

# SQL Injection Prevention Using the Metasploit Framework For web Application

Dr.S.Saravanan, M.E., Ph.D., Mrs. G Keerthana

M.E PROFESSOR & HEAD ASSISTANT PROFESSOR

Department of Information Technology Department of Information Agni College of Technology, Technology,  
Thalambur, Agni College of Technology, Chennai-600130.

Submitted: 05-05-2021

Revised: 17-05-2021

Accepted: 20-05-2021

**ABSTRACT:** These days the world is very much dependent on web applications. It may be for different kinds of transactions. These web applications are used by wide range of users and they are also vulnerable to various security threats. Hence providing security to these applications is of great importance. The prevention for the SQL injection was based on Osmap and we having the approach for preventing using the Metasploit framework. Metasploit framework which is used to penetrating and in this case we using filter passing to prevent the attack from the hackers.. SQL injection Attack It is the most common type of vulnerability in which a malicious mind person is inserts its own crafted query as input for retrieving personal information about others sensitive users. In this paper, for detection and prevention of SQL injection attacks using the Metasploit. As per increasing the dependency on these web applications also raises the attacks on these applications. SQL injection Attacks can be prevented using Metasploit framework Cross Site Scripting (XSS) are being a major problem for web applications.

## I. CHAPTER 1 INTRODUCTION

### 1.1 DOMAIN INTRODUCTION:

In recent years, widespread adoption of the internet has resulted in rapid advancement in information technologies. The internet is used by the general population for the purpose such as financial transactions, educational endeavors, and countless other activities. The use of the internet for accomplishing important tasks, such as transferring a balance from a bank account, always comes with a security risk. Today's web

sites strive to keep their users' data confidential and after years of doing secure business online, these companies have become experts in information security. The database systems behind these secure websites store non-critical data along with sensitive information, in a way that allows the information owners quick access while blocking break-in attempts from unauthorized users.

### INTRODUCTION:

A common break-in strategy is to try to access sensitive information from a database by first generating a query that will cause the database parser to malfunction, followed by applying this query to the desired database. Such an approach to gaining access to private information is called SQL injection. Since databases are everywhere and are accessible from the internet, dealing with SQL injection has become more important than ever. Although current database systems have little vulnerability, the Computer Security Institute discovered that every year about 50% of databases experience at least one security breach. The loss of revenue associated with such breaches has been estimated to be over four million dollars. Additionally, recent research by the "Imperva Application Center" concluded that at least 92% of web applications are susceptible to "malicious attack" (Ke Wei, M. Muthuprasanna, Suraj Kothari, 2007).

To get a better understanding of SQL injection, we need to have a good understanding of the kinds of communication that take place during a typical session between a user and a web application. The following figure shows the typical communication exchange between all the components in a typical web applications system.



Figure1:“Web application Architecture“  
 Figure1.1 SQL Query Request Result

A web application, based on the above model, takes text as input from users to retrieve information from a database. Some web applications assume that the input is legitimate and use it to build SQL queries to access a database. Since these web applications do not validate user queries before submitting them to retrieve data, they become more susceptible to SQL injection attacks.

## II. CHAPTER 2 LITERATURE SURVEY

### 2.1 INTRODUCTION:

The following shows survey did for SQL Injection Prevention. The most popular of the existing techniques is being discussed as follows.

### 2.2 LITERATURE SURVEY:

SQL injection comes with a bang and caused revolution in database attacking. In recent years, with the explosion in web-based commerce and information systems, databases have been drawing ever closer to the network and it is critical part of network security. Database is the storage brain of the website. A hacked database is the source of password and sensitive information like credit card number, bank account number and every important thing that is forbidden. SQL injection can cause severe damage to our database. Importance should be given for preventing database exploitation by SQL injection. The aim of this paper is to create awareness among web developers or database

administrators about the urgent need for database security. Our ultimate objective is to totally eradicate the whole concept of SQL injection and to avoid this technique becoming a plaything in hands of exploiters.[1] A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.[2]

### 1. HISTORY OF SQL INJECTION-

Ever since the advent of the computer, there have always been people trying to hack them. William D. Mathews of MIT discovered a flaw in the Multics CTSS password file on the IBM 7094 in 1965;

John T. Draper ("Captain Crunch") discovered a cereal toy whistle could provide free phone calls around 1971; The Chaos Computer Club, the Cult of the Dead Cow, 2600, the infamous Kevin Mitnick, even computing godfather Alan Turing and his World War II German Enigma-cipher busting Bombe, all and more have participated in hacking computers for as long as computers have existed.

Through the 1980s and 1990s, the world began to see the advent of the personal computer, the internet, and the world wide web. Telephone lines in millions of homes began screaming with the ear-piercing tones of dial up connections. AOL, CompuServe, Juno, and more began providing home users with information portals and gateways to the web. The information age was born; as was the age of information security (and, indeed, insecurity).

As websites began to form by the thousands per day, so did the technology behind them. Websites went from merely being static pages of text and images to dynamic web applications of custom-tailored content. HTML, CSS, and JavaScript grew into bigger and better systems for stitching content together in the browser, and the browser itself evolved, through Internet Explorer, Netscape, Firefox, Chrome, and more. PHP and Perl CGI, among others, became the languages of choice for backend website scripting to real-time generate the HTML and other elements browsers would render. Database systems came and went, but MySQL became the most popular. In fact, a lot of things came and went — Dot-Com bubble, anyone? — but one thing always remained: web application security.[3]

Here is a small sampling by Mavituna Security:

- In 2012, 97% of all data breaches world wide were SQL injection attacks.
- In one month, from the end of 2011 to early 2012, over 1,000,000 sites were successfully attacked with SQL injection.
- SQL injection has remained in the top 10 list of vulnerabilities compiled by the Open Web Application Security Project.

## 2. A SIMPLE SQL INJECTION

The injection process works by prematurely terminating a text string and appending a new command.[3] Because the inserted command may have additional strings appended to it before it is executed, the malefactor terminates the injected string with a comment mark "--". Subsequent text is ignored at execution time.

A simple SQL injection is shown through the following script

The script builds an SQL query by concatenating hard-coded strings together with a string entered by the user:

```
var Shipcity;  
ShipCity = Request.form ("ShipCity");  
var sql = "select * from OrdersTable where  
ShipCity = " + ShipCity + "";
```

The user is prompted to enter the name of a city. If she enters Redmond, the query assembled by the script looks similar to the following:

```
SELECT * FROM OrdersTable WHERE ShipCity = 'Redmond'
```

However, assume that the user enters the following: Redmond'; drop table OrdersTable--

In this case, the following query is assembled by the script:

```
SELECT * FROM OrdersTable WHERE ShipCity = 'Redmond';drop table OrdersTable--'
```

The semicolon (;) denotes the end of one query and the start of another. The double hyphen (-- ) indicates that the rest of the current line is a comment and should be ignored. If the modified code is syntactically correct, it will be executed by the server. When SQL Server processes this statement, SQL Server will first select all records in OrdersTable where ShipCity is Redmond. Then, SQL Server will drop Orders Table.

As long as injected SQL code is syntactically correct, tampering cannot be detected programmatically. Therefore, you must validate all user input and carefully review code that executes constructed SQL commands in the server that you are using. Coding best practices are described in the following sections in this topic.[2]

## 3. ATTACK INTENT

Attacks can also be characterized based on the goal, or intent, of the attacker. Therefore, we can define[4] several intents as follows:

Identifying injectable parameters: The attacker wants to probe a Web application to discover which parameters and user-input fields are vulnerable to SQLIA.

Performing database finger-printing: The attacker wants to discover the type and version of database that a Web application is using. Certain types of databases respond differently to different queries and attacks, and this information can be used to fingerprint the database. Knowing the type and version of the database used by a Web application allows an attacker to craft databasespecific attacks.

Determining database schema: To correctly extract data from a database, the attacker often needs to know database schema information, such as table names, column names, and column data types. Attacks with this intent are created to collect or infer this kind of information.

Extracting data: These types of attacks employ techniques that will extract data values from the database. Depending on the type of the Web application, this information could be sensitive and

highly desirable to the attacker. Attacks with this intent are the most common type of SQLIA.

**Adding or modifying data:** The goal of these attacks is to add or change information in a database.

**Performing denial of service:** These attacks are performed to shut down the database of a Web application, thus denying service to other users. Attacks involving locking or dropping database tables also fall under this category.

**Evading detection:** This category refers to certain attack techniques that are employed to avoid auditing and detection by system protection mechanisms.

**Bypassing authentication:** The goal of these types of attacks is to allow the attacker to bypass database and application authentication mechanisms. Bypassing such mechanisms could allow the attacker to assume the rights and privileges associated with another application user.

**Executing remote commands:** These types of attacks attempt to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users.

**Performing privilege escalation:** These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the attacker. As opposed to bypassing authentication attacks, these attacks focus on exploiting the database user privileges.

- Sources[5] of SQL Injection Attack
- Injection through user input

Malicious strings are introduced in web forms through user inputs.

- Injection through cookies

Modified cookie fields contain attack strings.

- Injection through server variables

Headers are manipulated to contain attack strings.

### 5.1 Second-order injection

- Trojan horse input seems fine until used in a certain situation.

Attack does not occur when it first reaches the database, but when used later on.

```
Input: admin- ===> admin\-
queryString = "UPDATE users SET pin=" +
  newPin +
  " WHERE userName=" + userName + " AND
  pin=" + oldPin;
queryString = UPDATE users SET pin=0 WHERE
  userName= admin- AND pin=1;
```

- Types of SQL Injection
- Piggy-backed Queries

Attack Intent: Extraction, modify datasets, execute remote commands, DoS

Different than other attacks not only because hacker attempts to execute two commands at once but also due to the first query not intended to modify or cause damage. First query is valid and runs normally but when delimiter is recognized DBMS executes second malicious query. System that is vulnerable to piggy-backed queries is generally due to misconfiguration which allows for multiple statements in one query

- Tautologies

This attack works by inserting an always true fragment into a WHERE clause of the SQL statement[7]. This is often used in combination with the insertion of a double dash — to cause the remainder of a statement to be ignored, ensuring extraction of largest amount of data. Tautological injections can include techniques to further mask SQL expression fragments, such as the following:

- or 'simple' like 'sim%' —
- or 'simple' like 'sim' || 'ple' —

The || in the example is used to concatenate strings, when evaluated the text 'sim' || 'ple' becomes 'simple'.

- Alternate Encodings

In this case, text is encoded to avoid detection by defensive coding practices. It can also be very difficult to generate rules for a WAF to detect encoded input. Encodings, in fact, can be used in combination with other attack classifications. Since databases parse comments out of an SQL statement prior to processing it, comments are often used in the middle of an attack to hide the attacks pattern. Scanning and detection techniques, including those used in WAFs, have not been effective against alternate encodings or comment based obfuscation because all possible encodings must be considered.

- Stored Procedure Attacks: These attacks attempt to execute database stored procedures. The attacker initially determines the database type (typically through illegal/logically incorrect queries) and then uses that knowledge to determine what stored procedures might exist.

Contrary to popular belief, using stored procedures does not make the database invulnerable to SQL injection attacks. Stored procedures can be vulnerable to privilege escalation, buffer overflows, and even provide administrative access to the operating system.

- Illegal/Logically Incorrect Queries : Attackers use this approach to gather important information about the type of database and its structure. Attacks of this nature are often used in the initial reconnaissance phase of the attack to gather critical knowledge used in

subsequent attacks. Returned error pages that are not filtered can be very instructive. Even if the application sanitizes error messages, the fact that an error is returned or not returned can reveal vulnerable or injectable parameters. Syntax errors identify injectable parameters; type errors help decipher data types of certain columns; logical errors, if returned to the user, can reveal table or column names.

The specific attacks within this class are largely the same as those used in a Tautological attack. The difference is that these are intended to determine how the system responds to different attacks by looking at the response to a normal input, an input with a logically true statement appended (typical tautological attack), an input with a logically false statement appended (to catch the response to failure) and an invalid statement to see how the system responds to bad SQL. This will often allow the attacker to see if an attack got through to the database even if the application does not allow the output from that statement to be displayed.

- **Union Query:** This attack exploits a vulnerable parameter by injecting a statement of the form: `foo'UNION SELECT <rest of injected query>` The attacker can insert any appropriate query to retrieve information from a table different from the one that was the target of the original statement. The database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

#### 4. PREVENTION TECHNIQUES

- **Defensive Coding Best Practices**

The root cause of SQL injection vulnerabilities is insufficient input validation.

