

Modern Technique to Detect & Prevent SQL Injection

Ankit Kumar, Shravan Singh, S Sreeji

ankitkumar3110@gmail.com, Shravans821@gmail.com, sreeji.cse@gmail.com
Galgotias University, Greater Noida -201308

Abstract – As in today’s world most of the developers make mistakes during their web development phase. This leads the attacker to gain unauthorized access of company’s sensitive data. The common vulnerability through which an attacker gains access is SQLi. Web application is very good target for attacker to gain access to sensitive data. If developers fails to deploy protective measures for data protection the attacker will get the data. Developer should think beyond traditional security measures for complete data security in today’s world. The SQLi vulnerability affects any websites or web application. SQLi is one of the oldest and dangerous vulnerability of web application. . Bank and government organizations should take unique and extraordinary steps to protect themselves against SQLi vulnerability. We will use penetration testing to Detect SQLi vulnerability in web applications before Attack. The purpose of this paper is to make developers aware about SQLi attack vulnerability and understand how work attacks and further implement extra security regarding SQLi and protect their organizations from sensitive data leakage.

Keywords– SQL injection, Database security, Web application, SQLite, SQLIA (SQL injection attack) Internet, security of data.

I. INTRODUCTION

With the huge development of web technology there are more security issues. According to open web application security project (owasp), SQL (Structure Query Language) is first position in top 10. SQL injection is a web and application security vulnerability that allows an attacker to alter with the queries that an application makes to its database. It generally allows an attacker to view data like users all information that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior. In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure, or perform a denial-of-service attack.

II. THE PRINCIPLE OF DETECT SQL INJECTION

SQL Injection involves entering SQL code into web forms, eg. data fetch, login fields, or into the browser address field, to access and manipulate the information from the database behind the site.

1. SQLI Through Get Based

When you make a request input parameters sent in the URL of your browser. When your input text goes through browser url to validate authentication or verify to access information .Server takes your input and makes a sql query to fetch data and return you as result.

The Simple SQL Injection through query. In its simplest browser address field, this is how the SQL Injection works. it’s possible to understand this below scenario.

`http://test.target.com/user?id=1`

```
SELECT * FROM users WHERE id ='id' AND WHERE name ='xyz' {background}
```

Suppose Entering the \ as user id value results in showing some SQL query error.

Id field:
`1' --`

Web application authenticates users by verifying whether the returned result set is empty, and balances the query.

```
SELECT id ='1' -- ' WHERE name ='xyz' {background}
```

User can make SQL query between the ' -- and fetching other users information.

-- is the SQL convention for Commenting code, and everything after Comment is ignored. So the actual routine now becomes:

```
SELECT * FROM users WHERE id=' OR 1=1
```

via the URL. In terms of fetching data from database .If a hacker thinks a site is vulnerable, there are cheat-sheets all over the web for login strings which can gain access to

weak systems. Here are a couple more common strings which are used to dupe SQL validation routines:

```
' ORDER BY 1,2,3--  
' UNION ALL SECEL 1,2,3--  
' UNION ALL SELECT 1,database(),version()---
```

2. SQLI Through Post Based

In Post Based SQL injection on form field or data going through the body. When we enter the credentials it goes through the body this protects the man in middle attack. Web application authenticates users by verifying the user given credentials. Web application uses the input as result in memory to construct SQL statements for modifying user's input credentials. The authorization SQL query that is run by the server, the query which must be satisfied to allow access, will be something along the lines of:

```
SELECT * FROM users WHERE username = 'XYZ '  
AND password = 'ABC'
```

Where XYZ and ABC are what the user enters in the login fields of the web form.

So entering `OR 1=1 --` as your username, could result in the following actually being run:

```
SELECT * FROM users WHERE username = XYZ' OR  
1=1 -- 'AND password = 'ABC
```

Two things you need to know about this:
[] closes the [username] text field.

is the SQL convention for Commenting code, and everything after Comment is ignored. So the actual routine now becomes:

```
SELECT * FROM users WHERE username = '1 OR 1=1
```

1 is always equal to 1, last time I checked. So the authorization is now validated, if you make request 1 equal to 2 so authorization is not validated. via a form. In terms of login bypass via Injection, the hoary old ' OR 1=1 is just one option. If a hacker thinks a site is vulnerable, there are cheat-sheets all over the web for login strings which can gain access to weak systems. Here is some common query for Sql injection

username field examples:

```
admin'--  
' or ('a'='a  
" or ('a'='a  
) or ('a'='a  
) or ("a"="a
```

so familiar with using styles, don't worry; the template arranges everything for you in a user-friendly way.

ds, too, a solution to solve the problem is to modify the database interpreter, but this method is very complex.

