DIWeDa - Detecting Intrusions in Web Databases

Alex Roichman
Department of Computer Science,
The Open University, Raanana, Israel
Alexr@Checkmarx.com

Ehud Gudes
Department of Computer Science,
The Open University, Raanana, Israel,
and Department of Computer Science,
Ben-Gurion University, Beer-Sheva,
Israel
Ehud@cs.bgu.ac.il

Abstract. There are many Intrusion Detection Systems (IDS) for networks and operating systems and there are few for Databases- despite the fact that the most valuable resources of every organization are in its databases. The number of database attacks has grown, especially since most databases are accessible from the web and satisfactory solutions to these kinds of attacks are still lacking.

We present DIWeDa - a practical solution for detecting intrusions to web databases. Contrary to any existing database intrusion detection method, our method works at the session level and not at the SQL statement or transaction level. We use a novel SQL Session Content Anomaly intrusion classifier and this enables us to detect not only most known attacks such as SQL Injections, but also more complex kinds of attacks such as Business Logic Violations. Our experiments implemented the proposed intrusion detection system prototype and showed its feasibility and effectiveness.

Keywords: Intrusion detection, web database security, database vulnerability, SQL content classification.

1. Introduction

Web applications have become very popular in recent years, but their primary focus on functionality and not on security. As a result, there are many security holes in web applications and according to [17], "70% of websites are at immediate risk of being hacked!" Web applications are accessible 24 hours a day, 7 days a week, and have direct access to back-end databases. The attack surface of such databases is very large and the existing technology cannot prevent many attacks.

The best known type of attack is the SQL injection attack and several attempts to deal with it were published (e.g., [3, 5]). However, the above methods cannot defend against another kind of web application attack which is the *Business Logic Violation* attack. For example, in many web forums there may exist a business rule that states that a user must be registered prior to participating in a forum. This logic can be violated at the application level by an intruder who participates in a forum without registering. Such attacks compromise the business logic and can be seen only at the session level. Databases cannot prevent them because the existing database access control can grant or revoke access to resources only according to the user identity or role. It cannot rely on the business logic of an enterprise. Thus the database's access control is inadequate and many web attacks remain unprevented.

The described situation has some serious impacts. Currently, the only means to prevent attacks on web databases is at the application level. Although many advances have been made in developing secure applications, trusting applications which are developed under time constraints, and by developers who are not security experts, presents a large risk to the database and therefore databases are threatened by these applications. Intrusion detection is therefore an important security measure in these applications.

An enterprise might have several applications, but one database. These applications are changed frequently, thus re-learning the application behavior by IDS requires much effort. On the other hand, business rules which are seen at the database level are stable. Thus, it is preferable to have only one IDS at the database level and to enforce stable business rules, and not to have an IDS for each application which would demand coping with continuous application changes.

Intrusion Detection Systems for operating systems and networks have existed for over 20 years, but IDS for databases is a relatively new field of research that has surfaced in the last few years. Very few practical solutions exist for database IDS (AppRadar, SQLGuard, see [14]) and most of them are signature-based, depend on a specific database provider, and cannot detect many anomalous SQL sessions, especially from web applications. Detection of business logic violations is beyond the scope of these IDS. Thus, despite the existence of academic and industrial research for database IDS, there is no suitable practical solution for web database intrusion detection and many attacks remain unnoticed and unresolved. The absence of an appropriate solution for web databases can be explained by the fact that there are several problems that a web database intrusion detection system must solve:

- In a typical n-tire (web) application, different users can run their SQL statements on the same database connection. This technique is called *Connection Pooling* [18] and contributes to application efficiency. But with this technique, IDS cannot distinguish between legal and intruder sessions. Without finding a way of identifying and partitioning web database sessions, connection pooling makes the web application's access to databases almost untraceable. Since the real user of a web session is unknown at the database level, it is also impossible to apply role base access control to web databases. Sometimes the actual role a user uses is determined dynamically only at run time.
- Web applications have a tendency to use the *Implicit Transaction* where each transaction consists of a single SQL statement. This makes transactional level detection not suitable for web applications. But there exist attacks, such as business logic violations, that cannot be seen at the statement level; they can be seen only at the session level (composed of multiple transactions).
- Many web database attacks are very specific to the enterprise business logic, thus the IDS cannot be signature-based and must be tailored to the enterprise by learning its profiles in a given enterprise. But different roles in an enterprise may have different authorization – what is legal for a one role may be intrusion for another. Thus the best strategy for web database IDS is to build profiles not per an enterprise, but per enterprise roles.
- Building profiles requires a long training period that must be free of attacks. For real web applications it is generally impossible to guarantee a free-of-attack period.

