

Intrusion Detection and Security Auditing In Oracle

White Paper

By Aaron C. Newman, CTO & Founder

**APPLICATION
SECURITY, INC.**

www.appsecinc.com

Tel: 1-866-9APPSEC

E-mail: info@appsecinc.com

INTRODUCTION

At its core, security is all about risk reduction. One of the most effective security practices, *defense-in-depth*, employs multiple layers of protection to reduce the risk of database intrusion. It's analogous to the many defensive layers surrounding a medieval castle: drawbridge, moat, the outer wall, the inner keep, archers manning the wall, soldiers stationed outside the wall, etc. No single level of defense is infallible; all these layers can't ensure the castle will be 100% impenetrable. Yet, these layers of protection can make the castle (and its crown jewels) less vulnerable to the attackers.

Database security is quite similar. Protecting your database (and its crown jewels) encompasses more than a set of permissions. There are actually many layers of protection to consider when protecting our databases. The first layer of defense should always be along the network perimeter, which is typically protected by a firewall. Too frequently, however, even experienced IT security professionals consider the database's network 100% protected at that point. Nothing could be further from the truth. Perimeter security is necessary, but these days it's no longer sufficient. Networks are so interconnected and dependent that it's unreasonable to believe attackers can't get behind, through, or around any firewall. Now is the time to think about securing the database with *layers* — at the perimeter, and at the source of the data.

One crucial layer in database defense is *Vulnerability Assessment*, which involves reviewing, analyzing, and even attacking your own database to find security holes. Why is vulnerability assessment so important? Because in order to patch and fix the holes in your database, you have to know what (and where) they are.

When assessing vulnerabilities on a larger scale, we invariably begin to realize fixing security holes is not always something that can be done on a timely basis. An additional layer of security is needed: *Intrusion Detection/Security Auditing*, a method of monitoring and responding to an attack when it does occur. (Or, in a lesser severe situation, it allows you to respond to valid-yet-potentially malicious activity). In a worst-case scenario, when an attack happens, you naturally want to be notified before the metaphorical barbarians reach your castle's crown jewels. Of course, you also want to locate and barricade the hole through which the barbarians came.

A recent storm of security legislation demands we take extra steps to secure our databases. Now, it is not only a matter of ethics or business savvy that compels us to secure information with a sound defense strategy — it's the law. Simply put, if we do not put the proper defenses in place, there will be consequences. Information security is based on preserving the CIA (Confidentiality, Integrity, and Availability) of systems. Without upholding these basic tenets, a database will not measure up to the requirements of handling commercial data. Without real-time auditing and monitoring data, CIA is impossible to maintain. While discussions continue about the need to provide some level of auditing and monitoring, there is little information available to help define what is appropriate auditing and monitoring. Subsequently, the aim of this paper is to merge "theoretical best-practices" with "real-world practicality" in order to define a usable policy for auditing and monitoring. By following the policies we outline in this paper, you can properly implement a solution that will work well and will not interfere with other aspects of the system.

We have all experienced, or heard horror stories about, determining how to comply with new regulations that require us to audit the access to our data. The legislation is well-meaning, but there is little clarity behind what it actually means. Do we really need to record each individual access to HIPAA data, or can we simply record an individual record for each session that accesses the data? If you try to record just access for a session, you risk legislative non-compliance (and the consequences). Alternately, if you try to record every access to certain data, you risk overwhelming an auditing system with massive reams of records.

A typical scenario follows. A firm's auditor is told he must be in compliance with regulation XYZ, which specifies all access to information type ABC must be audited. Since the vast majority of private and valuable information is stored in a database, the auditor goes to the database administrator (DBA) and asks, "Are we auditing access to all information of type ABC?" The DBA responds, "No. But what would you like me to audit?" The auditor, himself unsure about the exact details of what needs

to be audited, replies, "Hmmm, we need to record every 'read' and 'write' of sensitive data." The DBA, realizing how often sensitive information is accessed on a daily basis, naturally asks, "ARE YOU CRAZY!?" But the auditor can't lose face, so the effort begins to record every 'read' and 'write' access to data type XYZ, requiring 100 GB of archive space per day.

