-based architectures: Some DAM systems analyze and extract the information from the transaction logs (e.g., the redo logs). These systems use the fact that much of the data is stored within the redo logs and they scrape these logs. Unfortunately, not all the information that is required is in the redo logs. For example, SELECT statements are not and so these systems will augment the data that they gather from the redo logs with data that they collect from the native audit trails as shown in Figure 14.3. These systems are in many respects a hybrid between a true DAM system (that is fully independent from the DBMS) and a SIEM which relies on data generated by the database. These architectures usually imply more overhead on the database server.

There are multiple architectures for DAM systems but there is only one architecture for DAMP systems. All DAMP systems have an architecture resembling the interception-based DAM architecture. The reason is that in both the query-based and the log-based architectures the DAM system knows that something has occurred only after it has occurred—it needs the database to process the request for it to be available. Therefore, it can never prevent anything from happening.

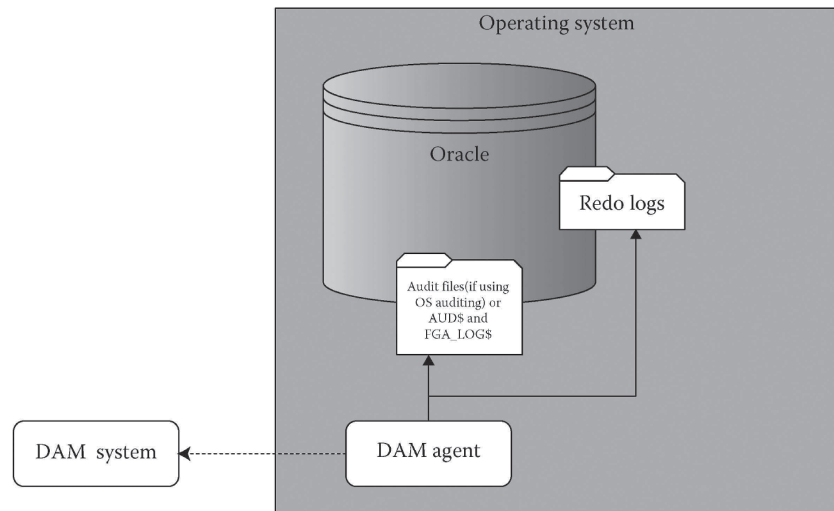

Figure 14.3 Log-based DAM architecture.

The interception-based architecture on the other hand has access to the request before it gets to the database and it can evaluate whether to allow it through or not. DAMP systems therefore have an interception-based architecture as shown in Figure 14.4. As in the DAM architectures, interception and prevention can occur on the network (by deploying the DAMP system between the database server and the network switch as shown in Figure 14.4 by system 1) or using a probe

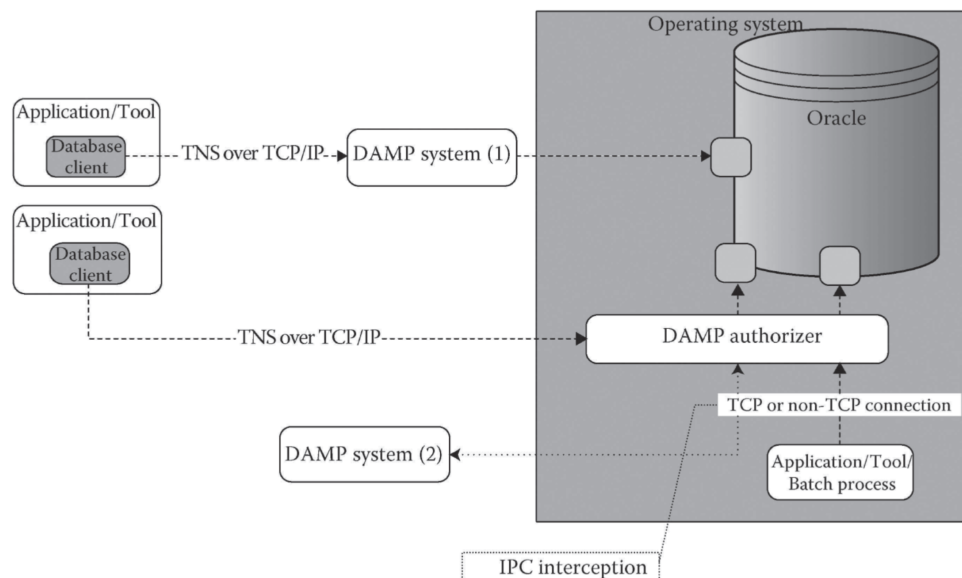

Figure 14.4 DAMP architecture.

that can decide whether or not to let the request reach the database using a policy defined by the DAMP system (as shown in Figure 14.4 by system 2). The latter architecture has two important advantages—it is far easier to deploy because it does not require network changes and it allows prevention for local connections as well as remote connections.

Functions Provided by DAM Systems

The term DAM is broad because it covers any type of access to the database and covers both logging/auditing as well as monitoring and real-time alerting. However, a few use cases that are very common and usually prompt people to adopt DAM technologies are

- Privilege user monitoring: DBAs, ISAs, developers, and other privileged users need to be monitored. This has become a best practice and is mandated by most regulations. Monitoring privileged user activity includes auditing their activities, identifying anomalous privileged activity and reconciling privileged activities with change requests. A very common example of this use case is a SOX implementation.
- Application activity monitoring: Application activity differs from users who connect directly to the database. Application activity tends to be very intensive but also highly repetitive. There are two forms of application activity monitoring. One usage of DAM monitors all application activity and generates a normative baseline. Once approved, this baseline can be used to identify anomalous application activity. The main goal in such a scenario is to detect and prevent an attack on the data from the application layer (possibly using a vulnerability in the application). A second usage type occurs when application users perform administrative activities or access sensitive information through the application and the application uses connection pooling to connect to the database. Audit trails are needed showing which transactions and queries were performed by which application user. The difficulty is that the credentials are at the application level whereas the activity is at a database level; this is what this form of application activity monitoring resolves. DAM can help you to identify the real user—more on that in Section 14.7.
- Access to sensitive data: Many regulations mandate the tracking of access to sensitive data. Sensitive data differs for different companies—sometimes it is personally identifiable information (PII), sometimes it is patient information, sometimes it is financial data, and other times it is intellectual property. The use cases around sensitive data sometimes require only the changes to be monitored (i.e., DML). In other cases you may be required to also monitor SELECT statements. When SELECT statements need to be monitored, the result sets (or subsets thereof) may also need to be monitored. In both cases the need to monitor this activity may be limited to some connections or may encompass all connections irrespective of the originator. Two common initiatives that fall in this use case are privacy initiatives and payment card industry (PCI) (see more in Appendix A).
- Access to encryption keys: As you saw in Chapter 8, you can use DBMS_CRYPTO to encrypt data or you can use TDE. TDE only encrypts data at the storage level and is not used as an access control mechanism. If you decide to use DBMS_CRYPTO then you are responsible for managing the encryption keys yourself. Normally, these keys are kept inside a database table so that application code and stored procedures can use them. The main issue then becomes whether or not you have enough controls in place to ensure that an unauthorized user cannot use the keys. One of the DAM use cases is to monitor and alert on unauthorized access to the encryption keys. Although this is really a subset of the sensitive data monitoring use case, it is common enough and specific enough to warrant special mention.

