SQL Injection Prevention System

User Manual

Ovak Technologies
2016

Contents

1. Introduction

1.1. Definitions and Acronyms

- **SQL (Structured Query Language)**

SQL is the high-level language used in numerous relational database management systems. SQL is a declarative computer language which has elements which include clauses, expressions, predicates, queries, and statements.

What makes SQL so powerful is its immense flexibility and its ability to be abstract. It allows a human being to use SQL to ask for what information he wants without outlining how the information is to be retrieved. Thus, this relieves the user of any programming knowledge needed to satisfy the query. In this sense, it is an even higher "high level" language than most traditional programming languages such as C++ and Java. A person with little or no programming background can still use SQL effectively. However, SQL includes powerful features and functions that allow users with programming knowledge to build complex queries and apply them to even more powerful uses.

- **SQL Injection**

SQL injection is a code injection technique, used to attack data-driven applications, in which nefarious SQL statements are inserted into an entry field for execution.

Web programmers often take string values entered by an Internet user on a form that represents user names and passwords and place them directly into the SQL statement to be run against a database. A simple test SQL statement that may be used is the following example.

```
SELECT username, password
FROM UserAuth
WHERE username = 'usernameFromForm'
AND password = 'passwordFromFrom';
```

In this example, the values *usernameFromForm* and *passwordFromForm* are the literal values obtained from the form. The intent is using the username and password obtained from the form to see if there is a matching username and password in the *UserAuth* table. If any rows are returned, the user is authenticated. However, if the web programmer is not careful and uses this method and the form values without checking them, a hacker may instead pass arbitrary values that the programmer did not originally anticipate.

- **AND/OR Attack**

One such attack is the basic attack that involves the *AND* or *OR* logic in the SQL predicate. The hacker can specify a valid username such as "John Doe" and then specify the password as *"' OR '1'='1"* in the form. The final test SQL query that uses these values will be:

```
SELECT username, password
FROM UserAuth
WHERE username = 'John Doe'
AND password = '' OR '1'='1';
```

Provided that "John Doe" is a valid user, the database will allow the hacker to log-in and proceed as "John Doe", because even though the password string is not empty (the first predicate), *'1'='1'* is a valid predicate that will always return *TRUE*. Thus, the hacker has just accessed the account without ever knowing the password, and now he/she has full access to the victim's information. The hacker does not need to know this is the way the form data is used to construct the SQL statement; he/she just simply needs to do several "probing" tests and see the messages returned to see if this is indeed the case. If the attack does not succeed, the attacker simply moves on and tries another method. If it succeeds, the DBMS will happily return the

username and corresponding password; our hacker now has unauthorized access to the database through that username.

- **Comments Attack**

SQL allows inline commenting within the SQL "code". This allows two variations of SQL-I comments attacks. One simple variation is assigning username to be a valid username followed by comment characters. For example, we assign username = *"admin' --"*. Then our SQL test query may look like the following:

SELECT username, password
FROM UserAuth
WHERE username = 'admin' --' AND password = 'anything';

Everything after the *"--"* in the WHERE clause will be ignored, so this will allow the hacker to log in as "admin". This is a method of using comments as a way of ignoring the rest of the query.

The variation of the comments attack is using comments as a way of obfuscating the signature of any SQL-I attack to avoid detection. Therefore, the use of C-style comments *"/*"* and *"*/"* can be combined with any of the previously discussed attacks as a way of attempting to circumvent signature-based detection. For example, if an application searches a string passed from a form for the *UNION* keyword to attempt to catch *UNION* injection attacks (discussed in more detail in a subsequent section), an attacker may choose to use comments to conceal this. For example, instead of using *"UNION ALL"*, the attacker may instead use *'UNION /**/ALL' or 'UN/**/ION A/**/LL'*.

Both of these are synonymous with "UNION ALL" in the context of an SQL statement. In addition to breaking up keywords, comments may be used in place of spaces.

A system using signature-based detection may miss keywords and SQL-I patterns if it is not careful to also consider SQL-I Comments attacks as well.