**Encoding of inputs:** Injection into a string parameter is often accomplished through the use of meta-characters that trick the SQL parser into interpreting user input as SQL tokens. While it is possible to prohibit any usage of these meta-characters, doing so would restrict a non-malicious user's ability to specify legal inputs that contain such characters. A better solution is to use functions that encode a string in such a way that all meta-characters are specially encoded and interpreted by the database as normal characters.

**Positive pattern matching:** Developers should establish input validation routines that identify good input as opposed to bad input. This approach is generally called

positive validation, as opposed to negative validation, which searches input for forbidden patterns or SQL tokens. Because developers might not be able to envision every type of attack that

could be launched against their application, but should be able to specify all the forms of legal input, positive validation is a safer way to check inputs.

**Identification of all input sources:** Developers must check all input to their application. As we outlined, there

- Penetration Testing
- Static Analysis of Code
- Safe Development Libraries

Are many possible sources of input to an application. If used to construct a query, these input sources can be a way for an attacker to introduce an SQLIA. Simply put, all input sources must be checked.

Although defensive coding practices remain the best way to prevent SQL injection vulnerabilities, their application is problematic in practice. Defensive coding is prone to human error and is not as rigorously and completely applied as automated techniques. While most developers do make an effort to code safely, it is extremely difficult to apply defensive coding practices rigorously and correctly to all sources of input. In fact, many of the SQL injection vulnerabilities discovered in real applications are due to human errors: developers forgot to add checks or did not perform adequate input validation [10, 11,12]. In other words, in these applications, developers were making an effort to detect and prevent SQLIAs, but failed to do so adequately and in every needed location. These examples provide further evidence of the problems associated with depending on developers' use of defensive coding.

Moreover, approaches based on defensive coding are weakened by the widespread promotion and acceptance of so-called pseudoremedies [9]. We discuss two of the most commonly-proposed pseudo-remedies. The first of such remedies consists of checking user input for SQL keywords, such as FROM, WHERE, and SELECT, and SQL operators, such as the single quote or comment operator. The rationale behind this suggestion is that the presence of such keywords and operators may indicate an attempted SQLIA. This approach clearly results in a high rate of false positives because, in many applications, SQL keywords can be part of a normal text entry, and SQL operators can be used to express formulas or even names (e.g., OBrian). The second commonly suggested pseudo-remedy is to use stored procedures or prepared statements to prevent SQLIAs. Unfortunately, stored procedures and prepared statements can also be vulnerable to SQLIAs unless developers rigorously apply defensive coding guidelines. Interested readers may refer to

[30,31,29,16] for examples of how these pseudo-remedies can be subverted.

### VIII. DETECTION AND PREVENTION TECHNIQUES

Researchers have proposed a range of techniques to assist developers and compensate for the shortcomings in the application of defensive coding.

**Black Box Testing.** Huang and colleagues [8] propose WAVES, a black-box technique for testing Web applications for SQL injection vulnerabilities. The technique uses a Web crawler to identify all points in a Web application that can be used to inject SQLIAs.

It then builds attacks that target such points based on a specified list of patterns and attack techniques. WAVES then monitors the applications response to the attacks and uses machine learning techniques to improve its attack methodology. This technique improves over most penetration-testing techniques by using machine learning approaches to guide its testing. However, like all black-box and penetration testing techniques, it cannot provide guarantees of completeness.

**Static Code Checkers.** JDBC-Checker is a technique for statically checking the type correctness of dynamically-generated SQL queries [28,29]. This technique was not developed with the intent of detecting and preventing general SQLIAs, but can nevertheless be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. JDBC-Checker is able to detect one of the root causes of SQLIA vulnerabilities in code improper type checking of input. However, this technique would not catch more general forms of SQLIAs..

## III. CHAPTER 3

### SYSTEM ANALYSIS AND DESIGN

#### 3.1 EXISTINGSYSTEM:

It permits attackers to get unauthorized access to the database by inserting malicious SQL code into the database application through user input parameters. In this paper, we propose input-based analysis approach to detect and prevent SQL Injection Attacks (SQLIA), as an alternative to the existing solutions. This technique has two part (i) input categorization and (ii) input verifier. We provide a brief discussion of the proposal w.r.t the literature on security and time cost point of view.

#### 3.2 DISADVANTAGESOFEXISTINGSYSTEM:

Due to use of SQLI codes, the attacker may using

the defender techniques to break the SQLI firewall or the OSWAL firewall.

It is more complicated to the previous SQL injection attacks.

The transmissions from SQLI that build the firewall which will allow the attackers to perform any kind of SQL queries in Platform oriented Login.

#### PROPOSED SYSTEM:

We will use the Metasploit Framework which is used to prevent any type of attacks that made from the attacker and it will act as an online firewall agent. When the Metasploit framework is deployed with the web application, it automatically creates a hidden firewall which cannot be visible by the attackers.

By using this method we can prevent multiple attacks that multiple attackers.

Structured Query Language (SQL) injection and cross-site scripting remain a major threat to data-driven web applications. Instances where hackers obtain unrestricted access to back-end database of web applications so as to steal, edit, and destroy confidential data are increasing. Therefore, measures must be put in place to curtail the growing threats of SQL injection and XSS attacks. This study presents a technique for detecting and preventing these threats using Knuth-Morris-Pratt (KMP) string matching algorithm. The algorithm was used to match user's input string with the stored pattern of the injection string in order to detect any malicious code. The implementation was carried out using PHP scripting language and Apache XAMPP Server. The security level of the technique was measured using different test cases of SQL injection, cross-site scripting (XSS), and encoded injection attacks. Results obtained revealed that the proposed technique was able to successfully detect and prevent the attacks, log the attack entry in the database, block the system using its mac address, and also generate a warning message. Therefore, the proposed technique proved to be more effective in detecting and preventing SQL injection and XSS attacks

#### Deployment of Metasploit framework over Internet

Internet is fast becoming a household technology with 4.39 billion users in January 2019 compared to 3.48 billion users in January 2018 [1]. This showed that more than one million new users got connected daily. This growth rate is being facilitated by data-driven web applications and services which enable users to transact their online

activities with ease. Most modern organizations and individuals heavily rely on these web applications to reach out to their numerous customers. Users' inputs via web applications are used to query back end databases so as to provide the needed information. This trend has therefore opened up web applications and services to attacks by hackers. Moreover, the popularity of web application in social networking, financial transaction, and health problems are increasing very rapidly; as a result, software vulnerabilities are becoming very critical issues, and thus, web security has now become a major concern [2]. The vulnerabilities are mostly application layer vulnerabilities such as domain name server attacks, Inline Frame flaws, remote file inclusion, web authentication flaws, remote code execution, XSS, and SQL injection [3, 4]. A survey carried out by Open Web Application Security Project (OWASP) identified top 10 vulnerabilities as at June 2019 to be injection flaws, broken authentication and session management, sensitive data exposure, XML external entity, broken access control, security misconfiguration, XSS, insecure deserialization, using components with known vulnerabilities, insufficient logging, and monitoring. However, among these forms of attacks, XSS and SQL injection have been identified as the most dangerous [5]. The WordPress Security Learning Center also submits that if SQL injection and XSS vulnerabilities could be handled in a code, then 65% vulnerabilities has been eliminated. Since web applications use data supplied by users in SQL queries, hackers can manipulate these data and insert SQL meta-characters into the input fields so as to access, modify, or delete the content of the database. For instance, the WHERE clause in the SQL query `SELECT*FROM users WHERE password = 1234` could be manipulated when hackers supply inputs like `'anything' OR '1' = '1'; #`. The WHERE clause now contains two conditions separated with the logical operator OR. The first condition might not be TRUE, but the second condition must be TRUE because 1 is always equals 1, and the logical operator "OR" returns TRUE if either or both of the conditions are TRUE. Hence, the hacker gains access without a need to know the password. Sometimes, wrong input values can also be supplied intentionally so that error messages that will help the attackers to understand the database schema will be revealed. Therefore, SQL injection is a serious threat for web application users.

Cross-site scripting (XSS) attacks

XSS is another similar attack where

hackers prepare and execute a fragment of JavaScript in the security context of the targeted domain thereby incorporating malicious contents into web pages presented by a trusted web application. Most web applications that do not properly screen user input before loading web pages are susceptible to XSS attacks. Once a site has been affected, users could be redirected to automatically open malicious websites, the entire user session could be hijacked, and users' login details could also be stolen. Since the content is claimed to be from a trusted server, it is processed like normal contents. For example, the pseudo code below shows how latest comments are displayed on a website using a simple sever-site script:

The scripts assume that the comments consist of only text. However, since the user's input is directly included, an attacker can submit his comment as `"<script>doSomethingEvil();</script>"`. Therefore, users who visit the page will receive the following response:

figureb

When the user's browser loads the page, it executes whatever JavaScript is contained inside the `<script >` tags. In this case, the attacker can write a JavaScript function that steals the victim's session cookie. This session cookie can be used to impersonate the victim subsequently.

XSS vulnerabilities have been categorized into three categories which are reflected, stored, and Document Object Model (DOM)-based [3]. DOM-based vulnerabilities occur when active contents on a web page (mostly JavaScript) accept user inputs which are malicious thereby causing the execution of injected code. Stored XSS vulnerabilities occur when inputs collected via web applications are malicious and stored in the database for immediate or future use. It is one of the most dangerous of all XSS vulnerabilities because in as much as it is in the database, the hacker can manipulate the contents of the database at will [1]. Reflected XSS vulnerabilities are different from other XSS vulnerabilities because it attacks clients who accesses or loads a malicious URL. Though several techniques aimed at curtailing the growing hazards of these attacks have been reported in literature, many have not been able to fully address all scope of the problem. Several security techniques have been proposed towards preventing data and information from unauthorized attacks [6,7,8], and attackers continually devise new security vulnerabilities that could be exploited. Therefore, new techniques

aimed at detecting and preventing these attacks are essential.

#### SQL injection attacks

SQL-injection attacks could be in six categories:

a) Boolean-based SQL injection or tautology attack:

Boolean values (True or False) are used to carry out this type of SQL injection. The malicious SQL query forces the web application to return a different result depending on whether the query returns a TRUE or FALSE result. For instance, "aaa OR 2 = 2" has been inserted into SQL query "SELECT \* FROM users WHERE password = aaa OR 2 = 2" as the password so as to alter the structure of the WHERE clause of the original query. This yields a SQL query with two different conditions separated with a logical operator OR. The first condition "password = aaa" might not be true, but the second condition "2 = 2" must be true. Therefore, the logical operator OR returns true if at least one of the operand is true thereby forcing the web application to return a different result.

b) Union-based SQL injection: this is the most popular of all the SQL injections. It uses the UNION statement to integrate two or more select statements in a SQL query thereby obtaining data illegally from the database. For instance, in the SQL query "SELECT \* FROM customers WHERE password = 123 UNION SELECT creditCardNo, pin FROM customers" the attacker injects the SQL statement "123 UNION SELECT creditCardNo, pin FROM customers" instead of the required password. The query therefore exposes all the credit card numbers with their PINs from the customer's table.

c) Error-based SQL injection: this is the simplest of all the SQL injection vulnerabilities; however, it only affects web applications that use MS-SQL Server. The most common form of this vulnerability requires an attacker to supply an SQL statement with improper input causing a syntax error such as providing a string when the SQL query is expecting an integer. For example, the SQL query: SELECT \* FROM customer WHERE pin = convert (int, (SELECT firstName FROM customer LIMIT 1)) tries to convert the first name of the first customer in the customer's table into integer type which is not possible. As a result, it causes the database server to throw an error containing the information about the structure of the table.

d) Batch query SQL injection/piggy backing attacks: this form of injection is dangerous as it attempts to take full control of the database. An attacker terminates the original query of the application and injects his own query into the database server. For instance, considering the SQL query: aaa; INSERT INTO users VALUES ('Abubakar', '1234');#, the first semicolon (;) terminates the original query, and query adds the username "Abubakar" and password "1234" to the users table, and the hash (#) comments out the remaining query so that it will not be executed by the server. However, this form of attack works on only SQL-Server 2005, because it is the only server that accepts multiple queries at a time.

e) Like-based SQL injection. This injection type is used by hackers to impersonate a particular user using the SQL keyword LIKE with a wildcard operator (%). For instance, an attacker can inject input: "anything OR username LIKE 'S%' ;# instead of a username to have SQL query: SELECT \* FROM users WHERE username = ' anything OR username LIKE 'S%'; #". The LIKE operator implements a pattern match comparison, that is, it matches a string value against a pattern string containing wildcard character. The query searches the user's table and returns the records of the users whose username starts with letter S. The wildcard operator (%) means zero or more characters (S...), and it can be used before or after the pattern.

f) Hexadecimal/decimal/binary variation attack (encoded injection): in this type of injection, the hacker leverages on 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 code. For instance, the traditional SQL injection code: UNION SELECT \* FROM users; # could be replaced with:

figurec

Therefore, SQL injection vulnerability is a serious attack that must be prevented. Its different categories have further revealed that a prevention technique that works for a specific category may not perfectly work for another category. This has made the quest to eradicate SQL injection vulnerabilities an open field of research.

