Online Detection of Malicious Data Access Using DBMS Auditing

José Fonseca
CISUC, University of Coimbra
Dep. of Informatics Engineering
3030 Coimbra - Portugal
+351 239 790 000
josefonseca@ipg.pt

Marco Vieira
CISUC, University of Coimbra
Dep. of Informatics Engineering
3030 Coimbra - Portugal
+351 239 790 000
mvieira@dei.uc.pt

Henrique Madeira
CISUC, University of Coimbra
Dep. of Informatics Engineering
3030 Coimbra - Portugal
+351 239 790 000
henrique@dei.uc.pt

ABSTRACT
This paper proposes a mechanism that allows concurrent detection of malicious data access through the online analysis of the Database Management Systems (DBMS) audit trail. The proposed mechanism uses a directed graph representing the profile of valid transactions to detect illegal accesses to data, which are seen as unauthorized sequences of Structured Query Language (SQL) commands. The paper proposes a generic algorithm that learns the graph representing the profile of the transactions executed by the users. This mechanism can be used to protect traditional database applications from data attacks as well as web based applications from SQL injection types of attacks. The proposed mechanism is generic and can be used in most commercial DBMS, adding concurrent detection of malicious data access to classical database security mechanisms. The paper presents a practical example of the implementation of the proposed mechanism using Oracle 10g. The Transaction Processing Performance Council benchmark C (TPC-C) and a real database installation were used to assess the detection mechanism and learning algorithm.

Categories and Subject Descriptors
H.2.7 Database Administration: *Security, integrity, and protection*

General Terms
Management, Security.

Keywords
Intrusion detection, SQL injection, DBMS auditing.

1. INTRODUCTION
A major problem faced by organizations today is the protection of their data against malicious access or corruption. Traditional database security mechanisms offer basic security features such as authentication, authorization, access control, data encryption, and auditing. However, these mechanisms do not assure protection against exploiting database applications bugs and are very limited in defending data from attacks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'08, March 16-20, 2008, Fortaleza, Ceará, Brazil.
Copyright 2008 ACM 978-1-59593-753-7/08/0003…$5.00.

According to a Computer Crime and Security Survey [5] done by the FBI in 2006, around 32% of the respondents had reported unauthorized access to information estimating a loss of $ 10.617.000 and a loss of $ 6.034.000 due to theft of proprietary info. Up to 52% of the respondents reported unauthorized use of computer systems and 10% did not know if they have been attacked. Furthermore, 92% of the correspondents reported more than 10 web site incidents.

Masquerade attacks where people hide their identity by impersonating other people on the computer are one of the most frequent forms of security attacks [9, 10, 15, 16], including in the database domain. Another common database attack is SQL injection in web applications, where unchecked input is passed to a back-end database for execution. The attacker can perform this by simply changing the SQL query sent to the server, getting access to sensitive data.

One important security mechanism in Database Management Systems (DBMS) is auditing [14]. In many database applications auditing is required by law, in order to assure that any action in the database can be traced back to an individual user/program if needed (e.g., hospitals, banking, electronic voting, etc). In less demanding applications, the audit trail is switched on only when the Database Administrator (DBA) suspects that the database is being subjected to anomalous accesses. Of course, the auditing causes some performance overhead, which is in general not very relevant unless the server is running close to its loading limits [14, 13, 18].

The audit trail can be used by the DBA to perform a posteriori analysis of the accesses to the data in order to identify potential malicious data accesses. However, the analysis of the audit trail is a difficult (or even impossible in databases with hundreds of users performing operations simultaneously) and time consuming task. Furthermore, DBMS lack in intelligent auditing tools able to help in the audit process [19]. More important, auditing is only useful for diagnosis or investigation purposes of past security attacks.

The general lack of concurrent detection of malicious data accesses capabilities in commercial DBMS is an important limitation when it is necessary to assure a strong data security policy. A practical mechanism for concurrent audit trail analysis in DBMS will provide an extra layer of security that cannot be assured by the basic DBMS security mechanism or by operating systems and networking intrusion detection. It is worth noting that malicious actions for a database application may not be seen as malicious by existing intrusion detection mechanisms at the network or the

operating system levels, which means that they would not be detected. For example, inside attacks (e.g., a disgruntled employee that may access and damage critical private data) are particularly difficult to detect and isolate, as the attacks are carried out by legitimate users that may have access rights to data and system resources. Furthermore, daily routine and long established habits tend to relax many security procedures. Even simple things such as choosing strong passwords and purging periodically unused database accounts are often neglected in many organizations [11].

The mechanism for concurrent detection of malicious data access proposed in this paper adds real-time analysis capabilities to the auditing mechanisms of DBMS. This way, a data attack can be detected and stopped in due time (e.g., by killing or isolating the database session of the attacker) while the mechanism may call the attention of the DBA (by sending a screen alert, an email or an SMS message). The DBA does not have to spend time analyzing the audit records because they are being analyzed on the fly and the malicious behaviors detected are immediately reported to the DBA. Additionally, the audit trail analysis mechanism can also be used to help in the traditional off-line analysis of audit entries. This mechanism can be easily implemented and used in commercial DBMS, as shown in the examples presented in this paper.

