Detecting malicious SQL

José Fonseca[1], Marco Vieira[2], Henrique Madeira[2]

[1]ESTG-ISUC, University of Coimbra, Portugal
josefonseca@mail.telepac.pt
[2]CISUC, University of Coimbra, Portugal
{mvieira, henrique}@dei.uc.pt

Abstract. Web based applications often have vulnerabilities that can be exploited to launch SQL-based attacks. In fact, web application developers are normally concerned with the application functionalities and can easily neglect security aspects. The increasing number of web attacks reported every day corroborates that this attack-prone scenario represents a real danger and is not likely to change favorably in the future. However, the main problem resides in the fact that most of the SQL-based attacks cannot be detected by typical intrusion detection systems (IDS) at network or operating system level. In this paper we propose a database level IDS to concurrently detect malicious database operations. The proposed IDS is based on a comprehensive anomaly detection scheme that checks SQL commands to detect SQL injection and analyses transactions to detect more elaborate data-centric attacks, including insider attacks.

Keywords: Web applications, Security, Intrusion Detection

1 Introduction

Web applications are extremely popular today because they are ubiquitous and can be easily maintained and updated. Users access the interface via a web browser and send requests to the web server, which in turn translates these requests to database SQL commands and, using the results of those commands, generates the response that is sent back to the browser for final presentation to the user.

A major problem is that web applications are often insecure. In fact, web application developers are normally not specialized in security and the usual time to market constraints direct the effort on satisfying the user's requirements, causing security aspects to be easily neglected. Additionally, rapid application development (RAD) environments (e.g., VS.NET, Eclipse, PHP-Nuke, Drupal, osCommerce) frequently used to build web applications may generate code with vulnerabilities, even when the developer follows the best security practices.

SQL-based attacks, such as SQL injection, are an important class of attacks in web applications as can be confirmed by innumerous vulnerabilities daily reported in specialized sites (e.g., www.securityfocus.com) [1]. SQL-based attacks basically exploit unchecked input fields at user interface to change the SQL commands that are sent to

the database. Although some flaws could be mitigated by means of simple operations (e.g., using bind variables, using correctly implemented stored procedures, granting the minimum privileges needed for every action, restricting the input character set, using escaping quotes, etc.), theses aspects are frequently disregarded. The altered commands may give the attacker access to unauthorized data (read, change or delete), access to privileged database accounts or even permission to execute server side commands (e.g., database stored procedures).

Typical intrusion detection systems (IDS) at network or operating system level cannot detect SQL-based attacks. Although they can be applied to prevent the use of some common malicious strings like "union", "or 1=1", they are quite restrictive, not exhaustive and can be evaded easily. Even traditional database security mechanisms cannot detect these attacks, as they are perceived as authorized commands executed by authorized users. End to end encryption is also useless to stop these attacks because commands are executed by users who have been granted with the appropriate application access privileges.

In this paper we defend that the best way to detect SQL-based attacks that exploit web application code vulnerabilities is to place an additional intrusion detection layer at the database level. At this level, malicious SQL can be detected independently from the web application that has been exploited to launch the attack. In addition, insider attacks launched by malicious users can also be detected.

In spite of all the classical database security mechanisms, current Database management Systems (DBMS) are not well prepared for assuring high privacy and confidentiality [2], especially in what concerns to intrusion detection features [3]. In fact, very few IDS for databases have been proposed so far [4, 5, 6, 7, 8, 9] and, to the best of our knowledge, there is no DBMS that offers intrusion detection as a security feature. It is worth noting that the only mechanism available today to detect malicious database actions is the analysis of database audit trails. However, as this analysis is done offline, audit trails can only be used for diagnosis purposes after attacks.