- **String Concatenation Attack**

SQL has an option to concatenate separate strings or characters to form complete strings. This is accomplished using + or "double pipe" *(||)*, or the function *CONCAT* (such as in MySQL). These operations can be used to create a variation of the *UNION* Injection attack by obfuscating the UNION keyword in a string concatenation operation.

For example, an attacker may use *'UNI' + 'ON A' + 'LL'* in place of *"UNION ALL"* if he suspects the system looks for the *UNION ALL* keyword.

Another use of string concatenation in an attack is when the attacker suspects the system searches for single quotes *(')*. Then the attack may choose to use the *CHAR()* function in conjunction with the string concatenation to issue characters indirectly without using any single quotes. For example, an attack may use

CONCAT(CHAR(65),CHAR(68),CHAR(77),CHAR(73),CHAR(78))

to represent 'ADMIN' so that the system will not find a single quote if it was looking for them.

- **UNION Injection Attack**

The UNION Injection attack may be the most dangerous, but certainly the most surprising of the SQL-I attacks. This is because if it is successful, the *UNION* Injection attack allows the attacker to return records from another table. For example, an attack may modify the SQL query statement that selects from the user authentication table to select another table such as the accounts table.

SELECT username, password FROM userAuth
UNION ALL
SELECT accountNum, balance FROM Accounts

The use of *UNION ALL* in this attack allows the attacker access to tables that the SQL query statement was not originally designed for. The resulting rows selected from both tables will appear on the resulting page.

The trickiness in this attack lies in the fact that the columns selected from the second table must be compatible in number (the same number of columns as the original table must be selected) and type. When trying to guess the correct number of columns, the attack may simply keep trying to use different number of columns in each attempt until he finds the right number. To match the type, the attack may try to try different types until he stumbles upon the right one or he may simply choose to use *NULL* instead. The *IDPS* system discussed later does not return any messages such as response or *HTTP* status codes and limits internal information being broadcast externally as much as possible.

- **Hexadecimal/Decimal/Binary Variation Attack**

Attackers can further try to take advantage of the diversity of the SQL language by using hexadecimal or decimal representations of the keywords instead of the regular strings and characters of the injection text. For example, instead of using the traditional SQL-I Attack text:

1 UNION
SELECT ALL
FROM WHERE

an attacker may substitute this with

1 UNION SELECT ALL FROM WHERE

to attempt to avoid detection by signature-based detection engines.
The system that does not look for hexadecimal or decimal characters will be susceptible to this variation of the SQL-I attack.

- **White Space Manipulation Attack**

Signature-based detection is an effective way of detecting SQL-I attacks. Modern systems have the capacity to detect a varying number of white spaces around the injection code, some only detect one or more spaces; they may overlook patterns where there are no spaces in between. For example, the SQL-I pattern ***' or 'a' <> 'b*** can be re-written as ***'or'a'<>'b***, containing no spaces in between. A DBMS SQL parser will be able to handle a variable around all of white space characters or keywords. If a signature-based detection method only takes into account the first pattern, it will completely overlook the second one.

In additional to the standard space character, white space characters also include the tab, carriage return, and line feed characters. To properly implement signature-based detection, the system must be able to handle white space characters.

- **SQL** – Structured Query Language
- **IPS** – Injection Prevention System
- **DBMS** – Database Management System
- **PII** – Personally Identifiable Information
- **ASCII** – American Standard of Code for Information Interchange

1.2. Overview

The Internet is a huge interconnected network, the largest in the world. In less than two decades, it has grown from an esoteric academic medium to a ubiquitous source of information. A great deal has been made about the accessibility of the Internet as well as how it will supposedly change all of our lives. Computer files are replacing paper files as electronic records in institutions such as hospitals, insurance providers, and banks are quickly replacing their carbon-based counterparts. As they are getting used to the idea of using it, people are going shopping and banking over the Internet. People are demanding more and more access to the Internet – always wanting the information faster, more constantly available, and increasingly diverse in content. Insurance providers are starting to encourage employers, physicians, and insurance brokers to submit medical claims information electronically in order to cut down on costs and improve turnaround time.