The proposed mechanism, named MDAD – Malicious Data Access Detector, includes two phases: learning and detection. The DBMS must be configured to record the audit entries for basic data access operations (select, insert, delete and update). This will feed the learning phase and the result is the graph of the transaction profiles for all the transactions recorded in the audit trail. These learned graphs are stored and used later on by the detection engine to detect malicious commands.

The structure of the paper is as follows. Section 2 provides some background on security in DBMS. Section 3 presents the proposed mechanism of learning the transaction profiles from audit entries and the corresponding detection mechanism. Section 4 presents the evaluation of the proposed mechanism using the TPC-C standard benchmark and a real database. Section 5 concludes the paper.

2. BACKGROUND AND PREVIOUS WORK

General methods for intrusion detection in computer systems are based either on pattern recognition or on anomaly detection. Pattern recognition is the search for known attack signatures in the commands executed. Anomaly detection is the search for deviations from an historical profile of good commands.

Schonlau et al [16] evaluated several anomaly detection approaches and concluded that methods based on the idea that operating systems commands not previously seen in the training data may indicate an intrusion attempted, are among the most powerful approaches for intrusion detection. The approach proposed in this paper uses this idea, extending it to the detection of malicious data accesses based on a set of SQL commands. However, unlike intrusion detection approaches used in distributed systems, that usually rely on sequences of predefined number of commands (normally a small number) or assume the commands are unrelated, in our approach, the SQL commands and their order in each database transaction are relevant.

The main goal of security in DBMS is to protect the system and the data from intrusion and unauthorized accesses, even when the potential intruder gets access to the machine where the DBMS is running. To protect the database from intrusion, the DBA must prevent and remove potential attacks and vulnerabilities. The system vulnerabilities are an internal factor related to the set of security mechanisms available (or not available at all) in the system, the correct configuration of those mechanisms (which is a responsibility of the DBA), and the hidden flaws on the system implementation. Vulnerability prevention consists of guarantying that the software used has the minimum vulnerabilities possible and this can be achieved by using adequate DBMS software. On the other hand, as the effectiveness of the security mechanisms depend on their correct configuration and use, the DBA must correctly configure the security mechanisms by following administration best practices. Vulnerability removal consists on reducing the vulnerabilities found in the system. The DBA must pay attention to the new security patches released by software vendors and install those patches as soon as possible. Furthermore, any configuration problems detected on the security mechanisms must be immediately corrected.

Security attacks are an external factor that mainly depends on the intentionality and capability of humans to maliciously break up into the system taking advantage of potential vulnerabilities. The prevention against security attacks includes all the measures needed to minimize (or eliminate) the potential attacks against the system. On the other hand, attack removal is related to the adoption of measures to stop attacks that have occurred before.

In spite of all the classical security mechanism developed in the database area, current DBMS are not well prepared for high-assurance privacy and confidentiality [2]. A very important component for the new generation of security aware DBMS are mechanisms able to automatically detect malicious data accesses and intrusion [1].

Recent works have addressed real-time (or concurrent) intrusion detection and attack isolation in DBMS, and this issue is clearly getting more and more attention. DEMIDS is a misuse detection system tailored to relational database systems. It uses audit logs to derive user profiles that describe typical behavior of users in the DBMS [4]. Chung introduces the notion of distance measure and frequent item sets to capture the working scopes of users using a data mining algorithm. Although also using audit log, our approach is different from [4] as it is applied at the very fine grain of SQL commands and transactions, instead of group of users' profiles.

In [3] a real-time intrusion detection mechanism based on the profile of user roles is proposed. An intrusion attack and isolation mechanism was proposed in [8]. This mechanism uses triggers and transaction profiles to keep track of the items read and written by transactions isolates attacks by rewriting user SQL statements. The use of data dependency relationships and Petri-Nets to model normal data update patterns was proposed in [6] to detect malicious database transactions. Using fingerprints for intrusion detection in databases is addressed in [7].

3. TRANSACTIONS LEARNING AND MALICIOUS ACCESS DETECTION

In a typical database environment transactions are programmed in the database application, which means that the set of transactions remains stable, as long as the database applications are not

changed. For example, in a banking database application users can only perform the operations available at the application interface (e.g., withdraw money, balance check account, etc). No other operation is available for the end-users. Normally, end-users cannot execute ad hoc SQL commands. So, it is possible to use transaction profiles for the detection of malicious data accesses with a reduced risk of false alarms.

Typically, there are several groups of users in a database environment, according to the transaction profiles they execute. There are regular database end-users executing predefined transactions by means of a database application, and a small set of exceptional users that may belong to decision support, DBA or developers that explore data for strategic decisions by executing all kinds of ad-hoc SQL commands. The target group of users of our application is the regular database clients, which constitute the vast majority of database users.

In [18] the authors addressed the detection of malicious DBMS transactions was addressed with the assumption that the transaction profiles (graph of the sequence of SQL commands in a transaction) was known in advance, and provided manually to the detection mechanism. In our opinion, this requirement is hard to fulfill in real and complex database installations. Thus, in this paper we propose a new approach based on automatic transaction learning.

