A Research in SQL Injection

LEUNG Siu Kuen

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Information Engineering

Supervised by

Prof. WEI Keh-Wei, Victor

June 2005

Abstract of thesis entitled:
A Research in SQL Injection
Submitted by LEUNG Siu Kuen
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in June 2005

As technology advances, more and more people will make use of web applications to handle their daily stuff, like their bank accounts. In the server side, the corporates will make use of database servers to store all information of their customers, including their personal information and their usernames and passwords. And the corporate application servers will retrieve the information stored in the database server for the operations.

However, once it becomes more and more popular, the attackers will be attracted to attack this kind of services. Unluckily, recently, hackers discovered that making use of some specially-designed inputs, which will be treated as part of the SQL query in the database, the attackers can retrieve all data stored in the database, including the sensitive information. This kind of attacks is called SQL Injection attacks.

The main purpose of this thesis is (1) to raise people awareness to this newly discovered problem, and (2) to propose a new defense system to solve this problem. In this thesis, we will first give a very brief introduction of what is SQL Injection attacks, with its working principles. Then we will present the current defense methods to this attack. Finally, we will present our proposed system – a system that can detect the current-known SQL Injection attacks and at the same time learn the new SQL Injection attacks.

概要

現今社會，科技越來越發達。越來越多人會利用網上服務來處理他們的日常事務，例如他們的銀行戶口。而在伺服器端，這些機構會利用資料庫伺服器來把所用客戶的資料，包括他們的個人資料及其賬戶之用戶名稱及密碼。而機構的應用伺服器則會從資料庫伺服器提取所需資料來進行相對應之工作。

但是，當網上服務越來越普及的時候，便會引來攻擊者來這些服務。不幸地，最近駭客發現在這些服務中，攻擊者可以利用一些特別設計的輸入字串，而這些字串會被資料庫伺服器視作SQL查詢句子的一部份。而攻擊者從中可以取得所有儲存在資料庫伺服器中的所有資料，包括所有敏感資料。這些攻擊就是SQL注入攻擊。

本論文的主要目的是（１）喚起所有人對這個新發現的問題的關注，及（２）建議一個新的防禦系統。在本論文，我們首先會有一個簡單地介紹什麼是SQL注入攻擊及其運作原理。跟著我們會介紹我們設計的防禦系統——該系統可以防禦已知的SQL注入攻擊，而同時可以學習一些新的SQL注入攻擊。

Acknowledgement

First of all, I would like to express my graditude to my supervisor, Prof. Victor Wei for his support and guidance during my final year project and my MPhil. studies. Besides, I have to thank for my classmates in the Information Security Laboratory – Allen Au Man-Ho, Tony Chan Kwok-Leung, Patrick Chan Pak-To, Fung Kar-Yin, Sebastian Fleissner, Joseph Liu Kai-Sui, Patrick Tsang Pak-Kong and Yuen Tsz-Hon, for their support. We have overcome a lot of challenges and developed invaluable friendship in this 2-year studies.

Besides, I would like to thank for other postgraduate classmates in the Department of Information Engineering, including Chen Chung-Shue, Anderson Cheng Ho-Ting, Ho Siu-Ting, Ho Siu-Wai, Kwok Pui-Wing, Michael Ng Cho-Yiu, Michael Ngai Chi-Kin, Poon Siu-San, Adrian Tam Sai-Wah, Tse Hok-Man and Kelvin Yeung Man-Chun, for giving me pleasurable moments during the MPhil. studies.

This work is dedicated to my mother, my father, my grandmother and my grandfather.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 A Story

Nowadays, people are very busy in working and they do not have much time to go to the bank. As a result, they will make use of the online banking system provided by the bank for their financial operations, like checking the amount of money in the account, handling the credit card payments, transferring money to their families, buying and selling stocks, ... etc.

As a normal user, you trusted the bank will provide enough security measures to ensure that the whole online banking process are secure enough, like the protocol used to transfer data between the bank and the clients is HTTPS, that is all the content transmitted will be encrypted; the network architecture of the bank is secure enough – only trusted traffic can pass through the firewall. Besides, to maximize the security, we will change our passwords frequently and will use a combination of letters, numbers and symbols as the passwords, and we will not tell the passwords to others as well.

Everything seems to work fine. In one day, you logged on to the banking system normally. However, you suddenly discovered that all the money in

your account had been stolen. You immediately reported it to the police and the bank. You were very sure that the password should not be known by others as you had just changed the password the day before. The bank started the investigation immediately – they checked the log file and found that no abnormal traffic has been broken in the firewall.

Later, the bank checked the log file of the application server, they discovered something strange – some strange strings had been inputted to the system using your login name in the log in page, without entering the correct password. However, the application server had still allowed that attacker to log in to the system. Then they continue checking the log file of the database server, which stored all sensitive information, including your username, password, and the details of your account, and they discovered that all the money had been transferred to another account.

You may now wonder why though the password has not been known by the attacker, he is still able to log on to the system to use your account. Actually, the attacker made use of the SQL Injection attacks.

1.2 Overview

In the information world, there are so many attackers trying to launch attacks in order to cause damages to other people. In the information security world, we can classify the attacks into different types according to their attack purpose.

- to gain the control of the system
- to destroy the system/ to make the system unusable

So we will describe the following attacks in this section

- Code imperfection
- Denial of Service attack
- Malware

Code Imperfection

Code Imperfection, or Code Exploits are the attacks which are caused by the flaws in the code in the software development stage. The main purpose of this attack is to compromise the victim's system. One of the famous examples is buffer overflow attack [1]. In this kind of attacks, the attackers will try to find out the location of the imperfection in the programs. Then they will provide some special-designed inputs to these locations.

However, as the codes are imperfect, the program will not be able to detect such inputs, one of the example is the developers forget to validate the inputs. As a result, the program continues its process with this specially designed inputs. Finally, the attackers' code will be executed and as a result, they successfully launch the attacks and the system will be compromised.

Denial of Service Attack

The main purpose of Denial of Service attack (DoS attack) is to overload the target system. One of the examples is ping flood. Once a machine is connected to the Internet, it will have the chance to be attacked by DoS attacks. The attackers will try to continuously send a lot of packets to the target system, in order to consume all the resources of the target system. For example, if the system is a web server, the attackers will try to send a lot of packets to the HTTP port of the system, so as to make the HTTP service unavailable to other clients.

A further approach of DoS attack is Distributed Denial of Service attack (DDoS attack), in which the attacker will first compromise many victim systems, using the virus or backdoor programs. Then the attacker will control those victims to attack the target system at the same time to overload it.

Malware

Malware is a harmful computer program installed to the victim's system. The common types of malware are virus, worm and backdoor.

Viruses are mainly executable files. When it is executed, it will try to destroy the system, like deleting some files in the operating systems or destroying the boot sectors. Other computers will get infected if they execute the infected files.

Worms are similar to viruses. However, worms will try to spread themselves instead of passively waiting other computers to execute the infected files.

For viruses and worms, their main purpose is to destroy the victim's system so as to make it unusable. However, a backdoor program is different.

A backdoor is a computer program which is installed to the victim's system. However, the victim will not be able to notice that backdoor programs are installed as these programs will run in a hidden way. The main purposes of the backdoor programs are

1. to monitor the usage of the victim's computer. For example, the program will try to listen all the keystrokes in the system, so as to get the username and password of the victim's system.
2. to leave a backdoor by opening ports in victim's system, so as to give a way for the attackers to access the system.

In order to make the victims not to aware of the installation of the backdoor programs, the attackers will try to integrate the backdoor programs into some normal programs. When the victims execute those normal programs, the backdoor programs will be executed silently.

1.2.1 Introduction of SQL Injection

Database Management System (DBMS) has become one of the most essential components in modern computing. Most applications make use of

DBMS to store and retrieve data. Database has many advantages for data management, including

- Data independence
- Efficiency
- Concurrent access
- Data Integrity
- Reducing the application development complexity and time

Structured Query Language (SQL) has become the standardized language used in database in order to provide the abstract view of data from the actual storage. Making use of SQL statements, we can achieve data retrieval, insertion and update very easily.

In the real world, many applications are using SQL statements to get the data in DBMS. Let's use online banking service as an example. A user will first input his/her username and password to verify his/her identity similar to that in Figure 1.1. This information will be transmitted to the server.

In the server side, the program will make use of the user input to compose a SQL statement. Then this statement will be used to query the database to see if the input username and password are correct.

Recently, hackers discovered that if the SQL statements are written improperly, or the server settings are not correct, they can make use of some well-designed inputs to launch attacks to retrieve all secret information, to alter the data without permission and even to take control of the whole system. This kind of attack is called SQL Injection, which is classified as a kind of code imperfection attack.