With the growth of digital traffic, we are affording ourselves great convenience, but also exposing ourselves to greater risk of having sensitive information intercepted or stolen. Systems that contain sensitive information must have safeguards to prevent the risk of external attacks on the system, for example with the use of some type of tool. We discuss the development of a software tool in this paper.

Personally identifiable information (PII) can be found in bank accounts, retirement or investment accounts, and credit card accounts. The volume of this information is often great as a result of institutions having so many customers and users. Therefore, the ideal way to store and retrieve all of this information is in a database. To satisfy people's desire to access information from this database from anywhere at anytime, the natural network of choice is the Internet. The feature of having such convenient access is precisely how we run into security issues. The Internet was first created without security in mind, because it was not an issue at the time. It was a simple, open way of communicating and sharing ideas in the academic setting. The modern version of the Internet, however, is accessible by anybody including those with dubious moral values. There needs to be protective of the PII in these databases. People want their names, addresses, phone numbers, credit card numbers, and social security numbers to be

private and protected. Now, more than ever, we need to strengthen the security of the systems which use these databases and make them secure.

One of the techniques available to the would-be information thieves is SQL-Injection (SQL-I). SQL-I attacks involve a variety of methods, but the intention of an attacker using it is to submit specially-chosen patterns when asked for the user's username and password on an internet form. The values passed from a form may be directly concatenated into a string with an SQL query string. Then the resulting SQL query string is dynamically parsed in order to test, for example, if the username and password are correct. When the text from the user input is unchecked and incorporated into the SQL query, these abnormal values can result in highly abnormal behavior such as gaining access to the system despite not supplying the correct password, selecting columns from completely different tables than the ones being queried in the original query, or even highly dangerous behavior such as deleting any tables. It is therefore imperative that these values are be checked before being submitted to DBMS parser to be run on the database.

The purpose of this toolkit is to detect some types of SQL injection. It is implemented with the signature-based method, i.e. it detects specific types of injections.

These special types are the following:

1. AND/OR Attack
2. White Space Manipulation Attack
3. Hexademical/Decimal/Binary Variation Attack
4. Comments Attack
5. String Concatenation Attack
6. UNION Injection Attack

The protection against the first three types of attacks is implemented in this program.

1.3. Purpose

The purpose of this document is to provide necessary information to LabVIEW developers on how to use the toolkit.

2. Injection Prevention System Design

2.1. Fundamentals

IPS enables to protect database from injection attacks. It detects and prevents intrusion attempts.

2.2. IPS Detection Model

There are two models of detection:

- Signature-based Detection Model (this model is used in the system)
- Anomaly-based Detection Model

2.2.1. Signature-Based (pattern) Detection Model

In the signature-based detection model the input obtained from HTML form is compared to known SQL-I attack patterns (or signatures). If the input is found to match a signature, access is denied and generic invalid username/password screen is displayed.

2.2.2. Anomaly-Based (behavioral) Detection Model

In the anomaly-based detection model the number of times user attempts to log into the system (does not matter successfully or not) is considered. If the user's attempts exceed a predetermined number, then the system will lock out this user's IP for a while. Once this time is expired, the user may retry. Mentioned period of time and threshold should be arbitrary. This enables the system administrator to determine appropriate values for each particular application as different systems have different requirements.

3. IPS Design and Implementation

3.1. SQL-I Attack Detection

IPS system uses signature-based model to identify threats and attacks on the system. Signatures are carefully selected to implement the signature-based model. IPS is case-insensitive while trying to use the signature-based detection method to detect SQL-I attacks. IPS deals with White Space Manipulation attack by removing any white space before comparing text with known SQL-I attack patterns. The system will also look for hexadecimal characters in the submitted text to catch instances of this SQL-I attack variation.

4. IPS Components and Interface

The system designed and implemented in this toolkit is a complete IPS which uses a signature-based detection model. IPS contains a list of signatures to detect attacks.

The system consists of VIs, where the input data are analyzed and its security is determined. It provides protection against three types of attacks:

1. AND/OR Attack
2. White Space Manipulation Attack
3. Hexadecimal Variation Attack

The system may contain the following VIs:

1. AND_OR.vi

This VI is intended to identify *"AND/OR"* Attacks. As input data VI receives the "username" and "password" sent by user. After receiving data, it analyzes password and checks if there is an intrusion threat. The output of this VI is a boolean value, which identifies whether there is an intrusion or not.