The proposed mechanism uses the profile of the transactions implemented by the database applications (authorized transactions) to identify user attempts to execute other SQL commands. A database transaction is represented by a directed graph describing the different execution paths (sequences of selects, inserts, updates, and deletes) from the beginning of the transaction to the commit or rollback command. The nodes in the graph represent commands and the arcs represent the valid execution sequences. Depending on the data being processed, several execution paths may exist for the same transaction and an execution path may include cycles representing the repetitive execution of sets of commands (a typical example of cycles in a transaction is the insertion of a variable number of lines in a customer's order). The transaction ends with a commit or rollback command.

The mechanism for online detection of malicious data access consists of two main phases (see Figure 1): transactions learning and malicious data access detection. Both phases use the database audit trail. In the learning phase, the audit trail is used offline to generate the graphs representing the valid transactions. In the detection phase, the audit trail is used online to obtain the sequence of commands (transactions) executed by each user, which is compared to the learned graph in order to detect unauthorized transactions.

It is worth noting that learning and detection phases may occur in a recurrent manner. In fact, when a new database application is deployed the learning phase must be revisited. Furthermore, as it is easy to see, the transactions learning depends on the utilization profile of the database. In many cases, large database applications include functionalities that are only executed from time to time, for example at the end of the week or end of the month. Until the DBA is not confident with the learned transaction profile the detection may not act drastically on the session (e.g., may not kill sessions that are considered as malicious). Instead the DBA should analyze those situations first and, possibly add the detected transaction to the learned profile. In practice, we expanded the detection phase into two phases: Conditional Detection and Regular Detection (Figure 1). When the DBA considers the conditional detection phase is completed then the system goes to the regular detection phase. In this phase if a malicious transaction is found a more defensive action may be executed. If there is an upgrade of the database application then the system should go to the learning phase again (including or not simultaneous conditional detection).

An important aspect is that the nodes in the graph do not represent concrete commands as commands may differ among executions. For example, consider the following SQL command to select the data from a given customer: *select name, address, phone from customer where name='John Carter'*. The name in the select criteria (name=?) depends on the target customer. This way, instead of considering concrete commands we have to represent those commands in a generic way. For example, the command to select data from a given customer can be represented by the following attributes: command type (select), target object (table customer), columns selected (name, address, and phone), and restriction field (name).

The audit entries must include the following information for each audited command: Username, Session ID, Command ID, Transaction ID, Action executed, Object name, Object owner, and Timestamp of the action.

This information corresponds to the information audited in typical DBMS, which normally can be configured to store different levels of detail of the audited data.

Although auditing is mandatory in high security database applications, in many less demanding applications the audit trail is only switched on when the DBA suspects that the database is being subject to anomalous accesses. In both cases, the proposed MDAD mechanism adds on-line analysis to audit trail, which helps the DBA in providing a quick response to attacks. In critical applications the time between a malicious action and its detection is of major importance and every second of delay may represent loss of privacy, risk of data destruction, and propagation of corrupted data after the attack.

As previously mentioned, the proposed detection technique does not apply to users that execute ad-hoc queries, as there are no predefined transaction profiles for ad-hoc queries. However, ad-hoc queries are used in decision support system and are not executed in typical database applications, as this type of queries would ruin the performance of the database system. Furthermore, it is quite easy to exclude a given user (e.g., a trusted user that could execute ad-hoc queries even in a traditional database) from the auditing trails that feed our detection mechanism, avoiding this way false positive detection alarms.

3.1 Transaction profile learning using audit entries

Learning transactions consists of identifying the authorized transactions and representing those transactions as a directed graph specifying the sequences of valid commands, where each node represents a command and each arc represents a valid execution sequence. The goal is to automatically learn the transactions profiles contained in the audit trail and save them as a directed graph to be used in the detection phase. Obviously, learning algorithms must be executed over audit trail collected in controlled condi-

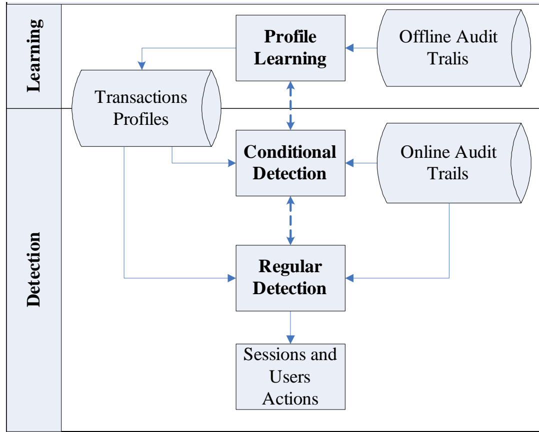

Figure 1. MDAD building blocks and workflow.

tions that guarantee the system is free of data attacks (which would potentially lead to the identification of malicious transactions as authorized ones).