3. SQLI Through Header Based

Some web application store users headers information into their database. like the user's user-agent ,referrer and x-forwarded-host HTTP header fields are components of the message header of requests and responses in the Hypertext Transfer Protocol (HTTP). They define the operating parameters of an HTTP transaction.

Example:

```
GET /index.php HTTP/1.1  
Host: host  
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:77.0)  
Gecko/20100101 Firefox/77.0  
Accept:  
text/html,application/xhtml+xml,application/xml;q=0.9,  
image/webp,*/*;q=0.8  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Referer: referer  
Connection: close  
Cookie: infosec_post_order=ORDER_DATE_DESC;  
infosec_post_order=ORDER_DATE_DESC  
Upgrade-Insecure-Requests: 1  
Cache-Control: max-age=0
```

X-Forwarded-For:

This header for identifying the originating IP address of a client connecting to a web server through an HTTP proxy or a load balancer.

```
$request = mysql_query("SELECT user,password FROM  
admins WHERE user='".sanitize($_POST['user'])."' AND  
password='".md4($_POST['password'])."' AND  
ip_adr='".ip_addr()."");
```

The variable login is correctly controlled due to the sanitize() method.

```
function sanitize($param){ if (is_numeric($parameter)) {  
return $parameter; } else { return  
mysql_real_escape_string($parameter); } }
```

Let us inspect the ip variable. It is allocating the output of the ip_addr() method.

```
function ip_addr() { if  
(isset($_SERVER['HTTP_X_FORWARDED_FOR'])) {  
$ip_addr =  
$_SERVER['HTTP_X_FORWARDED_FOR']; } else {  
$ip_addr = $_SERVER["REMOTE_ADDR"]; } if  
(preg_match("#^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-  
9]{1,3}#", $ip_addr)) { return $ip_addr; } else { return  
$_SERVER["REMOTE_ADDR"]; } }
```

Obviously, the IP address is retrieved from the HTTP header X_FORWARDED_FOR. This layer is controlled by the preg_match which verifies if this parameter does hold at least one IP address. As a matter of fact, the environment of variables HTTP_X_FORWARDED_FOR is not properly sanitized before its value being used in the SQL query. This can lead to fire any SQL query by injecting arbitrary SQL code into this field.

The simple modification of this header field to something like:

```
GET /index.php HTTP/1.1
Host: [host]
X_FORWARDED_FOR :127.0.0.1' or 1=1#
```

User-agent:

The User-Agent request header provides information about browsers and also identifies the application, operating system, vendor, and/or version of the requesting user agent in client and server communication.

HTTP query example:

```
GET /index.php HTTP/1.1
Host: host
User-Agent: aaa' OR 1=1#
```

Referer:

This request header contains the address of the previous web page from which request is generated. The Referer header allows servers to identify where people are visiting before this current request and may use that data for analytics, logging, or optimized caching.

```
GET /index.php HTTP/1.1
Host: host
User-Agent:
Referer: admin' OR 1=1#
```

4. SQLI Through Cookie Based

Cookie header is used to submit additional parameters that the server has issued to the client. cookies are not handled by users. Outside of session cookies which are random, cookies may contain data in clear or encoded in hexadecimal, base64, hashes (MD5, SHA1), serialized information. If we can determine the encoding used, we will attempt to inject SQL commands.

```
function is_user($user) {
    global $prefix, $db, $user_prefix;
    if(!is_array($user)) {
        $user = base64_decode($user);
        $user = explode(":", $user);
        $suid = "$user[0]";
        $pwd = "$user[2]";
    } else {
```

```
        $suid = "$user[0]";
        $pwd = "$user[2]";
    }
    if($suid != "" AND $pwd != "") {
        $sql = "SELECT user_password FROM
        ".$user_prefix."_users WHERE user_id='suid'";
        $result = $db->sql_query($sql);
        $row = $db->sql_fetchrow($result);
        $pass = $row[user_password];
        if($pass == $pwd && $pass != "") {
            return 1;
        }
    }
    return 0;
}
```