- Anomaly detection and intrusion detection: Because DAM systems can monitor all activity in the database without requiring auditing to be turned on, they can monitor for anomalous behavior of authenticated users and application and help identify an attack that is launched on the database. Because DAM systems inherently understand the SQL commands and the result sets, they act as an intrusion detection system (IDS) for databases.
- Support for notification laws: Most countries have some form of notification laws. Notification laws help combat identity theft. They specify that if you have PII of a person and have had a data breach in which that PII was compromised, you must notify that person that the breach occurred and usually you also have to compensate that individual (e.g., with free credit reports). The reason that these notification laws have emerged is that companies, left to their own devices, prefer to hide the data breach rather than make it public and notify the people whose information was stolen. Letting the public know about a data breach can damage a company's brand, cause the company to lose customers, and more. However, these data breaches often lead to identity theft and to damage that can be avoided if the individuals knew their data was compromised and that they are at risk. Simple monitoring of bank accounts, credit card accounts, credit reports, etc., is a very effective measure. Hence the notification laws—they ensure that companies do the responsible thing and tell the people whose data has been compromised of the breach. What DAM systems can help with (besides help identify the breach), is limit the number of people to whom a company needs to send this letter (and limit the number of people that need to receive remediation provisions). By monitoring access to PII and result sets, a DAM system can record precisely which records were accessed by an offending connection. If this offending connection extracted 2000 records from a table that has 2 million records, and if you monitor this access, you may be able to notify only 2000 people—much simpler and much cheaper.

Given these use cases, the main capabilities that DAM systems must have:

- Ability to monitor activity which is both local and remote, and cover all types of connections such as TCP, BEQ, IPC, etc.
- Ability to monitor activity no matter how SQL*Net is configured. Ability to monitor the activity even when connections are encrypted.
- Ability to set policies that determine what to audit, what to monitor and at which granularity to audit. Policy rules must be sensitive to users, IPs, source programs, commands, objects, etc.
- Ability to extract and report on all attributes such as the user name, the program, the OS user, the client host, the client OS, the SQL statement run, etc. There are typically over a 100 attributes in a modern DAM system that can be used in a report.
- Ability to manage an independent audit trail that cannot be modified. Ability to prove this.
- Ability to support full identification and accountability. For example, when the connection is made using the Oracle instance account using "/ as sysdba", show who is logged into the instance account (more on this in Section 14.7).
- Ability to monitor and record information about the requests and the responses. Ability to not only show the SQL statements but also data about the result sets and error conditions returned by the database.
- Ability to send real-time alerts.
- Ability to manage large quantities of data efficiently without huge storage costs. Ability to archive data securely and efficiently and restore data quickly when needed.
- Ability to support and prove separation of duties without incurring additional staffing costs.

14.1 HOWTO Protect against SQL Injection

SQL injection is a technique for exploiting bad coding practices in applications that use relational databases. The attacker uses the application to send a SQL statement that is composed from an application statement concatenated with an additional statement that the attacker introduces. Many application developers compose SQL statements by concatenating strings and do not use prepared statement; in this case the application is susceptible to a SQL injection attack. The technique transforms an application SQL statement from an innocent SQL call to a malicious call that can cause unauthorized access, deletion of data, or theft of information.

SQL injection has received a lot of press and is usually considered to be related to Web applications. This is not true. SQL injection can be present in any application architecture. The focus on Web applications is however justified because Web applications cater to a broad range of users—internal as well as external—so the chance of an attacker trying to exploit the application is much higher.

Let's start with the classic example of application authentication bypass using SQL injection. Suppose that you have a Web form that has two fields that need to be entered by a user when they want to login to the system as shown in Figure 14.5. The application receives a user id and a password and needs to authenticate the user by checking the existence of the user in the USER table and matching the password with the data in the PWD column in that table. If the query produces a result set then the user is logged on. If not, an error message is shown. Assume (and this is the really important assumption) that the application is not doing any validation of what the user types into these two fields and that the SQL statement is created by doing string concatenation. Let's look at what happens if you maliciously type in the following user ID and password:

User ID: **' OR "='**
Password: **' OR "='**

In this case the SQL string that would be used to create the result set would be

```
select USERID from USER where USERID = " OR "=" and PWD = " OR "="
```

This will surely return a result set and the attacker will be logged onto the application (usually as the first user in the table).

Another very popular SQL injection technique involves the use of UNION ALL SELECT to grab data from any table in the system. The syntax for this SELECT option is:

```
SELECT …
UNION [ALL | DISTINCT]
SELECT …
  [UNION [ALL | DISTINCT]
   SELECT …]
```

Figure 14.5 Sample application login form.

UNION is used to combine the result from many SELECT statements into one result set. If you don't use the keyword ALL for the UNION, all returned rows will be unique, as if you had done a DISTINCT for the total result set. If you specify ALL, you will get all rows from all the used SELECT statements. Therefore, most SQL injection attacks make use of UNION ALL.

Attackers use UNIONs to "piggy back" additional queries onto existing ones. Lists that are displayed following a conditional search issue a select and display the contents of a result set on the page. For example, suppose that you can look up all flights to a certain city by entering the city name to get a list of flights. Each line in the list shows you the airline, flight number, and departure time. Assume that the application is vulnerable to SQL injection—i.e., it uses string concatenation and does not do any validation on what you type into the city input field which is used in the WHERE clause. The normal SELECT issued by such an application may be

```
select airline, flightNum, departure from flights where city='ORD'
```

Suppose that instead of entering ORD (for Chicago) into the search input field you inject the following string:

```
ORD' union all select userid, 'dummy1',sysdate from USER where '1'='1
```

In this case the resulting select statement will be

```
select airline, flightNum, departure from flights where city='ORD' union all
select userid, 'dummy1',sysdate from USER where '1'='1'
```

The result set you will get will include all user names in the application as shown in Figure 14.6. The attacker is using SQL injection to get information that they should not have access to and that may be used to further launch an attack.

Airline	Flight#	Departure Date/Time
Delta	2362	8/20/2004 17:00
AA	62	8/20/2007 19:00
JetBlue	51	8/20/2007 16:30
Continental	144	8/20/2007 19:40
Delta	414	8/20/2007 22:00
United	2314	8/20/2007 20:00
RONB	dummy1	20-AUG-2007
JANE	dummy1	20-AUG-2007
JOHN	dummy1	20-AUG-2007
ALEX	dummy1	20-AUG-2007
URSULA	dummy1	20-AUG-2007
MITCH	dummy1	20-AUG-2007
SYSADM	dummy1	20-AUG-2007
SYSTEM	dummy1	20-AUG-2007

Figure 14.6 User names exposed in a UNION-based SQL injection attack.