When a user connects to the database and establishes a session, all the commands executed by that user are associated to a transaction. Thus, the user cannot escape to the transaction mechanism: when one transaction ends a new transaction begins. Two types of transactions can be considered: read-only transactions and regular (i.e. read and write) transactions. The read-only transactions are groups of queries mainly used to show information to the user on the screen or printer. Typically, for these transactions there is no information in the audit trail about their start or end because nothing is changed in the database. Actually, when developing applications programmers do not include commits at the end of read-only transactions because they are not needed.

One of the key points in the learning phase, and in the detection phase as well, is the detection of the first command of a transaction as in many commercial DBMS, such as Oracle 10g [12], the commit and rollback commands are not recorded in the audit trail. This way, the detection of the first command of a transaction is done by analyzing the transaction ID associated to the commands in the audit trail. This ID is normally null at the beginning. It changes to a non null value in the first writing command (insert, update, delete) and keeps the same value until the transaction ends, even if there are read-only commands in the middle or at the end of the transaction. In the next transaction the transaction ID will be null again until the first writing command is issued (typically, read-only commands in the beginning of a transaction have a null value associated). An important aspect is that the transaction ID values are always different from one transaction to another.

As commit and rollback commands are not recorded in the audit trail it is impossible to know if a transaction ends because of a commit or a rollback. Also, when there is a read only transaction (for which commands have a null transaction ID) and the start of the next transaction is a select command, the transaction ID maintains its null value and it is impossible to detect the start of the second transaction by simply reading the transaction ID. To solve these problems the Learning phase was split into three steps: **First-Learning**, **Extraction of Read Only Transactions** and **Final-Learning**.

The input of the First-Learning step is the audit trail previously collected and its objective is to split the trail into small groups of transactions based on the transaction ID information. These groups of transactions consist of regular transactions that may have one or more read only transactions attached at the beginning (see Figure 2). This mixture of several transactions occurs due to the fact that the end of read-only transactions is not explicitly recorded in the audit trails. Of course, when one regular transaction is preceded by another regular transaction, they are correctly identified in this step.

The result of the First-Learning step is used in the Extraction of Read Only Transactions step, where read only transactions are isolated by subtracting the groups of transactions from each other. The subtraction of the two transactions leads to the identification of a read only transaction when the two transactions differ one from the other by select commands at the beginning. As shown in Figure 2, this set of commands (representing the read-only transaction) is the result of the subtraction. The result of this step is the read-only transactions (and groups of read-only transactions seen as a single read-only transaction).

The reasoning behind the subtraction of transactions to isolate the read-only transactions is the following: as the normal (i.e., writing) transactions are well-defined by the transaction ID, reading commands that form read-only transactions may occur at the beginning of different transactions, which means that the read-only portions can be isolated by simple command subtraction.

The last step is the Final-Learning step where the off-line audit trail is processed along with the read-only transactions previously obtained. Again the audit trail is split into groups of transactions and the regular transactions are obtained by subtracting the read-only transactions from the beginning of those groups. Figure 2

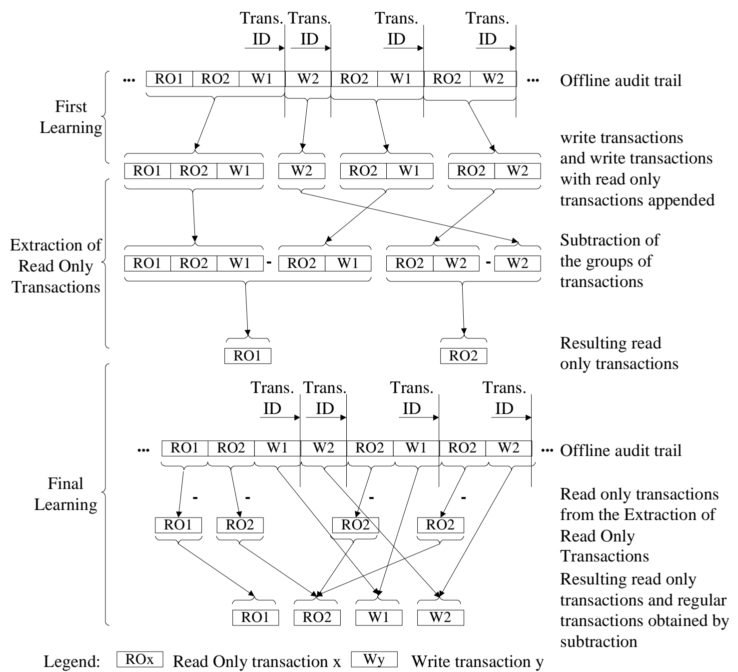

Figure 2. Learning phase in detail.

shows a visualization of this process and explanation comments.

Database transactions fall in one of the following transaction profiles that cover all the possibilities: linear (no branches or loops), with branches, with loops, with loops inside loops, with loops inside branches, and with branches inside loops.

Except for the last type of transaction profile, all the others are easily learned by an algorithm that can learn a linear transaction and loops. When a branch exists it is treated as a different transaction. The learning algorithm implements the detection of linear transactions and transactions with loops. The transactions with branches are split into as many transactions as there are branches.

