Blindfolded **SQL Injection**

Written By:

Ofer Maor
Amichai Shulman

WEBcohort

WEBcohort

Table of Contents

Overview

In the past few years, SQL Injection attacks have been on the rise. The increase in the number of Database based applications, together with various publications that explain the problem and how it can be exploited (in both electronic and printed formats), have led to many attacks and abuse of this type of attack.

Following the increase in attacks taking advantage of SQL Injection, many attempts have been made to find solutions to the problem. The obvious solution, of course, is, and always will be, to build the programs in a secure manner. Many documents have been published regarding secure development of Web applications with emphasis on Database access, yet not much has changed. Web developers are still, usually, not security aware, and the problems continue to appear.

As a result, security experts keep looking for other measures that can be taken against this problem. Unfortunately, the common solution to this problem took form in suppressing the detailed error messages. Since most documents describing SQL Injection rely on gathering information through the error messages (some even claim that specific tasks cannot be completed without detailed error messages), security experts developed a notion that SQL Injection cannot really be exploited without detailed error messages (or the source code itself).

Hiding the error messages, however, is just another implementation of the "Security by Obscurity" approach, which has been repeatedly proven through history as a flawed approach. The purpose of this document is to refute the notion that SQL Injection can only be exploited with detailed error messages, and to present simple techniques used by attackers when no detailed error messages are present. These techniques all come under the name 'Blindfolded SQL Injection'. The name reflects the feeling that one may get when trying to do SQL Injection without detailed error messages. It also reflects that with the right skill, even SQL Injection becomes so simple that it can be done blindfolded.

To understand how this is achieved, we first show how SQL Injection can be easily identified, based on minimal reaction of the server. Then, we go over ways to craft a valid syntax request, which can be replaced later on with any valid SQL request. Eventually, we discuss how UNION SELECT statements (often considered the highlight of SQL Injection attacks) can be exploited with no detailed error messages. Being blindfolded, the techniques in this document assume that we have zero knowledge of the application, type of database, structure of the tables, etc, and that these need to be detected throughout the injection process.

By this, we hope to make it clear that application level vulnerabilities must be handled by application level solutions, and that relying on suppressed error messages for protection from SQL Injection is eventually useless.

Two additional important notes: First, this document is not an SQL Injection guide, nor an SQL Tutorial. This document does not go into the details of specific SQL Injection attacks and exploitation, and assumes that the reader has basic understanding of what SQL Injection attacks are, and wants to understand how these can be hazardous against an application that does not provide detailed error messages. The second note regards the examples provided throughout this white paper. Although the provided examples refer to MS SQL Server and Oracle only, the same techniques can be applied to other Databases as well.

Identifying Injections

To make an SQL Injection work, the first step, obviously, is to identify it. To do that, the attacker must first establish some sort of indication regarding errors in the system. Although the error messages themselves are not being displayed, the application should still have some capability of separating right (a valid request) from wrong (an invalid request), and the attacker easily learns to identify these indications, find the errors and identify whether they are SQL related or not.

Recognizing Errors

First, we must understand the types of errors that an attacker can face. A Web application can generate errors of two major types. The first type of error is that generated by the Web server, as a result of some exception in the code. If untouched, these exceptions yield the all too familiar '500: Internal Server Error'. Normally, injection of bad SQL syntax (unclosed quotes, for instance), should cause the application to return this type of error, although other errors may lead to such an exception. A simple error suppression process will replace the default texts of this error with a custom-made HTML page, but observing the response line itself will reveal the fact that it is still a server error. In other cases, more effort is taken to suppress the errors, and the erroneous response may simply be a redirect to the main/previous page, or a generic error message which does not provide any information.

The second type of error is generated by the application code, and usually indicates better programming. The application expects certain invalid cases, and can generate a specific customized error for them. Although normally these types of errors should come as part of a valid (200 OK) response, they may also be replaced with redirects or other means of concealing, much like the 'Internal Server Error'.