This white paper should clarify what types of activities we really need to examine, and how to keep the "noise" to a minimum.

NATIVE DATABASE AUDITING

The Oracle Database Server provides a fairly robust set of auditing capabilities “out of the box”. This is implemented as a system, which writes activity to tables, log files, or even the Event Viewer on Windows. There are several ways you can record activity in Oracle. But along with each method, there are shortcomings. Below we explore each option.

DATABASE AUDITING

Oracle’s first form of auditing is a subsystem you can use to record failed and successful attempts on the server. Recording connection attempts is useful in being able to discover:

- 1) Who is attempting to connect to the database
- 2) When an attack is taking place
- 3) If an attack was successful.

To enable auditing in Oracle, start by configuring the proper settings in the init.ora file:

```
audit_trail=true
```

This is not enabled by default. You may need to execute the script `ORACLE_HOME\rdbms\admin\cataudit.sql` using the SYS account if the auditing subsystem was not installed, or was uninstalled. This is not usually needed because all auditing tables, views, and procedures are installed by default. You can then control the Oracle auditing subsystem using system commands such as:

```
AUDIT ALL BY user1 BY ACCESS;  
AUDIT SELECT TABLE, UPDATE TABLE, INSERT TABLE, DELETE TABLE BY user1 BY SESSION;  
AUDIT EXECUTE PROCEDURE BY user1 BY ACCESS;
```

The AUDIT command is fairly flexible. You can use it to set auditing on specific objects, commands, or actions. You can also use it to set auditing based on the user taking an action. You can record events on every access. Or, you can record just the first access for a session. To disable auditing, the corresponding NO AUDIT command takes identical parameters to disable the auditing you configured with the AUDIT command. Records are typically stored in a table called `SYS.AUD$`. This can, however, also be stored at the operating system level. To view the values from the table, use one of the following auditing views:

- DBA_AUDIT_EXISTS
- DBA_AUDIT_OBJECT
- DBA_AUDIT_SESSION
- DBA_AUDIT_STATEMENT
- DBA_AUDIT_TRAIL
- DBA_OBJ_AUDIT_OPTS
- DBA_PRIV_AUDIT_OPTS
- DBA_STMT_AUDIT_OPTS

An audit trail also contains a variety of data types. The most usable information is likely:

- Username
- Terminal
- Timestamp
- Object Owner
- Object Name
- Action Name

Auditing at this level can have several shortcomings. First, since auditing is based in the database, it can detract from the system's performance. This is especially true when you attempt to record every access to certain data; the constant reading and writing of auditing can result in substantial disk I/O on the database server, creating a bottleneck that significantly slows down database performance. Another disadvantage: since auditing data is stored in the `SYS.AUD$` table, it ends up sharing disk space with user data, resulting in possible application downtime when log files fill up.

These two disadvantages merit consideration. The bigger issue is this: control of the database implies full control of the auditing system. There is no way to:

- provide segregation of duties
- limit the DBA from disabling the auditing
- limit the DBA from deleting audit records
- limit the DBA from changing auditing configuration.

Segregation of duties is the key to meaningful security and regulatory compliance. The auditing subsystem must retain integrity, and must not be manipulated by the users it is meant to monitor. The "observer" and the "observed" can not be the same person.

This same shortcoming also applies to database intrusions. If I hack into your database, having an audit system to purge will leave you with no forensic evidence. For the audit trail to maintain an acceptable level of integrity, it must be able to withstand an attacker taking control of the database — and not lose the existing audit trail.

