Users Tracking and Roles Mining in Web-Based Applications

Yaron Gonen
Ben-Gurion University of the Negev,
Be’er Sheva, Israel
yarongon@cs.bgu.ac.il
Advisor: Prof. Ehud Gudes
ehud@cs.bgu.ac.il

ABSTRACT
When a database is accessed via a web application, web-users are not connect directly to the database, but rather via the web application. From a database point of view, such a connection is always established by the same db-user (i.e. the web application’s db-user) and specific data on the web-user is not available to the database. As a consequence, web-users’ specific data cannot be audited and fine-grained access control cannot be implemented. We propose a method that provide the ability to track the web-users in web databases. The new method can be applied to legacy web applications without requiring any changes in their existing infrastructure. Using the tracked database, we propose a method to identify logical sessions (business logic), which we will use to mine the true users-roles of the web application.

Advisor: Prof. Ehud Gudes

1. INTRODUCTION
1.1 Users Tracking
In today’s web application world, the HTTP protocol is used to connect users to application servers rather than directly to a database. This has important implications on applications security. Unfortunately, HTTP, while useful for its original purpose of static content, is stateless and is not optimal for many of today’s demands. Stateless means that there is a new connection established for every request. As such, the state of individual and recurring connections can’t be supported by the protocol. The stateless aspect of HTTP was solved by the application servers for client to application server communication. The solution utilizes browsers cookies and the ability of the application server to create and maintain transparent sessions and thus state for each client.

*Supported by the Lynn and William Frankel center for computer science.

However, this represents only half of the architecture. The connection from the application server to the database still remains a challenge. The application server connects to the database on the user’s behalf. How this is done makes all the difference for two critical reasons: security and performance. The ideal solution is to connect the users to the database using the 1:1 mapping model, i.e. every end user has a distinct database account, or schema. This is secure because the database knows who is connected and can employ its security capabilities, specifically access control and auditing. In the web environment the user is almost never directly connected to the database. The user is connected to the application, and the application is connected to the database (see Figure 1.1). One might therefore conclude the application should connect each application user to a distinct database account. However, this connection model design produces problems. One problem is security. This design would require the application to know the user’s database password. Another problem is resource limitation. Web applications typically support many end users. To connect each user to a private database account one will have to establish a dedicated database connection for each user. This could exhaust memory and computing resources on the application server, database or both. The solution is to use a Connection pool, a pool of pre-established database connections that links the application to a database schema. The application opens, and keeps open, several physical database connections. When an incoming request is made, the application will grab a connection from the pool, issue a query, and then return the connection back to the pool. Although this solves the resource limitation, it does not exempt the application from managing a list of database users and passwords, which is a security risk.

For many applications, the criterion of performance is so important that Connection pool is the only possible solution. Typically, web applications use a model of N:1 for mapping actual end users (application users) to database accounts. This all-to-one mapping means that the application connects all end users to the same database schema, while this schema has the union of all privileges for all users connected to it. This design implies several security problems:

- Auditing: The user’s identity is not supported and their individual actions are untraceable.
- Access Control: The principle of minimal privilege is violated. It is impossible to authorize the web application user with appropriate privileges at the database
level since all application users have access to the same data. It means that roles cannot be defined and no more fine-grained access control to the database exists. Ensuring that only the right privileges are available to the user is left only to the application.

- Intrusion Detection: Not only that the database does not identify the user who accesses it, but it is also impossible to follow SQL statements of the same user. Attackers can exploit this flaw to access sensitive data, or use unauthorized functions. There exist attacks, such as business logic violations [11], which could be seen only at the session level (composed of multiple transactions), and thus could not be detected at the database level.

There was an attempt to solve the problem of users’ identification in [11], but the proposed approach that is based on Parameterized views requires changes in both the web application and the database. Few methods for passing users’ identity from the web application to the database exist in latest Oracle [4] versions, but again, they require changes in legacy systems. Several third party vendors, such as Imperva [3] suggest a solution to this problem. However, there is no published technical information on how they do it and how good they are.