The proposed detection and prevention technique

With a view to come up with a technique that could detect and prevent the various forms of SQL injection and XSS attacks, the patterns for each attack were studied, and solutions were proffered based on these patterns. The



methodology employed in this study is in five phases: formation of SQL injection string pattern, designing parse tree for the various forms of attacks, detecting SQL-injection and XSS attacks, preventing SQL-injection and XSS attacks using KMP algorithm, and formulating the filter functions.

#### Formation of SQL injection string patterns

Every form of attacks has certain characters and keywords that hackers do manipulate to perpetuate their attacks. These are retrieved and documented as made available in Tables 1 and 2.

Table 1 Special characters used to compose SQL-injection code  
Full size table

Table 2 Keywords used to compose SQL-injection code  
Full size table

These characters and keywords are used to form malicious codes that are used to carry out the various forms of attacks. Identifying these injection codes will help in coming up with how to detect and prevent these attacks. The injection codes common to the various forms of attacks are provided in Table 3.

Table 3 Different forms of injection code with their common patterns  
Full size table

Designing parse tree for the various forms of attacks

Parse tree was used to represent the syntactic pattern of the various forms of SQL-Injection and Cross Site Scripting attacks. The parse trees are as follows:

#### (i). Boolean-based SQL injection attacks

figured

(`'`) followed by logical operator OR and a true statement such as `'1' = '1';#`,

#### Detecting SQL injection and XSS attacks

The various types of SQL injection and XSS attacks were detected thus:

(i). Boolean-based SQL-injection attacks: As presented in Table 3, it was deduced that most Boolean-Based SQL injection strings have a single quote (`'`) followed by logical operator OR and a true statement such as `'1' = '1';#`, `'a' <> 'b' ;#`, `'2 + 3' <= '10' ;#` (Fig. 1).

(ii). Union-based SQL injection attacks: Also, most

union-based SQL injection strings have a single quote (`'`) followed by a UNION keyword, the SQL keyword SELECT, one or more identifiers, the SQL keyword FROM, one or more identifiers then a semicolon (`;`) with hash (`#`). Example includes `'union select * from users;#` or `' union select name from a;#` (Fig. 2).

(iii). Error-based SQL injection attacks: The presence of a single quote (`'`) from the user's input, followed by zero or more SQL functions, indicates the presence of error-based SQL injection attacks. Example includes `111' convert (int, 'abcd'); A' avg('&%'$#@*')`, and `' round ('abc', 2)` (Fig. 3).

(iv). Batch query SQL injection attacks: Input strings with a single quote (`'`) followed by a SQL keyword "DROP", "DELETE", "INSERT" etc. then one or more identifiers, followed by semicolon (`;`) with a hash (`#`). Examples include `aaa'; delete * from users;#` or `' ; drop table users;#` (Fig. 4).

(v). Like-based SQL injection attack: from Table 3, category (e) shows the different forms of like-based SQL injection attack, and it is detected when the input string contains a single quote (`'`) followed by the logical operator OR, followed by one or more identifiers, followed by the SQL keyword LIKE, followed by a single quote (`'`), followed by the wildcard operator (`%`), followed by a single quote (`'`), followed by semicolon with hash. Example include `'OR username LIKE 'S%'#` and `'OR password LIKE '%2%' ;#` (Fig. 5).

(vi). XSS attack: this can be detected when a JavaScript open tag `<script>` is encountered from the input string, followed by zero or more characters and/or a single quote (`'`), followed by a JavaScript closing tag `</script>` as in `<script>alert('XSS');</script>`. If it were to be encoded XSS attack, such will have a JavaScript open tag `<script>` followed by one or more ASCII code, hexadecimal number, HTML name, or HTML number of a character and/or a single quote (`'`), followed by a JavaScript closing tag `</script>` as in `<script>alert(&#34; XSS &#34;);</script>` (Fig. 6).

Parse tree to depict Boolean-based SQL injection attacks. (ii). Union-based SQL injection string. A parse tree to depict union-based SQL injection attacks. (ii). Error-based SQL injection string. Parse tree to depict error-based SQL injection attacks. (ii). Batch query SQL injection attacks. Parse tree to depict SQL injection attacks using batch query. (ii). Like-based injection

attack. Parse tree to depict like-based injection attacks. (ii). XSS injection attacks

Parse tree to depict XSS injection attacks  
Preventing SQL-injection and XSS attacks using KMP algorithm

KMP string matching algorithm was used to compare user's input string with different SQL injection and XSS attacks patterns that have been formulated.

Formulating the filter functions

The filter() function was formulated to prevent SQL injection and XSS attacks. This function contains other functions that have been written each to detect a particular form of attack. If at least one function returns True, then, the filter () will block that user, reset the HTTP request, and display a corresponding warning message. The first statement in the algorithm below represents user's input which is collected from the web form using POST Method, and it is donated by I. The filter() then collects the user's input and firstly converts any ASCII String found in order to prevent encoded injection attack. If there is no any ASCII String and it is not empty, then, the user's input will be parsed to other functions in order to check whether it contains some injection code of Boolean-based SQLI, Union-based SQLI, Error-based SQLI, Batch query SQLI, Like-based SQLI, and XSS, and the outcomes of the functions are represented as a, b, c, d, e, and f respectively. If one of the result returns true, then, an injection string is found in the user's input, and it then triggers some functions: blockUser(), resetHTTP(), and warningMessage() so that to block the user, reset the HTTP request and issue a warning message. Otherwise, access is granted. The pseudo code illustrating this process goes thus Formulating the checkBooleanBasedSqli( ) function: this was used to prevent Boolean-based SQL injection attack. Formulating the checkUnionBasedSqli() function: this was used to prevent union-based SQL

injection attack. Formulating the checkBatchQuerySqli() function: this was used to prevent batch query SQL injection attack. Formulating the checkLikeBasedSqlis() function: this was used to prevent like-based SQL injection attack. Formulating the checkXss( ) function: this was used to prevent XSS attacks.

Therefore, to detect and prevent any of the attacks, every input strings will be passed through all the functions formulated. If at least one function return True, then, the following functions will be triggered: blockUser(), resetHTTP(), and warningMessage(). These functions are used to interact with the prospective hackers.

### 3.3 ADVANTAGES:

Provides alert to the admin of web application when the attacks happens.  
Prevent From Multiple Attcaks.  
Automatically it will create cloud firewall.  
Appnotificationtotheuser.

### 3.4 APPLICATIONS:

This project helps in preventing any time of Sql injection attack by the hackers to prevent in the Web based Application.  
Implementation of this project also gives rise many cyber security projects etc.,

## IV. CHAPTER 4 SYSTEMS SPECIFICATIONS

System Specifications is a structured collection of information that embodies the requirements of a system. The System Specification describes the functional and non-functional requirements posed on a system element (system, Enabling System or segment). In order to prepare the System Specification, the requirements will be derived from the specifications of higher system elements or from the Overall System Specification.

#### 4.1: System Architecture

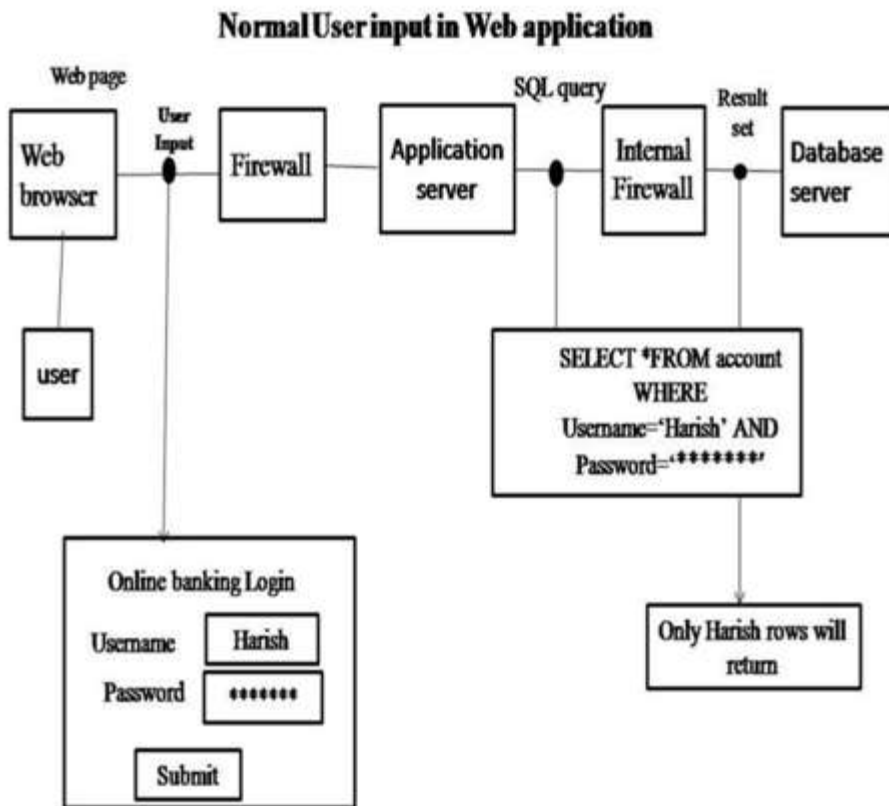


Figure 4.1: System Architecture

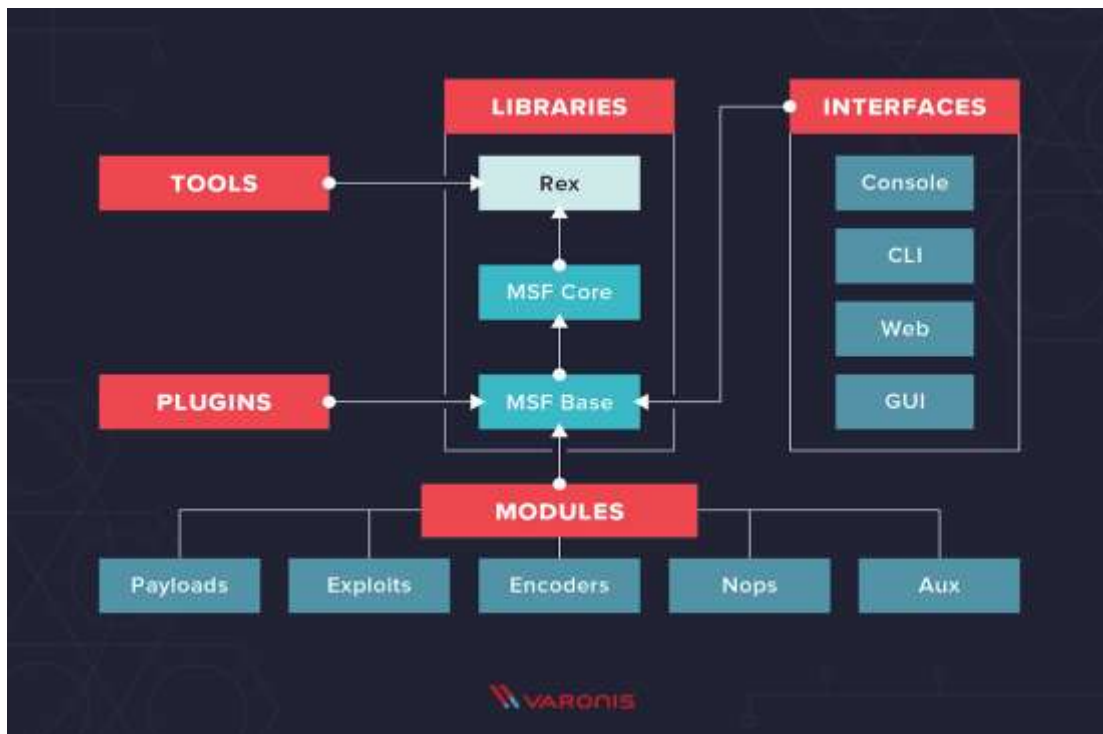


Figure 4.2 Metasploit Framework Overflow

### 4.3 WORKING

In this project This technique proposed serves as an effective method for preventing the SQLIA and session hijacking attack. The method involves the use of hashing technique, here the hashing algorithm used is SHA1. The SHA i.e. Secure Hash Algorithm is based on the concept of hash function. The basic idea of a hash function is that it takes a variable length message as input and produces a fixed length message as output which can also be called as hash or message-digest. The technique behind building a good, secured cryptographic hash function is to devise a good compression function in which each input bit affects as many output bits as possible. It is used with the Digital Signature Standard (DSA) for digital signature so it has a particular importance.[3] SHA-1 has a set of cryptographic hash functions very similar to the MD family of hash functions. But MD family uses more bits in hash function. This is the main difference between MD and SHA1. Because of this difference SHA-1 is more secure. SHA-1 differs from SHA-0 only by a single bitwise rotation in the message schedule of its compression function. SHA-1 appears to provide greater resistance to attacks. In SHA-1 input data is called message and the hash value is called message digest. Hash function takes a variable length message as an input and as an output produce a fixed length message which can also be called hash or message digests. SHA-1 has a message size of 264 bits and a message digest of 160 bits. SHA-1 is designed so that it is practically infeasible to find output of the two input messages to be the same. It is also impossible to get back the input message from the obtained message.