AUDIT_SYS_OPERATIONS

The `AUDIT_SYS_OPERATIONS` parameter logs `SYS` user operations to the operating system file that contains the audit trail. This parameter was added to Oracle because, in earlier versions, these actions could not (and still can't) be logged to the `SYS.AUD$` table. To configure the operating system file to which to log data, set the parameter `AUDIT_FILE_DEST` in the `init.ora` file. This parameter does not affect the Microsoft Windows environments since, by default, all audit data is written to the event log. Nor does the parameter affect other parameters, such as `AUDIT_TRAIL`. This parameter is new as of Oracle9i release 2. By default this value is set to `false`. You can enable this setting by adding the following line to the `init.ora` file:

```
AUDIT_SYS_OPERATIONS=true
```

After changing this value, you must stop and restart the database. (For additional details on this security settings, review the Oracle documentation at http://www.oracle.com/pls/db92/db92.drilldown?remark=&word=AUDIT_SYS_OPERATIONS&expand_all=1.)

The `AUDIT_SYS_OPERATIONS` also has shortcomings similar to those found in other Oracle native auditing methods. Auditing directly impacts system performance since it runs in the Oracle software. In addition, audited data is not protected against attackers who successfully break in and gain control of the database. Finally, the data is not protected against the DBA, who is the individual `AUDIT_SYS_OPERATIONS` is designed to track and monitor.

SUMMARY: NATIVE AUDITING BENEFITS AND SHORTCOMINGS

Oracle provides the most comprehensive native audit functionality offered by any commercial database vendor. These built-in database functions allow administrators to roll out a granular auditing system, capable of monitoring and logging any and all database activity. While this functionality is extremely powerful, it is far from an ideal solution.

The ultimate shortcoming of native auditing is that there is no intelligence built into the interfaces. For `AUDIT_SYS_OPERATIONS`, tuning or filtering criteria does not exist. There is simply an on/off switch. No adjustments can be made to what, when, or who is being audited. Database Auditing is slightly better, however the problem persists in that there is no intelligence built into the auditing feature. It takes a significant amount of effort to turn on all the right switches and turn off all the wrong switches, plus this must be done individually for each database server. This creates a management nightmare, as any changes must be configured server-by-server, wasting time and introducing risk of human error.

Oracle Database Auditing includes no logic to detect and highlight malicious activity, nor can this be configured. Database auditing is great at amassing a huge amount of data, but is useless in finding the “needle in the haystack” that is evidence of malicious activity.

Additionally, all of these audit systems store data in a local file or table. This is inherently insecure since it means the data is not well protected. The data is stored locally where the very person it is attempting to audit or monitor can access it. If an attacker breaks into a system, the first action he or she will take is to clear or truncate any audit trail, or even simply delete individual records to hide their footprints. The same will happen for a database administrator that wants to perform actions they are not authorized to perform. They will simply remove any record of the malicious activity from the local audit logs.

WORKING TOWARD AN IDEAL MONITORING SOLUTION

Now that I've outlined what I consider the shortcomings of native database auditing options, let's consider the characteristics of an ideal solution. We will start with the assumption that monitoring data is a complex task, and collecting data is simply part of the work. The trickiest aspects of monitoring a database are:

- 1) Deciding *what* to monitor for
- 2) Handling the *volume* of data that needs to be monitored
- 3) Detecting when something *malicious* has occurred
- 4) Ensuring the *integrity* of the audit data

WATCHING THE DBA

Presently, in most organizations, the DBA is the unrestricted owner of the database. An organization's most critical information is entirely exposed and controlled by this small handful of technologists. This leaves both the DBA, and the entire organization, in a precarious position. The DBAs are afraid they will be blamed for any information leak. The organization is forced to trust a small group of professionals in its technology group.

One way to mitigate risk is to audit and monitor DBA activities. Limit the amount of work a DBA does on a production server. Auditing and monitoring this data should not add significant overhead to any system.

How do you properly audit database activity? Not through native auditing, which fails here because it is fully under the control of the DBAs, who can turn off auditing, clear the audit logs, manipulate an audit record, or even reconfigure auditing to filter their own malicious activity. Auditing should ultimately enable a separation of duty. An ideal audit system is intelligent enough to distinguish database administration accounts, filter out "noise" and irrelevant events, and succinctly illustrate its activities. As well, the system should write audited data to a secure location where even the DBA would not have direct control over the recorded activity.