We propose a new method and system that provides a way to link each SQL statement in the database to the end user that performed it (via the web application), thus making the web application’s access to databases more traceable. Furthermore, we propose a method for fine grained access control and intrusion detection based on the tracking ability.

1.2 Logical Session Tracking

Logical Session is a sequence of consecutive actions that could be applied on the web application in a single business oriented task. For example, in a banking web application, in order to transfer money to another account, the following actions should be taken in their presented order:

1. Request a transfer form.
2. Fill the form.
3. Submit the form.

Identifying the different logical sessions is important because a logical sessions basically defines a role in the web application. Once discovering all the possible logical sessions (or roles), a deviation from any of them may indicate an attempt for intrusion.

1.3 Roles Mining

Usually, in a web-application architecture such as the one described in this work, the users of the application have access rights to certain modules of the application and they do not have rights at the database objects level. Using the tracked application users’ activities we can learn the permissions they have in terms of access rights to database objects and create permission-profiles of users, or in other words, roles.

The challenge we are facing is mining roles from the actual web-application usage information. Our data is the logical sessions, as tracked by the session identification method (Section 3.7). Each set of permissions used together within a logical session is a hint that these permissions should be granted as a group. The collected data holds information such as the set of permissions a user has, the set of users that access the same set of permissions, the frequency of using sets of permissions together, etc.

Existing research on this subject use a user-permission assignment table gained from the database definition data. In our research we would like to take advantage of the knowledge hidden in the monitored data in order to get an accurate mapping of users-permissions needs in the application.

1.4 Motivational Example

Consider a web-application that serves a brokerage firm. The web-application manages customer accounts, executes customers trade orders, handles interactions of customers with financial markets and updates news from the markets. Users of the web-application are divided into groups (or roles): Guest users a guest can query for security details and read market news. Customers a customer can submit trade requests on her account. Brokers a broker submits the trades to the market on the behalf of the customers. Markets the market is a group of users that actually make the transaction and updates the status of the submission.

Each role has different permissions at the database level. For example, Newsmen have an insert permission on the News table, but all the other roles have a select permission. CUSTOMERS have insert, update and delete permissions on the Trades table, Brokers have update permission and all the other roles have no permission at all.

Like most web-applications today, the database maintains a Users table that holds information about the web users like user-name, password, role, etc. The connection to the database is established via a single user, the web-application user. Notice that the connection to the database is performed before any user logs in, since the authentication process needs to check the given user name and password against the Users table.

From the database point of view all queries are performed by the same user. For example, all the following queries are performed by the same database user, however, it is obvious they are not performed by the same web user:

```sql
select from users where username =:user and password = md5(:pass) ;
insert into news (item) values (:news_item) ;
```
insert into trades values (:date, :action, :security);

We present a method to tag each query with its originating web user. Once we have done so, we can mine patterns of usage and easily track off course usage, which is probably a misuse or an intrusion.

1.5 Web Applications Frameworks

Many popular cutting-edge web applications frameworks today, such as Drupal [2], CakePHP [1] and Ruby on Rails [5], use this architecture of connecting to a database, so the need for such a tagging method is both for new and legacy systems.

2. PREVIOUS WORK

2.1 Users Tracking

Most database protection methods rely heavily on the identity of the user accessing the database [6]. Using such identity, the authorization provided by views, roles and the Grant/Revoke mechanism can be applied [10]. Much effort was invested in developing methods for detecting intrusions to databases. Various intrusion detection systems for databases that also rely on user identity have been proposed in the literature [12, 9]. Intrusion detection methods can be divided into signature-based and anomaly detection techniques. Signature-based techniques typically rely on the prior explicit knowledge of attacks patterns, which is in turn is represented by one or more signatures or rules that are stored in a database. The database is frequently updated with new signatures, based on new observations. Anomaly detection systems build models of a normal system behavior during a training phase, and then, using the models the systems attempt to detect deviations from normal behavior during a detection phase. Any deviation from normal behavior is treated as a potential attack.