#### 2.1 SQL Injection Attack

The technique uses an authentication query to check for registered users of the application. The authentication query matches the user entered credentials to the credentials stored in the database during user registration. In this implementation, for each user authentication query to access the database, a unique fingerprint is generated using a hashing algorithm, based on the authentication credentials of the user, provided during the user registration. This unique fingerprint of the user is stored along with the access credentials in the database.[2]

When the user logs into the application providing the access credentials a hash digest is dynamically calculated from the user provided credentials. This dynamically calculated hash digest is then matched to the hash digest already stored in the database

which is calculated during the user registration. The user is permitted access to the application only if the two hash digest match. This can be considered similar to storing a unique digital fingerprint of the user during the registration and checking this digital fingerprint upon every user login. When the attacker tries to perform an SQLIA the hash digest generated dynamically will not match hence preventing the attacker from performing the SQLIAs.[8]

For Example: When the SQL query

```
" Select * from TableName where username =  
UserName and password = Password"
```

is passed through the SHA1 hash algorithm it generates a message digest, which is unique. When the user logs in with his login credentials, the hash digest of the user authentication query is dynamically calculated and compare it against the already stored hash digest. If the hash digest matches then there is no SQLIA. This methodology works because SQL has a fixed syntax for authentication query and if

the adversary tries to perform a SQLIA then the syntax of the query would be different. Thus the hash digest computed is different and the user is not authenticated.

The hash function is used for authentication as it is collision resistant. Also hash functions have avalanche property i.e. if even a character of input query changes, the output hash digest varies by more than half the output characters. The process flow for this implementation is shown in figure 1.

Pseudo Code for prevention of SQLIA:

1. On user registration, generate hash of the select query  
HashDigest = SHA1( Select from TableName where username = UserName and password
2. Store HashDigest as an attribute in the database against the user information.
3. On user login, during authentication calculate dynamically the hash value of the query against the user name and password entered using SHA1.
4. Compare the dynamically calculated hash against HashDigest.
5. User is authenticated only if the two hash digests match
6. Else, either the user authentication provided is invalid or it is a session hijacking attack.

Through the implementation model, the following SQLIAs are prevented.

1. Tautology Attack: In this attack the adversary tries to bypass the user authentication fields required to access the application. The authentication query for providing access to the user,

checks the database for registered users. This authentication query usually has a WHERE condition. The attacker takes

16

Figure 1. Process ow of SQL Injection prevention Technique

17

advantage of this condition to perform a SQLIA. The attacker can insert the condition which would always evaluate to true.

' or 1=1--

With this the attacker will close the actual authentication query, insert an OR condition which will always evaluate to true and comment out any other authentication conditions present in the query.

2. Piggy Backing Attack: The main aim of this attack is to modify or add data, data extraction, dropping or deleting user or database tables, remote execution of commands and performing service denials. The attacker can use the statement "' or 1=1;drop table AdminTable;{" to perform this attack.

Here the attacker closes the current authentication statement and then piggy backs another SQL query which can extract information, delete or drop any other table from the database. Using this attack the attacker can delete the permissions table and gain access to any database tables.

3. Union Attacks: The attacker uses the SQL UNION operator to retrieve records from another table. An attacker can use the SQL statement "'UNION select \* from AllTables;{" for Union attack.

The attacker here tries to get information from the other database tables by using union operator.

4. Blind SQL Injection attack: A way of evaluating if a system is vulnerable to attacks is by considering the query obtained from string

Select \_ from users WHERE username =0 user0

18

and suppose it is used to display public user information on a Web page. In particular, data from the \_rst returned record are shown. To see if this can be

exploited in a blind injection it is enough to inject the following two usernames:

luccio AND 1 = 0

luccio AND 0 = 0

If the system is not vulnerable to attacks, e.g., by \_ltering user input, it will

behave in the same way in the two cases. If it is vulnerable, instead, the results of the query will be empty in the \_rst case (1=0 is not true), and the same as for luccio in second case (given that 0=0 is

always true). What will be

displayed in case of an empty query depends on how the application handles that case: it could be either an error message or a broken Web page. In any case, the ability of distinguishing true and false answers is enough to mount a BSQLi attack.

The attack proceeds by

(a) Injecting a query

(b) Comparing the result with the previous pages to check if the resulting query is true or false. Items 4a and 4b are run again as many times as necessary

## 2.2 Session Hijacking

Hypertext Transfer Protocol (HTTP) uses session based communication to keep a user/browser state; hence an HTTP session is prone to the session injection attack. An HTTP session cookie is stored in a users browser to preserve the temporary state of a users session. Like for example, once the user is authenticated by the web application, the session state is saved so that querying the database is not required again and again for authentication. Also session cookies can be used to save the intermediate state of users session between navigation of pages. It can be thought of as a volatile quick accessible memory assigned to a user who is accessing the web application.

The session cookies are usually stored in the clients browser; these cookies are usually transmitted over an insecure channel. If this session cookie is obtained by a passive attack then the user's session can be compromised. This type of attack is known as session sidejacking, here attacker can use packet sni\_ng tools to capture the network tra\_c between two principals and to steal their session cookies.

Another type of session hijacking attack is Session \_xation, here the attacker can set the users session id to any session id know to him. This could be done by the attacker by sending an email to the user with a link that opens a session whose session id is known to the attacker. Alternatively, another type of session hijacking attack can be done by cross-site scripting, where the adversary tricks the user to run a code which is treated as trustworthy as it appears to belong to the server. This allows the attacker to obtain the session cookie and perform session hijacking. Any legitimate user of an application can be the victim of session hijacking attack. Initially the legitimate user sends the application request to the application server. The server then asks for user authentication, for which the user responds with his credentials (here it is considered a user is already registered with the applicati on).

The server now authenticates the user and establishes a user session with a unique session ID

to keep track of the user session.

## V. CHAPTER 5

### MODULE EXPLANATION

#### 5.1 LIST OF MODULES:

5.1.1 Deploy Metasploit Framework Over Web Application

5.1.2 Pass the Required Filter To Prevent Attack

#### 5.1.1 Deploy Metasploit Framework Over Web Application

The Metasploit framework should embed with the web application in case to prevent the Web application from the hackers or the attackers from the malicious attack. Metasploit is recognized as a popular penetration testing framework as well as a toolkit. In general, it is an important tool being used to exploit security vulnerabilities in programs and taking advantage of such vulnerabilities to put control over an information system. It provides persons with the facility to create their own exploits for security vulnerabilities and use them to attack machines.

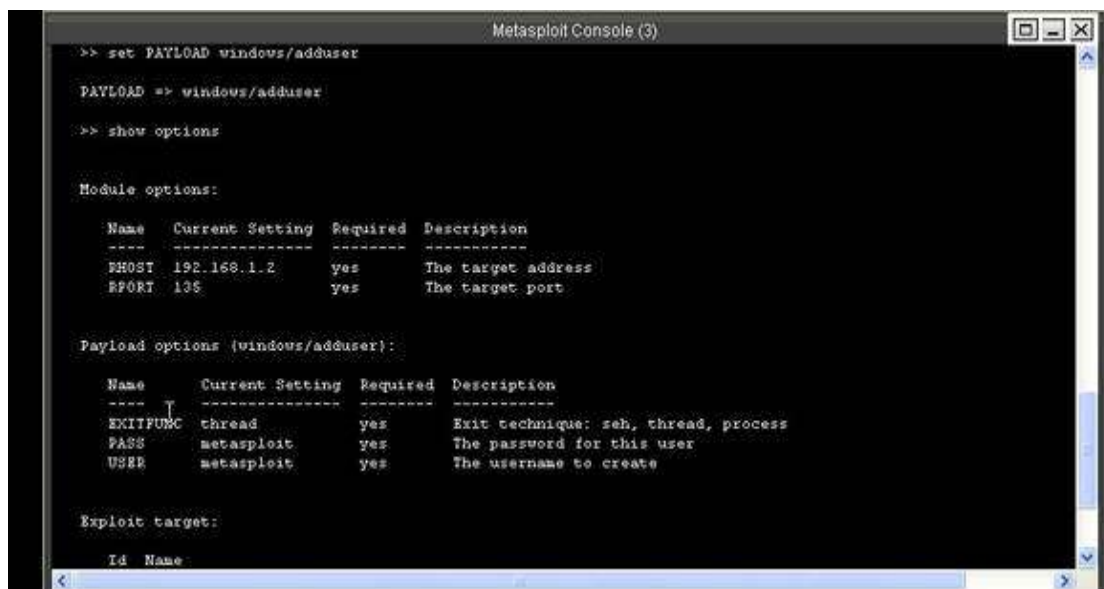
Particularly talking about the automated assessment of security vulnerabilities, it has emerged as the most popular tool to perform hacking operations. Alongside this, it has been a critical tool used to protect an organization's network. It is an effective tool used to identify and exploit an organization's security holes that most of the attackers use Metasploit as a tool to attack a vulnerable system.

How it works

Metasploit is a suite of several applications being used to automate several stages of penetration testing. It can extend its use to its framework in the case to identify a security vulnerability and exploit it using the controlling interface along with the post-exploitation and reporting tools. Its framework extracts data from a vulnerability scanner using the information related to the vulnerable hosts to detect vulnerabilities to exploit them and performing an attack with the help of a payload and exploit the system.

Attackers exploit results extracted from the vulnerability scanner and import them into Armitage, a graphical cyber attack management tool for the Metasploit Project to recognize vulnerabilities with its modules. After identifying the vulnerabilities attackers utilize a usable exploit to affect the system and get a shell and launch Meterpreter, a dynamically extensible payload, to control the system.

Payloads refer to the commands being used to execute on the local system after gaining access through an exploit. It might include documentation and a database of techniques utilized to develop a functioning exploit after the identification of vulnerability. These payloads typically comprise components to extract passwords from the local system, install other software or to restrain the hardware alike recently available tools like BO2K.



```

Metasploit Console (3)
>> set PAYLOAD windows/adduser
PAYLOAD => windows/adduser
>> show options

Module options:

  Name      Current Setting  Required  Description
  ----      -
  RHOST     192.168.1.2     yes       The target address
  RPORT     135              yes       The target port

Payload options (windows/adduser):

  Name      Current Setting  Required  Description
  ----      -
  EXITFUNC  thread          yes       Exit technique: seh, thread, process
  PASS      metasploit      yes       The password for this user
  USER      metasploit      yes       The username to create

Exploit target:

  Id  Name
  --  -
  
```

#### 5.1.1 Deploy Metasploit Framework Over Web Application

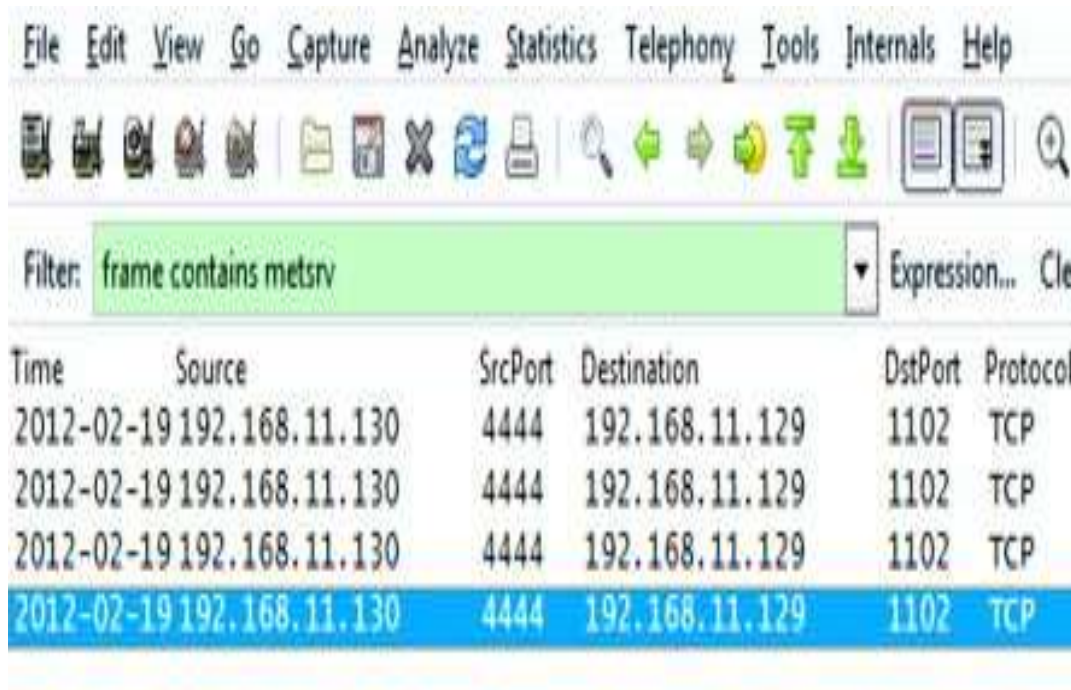
Devoiding Metasploit-oriented attacks

Being an information security tool, Metasploit finds its applications in both security defense and attacks. Malicious hackers utilize it against

organizations to exploit security vulnerabilities and allowing them unauthenticated access to the networks, applications, and information systems.