WATCHING TEMPORARY ACCOUNTS

Another type of activity that requires monitoring is the use of temporary and special accounts. Many companies have procedures through which the database administrator can request a temporary account for others or for themselves to manage databases as required. For instance, the DBA will request that the operations team create a temporary account for which to logon and manage the database when the database goes down or when backups need to be recovered. This account will be set to expire in several hours after which the account will be deleted.

This is an adequate system for reducing the exposure of a malicious database administrator. However it still leaves some exposure in that it is difficult to track exactly what that administrator does during the period of time the temporary account exists. An ideal monitoring and auditing system can provide real value in this situation. A system that can track the activity of the temporary database administrator can help you to easily review the activities and ensure that nothing malicious occurred.

AUDITING ACCESS TO SENSITIVE DATA

Your auditing system should also monitor access to sensitive data in a subset of tables. A typical database contains massive amounts of data. Some of this data is not sensitive at all. However, if other data falls into the wrong hands, the consequences could be disastrous. Auditing every database action can lead to information overload. For instance, if you have a lookup table to map a product to a product ID, there is not much value in auditing access to that table. That table may be accessed thousands of times a day, and auditing all those accesses to the table would result in so much “noise” it could bury a real attack. Other tables may include credit card numbers, payroll information, or social security numbers. Access to these tables should, of course, be audited and monitored closely.

This type of auditing requires that DBAs or application owners decide before-hand what data is sensitive, and define it as such in the auditing system. The auditing system should be able to accept and configure the list of databases, tables, objects, and columns to monitor, and should also be easily configured to monitor specified actions on the table. For instance, if you have static data that is public information, you may not want to audit who performs a `SELECT` from the table. However, you indeed want to record who *modifies* the data. In that case, you need the ability to audit any `UPDATE`, `DELETE`, or `INSERT` made by any user.

FLEXIBILITY TO FILTER RESULTS

There exists a real need to filter *how* data is audited based on *who* is accessing the data. For instance, HIPAA regulations require strong accountability to access of patient records. If a system administrator accesses patient Jane Smith’s medical records on June 15th, there must be a record of the action to ensure accountability for the data. On the other hand, if the patient’s doctor accesses the data twenty times in a day, there’s little value in recording this activity multiple times. Auditing should record unauthorized users’ attempts to access data — yet should be flexible enough to minimize the “noise” level. Keeping “noise” level down can be accomplished by minimizing the recording of activity performed by *authorized* personnel. An ideal auditing solution allows you to filter auditing based on factors such as account name, source of activity, and the time of the activity.

ATTEMPTS TO CIRCUMVENT AN APPLICATION

Another common problem that should be monitored and audited is when users circumvent an application and connect directly to the database. This is especially problematic with two-tier architectures, such as a Visual Basic executable connected directly to a database. Or, if you are using a three-tiered architecture such as a web application connected to the back-end database. Both architectures require you monitor for the use of utilities such as Microsoft Access, Microsoft Excel, or even SQL*Plus to connect directly to the database. Users doing this may simply be trying to make their job easier, but they are, in effect, opening a security hole in the database. On frequent high-risk scenario involves users placing a linked Microsoft Excel spreadsheet on an open file share, allowing other users to see the results of their work in the database. Unfortunately, this linked spreadsheet can be manipulated to pull back other information from the schemas or tables.

There are two ways to audit for this particular problem. One way is to watch specifically for people using tools such as Microsoft Office to access the database. To do this effectively, you must be able to list the most common applications that a curious or well-meaning user might employ. The other strategy is to audit any connections not from the expected application name. For instance, consider a database connected to via an application called `HRPayrollApp`. You should be able to configure the auditing system to alert you when someone connects to the database using any application *other than* `HRPayrollApp`.