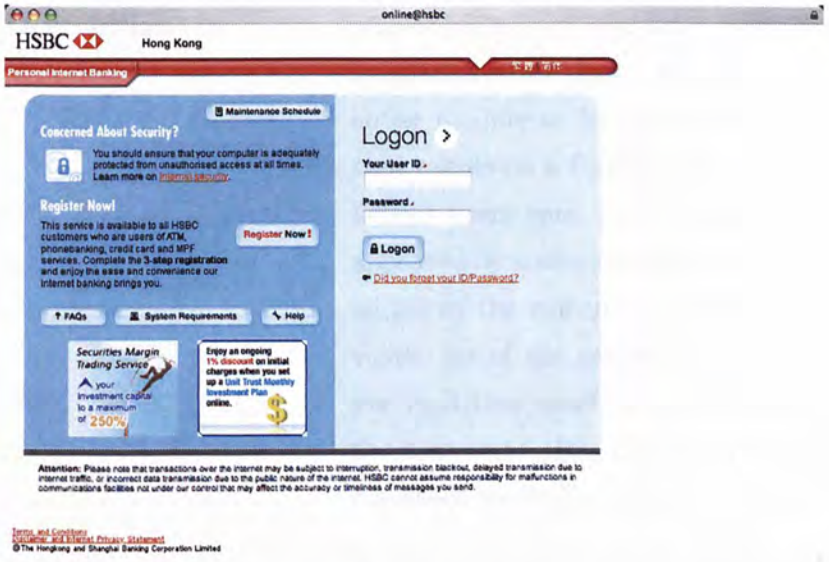

Figure 1.1: An example of online banking, it first authenticates the user by username and password.

1.3 The importance of SQL Injection

Someone may wonder the story in Section 1.1.1 is exaggerated. In order to see how important the SQL Injection is, we would like to demonstrate some surprising statistics.

According to a study performed by the Gartner Group on over 300 websites, 97% of them were discovered to be vulnerable to web attacks, including SQL Injection attacks.

Another statistics comes from CERT Advisory. According to CERT Advisory, currently there are 10 Vulnerability Notes about SQL Injection. Let's take a few as examples.

CERT Vulnerability Note VU#925166 (19^{th} October 2004)

Overview	PhpWebSite calendar module contains a SQL Injection vulnerability

Description PhpWebSite is an open-source web content management system that includes a web-based calendar module to let users to create, post, and view events on a PhpWebSite managed site. By default users must have requests for new events approved by a site administrator before they are added to the calendar. However, lack of input validation of the cal_template variable may allow malicious users to inject a SQL query into the new event. If a site administrator approves the event the SQL query will be executed.

Impact A remote attacker may be able to execute SQL queries on a server with the privileges of a PhpWebSite administrator.

CERT Vulnerability Note VU#264097 (17^{th} March 2005)

Overview NotifyLink contains multiple SQL injection vulnerabilities

Description There are multiple vulnerabilities in NotifyLink that allow unauthenticated remote users to view or modify the contents of the NotifyLink SQL database. Possible modifications include the addition of unauthorized user and administrator accounts.

Impact These vulnerabilities allow unauthenticated remote users to view or modify the contents of the NotifyLink SQL database. This database may contain email message text, encryption keys, and various authentication credentials. Possible modifications include the addition of unauthorized user and administrator accounts.

CERT Vulnerability Note VU#961579 (9^{th}June 2004)

Overview	Oracle E-Business Suite SQL Injection vulnerabilities
Description	Oracle E-Business Suite fails to filter user input permitting the exploitation of SQL injection vulnerabilities. These vulnerabilities may allow a remote attacker to execute procedures or SQL queries and updates on the vulnerable database application. This vulnerability is not platform specific.
Impact	An unauthenticated attacker may be able to exploit this vulnerability to execute procedures or SQL queries and updates inside the database. This may lead to compromise of the system and data integrity issues.

Even though many vulnerabilities have been discovered, SQL Injection attacks still continue to be an important security problem as there are no effective solution to detect and prevent this attacks.

1.4 Thesis organization

The rest of this thesis will be organized as follows:

Chapter 2	In this chapter, we will discuss some background material related to my work. It will briefly introduce the working flow of the web application using database management systems. Then it will start discussing the structure of database management systems. Finally, we will explain the syntax of some SQL statements which will be frequently used in this thesis.

Chapter 3	In this chapter, we will start discussing the main focus of this thesis, SQL Injection. We will briefly explain the approaches of different SQL Injection attacks, including SELECT Injection and UPDATE Injection, with the aid of some examples.
Chapter 4	In this chapter, we will first point out the causes of SQL Injection attacks. Then we will present the current defense methods to SQL Injection attacks.
Chapter 5	In this chapter, we will discuss our work, which is a new defense system to defense SQL Injection with learning ability to learn the new SQL Injection attacks. We will break the whole system into 5 stages to explain how does the system work in defensing the existing SQL Injection attacks together with learning the new attacks which does not known by the system beforehand. We will then use some examples to demonstrate how does this system work.
Chapter 6	In this chapter, we will provide a summary of this thesis.

□ **End of chapter.**

Chapter 2

Background

2.1 Flow of web applications using DBMS

Figure 2.1 demonstrates the overall work flow of a client requesting the information stored in the database in the Internet using the web applications stored in the application server.

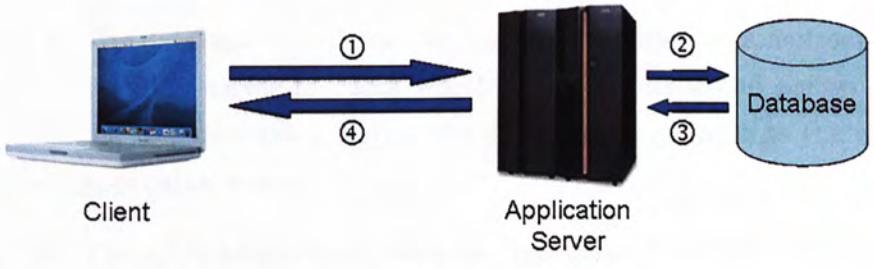

Figure 2.1: The work flow of a client requesting information stored int database in the Internet using web application.

We can divide the whole process into 4 stages.

STEP 1 — First of all, the client sends the request to the program like the PHP, ASP, JSP running in the application server.

STEP 2	The program sends the requests (SQL statements) to the database server for the corresponding database operations.
STEP 3	The database server executes the SQL statements and return the corresponding output to the application server.
STEP 4	The application server processes the output from the database server, and then provide suitable outputs or actions to the client.

Using the bank account login example in Figure 1.1, the whole process is

1. A user inputs his/her username and password, and send it to the application server.
2. The program executing in the application server constructs SQL statements using the user's input, and sends the SQL statements as requests to the database server.
3. The database server executes the SQL statements came from the application server to check whether the username and password from the user is correct. Then the database server returns the result to application server.
4. The application server bases on the output from the database server for further actions. For example, if the username and password are correct, the application server will forward the client to next page about his/her account information. Otherwise, an error message will be prompted to the client.

2.2 Structure of DBMS

2.2.1 Tables

Every database is composed of a collection of one or more relations, which are represented by tables. That is, in each table, all data stored will have some relations. It makes other people more easy to understand the contents of the database. Table 2.1 is an example of a table *SystemUserInfo*, which stores the information (username, name and password) of all users in the system.

login	name	password
robert	Robert Leung	123456
adrian	Adrian Tam	abcdef
jacky	Jacky Chan	testing
admin	System Administrator	pAsSwOrD

Table 2.1: Table *SystemUserInfo*

2.2.2 Columns

In the database, a table consists of many columns. Each column will be identified by a unique column name. For example, Table 2.1 has 3 columns, namely the *login*, which store the login name of all users; *name* for the full name and *password* for the password.

2.2.3 Rows

In the table, all records are stored in the form of rows. Actually a row is an instance of a relation. For example, in Table 2.1 , there are 4 rows, each row representing the record of a user; like in the first row, it stores the record fullname, login name and the password for the user robert.

2.3 SQL Syntax

For the ease of understanding, in the section, we will use the table *UserDB* (Table 2.2) to explain the SQL syntax.

UserID	Username	Address	Phone
100	Robert Leung	Sham Shui Po	26098482
80	Patrick Tsang	Sham Shui Po	26098482
120	Patrick Chan	Fan Ling	23456712
99	Tony	Hong Kong Island	25123452

Table 2.2: Table *UserDB*

2.3.1 SELECT

The basic form of a SQL query is

SELECT [*fields*]
FROM [*table*]
WHERE [*conditions*]

If we want to select all the fields, we can use the wildcard character *.

Explanation: The above statement is used to retrieve the information we want from the specified table. The following example is to get *Username, Address* from the table *UserDB* given that the *UserID* is greater than 100.