Recent works have addressed concurrent intrusion detection and attack isolation in DBMS. Valeur et al [4] presented an IDS for SQL injection attacks using several detection models for the different types of attacks. In [5] the authors use the audit logs to derive user profiles that describe typical behavior of users in the DBMS, using the notion of distance measure and most frequent item sets. In [6] a real-time intrusion detection mechanism based on the profile of user roles and three levels of precision in the definition of the data is proposed. In Vieira et al. [7] and Chung et al. [5] the detection of malicious DBMS transactions was addressed with the assumption that the transaction profiles was known in advance and provided manually to IDS.

In this paper we propose an IDS composed of a comprehensive anomaly detection scheme based on automatic learning of SQL commands and transaction profiles. The proposed IDS uses intrinsic characteristics of database applications that allow the definition of an abstraction of the utilization of the database using two levels of detail: 1) *SQL commands* to detect SQL injection attacks and 2) *database transactions* to detect more elaborate data-centric attacks, including insider attacks. These two levels actually represent a fingerprint of every web database application.

The structure of the paper is as follows. Section 2 presents the proposed IDS mechanism. Section 3 presents a two level definition of profiles. Section 4 presents

the evaluation of the proposed mechanism using the TPC-W standard benchmark and real database applications. Section 5 concludes the paper and introduces future work.

2 Intrusion detection at database level

Web applications normally rely on a back-end database where the information is processed and stored. Typically the database is located inside a LAN and benefits from the enterprise network security systems. Although security mechanisms at network and operating system level are essential, many web applications have vulnerabilities that allow SQL-based attacks that cannot be detected by IDS at operating system (OS) and network levels. Additionally, database attacks may also come from inside the organization where the attacker has physical access to terminals or even to the database server machine. In this case the network security mechanisms are overridden and useless because the user is already inside the network containment barrier. Thus, we believe that it is important to provide additional intrusion detection capabilities at the DBMS level aimed to cover specifically SQL-based attacks.

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. Anomaly detection is the search for deviations from an historical profile of good behavior. To use the pattern recognition approach we need the signatures of known attacks. The problem is that new attacks related to web-based database applications are discovered every day (and it is trivial to change an attack slightly) and the creation of new signatures in a daily basis requires a substantial investment. On the other hand, anomaly detection is able to detect both known and unknown attacks whenever there is a deviation from the expected behavior profiles.

2.1 IDS architecture

The IDS proposed in this paper includes comprehensive anomaly detection at SQL command and at database transaction level and comprises two phases: a learning phase, where SQL commands and transaction profiles are extracted and a detection phase, where learned profiles are used to concurrently detect SQL-based attacks.

The architecture of the proposed IDS is shown in Fig. 1. The Database Interface intercepts the data flow between the application and the database server, and is used for both the learning and the detection phases. During the *learning phase* the Command Capturing component logs the SQL commands executed by each user. Commands are parsed (by the Parsing component) in order to remove the data variant part (if any) of SQL commands and a hash code is generated to uniquely identify each command. The Learning component examines the SQL command sequence, learns the execution flow (including branches and loops), and generates a list of hash codes of the commands executed and a directed graph representing database transactions profiles for each database user. Different database users will have their own collection of profiles. Although the number of the application users may be quite large, the number of database users is usually restricted corresponding to the several types of

users of the application. During the *detection phase* the commands and transaction profiles previously learned are used to detect intrusions. When a potential intrusion is detected the Action component performs an automatic predefined function (e.g., killing the attacker session, warning the database administrator).

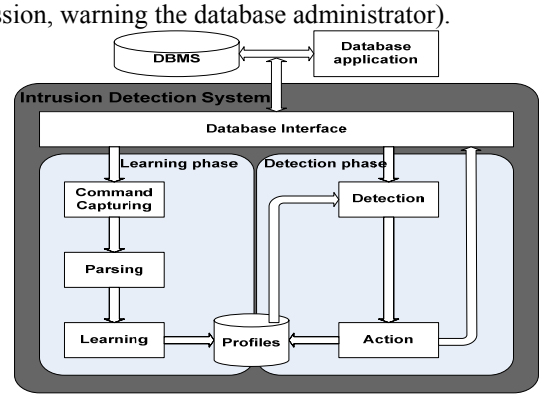

Fig. 1. IDS building blocks and workflow.