A simple example will differentiate between the two: Let's take two similar eCommerce applications, named A and B. Both applications are using a page called proddetails.asp. This page expects to receive a parameter called ProdID. It takes the received ProdID, and retrieves the product details from the Database, then performs some manipulations over the returned record. Both applications only call proddetails.asp through a link, therefore ProdID should always be valid. Application A is satisfied with this, and does no additional checks. When an attacker tampers with ProdID, inserting an id that has no row in the table, an empty recordset will be returned. Since application A does not expect an empty recordset, when it tries to manipulate the data in the record, an exception is likely to occur, generating a '500: Internal Server Error'. Application B, however, verifies that the recordset size is larger than 0 before any manipulation of it. If it is not, an error appears claiming 'No such

product', or, if the programmer wants to hide the error, the user is simply presented back with the product list.

An attacker attempting to perform Blindfolded SQL Injection would therefore try, at first, to generate a few invalid requests, and learn how the application handles errors, and what could be expected of it when an SQL error occurs.

Locating Errors

With that knowledge of the application at hand, the attacker can now proceed to the second part of the attack, which is locating errors that are a result of manipulated input. For that, normal SQL Injection testing techniques are applied, such as adding SQL keywords (OR, AND, etc.), and META characters (such as ; or '). Each parameter is individually tested, and the response is closely examined to determine whether an error occurred. Using an intercepting proxy or any other tool of choice, it is easy to identify redirects and other supposedly hidden errors. Each parameter that returns an error is suspicious, as it may be vulnerable to SQL Injection.

As always, all parameters are tested individually, with the rest of the request being valid. This is extremely important in this case, as this process must neutralize any possible cause of error other than the injection itself. The result of this process is usually a long list of suspicious parameters. Some of these parameters may indeed be vulnerable to SQL Injection and may be exploited. The others had errors that are unrelated to SQL, and can be discarded. The next step for the attacker is therefore identifying the pick of the litter, which, in our case are those that are indeed vulnerable to SQL Injection.

Identifying SQL Injection Vulnerable Parameters

To better understand how this is done, it is important to understand the basic types of data in SQL. SQL fields can normally be classified as one of three main types: Number, String or Date. Each main type has many different flavors, but these are irrelevant for the injection process. Each parameter transferred from the web application to the SQL query is considered as one of these types, and it is usually very simple to determine the type ('abc' is obviously a string, whereas 4 is likely to be an number, although it must be considered as a string as well).

In the SQL language, numeric parameters are passed to the server as is, whereas strings or dates are passed with quotes around them. For example:

```
SELECT * FROM Products WHERE ProdID = 4
```

vs.

```
SELECT * FROM Products WHERE ProdName = 'Book'
```


The SQL server, however, does not care what type of an expression it receives, as long as it is indeed of the relevant type. This behavior gives the attacker the best way of identifying whether an error is indeed an SQL one or unrelated. With numeric values, the easiest way to handle this is by using basic arithmetic operations. For instance, let's look at the following request:

```
/myecommercesite/proddetails.asp?ProdID=4
```

Testing this for SQL Injection is very simple. One attempt is done by injecting `4'` as the parameter. The other is done using `3 + 1` as the parameter. Assuming this parameter is indeed passed to an SQL request, the result of the two tests will be the following two SQL queries:

```
(1)   SELECT * FROM Products WHERE ProdID = 4'
(2)   SELECT * FROM Products WHERE ProdID = 3 + 1
```

The first one will definitely generate an error, as this is bad SQL syntax. The second, however, will execute smoothly, returning the same product as the original request (with 4 as the ProdID), indicating that this parameter is indeed vulnerable to SQL Injection.

A similar technique can be used for replacing the parameter with an SQL syntax string expression. There are only two differences. First, string parameters are held inside quotes, so breaking out of the quotes is necessary. Secondly, different SQL servers use different syntax for string concatenation. For instance, Microsoft SQL Server uses the `+` sign to concatenate string, whereas Oracle uses `||` for the same task. Other than that, the same technique is used. For instance:

```
/myecommercesite/proddetails.asp?ProdName=Book
```

Testing this for SQL Injection involves replacing the ProdName parameter, once with an invalid string such as `B'`, the other with one that will generate a valid string expression, such as `B' + 'ook` (or `B' || 'ook` with Oracle). This results with the following queries:

```
(1)   SELECT * FROM Products WHERE ProdName = 'Book''
(2)   SELECT * FROM Products WHERE ProdID = 'B' + 'ook'
```

Again, the first query is likely to generate an SQL error, while the second is expected to return the same product as the original request, with `Book` as its value.