We will present our method for web database intrusion detection that will give a practical solution to the above problems and improve database systems security. Our method works with any existing database and is capable of associating each SQL statement reaching a database with its actual user. We identify database roles from the learnt profiles and look for intrusions from one role to another. We detect intrusions at the session level, thus we are able to detect attacks such as the business logic violation, which cannot be seen at the statement/transaction level. We classify each session by a classifier called the *SQL Content Anomaly* classifier. This approach enables us to detect enterprise roles and analyze an entire session by looking for a deviation from previously learnt roles. Furthermore, our model is able to learn profiles by observing the normal working application with no assumption that the learning period is clear from attacks.

The rest of this article is organized as follows: Section 2 presents related work. Section 3 presents our method and in Section 4 we analyze and evaluate it. The last section concludes the article and discusses future work.

2. Related Work

IDS for databases, and especially web databases, is a relatively new field of research. One such research idea is to learn the structure of each SQL statement possible in the system and to fingerprint that structure. There are a large number of such possible statements, but most of them differ only in constants that represent the user's inputs. If we replace the constants in each statement with variables, we get some high level representation of the SQL sentence called the fingerprint (for more detail, see Section 3.3.1 Fingerprint Set Builder). [3] suggest detecting SQL injections by comparing a fingerprint before inclusion of user input with that resulting after inclusion of input. [5] develop this approach by combining static code analysis and runtime monitoring of possible fingerprints. [7] suggest also imposing order on possible fingerprints. But the disadvantage of these techniques is in its inability to correlate each fingerprint with an appropriate application role

An additional approach is to refer to some interesting properties of each SQL statement such as referenced tables and fields. [2] assign each SQL sentence to some role defined by the SQL's properties. If a new SQL statement arrives, the IDS classifies it to one of the existing roles and compares the predicted role with the role of the user who submitted the SQL query. When the predicted role is different from the user role, the alarm is raised. However, this method is not suitable to the web applications where we do not know the user's role in advance.

Another approach to database IDS is to build profiles for each database user. Users of a database do not usually access all the data, but only a small part of it. [4] identify a working scope of each user and measures the distance of each user's session from the built profiles. When this distance is greater than some predefined threshold, the IDS raises an alert. But for many web applications the number of users is tremendous and it is very difficult to maintain such a great quantity of profiles. Another problem with this approach is erroneously creating a legal working scope for the attacker who accesses the data of different users.

Another approach to the database intrusion detection problem is to search for data dependencies among the data items in a database [see 6, 12]. The data dependencies are the access correlation between the items that are the tables' fields. Transactions not compliant to those dependencies are marked as intrusions. But this method ignores the structure of an SQL sentence and thus may suffer from a high false negative rate.

The proposed methods are capable of detecting several data-centric attacks, but have some weaknesses. When only looking for fingerprints, without associating the fingerprints with roles, many attacks will go undetected. For example, the fingerprint of an SQL run by a professor cannot be used for classification of a student query. Thus it is desired that the IDS will be role-based. But as we already mentioned, for existing web applications roles cannot be known in advance, but must be learned by the IDS. Moreover, when the IDS works only at the SQL statement level, many attacks that can be seen only at the session level remain undetected. Our approach will use new ideas, enabling us to detect previously undetected attacks such as business logic violations. Our model will learn the database access roles (where business rules are wired). With this information, we can look for anomalous sessions which deviate from these roles. In the next section, we will present the architecture of our system and describe how it works.

3. Our Approach

3.1 The Architecture

The software architecture for the proposed IDS design is shown in Figure 1:

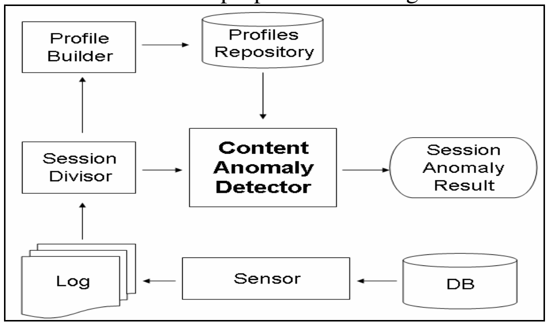

Figure 1: System architecture

The purpose of the Sensor is to catch every SQL statement that arrives at the database and to write it to the Log. This log is then divided by the Session Divisor to be used by the Profile Builder during the IDS learning phase, and by the Detection Engine during the detection phase. The profile builder generates sets of application profiles that are stored in the Profiles Repository. The Detection Engine applies the Content Anomaly Detection algorithm to this repository, and outputs the Session Anomaly Result. Next we discuss each of the above components in more detail.

3.2 Session Divisor

All SQL statements submitted by a user from the moment she opens the web application until the application is closed, belong to a user's application session. But because of the connection pooling techniques that are used in web applications, SQL statements of different users from different sessions are mixed. As a result, we cannot distinguish between statements from different sessions without partitioning the SQL log. The first task is therefore, the partitioning of the log by sessions.