RECENT POSTS

Subject	Author	Date/Time (ET)
[illegible]	[illegible]	11:29am, Aug 30
[illegible]	[illegible]	11:29am, Aug 30
[illegible]	[illegible]	11:29am, Aug 30
[illegible]	[illegible]	11:29am, Aug 30
[illegible]	[illegible]	11:23am, Aug 30
[illegible]	[illegible]	11:21am, Aug 30
[illegible]	[illegible]	11:19am, Aug 30
[illegible]	[illegible]	11:16am, Aug 30
[illegible]	[illegible]	11:14am, Aug 30
[illegible]	[illegible]	11:13am, Aug 30
[illegible]	[illegible]	11:04am, Aug 30
[illegible]	[illegible]	11:01am, Aug 30
[illegible]	[illegible]	10:57am, Aug 30
[illegible]	[illegible]	10:53am, Aug 30
[illegible]	[illegible]	10:51am, Aug 30
[illegible]	[illegible]	10:50am, Aug 30
[illegible]	[illegible]	10:46am, Aug 30
[illegible]	[illegible]	10:21am, Aug 30
[illegible]	[illegible]	10:18am, Aug 30
[illegible]	[illegible]	10:18am, Aug 30

Figure 14.7 Posting a message to a message board.

Finally, let's quickly look at another SQL injection pattern—one involving insert selects. This method makes use of the fact that SELECT subqueries can be used within an INSERT request. As an example, suppose that you have a screen that allows you to add a message to a message board as shown in Figure 14.7. The application functionality may be as simple as inserting this message to a MESSAGE table and allowing all members to review messages posted to the board as shown in Figure 14.8 (blurred to protect the innocent).

Figure 14.8 Viewing messages on the message board.

Building a message board can use a table in the database called MESSAGES. The application can do a SELECT on this table, and the posting function can do an INSERT into this table. For simplicity, assume that the columns in the MESSAGES table are called SUBJECT, AUTHOR, TEXT, and TIMESTAMP and that the timestamp is auto generated. In this case the application code for posting a message may simply do

```
INSERT into MESSAGES (SUBJECT, AUTHOR, TEXT) values (<whatever you type
in the subject field>, <your login name in the application>, <whatever
you type in the message text area>)
```

This simple function is vulnerable to an injection attack using an insert select command. If you type in the following into the appropriate fields (with the proper escape characters omitted here for the sake of clarity):

Subject field: `start’, ‘start’, ‘start’); insert into messages (subject, author, text) select owner, object_name, object_type from all_objects; insert into messages values (‘end`
Author field: `end`
Text field: `end`

The following SQL statements will be sent to Oracle:

```
INSERT into MESSAGES (SUBJECT, AUTHOR, TEXT) values ('start', 'start', 'start')
insert into messages (subject, author, text) select owner, object_name,
object_type from all_objects
insert into messages values ('end', 'end', 'end')
```

In this case you will be able to see all the table object names listed on the message board.

Combating SQL Injection

There are a number of things you can do to combat SQL injection, including limiting application vulnerabilities, discovering SQL injection vulnerabilities and requiring that they be fixed, and protect your database by using DAM. As you’ve seen, SQL injection is not really a vulnerability of the database. It is a vulnerability in the application code that exposes the database and the data.

The first implementation option is to remove the application vulnerabilities. This is normally the responsibility of the application owner but sometimes it is appropriate for you as the database owner to be involved. By now there are some very good SQL injection guidelines for application developers; guidelines such as:

- All data entered by users needs to be sanitized of any characters or strings that should not be part of the input expression. All input fields must be validated.
- SQL used to access the database from application code should never be formed using string concatenation.
- Strongly typed parameters (usually in combination with stored procedures) should be used wherever possible.
- Prepared statements, parameter collections, and parameterized stored procedures should be used wherever possible.
- Application login should be implemented as a stored procedure.

These guidelines are for developers. If you have some leverage—use it. Make developers adhere to these guidelines. If you are fortunate you can even require a code review in which you participate. If you do, stress the use of prepared statements. When you use prepared statements as opposed to string concatenation the SQL strings are distinct from the values that you get from the user and thus there is no mixing of SQL and parameters. This is therefore one of the simplest ways to combat SQL injection. Beyond better security, they also can imply better performance if used correctly.

Monitoring and tracking whether or not prepared statements are used is one of the simplest things uses of a DAM system—you can see the difference in the SQL that travels on the network when using prepared statements and you can easily look at all the SQL traffic generated by an application to make sure that only prepared statements are used. With prepared statements, the SQL looks like:

```
update test set a = :1
```

The value would be communicated in an adjoining packet. Without prepared statements, the SQL looks like:

```
update test set a = 'ABC'
```

By monitoring this access and producing a report per application you can work towards more widely used prepared statements and a more secure environment.

In addition to code and design reviews, you can also make use of SQL injection detection tools. Tools can help you simulate a SQL injection attack to test your applications. These tools should be used by the developers themselves but in case you are the last bastion of hope for the data, you might want to explore the use of these tools yourself. Note that although these tools are effective, they are not comprehensive and are not always easy to use. The good news is that these tools are usually free of charge. As an example, SQL injector is a tool offered as part of the SPI Toolkit by SPI Dynamics (now HP) (http://www.spidynamics.com/products/Comp_Audit/toolkit/SQLinjector.html). This tool conducts automatic SQL injection attacks against applications making use of Oracle and test if they are vulnerable to SQL injection. The tool only supports two of the common SQL injection attacks—but even this limited test can be useful.

If you already have code deployed, or don't have the authority or energy to conduct code reviews you can use a DAM systems to identify and prevent SQL injection by monitoring the application activity and generating a baseline. A baseline of application access is a set of SQL structures that the application uses in normal operation. When a SQL injection attack occurs, these structures change. For example, in the login example a new OR condition suddenly appears. In the message board example, new INSERT statements appear, and in the flights example a UNION command suddenly appears. Because an application is generating these SQL statements and not a user, there should not be any changes to the SQL structures that the application is sending to the database. A DAM system generates a baseline of "normal behavior" and is able to identify an attack by seeing that there is a divergence from normal SQL structures and normal sequences. This is the only method that is effective on all types of SQL injection attacks and that does not introduce many false positives.

Before moving on to the next subject, be aware that there are IDSs that have SQL injection signatures. Unfortunately, signatures are not an effective tool in an Oracle environment simply because SQL and PL/SQL are very rich and any attack can be carried out in any number of ways. Looking for an attack based on a signature is not effective because of the many permutations that can occur. Signatures try to identify certain patterns as an indicator of an attack. The signatures that IDS use are usually the commonly used techniques of SQL injection. For example, you can

look for signatures such as 1=1 or UNION SELECT. The problem with this approach is that there are too many ways to carry out such an attack. For example, think how many different predicates you can think up that compute to an always true value. It may be '1'='1', or 'a'='a' or 'my dog'='my dog' or 'ron was here'='ron was here' or 'ron' LIKE 'ro%' or 1<2 or …—an infinite number of ways. The second problem is that some of these signatures may actually be used in real systems—it is not unheard of for people to use UNION ALL—this is why SQL supports the function. So your IDS may alert you on completely legal SQL and DAM is your best choice for SQL injection.

Three Things to Remember about SQL Injection

1. SQL injection has many variants and can happen in any application architecture. It is a vulnerability at the application level that occurs when user input is not validated and is used blindly to construct SQL statements that are sent to the database.
2. Use prepared statements or check user input at the application level and you will not have SQL injection vulnerabilities.
3. Use DAM systems with baselining features by identifying when a SQL statement has been modified from its normal structure.