Example:

```
GET /index.php HTTP/1.1
Host: host
User-Agent: Mozilla/5.0 (X11; Linux
Cookie: sessionid=xyz' OR 1=1#
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```



Fig. 1 SQL injection detect system

5. Database Keywords

The keywords for each SQL query is shown in Table 1, each type of SQL statement has a primary keyword which is formatted by bold font. Due to the WHERE expression in Insert, Delete, Update statement can contain subqueries, these types of SQL statements should involve the query keywords in addition to its own keywords.

Table 1 Sql Keywords

TYPE	KEYWORDS
Query	SELECT , FROM, WHERE, ORDER BY, GROUP BY, HAVING, UNION, OR, AND
Insert	INSERT , INTO, VALUES, the query keywords
Delete	DELETE , FROM, WHERE, the query keywords
Update	UPDATE , SET, WHERE, the query keywords
Table operator	CREATE TABLE , ALTER TABLE , DROP TABLE

3.2 Types OF Sql Injection

SQL injections divided mainly three categories: In-band SQLi (Classic), Inferential SQLi (Blind) and Out-of-band SQLi.

IV. IN-BAND SQLI (CLASSIC)

In-band SQL Injection is the most common and simple to exploit of SQL Injection attacks. In-band SQL Injection occurs when an attacker is able to use the same communication channel to launch the attack and gather results both. There are two types of in-band SQL Injection.

1. Error Based SQL Injection:

Error-based SQLi is an in-band SQL Injection technique that relies on error messages thrown by the database server to obtain information about the structure of the database. When insert malicious query as input.

For example, SQL syntax error should be like this: Warning: mysql_fetch_array() expects parameter 1 to be resource, boolean given in /home1/aquacity/public_html/page.php on line 6 The error message gives information about the database used, where the syntax error occurred in the query. Error based technique is the easiest way to find SQL Injection.

2. Union-based Query:

Union-based SQLi is an in-band SQL injection technique that leverages the UNION SQL operator to combine the results of two or more SELECT statements into a single result which is then returned as part of the HTTP response. This technique takes advantage of the UNION SQL operator. The UNION operator is used for combining multiple tables to select queries at the same time. In union operators, they remove duplicate rows or columns from the table which we try to execute at the same time.

V. INFERENTIAL SQLI (BLIND SQLI)

Inferential SQL Injection may take longer for an attacker to exploit, however, it is also dangerous like other sql injection. In an inferential SQLi attack, no data is actually transferred or displayed via the web page and due to which the attacker would not be able to see the result of an attack in-band it is also called blind injection. In which attacker make payload and fire then observing the web application's response and the resulting behavior of the database server. There are two types of inferential SQL Injection are:

1. Boolean:

Boolean-based SQL Injection is an inferential SQL Injection technique in which an attacker sends a malicious combine payload of SQL Injection to the database. which forces the application to return a different result depending on whether the query returns a TRUE or

FALSE result. Based on the result of the query, the information within the HTTP response will modify or stay unchanged. The attacker can then work out if the message generated a true or false result.

2. Time-based:

Time-based SQL Injection is an inferential SQL Injection technique that an attacker sends an SQL query to the database which forces the database to wait for a specified amount of time (in seconds) before responding. The response time will indicate to the attacker whether the result of the query is TRUE or FALSE. Based on the result, an HTTP response will be generated instantly or after a waiting period. The attacker can thus work out if the message they used returned true or false, without relying on data from the database.

VI. OUT-OF-BAND SQLI

Out-of-band techniques offer an attacker an alternative to inferential time-based techniques, especially if the server responses are not very stable (making an inferential time-based attack unreliable). The attacker can only carry out this form of attack when certain features are enabled on the database server used by the web application. This form of attack is primarily used as an alternative to the in-band and inferential SQLi techniques.

VII. SQLI PREVENTION AND MITIGATION

It's somewhat shameful that there are so many successful SQL Injection attacks occurring, because it is EXTREMELY simple to avoid SQL Injection vulnerabilities in your code. Despite all its different manifestations, and the complexities that can arise in its very easy exploitation. The user, provided credentials during the user registration. This unique fingerprint of the user is stored against his access credentials in the database.

1. Parameterized Statements

The use of Parameterized statements with variable binding is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied. Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker. In the safe example below, if an attacker were to enter the userID of tom' or '1'=1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'=1.

Language specific recommendations:

1. Java EE –	use Prepared Statement() with bind variables
2. .NET –	use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
3. PHP –	use PDO with strongly typed parameterized queries (using bindParam())
4. Hibernate –	use createQuery() with bind variables (called named parameters in Hibernate)