The filter are used to prevent the attacker which are associated with the framework and it doesn't allow the hackers or the attackers to make an SQL injection over the website.

**5.1.2 Pass the Required Filter To Prevent Attack**



**5.1.3 Pass the Required Filter To Prevent Attack**

Once you have used Wireshark and built filters enough times, you tend to remember the ones you use most often and can simply type them in the filter window. One feature which is a great help is the fact that the filter window will change color, letting you know if the filter you are creating is correct or not. As you type in your filter, the window will turn almost a reddish pink color indicating that the statement is incorrect. Once you have fixed or finished your statement and it is correct, the window will be green. In the below example, I created a simple filter that will only show me the packets that pertain to the listed IP address. In this instance, Wireshark will display the packets that the listed IP address was either the sender or receiver. Once you are satisfied with your filter.

Once we have our capture file filtered so that we are only looking at what we want, we can now look at the remaining streams. As you can see in the picture below, I have filtered my capture file by using a single IP address and the first two

packets listed are the SYN and SYN-ACK of a connection. These two packets are colored according to my created coloring rules. To view the entire TCP stream, I right click on either one of the packets and select, Follow TCP stream, from the drop-down menu. A new window will open containing the contents of the entire TCP stream (see Figure 5.16).

**VI. CHAPTER-6**

**CONCLUSION AND FUTURE ENHANCEMENT**

**6.1 CONCLUSION**

In this project we will protect the web based applications and E commerce application using the new methodology using the filters of the Metasploit framework which will prevent the Sql injection attack from the attackers and hackers to prevent from stealing the details and ordering without money.

**6.2 FUTURE WORK**

In the coming future, we review the applic

ation of our project and we will improve by building a dynamic cloud metasploitvpn which will act as a cloud firewall which will act like a platform independent and that will prevent

all the attacks if the user just connected with metasploit cloud VPN.

## APPENDIX-A (CODING)

Main.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package testInjection;

import detectapp.Config;
import javax.swing.JOptionPane;

/**
 *
 * @author SSE
 */
public class TestLoginForm extends javax.swing.JFrame {

    /**
     * Creates new form TestLoginForm
     */
    public TestLoginForm() {
        initComponents();
    }

    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initComponents() {

        jLabel1 = new javax.swing.JLabel();
        jSeparator1 = new javax.swing.JSeparator();
        jLabel2 = new javax.swing.JLabel();
        jTextField1 = new javax.swing.JTextField();
        jLabel3 = new javax.swing.JLabel();
        jButton1 = new javax.swing.JButton();
        jPasswordField1 = new javax.swing.JPasswordField();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

        jLabel1.setFont(new java.awt.Font("Tahoma", 0, 18)); // NOI18N
        jLabel1.setText("Secure Login");

        jLabel2.setText("Enter Username: ");
```



```
jLabel3.setText("Enter Password:");

jButton1.setText("Login");
jButton1.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jButton1ActionPerformed(evt);
    }
});

javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
getContentPane().setLayout(layout);
layout.setHorizontalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addComponent(jSeparator1)
        .addGroup(layout.createSequentialGroup()
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                    .addGap(172, 172, 172)
                    .addComponent(jLabel1)
                    .addGap(0, 170, Short.MAX_VALUE))
                .addGroup(layout.createSequentialGroup()
                    .addGap(20, 20, 20)
                    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                        .addComponent(jLabel2)
                        .addComponent(jLabel3))
                    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
                    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                        .addComponent(jTextField1)
                        .addComponent(jPasswordField1)))
                .addGroup(javax.swing.GroupLayout.Alignment.TRAILING, layout.createSequentialGroup()
                    .addGap(95, 95, 95)
                    .addComponent(jButton1, javax.swing.GroupLayout.PREFERRED_SIZE,
                        javax.swing.GroupLayout.PREFERRED_SIZE)))
            .addContainerGap()
        );
layout.setVerticalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup()
            .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                    .addGap(23, 23, 23)
                    .addComponent(jLabel1)
                    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                    .addComponent(jSeparator1, javax.swing.GroupLayout.PREFERRED_SIZE,
                        javax.swing.GroupLayout.PREFERRED_SIZE), 10,
                    .addGap(18, 18, 18)
                    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                        .addComponent(jLabel2)
                        .addComponent(jTextField1, javax.swing.GroupLayout.PREFERRED_SIZE,
                            javax.swing.GroupLayout.PREFERRED_SIZE))
                    .addGap(18, 18, 18)
                    .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                        .addComponent(jLabel3)
                        .addComponent(jPasswordField1, javax.swing.GroupLayout.PREFERRED_SIZE,
                            javax.swing.GroupLayout.PREFERRED_SIZE))
                    .addGap(18, 18, 18)
                    .addComponent(jButton1)
                    .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
            )
        );
```

```
);  
  
pack();  
} // </editor-fold>  
  
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    JOptionPane.showMessageDialog(null, Config.monitor);  
    if (Config.monitor.equals("Yes")) {  
        if (this.jPasswordField1.getText().contains("1=1")) {  
            JOptionPane.showMessageDialog(null, "Your are supposed to passing sql injection stopped by Metasploit");  
        } else {  
            JOptionPane.showMessageDialog(null, "Fine Everything is Ok You are Done");  
        }  
    } else {  
        JOptionPane.showMessageDialog(null, "Login Successful");  
    }  
  
}  
  
/**  
 * @param args the command line arguments  
 */  
public static void main(String args[]) {  
    /* Set the Nimbus look and feel */  
    //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">  
    /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.  
     * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html  
     */  
    try {  
        for (javax.swing.UIManager.LookAndFeelInfo info :  
            javax.swing.UIManager.getInstalledLookAndFeels()) {  
            if ("Nimbus".equals(info.getName())) {  
                javax.swing.UIManager.setLookAndFeel(info.getClassName());  
                break;  
            }  
        }  
    } catch (ClassNotFoundException ex) {  
        java.util.logging.Logger.getLogger(TestLoginForm.class  
            .getName()).log(java.util.logging.Level.SEVERE, null, ex);  
    }  
  
    catch (InstantiationException ex) {  
        java.util.logging.Logger.getLogger(TestLoginForm.class  
            .getName()).log(java.util.logging.Level.SEVERE, null, ex);  
    }  
  
    catch (IllegalAccessException ex) {  
        java.util.logging.Logger.getLogger(TestLoginForm.class  
            .getName()).log(java.util.logging.Level.SEVERE, null, ex);  
    }  
}
```

```
    }

    catch (javax.swing.UnsupportedLookAndFeelException ex) {
    java.util.logging.Logger.getLogger(TestLoginForm.class

    .getName()).log(java.util.logging.Level.SEVERE, null, ex);
    }
    //</editor-fold>

    /* Create and display the form */
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new TestLoginForm().setVisible(true);
        }
    });
    }

    // Variables declaration - do not modify
    private javax.swing.JButton jButton1;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JPasswordField jPasswordField1;
    private javax.swing.JSeparator jSeparator1;
    private javax.swing.JTextField jTextField1;
    // End of variables declaration
    }
    Controlpanel.java

    /*
    * To change this license header, choose License Headers in Project Properties.
    * To change this template file, choose Tools | Templates
    * and open the template in the editor.
    */
    package detectapp;

    import injectFileters.RegexObj;
    import static injectFileters.SQLFilter.outputToFile;
    import static injectFileters.SQLFilter.regexes;
    import static injectFileters.SQLFilter.sqlHandler;
    import static injectFileters.SQLFilter.sqlRegexChecker;
    import static injectFileters.SQLFilter.sqlStringChecker;
    import static injectFileters.SQLFilter.stringsToCheck;
    import java.io.BufferedReader;
    import java.io.BufferedWriter;
    import java.io.File;
    import java.io.FileNotFoundException;
    import java.io.FileReader;
    import java.io.FileWriter;
    import java.io.IOException;
    import java.util.regex.Matcher;
    import javax.swing.JOptionPane;

    /**
     *
     * @author SSE
```

```
*/
public class ControlPanel extends javax.swing.JFrame {

    public static final String stringsToCheck[] = { "select", "drop", "from",
        "exec", "exists", "update", "delete", "insert", "cast", "http",
        "sql", "null", "like", "mysql", "()", "information_schema",
        "sleep", "version", "join", "declare", "having", "signed", "alter",
        "union", "where", "create", "shutdown", "grant", "privileges" };

    // for reference, regex metachars that need escaped <({{^-=#!}})?*+.>

    // instantiate each RegexObj with the expression and a plain english
    // description

    // /* and */
    public static RegexObj regex1 = new RegexObj("(\\/*).*(\\*/)",
        "Found /* and */");

    // -- at the end
    public static RegexObj regex2 = new RegexObj("(--.*)$", "-- at end of sql");

    // ; and at least one " or '
    public static RegexObj regex3 = new RegexObj(";+\\\"\\'",
        "One or more ; and at least one \" or \\'");

    // two or more "
    public static RegexObj regex4 = new RegexObj("\"{2,}+", "Two or more \\"");

    // two or more '
    public static RegexObj regex5 = new RegexObj("\\' {2,}+", "Two or more \\'");

    // anydigit=anydigit
    public static RegexObj regex6 = new RegexObj("\\d=\\d", "anydigit=anydigit");

    // two or more white spaces in a row
    public static RegexObj regex7 = new RegexObj("(\\s\\s)+",
        "two or more white spaces in a row");

    // # at the end
    public static RegexObj regex8 = new RegexObj("(#.*)$", "# at end of sql");

    // two or more %
    public static RegexObj regex9 = new RegexObj("% {2,}+",
        "Two or more %% signs");

    // admin and one of [; ' =] before or after admin
    public static RegexObj regex10 = new RegexObj(
        "(;\\\"\\'|=)+.*(admin.*))((admin.*).*;\\\"\\'|=)",
        "admin (and variations like administrator) and one of [; ' \\ =] before or after
admin");

    // ASCII in hex
    public static RegexObj regex11 = new RegexObj("%+[0-7]+[0-9|A-F]+",
        "ASCII Hex");
```

```
// declare array to hold each regex, can add to this easily
public static final RegexObjregexes[] = { regex1, regex2, regex3, regex4,
                                           regex5, regex6, regex7, regex8, regex9, regex10, regex11 };

/**
 * Creates new form ControlPanel
 */
public ControlPanel() {
initComponents();
this.jButton2.setEnabled(false);
this.jButton3.setEnabled(false);
    ///this.jButton4.setEnabled(false);
}

public void startUp(){
    File file = new File("output20.txt");

    // if file doesn't exist then create it
    if (!file.exists()) {
        try {
            file.createNewFile();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    } else {
        // otherwise delete it for fresh output
        file.delete();
    }

    // vars for stats classification
    int condPos = 0;
    int condNeg = 0;
    int truePos = 0;
    int trueNeg = 0;
    int falsePos = 0;
    int falseNeg = 0;