14.2 HOWTO Categorize and Identify Misuse and Intrusions

Because the database performs many activities on behalf of many different users, it is not trivial to identify an intrusion or misuse of the data. For example, the use of DBMS_CRYPTO or UTL_HTTP may be completely legitimate for some users and some applications and may be a sign that something malicious is going on in other cases. Identifying intrusions needs to be based on context as well as content.

Because DAM monitors all database activity and has full visibility into what is being asked of the database, it is necessary for identifying misuse and intrusions. It is necessary because without it you only see what you have setup in your audit trails. DAM is a necessary technology but not all DAM systems can identify intrusions. It is not enough that the DAM system can see everything—it also needs to have analysis capabilities. Looking for intrusion or misuse is like looking for a needle in a haystack.

DAM systems that can differentiate between normal behavior and between intrusions look at content, context, and historical information. Content includes the use of signatures (or the signature equivalent in the database world), the use of profiles and the use of data. For example, the use of UTL_FILE or UTL_SMTP will trigger more scrutiny because of possible exposure. Context includes analysis of environmental parameters as well as the relationships between different SQLs. For example, a session in which 50,000 identity numbers are selected in sequence needs to be handled differently versus a session that does the normal sequence of application behavior which includes two selects, an update, another select, and an insert. Intrusion and anomaly detection for Oracle requires at least the following categories of analysis:

- Baselines and behavioral divergence: There are two forms of database activity—there are DBAs, developers, and power users that connect to the database and issue SQL statements and there are users that navigate through application modules and in the process cause SQL statements to be

sent to the database. For the first class of users it is hard to identify patterns because such users can potentially perform any SQL statement. However, these tend to have low volume and it is easier to monitor these connections closely. It is much harder to monitor application activity because of the volume. However, because the applications have a fixed set of SQL statements that they may call (it is fixed code after all), it is not too difficult to create a baseline of all observed application activity. This is called an application baseline and it encodes the state machine of the application as it accesses the database. Once such a baseline has been created it can be used for detecting intrusions. For example, if someone tries to launch a SQL injection attack then the SQL statement will have a different structure (e.g., different number of conditions) and the intrusion can be identified.

- Sequence monitoring: Baselines are usually constructed as points in a multidimensional space comprising of all the defining attributes of a SQL access. These dimensions include the objects, the statement, the structure of the SQL, the user, the application, the time, etc. However, you can get an even more granular definition of application activity if you account for sequences of statements. This allows you to identify intrusions not only based on a single activity but on a pattern of activity, frequency of activity, and the order between different activities.
- Errors and exceptions: In addition to monitoring SQL statements it is important to monitor ALL errors generated by Oracle and what SQL statements (or other activity) caused such an error to occur. An attack on a database does not take one second—it takes time and usually requires trial and error (especially if it is an attack launched through an application as opposed to an attack performed by a trusted insider). As an example, the attack may involve a UNION-based SQL injection attempt in which the types of columns may be wrong. It may involve trying to get data and causing privilege errors or errors when a column or table does not exist. All these errors are of utmost importance where intrusion detection is concerned and your DAM system must be able to monitor these events.
- Data extrusion: Another form of detection involves inspecting what data is returned on which session, how much data is returned, etc.
- Signatures of statements and packages: There are many Oracle procedures and packages that are either vulnerable (if you don't have the latest critical patch updates (CPUs) installed) or that are useful when performing an attack. You can monitor usage of these procedures to identify an attack—especially if you can profile under what conditions they are used normally.
- White lists and black lists: Almost all detection of intrusions involves some form of white list and black lists. You can have lists of users, lists of statements, list of objects, list of applications, lists of IPs, etc. White lists enumerate the elements from which you expect certain behavior and through which you approve certain behavior. For example, you can create a white list of users and a white lists of IPs for the use of UTL_SMTP. If you see any use of UTL_SMTP from a user or an IP not within the relevant white list you can classify this access as an intrusion. A black list is an enumeration of elements that you do not allow. For example, you can list a set of errors that you do not allow for any session—any session that will generate such an alert will be immediately flagged as an intrusion or as misuse.

Finally, note that to identify misuse you need to be able to analyze across multiple sessions. It is not enough to just monitor each session separately—sometimes understanding what is happening requires you to look at multiple connections and even multiple databases. As an example, a single user credential that is concurrently being used from different IPs is at least a misuse of credentials and sometimes an intrusion.

Two Things to Remember about Identifying Database Intrusions

1. Identifying user intrusions requires a combination of technologies that inspect the SQL requests, error states, and the responses for the database.
2. Detecting intrusions by users accessing the database directly uses different techniques than detecting intrusions that take advantage of applications. In the first you can use combinations of signatures and error messages while in the second using a baseline along with error detection is more effective.

14.3 HOWTO Understand the Compliance Landscape

Much of the investment in database security is driven by compliance. When using the term compliance the first question is “compliance with what?” Usually the answer is that your company needs to comply with a certain regulation or with an internal policy. There are thousands of regulations today that affect database security; yes—thousands. This certainly does not mean that all of them affect you—most regulations are geographical, some are related to only certain industries, and some to the size and type of the company. As an example, Table 14.1 provides a very partial list from a few national regulations relevant in certain countries (partial both in terms of countries and the regulations within each country). As you can see, different countries have different mandates. Also, Table 14.1 only lists national regulations. There are also separate regulations which are enforced through local government, for example, in the United States there are 38 separate state-level regulations that affect privacy of PII. There are regulations that are enforced by industry leaders. For example, the Payment Card Industry (PCI) Data Security Standard (DSS) is not a national regulation—it is an industry regulation which has been created by the credit card companies. There are even regulations that have been adopted from one country to another. For example, J-SOX is a Japanese regulation which has been created based on the American Sarbanes–Oxley regulation. Euro-SOX is similar within the European Union. Finally, there are internal controls and policies that are usually defined by internal audit or the internal risk management group that define requirements that you need to abide by in terms of database security, database auditing, and database risk management.

Table 14.1 can be frightening. No one has time to even understand all these requirements, let alone implement for them. The good news is that there aren’t many variants in terms of the requirements posed by regulations on what you need to do in a database environment as inferred by the columns of Table 14.1. As the table shows, different regulations imply different categories within database security/monitoring/auditing that you should invest in, but there aren’t hundreds of different permutations.

At a high level there are two main classes of regulations. There are regulations which are focused on governance and regulations that focus on sensitive data. Regulations that focus on governance are primarily focused on the controls you have in place and on risk management. These regulations usually put a very strong emphasis on DBAs and privileged users and almost always the first step in such implementations is to produce comprehensive audit trails for DBAs and define controls around changes that may affect the applications. The 900 lb gorilla of this category is SOX.

The second set of regulations focus on access to sensitive data. Such regulations also have a set of requirements that deal with DBAs (e.g., making sure that DBAs cannot access PII) but overall the focus of these regulations is largely on sensitive data—i.e., a subset of database objects. Examples include PII in various privacy regulations, PCI DSS where the focus is on credit holder information, credit card information, etc.