2.2 Database profile learning

The SQL commands and transactions learning curve depends on the utilization profile of the database application. Many 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 Database Administrator (DBA) is not confident with the learned profile the Detection component should not act drastically on the session (e.g., should not kill sessions that are considered as intrusion). Instead the DBA should analyze those situations first and, possibly, add the detected transaction to the learned profile. In a real database application, the DBA knows exactly when there is an upgrade and when new functionalities are added to the application. When this happens it is common to have new transactions and, after a short period they would be learned by the IDS mechanism. The set of transactions remains stable, as long as the database application is not changed. There are two ways to obtain the new profiles automatically: concurrently during normal utilization of the applications and by running application tests. In addition to profile learning, some other alternatives could be considered, such as manual gathering and static analysis. Manual gathering of profiles assumes that database transactions are well documented and, usually, this is not the case. Static analysis of the source code could also be used [10, 11]. However this is a complex task and fails when dynamic SQL is used, which is usually the case.

3 Database utilization profiles

In a typical web application the code includes the sequence of SQL commands organized as database transactions. When a user connects to the DBMS and establishes a session, the user starts the first transaction. That is, the user cannot escape to the

transaction mechanism, as all the commands executed always belong to a transaction. When one transaction ends a new transaction begins. Although the SQL commands can be generated dynamically by the application, users cannot execute pure ad hoc SQL commands. The set of transactions and corresponding SQL commands hard-wired in a web application code represent a well defined set, which allow an exhaustive learning of all commands and transactions. For example, in a banking web application users have only access to the functionalities available at the interface (e.g., withdraw money, balance check account, etc) and no other operation is allowed.

The proposed IDS is based on a set of security constraints defined at two abstraction levels: *command level* and *transaction level*. Intrusion detection activity starts at the lowest level (command level). If no intrusion is detected at this level, the detection continues at the next level (transaction level). If no restriction is violated after having passed both levels, the command is considered valid.

3.1. Command level abstraction

SQL commands represent the basic data needed to generate the information required at the two abstraction levels. SQL commands also represent the entry data used to feed the IDS in both the learning and detection phases.

The basic information on each command required for intrusion detection is the following: 1) name of the user who executes the command; 2) identification of the session established when the client application connects to the database server; 3) full text of the SQL command executed; 4) time stamp of the execution of the command.

An important aspect is that the information stored by the IDS does not represent the exact command text, since commands may differ slightly in different executions, while keeping the same structure. For example, in the command "*SELECT * from EMP where job like 'CLERK' and SAL >1000*", the job and the salary in the select criteria (job like ? and sal > ?) depend on the user's choices. This way, instead of considering the full command text, we just represent the invariant part of it. After removing the variant part of each command it is possible to calculate the command signature using a hash algorithm. These signatures are used at both abstraction levels to represent the command in a compact form.

To perform an SQL injection the attacker alters the structure of the SQL command in order to exploit an unchecked input in an application page. Usually as a first step the attacker adds a condition in the where clause of the SQL command to gain privileged access. Then the attacker executes SQL commands returning valuable information (usually using a union clause with the malicious select), changing the database (performing inserts, deletes or updates) or even performing OS commands. Command level abstraction can be used for detection in both the first and the second steps of the SQL injection attack as both steps require a change in the structure of the query.

The command level abstraction is not sensitive to attacks that do not alter the structure of the SQL commands. In order to execute malicious actions without being detected the attacker has to execute the authorized commands by changing the criteria values in a way that makes the altered command useful for his purposes. In [4] the authors parse the SQL commands and one of the models used is the string model

where the strings present in the SQL commands are analyzed. However this approach has a limited detection capacity and inevitably it increases the false positives rate because of the difficulties in modeling most of the string variations. To overcome this problem we propose another level of abstraction: the transaction level.