SELECT $UserID, Username, Address, Phone$
FROM $UserDB$
WHERE $UserID >= 100$

And the result set of the above query will be

100	Robert Leung	Sham Shui Po	26098482
120	Patrick Chan	Fan Ling	23456712

2.3.2 AND/OR

From the above SELECT statement, we can see that users can select particular data fulfilling some particular conditions. In the example in Section 2.3.1 , the condition is *UserID >= 100.*

However, if we want to have more condition in one query, you need to use **AND** or **OR**. The usage of **AND** and **OR** is

[condition 1]

AND/OR

[condition 2]

AND/OR

⋮

[condition n]

The difference between **AND** and **OR** is that for **AND**, a data will be returned if both conditions are fulfilled while in **OR**, only one of them is enough. Let's see two examples to illustrate this difference.

SELECT $*$
FROM $UserDB$
WHERE $UserID >= 100$
AND $Address =' ShamShuiPo'$

The database will return the data from the table *UserDB* if the data's *UserID* is greater than or equal to 100 and the *Address* must be equal to

Sham Shui Po. So the following result set with only 1 instance will be returned.

100	Robert Leung	Sham Shui Po	26098482

However, if the **AND** has been changed to **OR** in the above query, that is, the query becomes

SELECT $*$
FROM $UserDB$
WHERE $UserID >= 100$
OR $Address =' ShamShuiPo'$

The selection criteria will be changed to select those data in which the *UserID* is greater than or equal to 100, or the *Address* is equal to Sham Shui Po. So the following result set with 3 instances will be returned.

100	Robert Leung	Sham Shui Po	26098482
80	Patrick Tsang	Sham Shui Po	26098482
120	Patrick Chan	Fan Ling	23456712

2.3.3 INSERT

The basic syntax to insert a row into the database is

INSERT INTO $[tables](field1, field2, ...)$
VALUES $(value1, value2, ...)$

Explanation: The above statement is used to insert a new row to the table. The following example is used to insert a new row containing the user ID,

name, address and telephone number of a user *John* into the table *UserDB*.

INSERT INTO *UserDB*
VALUES (123,'*John*','*Kowloon*', 12345678)

So after the above operation, the table *UserDB* will become

UserID	Username	Address	Phone
100	Robert Leung	Sham Shui Po	26098482
80	Patrick Tsang	Yuen Long	26098482
120	Patrick Chan	Fan Ling	23456712
99	Tony	Hong Kong Island	25123452
123	John	Kowloon	12345678

2.3.4 UPDATE

The basic statement to update a current row in the database is

UPDATE FROM [*tables*]
SET [*field*1] = [*value*1]
WHERE [*condition*]

Explanation: The above statement is used to update fields in existing rows in the table. We can set some conditions in the where field to specify which rows will be updated. For example, given that we know Patrick Chan's ID is 120, he has moved from Fan Ling to Ma On Shan, so we use the following statement to update Patrick Chan's details in the table *UserDB*.

UPDATE FROM *UserDB*
SET *address* = '*NT*'
WHERE *id* = 120

After the above operation, the table *UserDB* will become

UserID	Username	Address	Phone
100	Robert Leung	Sham Shui Po	26098482
80	Patrick Tsang	Yuen Long	26098482
120	**Patrick Chan**	**Ma On Shan**	**23456712**
99	Tony	Hong Kong Island	25123452
123	John	Kowloon	12345678

2.3.5 DELETE

The basic syntax to delete rows in the database is

DELETE FROM [*tables*]
WHERE [*conditions*]

Explanation: The above statement is used to delete rows from the table according to the given condition. The following example is used to delete the rows in *UserDB* where *Username* is *John*.

DELETE FROM *UserDB*
WHERE *Username = 'John'*

So the table *UserDB* will become

UserID	Username	Address	Phone
100	Robert Leung	Sham Shui Po	26098482
80	Patrick Tsang	Yuen Long	26098482
120	Patrick Chan	Ma On Shan	23456712
99	Tony	Hong Kong Island	25123452

2.3.6 UNION

Sometimes we need to combine the result from more than one query. In this case, we may need to use the keyword **UNION**. The basic statement to joint two queries is

[SQL Statement 1]

UNION

[SQL Statement 2]

Explanation: The above statement will joint the two result sets from two different SQL statements into one. The columns need not to have the same name. However, they must have the same data type. Besides, the number of columns in *Statement 1* and *Statement 2* must be the same.

For example, we would like to find out all user names in the table *UserInfo* together with that in *SystemUserInfo* in Section 2.2, we will then make use of the following statement to achieve this.

SELECT *Username*
FROM *UserDB*
UNION
SELECT *name*
FROM *SystemUserInfo*

So the result set returned from the above UNION query is

□ **End of chapter.**

Robert
Patrick Tsang
Patrick Chan
Tony
Robert Leung
Adrian Tam
Jacky Chan
System Administrator

Chapter 3

Details of SQL Injection

3.1 Basic SELECT Injection

In many applications like web applications, data will be stored in the database server. For example, we are developing a user login system, where the usernames and passwords have been stored in the database server. Table 3.1 describes the contents of the table *UserInfo*.

In order to validate the user identity, the user will first input him login and password. These data will then be transmitted to the server side. The server program will then construct a SQL statement from these username and password like

login	password
robert	123456
adrian	abcdef
john	testing
admin	pAsSwOrD

Table 3.1: Table *UserInfo*

SELECT $*$
FROM $UserInfo$
WHERE $login = '[user\ input\ login\ name]'$
AND $password =' [user\ input\ password]'$

For example, if the user *robert* logs in with his correct password, the whole SQL statement should be

SELECT $*$
FROM $UserInfo$
WHERE $login = 'robert'$
AND $password =' 123456'$

For those people without accounts, they cannot login to the system. However, consider the case when a hacker input the login name as *abc'* and the password to be *cde*. The SQL statement will become

SELECT $*$
FROM $UserInfo$
WHERE $login = 'abc''$
AND $password =' cde'$

However, the above statement is invalid because there are 2 's after *abc*. Actually ' is the most important thing to make SQL Injection works. For example, if John, a user in the *UserInfo* is a hacker and he makes use of his legitimate username and password to input the login as *john* and password as *testing' AND '1'='1*, the login SQL statement will become the following

SELECT *
FROM *UserInfo*
WHERE *login* = *'john'*
AND *password* =*' testing'*
AND '1' =' 1'

Even though the password John uses a wrong password *testing' AND '1'='1* instead of *testing*, he can still login successfully to the system. Why?

Let's look at the SQL statement again. In the WHERE clause, the database server finds out the rows with *login* equals to *john*, *password* equals to *testing*, together with a logical check if 1 equals to 1 or not. Since the check *'1'='1'* always returns true, actually the database server will find out the rows fulfilling the first two criteria. This kind of meaningless check which will always return true is called tautology. From Table 3.1, we can see that john's row will be selected and so John can still be granted access to the system.

However, you may wonder if a hacker doesn't have a legitimate username and password, how can he get into the system. Actually using *OR* keyword in SQL statement, the hacker can easily achieve this. Let's see the following SQL statement with the login name is entered as *abc* while the password is *abc' OR '1'='1'*.

SELECT *
FROM *UserInfo*
WHERE *login* = *'abc'*
AND *password* =*' abc'*
OR '1' =' 1'

The above statement will always give all contents in *UserInfo* as the result.

login	title	message	date
admin	First	I'm the first	1-1-2005
robert	Second	I'm the second	2-1-2005
adrian	Hello	Hello everyone	3-1-2005

Table 3.2: Table *Message*

Why? It's because in the last part of the WHERE clause, *OR '1'='1'* is inserted. So no matter what are inputted, the WHERE clause will always return TRUE since '1' is always equal to '1'. So even though a hacker know nothing, he can still break into the system.

3.2 Advanced SELECT Injection

3.2.1 Single Line Comment (−−)

Before going further, we need to know about single line comment in SQL. Single line comment, which is represented by −− in SQL, is used to tell the SQL server that it only needs to consider the statement before −−. That is, the thing after −− will be ignored.

3.2.2 Guessing the number of columns in a table

After breaking into the system, the most attractive thing to do is to find out users', especially root's password. Let's use a simplified version of a forum (Table 3.2) as an example to show how to get others' password.

In the forum, a search function has been implemented. Selected users' messages will be listed. Obviously, the SQL statement to do this should be

SELECT $*$
FROM *Message*
WHERE *login* = '[*search condition*]'

So one direct approach to get users' password is to make use of the above statement together with *UNION*. Using *abc' UNION SELECT * FROM UserInfo* as the input, the above SQL statement will become

SELECT $*$
FROM $Message$
WHERE $login = 'abc'$
UNION
SELECT $*$
FROM $UserInfo'$

Seems everything works. However, the system returns error. What's wrong? First of all, remember there is a ' at the end of the original statement. However, we haven't properly treat it. So it remains in the injected statement and so the above statement is not correct in syntax. So what can we do to make the statement correct in syntax? Yes, we have to use the Single Line Comment in Section 3.2.1. By adding – at the end of the input, that is the input becomes *abc' UNION SELECT * FROM* UserInfo––, the SQL statement will then become

SELECT $*$
FROM $Message$
WHERE $login = 'abc'$
UNION
SELECT $*$
FROM $UserInfo - -'$

So the ' after – will be ignored, and the database will only treat the statement as

SELECT *
FROM *Message*
WHERE *login* = *'abc'*
UNION
SELECT *
FROM *UserInfo*

Therefore, the statement is now correct in syntax. However, this statement still returns an error. What's wrong? Remember in Section 2.3.6, both statements in UNION should have the same number of columns. However, being an outsider, the hacker should have no knowledge about the structure of the table *Message*, like how many fields it has.