Few methods for session boundaries identification that rely on known user identity have been proposed [8, 15]. The session identification task involves identifying session boundaries from the log of database events. The importance of session identification is derived from the following motivations: it enables grouping all user activities within one session and can be useful for performance optimization by utilizing the prediction of future queries; such grouping basically defines a role that can be used to derive the actual access rights of a user; a deviation from such a grouping may indicate an attempt for intrusion, thus it can be used for intrusion detection.

However, as explained previously, the user identity is unknown at the database level in the web environment. In spite of the importance of this problem and in spite of a lot of efforts we have found only one academic paper that deals with the problem of users tracking at the database level. In [11] Roichman and Gudes proposed a Parameterized views method that allows transferring the web user identity to the database, thus enabling the using of database’ built-in access control mechanisms. The drawback of the approach is that it requires changes in both, the web application and the database, thus making it unpractical for legacy systems.

Latest versions of Oracle [4] support several ways of preserving user identity through the web application to the database. For users known to the database, the proxy authentication in OCI, for example, can be used. OCI enables a web application to set up, within a single database connection, a number of “lightweight” user sessions, each of which uniquely identifies a connected user. The application can switch between these sessions as required to process transactions on behalf of users. But in order to support the fine-grained authorization, Oracle must maintain all the application users. Since these users are created/dropped dynamically by the web application and the number of such users may be tremendous, this solution is not practical for a typical web application. Further, in order to use this method, legacy application should be modified.

Several 3rd party products that refer the problem of user identification at the database level have been proposed. Imperva [3], for example, has developed multiple methods for identification of end users regardless of how these users access the database. According to vendors, Imperva’s SecureSphere product supports architectures that use connection pooling to aggregate multiple user sessions into a single database connection. SecureSphere tracks end users without requiring any changes in existing databases and web applications. However, Imperva has not published any technical information on how they do it and has not produced any measurements for their success.

2.2 Roles Mining

The role mining problem was defined by Vaidya et al [14] as the problem of discovering an optimal set of roles from existing user permissions. Their definition of the problem bound the approximation produced by an inaccurate number of roles found: Given a set user users U, a set of permissions P and a user-permission assignment UPA, find a set of roles, R, a user-to-roles assignment UA and a role-to-permissions assignment PA δ – consistent with UPA and minimizing |R|. In most role mining researches the user-permission assignment (UPA) are the basic information given as input to any role mining algorithm. This information is taken from the actual permissions granted at the database level, and the task of the mining is to find the optimal set of roles that can cover the UPA.

3. PROPOSED METHOD

Our proposed method will contain two layers, the Network Management layer, and the Users Tracking layer. The Network Management Layer will include modules which handle the acquisition and management of network packets. It will handle the TCP sessions reconstruction and HTTP and DB protocols parsing. The output of this layer is a full HTTP session between a Web Server and a user and a full session between a Web Server and a database (SQL statements). The Users Tracking layer will include algorithms which handle the Web Application users identification and mapping of users’ actions to SQL statements in the database. This layer is responsible for the discovering of the actions available in the Web Application, discovering of SQL statements possible in the system, and matching between the discovered actions and SQL statements. This layer can be operated in two modes: learning mode and real-time mode. In the learning mode the output of this layer is a set of actions existing in certain Web Application, a set of SQL statements usable in the database, and for each action the mapping to the set of SQL statements in the database. In the real-time
mode, the output of this layer is two log files. The first one contains a log of users’ actions in terms of Web Application actions. The second one contains a log of SQL statements with a username attached.