AUDITING EXCEPTIONS

Audit systems should be able to “approve” traffic to prevent valid activity from continuing to trigger alerts. For instance, an application may legitimately access data in the database that is being monitored by the auditing system. This is good to know when you first install and set up the auditing system. However, it becomes “noise” after you see that data a few hundred times. By “noise”, in this case, I’m referring to scenarios where you gain no value from seeing the alert, and it only contributes to drowning out other more valuable audit data. If you get 10,000 audit records a day, it’s going to be hard to see the one record that really matters because it’s buried in information overload. This is why it is so important for an effective audit system to reduce the number of items audited, and only catch the things we care about. This goal is accomplished by allowing “exceptions”. For instance, an exception might say: “Do not record access to data when a specific SQL statement comes from user XYZ from machine ABC”. When any other access to the data occurs, the audit system should record the activity. As well, a proper auditing system should be able to record any activity that does not match specific criteria. For instance, the system should allow you to say: “Record all activity except for SQL statements from application XYZ.”

MANIPULATING WEB APPLICATIONS

Monitoring database activity originating from a web application is also important. An ideal audit solution should detect when someone manipulates a web application. How is this accomplished? To start with, the monitoring system has to learn what kind of data the web application sends to the database. This activity would compile a complete list of each SQL statement executed by the web application. Then, when a query is executed, the audit solution should compare it against a list of “known” SQL statements. If the SQL statement has been modified — for example, by injecting additional SQL statements into the web application — the SQL statement will not match the list of “known” SQL statements. The audit system then triggers an alert, and the monitoring system notifies you of the unauthorized activity.

How would a system go around getting a complete list of SQL statements generated by the web application? Three methods follow:

- Method #1: Run the monitoring system in learn mode during regular operations
- Method #2: Feed the application a list of the valid SQL statements
- Method #3: Enumerate the list of SQL statements

Recording the list of valid SQL statements requires a certain amount of intelligence. You cannot simply record all SQL statements and then use that as the list, because these SQL statements are parameterized. For instance, you may have a SQL statement to look for all the orders for a customer. This would generate the following SQL statement:

```
SELECT * FROM ORDERS WHERE CUSTOMER_ID = 5
```

Searching on a different customer would yield:

```
SELECT * FROM ORDERS WHERE CUSTOMER_ID = 6
```

Are these one or two different SQL statements?

For the purposes of watching for the manipulation of a web application, these are the same statement, and the second example should not generate an alert. The monitoring solution should be able to normalize these SQL statements into representations such as the following:

```
SELECT * FROM ORDERS WHERE CUSTOMER_ID = ?
```

Then, as each statement is reduced to its normalized form, it can be compared and effectively evaluated as the same SQL statement.

On the other hand, the following SQL statement should not be evaluated as the same, because the normalized form is different:

```
SELECT * FROM ORDER WHERE CUSTOMER_ID = 10 UNION SELECT USER, PASSWORD FROM  
master.dbo.syslogins
```

IDENTIFYING UNUSUAL ACTIVITY

Another important aspect of a monitoring system is its ability to identify atypical activity, i.e., activity that is unusual, and may be in violation of corporate policy. An ideal tool should classify activity into patterns, and based on those activity patterns, identify usage patterns. This is useful in determining if unauthorized activities are taking place, or if corporate policies are being broken.

Consider the mapping of typical administrator activity. If a monitoring system can detect when an administrator makes an uncharacteristic act, this can help ferret out wrongdoings. For example, an administrator breaks corporate policy by remotely administering the database from a home computer. Or perhaps the administrator logs into the network late at night from a remote office, raising a “red flag” for an attack from an internal employee.

An auditing tool should be able to properly characterize your system, then monitor for attacks, breaches in security, valid users performing unauthorized activities, and violations of corporate policies.

KNOWN ATTACKS

It is imperative that your database monitoring system detect and recognize attacks. Attacks come in many forms. When an attack occurs on your system, it should notify you that you’re under attack. Below is a sampling of the type of attacks a monitoring system should pick up on:

1. Buffer overflows being executed from PL\SQL
2. Web application attacks
3. Privilege escalations
4. Accessing OS resources
5. Password attacks
6. Pen Testing or hacker tools used against the database
7. Database starting and stopping

APPRADAR – THE LEADING SOLUTION

AppRadar is a patent-pending revolutionary new monitoring and auditing solution for protecting data at the source – the database. It is complementary to the many monitoring and intrusion detection tools available for the network, as AppRadar is focused squarely on database applications. With the unique ability to audit user activity and alert on database attacks, AppRadar fills a gap left by native Database Audit tools and by traditional Intrusion Detection Systems.