So what should we do in order to make the above statement works? A small trick is needed. Let's see what will be the output if the following statement is executed.

SELECT 1, 1, 1, 1

You may think the above statement is not logical, yet it is valid. A row with 4 columns of 1 will return. So how can a hacker makes use of this "special" property to successfully retrieve users' password. Actually he needs to do this by bruce-force. Since he doesn't know the number of fields in *Message,* first he needs to guess how many columns will be returned using the input *abc' UNION SELECT 1,1--* to make the following statement.

SELECT $*$
FROM *Message*
WHERE $login = 'abc'$
UNION
SELECT $1, 1 - -$

Again, in Section 2.3.6, the corresponding columns in two different queries should return the same data type. Since 1 has more than 1 data type, including integer and string, it is used in the above statement for the ease of checking. If an error returns, it means the number of columns in the second query are different from the first one. So we can vary the number of 1's in the second part of the query until no error has been returned. In the above example, 4 1's are needed in order to have a valid SQL statement. That is,

SELECT $*$
FROM *Message*
WHERE $login = 'abc'$
UNION
SELECT $1, 1, 1, 1 - -$

The above statement will return all rows in *Message* which is created by *abc* together with a row with all 1's.

3.2.3 Guessing the column name of a table (Easy one)

Getting the number of columns in *Message* is just the first step. After that, we need to guess the name of the tables and the name of columns we are interested in. If the system is poorly configured, we can get them easily. What is meant by a poorly configured system?

Let's use web server with PHP as an example. Suppose a web server is configured to display all errors to user (*display_errors=on*, which is the default configuration). If a hacker tried the injection statements in Section 2.3.6, detailed error messages, including the whole SQL statement will be displayed. Hackers are very happy to this configuration as it leaks too much information, they can get the column names of the table without paying any effort.

Once the hacker gets the tables and columns name, he can launch the attacks easily. Using the above example, he knows that the table storing the user information is *UserInfo* while the columns in it are *login* and *password.* Also he knows there are 4 columns in *Message.* So he will use *abc' UNION SELECT login,password,1,1 FROM UserInfo* as the input to construct the following SQL statement.

$$
\begin{array}{ll}
\textbf{SELECT} & * \\
\textbf{FROM} & Message \\
\textbf{WHERE} & login = 'abc' \\
\textbf{UNION} & \\
\textbf{SELECT} & login, password, 1, 1 \\
\textbf{FROM} & UserInfo
\end{array}
$$

So you can guess what will be returned from the above statement. After this statement is executed and a result set has been produced, the search function mentioned will display all contents in the result set to user as usual. So the hacker will get all users' logins and passwords easily.

3.2.4 Guessing the column name of a table (Difficult one)

The attacks in Section 3.2.3 works only if the server is configured poorly. However, if the server is configured such that those error messages will not be directly displayed to users, can the hacker still be able to use the above

method to retrieve users' login and password? The answer is yes. However, more effort is needed.

Actually the method is very simple, just bruce-force. What the hacker needs to do is just simply guessing the tables name and the columns name.

SELECT $*$
FROM $Message$
WHERE $login = 'abc'$
UNION
SELECT $1, 1, 1, 1$
FROM $[guess\ table\ name] - -$

First of all, the hacker needs to get the correct table name by varying the *[guess table name]* until no error occurs. Once he get the table name, like *UserInfo* in the above example, he can start guessing the column name one by one.

SELECT $*$
FROM $Message$
WHERE $login = 'abc'$
UNION
SELECT $[guess\ 1^{st} column\ name], 1, 1, 1$
FROM $UserInfo - -$

Similar to guessing the column name, the hacker will try varying the *[guess* 1^{st} *column name]* until no error occurs. Once he get the first column, he will continue to guess for second column by varying the column name in the place of the second 1 in the above SQL statement.

However, you may wonder if guessing the table and column name is a very hard task or not. Absolutely it is. However, most of the time the adminis-

login	password	right
robert	123456	user
adrian	abcdef	user
john	testing	user
admin	pAsSwOrD	admin

Table 3.3: Modified Table *UserInfo*

trators or the programmers will name the tables and columns according to the their purpose. We can actually launch a dictionary-attack. For example, we will guess the column with password with the name *password, key, keyword,* ... etc. For details, please refer to the Appendix.

3.3 UPDATE Injection

The hacker will not be satisfied to just viewing all the data in the database. He will continue his work to get benefits. For example, if the database is storing the bank account information, he will alter his account so that he can get a lot of money.

Let's modify the above example a little bit. We add a column *right* to Table 3.1 so that it becomes Table 3.3.

The hacker, John would like to change himself from normal user to the administrator. Suppose after the above attacks, he knows the structure of the whole database, including the tables and columns name.

Suppose there is a "Change Password" function in the above forum, allowing the users to change their passwords. John, then can make use of this function to alter him privilege. Suppose the original SQL statement to change the password of a user is

UPDATE FROM *UserInfo*
SET *password =' [user password]'*
WHERE *login =' [user login]'*

The above SQL statement will obviously alter the password a user to him/her defined one. However, if *john* enters *testing', right='admin* as the password, the SQL statement will become

UPDATE FROM *UserInfo*
SET *password =' testing', right =' admin'*
WHERE *login =' john'*

In the above injected statement, the system will alter *john*'s password to *testing,* together with setting his *right* to *admin.* So from now on, *john* becomes the administrator of the forum without the real administrator's approval.

3.4 Other Attacks

In the above examples, the damages of the above SQL Injection attacks are just limited to the database management system, like revealing and altering the data in the database. However, current database management systems provide some very advance functions, for example, accessing the system shell, reading and modifying the system data. As a result, if a hacker can discover which database is using, like Oracle or MSSQL, he can making use of these vendor-dependent functions to break into the whole system, not just the database management system.

Let's use the MSSQL Server as an example. In MSSQL Server, there are several functions like **EXEC master..xp_cmdshell** and **EXEC mas-**

ter..xp_regread which are used to provide a means for the database management system to communicate with the Windows to get a shell to execute some commands and to read the registry file respectively.

For example, in the following statement, the MSSQL Server will execute a command dir; which is used to list the files in the command shell.

EXEC *master..xp_cmdshell 'dir'*

Actually once a hacker can successfully break into the database system using the above attacks, he must leave a backdoor so that he can easily get into the system next time. The **EXEC master..xp_cmbshell** does help him a lot. Let's take a look at the following statements.

EXEC *master..xp_cmdshell 'netuserjohn/add'*
EXEC *master..sp_grantlogin 'TestComputer\john'*
EXEC *master..sp_addsrvrolemember 'TestComputer\john'*
,' sysadmin'

□ **End of chapter.**

Chapter 4

Current Defenses

Since SQL Injection can caused a lot of damages, including revealing and modifying people's information stored in the database, like the bank accounts, it has raised people's awareness to this problem. As a result, several solutions have been proposed to overcome this problem.

In order to understand how do these defenses work. We will first present what is the causes of SQL Injection attacks.

4.1 Causes of SQL Injection attacks

Figure 4.1 shows the whole database query process starting from the request from the client, which can be divided into three steps.

STEP 1 The application server first obtains the inputs from the clients, then constructs the corresponding SQL statements and sends them to the database server for the database operations.

STEP 2 The database server receives the SQL statements from the application server, then it will first analyze the correctness of the SQL statements. If

no error is found in the SQL statements, the database server will then analyze these statements to find out the operations which it needs to execute.

STEP 3 The database server operates in its storage according to the operations required in the SQL statements, like retrieve the information in the storage with certain conditions, or update the information.

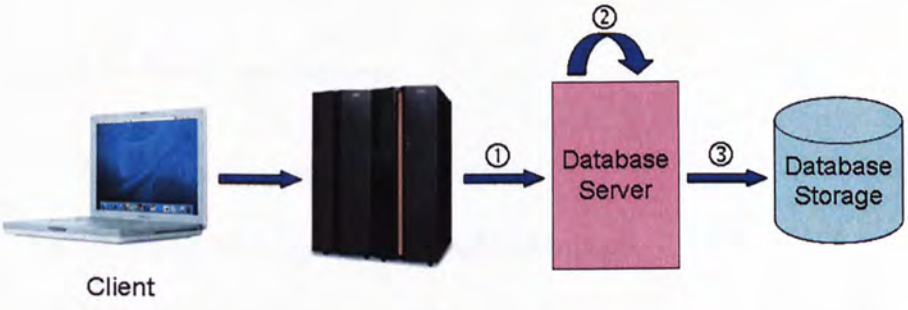

Figure 4.1: The three steps of the whole database query in the web applications

Actually the causes of SQL Injection attacks is due to incooperability between STEP 1 and STEP 2 – It is because the SQL Injection attack makes use of some "correct" SQL statements to retrieve and modify the information in the database illegally, and it is mainly caused by

1. The application server allows to process such injected inputs, and
2. The database server does not alert of the dangerous SQL statements constructed by these inputs.