3.1 Web Application Actions Discovery

Each Web application has a way of working to accomplish its functionality and this way is represented by the ordered sequences of actions that users can perform. Users can only perform the operations available at the application interface and no other operation is available for the end-users. The main question we refer in this section is which actions are supported by any arbitrary Web Application. Typically, these actions are applied by HTTP GET or POST requests from the user’s browser. Let us look at the following HTTP request:

```
```

It contains the php script name (other script-engines such as ASP may be considered) followed by the list of parameters for the script. So, if we assume that there’s a different script to each of the actions in the Web application, we can harvest all the scripts, and compile a list of all the actions. However, let as look at the following example:

```
```

We face the problem: The script is the same for both requests, but the action itself resides in a parameter, whose name is of course arbitrary. As can be guessed, the script act.php adds the input name to a list if the parameter op equal to "add", and removes the input name from the list if the parameter op equal to "del". Our approach is to define action to be a concatenation of the script name and some relevant parameters of the script. The key question is how to identify the relevant parameters. We propose a method comprising of two phases: a training phase and a pruning phase. We assume we have a Web Server log file with all HTTP requests submitted by users to certain Web Application during specified period of time (this is a task of the Network Monitoring layer of our system to create such a log). First we split this log into two parts - one for the training and one for the pruning. We assume that all the possible actions appear in the training part, and we compile a list of possible actions out of all the HTTP parameter and their values. Next, we go over the pruning part. Every time we encounter an HTTP parameter with a different value then in the possible action list we remove it, since the value difference indicates this HTTP parameter is a web-application parameter.

3.2 User Name Identification

The goal of this task is to detect when a user is logging in into a Web Application and what is his/her ID (username, for example). Since the parameter’s name (login) varies and it’s application-dependent, we can choose one of the following methods to identify the username: Keywords - to maintain a list of words that might indicate a user logging in. Manual - every Web application wished to be analyzed will have to describe the login process, so the system will be able to identify the username field in HTTP requests.

3.3 Users Tracking in Terms of Web Application

After user login event has been detected, the goal of Users tracking task is to attach the user ID to all the actions user performs with Web application. This can be achieved by associating the user ID with HTTP session ID, which typically generated and sent from a web server to a client to identify the current interaction session. The client usually stores and sends the session ID as an HTTP cookie and/or sends it as a parameter in GET or POST queries. This approach will allow us to track all user activity in terms of Web application actions.

3.4 SQL Statements Identification

Our goal in this section is to find a set of SQL statements possible in certain Web database, given a log of SQL statements for some time period. Our approach 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 (or Web Application) inputs. If we replace the constants in each statement with variables, we get some high level representation of the SQL sentence called the fingerprint. In order to do so we first use an SQL lexical analyzer to create the parse-tree. Since the SQL statements in the real database log do not contain '?' as parameters, but the values themselves, we need a schema-description in order to differentiate schema objects from parameters. Using such schema-description given by the user/extracted from the database, we automatically extract the parameters from the statement and create a fingerprint.

3.5 Mapping from Actions to SQL Queries

Our main goal is to attach the username to each SQL query performed by the Web application on the database. So far we are able to attach the username to each action in terms of the Web application. All we need in order to complete the task is to know the mapping function from the Web application action to actual queries in the database for each action existing in the Web application. We propose the following method to achieve this goal. First of all, we define the problem formally. Given a set of actions, supported by the Web application \( A = a_1, a_2, ..., a_k \), a set of SQL queries supported by the database \( Q = q_1, q_2, ..., q_l \), a sequence of actions performed by the web-application during some specific period of time \( S_a = (a'_1, a'_2, ..., a'_i) \), \( a'_i \in A \) and a sequence of appropriative SQL queries \( S_q = (q_1, q_2, ..., q_m) \), \( q_i \in Q \) we should produce a subjective function \( f : A \rightarrow Q^{\omega} \) that maps every action to a sequence of SQL queries. For example, \( f(a_2) = (q_5, q_3, q_1) \), meaning that action \( a_2 \) is mapped to the sequence \((q_5, q_3, q_1)\). The algorithm for finding the map function goes as follows: for each action \( a'_i \in S_a \) we know from the input web log its start time and stop time, denoted \( t_{start}(a_i) \) and \( t_{stop}(a_i) \) respectively. Let \( S_2(a'_i) \) be the sequence of SQL queries executed on the database between \( t_{start} \) and \( t_{stop} \). Notice that \( S_2(a'_i) \) may contain queries executed by other actions then \( a_i \). We construct a dataset \( D(S_2(a'_i)) \) for all \( S_2(a'_i) \) (one sequence for each instance of action \( a_i \) in \( S_a \)). For each such dataset we find the longest frequent subsequence with a support of 1.0 using a sequence mining algorithm such as CAMLS [7]. Each such sequence represents a set of SQL queries that appropriative action \( a_i \) is mapped to.

