QUAN DO

14/12/2011

Dr. Hartness

Project Report

In this project, I am going to build a database management system called SHSU Airlines. The project will be designed for the airline company’s employee. The system’s main focus will be on the customers’ reservations after they have already booked their flights. The company only offers airline tickets inside the U.S.

There are some basic functionalities in the employee’s application. Employee can view flights’ schedules. Every flight that departs and arrives on time or delayed will be viewed, and recorded by employees. In addition, if new flights are available, they have to add them to the database, or if for some reasons, flights are canceled, they have to delete them from the database. The company will allow customers to ask for their flights’ statuses through phone calls. When customers provide their confirmation numbers, employees will be able to respond to them their flights’ statuses. Moreover, employees will be able to list all the customers in the database and search for one particular one by using his or her driver license number. When a new customer signs up for the company’s service, the application will add him or her to the database.

Also, I have made some assumptions in my project. Because this is an employee’s application, I will not be able to implement the customers’ reservations. I assume that it should be done by a customer’s application. The data from table *ReservationCus and ReservationFlig* are added by this application. Another assumption is that when customers register for flights, they will be sent confirmation numbers and they will use them for checking flights’ details. Lastly, my database includes a lot of functionalities that the application does not implement such as baggage claim, hotel and car rental service.

There are some security aspects that I would like to apply to my database. First of all, illegal access is one of the most common database security problems. This basically means that unauthorized users access the database system or legitimate users access with illegal purposes. According to Coffin Murray, a computer science professor at Kennesaw State University, “access control is a primary method used to protect data and limiting access to data. This can be done through authentication, authorization, and access control” (Murray, 2011,p.63). For example, in order to get access to data, most of DBMS require usernames and passwords from authorized users. However, authorized users are not allowed to access the entire DBMS. Instead, they will be assigned to some specific privileges based on their authentications. Second, SQL injection is a security database problem that penetrates and exploits DBMS by using SQL query. The issue usually takes place at logon section of applications. By putting malicious input into username textfield, illegal users will be able to manipulate different queries to the database causing many damages such as modifying, deleting and updating database records inappropriately. For example, the query:

SELECT\* FROM Flight WHERE fnum = 1234;

instead of typing 1234, illegal users might type 1 or ‘1’=1’, with this query, DBMS will be executed and will show the information of every flight, not just the flight with fnum = 1234. Displaying information that would not normally be available is very insecure for DBMS. This kind of vulnerability requires attackers to have a great understanding of database schema. However, it is not always easy to interpret what attributes are contained in a particular relation of some DBMSs, and then inject malicious input to change the implementation of SQL queries. Yeole and Meshram, computer science professors in Mumbai, India define “malformed queries”, a form of attack that attackers try to implement and debug the error message that DBMSs use to respond to them. Instead of typing string as‘1’=1’, attackers will put actual codes that “cause the query to contain syntax error, type mismatches, or logical errors” (Yeole, Meshram, 2001).

One of the most common and effective solutions for SQL injection is to use *Prepared Statement*. For example,

findCustomer=conn.prepareStatement(“SELECT \* from Customer

WHERE dl = ?”);

If users enters something like OR ‘1=’1’, it will not execute the input within the SQL query above, but instead, it will actually look at table Customer and find dl = ‘1=’1’. If it does not match any value, it will return the SQL exception. Another method that can prevent SQL injection is to use “tokenization”(Yeole, Meshram, 2001). The common characters that attackers use to inject SQL queries are “a space, double dashes, and single quotes”. So the main purpose of this method is to divide a SQL query into separated tokens, and detect the common characters from a particular one. For example, consider this SQL command:

“SELECT\* FROM Flight WHERE fnum = ?”;

this command will be tokenized into token 1: “SELECT\*”, token 2: “FROM”, token 3: “Flight”, token 4:”WHERE”, and token 5: “ fnum = ?”. So a programmer will write a program that looks at token 5, and be able to report to administrators if there is an attempted attack.

Another security database problem that is considered to be very difficult to detect is database inference. Database inference occurs when authorized users attempt to look at legitimate data and infer other information from restricted sources. This problem has remained very difficult to solve because it does not involve technical, but ethical issue. “The only recommended solutions include controls related queries (suppression) or controls related individual items in a database (concealing)” (Murray, 71). In other words, database administrators should be able to provide sensitive data carefully so that the users do not find it easy to infer other data. Motivation of these attempted users could be selling personal information or looking at some confidential data which might violate a company’s policy. Database security, as its core, has played an important role in DBMS. Some issues are easy to deter and prevent from happening again. However some are very complicated and difficult to block. Although sophisticated attacks are not easy to stop, there are some techniques that have been developed to detect the malicious sources. Auditing database is a technique that uses to track every step that users interact with database. By investigating DBMSs’ breaches, we will uncover vulnerabilities in DMBSs that were unable to foresee during systems’ designing process.

Cite Work

Murray, Coffin. Database Security: What Students Need to Know. *Journal of Information Technology Education: Innovations in Practice.* Volume 9, 2011. Kennesaw State University, Kennesaw, GA, USA.

Yeole, A S, Meshram, B B. Analysis of Different Technique for Detection of SQL Injection. *Journal of ACM Digital Library.* Copyright 2011.