3.2 Malicious data access detection

Having concluded the learning phase, the MDAD is ready to detect malicious data accesses. The audit trail is then used to concurrently obtain the sequence of commands executed by each user, which is compared to the profile of the authorized transactions to identify potential malicious commands. To minimize the storage overhead, the audit entries may be deleted as soon as they are processed and no malicious data access is detected. If an attack is detected the audit entries are kept for future reference.

An important aspect is that the detection is done at SQL command level. That is, it is not necessary to reach the end of the transaction in which the suspicious command was found to detect a potential attack. All the transactions that have suspicions commands (i.e., that deviate from a known authorized profile) are immediately considered malicious.

The detection mechanism can be implemented inside the DBMS, outside the DBMS (in the same machine) or even in a different computer (to reduce performance overhead). In our current implementation, the whole detection mechanism is implemented outside the DBMS and in a different computer.

If a malicious transaction is detected one or more of the following actions may be executed, depending on the DBA choice: notify the DBA about the attack, immediately disconnect the user session in which the malicious transaction was attempted, or activate a damage confinement and repair mechanism [8].

As mentioned before, the detection phase may work in Conditional Detection mode where the erroneous transactions are analyzed and evaluated by the DBA. If they are considered valid transactions they should be added to the learned transaction profiles. If they are considered suspicious transactions, the DBA should investigate why they were executed. If there are new functionalities or reconfiguration of the software, the Regular Detection mode may be changed to Conditional Detection in order to update the transaction profiles collection.

4. EVALUATION AND RESULTS

This section demonstrates the use and discusses the evaluation of the proposed intrusion detection mechanism.

4.1 Setup and evaluation scenarios

We used two different database application scenarios for the evaluation experiments:

- A well-known database performance benchmark, the TPC-C [17], which provides us with a controlled database environment quite adequate for initial evaluation of the learning algorithm and for the evaluation of performance overhead and latency. The coverage and latency of the detection mechanism was mainly evaluated using this application scenario (i.e., the TPC-C).
- A real (and large) database application to assess in particular the transaction learning curve in a real situation. This allows us to assess the need for conditional detection due to false positives resulting from incomplete transaction learning.

The TPC-C performance benchmark [17] is an OLTP workload. It is a mixture of read only and update intensive transactions that simulate the activities found in complex OLTP application environments. The performance metric reported by TPC-C is a "business throughput" measuring the number of orders processed per minute. Multiple transactions are used to simulate the business activity of processing an order, and each transaction is subject to a response time constraint. The performance metric for this benchmark is expressed in transactions-per-minute-C (tpmC).

The SCE is an application currently in use in the Central Service of Sterilization of a large hospital. The SCE is an administrative application used to manage the whole process of the sterilized material to and from all services in the hospital. This workflow comprises the reception of the material, the selection and the sterilization of the material within a central with vapor autoclaves and ethylene oxide, various modes of drying, packaging, sealing, request and delivery. In every phase of the process the material is subject several times to inspections.

As shown in Figure 3, the setup used in the evaluation experiments with TPC-C includes three computers connected through a 100 Mbit LAN Ethernet broadband router/switch. The database server is a desktop AMD Athlon XP 2800+ with 1GB RAM, one 180GB SATA hard disk, running the Oracle 10g R2 DBMS over the Mandriva Linux 2006 operating system. The machine used for the malicious data access detection is a 1.6 GHz notebook Pentium 4, with 256MB RAM, one 30GB hard disk, running the Windows XP SP2 operating system and Oracle 10g R2 client installed. The machine emulating the TPC-C terminals is 3 GHz desktop Pentium 4, with 480MB RAM, one 80GB hard disk, running the Windows XP SP2 operating system and Oracle 10g R2 client installed. Note that the hardware features of the different machines do not have particular impact on the experimental results and are mentioned for the sake of completeness.

4.2 Evaluation of the learning algorithm

The learning algorithm was first evaluated using the TPC-C benchmark. TPC-C has five transaction profiles called Delivery, NewOrder, OrderStatus, Payment and Stock-Level. OrderStatus and StockLevel are read-only transactions. For the evaluation of the learning algorithm an audit trail was generated corresponding to one hour execution of the benchmark. This trail comprised 989,540 commands corresponding to the execution of 96,585 transactions from 50 sessions. As a result we obtained 42 different transactions in the first step (First Learning step; see section 3.1). In the second step of the algorithm (Extraction of Read Only Transactions) we obtained two read only transactions of TPC-C (OrderStatus and StockLevel), one transaction for the login, and another transaction representing the merge of the OrderStatus and StockLevel. The login transaction is learned because the TPC-C

terminal emulation executes several commands after the login. The merged transaction appears because the last command of the OrderStatus (select order line table) is equal to the first command of the StockLevel (this is filtered in the next step). After the third step (Final-Learning) we obtained the results shown in Table 1, ordered by the number of times each transaction was identified in the audit trail.