3.2. Transaction level abstraction

At transaction level, our intrusion detection mechanism uses the profile of the transactions implemented by database applications (authorized transactions) to identify user attempts to execute unauthorized transactions. The profile of a database transaction is represented as 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 commands that terminate the transaction. 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).

This command level IDS can be used to detect, among others, attacks from inside the organization. In this kind of attacks the user knows very well and already has access to the database application. The attacker may use his own account or he can impersonate another user and may use a SQL terminal to access the database instead of using the application. The attacker could mimicry a SQL command because of the privileged access to information. However it would still be difficult to mimicry the transactions in order to override the transaction level of the IDS.

To bypass this level a malicious user has to execute SQL commands in the correct order inside the transaction. To execute malicious actions without being detected he must choose and execute adequate dummy commands (commands that have no particular interest for the attacker, except to dodge the IDS) in the correct order and change the criteria in one of them in a way that makes the command useful for him.

4 Database IDS evaluation and experiments

We consider the following typical IDS evaluation metrics: 1) *false positives rate*: number of valid commands that are seen as malicious by the IDS over the total number of commands; 2) *coverage*: represents the percentage of malicious commands detected of all the malicious commands; 3) *impact on server performance*: represents the decrease in database performance due to the presence of the IDS; 4) *latency*: time between the execution of a malicious command and its detection.

Key points in assessing these metrics is how the attacker is modeled, which weakness of the system will be used, what commands will be executed and in what order. Another important issue to be addressed is how we can test unknown attacks. In the evaluation experiments we consider that the attacker knows exactly how the IDS works. Before starting the attack the adversary spends some time analyzing the sys-

tem looking for the weakest point and the right moment. Relying on the ignorance of the attacker seems to be unrealistic. If the database under sight is widely deployed it may be possible that the attacker knows their commands and transactions.

4.1 Experimental setup

In the present work the IDS was built as a SQL command sniffer that can be used independently of the target DBMS. However, the proposed IDS could also have been included inside the DBMS. In this case, the IDS can use standard DBMS functionalities such as SQL parser, transaction control, and data dictionary access, which would simplify its implementation.

As we want to test our mechanism with real database applications and independently of the target DBMS we have to setup the IDS using the least intrusive manner. The sniffer approach is the best option because the IDS can be placed in the local network near the database server or it can be placed inside the database server machine. One clear limitation of the sniffer approach is the need of clear network packets (or having access to the decryption function). Because we are focusing our work on the database IDS itself and not on the topology and related questions about its setup we are not going to discuss some well-known technical issues about network IDS, like packet splitting and Host-Based IDS vs Network-Based IDS [12].

The experimental setup consists of a Database Server, a Client Computer and an IDS Computer (where the IDS acting as a sniffer is installed) connected through a fast-Ethernet network. We used the following database application scenarios, running Oracle: 1) a well-known database performance benchmark, the *TPC-W* [13], which simulates the activities of an e-commerce business oriented transactional web server; 2) an academic and financial management application of the University of Coimbra, the *Pk_2005*; 3) a real (and large) hospital database application, the *SCE* (Central Service of Sterilization) currently in use in Coimbra University Hospitals.

4.2 Results discussion

To evaluate both the learning and detection phases of the IDS and its response to two different kinds of synthetic attacks (exploring command and transaction levels) we used the TPC-W. All the experiments using the TPC-W are based on a training data obtained by a 180 minutes learning phase where 51126 commands were executed. The last transaction profile, as well as the last SQL command, were learned 140 minutes after the beginning of the experiment, which corresponded to the execution of 40419 commands. To test the completeness of the profiles learned the detection phase of the IDS was used with an eight-hour execution of TPC-W, corresponding to the execution of 137233 SQL commands. During this test all the commands and transactions were considered valid, hence no false positives were observed. This means that the learning phase was exhaustive. The TPC-W profiles could be completely covered by the learning algorithm in a couple of hours because of the specific nature of a benchmark. The results should be similar in a real application when application tests are used to exercise the application during the learning phase.