Our partitioning algorithm is based on the use of *Parameterized Views* as we proposed in [10]. As has been shown there, parameterized views are used as the means of access control to web databases and each such view retrieves information relevant to the current parameter. This parameter is unique for each session, and is very difficult to fake. For example, in a university system a student can retrieve her marks by selecting a course and submitting the following statement:

```
SELECT * FROM Student_Marks_View(0xA287B5)
WHERE Course_No = 12345
```

Figure 2: Parameterized view example

The parameter of the view is a random number which the web application uniquely associates with the user and the database has access to it as is depicted in Figure 3.

```
CREATE VIEW Sudent_Marks_View WITH pAS_key
SELECT * FROM Student_Marks_Table WHERE Student_No IN
(SELECT Student_No FROM Users_Table
WHERE Users_Table.AS_key=:pAS_key)
```

Figure 3: Parameterized view definition

Although the course number `12345` is a user's input and thus the SQL is vulnerable to SQL injection, `Student_Marks_View` returns only the data of the current student with the parameter of `0xA287B5` and thus SQL injection can affect only the student's data. Namely, the student may access information about her marks for different courses, but not marks of different students (for this she must guess a random parameter belonging to another student- an improbable task). Using the parameterized view technique, we can partition the log by parsing each SQL statement and retrieving its parameter, thus all sentences of the same session will have the same parameter. Furthermore, the actual user of each session is easily identified.

Even without the use of parameterized views, the recent tendency in the development of n-tiered web applications is to transfer the real user identity not only up to the basic application layer, but through all the various layers. Oracle's "lightweight session" [15] allows multiple-user sessions to be maintained within a single database session, so that each user can be authenticated by a database password, without the overhead of a separate database connection. IBM suggested using "trusted context" to connect to DB2 [13]. The last mechanism defines how a trusted application can connect to DB2, and while on the same connection, manage transactions of multiple users simultaneously. Using these techniques, a database continually tracks application users/sessions, and provides the tools ready for partitioning the SQL log.

3.3 Profile Builder

Each profile consists of an SQL Fingerprint Set and a Cluster Set that represents the SQL content of each access role. In the next two sections we will show how DIWeDa builds its profiles. It is important to mention that contrary to previously proposed IDS, our system can learn not only from an attack-free log, but also from any log using a single assumption about the *Session Intrusion Rate*, as is explained in the next section.

3.3.1 Fingerprint Set Builder

The SQL fingerprint is the SQL structure abstraction. Each SQL statement may consist of three types of tokens: reserved words (SELECT, WHERE, AND…), names of database objects (tables, rows, stored procedures…), and constants which only contain user inputs. The SQL fingerprint is generated from each SQL sentence by parsing the SQL and replacing the constants with special place holders. Some attacks, such as SQL injection, work by changing the structure of the SQL. So if we generate all possible fingerprints for a web application, DIWeDa will be able to detect SQL injections as they will not fit into any generated fingerprint [3].

We can create a fingerprint for each SQL sentence submitted by a web application. In this way, we may also fingerprint illegal SQL statements submitted by intruders. To cope with this problem, we define the *Session Intrusion Rate*:

> ***Definition 1****: The Session Intrusion Rate (SIR) is the ratio between the number of attacked sessions and all sessions.*

If we assume that from all user sessions only a few may be under attack, we can define an SIR of 0.01 or less. Notice that for a real web application most of its users are legal ones and not attackers. Our assumption is that the analyzed application is not under DDoS attack during its training phase. But detecting DDoS can be done by other existing DDoS detection tools, see [8].

> ***Definition 2****: The support of an SQL fingerprint is the ratio between the number of application sessions that submit this fingerprint and the number of all sessions in the training set.*

Using the above definition, fingerprints with support that is less than the Session Intrusion Rate can be ignored. In this way, false negatives are avoided when the IDS creates a fingerprint to an SQL sentence under injection attack, and thus erroneously classify an illegal event as a legal one at the detection phase. The Builder uses the log to learn all possible fingerprints with support not less than SIR. The result will be a fingerprint set of size n.

3.3.2 Cluster Set Builder

Each enterprise role accesses different parts of the information in the database and thus the SQL contents of different roles are far apart, while the SQL contents of users from the same role are very close. For example, in the University system both a student and a professor can login to a web application. A student can enroll in a course and a professor can give grades to her students. It is obvious that the SQL content of a student's application session is different from the SQL content of a professor: there are many SQL statements a student can submit that a professor cannot and vise versa. If we are able to differentiate between student and professor session contents, we can detect intrusions from one role to another.

Another example is the Bookstore web application with two different access roles: Searcher and Buyer [16]. One business rule of the bookstore may state that in order to buy a book each Buyer must submit her payment details and receive an invoice. This business logic is implemented in the application, where each Buyer must choose at least one book (select from Books table), submit her credit card details (insert into Credit Card table), order the book (insert into Orders table), and get an invoice (insert into Invoice table). These statements are common only to a Buyer session and do not exist in a Searcher session: when a Searcher suddenly insert into the order table without submitting other statements, her session SQL content is far from the Searcher role (because Searcher never accesses Order table) and from the Buyer role (because Buyer always accesses both Order, Credit Card and Invoice tables). Thus her session violates the business logic and should be classified as an intrusion.