4.2 Defense Methods

As mentioned in Section 4.1, SQL Injection is mainly caused in the application server stage and the analyzing phase in the database server. As

a consequence, current defense methods are mainly focused in these two places.

Focus in Application Servers

- Defensive Programming
- hiding the error messages
- Filtering out the dangerous characters

Focus in Database Servers

- Using pre-compiled SQL statements
- Checking for tautologies in SQL statements
- Instruction set randomization
- Building the query model

We will discuss the defense methods and its disadvantages one by one.

4.2.1 Defensive Programming

One of the most trivial defense methods is defensive programming – that is the developers should write their applications involving database operations more carefully by checking the input every time.

However, this defense method very difficult to achieve as we cannot ensure every developers write their code properly.

4.2.2 hiding the error messages

The hackers will encounter more difficulties in using SQL Injection to attack the system if the system is configured properly. By comparing the attacks in Section 3.2.3 and Section 3.2.4, we can find out that hiding the error messages to the users will increase the difficulties in attacking the system as the attackers need a lot more time to do the brute-force trial-and-error procedures to get the table and column names.

As a result, we will suggest all administrators to disable the function which can show the error messages to users in details. For example, in PHP, the administrators can change the default setting *display_errors = on* in the configuration file *php.ini* to *display_errors = off*.

This method, of course cannot solve the problem completely as it will only increase the difficulties for the hackers to successfully attack the system.

4.2.3 Filtering out the dangerous characters

One of the reasons that SQL Injection attacks succeed to attack the system is that there is no difference between the original ' in the SQL statements and the user inputted '. That is, if the application server is configured such that the user inputted ' is different from the original ', some of these attacks can be prevented. As a result, we will suggest all administrators to configure the application server to replace all ' retrieved from user input to \', which is the same for viewing purpose.

Using the example in Section 3.1 as an example, the original SQL statement is

SELECT $*$
FROM $UserInfo$
WHERE $login = '[user\ input\ login\ name]'$
AND $password = '[user\ input\ password]'$

If the application server is configurated to replace all ' to \', and the attacker inputs the login as *john* and password as *testing' AND '1'='1*, the login SQL statement will now become

SELECT $*$
FROM $UserInfo$
WHERE $login = 'john'$
AND $password =' testing\$$
AND $\$1\$ = \$1\$$

We have replaced all \' to \$ for the ease of understanding. Obviously the above SQL statements is not correct in syntax, and so the database server will return error message immediately instead of being injected originally. As a result, we will suggest all administrators to configure their application servers such that the character ' can be filtered out. For example, in PHP, the administrators can enable the function by setting *magic_quote_gpc = on* in the configuration file *php.ini*.

However, if the developers do not have the administrator privileges to configure the server settings, they can still use this features by manually checking the user input and replace all ' in the user input strings to \'.

Again, this method will only increase the difficulties for the attackers to attack the system, as the attack can try to avoid the use of '.

4.2.4 Using pre-complied SQL statements

As mentioned in Section 4.1, the application servers will construct the SQL statements from the user inputs for the database operations. However, both the user inputs and SQL statements are strings. That is, the database server cannot identify which parts of the SQL statements are come from the user input and which parts of the statements are the original statement skeleton.

So the attackers can pretend their inputs as part of the original skeleton to launch the SQL Injection attacks.

If the database server can accept some pre-complied SQL statements, that is the statements have already been converted to bytecode already, the database server will operate according to the pre-complied statements and the input from the application server, instead of just the SQL statements constructed by the application server. That is the database server can identify which parts are user inputs and which parts are original SQL statements. And since in this situation the database servers are not operate in the SQL statements in the form of string, the attackers will find difficulties in injecting the input to launch the SQL Injection attacks.

However, this features will only be supported by some commercial databases, like Oracle and DB2, which are very expensive.

4.2.5 Checking for tautologies in SQL statements

In [2], Wassermann and Su suggested to check for the existence of tautologies in the SQL statements to prevent SQL Injection attacks.

Tautology is some meaningless checking, which will always return true. For example, in the WHERE clauses of the SQL statements, the existence of $1=1$, $2>1$, $a>=b$ *OR* $b>=a$ are some examples of tautologies.

In normal cases, taxtologies should not exist in the SQL statements as taxtologies will have no use but just wasting the server resources for calculation. As a result, Wassermann and Su proposed the method using the Finite State Automata to check for the existence of tautologies in the SQL statements. If tautologies exist in the statements, the system is most probably under the SQL Injection attacks.

However, this method will only be applicable to those attacks involving the use of tautologies. That is, this method cannot prevent the attacks not using tautologies, like the attacks described in Section 3.2 and 3.3.

4.2.6 Instruction set randomization

In [3], Boyd and Keromytis suggested to use instruction-set randomization to defense the SQL Injection attacks. This system sits in front of the database server, like a proxy.

This system focus on the keywords used in SQL statements. First, both the clients and the system will share the same "key" which should not be known by other people. We will use the key **123** as an example. Suppose the client want to execute the following query.

$$\begin{array}{ll} \textbf{SELECT} & * \\ \textbf{FROM} & UserInfo \\ \textbf{WHERE} & login = \ 'robert' \\ \textbf{AND} & password =' \ 123456' \end{array}$$

Then the client will use some special keywords to replace the original **SELECT**, **FROM**, **WHERE** and **AND** in the above query, and the keyword is just to append the key at the end of each keyword. That is the following will be the actual query sent to the system from the client.

$$\begin{array}{ll} \textbf{SELECT123} & * \\ \textbf{FROM123} & UserInfo \\ \textbf{WHERE123} & login = \ 'robert' \\ \textbf{AND123} & password =' \ 123456' \end{array}$$

In the system side, when it receives the statement, it will remove the key in the statements and a correct SQL statement will be reconstructed and sent to the database server for query.

However, if an attacker uses an injected input (login as *john* and password as *testing' AND '1'='1*), the following SQL statement will be constructed in the client side and send to the system

SELECT123	*
FROM123	$UserInfo$
WHERE123	$login = 'john'$
AND123	$password =' testing'$
AND	$'1' =' 1'$

However, in the system side, since **AND** is not a correct keyword, the system will change the original **AND** to something which is not a keyword in SQL, like **ABC**. The reconstructed SQL statement will be

SELECT	*
FROM	$UserInfo$
WHERE	$login = 'john'$
AND	$password =' testing'$
ABC	$'1' =' 1'$

Obviously the above statement is not correct in syntax, and as a result an error will be returned from the database server.

Unfortunately, this method will not work if the key is known by the attackers. Suppose an attacker knows the key is 123, then in the above example, he will inject the password as password as *testing' AND123 '1'='1* instead of *testing' AND '1'='1*. Then the SQL statement constructed in the client sent to the system will be

SELECT123 $*$
FROM123 $UserInfo$
WHERE123 $login = 'john'$
AND123 $password =' testing'$
AND123 $'1' =' 1'$

So when the system receives the above query, the keyword will be replaced and so the final query will become

SELECT $*$
FROM $UserInfo$
WHERE $login = 'john'$
AND $password =' testing'$
AND $'1' =' 1'$

and since the statement is correct in syntax, the database server will return the corresponding result to the attacker. So the system fails to defense the SQL Injection attacks. Actually the hacker can use the trial-and-error approach to get the key.

4.2.7 Building the query model

In [4], Halfond and Orso presented the idea to build the query model for each query in the web applications to defense the SQL Injection attacks.

Their idea is first identify the hotspots – that is the statements containing database operations in the web applications. Then they will build the model for each query. For example, if the original query is

SELECT $*$
FROM $UserInfo$
WHERE $login = \text{'}[user\ input\ login\ name]\text{'}$
AND $password = \text{'}\ [user\ input\ password]\text{'}$

The corresponding SQL model, which is constructed from the SQL keywords, and other strings will be constructed like the following

$SELECT \rightarrow * \rightarrow FROM \rightarrow UserInfo \rightarrow WHERE \rightarrow login \rightarrow = \rightarrow \text{'} \rightarrow$ **VAR** $\rightarrow \text{'} \rightarrow AND \rightarrow password \rightarrow = \rightarrow \text{'} \rightarrow$ **VAR** $\rightarrow \text{'}$

In the above model, **VAR** is any string not containing the SQL keywords and special characters including '. For example, if a normal statement is constructed from a honest user like

SELECT $*$
FROM $UserInfo$
WHERE $login = \text{'}robert\text{'}$
AND $password = \text{'}\ 123456\text{'}$

In the runtime, a runtime monitor in the web application will first the query to see if it matches the model constructed before. If these two things match, the runtime monitor will forward the query to the database server for database operations. Otherwise, if these two things do not match with each other, the runtime monitor will stop the query from forwarding to database server for database operations.