3.6 Users Tracking in Terms of the Database

Given a mapping function \( f : A \rightarrow Q^{\omega} \), a tagged se-
sequence of actions $S'_A = \langle a_1^1, a_2^1, \ldots, a_n^1 \rangle$ and a sequence of SQL queries $S'_q = \langle q_1, q_2, \ldots, q_m \rangle$ we can produce a tagged sequence of SQL queries $S'_q = \langle q_1^1, q_2^1, \ldots, q_m^1 \rangle$.

3.7 Finding Logical Sessions

A logical session is a sequence of consecutive web actions that are applied in a single business oriented task. For example, in a banking web application, in order to transfer money to another account, the following task should be performed in that specific order:

1. Select an account.
2. Request a transfer form.
3. Submit the form.

The logical session identification task involves identifying session boundaries from the web-application log. We introduce a concept of Execution Tree. An Execution Tree represents all possible ways of working (business logic) for a given web application. We build this tree as follows: first, we create the root of the tree, which is the login action. Next, we create the set $A_f$ that contains all the actions in $A$ that appear first right after the login action, and add them all as children of the root. We now go over the web log, and for each action traverse the tree, adding children nodes as needed. Every time we encounter an action $a_i$ such that $a_i \in A_f$ we start a new traverse path at the root of the tree (this can happen when a user starts a new logical session without logging off).

3.8 Roles Mining

The input we get from the logical sessions identified consists of a set of tuples (one for each session), containing a session number, a user id and a set of permissions used in that session. From that input we construct a Permission Usage Lattice (PUL). A node in the lattice represents a role and consists of the permissions of the role and the users who had used these permissions in the same session. A node $n$ in the lattice is a sub-node of another node $m$ if $\text{SetOfPermissions}(n) \subseteq \text{SetOfPermissions}(m)$. The cardinality of a node $n$ is defined by the size of $\text{SetOfPermissions}(n)$. Constructing the lattice from the input is rather straightforward: tuples having identical set of permission are represented by a single node containing all the different users.

Keeping in mind that a node in the lattice represents a role, preserving all sets of permissions that appeared in the input may result with a large number of initial role candidates. In the worst case it may lead to $2^n$ nodes, where $n$ in the total number of permissions. However, aiming at a significantly smaller number of roles, we wish to reduce the number of roles by applying the following steps:

1. Removing role $n$ whose set of permissions is contained in a more general role $m$, and all users in $n$ also appear in $m$. This way we do not grant the users additional permissions.
2. Constructing a mechanism for an optimal score where both keeping a role and giving a user permissions results in bad scoring.

4. EVALUATION

In order to evaluate our system we need a comprehensive environment with thousands of users working in. We have decided to simulate it using a log generator that will generate HTTP sessions between a Web Server and a user and sessions between a Web Server and a database (SQL statements) from the network traffic. This generator will be based on the industry-standard TPC-E Benchmark [13]. TPC-E is an On-Line Transaction Processing (OLTP) workload. It is a mixture of intensive transactions that simulate the activities found in complex OLTP application environments. The database schema, data population, transactions, and implementation rules have been designed to be broadly representative of modern OLTP systems. The TPC-E emulator, which the log generator is based upon, models the activity of a brokerage firm that manages customer accounts, execute customer trade orders, and be responsible for the interactions of customers with financial markets. We omit here the database design and the transaction flow of the business model. TPC-E generates all the events in the system and passes them to our log generator which creates a web log record and a series of SQL statements. Since we generated the logs ourselves we know:

- The user of each web-action.
- Which web-action generated which SQL statements.
- The true roles of the web-application.

An evaluation for success can easily be set by comparing the results of our methods with the expected ones.

5. REFERENCES


technologies, SACMAT ’07, pages 31–40, New York, NY, USA, 2007. ACM.