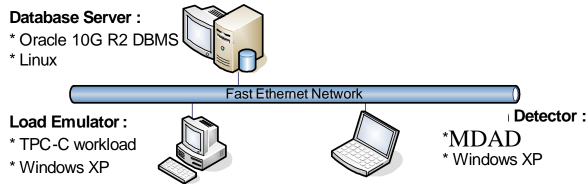

Figure 3. Experimental setup.

Because TPC-C specifies that the NewOrder may not complete due to a rollback an extra transaction is learned based on the incomplete NewOrder. We call the extra transaction as NewOrder with rollback. Additionally, the TPC-C Payment transaction also leads to two learned transaction profiles (PaymentByName and PaymentByID). This is because the Payment transaction has a condition right at the beginning resulting in a branch and, as we mentioned previously, each branch is learned as a separate transaction. However, these small differences in the learned profiles when compared to the real TPC-C transaction profiles have no impact at all in the detection algorithm.

Table 1. Learned transaction profiles for TPC-C.

Transaction #	Count	% total	TPC-C Transaction
6	43,255	44.784	NewOrder
5	24,950	25.832	PaymentByName
4	16,323	16.900	PaymentByID
7	3,884	4.021	Delivery
1	3,881	4.018	OrderStatus
2	3,809	3.944	StockLevel
8	433	0.448	NewOrder with rollback
3	50	0.052	Login
Total	**96,585**	**100.000**	

In the next step we evaluated the learning algorithm in a real database scenario. The main goal was to assess the learning transaction curve and estimate false positives caused by incomplete learning and leading to extra transactions that have to be added to the graph later on.

We started with the audit log of one working day of real utilization of the database of the SCE, having 8,750 commands from 609 sessions and accesses 17 tables. This log was applied to the First-Learning step resulting in 33 different transactions. In the Extraction of Read Only Transactions, two of them were learned and the Final-Learning step showed 31 different transactions.

Figure 4 shows the learning transaction curve. As we can see, most of the transactions (27 out of 31) were learned very quickly, during the first 1,000 commands (858 commands actually, as seen in Table 2). It is also quite evident that two new groups of database functionalities (and corresponding transactions) were executed around the command number 4,000 and command number 6,500, corresponding to the two steps in the learning curve. In a real situation in which the learning phase stopped after the initial 858 commands, these two moments would correspond to conditional detection. In this case the DBA would have to analyze the new transactions and add them to the graph. Table 2 shows details (commands executed so far, transactions, etc) at these two moments when conditional detection would appear. A total of 4 transactions would have to be validated manually by the DBA.

For this SCE application we can conclude that there are 27 transactions regularly executed during the day and 4 transactions that are executed after a certain hour in the day. This kind of behavior may appear during a wider window of time with different groups of transactions being executed only in one particular day of week or month, for instance. Thus, we decided to analyze the audit logs for an entire week. The audit log of one week of the SCE application had 65,340 commands from 4,187 sessions and accesses 22 tables. This log was applied to the First-Learning step

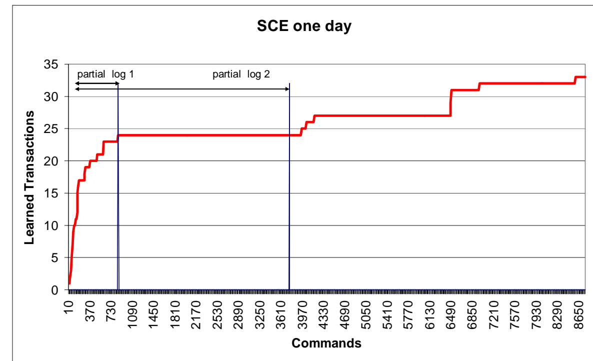

Figure 4. Evolution of the transactions during one day in the SCE application.

resulting in 56 different transactions learned out of 13,763. In the Extraction of Read Only Transactions step, 5 extra transactions were learned. The introduction of these read only transactions and the audit log in the Final-Learning step resulted in the learning of 57 different transactions, from a total of 16,097 executed transactions.

Table 2. Three different log situations compared.

	Complete Log	Partial Log1	Partial Log2
Commands	8,750	858	3,726
Sessions	609	107	381
Number Transactions	1,954	228	1,455
Tables	17	16	16
First-Learning step Transactions	33	24	24
Read Only Transactions	2	0	0
Final-Learning Transactions	**31**	**27**	**27**

Figure 5 shows the entire learning profile curve. As we can see in the chart new transactions were executed during the whole week, showing that this (real) application would required at least an entire week to allow complete transaction learning (although most of the transactions have been learned in the first two days).

In some cases the learning process may take a considerable time to learn all the transactions if the transactions are evenly spread in a large period of time. In practice, the conditional detection mode has to be kept active for enough time to assure a complete learning. It is worth noting that even in this mode, the proposed algo-

rithm does its job of adding concurrent malicious data access detection to audit trail. The only overhead the learning phase introduces to the system is the audit itself, because the learning may be executed in another computer.