In the above example, the runtime monitor will check the above statement with the model to see if the query violates the model. Obviously the above statement matches the model and so the runtime monitor allows the query to be forwarded to the database server.

However, if a hacker constructed the following query using some injected input

SELECT $*$
FROM $UserInfo$
WHERE $login = 'john'$
AND $password =' testing'$
AND $'1' =' 1'$

Once the runtime monitor checks the above query, it will discover that the query does not match with the query model as the above query contains 2 **AND** will is not allowed in the model. As a result, the runtime monitor will not forward the query to database server for further operations and errors will be returned from the monitor to identify that it is under SQL Injection attacks.

However, in [4], Halfond and Orso has also mentioned that the above method may not be successful in medium or large programs due to the scalability reasons. Besides, in constructing the query model, this method needs to identify the SQL keywords, which is a little bit difficult as different database may have different sets of SQL keywords. So different systems may be constructed for different database servers.

□ **End of chapter.**

Chapter 5

Proposed Solution

5.1 Introduction

In Chapter 4, we have presented the current methods to defense SQL Injection. However, those methods are quite passive – that is they will only defense the current well-known SQL Injection attacks. However, if there are some new attacks, they may not be able to defense them.

As a consequence, we have proposed a new defense system, which can not only detect the current SQL Injection attacks, but also can learn for new SQL Injection attacks.

5.2 Natures of SQL Injection

Before understanding the details of the system, we would like to give some basic ideas of the nature of SQL Injection attacks.

Due to the nature of SQL, the user input will always be part of the query in the **WHERE** clauses. That is, the SQL Injection attacks will always exist after the keyword **WHERE**.

5.3 Our proposed system

For our proposed system, it will be similar to a SQL proxy, that is it will be placed between the SQL server and the application server. That is, all SQL queries will be first passed to the proposed system instead of directly sent to the SQL server. The overall system flow is shown in Figure 5.1.

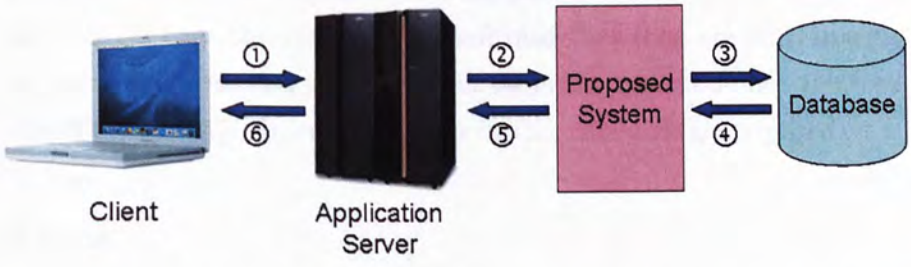

Figure 5.1: The overall working flow of the proposed system

Actually we can divide the process of the system into 5 stages.

Stage 1	Checking with current signatures
Stage 2	SQL Server Query
Stage 3	Error Triggering
Stage 4	Alarm
Stage 5	Learning

5.3.1 Features of the system

The feature of this system is its analyzing and learning functions. Similar to other intrusion detection system, they are the most complicated functions to implement.

However, in this proposed system, we make use of the nature of SQL Injection mentioned in Section 5.2, so that the system will just need to analyze the latter part of the query instead of the whole one. So the complexity of the system has been reduced a lot.

5.3.2 Stage 1 – Checking with current signatures

Purpose

The purpose of this stage is to check the query statement against current signatures. There are 2 levels for signatures, the first level is the signatures is **confirmed** and the second level is **suspected**. The signatures in confirmed state mean that the system has confirmed that they are SQL Injection attacks, while those in suspected state mean the system do not 100% confirm that they are SQL Injection and so further analyses are required.

Action

If the system finds that the query exists in the signature database, it will deny the access from that clients, and record all the information needed, including the address of the clients. If the signature is a **suspected** one, the system will then go to Stage 5 directly to further analyze the signature.

Otherwise, if the query doesn't match the signatures in the database, the system will go to Stage 2 to continue the process.

5.3.3 Stage 2 – SQL Server Query

Purpose

The purpose of this stage is to pass the query to the SQL server for actual query, then the system will obtain the result for further process.

Action

If the result returned from the SQL server is not an error, the system will return the result to the clients, and the system will finish its works. However, if an error is returned by the SQL server, the system will continue the process by going to Stage 3.

5.3.4 Stage 3 – Error Triggering

Purpose

The purpose of this stage is to make the system alert to coming SQL Injection attacks.

In normal cases, the SQL server will not return error in the runtime because the applications must be properly debugged before releasing to the public. However, as mentioned in Chapter 3, the attackers must start launching the SQL Injection by trail-and-error approach. They will get information from the error messages, like the name of table and columns and number of rows in the table. That is nearly all the time the input of the attackers will cause the SQL server to return errors as the SQL statement are invalid. As a result, if an error is returned by the SQL server, it is a sign that some attackers are launching SQL Injection to the system.

Action

If the system receives the error returned from the SQL server, it will start the alert system – that is the client may be launching the SQL Injection attack. Then the system will record the client's information, including the IP address, and the SQL statement.

At the same time, the system will return an error message to the client saying that there are some problems in the server. However, we will hidden all the details so that the attackers cannot find any useful information from these error messages.

Criteria go to Stage 4

There are 2 cases that the system will go to Stage 4.

Case 1 The first case is that the system will have a threshold value n, if within a certain period of time t, says 5 minutes, the system finds out that the queries from a particular user generated n errors to the SQL server, the system will decide that this user is launching SQL Injection to the system, and so the system will go to Stage 4.

If within t, the number of errors is less than n, the system will not consider these errors as SQL Injection attacks. However, some attackers may try to inject some "junk" input between the actual injections – that is they will try to enter some normal inputs in between the injections in order to avoid reaching the threshold value n with the time period t.

As a result, in order to avoid the above situation, the value t will be increased exponentially after the end of each period. For example, if an attacker starts to inject, and error is returned, the system will then start triggering with the time period t. Then the attacker starts entering the "junk" input within this time period and as a result the criteria is not met. Then at the end of this time period, the system will increase the value of t exponentially, says $2t$. As a result, if attacker produces n errors within the period of time $2t$, the system will go to Stage 4.

This practice will at least slow down the attacker's action a lot.

Case 2 The second case is studying the pattern of input. As mentioned in Chapter 3, the attackers will launch the SQL Injection by trial-and-error approach. However, most of the time, this approach will have some pattern. Let's use Section 3.2.2 as an example. In the beginning, the attacker will input *abc' UNION SELECT 1,1* –, and the SQL statement constructed will be

SELECT $*$
FROM *Message*
WHERE *login* = *'abc'*
UNION
SELECT $1,1--$

Since it is not an valid query, an error will be returned by the SQL server, and then the error will be forwarded to the client. So the attacker will continue increasing the number of **,1** in the input, that is the second one will be *SELECT 1,1,1–*, that is the query constructed will be

SELECT $*$
FROM *Message*
WHERE *login* = *'abc'*
UNION
SELECT $1,1,1--$

If error continues occur, he will continue increasing the number of **,1** again. Therefore the input will become *SELECT 1,1,1,1–* and the constructed query will be

SELECT $*$
FROM *Message*
WHERE *login* = *'abc'*
UNION
SELECT $1,1,1,1--$

Once the system can collect these 3 queries, it is very easy to find out the difference between the first two queries, Δ_1 and the last two queries, Δ_2,

and also the location of the differences. Actually it can be easily achieved even by using the *diff* function in UNIX system.

In order to study the pattern accurately, the system will convert all numbers to **1** and all characters to **a** in the difference. Therefore, the system will find out the difference Δ_1, between these two inputs – this current one has one more ***,1*** and is located before $--$ in the input. The same result will be discovered for Δ_2.

After the system obtains the differences, it will then compare and find out the trend of attacks. The system will make the decision that the queries are SQL Injection attacks to the system if

1. $\Delta_1 = \Delta_2$, where both Δ_1 and Δ_2 locate nearly at the same position, or
2. Δ_1 includes Δ_2, or vice versa, where Δ_1 and Δ_2 locate nearly at the same position.
3. Either Δ_1 or Δ_2 is a **zero difference**. That is, there are no difference between two consecutive queries.

Actually the condition 1 and 2 above are used to detect the trial-and-error attacks in guessing the number and the name of columns in the tables. For the condition 3, it is particularly used for detecting the trial-and-error attacks in guessing the name of columns.