AppRadar provides a centralized approach to database auditing. Architected to scale to the largest of IT infrastructures, AppRadar uses distributed sensors to monitor and collect data on enterprise databases. Management is accomplished via a single web-based interface, the AppSecInc Console. The Console provides role based access control creating a clean separation of duties between those that can configure database monitoring and those that are monitored. Policy can be centrally created and automatically distributed, eliminating the enormous amount of work associated with managing a large deployment of native database auditing. The AppSecInc Console utilizes its own SQL Server database as a central store for all audit data, allowing for aggregate reporting, and providing better protection for the audit data.

AppRadar provides alerts on attacks that traditional Intrusion Detection systems are not designed to detect. IDS engines such as Snort are very useful at detecting attacks from the network perimeter, but do nothing to identify attacks within the database. These engines are not designed to see a malicious user connecting to the database executing SQL attacks, SQL Injection, or even detect an unauthorized user trying to access password hashes. Traditional Intrusion Detection Systems focus on perimeter security, while AppRadar is designed to detect misuse (whether from authorized or unauthorized users) of your most valuable assets – within the database.

AppRadar is a highly flexible solution designed to work effectively out-of-the-box with the ability to be customized to handle the unique constraints of any environment. AppRadar is a hybrid system, providing a combination of rule-based signatures, user monitoring, and behavioral analysis. By combining these into a holistic approach, the highest levels of protection can be achieved.

AUDITING

AppRadar provides the ability to audit database activity. This includes monitoring for changes to objects, system tables, permissions, configuration systems, and many other options. AppRadar can be configured to audit specific objects or entire object classes. This can be customized to suit your environment by determining what data is sensitive and then specifically setting AppRadar to monitor those objects.

Below are some of the audit events AppRadar can monitor for:

ALTER DATABASE Statement	CREATE PFILE Statement	DROP USER Statement
ALTER FUNCTION Statement	CREATE PROCEDURE Statement	DROP VIEW Statement
ALTER INDEX Statement	CREATE PROFILE Statement	INSERT Statement
ALTER MATERIALIZED VIEW Statement	CREATE SCHEMA Statement	Listener command - SERVICES
ALTER PACKAGE Statement	CREATE SEQUENCE Statement	Listener command - STATUS
ALTER PROCEDURE Statement	CREATE TABLE Statement	Listener command - STOP
ALTER PROFILE Statement	CREATE TABLESPACE Statement	Listener command - VERSION
ALTER TRIGGER Statement	CREATE TRIGGER Statement	NOAUDIT Statement
ALTER VIEW Statement	CREATE USER Statement	SELECT Statement
AUDIT Statement	CREATE VIEW Statement	UPDATE Statement
CREATE CONTROLFILE Statement	DELETE Statement	Use of ALTER SESSION
CREATE DATABASE Statement	DROP DIRECTORY Statement	Use of ALTER SYSTEM
CREATE DIRECTORY Statement	DROP FUNCTION Statement	Use of ALTER USER
CREATE FUNCTION Statement	DROP INDEX Statement	Use of GRANT
CREATE INDEX Statement	DROP LIBRARY Statement	Use of GRANT DBA
CREATE LIBRARY Statement	DROP MATERIALIZED VIEW Statement	Use of GRANT SYSDBA
CREATE MATERIALIZED VIEW Statement	DROP TABLE Statement	Use of GRANT SYSOPER
CREATE PACKAGE Statement	DROP TABLESPACE Statement	Use of REVOKE
	DROP TRIGGER Statement	

AppRadar also provides the ability to monitor the activity of specific accounts. For instance, you can configure AppRadar to monitor the activities of the Database Administrator (DBA). AppRadar can also be configured to capture the activity of all users other than the Web Application.