    // string counter
    int stringCounter = 0;

    System.out.println("--Welcome to the SQL Injection Filter--");

    // for each line in the text file, read each line of the SQL strings in
    // from the text file - the label and the string itself
    try {

        // create new file object, hardcode and pass in the dataset file e.g. Queries80.txt or
        // Queries20.txt. If the file is not present in the project directory
        // you will need to specify the absolute path
        File sqlFile = new File("Queries20.txt");

        // create new file reader
```

```
FileReaderfileReader = new FileReader(sqlFile);

// create new buffered reader
BufferedReaderbuffReader = new BufferedReader(fileReader);

// declare string to hold a line
String line;
// initialise it to hold an empty string
line = "";

// int hold 0 or 1, 0 is benign and 1 is malicious string
int prediction;

// read in a line of text from the file
line = buffReader.readLine();

while (line != null) {

    // increment string counter
    stringCounter++;

    // print it
    System.out.printf("-----");
    System.out.printf("\nSample %d = %s\n", stringCounter, line);

    // get the label and store it
    char label = line.charAt(0);

    // increment the true totals accordingly
    if (label == '0') {
        condNeg++;
    } else if (label == '1') {
        condPos++;
    } else {
        System.out.println("Invalid label...");
    }
    // trace
    System.out.printf("True Label = %c\n", label);

    // get the sql string
    String sqlString = line.substring(2);
    System.out.printf("SQL = %s\n", sqlString.toLowerCase());

    // if true is returned, then classify as malware, otherwise
    // benign
    if (sqlHandler(sqlString)) {
        prediction = 1;
        if (Character.getNumericValue(label) == prediction) {
            // hit
            truePos++;
        } else {
            // false alarm
            falsePos++;
        }
    }
    } else {
        prediction = 0;
    }
}
```

```
        if (Character.getNumericValue(label) == prediction) {
            // correctly rejected
            trueNeg++;
        } else {
            // missed it
            falseNeg++;
        }
    }

    // write to output file - line, label,
    outputToFile(sqlString, label, prediction, file);

    // and read in the next line
    line = buffReader.readLine();
}

// close resources
buffReader.close();
fileReader.close();

// print results
System.out.println("*****");
System.out.printf(
    "Results for dataset file %s.\nOutput file is %s\n",
    sqlFile.getAbsolutePath(), file.getAbsolutePath());
System.out.printf("\nTotal strings read = %s\n", stringCounter);
System.out
    .printf("True malware = %d : Hits (true positives) = %d, Misses
(false negatives) = %d\n",
            condPos, truePos, falseNeg);
System.out
    .printf("True benign = %d : Correct Rejections (true negatives) =
%d, False Alarms (false positives) = %d\n",
            condNeg, trueNeg, falsePos);
System.out.println();
System.out
    .printf("Detection Rate (True Positive Rate - how well the filter
correctly classifies malware) = %.1f%%\n",
            (double) truePos / (double) condPos * 100);
System.out
    .printf("Rejection Rate (True Negative Rate - how well the filter
correctly classifies benign) = %.1f%%\n",
            (double) trueNeg / (double) condNeg * 100);
System.out.printf("Accuracy = %.1f%%\n",
    (double) (truePos + trueNeg) / (double) (condPos + condNeg)
    * 100);

} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

}

public static boolean sqlRegexChecker(String sqlToCheck) {
```

```
System.out.println("\nRunning SQL Regex Checker");

// bool for each regex
boolean pass = false;
// bool to return overall
boolean overall = false;

Matcher matcher;

// convert to lower case to handle obfuscation with mixed upper and
// lower case
sqlToCheck = sqlToCheck.toLowerCase();

// regex checking
for (RegexObj regex : regexes) {

    // check sqlToCheck vs regex, if pattern returns i.e. regex returns
    // true

    matcher = regex.getRegexPattern().matcher(sqlToCheck);

    pass = matcher.find();

    if (pass) {

        System.out
            .printf("Malicious input found via regex (%s), predicted
                label = 1\n",
                    regex.getDescription());

    } else {

        System.out
            .printf("No malicious input found via regex (%s),
                predicted label = 0\n",
                    regex.getDescription());

    }

    // if a regex returns true for the first time (i.e. overall is still
    // false), then make overall true
    if ((pass) && (!overall)) {
        overall = true;
    }
}
return overall;
}

public static boolean sqlStringChecker(String sqlToCheck) {

    boolean pass = false;

    System.out.println("\nRunning SQL String Checker");

    // convert to lower case to handle obfuscation with mixed upper and
    // lower case
    sqlToCheck = sqlToCheck.toLowerCase();
```



```
// for each string in stringsToCheck
for (String command :stringsToCheck) {

    if (sqlToCheck.contains(command)) {

        System.out
            .printf("SQL string found (%s), predicted label = 1\n",
                command);

        if (!pass) {
            pass = true;
        }
    }

    if (!pass) {
        System.out.println("No SQL command found, predicted label = 0");
    }

    return pass;
}

public static boolean sqlHandler(String sqlString) {

    // use two more bools for returns from sqlStringChecker and sqlRegexChecker
    boolean pass1 = false;
    boolean pass2 = false;

    // call both, pass in the string
    pass1 = sqlStringChecker(sqlString);
    pass2 = sqlRegexChecker(sqlString);

    // if either checker is true return true otherwise return false
    if (pass1 || pass2) {
        return true;
    } else {
        return false;
    }
}

public static void outputToFile(String SQL, char label, int prediction,
    File file) {

    // create result string
    String result = SQL + " True Label = " + label + " Predicted Label = "
        + prediction + "\r\n";

    try {

        // true = append to end of file, false = write from the start
        FileWriter fileWriter = new FileWriter(file.getAbsolutePath(), true);

        // do the writing
```

```
        BufferedWriterbufferWriter = new BufferedWriter(fileWriter);
        bufferWriter.write(result);

        // close resources
        bufferWriter.close();
        fileWriter.close();

    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

/**
 * This method is called from within the constructor to initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is always
 * regenerated by the Form Editor.
 */
@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code">
private void initComponents() {

    jPanel1 = new javax.swing.JPanel();
    jLabel1 = new javax.swing.JLabel();
    jPanel2 = new javax.swing.JPanel();
    jPanel3 = new javax.swing.JPanel();
    jScrollPane1 = new javax.swing.JScrollPane();
    jTextArea1 = new javax.swing.JTextArea();
    jPanel4 = new javax.swing.JPanel();
    jButton1 = new javax.swing.JButton();
    jButton2 = new javax.swing.JButton();
    jButton3 = new javax.swing.JButton();
    jButton4 = new javax.swing.JButton();

    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

    jPanel1.setBackground(new java.awt.Color(0, 0, 0));
    jPanel1.setBorder(new
javax.swing.border.SoftBevelBorder(javax.swing.border.BevelBorder.LOWERED));

    jLabel1.setFont(new java.awt.Font("Times New Roman", 1, 36)); // NOI18N
    jLabel1.setForeground(new java.awt.Color(102, 255, 102));
    jLabel1.setText("METASPLOIT CONTROL PANEL");

    javax.swing.GroupLayout jPanel1Layout = new javax.swing.GroupLayout(jPanel1);
    jPanel1.setLayout(jPanel1Layout);
    jPanel1Layout.setHorizontalGroup(
        jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING, jPanel1Layout.createSequentialGroup()
                .addGroup(jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                    .addComponent(jLabel1)
                    .addGroup(jPanel1Layout.createSequentialGroup()
                        .addGap(143, 143, 143))
                )
            )
    );
}
```

```
);
jPanel1Layout.setVerticalGroup(
    jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addGroup(javax.swing.GroupLayout.Alignment.TRAILING, jPanel1Layout.createSequentialGroup())
.addContainerGap(46, Short.MAX_VALUE)
.addComponent(jLabel1)
.addGap(40, 40, 40)
);

jPanel2.setBackground(new java.awt.Color(0, 0, 0));
jPanel2.setBorder(new
javax.swing.border.SoftBevelBorder(javax.swing.border.BevelBorder.LOWERED));
jPanel2.setForeground(new java.awt.Color(51, 255, 51));

jPanel3.setBackground(new java.awt.Color(0, 0, 0));
jPanel3.setBorder(javax.swing.BorderFactory.createTitledBorder(null, "FILTERS",
javax.swing.border.TitledBorder.DEFAULT_JUSTIFICATION,
javax.swing.border.TitledBorder.DEFAULT_POSITION, new java.awt.Font("Tahoma", 0, 11), new
java.awt.Color(51, 255, 51))); // NOI18N

jScrollPane1.setHorizontalScrollBarPolicy(javax.swing.ScrollPaneConstants.HORIZONTAL_SCROLLBAR_A
LWAYS);

jScrollPane1.setVerticalScrollBarPolicy(javax.swing.ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWA
YS);

jTextArea1.setBackground(new java.awt.Color(0, 0, 0));
jTextArea1.setColumns(20);
jTextArea1.setForeground(new java.awt.Color(0, 255, 0));
jTextArea1.setRows(5);
jScrollPane1.setViewportView(jTextArea1);

javax.swing.GroupLayout jPanel3Layout = new javax.swing.GroupLayout(jPanel3);
jPanel3.setLayout(jPanel3Layout);
jPanel3Layout.setHorizontalGroup(
    jPanel3Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addGroup(jPanel3Layout.createSequentialGroup()
.addContainerGap()
.addComponent(jScrollPane1, javax.swing.GroupLayout.DEFAULT_SIZE, 418, Short.MAX_VALUE)
.addContainerGap()
);
jPanel3Layout.setVerticalGroup(
    jPanel3Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addGroup(jPanel3Layout.createSequentialGroup()
.addComponent(jScrollPane1, javax.swing.GroupLayout.DEFAULT_SIZE, 366, Short.MAX_VALUE)
.addContainerGap()
);

jPanel4.setBackground(new java.awt.Color(0, 0, 0));
jPanel4.setBorder(javax.swing.BorderFactory.createTitledBorder(null, "CONTROLS",
javax.swing.border.TitledBorder.DEFAULT_JUSTIFICATION,
javax.swing.border.TitledBorder.DEFAULT_POSITION, new java.awt.Font("Tahoma", 0, 11), new
java.awt.Color(0, 255, 51))); // NOI18N
jPanel4.setForeground(new java.awt.Color(0, 255, 51));
```

```
jButton1.setBackground(new java.awt.Color(255, 255, 255));
jButton1.setFont(new java.awt.Font("Tahoma", 1, 14)); // NOI18N
jButton1.setForeground(new java.awt.Color(0, 255, 0));
jButton1.setText("Start Metasploit Controller");
jButton1.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jButton1ActionPerformed(evt);
    }
});

jButton2.setBackground(new java.awt.Color(255, 255, 255));
jButton2.setFont(new java.awt.Font("Tahoma", 1, 14)); // NOI18N
jButton2.setForeground(new java.awt.Color(51, 255, 51));
jButton2.setText("Load Filters");
jButton2.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jButton2ActionPerformed(evt);
    }
});

jButton3.setBackground(new java.awt.Color(255, 255, 255));
jButton3.setFont(new java.awt.Font("Tahoma", 1, 14)); // NOI18N
jButton3.setForeground(new java.awt.Color(51, 255, 51));
jButton3.setText("Start Monitoring");
jButton3.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jButton3ActionPerformed(evt);
    }
});

jButton4.setBackground(new java.awt.Color(255, 255, 255));
jButton4.setFont(new java.awt.Font("Tahoma", 1, 14)); // NOI18N
jButton4.setForeground(new java.awt.Color(51, 255, 51));
jButton4.setText("Logout");
jButton4.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jButton4ActionPerformed(evt);
    }
});

javax.swing.GroupLayout jPanel4Layout = new javax.swing.GroupLayout(jPanel4);
jPanel4.setLayout(jPanel4Layout);
jPanel4Layout.setHorizontalGroup(
    jPanel4Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
        .addGroup(javax.swing.GroupLayout.Alignment.TRAILING, jPanel4Layout.createSequentialGroup()
            .addContainerGap()
            .addComponent(jButton1, javax.swing.GroupLayout.DEFAULT_SIZE, 326, Short.MAX_VALUE)
            .addComponent(jButton2, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
            .addComponent(jButton3, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
            .addComponent(jButton4, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
            .addGap(21, 21, 21))
);
```