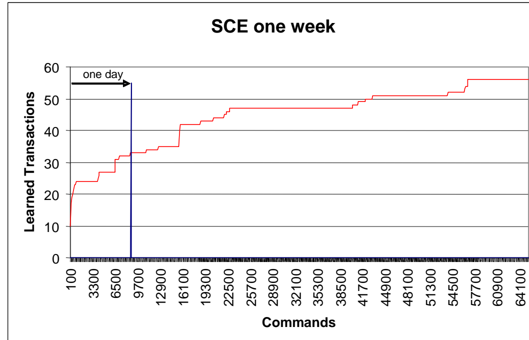

Figure 5. Evolution of the transactions during one week in the SCE application.

4.3 Evaluation of detection coverage and latency

We have also evaluated detection coverage and latency in two different experiments with the TPC-C setup: automatic injection of random transactions and human attempts to break the mechanism and access or damage the database without being detected.

The coverage represents the percentage of malicious transactions detected. A first evaluation of the coverage of the proposed mechanism was done by submitting random transactions while the system was executing the TPC-C transactions. A total of 653 random (extraneous) transactions have been submitted, corresponding to the execution of 2,558 SQL commands. MDAD mechanism has detected 648 of these injected transactions, resulting in a detection coverage of 99.23%, which is a quite good result. The small number of undetected transactions (five transactions) was caused by random transactions that mimic exactly the smaller transactions of TPC-C. In this experiment, all the TPC-C transactions have been correctly and completely learned, resulting in zero false positives.

The percentage of undetected transactions (0.77%) can be reduced by adding more information on the fixed structure of SQL commands, usually available in the audit trail. This would make much more difficult to mimic the correct SQL command.

The latency represents the time between the execution of a malicious command and its detection. Experiments showed that the latency varies between 1 second and 1.6 seconds. The lower bound of the latency (1 second) is because the MDAD checks the audit log every second. Obviously, if the frequency of checking the audit trail for malicious transactions were higher the average latency would decrease (but the performance impact would be higher). Note that the users take some time between executions of commands, which means that a latency of less than 2 seconds is extremely good.

The number of valid transactions executed between the moment a malicious transaction is submitted and the moment it is detected is also important. Typically, this number ranges between 20 and 70 transactions depending on the system load. Note, however, that the system is executing at a rate of thousands of transactions per minute, which makes this number insignificant.

The use of simple random generated transactions is acceptable for a very first evaluation of the coverage of the mechanism (and to provide a good evaluation of latency), but it is not enough to gain confidence in the mechanism. We decided to go for a test with real (human) users. The human users were volunteers (several students and a professional DBA) that have accepted the challenge of trying to beat our detection mechanism.

For these tests with humans an Oracle server was used within a LAN. The TPC-C database was installed and several triggers were created to record changes in the database. A web front-end was built to let the users enter SQL commands from any computer inside the LAN. This web front-end also had a background task to record the history of all the commands executed for latter analysis. A short document was distributed to the testers explaining the objectives, including also the database schema giving insider knowledge to the attackers.

The volunteers started 142 sessions and submitted 691 commands. All the sessions were detected as malicious, leading to 100% detection coverage. However, five sessions (3.5% of the total) were able to introduce changes in the data just before being detected as malicious in the next command executed. Note that, before being able to change the data, the users tried several times (from 8 to 36 times) and, in all those attempts, the sessions were detected as malicious and killed, giving the DBA enough warnings of something that deserved inspection.

Analyzing these five sessions we conclude that three correctly executed the initial commands of a correct transaction and then committed the changes to the database. This corresponds to a commit made at the middle of a transaction. The other two malicious sessions were able to made unauthorized changes in the database by sending the commands inside a PL/SQL anonymous block. However, they were almost immediately detected and the session was killed before they could execute another command. Because the detection is based on the audit trail, the detection of a suspicious write command (as was the case) can only be performed after its execution. In these two cases the user (the expert user) has sent two commands in a PL/SQL anonymous block, which correspond to the worst case concerning latency, as the two commands are executed almost at the same time. Although in theses cases the detection is done after the unauthorized change in the database, it would be possible to avoid damage by using damage confinement mechanisms [8].

4.4 Impact on database server performance

To measure the impact of detection on the server performance we used the TPC-C setup to emulate 10 online session terminals inputting transactions with variable throughput. Three configurations have been considered representing the server without the audit activated, with the audit activated (but no malicious data access detection) and with the detection mechanism (Figure 6).

As we can see, with 100% load the audit reduces in 25% the maximum number of transactions while the detection reduces additional 6%. This is the worse scenario possible, as the server was with 100% load. With 60% load the audit reduces only about 3% the maximum number of transactions while the detection reduces 3%. Below this load the influence of both the audit and the detection is quite small. The only overhead the learning phase

introduces to the system is the audit itself, because the learning may be executed in another computer.

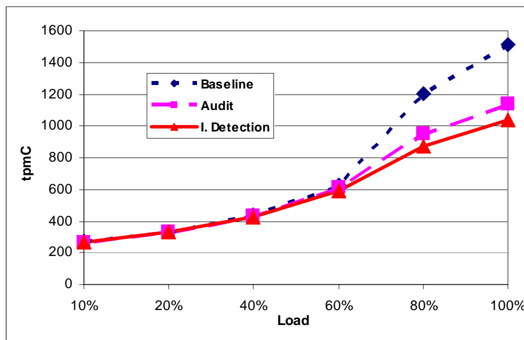

Figure 6. Performance for the three configurations considered.