Assume that an application has n different fingerprints. We can associate each application session with its SQL *Session Vector*, which is an abstraction of the SQL content:

> ***<u>Definition 3</u>**: a Session Vector is a binary vector SV with the length equal to the number of fingerprints in the application, where the ith bit is 1 if the application session submits SQL with the ith fingerprint, else bit i is 0.*

The Session Vector enables us to formally define the session's SQL contents. We can think about the web application's SQL content as an n-dimensional space, where n is the number of fingerprints for the application. Then each session's SQL content can be seen as a vector in the n-dimensional space. If a session's SQL content was absolutely random then the distribution of vectors in the space should be uniform. But in reality, a session's SQL content is not random and has a very regular structure and the distribution of vectors is not uniform. They are consolidated into several groups (see Figures 4 and 5) where each such group corresponds to an application role.

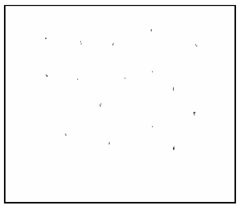
Figure 4: Abstraction of distribution in the SQL space for an unrealistic application

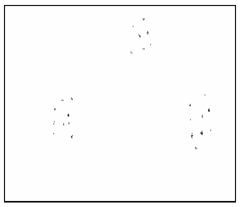
Figure 5: Abstraction of distribution in the SQL space for a real application

The distribution is not uniform since two sessions of the same role are likely to produce similar sets of SQL fingerprints. Thus it is reasonable to check the closeness of two sessions by the closeness of their session vectors. If two session vectors are very close, we can assume that they belong to the same role. Each role will be represented by its *Cluster*, and vectors of the same role will be merged to the same cluster.

> ***<u>Definition 4</u>**: a Cluster is a group of highly similar Session Vectors.*

Each Cluster has its mean called the *Cluster Centroid*, which is defined as follows:

> ***<u>Definition 5</u>**: the Cluster Centroid is a vector CC with vector values that are the respective means of the cluster vectors.*

We can define *Support* of a Cluster as follows:

Definition 6: *The Cluster Support is the ratio between the number of Session Vectors belonging to this cluster and the number of all Session Vectors in the training set.*

Definition 7: *the distance D between two clusters presented by their centroids CC_1 and CC_2 is computed as:*

$$\boxed{D(CC_1, CC_2) = \sum_{i=1}^{n} \frac{|CC_1[i] - CC_2[i]|}{n}}$$

Now a clustering method is applied to produce a set of clusters (roles). If the exact number of distinct roles is known in advance, this is the desired number of clusters. Otherwise the main question is how many clusters should be expected? We propose using the hierarchical clustering algorithm of [1]. The algorithm builds the cluster sets layer by layer; starting with a very large set of clusters and ending with one big cluster (see Algorithm 1 below). The resultant tree is called the cluster Dendrogram tree [11]:

Algorithm 1:

```
Build_Dendrogram
{
    1. For each application session build its Session Vector.
    2. Start with each Session Vector as a separate cluster.
    3. Save all clusters received at this stage in a Cluster Set
       $CS_1$ and initialize i to 1.
    4. Select two closest clusters to merge into a single
       cluster.
    5. Compute the new cluster centroid for the merged cluster.
    6. Save all clusters received at this stage in a
       corresponding Cluster Set $CS_i$ and advance i by 1.
    7. Repeat steps 4-6 until we get a single cluster.
}
```

Figure 6 shows an example of the Dendrogram tree:

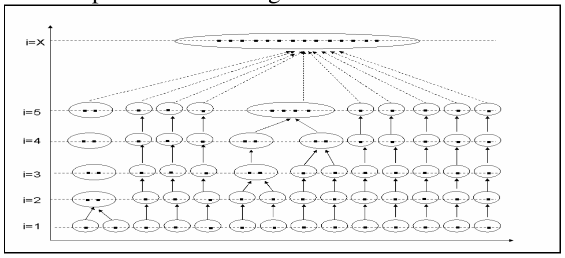

Figure 6: Dendrogram of clustering algorithm.

Our next task is to choose the best Cluster Set (layer of the dendrogram tree) to serve as a profile for DIWeDa. Different strategies exist to make such a choice, and to evaluate cluster set quality [9]. For our purpose, we will use the following criteria:

Definition 8: ***Intra-Cluster Distance*** *represents the compactness of clusters in a cluster set and is computed as:*

$$\boxed{\text{Intra\_Cluster\_Distance(Cluster - Set}_\text{k}) = \sum_{Cluster_j \in Cluster-Set_k} Support_j * \sum_{SV_i \in Cluster_j} D(SV_i, CC_i)}$$

where $Support_j$ implies the support of the cluster *j* (Definition 6). It is zero for the bottom layer of our dendrogram when each cluster contains only one session vector. It increases where the number of clusters decreases and clusters are spread-out. Thus Intra-Cluster Distance is a weighted sum of distances for each cluster in a given cluster set.