```

jPanel4Layout.setVerticalGroup(
    jPanel4Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addGroup(jPanel4Layout.createSequentialGroup())
.addContainerGap()
.addComponent(jButton1,                javax.swing.GroupLayout.PREFERRED_SIZE,           72,
javax.swing.GroupLayout.PREFERRED_SIZE)
.addGap(18, 18, 18)
.addComponent(jButton2,                javax.swing.GroupLayout.PREFERRED_SIZE,           63,
javax.swing.GroupLayout.PREFERRED_SIZE)
.addGap(18, 18, 18)
.addComponent(jButton3,                javax.swing.GroupLayout.PREFERRED_SIZE,           69,
javax.swing.GroupLayout.PREFERRED_SIZE)
.addGap(18, 18, 18)
.addComponent(jButton4,                javax.swing.GroupLayout.PREFERRED_SIZE,           63,
javax.swing.GroupLayout.PREFERRED_SIZE)
.addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
);

javax.swing.GroupLayout jPanel2Layout = new javax.swing.GroupLayout(jPanel2);
jPanel2.setLayout(jPanel2Layout);
jPanel2Layout.setHorizontalGroup(
    jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addGroup(jPanel2Layout.createSequentialGroup())
.addContainerGap()
.addComponent(jPanel3,                javax.swing.GroupLayout.PREFERRED_SIZE,           72,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
.addComponent(jPanel4,                javax.swing.GroupLayout.PREFERRED_SIZE,           63,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
.addContainerGap(19, Short.MAX_VALUE))
);
jPanel2Layout.setVerticalGroup(
    jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addGroup(jPanel2Layout.createSequentialGroup())
.addContainerGap()
.addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addGroup(jPanel2Layout.createSequentialGroup())
.addComponent(jPanel3,                javax.swing.GroupLayout.PREFERRED_SIZE,           72,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
.addGap(0, 0, Short.MAX_VALUE))
.addComponent(jPanel4,                javax.swing.GroupLayout.Alignment.TRAILING,
javax.swing.GroupLayout.DEFAULT_SIZE,                javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
.addContainerGap()
);

javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
getContentPane().setLayout(layout);
layout.setHorizontalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addComponent(jPanel1,                javax.swing.GroupLayout.DEFAULT_SIZE,           72,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
.addComponent(jPanel2,                javax.swing.GroupLayout.DEFAULT_SIZE,           63,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
);
layout.setVerticalGroup(

```

```
layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addGroup(layout.createSequentialGroup()
.addComponent(jPanel1,
            javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
.addComponent(jPanel2,
            javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
.addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
);

pack();
} // </editor-fold>

private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
System.exit(0);
}

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

this.jButton2.setEnabled(true);
this.jButton3.setEnabled(true);
this.jButton1.setEnabled(false);

JOptionPane.showMessageDialog(this, "Successfully Started....");
}

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {

try{
BufferedReader in = new BufferedReader(new FileReader("Queries20.txt"));
String line = in.readLine();
while (line != null) {
this.jTextArea1.append(line + "\n");
line = in.readLine();
}
}catch(Exception e){

}

JOptionPane.showMessageDialog(this, "Loaded with Queries for Filters....");
}

private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
JOptionPane.showMessageDialog(this, "Started Monitoring....");
Config.monitor = "Yes";
this.startUp();
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
/* Set the Nimbus look and feel */
```

```
//<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
/* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
 * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
 */
try {
    for (javax.swing.UIManager.LookAndFeelInfo info :
        javax.swing.UIManager.getInstalledLookAndFeels()) {
        if ("Nimbus".equals(info.getName())) {
            javax.swing.UIManager.setLookAndFeel(info.getClassName());
            break;
        }
    }
} catch (ClassNotFoundException ex) {
    java.util.logging.Logger.getLogger(ControlPanel.class.getName()).log(java.util.logging.Level.SEVERE, null,
ex);
} catch (InstantiationException ex) {
    java.util.logging.Logger.getLogger(ControlPanel.class.getName()).log(java.util.logging.Level.SEVERE, null,
ex);
} catch (IllegalAccessException ex) {
    java.util.logging.Logger.getLogger(ControlPanel.class.getName()).log(java.util.logging.Level.SEVERE, null,
ex);
} catch (javax.swing.UnsupportedLookAndFeelException ex) {
    java.util.logging.Logger.getLogger(ControlPanel.class.getName()).log(java.util.logging.Level.SEVERE, null,
ex);
}
//</editor-fold>

/* Create and display the form */
java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
        new ControlPanel().setVisible(true);
    }
});
}

// Variables declaration - do not modify
private javax.swing.JButton jButton1;
private javax.swing.JButton jButton2;
private javax.swing.JButton jButton3;
private javax.swing.JButton jButton4;
private javax.swing.JLabel jLabel1;
private javax.swing.JPanel jPanel1;
private javax.swing.JPanel jPanel2;
private javax.swing.JPanel jPanel3;
private javax.swing.JPanel jPanel4;
private javax.swing.JScrollPane jScrollPane1;
private javax.swing.JTextArea jTextArea1;
// End of variables declaration
}

Testform:
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
```

```
package testInjection;

import detectapp.Config;
import javax.swing.JOptionPane;

/**
 *
 * @author SSE
 */
public class TestLoginForm extends javax.swing.JFrame {

    /**
     * Creates new form TestLoginForm
     */
    public TestLoginForm() {
        initComponents();
    }

    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initComponents() {

        jLabel1 = new javax.swing.JLabel();
        jSeparator1 = new javax.swing.JSeparator();
        jLabel2 = new javax.swing.JLabel();
        jTextField1 = new javax.swing.JTextField();
        jLabel3 = new javax.swing.JLabel();
        jButton1 = new javax.swing.JButton();
        jPasswordField1 = new javax.swing.JPasswordField();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

        jLabel1.setFont(new java.awt.Font("Tahoma", 0, 18)); // NOI18N
        jLabel1.setText("Secure Login");

        jLabel2.setText("Enter Username: ");

        jLabel3.setText("Enter Password:");

        jButton1.setText("Login");
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });

        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                .addComponent(jSeparator1)

```



```
.addGroup(layout.createSequentialGroup())
.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addGroup(layout.createSequentialGroup())
.addGap(172, 172, 172)
.addComponent(jLabel1)
.addGap(0, 170, Short.MAX_VALUE))
.addGroup(layout.createSequentialGroup())
.addGap(20, 20, 20)
.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addComponent(jLabel2)
.addComponent(jLabel3))
.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addComponent(jTextField1)
.addComponent(jPasswordField1)))
.addGroup(javax.swing.GroupLayout.Alignment.TRAILING, layout.createSequentialGroup()
.addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
.addComponent(jButton1, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
.addContainerGap())
);
layout.setVerticalGroup(
layout.createSequentialGroup()
.addGroup(layout.createSequentialGroup()
.addGap(23, 23, 23)
.addComponent(jLabel1)
.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
.addComponent(jSeparator1, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
.addGap(18, 18, 18)
.addGroup(layout.createSequentialGroup()
.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
.addComponent(jLabel2)
.addComponent(jTextField1, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
.addGap(18, 18, 18)
.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
.addComponent(jLabel3)
.addComponent(jPasswordField1, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
.addGap(18, 18, 18)
.addComponent(jButton1)
.addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
);
pack();
} // </editor-fold>

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
// TODO add your handling code here:
OptionPane.showMessageDialog(null, Config.monitor);
if (Config.monitor.equals("Yes")) {
if (this.jPasswordField1.getText().contains("1=1")) {
OptionPane.showMessageDialog(null, "Your are supposed to passing sql injection stopped by Metasploit");
} else {
OptionPane.showMessageDialog(null, "Fine Everything is Ok You are Done");
}
}
```

```
    } else {
OptionPane.showMessageDialog(null, "Login Successful");
    }

}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    /* Set the Nimbus look and feel */
    //<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">
    /* If Nimbus (introduced in Java SE 6) is not available, stay with the default look and feel.
    * For details see http://download.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html
    */
    try {
        for (javax.swing.UIManager.LookAndFeelInfo info :
javax.swing.UIManager.getInstalledLookAndFeels()) {
            if ("Nimbus".equals(info.getName())) {
javax.swing.UIManager.setLookAndFeel(info.getClassName());
                break;

            }
        }
    } catch (ClassNotFoundException ex) {
java.util.logging.Logger.getLogger(TestLoginForm.class
.getName()).log(java.util.logging.Level.SEVERE, null, ex);
    }

    catch (InstantiationException ex) {
java.util.logging.Logger.getLogger(TestLoginForm.class
.getName()).log(java.util.logging.Level.SEVERE, null, ex);
    }

    catch (IllegalAccessException ex) {
java.util.logging.Logger.getLogger(TestLoginForm.class
.getName()).log(java.util.logging.Level.SEVERE, null, ex);
    }

    catch (javax.swing.UnsupportedLookAndFeelException ex) {
java.util.logging.Logger.getLogger(TestLoginForm.class
.getName()).log(java.util.logging.Level.SEVERE, null, ex);
    }
    //</editor-fold>

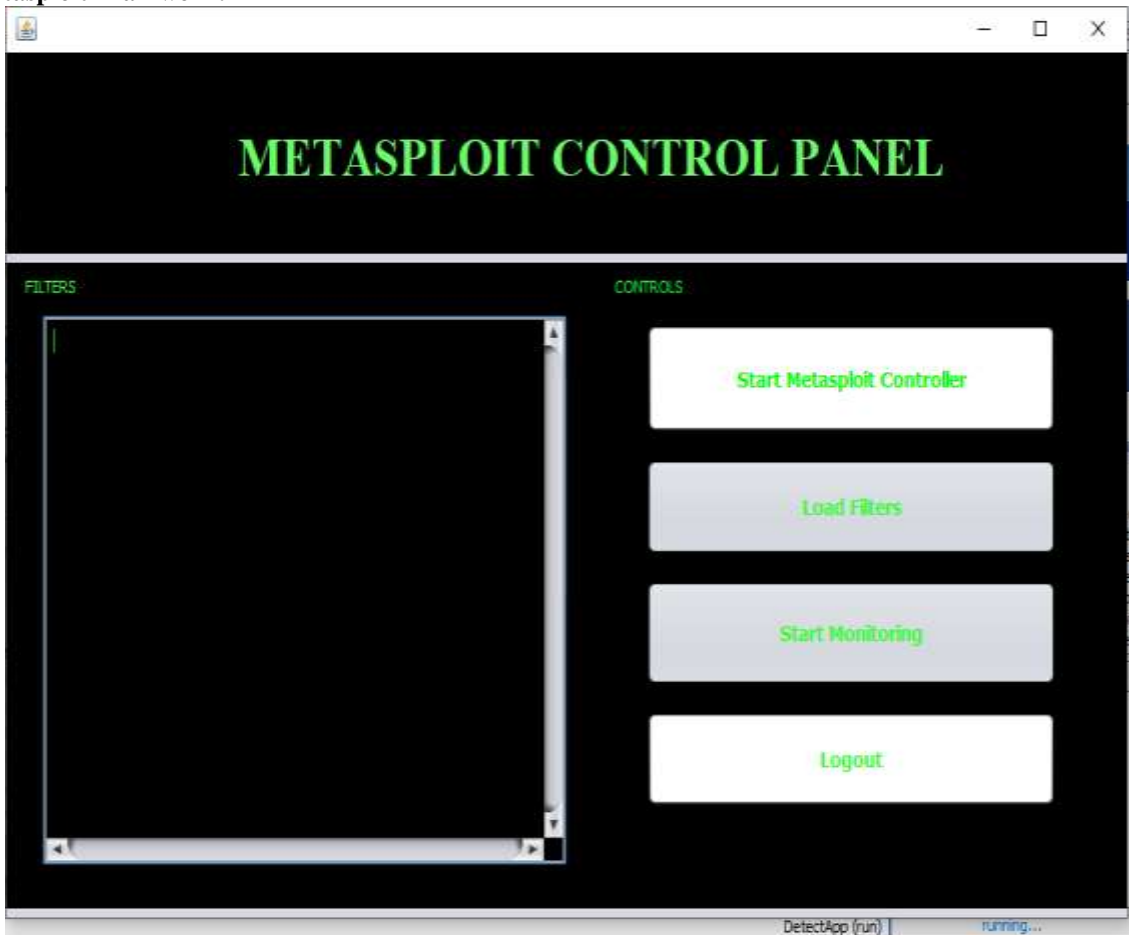
    /* Create and display the form */
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new TestLoginForm().setVisible(true);
        }
    })
}
```