Next we evaluated the IDS against a battery of malicious commands and transactions. A well informed attacker (for example an insider) will not just execute a random collection of SQL commands easily detected by the IDS presented in this paper. Instead, the attacker will try to be stealthy by executing commands similar to those of the application. Thus, the commands that are used to simulate SQL-centric attacks should be based on variations of the SQL commands that are actually generated by the application in order to simulate plausible (and hard to detect) attacks. Random tests are also used for the sake of completeness. To exercise the IDS more thoroughly, both in the command level and in the transaction level, we developed an application to automatically create and inject the attacks.

We executed 14 types of attacks for the command level IDS (Table 1). For each test an input file was created containing 100 SQL commands that were executed in the TPC-W database while the IDS was using the command level abstraction. The IDS detected every command as malicious except for the "Alter the text inside the strings and the values in the where clause" test. As we already expected this test would fail, because we developed the IDS in such way that ignores what is inside the strings and values, so the SQL commands that are exactly as expected, but with different information on the variable parts are not detected as malicious. Note that processing the variable parts is an error prone approach because it is extremely difficult to guarantee that learning algorithm will cover all the possible range of values. The "Place another SQL command at the end of the current command" test could not be executed because the TPC-W implementation used was built in Oracle and it does not allow these kinds of commands, unlike other database engines (SQL server, Mysql).

Table 1. Command level attack tests.

Command Test	# attack commands	# false positives
Random queries	100	0
Delete fields from select statements	100	0
Scramble the order of the fields in the select statement	100	0
Insert fields (may be functions) in select statements	100	0
Delete tables from select statements	100	0
Scramble the order of the tables in the select statement	100	0
Insert tables in select statements.	100	0
Delete conditions from the where clause	100	0
Scramble the order of the conditions from the where clause	100	0
Insert conditions from the where clause	100	0
Create an SQL anonymous block	100	0
Create a compound SQL query using UNION, UNION ALL, INTERSECT and MINUS	100	0
Place another SQL command at the end	-	-
Alter the text inside the strings and the values in the where clause	100	100

To exercise the transaction level IDS we have executed 6 tests (Table 2). All the malicious transactions where spotted as soon as the erroneous command was executed.

The learning phase is a critical step that was tested with two real applications (the Pk_2005 and the SCE) during their normal use. The Pk_2005 executed 731438 SQL commands during one week (left side graphic in Fig. 2). The last transaction was learned after the 731373 command and the last different command was learned after

the 731327 command. As shown in Fig. 2, there were some bursts of learning during this week test, which is related to some new procedures executed in those occasions.

Table 2. Transaction level attack tests.

Transaction Test	# attack transactions	# false positives
Random transactions	100	0
Delete SQL commands from the transaction	100	0
Scramble the order of the SQL commands in the transaction	100	0
Insert SQL commands in the transaction	100	0
Commit the transaction before its end	100	0
Rollback the transaction before its end	100	0

The SCE executed 753699 SQL commands during 64 days (right side graphic in Fig. 2). The last transaction was learned after the 728424 command and the last different command was learned after the 718265 command. Like the Pk_2005, the SCE shows bursts of learning, confirming the conclusion of some procedures being executed only in certain times of the day, week, month, etc. The learning phase is considered complete only when the number of new profiles and commands stabilizes.

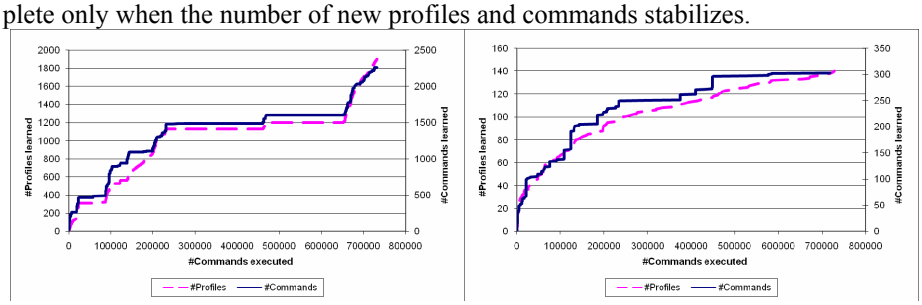

Fig. 2. Pk_2005 (on the left) and SCE (on the right) learning curves.