Table 14.1 Partial Mapping of National Regulations to Database Security and Monitoring Categories

Regulatory Requirement	*Access Control and Data Access Management/ Control*	*Change Management*	*Data Integrity and Availability*	*Monitoring, Auditing, and Reporting*	*Risk Management and Process Management*	*Segregation of Duties*	*Controls Management and Oversight*
Australia							
AS/NZS:4360	✓	✓		✓		✓	✓
CLERP9	✓				✓	✓	✓
Crimes Act	✓	✓	✓	✓	✓	✓	✓
DIRKS	✓	✓		✓	✓	✓	✓
Federal Freedom of Information Act	✓	✓			✓	✓	✓
Privacy Act	✓	✓		✓		✓	✓
Brazil							
Bill 3494/2000	✓	✓	✓	✓	✓	✓	✓
Computer Crimes Act of 2000	✓	✓	✓	✓	✓	✓	✓
Federal Senate Bill No. 61	✓	✓	✓	✓	✓		
Informatics Law of 1984	✓		✓	✓	✓	✓	✓
Projeto de Lai da Camara (multiple)	✓	✓	✓	✓	✓	✓	✓
Senate Bill No. 61	✓		✓	✓	✓	✓	✓
Canada							

Antiterrorism Act (C-36)	✓	✓	✓			✓	✓
Canadian Security Intelligence Service Act	✓	✓	✓	✓			✓
CSA 52-313	✓			✓	✓	✓	
Federal Access to Information Act	✓		✓				✓
Federal Privacy Act	✓		✓	✓			✓
PIPEDA CAN/CSA-Q830-96	✓		✓			✓	
Telecommunications Act	✓	✓	✓	✓		✓	✓
European Union (EU)							
EU 8th Directive/ FDICIA-EU Corporate Governance	✓			✓	✓		
EU Directive 95/46	✓	✓	✓	✓	✓	✓	✓
EU Directive 97/66	✓		✓		✓	✓	✓
EU Directive 2002/58	✓	✓	✓		✓	✓	✓
EU Directive on the Transfer of PII	✓			✓		✓	✓
MiFID	✓		✓				
Germany							
4th Financial Market Support Law	✓					✓	
AktG	✓	✓			✓	✓	✓

(continued)

Table 14.1 (continued) Partial Mapping of National Regulations to Database Security and Monitoring Categories

Regulatory Requirement	*Access Control and Data Access Management/ Control*	*Change Management*	*Data Integrity and Availability*	*Monitoring, Auditing, and Reporting*	*Risk Management and Process Management*	*Segregation of Duties*	*Controls Management and Oversight*
Cromme Commission Report	✓	✓		✓		✓	
Datenschuts Law	✓	✓		✓			
Federal Data Protection Act	✓				✓	✓	✓
Telecommunications Carriers Data Protection Ordinance	✓	✓		✓	✓	✓	✓
TransPuG	✓		✓	✓		✓	
Italy							
Constitution	✓	✓	✓	✓	✓	✓	✓
Employee Data Protection Provisions	✓	✓	✓		✓	✓	✓
Italian Data Protection Act	✓					✓	✓
Telecommunications Privacy Directive	✓	✓		✓		✓	✓
Japan							
Article 21 of the Constitution	✓	✓	✓			✓	✓
Act on Disclosure of Information by Public Agencies	✓	✓		✓	✓	✓	✓

Act for the Protection of Computer Processed Personal Data held by Administrative Orgs.	✓	✓	✓	✓		✓	✓
MIC	✓	✓	✓		✓	✓	✓
Personal Information Protection Act	✓	✓	✓	✓	✓	✓	✓
Russia							
Communications Act	✓	✓	✓			✓	✓
Freedom of Information	✓	✓	✓	✓	✓	✓	✓
Information Protection Act	✓		✓	✓	✓	✓	✓
South Africa							
Access to Information Act	✓	✓	✓			✓	✓
Electronic Communications and Transactions Bill	✓			✓		✓	✓
Interception and Monitoring Act	✓	✓	✓		✓	✓	✓
King II (Corporate Governance)	✓	✓	✓	✓		✓	✓
United Kingdom							
Data Protection Act	✓			✓		✓	✓
Employee Privacy Legislation	✓	✓				✓	✓
Freedom of Information Act	✓	✓		✓	✓	✓	✓
Health and Social Care Bill	✓					✓	✓
United States							
Bank Protection Act	✓	✓	✓	✓	✓	✓	✓

(continued)

Table 14.1(continued) Partial Mapping of National Regulations to Database Security and Monitoring Categories

Regulatory Requirement	*Access Control and Data Access Management/ Control*	*Change Management*	*Data Integrity and Availability*	*Monitoring, Auditing, and Reporting*	*Risk Management and Process Management*	*Segregation of Duties*	*Controls Management and Oversight*
CFR 332	✓	✓	✓	✓	✓	✓	✓
CJCSI (multiple)	✓					✓	✓
CJCSM (multiple)	✓	✓				✓	✓
Computer Fraud and Abuse Act (CFAA)	✓	✓	✓	✓	✓	✓	
Computer Security Act	✓	✓	✓	✓	✓	✓	
Computer Security Enhancement Act (2001)	✓	✓	✓				✓
Critical Infrastructure Protection	✓	✓					
DoD Directives (multiple)	✓	✓	✓	✓	✓	✓	✓
DoD Instruction (multiple)	✓	✓	✓	✓	✓	✓	✓
DoDD (multiple)	✓	✓	✓	✓	✓	✓	✓
Electronic Communications Privacy Act	✓	✓			✓	✓	
Energy Policy Act	✓					✓	✓
Federal Reserve Act 12 (multiple)	✓	✓	✓	✓	✓	✓	✓
FIL-124-97	✓	✓	✓	✓	✓	✓	✓

FIL-14-97	✓	✓	✓		✓	✓	✓
GISRA	✓	✓		✓			✓
Government Network Security Act (2003)	✓	✓	✓				✓
Gramm–Leach–Bliley Act of 1999	✓			✓			✓
Health Insurance Portability and Accountability Act (HIPAA)	✓	✓		✓	✓		
IGES (multiple)	✓	✓	✓	✓	✓	✓	✓
NIST 800 (multiple)	✓	✓	✓	✓	✓	✓	✓
NSTISSP	✓					✓	✓
OMB (multiple)	✓	✓	✓	✓	✓	✓	✓
Patriot Act of 2001	✓			✓		✓	✓
PCI DSS	✓			✓	✓	✓	
PPA	✓	✓	✓	✓	✓	✓	✓
Privacy Act	✓	✓	✓	✓		✓	✓
Sarbanes Oxley Public Company Accounting Reform and Investor Protection Act of 2002		✓		✓		✓	✓
TCPA	✓	✓	✓	✓	✓	✓	✓
Telecommunications Act of 1996	✓	✓	✓				✓
USC 1030	✓	✓	✓	✓	✓	✓	✓
USC 5311	✓	✓	✓	✓	✓	✓	✓
Various State Notification Laws	✓			✓		✓	✓

The good news is that if you go through a fairly comprehensive implementation of database security you will likely cover multiple regulations that you may have to comply with. Certainly if you implement one of the regulations you will be covering all regulations within that category, but if you also adhere to the best practices rather than looking at the regulation then you will likely cover many more requirements. In fact, you should consider implementing security best practices rather than addressing a compliance checklist. If you implement elements of what this book covers (e.g., assessments, monitoring, auditing, change tracking, encryption—even if you implement each partially), and if you put a process in place that enforces best practices, then you will almost certainly have a secure environment and you will be in compliance with the relevant regulations. If you invest in complying with a very specific checklist you might find that you have not achieved better security, that you have not brought yourself closer to compliance with other requirements, and you may even be at risk with your main compliance driver because the technical implementation is an interpretation of the regulation. The bottom line is that good security implies compliance (and compliance with multiple regulations) whereas compliance with a certain requirement does not necessarily lead to better security and does not imply compliance with multiple regulations.