***Definition 9*: Inter-Cluster Distance** *represents the isolation of clusters and is computed as:*

$$\boxed{\text{Inter\_Cluster\_Distance(Cluster - Set}_\text{k}) = \sum_{Cluster_j \in Cluster-Set_k} D(CC_j, Global\_Centroid)}$$

where *Global_Centroid* implies the mean of all Cluster Centroids in a given cluster set. It is zero for the top layer of our dendrogram, and it is one at the bottom of the dendrogram when each cluster contains only one session vector.

As can be seen, when Intra-Cluster Distance increases, Inter-Cluster Distance decreases and vice versa. To estimate the quality of each cluster set (layer in dendrogram), we use their intra- and inter-distances with the approach known in the literature as *"Minimum Total Distance"* [9]. This approach finds a cluster set with small specific clusters that are far from the global centroid. By using this approach, DIWeDa will find specific separated roles.

$$\boxed{\begin{array}{l}\text{Minimum\_Total\_Distance} = MIN\langle\text{Intra\_Cluster\_Distance(Cluster - Set}_\text{k}) + \\ \text{Inter\_Cluster\_Distance(Cluster - Set}_\text{k})\rangle \mid \textit{for each layer k of Dendrogram}\end{array}}$$

For example, let us assume we found a cluster set which has a minimum cluster distance and this cluster set has 3 clusters which are presented by the following Centroids and have the following Supports:

CC_1 = {0, 0, 0, 1/7, 0, 0, 0}, Support = 1/120

CC_2 = {1/7, 3/7, 1, 0, 1/7, 3/7, 3/7}, Support = 61/120

CC_3 = {1, 1, 6/7, 5/7, 1, 6/7, 5/7}, Support = 58/120

As can be seen, we found 3 roles, but the support of the first role is very small. This can happen in two cases: either this role is not significant at all or this is a role of an intruder. Thus, after finding the best cluster set with the minimum total distance, it is reasonable to delete the clusters (roles) with Support < SIR.

To summarize, the algorithm of building cluster-based profiles is as follows:

Algorithm 2:

```
Build_Cluster_Based_Profile
{
    1. Find all application fingerprints with support > SIR and
       save them in Fingerprint Set
    2. Run Build_Dendrogram (Algorithm 1)
    3. Select the appropriate cluster set (layer in Dendrogram)
       with the Minimum Total Distance and save it in Cluster
       Set
    4. Delete Clusters from the selected Cluster Set with
       Support < SIR
}
```
Note that even when the distinct roles in an application are known precisely, using the above algorithm may be valuable: since some groups of users may behave differently within one application role, applying this algorithm would actually produce two different roles.

3.4 Content Anomaly Detector

At the learning phase, DIWeDa builds its profile that is based on a cluster set. At the detection phase DIWeDa will detect the session content anomalies by first computing the probability of an analyzed session to be abnormal. We assume the following two things influence the anomaly degree of an analyzed session:

- The distance of a session vector to the closest cluster centroid – the farther a session vector is from any existing cluster, the more abnormal a session is.
- The number of unexpected statements (NUS) in a session. An Unexpected Statement is a statement for which DIWeDa finds no corresponding fingerprint. For example, if an attacker changes the SQL structure by an SQL injection attack, DIWeDa will classify such a statement as an unexpected one, since it will not find the corresponding fingerprint in learned profiles. The more unexpected statements a session has, the more abnormal a session is. Notice, sometimes a legal session might have some unexpected statements: these statements are legal ones but simply were not learned during the training phase. But as the number of unexpected statements increases, the probability of a session being legal decreases rapidly.

Definition 10: *the probability of an analyzed session represented by its session vector SV to be abnormal is defined by the formula:*

$$\boxed{P(SV\ \text{is abnormal}) = \frac{MD + NUS^2}{1 + NUS^2}}$$

where *MD (Minimum Distance)* is the distance (defined in Definition 7) between SV and the closest *Cluster Centroid* from the cluster set and *NUS* is the *Number of Unexpected Statements* in the analyzed session. Notice, we use NUS^2 since we want to give a high weight to *NUS*.