From the analysis of the results in Fig. 2 we can see that the learning period for the command level and for the transaction level are similar, showing that different transactions are usually made of different commands. We can also conclude that an intrusion detection mechanism based on learning the profiles while the application is in production may take a long time. If the application could be exercised by automatic test procedures or with users executing the applications functions specifically for the IDS the learning period would be drastically reduced.

Because we used the IDS as a network sniffer it introduces no load in the database server and we experienced no packet drop during the experiments. For the sake of completeness we also measured the load impact on server performance for the case where the IDS is located in the DBMS machine. This was done with the TPC-W database and, in the worst scenario, the IDS caused a degradation of almost 11% in the number of transactions executed per minute. By reducing the load to 50%, the impact in the performance decreases to only 5%, and below 40% load it is less than 0.1%. The analysis of these results must take into account that the IDS application tested has not been thoroughly revised for speed as a commercial application should be.

The latency observed is less than 2ms. In most of our tests the malicious com-

mands were detected even before the DBMS could send the responses back to the client. This is a very important reference value because it indicates that a malicious action can be stopped right in the first malicious command, thus preventing the spread of attack consequences. Implementing the IDS inside the DBMS core allows the detection to be made before the SQL command ever reaches the database server. In this case there is a tradeoff between the detection latency and the server response time.

5 Conclusion

This paper presents an intrusion detection system targeted for web-based database applications. It uses the anomaly detection approach and a two level definition of profiles (SQL commands and transactions) to represent the normal utilization.

Our implementation of the IDS was evaluated using a standard benchmark for database systems and two production databases. The detection coverage observed for the nineteen types of attacks tested was 100%, except for one of them. There is no relevant performance penalty using the IDS as a network sniffer.

The experiments show that the learning times can be significant if only normal usage of the application is considered for profile identification. The automatic or manual execution of existing tests or application functions may be used to shorten this period. Both SQL command learning and transaction learning require similar periods of training, but transaction level detection can be used with a wider range of attacks.

References

1. Acunetix, available at: http://www.acunetix.com
2. Anton, A., Bertino, E., Li, N., Yu, T.: A roadmap for comprehensive online privacy policies. CERIAS Technical Report, 2004.
3. Agrawal, R., Kiernan J., Srikant R., Xu, Y.: Hippocratic databases. in proc. VLDB, 2002
4. Valeur, F., Mutz, D., Vigna, G.: A Learning-Based Approach to the Detection of SQL Attacks. DIMVA 2005
5. Chung, C., Gertz, Levitt: DEMIDS: A Misuse Detection System for Database Systems. in proc. of Third International IFIP TC-11 WG11.5 Working Conference on Integrity and Internal Control in Information Systems, Kluwer Academic Publishers, 1999
6. Bertino, E., et al.: Intrusion detection in RBAC-administered databases. ACSAC 2005
7. Vieira, M., Madeira, H.: Detection of malicious transactions in DBMS. PRDC2005
8. Lee, S.Y., Low, W.L., Wong, P.Y.: Learning Fingerprints For A Database Intrusion Detection System. ESORICS 2002
9. Low, W.L., Lee, J., Teoh, P.: DIDAFIT: Detecting Intrusions in Databases Through Fingerprinting Transactions. International Conference on Enterprise Information Systems, 2002
10. Viega, J., Bloch, J. T., Kohno, Y., McGraw, G.: ITS4: A static vulnerability scanner for C and C++ code. Computer Security Applications Conference, 2000.
11. Bergeron, et al.: Static Detection of Malicious Code in Executable Programs, SREIS, 2001
12. Internet Security Systems: Network- vs. Host-based Intrusion Detection, 1998
14. TPC Benchmark W, 2002, available at: http: //www.tpc.org/tpcw