**From ER to relational model:**

Flight(fnum,bid,price, meal,cabin, status, aircraft,pilot)

Customer(dl,vin,car\_out,car\_in,hcode,hotel\_in,hotel\_out,

name, email, phone, street, city, state, zip)

Depart(fnum,ddate,dtime,apcode,apname,city,state,zip)

Arrive(fnum,adate,atime,apcode, apname,city,state,zip)

Baggage(bid,fnum,fee,quanity)

LostandFound(bid,,reported,pickup)

CarRental(VIN,dl,company,cartype,color,apcode)

Hotel(hcode,dlname,address)

NORMALIZATION

NF ON TABLE FLIGHT:

Flight(fnum,bid,price, meal,cabin, status, aircraft,pilot)

FD:

Fnum->(fnum,price,meal,cabin,status,aircraft,pilot)

Fnum-> price

Fnum-> meal

Fnum-> cabin

Fnum-> status

Fnum-> aircraft

Fnum->bid

Fnum: is a candidate key

The table is in 1st NF because there is no multiple values in each attribute. It is also automatically in 2nd NF. Because there is no compound key, all of the non-key attributes depend upon the entire candidate key (one attribute: fnum). In addition, each non-key attribute stays independently from one another, so it is in 3rd NF. It is also in BFCN because fnum is the super key.

Flight(fnum, price,meal,cabin,status,aircraft)

FlightBag(fnum,bid)

NF ON DEPART TABLE: (note: it is also the same as Arrive table)

Because Depart and Arrive tables are weak entities that reference Flight entity, so they include its primary key (fnum).

Depart({fnum},ddate,dtime,apcode,apname,city,state,zip)

FD:

{fnum} -> (fnum,ddate, dtime,apcode, apname,city,state,zip)

Again, fnum is a single candidate key, so it is 2nd NF. However, there are transitive rules between non-key attribute, so it violates 3rd NF.

{fnum}->{apcode}

{apcode}->(apname,zip,city,state)

Thus, I will create a new table with apcode being a primary key. In this case, I keep (zip,city,state) in the same table because I assume that there must be only one airport in every city.

Airport({apcode}, apname,zip,city,state)

When Airport becomes a new table, it is automatically in BCNF.

So from the original Depart table, I come up with:

Depart({fnum},ddate,dtime,apcode)

Airport({apcode}, apname,zip,city,state)

Same thing with Arrive table:

Arrive({fnum},adate,atime,apcode)

Airport({apcode}, apname,zip,city,state)

NF ON TABLE CUSTOMER:

Customer(dl,vin,car\_out,car\_in,hcode,hotel\_in,hotel\_out,

name, email, phone, street, city, state, zip)

FD:

(dl)->(name, email, phone, street,city,state,zip)

(dl,vin)->(vin,car\_in,car\_out)

(dl,hotel)->(hcode,hotelin,hotelout)

(zip)->(city,state)

(dl,vin,hotel)->( dl,vin,car\_out,car\_in,hcode,hotel\_in,hotel\_out,

name, email, phone, street, city, state, zip candidate key)

The table is in 1st NF because there is no multiple values in any attribute. However, it is not in 2nd NF because subsets of the candidate key can determine sets of non-key attributes. So the table is decomposed as the following:

CustomerInfo(dl, name, email, phone, street,city,state,zip)

CustomerCar(dl,vin,carin,carout)

CustomerHotel(dl,hcode, hotelin,hotelout)

CustomerCar andCustomerHotel are also in 3rd NF because there is transitive rule between non-key attribute. However, CustomerInfo violates 3rd NF because dl->zip and zip->city,state. So the table is decomposed as the following:

Customer (dl, name, email, phone, street,zip)

Zipcode(zip,city,state)

Now CustomerInfo, CustomerCar, CustomerHotel and Zipcode are in 3rd and BCNF

NF ON RESERVATION TABLE:

Reservation(dl,fnum)

The relationship between Flight and Customer is many-to-many, so I create Reservation table to present this relationship. The reason I make it many-to-many relationship is because a customer can have many flights. For example, a customer is on a round-trip travel so he or she has at least 2 flights. And of course, a flight can have many customers. As a many-to-many relationship, there will be repeated rows of either dl or fnum, so I add another attribute *Confirmation number* to the relation and make it an only primary key. Then the relation would be become:

Reservation(conf,dl,fnum)

FD:

Conf->dl,fnum

Dl->fnum

Fnum->dl

then I can decompose this relation into

R1(conf,dl)

R2(conf,fnum)

Now R1 and R2 are in 3rd and BCNF.

TABLES IN MY DATABASE

FlightDetail(fnum,price,cabin,meal,status,aircraft)

FlightBag(fnum,bid)

Depart(fnum ,data,time,aport)

Arrive(fnum ,data,time,aport)

Airport(apcode, apname,zip,city,state)

ReservationCus(conf,dl)

ReservationFlig(conf,fnum)

Customer(dl,name,email,phone,street,zip)

CustomerCar(dl,vin,carin,carout)

CustomerHotel(dl,hcode, hotelin,hotelout)

Zip(zip,city,state)

LostandFound(bid,,reported,pickup)

CarRental(VIN,dl,company,cartype,color,apcode)

Hotel(hcode,dlname,address)

Baggage(bid,fnum,fee,quanity)