For example, if we have a set of the two following clusters as the profile, where each cluster represented by its centroid: CC_1 = {0, 0, 0, 1/7, 0, 0, 0}, CC_2 = {1/7, 3/7, 1, 0, 1/7, 3/7, 3/7} and we have a session with no unexpected statements and represented by the following Session Vector: SV = {0, 0, 0, 1, 1, 1, 0}, then the distance to CC_1 is:

$$D(SV, CC_1) = \frac{|0-0|+|0-0|+|0-0|+|1-1/7|+|1-0|+|1-0|+|0-0|}{7} \approx 0.41$$

and the distance to CC_2 is:

$$D(SV, CC_2) = \frac{|0-1/7|+|0-3/7|+|0-1|+|1-0|+|1-1/7|+|1-3/7|+|0-3/7|}{7} \approx 0.76$$

Because $D(SV, CC_1) < D(SV, CC_2)$ we can compute the probability of the session as:

$$P(SV \text{ is abnormal}) = \frac{0.41 + 0^2}{1 + 0^2} \approx 0.41$$

Based on evaluating its SQL content, the last result means that an analyzed session has a probability of 0.41 to be abnormal. Next we show how the result helps to define an analyzed session as legal or intrusive based on the definition of *Session Intrusion Threshold:*

> ***Definition 11***: *the Session Intrusion Threshold (SIT) is represented by a number in the range [0, 1], where each session with the probability to be abnormal (Definition 10) is greater than this threshold, will be classified as an intrusion.*

Different choices of the Session Intrusion Threshold will lead to different behaviors of our IDS. A very high threshold will lead to a low false-positive rate, but a high false-negative rate; and low threshold will lead to a high false-positive rate and a low false-negative rate. During the empirical evaluation of our system (Section 4), different thresholds were tried, and for each threshold the ratio between the true positive rate and the false positive rate was computed. This ratio is called the *Receiver Operating Characteristic (ROC)* of the system and is used to estimate the effectiveness of the system for intrusion detection. The interesting thing that was learned from our evaluation was that the best threshold was very close to the ***Cluster Set Maximum Distance***.

> ***Definition 12:*** *the Cluster Set Maximum Distance is the maximum between Cluster Maximum Distances over all clusters in the cluster set.*
> where the Cluster Maximum Distance is defined as follows:
>
> ***Definition 13:*** *the Cluster Maximum Distance is the maximum distance between a vector in a cluster and its centroid over all vectors in the cluster.*

It seems very rational that sessions with a distance greater than the cluster set maximum will be classified as intrusions, since they are far apart from any existing cluster (role). Thus our proposal to use the Cluster Set Maximum Distance as the Session Intrusion Threshold is very intuitive. In Section 4 we experiment with different thresholds, and show that the system with the best performance results is the one when the threshold is chosen on the basis of the maximum distance.

4 Analysis and Evaluation

4.1 Experimental Setup

We have implemented a prototype of the proposed system and used it to evaluate the system's feasibility, efficiency and correctness. The prototype was developed with C# and SQL Server 2005. The input to the system was a Log file – a text file where each line presents a single SQL statement submitted by an analyzed application. The online bookstore application [16] was used as a web application benchmark. The profiles were built by manually operating this application. The analyzed sessions were created by synthetic data, as will be explained in Section 4.3. We used the following criteria for the quantitative evaluation of our system:

$$\text{True Positive Rate (TPR)} = \frac{\#\text{ of True Positives}}{\#\text{ of True Positives} + \#\text{ of False Negatives}}$$

$$\text{False Positive Rate (FPR)} = \frac{\#\text{ of False Positives}}{\#\text{ of False Positives} + \#\text{ of True Negatives}}$$

$$\text{Receiver Operating Characteristic (ROC)} = \frac{\text{True Positive Rate}}{\text{False Positive Rate}}$$

4.2 Training Set Description

The training set was built by the manual operation of the benchmark application. 200 different sessions were created, from which 198 were legal sessions and 2 with exploiting of application vulnerabilities of SQL injections and business logic violations. Thus, the Session Intrusion Rate (SIR) was 0.01.

All SQL sentences of the 200 sessions were written into the SQL log, which contained 7140 SQL statements. From this log, 165 different fingerprints were deduced by parsing each SQL statement. The 3 fingerprints with support less than SIR were deleted. Notice that of the 3 SQL sentences that have been deleted, two of them were from two different intrusion sessions and the last one was legal, but with a very small support. After this step we had 162 fingerprints in the fingerprint set.

After creating a Session Vector for each session, we used the Dendrogram Building algorithm described in the previous section. From 200 layers of saved cluster sets, using the Minimum Total Distance measure (see Section 3.3.2), we chose the best layer to serve as the DIWeDa profile. The chosen cluster set was a cluster set with 11 clusters, and the maximum distance in this cluster set was 0.07 – this distance was used to define the Session Intrusion Threshold (see Section 3.4).