Similarly, any other expression can be used to replace the original parameters. Specific system functions can be used to return either a number, a string or a date (for instance, in Oracle, *sysdate* returns a date expression, whereas in SQL Server *getdate()* does the same task). Other techniques can also be used to determine whether SQL Injection occurs.

As can be seen, identifying whether SQL Injection occurs is a very simple task even without detailed error messages, allowing the attacker to easily continue with the attack.

Performing the Injection

Once the injection has been identified by the attacker, the next step will be trying to exploit it. For that, the attacker must be able to generate valid syntax, identify the specific Database Server, and build the required exploit.

Getting the Syntax Right

This is usually the trickiest part in the Blindfolded SQL Injection process. If the original queries are simple, this is simple as well. However, if the original query was complex, breaking out of it may require a lot of trial and error. In any case, only a few basic techniques are required to perform these tests.

The basic syntax identification process goes for standard SELECT … WHERE statements, with the injected parameter being part of the WHERE clause. In order to get a valid syntax, the attacker must be able to append data to the original WHERE statement so that it will return different data than it should. In simple applications, simply adding `OR 1=1` can often do the trick. In many cases, however, this will not be sufficient for a successful exploit. Often, parenthesis must be closed, so they match the originally opened ones. Another problem that may occur is that a tampered query will cause the application to generate an error, which cannot be distinguished from an SQL error (for instance, if only one record is expected, and OR 1=1 caused the Database to return 1000 records, the application may generate an error).

Since each WHERE clause is basically a set of expressions evaluating as True or False, joined together with OR, AND and parenthesis, learning the right syntax that breaks out of parenthesis and properly terminates the query is done by attempting different combinations. For instance, adding 'AND 1=2' turns the entire phrase to a false one, whereas adding 'OR 1=2' has zero influence, except for operator precedence.

With some injections, simply altering the WHERE clause may suffice. With others, such as UNION SELECT Injection, or stored procedures injections, it is not enough to change the WHERE clause. The entire SQL statement must be properly terminated, so that additional syntax can be appended. For that, a very simple technique can be used. After the attacker establishes a valid combination of AND, OR, 1=2, and 1=1 expressions, the SQL comment sign can be used.

This sign, represented by two consecutive dashes (--), instructs the SQL server to ignore the rest of input in the line. For instance, let us look at a simple login page, taking both the Username and Password into the query, such as this one:

```
SELECT Username, UserID, Password FROM Users
WHERE Username = 'user' AND Password = 'pass'
```

By sending `johndoe' --` as the user, the following WHERE clause is generated:

```
WHERE Username = 'johndoe' --'AND Password = 'pass'
```

In this case, not only the syntax was right, but the authentication was bypassed. However, let us look at a slightly different WHERE statement:

```
WHERE (Username = 'user' AND Password = 'pass')
```

Notice the parenthesis around the statement. With the same injection (`jonhdoe' --`), the query will now fail:

```
WHERE (Username = 'johndoe' --' AND Password = 'pass')
```

The query has unmatched parenthesis, and can therefore not be executed.

This example, therefore, demonstrates how the comment sign can be used to identify whether the query has been properly terminated. If the comment sign was added and no error occurred, it means that it terminated properly right before the comment. Otherwise, additional trial and error is required.

Identifying the Database

The next step the attacker needs to take, just before starting to exploit the SQL Injection, is to identify the specific database used. Luckily (for the attacker, at least), this is a much easier task than finding the valid syntax.

Several simple tricks allow the attacker to identify the Database type, all based on differences which exist between specific implementations of Database engines. The following examples focus on differentiating between Oracle and Microsoft SQL Server. Similar techniques, however, are easy to use to identify other Database engines.

A very simple trick, which was mentioned earlier, is the string concatenation difference. Assuming the syntax is known, and the attacker is able to add additional expressions to the WHERE clause, a simple string comparison can be done using this concatenation, for instance:

```
AND 'xxx' = 'x' + 'xx'
```

By replacing the `+` with `||`, Oracle can be easily differentiated from MS SQL Server, or other Databases.