5. SQLite –	use sqlite3_prepare() to create a statement object

Programming languages talk to SQL databases using database drivers. A driver allows an application to You should always use parameterized statements where available, they are your number one protection against SQL injection.

2. Stored Procedures

Stored procedures are not always safe from SQL injection. However, certain standard stored procedure programming constructs have the same effect as the use of parameterized queries when implemented safely which is the norm for most stored procedure languages.

There are many cases where stored procedures can not be secure. For example, on MS SQL server, you have 3 main default roles: db_datareader, db_datawriter and db_owner. Before stored procedures came into use, DBA's would give db_datareader or db_datawriter rights to the webservice's user, depending on the requirements. However, stored procedures require execute rights, a role that is not available by default.

3. Escaping Inputs

This technique is to escape user input before putting it in a query. It is very database specific in its implementation. It's usually only recommended to retrofit legacy code when implementing input validation isn't cost effective. Applications built from scratch, or applications requiring low risk tolerance should be built or re-written using parameterized queries, stored procedures, or some kind of Object Relational Mapper (ORM) that builds your queries for you. Injection attacks often rely on the attacker being able to craft an input that will prematurely close the argument string in which they appear in the SQL statement. (This is why you will often see ' or " characters in attempted SQL injection attacks.) Escaping symbol characters is a simple way to protect against most SQL injection attacks, and many languages have standard functions to achieve this. There are a couple of drawbacks to this approach, however:

You need to be very careful to escape characters everywhere in your codebase where an SQL statement is constructed. Not all injection attacks rely on abuse of quote characters. For example, when a numeric ID is expected in a SQL statement, quote characters are not required. The following code is still vulnerable to injection attacks, no matter how much you play around with quote characters:

4. Sanitizing Inputs

Sanitizing inputs is a good practice for all applications. Various parts of SQL queries aren't legal locations for the use of combine variables, such as the names of tables or columns, and the sort order indicator (ASC or DESC). In such situations, input validation or query redesign is the most appropriate defense. For the names of tables or columns, ideally those values come from the code, and not from user parameters.

But if user parameter values are used for targeting different table names and column names, then the parameter values should be mapped to the legal/expected table or column names to make sure unvalidated user input doesn't end up in the query. Please note, this is a symptom of poor design and a full rewrite should be considered if time allows. In our example hack, the user supplied a password as ' or 1=1--, which looks pretty suspicious as a password choice. Developers should always make each variable that rejects inputs that look suspicious in query,. For instance, your application may clean parameters supplied in GET and POST requests in the following ways: Check that supplied fields like email addresses match a regular expression. Ensure that numeric or alphanumeric fields do not contain symbol characters. Reject (or strip) out whitespace and new line characters where they are not appropriate.

VIII. CONCLUSION AND FUTURE WORK

This report discussed web application security principles and fundamental information that can help us to prevent web exploits in our system. Web applications are considered the most exposed and least protected, thereafter vulnerable because the standards somehow are not focused on security but more on the server's need for functionality. Security threats are more common than before because the internet has become today's economy the most valuable tool for everyone. So there is indeed a need to protect our resources, data and user privacy information. As technology moves forward and brings new strategies, tools, models and methods to increase security levels, hackers will be part of this never end game. The proposed system is developed to detect the vulnerabilities like SQLi, Cross Site Scripting, Local and Remote File Inclusion, Command Injection in web applications and it will also provide information about remediation of vulnerable URL and its vulnerability. Our

formalization goes at the heart of the problem and captures seemingly different types of above mentioned vulnerabilities. The simple and effective strategy is meant to be cost effective and is openly targeted toward large commercial applications. The results of the proposed systems are satisfactory.

REFERENCES

- [1]. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- [2]. YAN L, LI X H, FENG R T et al 'Detection method of the second-order SQL injection in Web applications[J] 'Lecture Notes in Computer Science' 2014' 8332. 154-165'.
- [3]. TIAN Y J, ZHAO Z M ' ZHANG H C ' et al 'Second-order SQL injection attack defense model[J] ' Netinfo Security, 2014, (11) 70-73' Sqli-labs
- [4]. Project [EB/OL]. <https://github.com/Audi-1/sqli-labs>.
- [5]. Protecting Against SQL Injection <https://www.hacksplaining.com/prevention/sql-injection>
- [6]. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html