Then the system will notice that $\Delta_1 = \Delta_2$, and both of them are located just before $--$, the system will make the decision that the probability that these queries are SQL Injection attacks are very large, and so the system will go to Stage 4.

5.3.5 Stage 4 – Alarm

Purpose

The purpose of this stage is to tell the system administrator that the system in under SQL Injection attacks.

Action

If the system has made the decision that it is now under the SQL Injection attacks, it will first terminate all the connections from the attacker, and will not allow any connections made from the attacker. This practice is to prevent the system from further attacks.

5.3.6 Stage 5 – Learning

Purpose

The purpose of this stage is to learn the new SQL Injection signature for future filtering use.

Action

There are 2 different cases for Stage 5.

Case 1 If the error comes from case 2 in Stage 3 mentioned in Section 5.3.4, the function of this stage is just put the Δ_1 or Δ_2 to the confirmed signature. It is because in Stage 3 the system has already analyzed the pattern, the system needs not do it again in this stage to increase the performance.

Case 2 However, if the error does not come from case 2 in Stage 3, it means the system does not have enough information to analyze the attack pattern. As a consequence, we will try to find out the signature of these queries, and place them into the suspected signature at the first time. So later if the same signatures are discovered, the system can confirm that these queries are SQL Injection attacks to the system. Then the system will move these signatures from the suspected signature database to the confirmed signature database.

5.4 Examples

In this section, we will use the examples presented in Chapter 3 to present how the proposed system works to defense the current SQL Injection attacks.

As mentioned in Section 5.3.2, the system must have a signature database. In this section, we will assume that at the beginning, the signature database contains 10 signatures, which are

- **OR '1' = '1'**
- **OR 1 = 1**
- **OR 'a' = 'a'**
- **OR '1' = 'a'**
- **OR 'a' = '1'**
- **AND '1' = '1'**
- **AND 1 = 1**
- **AND 'a' = 'a'**
- **AND '1' = 'a'**
- **AND 'a' = '1'**

For the ease of understanding, we use **1** to represent the numerical characters (0-9) and **a** to represent the alphabetic characters (a-z and A-Z).

Actually the above 10 signatures are very similar, they are just a logical test to the database system. The purposes of the above signatures are

1. To test if the system suffers from SQL Injection attacks, and
2. To reveal all the information in the table (the first, second and third signature)

5.4.1 Defensing BASIC SELECT Injection

In Section 3.1, we have mentioned the following SQL statement to reveal all information in the table.

SELECT $*$
FROM $UserInfo$
WHERE $login = 'abc'$
AND $password =' abc'$
OR $'1' =' 1'$

This kind of injected statements should contain something like *OR '1'='1'* or *OR 1=1* to reveal all information stored in the table. However, the statement should not be able to pass the Stage 1 in the defense system as these signatures should be exist in the signature database. As a result, the system is able to defense the BASIC SELECT Injection.

5.4.2 Defensing Advanced SELECT Injection

Defensing the guessing of number of columns

As stated in Section 3.2.2, before the attackers can successfully launch the SQL Injection attacks, they should be able to know the number of columns

first. As a result, they will use the trial-and-error approach to guess the number of columns. In the example in Section 3.2.2, they need to inject the statements containing *SELECT 1,1–*, *SELECT 1,1,1–* and *SELECT 1,1,1,1–* continuously to guess the number of columns.

However, once the attackers inject the statement containing *SELECT 1,1–* to the system, the SQL statement constructed becomes

SELECT $*$
FROM *Message*
WHERE $login = 'abc'$
UNION
SELECT $1, 1 - -$

It will not be able to pass the Stage 3 of the system because the database server returns errors. As a result, the attacker needs to continue the trial-and-error process to guess the number of columns in the table. As a result, the following constructed statements will be input to the database server.

SELECT $*$
FROM *Message*
WHERE $login = 'abc'$
UNION
SELECT $1, 1, 1 - -$

SELECT $*$
FROM *Message*
WHERE $login = 'abc'$
UNION
SELECT $1, 1, 1, 1 - -$

After these 3 statements have been collected by the system, the system will realize that the difference between the first 2 statements, Δ_1, and the difference between the last 2 statements, Δ_2, are the same. As a result, it fulfills the case 2 criteria in Section 5.3.4 to go further to Stage 4 and 5 mentioned in Section 5.3.5 and 5.3.6 respectively. After the system reaches Stage 5, it will process the learn the syntax. Since in Stage 3, the difference has already been recognized, the system will just place the difference, that is *1, (located before –)* will be placed in the confirmed signature database. As a result, this kind of Advanced SELECT Injection to guess the number of columns has been defended.

Defensing the guessing of name of tables and columns

Besides the guessing the number of columns, there are other injection attacks to guess the name of tables and columns as stated in Section 3.2.3 and 3.2.4. So we will demonstrate how the system can defense this attack.

For the attack demonstrated in Section 3.2.3, the criteria to make the attack success is that the server is configurated poorly so that the error message will be displayed to attackers directly so that they can get the table and column names easily.

In the proposed system, however, the error message will be hidden to attackers. As a result, the attack demonstrated in Section 3.2.3 can be successfully defended.

For the attack described in Section 3.2.4, we will use the examples in that section to demonstrate how the proposed system works to defense the attack.

In the example, the table to store the user information is *UserInfo*. Suppose the attack use the input *abc' UNION SELECT 1,1,1,1 FROM User–* to attack the system, where *User* is the table name guessed by the attack, which is not exist in the database. And the constructed SQL statement is

SELECT $*$
FROM $Message$
WHERE $login = 'abc'$
UNION
SELECT $1, 1, 1, 1$
FROM $User --$

Since *User* does not exist, error will be returned. As a result, this statement cannot pass Stage 3. So the attacker will continue using the trail-and-error approach to get the table name. Suppose he continues using *UserDB* and *UserDataBase* as the guessed table name. The constructed SQL statements will become

SELECT $*$
FROM $Message$
WHERE $login = 'abc'$
UNION
SELECT $1, 1, 1, 1$
FROM $UserDB --$

and

SELECT $*$
FROM $Message$
WHERE $login = 'abc'$
UNION
SELECT $1, 1, 1, 1$
FROM $UserDataBase --$

After these 3 statements have been collected, the difference between the first 2 statements Δ_1, which is *aa (located before -)* and the last 2 statements Δ_2, which is *aaaaaaa (located before -)* has been found. Obviously Δ_1 is part of Δ_2, so it fulfills the requirement, and as a result the system will go to Stage 4. And the difference Δ_1 will be placed in the confirmed signature database in Stage 5.

The case for defensing the guessing the column name is very similar. Suppose the attacker gets the table name, *UserInfo*. And he use the *name* as the guessed column name, while the correct one is *loginname*. The constructed SQL statement is

SELECT $*$
FROM $Message$
WHERE $login = 'abc'$
UNION
SELECT $name, 1, 1, 1$
FROM $UserInfo--$

Again, the statement will cause the database to return error and as a result it cannot pass the Stage 3 of the proposed system. So the attackers will continue entering *login* and *uname* as the guessed column names. So the constructed SQL statements are

SELECT $*$
FROM $Message$
WHERE $login = 'abc'$
UNION
SELECT $login, 1, 1, 1$
FROM $UserInfo--$

and

SELECT $*$
FROM $Message$
WHERE $login = 'abc'$
UNION
SELECT $uname, 1, 1, 1$
FROM $UserInfo--$

So the difference for first 2 statements, Δ_1 is *a (located between second SELECT and ,1)*. However, the difference for the last 2 statements is *a zero difference (located between second SELECT and ,1)*. According to the criteria stated in Section 5.3.4, the system can confirm that these inputs are injection attacks, and the system will continue the process for learning. As a result, the proposed system can defense the Advanced SQL Injection to guess the table and column names.

5.4.3 Defensing UPDATE Injection

Besides the SELECT Injection mentioned in Section 3.1 and 3.2, our proposed system can also defense the UPDATE Injection attacks mentioned in Section 3.3. We will use the examples mentioned in Section 3.3 to describe how the proposed system works to defense the UPDATE Injection.

Similar to the Advanced SELECT Injection attacks described in 3.2, the attackers must be able to get the name of the columns before launching this attacks. Obviously, they must use the trial-and-error approach to guess the name of columns like that in Section 5.4.2.