4.3 Test Set Description

The test-case was built on the following patterns which will be referred to in our Evaluation Discussion Section:

1. Attacks Free Pattern (legal sessions)
2. Where clause modification pattern (sessions built on legal SQLs with where clause modification)
3. Field clause modification pattern (sessions built on legal SQLs with select clause modification)

4. From clause modification pattern (sessions built on legal SQLs with from clause modification)
5. SQL randomization pattern (sessions with randomly created SQLs)
6. Business logic escalation (mix of SQL contents of sessions belong to different roles)
7. Business logic escalation (mix of randomly chosen legal SQLs)
8. Business logic escalation (sessions with random order of SQLs)
9. Business logic escalation (sessions built on SQLs without their original contents)
10. Complex attacks scenario pattern (sessions with mix of previous patterns)

In the following table we summarize our results. The row ROC (Receiver Operating Characteristics) is computed as TPR/FPR and shows the ratio between True Positive Rate (TPR) and False Positive Rate (FPR) for Session Intrusion Thresholds (SIT) from 0.02 to 0.11:

Table 1: Summary of Session Intrusion Thresholds evaluation

	Session Intrusion Threshold								
	0.02	0.04	0.05	0.06	0.07	0.08	0.09	0.1	0.11
TPR	1	1	1	0.925	0.925	0.728	0.728	0.728	0.728
FPR	1	0.3	0.1	0.05	0.05	0.05	0.05	0.05	0.05
ROC	1	3.33	10	18.5	18.5	14.56	14.56	14.56	14.56

The following graph summarizes ROC for thresholds in the range [0.02, 0.11]:

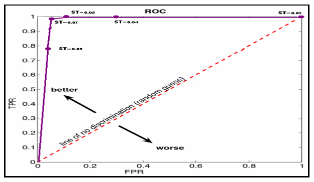

Figure 7: ROC for different Session Intrusion Thresholds (SIT).

The highest ROC achieved was 0.925/0.05= 18.5 with a Session Intrusion Threshold of 0.07. At this threshold the TPR = 92.5% and the FPR=5%.

4.4 Evaluation Discussion

Table 2 shows the results for each specific pattern.

Table 2: Evaluation summary

Pattern #	# of instances	# of TP	# of TN	# of FP	# of FN
1	200	0	191	9	0
2	5	5	0	0	0
3	5	5	0	0	0
4	5	5	0	0	0
5	10	10	0	0	0
6	5	5	0	0	0
7	5	5	0	0	0
8	5	0	0	0	5
9	15	15	0	0	0
10	5	5	0	0	0

As can be seen in Table 2, all attacks targeting the skeleton of an SQL sentence were detected (Patterns 2–5). This is very important because SQL injection attacks are too common today and many web applications are prone to them.

All cases of business logic escalation (Pattern 6) were classified as intrusion. This pattern merges SQL contents of different roles and shows that, for example, a student who also tries to act as a professor in the University system will have abnormal session SQL contents, and these SQL contents will be detected by DIWeDa.

Very interesting cases are presented in business logic escalation (Pattern 7) and also classified as intrusion by DIWeDa. This pattern shows that the SQL contents of web application sessions are not random, but have a regular structure which can be learned, and deviations from this structure can be detected. We see this by comparing the anomaly degree of the Attack Free Pattern (under 0.05) and the anomaly degree of sessions with random SQL contents (above 0.11). We conclude that the SQL content of a session which is presented by the Session Vector has a regular structure: sessions with the same role are very close one to another and can be consolidated to the same cluster, which is an abstraction of an enterprise role. Intrusion sessions have irregular structure and their session vectors are a great distance from any existing cluster.

Business role escalation (Pattern 8) was not detected and thus, our true positive rate was decreased. Since our method does not impose the order of SQL statements in the session, scrambling of an SQL order cannot be detected in the current system. At this stage, it is clear that business rules can be order sensitive, thus we intend to improve the algorithm by measuring the distance between sessions not only by using common SQL sentences that were issued by both sessions, but also by using the order of these sentences. This will be included in our future work.

All cases of business role escalation (Pattern 9) were classified as intrusion. This pattern shows that a single SQL statement has strong dependencies to other sentences, and if we run some statements without their original SQL contents, DIWeDa can detect the absence of these dependencies and thus is able to detect an intruder, trying to buy books without being authenticated or without paying.

To summarize our true positive rate – we achieved a rate of 0.92, which means that we are able to detect 92% of attacks. Included in the 8% of undetected attacks, there are attacks targeting the order of SQL sentences in a session. This is the main system improvement that can be done in our future work.

Analysis of proposed algorithm's time complexity shows that building profiles is a polynomial task in the number of SQL statements in the log, but analyzing a session is a linear task in the number of session's SQL statements. The following tables summarize the system performance evaluation:

Table 3: Session analyzing performance

# of sessions	Time in sec.
2 Sessions	3
4 Sessions	7
8 Sessions	13
16 Sessions	26
100 Sessions	140
200 Sessions	280

Table 4: Profile building performance

# of sessions	Time in sec.
2 Sessions	2
4 Sessions	3
8 Sessions	5
16 Sessions	32
100 Sessions	215
200 Sessions	480

To summarize our false positive rate – we achieved 0.05 on the test set. This means that 5% of classified sessions are false positives. It should be noted that our system is profile-based and for such systems this rate is low enough (for comparison with other systems, see, for example, Table 2 from [2] or Figure 3 from [6]). Some signature-based systems have achieved false positive rates below our rate, but this is done with a lower true positive rate. The main reason for the level of FPR achieved is that there are legal sessions in which SQL contents are slightly different from learned contents, thus DIWeDa classifies such sessions as intrusion. We assume that our training set, which was created manually, was relatively small and thus DIWeDa was unable to learn all the session's SQL contents. It seems that as real application logs will contain more information, DIWeDa will be able to learn more, thereby possibly making the FPR less than 5%. We plan to evaluate this on additional web applications in our future work.