Two Things to Remember about the Compliance Landscape and Mapping Compliance Requirements to a Technical Implementation

1. Secure your database environment well following the guidelines and tools presented in this book—this will guarantee that you are compliant with any regulation your database environment needs to comply with.
2. There is a high degree of commonality between many regulations—don't let the sheer number of regulations overwhelm you. Identify which regulations focus on governance and risk management and which regulations focus on data access and data privacy and implement generic controls and policies that will cover all relevant regulations.

14.4 HOWTO Determine Whether You Need DAM or DAMP

Technically, it seems very simple to know when you need DAMP. You need DAMP when you need to prevent activities from happening versus the use of monitoring and real-time alerting and using these as prevention-through-deterrence. At a business-level, it is harder to know when DAM (and the right process) is enough versus when DAMP is truly required. You should make the distinction because the cost associated with DAMP is usually higher than the cost associated with DAM and because DAM is a truly nonintrusive technology whereas DAMP is not. Here are some important use cases for DAMP deployments:

1. Privileged user access to sensitive data such as PII: DBAs and other privileged users with system privileges can access any data in any schema. There is no need to explicitly grant privileges to objects containing PII data and there is no way (in Oracle or any other database) to limit such access. Preventing such access (mostly preventing SELECTs) is required by multiple regulations and in many geographies. This is usually solved through monitoring and real-time alerting but many are moving to prevention as a way to reduce cost (of review). Monitoring is also viewed as a compensating control whereas preven-

tion is the control that compliance requires. Note that although SELECT statements are the most common, there are many activities that may need to be prevented—e.g., creation of views, creating data pump tables, dropping tables, etc. This use case covers many scenarios that differ only in which rules are defined—based on user accounts, based on which IP or subnet the connection comes from, based on which application is making the request, based on the time of day, based on which schema and database objects are accessed, etc.

2. Outsourced DBAs and cross-boundary laws: Companies that outsource DBA positions are extremely sensitive to access by nonemployees. This use case is a subset of the previous one but gets more visibility and priority. This use case also gets special attention due to the possibility of changes to business-affecting data. In the previous use case the focus is often on access to sensitive data and is often related to PII. However, there is also a similar use case involving business-affecting data. In this use case the focus is usually not only on access to data but also on the ability that a DBA has to change data (i.e., DML versus SELECT). Although this scenario is as valid for internal DBAs as it is to outsourced work, addressing this issue when internal employees are involved is often based on monitoring and auditing whereas outsourced scenario often require better preventive controls.
3. Segmentation of applications and their related data: Because multiple applications often live in the same instance or on the same operating system and because users with system privileges can access data across these schema and database boundaries, there is a need to enforce policies in which privileged users of one application cannot connect to the schema of another application, even when running on the same instance or machine.
4. Separation of duties: There are often requirements for more than one person to collectively perform a highly privileged activity. This is hard or impossible to implement when users have all privileges. Prevention based on multiple factors is used to implement such schemes (metaphorically similar to two people turning a key to launch a missile). This use case includes handling of patch installations, upgrades, backups, replication changes, etc. This use case is also relevant when database links are involved and where data from one application is exposed through another, and the linked application cannot rely on the security rules built into the linking application.
5. Implementation of encryption schemes: When implementing data encryption using DBMS_CRYPTO (versus TDE), the encryption keys are often stored in the database within tables. Prevention is used to ensure that DBAs do not access the encryption keys. This is another special case of use case #1.
6. External policy definition for legacy applications with limited access control: Applications (especially legacy application and database schemas that are part of a purchased application) often have very limited built-in schema access control. It is not possible to change these applications and yet access control is mandated by regulations. To maintain compliance and not have to replace applications, external prevention is used.
7. Data leak prevention: Vulnerabilities in application and lax access control definitions increase the risk of leakage of sensitive data. For example, a vulnerable script or application can be changed to extract ALL records from the database. Prevention can be based on extrusion of data and not on access. Prevention in this case is coupled with reporting to support notification—e.g., rather than have all 2 million customer records extracted, terminate the connection as soon as the breach is identified (e.g., after access to 250 records), report on which 250 records were extracted and supply the records to allow notifying only these 250 customers.

8. Rogue application prevention: Application data in the database should only be accessed and changed through the application. However, access control is almost always based on user name. Compliance requirements often specify that changes to data (and sometimes also access to data) should only be made through the application where application-level security is enforced. Any connection from another application, script or tool should be prevented. An example is the use of Excel to connect to the database, extract data and even change data using VBA.
9. Enforcement of change request process: Most companies manage DBA tasks in a change request/ticketing system. When DBAs needs to perform work they should have a valid ticket number that references at least the instance. Most companies rely on auditing and reconciliation of activities produced by the auditing system with the ticket description. A more advanced approach is to require ticket information to be entered as part of the session and prevent certain activities if the ticket number is invalid or is not related to the instance. This can be used for all DBA work or, more likely, for security-affecting work such as changes to authentication attributes, changes to encryption options, changes to backup options, etc.

It is interesting to note that DAMP, like DAM, is very much driven by compliance and risk management.

Three Things to Remember about Using DAMP versus DAM

1. Most DAMP implementations are still driven by compliance, but they have a very strong security orientation because they create an external access control overlay.
2. DAMP implementations are simple because they are rule based and can easily support any requirement for access control based on any number of factors—all without modifications to Oracle.
3. The main use cases for DAMP are controls around users with system privileges and breaches that occur and extract data from the database.

14.5 HOWTO Analyze Impact on Performance

DAM systems (especially those with interception architectures) have less impact on performance than the alternative options. However, even within this DAM architecture there are different attributes that can affect impact on the database server.

Looking back at Figure 14.1, a DAM system can intercept database communications on the database server itself or by using network gear. The only impact that a DAM system can have is when the probe is running on the database server; if packets are inspected using a switch port mirror (e.g., a SPAN port) or using a network tap then the impact on performance is zero. SPAN ports and network taps create a copy of the packets as they are being placed on the port on their way to the database server, and placing this copy on the port mirror (see Figure 14.9). They do so while guaranteeing that there is absolutely no impact on performance—there is no added latency to the real packets on their way to the database, there is no impact on the throughput, etc. Therefore, when you inspect through network gear you don't even have to do any testing.