Suppose in the example in Section3.3, the column name the attackers want to modify is *right*. However, the first guess from an attacker is *group*, which is not exist in the table. That is, the SQL statement constructed from the injected input is

UPDATE FROM	$UserInfo$
SET	$password =' testing', group =' admin'$
WHERE	$login =' john'$

And obviously this statement cannot pass through Stage 3 of the proposed system as error is returned from the database system. Then suppose the attackers continue guessing the column name as *power* and *usergroup*. Then the constructed SQL statements will be

UPDATE FROM	$UserInfo$
SET	$password =' testing', user =' admin'$
WHERE	$login =' john'$

and

UPDATE FROM	$UserInfo$
SET	$password =' testing', power =' admin'$
WHERE	$login =' john'$

respectively. Again, once these 3 statements have been collected, the difference between the first two statements Δ_1, which is *aaaaa (located after , in the SET clause)* and the difference between the last 2 statements, which is *aaaaa (located after , in the SET clause)* has been determined. Obviously $\Delta_1 = \Delta_2$, so the system will determine the previous inputs are SQL Injection attacks to the system and as a result it will go the later stage to learn and store the pattern in the confirmed signature database. As a result, the UPDATE Injection attacks can be defended by the proposed system.

5.5 Comparison

In this section, we would like to show the advantages of our proposed system to the current defense methods.

Comparison with Defensive Programming, hiding the error messages and Filtering out the dangerous characters

For the defensive programming method mentioned in Section 4.2.1, it is just a passive defensive method to avoid the system being able to be attacked by the attackers.

For the hiding error messages method in Section 4.2.2 and the filtering out dangerous characters in Section 4.2.3, they are just used to increase the difficulties for the attackers to be able to successfully attack the system.

Actually, the three methods mentioned above do not solve the problem completely. As a result, we cannot direct compare these three methods with our proposed system, as our system is designed to defense the problem while these three methods are to prevent.

Comparison with pre-complied SQL statements

The method mentioned in Section 4.2.4 is to avoid the use of normal SQL statements, by changing them to pre-complied ones, to differentiate between the original statements and the user input.

However, this method is only available in some large-scale commercial database servers like Oracle and IBM DB2, but not available for some small-scale servers like MySQL and PostgreSQL.

The advantage of our proposed system to this method is our system is vendor-independent. It is because our system is located in between the application servers and the database servers, it is not important of what

database server is used as we just need to forward the request and receive the response from the database server. As a result, our proposed system can not only deploy to the large-scale commercial database servers, but also the small-scale ones.

Comparison with tautologies checking

The defense method stated in Section 4.2.5 is mainly focused to check for the existence of tautologies in the SQL statements, which should rarely occur in normal SQL queries from the application servers.

However, as mentioned in the same section, this defense method can only be applicable to the attacks involving the use of tautologies. As a result, this method may not be able to defense the attack not using tautologies, like the attacks described in Section 3.2 and 3.3.

But as discussed in Section 5.4.2 and Section 5.4.3, our proposed system can be used to defense the attacks not involving the usage of tautologies.

Comparison with Instruction set randomization

The working principle of the method stated in Section 4.2.6 is to add a secret key which is only shared by the client and the database server to those SQL keywords, like **SELECT**, **FROM**, **WHERE**, **AND**, **OR** ... etc.

However, since different vendors may have different keywords in their database servers, like **EXEC** in MSSQL. As a result, this method is not platform-independent. Besides, the attackers can still have the chance to guess the common key by trial-and-error approach.

In our proposed system, we are actually waiting for attackers' trial-and-error actions to detect and learn their attack pattern. As a result, we will not encounter the same problem. Besides, our proposed system is platform-independent, while the tautologies checking method is platform-dependent.

As a result, our proposed system is a little bit better than that method as the we just need to develop once so that we can defense the problems in any database servers.

Comparison with query model building method

The working principle of the method discussed in the section 4.2.7 is to build the query model for every database query in the web applications. If the SQL statements constructed using the user inputs do not match with the model, the application will terminate the process to prevent the attacks.

However, the writers in [4] also mentioned that this method may not be able to work in medium and large programs due to the scalability reasons. Besides, as the method needs to identify the SQL keywords, this method may not be platform-indepednet as different database servers from different vendors may have different sets of SQL keywords.

Our proposed system works in two different phases, identifying the signatures and learning the new attack patterns. Actually there are similar technologies in these two areas which are believed to be efficient. As a result, we believed that the proposed system should work in a good performance. Besides, our solution is platform-independent, which is a little bit better than the model building methods, which is not fully platform-independent.

□ **End of chapter.**

Chapter 6

Conclusion

This thesis focused in an arising problem exists between the web applications and the database servers – SQL Injection attacks. SQL Injection attack is a kind of code-injection attacks, in which the attackers will input some specially-designed strings to the web applications. And the web applications will construct the SQL statements from the user input. Then these attackers specially-designed inputs will become part of the code in the query statements. Using this query, the attackers can retrieve and modify all information stored in the database server, including some sensitive information like the customer personal information, and all users' username and password pairs.

In Chapter 1, we have used a story to describe the problems which may be caused by SQL Injection attacks. Besides, we have made use of some statistics and real-world examples to show the importance of SQL Injection attacks.

Chapter 2 has been used to provide some background information for the readers. We have present some basic knowledge like the structure of a database management system together with the syntax of some basic SQL statements like SELECT, INSERT, UPDATE and DELETE statements, which will be frequently used in SQL Injection attacks.

Chapter 3 has described the details of this thesis, SQL Injection attacks. We have made use of some examples to describe how do the SQL Injection attacks, including the Basic and Advanced SELECT Injection, together with the UPDATE Injection work.

In Chapter 4, we have first presented the main cause of the SQL Injection attacks, which is the incooperability between the application servers and the database servers – the application server does not validate the SQL statements constructed before passing to the database servers for database operations, while the database servers do not aware of these dangerous statements constructed by the attackers. Then we have described the current defense methods to SQL Injection attacks. The defense methods can be divided into two categories, which are either focusing in the application server or focusing in the database server. And we have presented the working principles, together with some weaknesses of these methods.

Chapter 5 has been used to present our proposed system, which is a system which can detect the current-known SQL Injection attacks, and at the same time, learn the newly-discovered attacks. We have divided the system into 5 stages, and we have stated the purposes and the working principles of each stage. Finally, we have made use of some examples to demonstrate how does our proposed system work to defense the current-known SQL Injection attacks and learn the newly-discovered attacks at the same time.

□ **End of chapter.**

Appendix A

Commonly used table and column names

Actually, there are several hacking tools available in the Internet [5, 6] to launch the SQL Injection attacks automatically. One interesting thing is that most of them will have a dictionary file storing the commonly used table and column names for their trial-and-error approach. As a result, we would like to present these commonly used names.

A.1 Commonly used table names for system management

- admin
- manage
- a_admin
- x_admin
- m_admin
- password
- admin_userinfo
- clubconfig
- userinfo

- config
- company
- book
- adminuser
- article_admin
- art
- user
- bbs
- giat
- member
- members
- userlist
- memberlist
- yonghu
- admin_user
- list
- users
- info

A.2 Commonly used column names for password storage

- userpass
- password
- pass
- pwd
- pword
- adminpassword
- adminpass
- user_pass
- admin_password
- user_password
- user_pwd

- adminpwd
- dw
- pws
- admin_pass
- admin_password
- passwd

A.3 Commonly used column names for username storage

- username
- user
- name
- u_name
- administrators
- userid
- adminuser
- adminname
- user_name
- admin_name
- usr_n
- usr
- nc
- uid
- admin
- admin_user
- admin_username
- user_admin
- adminusername

□ **End of chapter.**

Bibliography

[1] One, A.: Smashing the stack for fun and profit, http://www.phrack.org/phrack/49/p49-14 (1996)

[2] Wassermann, G., Su, Z.: An analysis framework for security in web applications. In: Proceedings of the FSE Workshop on Specification and Vertification of Component-Based Systems (SAVCBS 2004). (2004)

[3] Boyd, S., Keromytis, A.: SQLrand: Preventing SQL Injection Attacks. In: Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference. (2004)

[4] Halfond, W.G., Orso, A.: Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In: Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005), St. Louis, MO, USA (2005)

[5] http://www.netxeyes.com/main.html: Web entry detector (2004)

[6] http://www.ayxz.com/soft/5632.htm: 絕世好猜 1.0(2004)

[7] Raghu Ramakrishnan, J.G.: Database Management Systems. McGraw-Hill Highber Education (2000)

[8] http://www.securiteam.com/securityreviews/5DP0N1P76E.html: Sql injection walkthrough (2002)

[9] Maor, O., Shulman, A.: Sql injection signatures evasion. Technical report, iMPERVA (April 2004)

[10] Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: WWW '04: Proceedings of the 13th international conference on World Wide Web, New York, NY, USA, ACM Press (2004) 40–52

[11] Huang, Y.W., Huang, S.K., Lin, T.P., Tsai, C.H.: Web application security assessment by fault injection and behavior monitoring. In: WWW '03: Proceedings of the 12th international conference on World Wide Web, New York, NY, USA, ACM Press (2003) 148–159

[12] McDonald, S.: Sql injection: Modes of attack, defence, and why it matters http://www.governmentsecurity.org/articles/sqlinjectionmodesofattackdefenceandwhyitmatters.php (2004)

[13] Anley, C.: Advanced sql injection in sql server applications. Technical report, NGSSoftware Insight Security Research (2002)