5 Conclusions and Future Work

The motivation for this article was to propose a practical solution to the web database intrusion detection problem. DIWeDa profiles the normal behavior of different roles in terms of the set of SQL queries issued in a session, and then compares a session with the profile to identify intrusions. We look for intrusions at the session level and not at the statement/transaction level, as more traditional models do. We learn enterprise roles and look for anomalous sessions far from the learnt roles, enabling us to see anomalies which cannot be seen nor detected using previous models. One possible extension of our algorithm is its ability to deduce enterprise/application roles, which were previously unknown for web applications. RBAC models are widely used for old desktop applications, but most of the web applications do not use roles and the proposed algorithm can be very useful in porting web applications to RBAC models.

As we have demonstrated, our method detects attacks using SQL structures and session's SQL contents based on these structures. This enables us to detect new types of attacks, such as business logic violations. But sometimes data centric attacks can be accomplished without changing the SQL structure, but just by passing unauthorized SQL **parameters.** To detect parameter-based violations, we developed a similar framework and classifiers which are able to learn the distribution of parameters' values and detect deviations from them. We are currently experimenting with detecting such attacks via SQL parameter changes. We are also working on detecting invalid order of SQL statements in a session. We will present our results in a future paper.

References

[1] P. Berkhin. Survey of Clustering Data Mining Techniques. Tech. rep., *In Accrue Software,* San Jose, CA. (2002).

[2] E. Bertino, E. Terzi, A. Kamra, A. Vakali. Intrusion Detection in RBAC-administered Databases. *In Proceeding of 21st Computer Security Applications Conference, USA* (2005).

[3] T. Buehrer, B. W. Weide, P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. *In Proceedings of the 5th international workshop on Software Engineering and Middleware*, Portugal (2005).

[4] C. Chung, M. Gertz, K. Levitt. Demids: A misuse detection system for database systems. *In Proceedings of IFIP TC11 WG11.5 Third Working Conference* (1999).

[5] W. Halfond, A. Orso. Preventing SQL Injection Attacks Using AMNESIA. In *Proceedings of* 28th *International Conference on Software Engineering*, China (2006).

[6] Y. Hu, B. Panda. A Data Mining Approach for Database Intrusion Detection. In *Proceedings of the ACM Symposium on Applied computing, Cyprus*. Pages: 711 – 716 (2004).

[7] W.L. Low, S.Y. Lee, Peter Teoh , DIDAFIT: Detecting Intrusions in Databases Through Fingerprinting. *In Proceedings of the 4th International Conference on Enterprise Information Systems* (2002).

[8] J. Mirkovic, S. Dietrich, D. Dittrich and P. Reiher. Internet Denial of Service: Attack and Defense Mechanisms, *Publisher: Prentice Hall* (2005).

[9] B. Raskutti, C. Leckie. An Evaluation of Criteria for Measuring the Quality of Clusters. *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Sweden (1999).

[10] A. Roichman, E. Gudes. Fine-grained Access Control to Web Databases. *In Proceeding of 12th SACMAT Symposium, France* (2007).

[11] J. Seo, B. Shneiderman. Understanding Hierarchical Clustering Results by Interactive Exploration of Dendrograms. A Case Study with Genomic Microarray Data, Tech. rep., *In IEEE Computer Society Press* (2002).

[12] A. Srivastava, S.R. Reddy. Intertransaction Data Dependency for Intrusion Detection in Database Systems. Part of Information and System Security course, School of Information TEchnology, IIT Kharagpur (2005).

[13] S. Tran, M. Mohan. Use trusted context in DB2 client applications. Available at URL: http://www.ibm.com/developerworks/db2/library/techarticle/dm-0609mohan/index.html (2006).

[14] J. Woo, S. Lee, C. Zoltowski. Database Auditing, Available at URL: http://www.cs.purdue.edu/homes/akamra/cs541/DB_auditing_survey_paper.pdf.

[15] Controlling Database Access, Oracle9i Database Concepts Release 2 . Available at URL: http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96524/c23acces.htm.

[16] Online Book-Store application. The open source from: http://www.gotocode.com/apps.asp?app_id=3&.

[17] Acunetix, Available at URL: http://www.acunetix.com/.

[18] Connection Pool. Available at URL: http://en.wikipedia.org/wiki/Connection_pool.