5. CONCLUSION

This paper proposes new malicious data access detection mechanism for DBMS. This method adds concurrent analysis to auditing mechanisms presented in most of the commercial DBMS. It has one phase devoted to the learning of transaction profiles and another phase where the detection of malicious users is made. In the learning phase a graph is built having the sequence of commands that compose each valid transaction. In the detection phase the mechanism catches malicious users by detecting the transactions that fall outside the learned profile. Then the DBA is warned while the malicious session is killed. During the initial period of detection time the system may work in a warning only mode allowing the DBA to take the appropriate actions towards the suspected session. If the wrong transaction happens to be a good transaction not yet learned, the DBA may add it to the profile.

The paper presented an implementation of the proposed system using the Oracle 10g R2 DBMS and it has been evaluated using the standard benchmark for database systems (TPC-C) and a production database (SCE) used by a large hospital. The detection coverage observed for random transactions was above 99%. The small percentage of undetected transactions corresponds to very small TPC-C transactions whose commands were occasionally mimicked by the random transaction injector. However, in all cases the attack was immediately detected in the following command. In reality, the detection coverage was 100%, if we consider the sequence of commands inside the transaction.

Concerning the tests with human users, the attacks have been detected in all cases. In 5 sessions, the user managed to introduce changes in the database, but they were spotted as intruder in the subsequent command. The detection latency is consistently low, ranging from 1 to 1.6 seconds. The performance penalty in normal load conditions is 6% or less. In heavy load conditions, performance overhead raises up to 25%.

6. REFERENCES

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, “Hippocratic databases”, 28th international conference on Very Large Data Bases (VLDB), Morgan-Kaufmann, 2002.

[2] A. Anton, E.Bertino, N.Li, and T.Yu, “A roadmap for comprehensive online privacy policies”, In CERIAS Technical Report, 2004-47, 2004.

[3] Elisa Bertino, Ashish Kamra, Evimaria Terzi, Athena Vakali, “Intrusion detection in RBAC-administered databases”, 21st Annual Comp. Security App. Conference (ACSAC) 2005.

[4] Christina Yip Chung, Michael Gertz, Karl Levitt, “DEMIDS: A Misuse Detection System for Database Systems”, Third International IFIP TC-11 WG11.5 Working Conference on Integrity and Internal Control in Information Systems, Kluwer Academic Publishers, 1999, 159 – 178.

[5] Lawrence A. Gordon, Martin P. Loeb, William Lucyshyn and Robert Richardson, Computer Security Institute. Computer crime and security survey, 2006.

[6] Y. Hu and B. Panda, “Identification of malicious transactions in database systems”, International Database Engineering and Applications Symposium (IDEAS), 2003.

[7] Sin Yeung Lee, Wai Lup Low, Pei Yuen Wong, “Learning Fingerprints for a Database Intrusion Detection System”, 7th European Symposium on Research in Computer Security (ESORICS 2002).

[8] Peng Liu, “DAIS: A Real-time Data Attack Isolation System for Commercial Database Applications”, 17th Annual Computer Security Applications Conference, 2001.

[9] Maxion, Roy A. and Townsend, Tahlia N. "Masquerade Detection Using Truncated Command Lines." International Conference on Dependable Systems and Networks (DSN-02), Washington, D.C. 23-26 June 2002.

[10] Roy A. Maxion, “Masquerade Detection Using Enriched Command Lines”, Intl Conf on Dependable Systems & Networks (DSN-03), San Francisco, California, 2003.

[11] Andrew Conry-Murray, “The Threat From Within”,http://www.itarchitect.com/shared/article/showArticle.jhtml?articleId=166400792, 2005

[12] Oracle Corporation, “Oracle® Database Concepts 10g Release 1 (10.1)”, 2003.

[13] Pen Test Limited, “Oracle security white paper series exploiting and protecting oracle”, 2001.

[14] R. Ramakrishnan, J. Gehrke, “Database Management Systems” 3rd Ed., McGraw Hill, ISBN 0072465638, 2002.

[15] M. Schonlau, M. Theus, "Detecting Masquerades in Intrusion Detection Based on Unpopular Commands," Information Processing Letters, 76, 33-38, 2000.

[16] M. Schonlau, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theus, and Y. Vardi, “Computer intrusion: Detecting masquerades”, Statistical Science, 16(1):58–74, February 2001.

[17] Transaction Processing Performance Council, “TPC Benchmark C, Standard Specification, Version 5.4”, 2005, available at: http://www.tpc.org/tpcc/.

[18] Marco Vieira, Henrique Madeira, "Detection of malicious transactions in DBMS", The 11th IEEE Intl Symposium Pacific Rim Dependable Computing, PRDC2005, Changsha, Hunan, China, December-2005.

[19] Noel Yuhanna, “Comprehensive Database Security Requires Native DBMS Features and Third-Party Tools”, Market overview, Forrester Research Inc., May 2005

.