ATTACK SIGNATURES

AppRadar provides the most comprehensive library of database attack signatures. AppRadar covers the following attacks out of the box:

BUFFER OVERFLOWS

AppRadar monitors for attacks that take advantage of buffer overflow vulnerabilities. Oracle is susceptible to a number of buffer overflows which result in either a database crashing or the memory in the stack being overwritten. This can result in an exception being thrown, or worse yet, an attacker taking full control of the system. AppRadar Sensors can pick up on buffer overflow attack patterns such as the following:

BFILENAME buffer overflow	NUMTOYMINTERVAL buffer overflow
Database link buffer overflow	SERVICE_NAME buffer overflow
DROP_SITE_INSTANTIATION buffer overflow	TIME_ZONE buffer overflow
FROM_TZ buffer overflow	TO_CHAR buffer overflow
INSTANTIATE_OFFLINE buffer overflow	TO_TIMESTAMP_TZ buffer overflow
INSTANTIATE_ONLINE buffer overflow	TZ_OFFSET buffer overflow
NUMTODSINTERVAL buffer overflow	

WEB APPLICATION ATTACKS

Rules within this category can be enabled to monitor against possible access-related attacks. Attacks may include attempts to elevate privileges and gain access to powerful resources within an Oracle database

PRIVILEGE ESCALATION

It is possible for a low-privileged user to exploit Oracle vulnerabilities to effectively bypass access controls. This category of rules alerts on the exploitation of these kinds of vulnerabilities.

ACCESSING OPERATING SYSTEM RESOURCES

This category focuses on monitoring database features that allow operating system access. For example, by changing the UTL_FILE_DIR parameter, Oracle can be fooled into allowing SYS.UTL_FILE to overwrite important files on the operating system thus giving access to the OS through a database attack.

PASSWORD ATTACKS

Attempts to guess passwords by trying likely combinations of characters or exploiting certain Oracle vulnerabilities are simplistic attacks that can be used against a database.

SYSTEM EVENTS

These rules uncover system level events such as the starting and stopping of the database being monitored and the starting and stopping of the AppRadar Sensor.

CONCLUSION

Monitoring your database applications is a critical component of achieving a strong defense-in-depth around your sensitive data. However, to be efficient and effective you must use the right combination of tools. Monitoring should never replace other layers in the security stack, instead it should complement the existing pieces. Database intrusion detection and security auditing continues to grow in importance because of the rising volume of successful database attacks, and the resulting security legislation and regulations, including:

- Payment Card Industry Data Security Standard (PCI-DSS)
- Sarbanes-Oxley Act
- HIPAA (Health Insurance Portability and Accountability Act)
- European Union Data Protection Directive
- California's Database Security Breach Notification Act (California Senate Bill 1386)
- Gramm-Leach-Bliley Act
- Federal Information Security Management Act

Clearly, database intrusion detection and security auditing comes with its complexities. Monitoring your databases is a useful tactic, but only if used in conjunction with a well-conceived and balanced security plan. Database monitoring should be a layer of defense augmenting your overall database security strategy. When used in conjunction with vulnerability assessment, encryption, and database integrity solutions, an extremely solid security solution can be implemented. When considering the use of database monitoring be sure to select a tool that will work well with other database security products. This will ensure an effective and holistic approach to security by incorporating and integrating all the different layers. By doing so, you will more effectively fortify your castle (database) and crown jewels (sensitive data) from the barbarians of these modern times.

ABOUT APPLICATION SECURITY, INC. (APPSECINC)

AppSecInc is the leading provider of application security solutions for the enterprise. AppSecInc's products - the industry's only complete vulnerability management solution for the application tier - proactively secure enterprise applications at more than 400 organizations around the world. By securing data at its source, we enable organizations to more confidently extend their business with customers, partners and suppliers while meeting regulatory compliance requirements. Our security experts, combined with our strong support team, deliver up-to-date application safeguards that minimize risk and eliminate its impact on business. Please contact us at 1-866-927-7732 to learn more, or visit us on the web at www.appsecinc.com.