```
});  
}  
  
// Variables declaration - do not modify  
private javax.swing.JButton jButton1;  
private javax.swing.JLabel jLabel1;  
private javax.swing.JLabel jLabel2;  
private javax.swing.JLabel jLabel3;  
public javax.swing.JPasswordField jPasswordField1;  
private javax.swing.JSeparator jSeparator1;  
private javax.swing.JTextField jTextField1;  
// End of variables declaration  
}
```

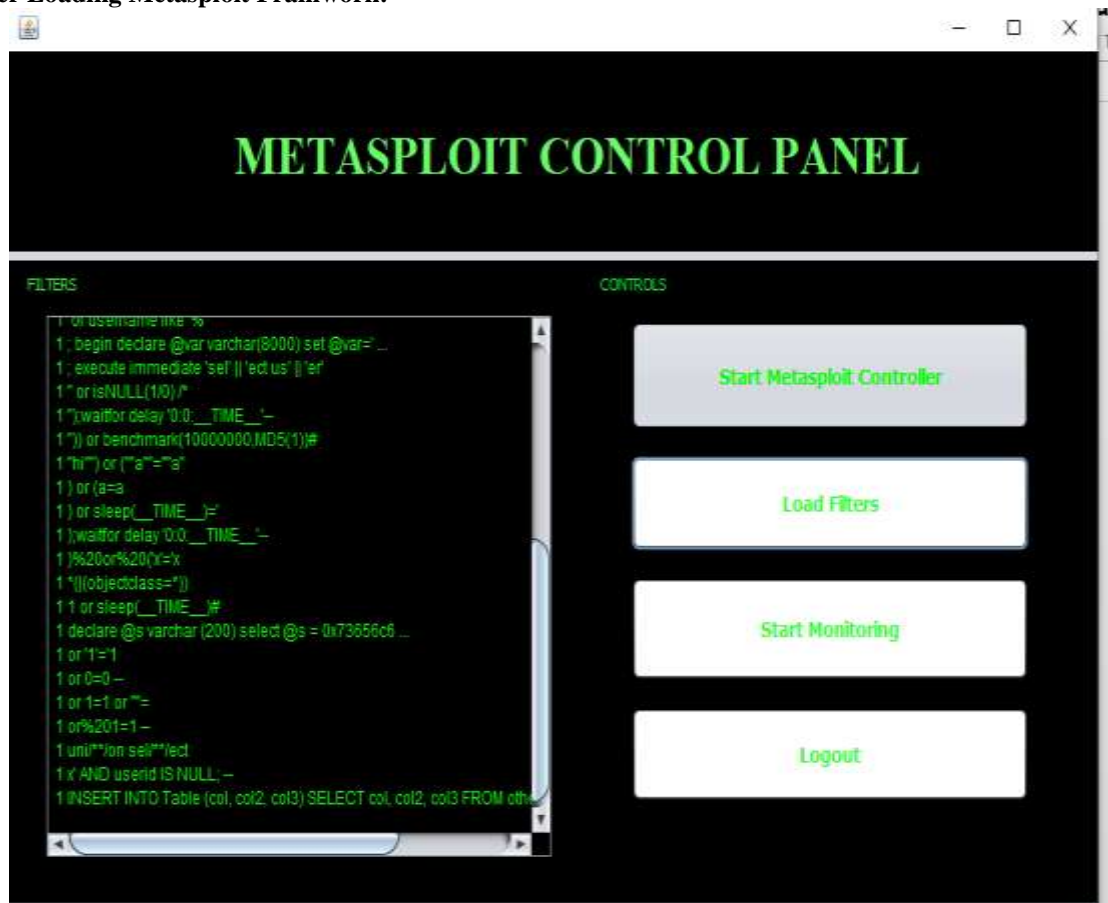
## APPENDIX-B (OUTPUT)

### Output

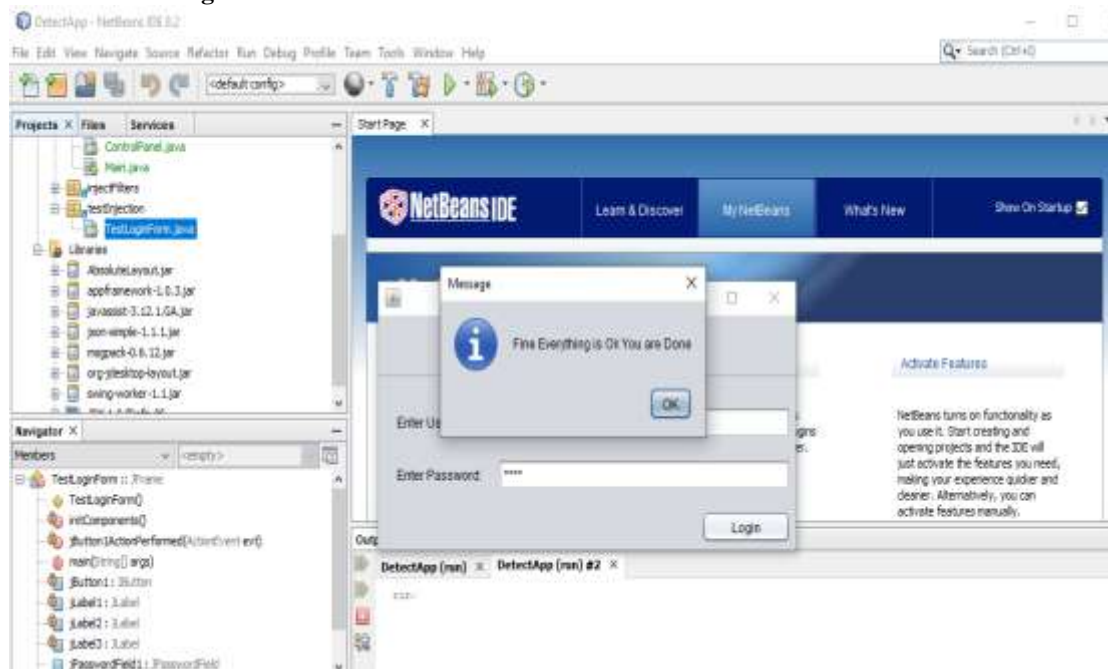
#### Metasploit Framework:



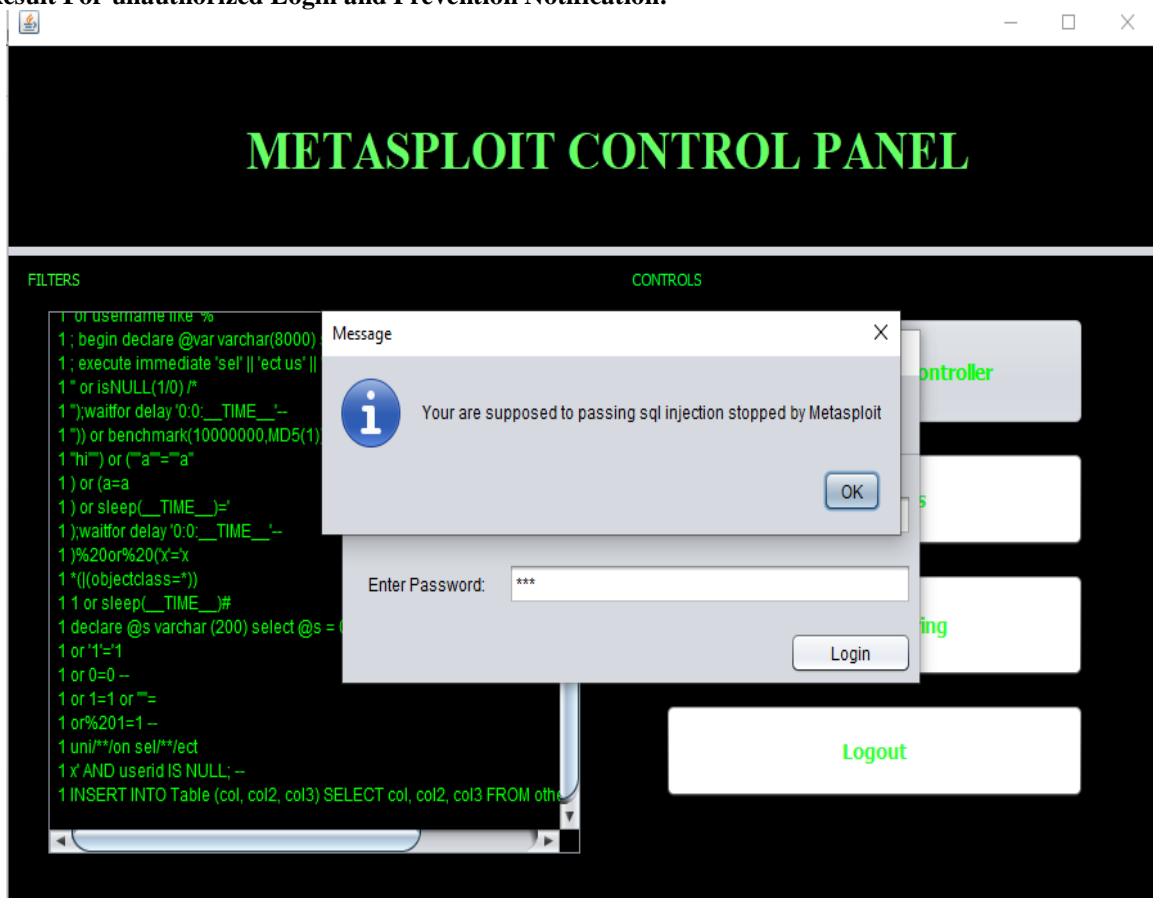
### Filter Loading Metasploit Framework:



### Authenticated User Login:



**Result For unauthorized Login and Prevention Notification:**



**ACKNOWLEDGEMENT**

I would like to express my deepest gratitude to the management of “AGNI COLLEGE OF TECHNOLOGY” and would like to thank our respected Principal **Dr.S. CHANDRAVADHANA** for his words of inspiration and for providing necessary facilities to carry out our project worksuccessfully.

I am immensely thankful to **Dr.S. SARAVANAN, M.E., Ph.D.**, Head of the Department, Information Technology for his words of wisdom and his constant source of inspiration.

I would like to offer my heartfelt thanks to my guide **Mrs. G Keerthana, M.E.**, Assistant Professor Department of Information Technology who molded me accordingly and gave valuable suggestions for completing my project work successfully.

We extend my warmest thanks to all the faculty members of my department for their assistance and I also thank all my friends who helped me in bringing out my project in good shape and form.

Finally, I express my sincere benevolence to my beloved parents for their perpetual encouragement and support in the entire endeavor.

**REFERENCES**

- [1]. Wei, K., Muthuprasanna, M., & Suraj Kothari. (2006, April 18). Preventing SQL injection attacks in stored procedures. Software Engineering IEEE Conference. Retrieved November 2, 2007, from <http://ieeexplore.ieee.org>
- [2]. Thomas, Stephen, Williams, & Laurie. (2007, May 20). Using Automated Fix Generation to Secure SQL Statements. Software Engineering for Secure Systems IEEE CNF. Retrieved November 6, 2007, from <http://ieeexplore.ieee.org>
- [3]. Merlo, Ettore, Letarte, Dominic, Antoniol & Giuliano. (2007 March 21). Automated Protection of PHP Applications Against SQL-injection Attacks. Software Maintenance and Reengineering, 11th European Conference IEEE CNF. Retrieved November 9, 2007, from <http://ieeexplore.ieee.org>
- [4]. Wassermann Gary, Zhendong Su. (2007, June). Sound and precise analysis of web applications for injection vulnerabilities.

- ACM SIGPLAN conference on Programming language design and implementation PLDI, 42 (6). Retrieved November 7, 2007, from <http://portal.acm.org>
- [5]. Friedl's Steve Unixwiz.net Tech Tips. (2007). SQL Injection Attacks by Example. Retrieved November 1, 2007, from <http://www.unixwiz.net/techtips/sql-injection.html>
- [6]. Massachusetts Institute of Technology. Web Application Security MIT Security Camp.
- [7]. Massachusetts Institute of Technology. Web Application Security MIT Security Camp. Retrieved November 1, 2007, from <http://groups.csmail.mit.edu/pag/readinggroup/wasserman07injection.pdf>
- [8]. Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. The Ohio State University Columbus, OH 43210 Using Parse Tree Validation to Prevent SQL Injection Attacks. Retrieved January 2005, from <http://portal.acm.org>
- [9]. Zhendong Su, Gary Wassermann. University of California, Davis. The Essence of Command Injection Attacks in Web Applications. Retrieved January 11, 2006, from <http://portal.acm.org>
- [10]. William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. College of Computing – Georgia Institute of Technology. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. Retrieved November 11, 2006, from <http://portal.acm.org>