2. Hex_bin.vi

This VI is intended to identify *"Hexadecimal Variation"* Attacks. It does not determine whether there is attack risk or not. It calls an auxiliary VI *"Hex_to_ASCII"* where hexadecimal password is transformed into ASCII code and the modified password (ASCII) is returned.

3. Hex_to_ASCII.vi

This VI is called *from* Hex_bin.vi. It receives the hexadecimal password as the input data and returns the corresponding ASCII password as a result.

4. DetectAttackByType.vi

This VI is intended to determine what type of attack an attempt is made and call VI corresponding to the detected attack.

Several types of attack may be combined, for example, it is possible that *"AND/OR"* Attack can be received by hexadecimal code. This system also includes such cases.

5. Examples

Complete the following steps to run this example:

1. In the **Project Explorer** window open **Example_for_All_Attacks.vi**
2. Enter **"UserName"** and **"Password"** parameters (if these parameters are not entered, then VI will work with default parameters)
3. Click **Run** button

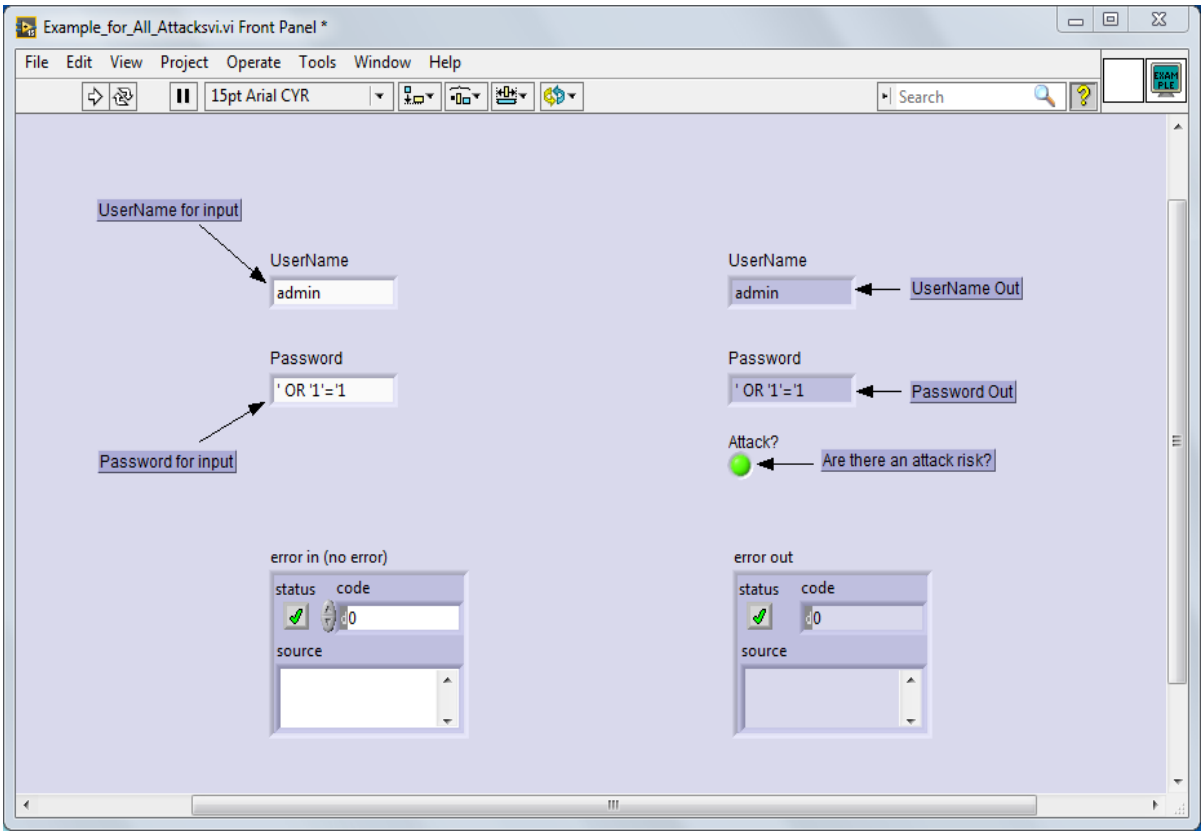

Figure 1 Example Front Panel

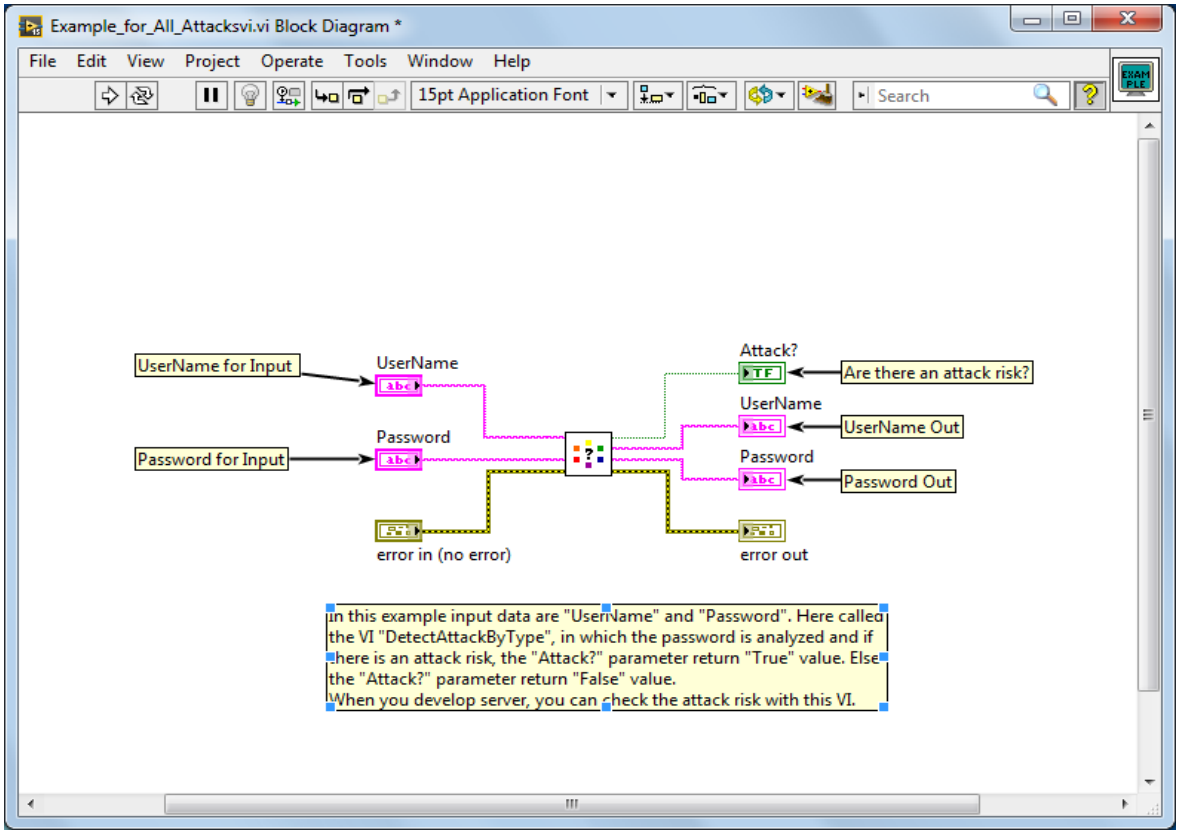

Figure 2 Example Block Diagram

5. LabVIEW Features and Concepts Used

- Case structures
- Arrays
- Shift registers
- While Loops
- For Loops
- String Functions
- “Asynchronous Call” functions group

6. System Requirements

- LabVIEW Base, Full, or Professional Development System
- Windows 7 and later

7. Support Information

For technical support, please, contact Ovak Technologies at:

Phone: +1.281.506.0020

Email: support@ovaktechnologies.com

Web: www.ovaktechnologies.com