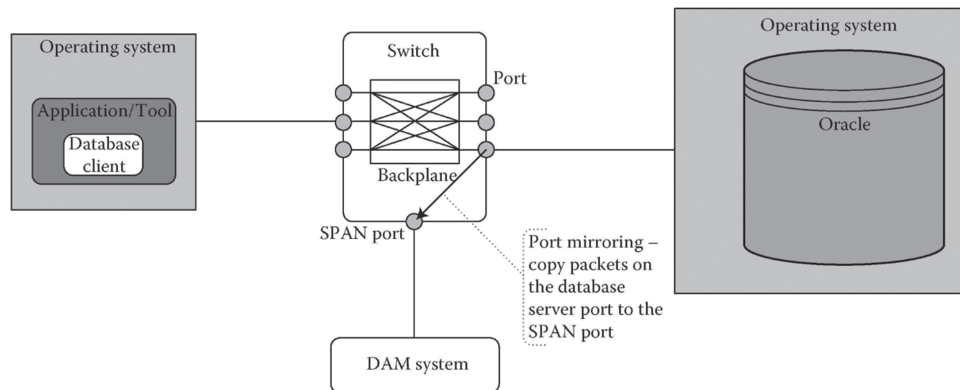

Figure 14.9 Using a switch to get a mirror stream of database activity.

Technically, doing packet sniffing is an optimal solution (at least as far as performance is concerned). However, there are a number of issues with packet sniffing including the fact that local traffic cannot be analyzed, the fact that encrypted traffic cannot be inspected, and the fact that for an environment with a large number of servers it may be impractical to intercept at the switch. Therefore, it is important to understand how these probes work so you can better analyze their impact on performance.

As shown in Figure 14.1, probes do two things—they create a copy of the database activity at real time by inspecting the interprocess communication that the database is involved with, and they send this data to the DAM server. The fact that the probe does so little is key to the ability of a DAM system to remain nonintrusive and not impact the database performance. However, the probe is running on the database server and is consuming resources. You should understand what attributes such probes can have and how this can affect performance:

1. The impact of a DAM probe on performance is directly related to the amount of database activity that it needs to monitor. The probe makes a copy of data packets and writes them—usually on a socket. Writing ten times more information generally means that the probe has ten times more work to do and thus consumes ten times more resources. Therefore, make sure that the DAM system that you're evaluating allows you to filter out various database activities, for example, based on who the user is.
2. At least half of what the probe does is write the data to a socket. Apart from consuming resources, this also has an impact on network load because this data needs to be sent to the DAM server. A DAM probe that can filter what gets sent to the DAM server not only consumes less resources on the host, it also adds less overhead from a networking perspective. Some probes will even allow you to fine-tune whether or not you want long result sets sent to the DAM server. This gives you even more control on the impact on resources.
3. Encryption is a relatively expensive computational operation. DAM probes have the ability to encrypt the traffic (usually using Secure Sockets Layer, SSL) between the probe and the DAM system. However, think twice about whether or not you need this capability because

it does affect performance. It is not unrealistic to have a probe consume 2 percent CPU without SSL and seeing it go up to 4 to 5 percent when SSL is enabled. As with other operations performed by the probe, the impact on performance is proportional to the amount of data sent to the DAM server—if you need to encrypt ten times more data you will consume ten times more resources. As a rule of thumb, you only need to use encryption between the DAM probe and the DAM server if you are encrypting the data-in-transit between the database clients and the database server.

To summarize, DAM systems have the best audit-to-performance ratio and have the least impact on server performance. Using network inspection has the potential to reduce the impact to zero. Using host probes will have some impact but can be very small especially if your probe allows you to control just how much you write to the network.

Three Things to Remember about Analyzing DAM's Affect on Performance

1. Impact on performance is directly correlated to the amount of data you are inspecting for all but network-based DAM inspection.
2. Use filters to define what you are collecting and what you are auditing—there is little sense in auditing what you do not need.
3. If you are using a host probe only, remember that encrypting DAM communication is relatively time consuming. Do this only if your database traffic is encrypted in the first place.

14.6 HOWTO Analyze Impact on Storage

DAM systems potentially collect a lot of information. For example, when a vendor tells you that it can audit at a rate of 2500 statements per second, you can do some simple math to see what the impact on storage can potentially be. For example, if you assume that an audit record takes 200 bytes then a simple calculation implies that the audit records from a single day will consume over 40 GB of disk space. If this data is archived it can be compressed, but will still take a lot of disk space. More importantly, you cannot always compress this data and move it to secondary storage immediately—you may need to keep it online for a certain period of time for compliance reporting purposes. As an example, if you need to keep it online for a period of 60 days, then just that one system may require over 2.5 TB of disk—this is a very expensive proposition. If the average audit record consumes only 20 bytes then this becomes a more manageable 250 GB of disk space. If the average audit record is 800 bytes then this number becomes an unmanageable 10 TB for a period of 60 days. The most important number is the average size of an audit record. Other attributes that will affect the amount of storage you will require for your implementation are:

- Appliance packaging or software-only deployments: Many DAM systems are packaged as a security appliance. Beyond the advantages that this packaging provides in terms of the system being hardened by design, these systems include their own disks so that they can collect and report on data without requiring external storage. For these systems you need to validate how

long data can be kept on the appliance's disks. At some point you will need to archive the data to make room for new data but if the DAM system is well designed this goes directly to secondary storage so that you do not have to analyze storage requirements. For software-only DAM solutions you need to size the disks on the server that will be running the DAM system.

- Monitoring versus auditing: Many people do not distinguish between the requirements for DAM monitoring versus DAM auditing. If you audit at a rate of 2500 records per second there is no doubt that no one will review this audit trail. Therefore, auditing at such rates is only valuable for potential future investigations. Much more typical are requirements which include monitoring database activity at very high rates (tens of thousands per second) and auditing a small subset of these activities. From a storage perspective, the monitoring rate is not important; it is just the auditing rate that is important. Therefore, make sure that when you look at DAM systems you have a clear understanding of what your monitoring requirements are, what your auditing requirements are, and that the DAM system is able to define monitoring policy rules and auditing policy rules that may be distinct. As an example, you may need to monitor all database activity to detect misuse and intrusions but will only audit access to sensitive data and statements that are identified as an intrusion.
- Normalization: One of the biggest issues with some DAM systems is that they do not normalize the data before they store it. In Chapter 13 you saw that the Audit Vault schema has a FACT table that contains the statement and that all of the attributes that qualify this statement are stored in DIM tables. The data is normalized in that you do not save all of the dimension information multiple times, one per access. This is the convention with DAM systems—most of them store the data in a database in a normalized manner. However, some first-generation DAM systems store the data in flat files and repeat the data per statement. An audit record of that form may look like:

```
14 February 2008 10:32:36, 192.168.1.8,192.168.2.222,3920,1521,
"Risk Management DB-Production","ORACLE","aqua_data_studio",
"user2","user2","client.dbasecurity.com",
"select * from t1","select* from t1","user2", …
```

  Note that the dimension information is written per audit record. This means that if you audit one thousand invocations of "select * from t1" you are not storing one thousand times 30 bytes (which is the length of the statement plus some pointers to DIM records) but one thousand times 500 bytes! There is no reason to duplicate the session information per audit record.

Three Things to Remember about Analyzing DAM Storage Requirements

1. Make sure to use a DAM system that normalizes data to avoid expensive duplication of data.
2. Check the average audit record size and your policies and compute the estimated storage requirement for 30–60 days; this is normally the amount of data you need to keep online.
3. Distinguish between what you need to monitor versus what you need to audit to reduce unnecessary storage requirements.

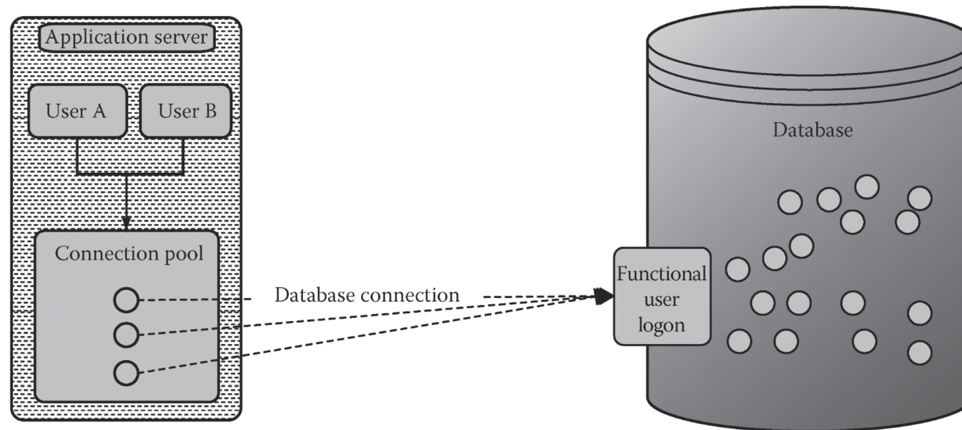

Figure 14.10 Database connection pooling scenario.

14.7 Discussion: Identifying the Real User

One of the hardest problems that DAM solves involves end-user credentials. Most auditors will require full accountability—i.e., every audit record must be associated with a single individual. The same applies to monitoring and security—every action at the database level needs to be mapped to a single user—a unique individual. There are two important scenarios that make it difficult to meet this requirement. The first scenario is that of application servers. As Figure 14.10 shows, application servers manage a pool of connections. These connections all logon to the database when the application server starts up using a single functional ID. Therefore, any audit trail or monitoring system will only be able to mark these activities as being performed by the functional account. Unfortunately, this is not full accountability and is not useful from either a security or an auditing perspective. Instead, you need to be able to tell which database activity was performed on behalf of user A and which database activity was performed on behalf of user B.

The second scenario involves the Oracle instance account. In many environments there are certain activities that you would do from the Oracle instance account. On Unix you might connect to the host using your own operating system account using ssh and from there su into the Oracle instance account and connect using "/ as sysdba". The problem here is again the lack of full accountability. When you look at the audit trail you will see the database user as SYS and the operating system user as oracle (the instance account). What is really required is an indication that these activities are being done by you—i.e., the account from which the su to the Oracle account occurred.

In both these scenarios inspection-based DAM systems can help you with full accountability. Because these systems live outside the database, they can look at additional information that can be used to augment the information on the database connection. This is a very important advantage afforded by DAM that will make your auditors very happy.

Getting end-user credentials in the application server scenario is done either by instrumentation of the code or by instrumentation of the application server. You've already seen that if you use dbms_session.set_context or dbms_session.set_identifier as shown in Figure 14.11 then standard auditing will log this user, and DAM systems will also pick that up as the real user. In this case, when the application gets a connection from the connection pool it makes a call to set_context

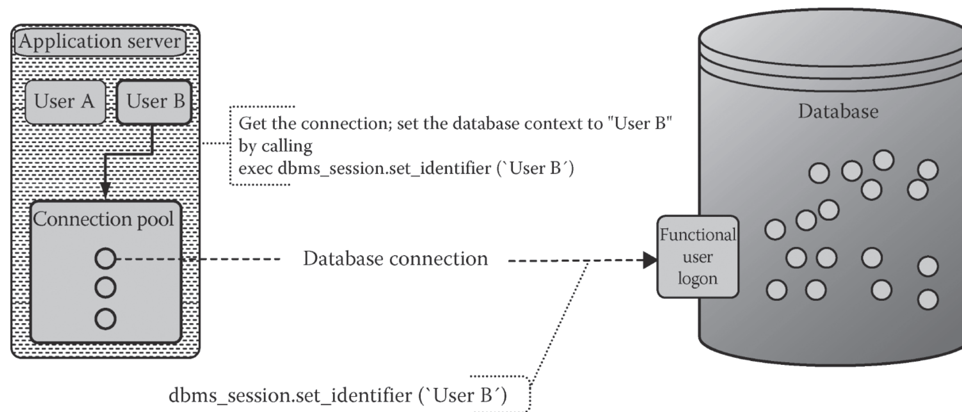

Figure 14.11 Using dbms_session to set the end-user credentials.

and the DAM system knows that all SQL statements on this connection (up until the next call to set_context) belong to user B.

DAM systems can do more. DAM systems can observe what is happening on the application server. They can see the requests when they come into the application server and match them with the requests that the application server makes to the database. This matching still requires instrumentation—at the application server. Don't fall for solutions that tell you that they can inspect the requests coming to the application server and the requests the application server makes to the database and correlate them. Time-based correlation and value-based correlation will only give you the right answer sometimes, but unless you can match these requests up with 100 percent accuracy all of the time, you cannot use the solution for monitoring or auditing.

Similarly, in the scenario involving the Oracle instance account, because the DAM probe lives at the operating system it knows that the connection to Oracle has been made from a shell that was initiated by another shell that is owned by the real user. DAM looks wider than just the database level and therefore can help you achieve full accountability.

Database security

HOWTO Secure and Audit Oracle 10g and 11g

Ron Ben Natan

Foreword by Pete Finnigan

Oracle is the number one database engine in use today. The fact that it is the choice of military organizations and agencies around the world is part of the company's legacy and is evident in the product. Oracle has more security-related functions, products, and tools than almost any other database engine. Unfortunately, the fact that these capabilities exist does not mean that they are used correctly or even used at all. In fact, most users are familiar with less than 20 percent of the security mechanisms within Oracle.

Written by Ron Ben Natan, one of the most respected and knowledgeable database security experts in the world, ***HOWTO Secure and Audit Oracle 10g and 11g*** shows readers how to navigate the options, select the right tools and avoid common pitfalls. The text is structured as *HOWTOs* — addressing each security function in the context of Oracle 11g and Oracle 10g.

Among a long list of *HOWTOs*, readers will learn to —

- Choose configuration settings that make it harder to gain unauthorized access
- Understand when and how to encrypt data-at-rest and data-in-transit and how to implement strong authentication
- Use and manage audit trails, and advanced techniques for auditing
- Assess risks that may exist and determine how to address them
- Make use of advanced tools and options such as Advanced Security Options, Virtual Private Database, Audit Vault, and Database Vault

The text also provides an overview of cryptography, covering encryption and digital signatures and shows readers how Oracle Wallet Manager and orapki can be used to generate and manage certificates and other secrets.

While the book's 17 chapters follow a logical order of implementation, each *HOWTO* can be referenced independently to meet a user's immediate needs. Providing authoritative and succinct instructions highlighted by examples, this ultimate guide to security best practices for Oracle bridges the gap between those who install and configure security features and those who secure and audit them.

CRC Press
Taylor & Francis Group
an **informa** business
www.taylorandfrancisgroup.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
270 Madison Avenue
New York, NY 10016
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

AU4127

www.auerbach-publications.com

Compliments of:

For more information contact:

IBM InfoSphere Guardium
5 Technology Park Drive
Westford MA 01886
guardium@us.ibm.com
ibm.com/software/data/guardium