Another technique uses the semicolon character. In SQL, a semicolon is used to chain several SQL statements in the same line. While with SQL Injection this can be used inside the injection code, the Oracle drivers do not allow use of semicolons in this manner. Assuming the comment is properly working (meaning, it generates no error), adding a semicolon before it has no influence on an MS SQL Server, yet generates an error with Oracle. Additionally, verifying whether additional commands can be executed after the semicolon can be done by adding the COMMIT statement (for instance, injecting `xxx' ; COMMIT --`). Assuming statements can be injected there, this should not generate an error.

Eventually, some expressions can be replaced with system functions which return the right value. Since each database uses slightly different functions, it is easy to identify the database type this way (much like the above-mentioned date functions, *getdate()* with MS SQL Server vs. *sysdate* with Oracle).

Exploiting the Injection

With all the relevant information at hand, the attacker can now proceed to perform the injection itself. While building the exploit code, the attacker no longer needs information from the error messages, and can build the exploits based on all the known SQL Injection techniques.

This paper does not discuss normal SQL Injection exploitation techniques, as they have already been discussed in detail in other papers. The only exploitation technique further discussed in this document is injection of UNION SELECT statements, which is described in the next section.

UNION SELECT Injections

Although tampering with SELECT … WHERE statements can be very rewarding in many applications, attackers often want to be able to perform a UNION SELECT injection. Unlike WHERE clauses manipulation, successfully performing a UNION SELECT injection gives the attacker access to all tables in the system, which may not be otherwise accessible.

Since performing a UNION SELECT statement requires knowledge of the number of fields in the query as well as the type of each field, it is often considered that it cannot be performed without detailed error messages, especially when the number of fields in the original query is large. The following section is going to refute this notion, by showing a set of very simple techniques that easily solve this problem.

Before proceeding, it is obvious that the attacker must have the correct syntax. The previous sections, however, have already established that this is possible, and demonstrated how this is done. The only important note regarding the syntax is that with UNION SELECT the syntax identification **must** be to the level that clears out all parenthesis of the request, and allows free injection of the UNION or other commands. (As explained before, this can be verified by appending the comment sign to the injection).

Once the syntax is right, a UNION SELECT statement can be appended to the original query. The UNION SELECT statement must have the same number of columns, and the same types of columns as the original statement, or an error is generated.

Counting the Columns

Counting the Columns can appear to be nearly impossible with the normal techniques of 'UNION SELECT' SQL Injection. The way this is done, with detailed error messages, is by simply creating a UNION SELECT injection, and trying it with a different number of fields (one more in each attempt). When the error indicating a 'column number mismatch' is replaced with a 'column type mismatch', the right number of columns has been reached, and the next step can be taken. When working blindfolded, however, we have absolutely no indication regarding the type of the error, and therefore this method is completely futile.

A different technique must therefore be used to identify the number of the columns, and this technique comes in the form of the ORDER BY clause. Adding an ORDER BY clause to the end of the SELECT statement changes the order of the results in the record-set. This is normally done by specifying the name of a sort column (or the names of several sort columns).

Looking at an example of a financial query, a valid injection into the Credit Card number parameter would be `11223344) ORDER BY CCNum --`, resulting in the following query:

```
SELECT CCNum FROM CreditCards
WHERE (AccNum=11223344) ORDER BY CCNum --
AND CardState='Active') AND UserName='johndoe'
```

What is usually overlooked, however, is the fact that the ORDER BY clause can have a numeric form. In this case the expression refers to a column number rather than its name. This means that injecting `11223344) ORDER BY 1 --` will also be legitimate, and would do exactly the same, since `CCNum` is the first field in the result of the query. Injecting `11223344) ORDER BY 2 --`, however, will generate an error as this query has only one field, meaning the result cannot be sorted by its second field.

Therefore, when coming to count the number of fields, ORDER BY can be very useful. First, the attacker adds an `ORDER BY 1` clause to the basic syntax he identified. Since every SELECT query must have at least one field, this should work. If an error is received on that, the syntax should be tampered with some more, until it no longer appears. (Although unlikely, it may also be that the sorting causes an error in the application. In this case, adding ASC or DESC may solve the problem). Once a valid syntax containing the ORDER BY works with no errors, the attacker changes the ordering from column 1 to column 100 (or 1000 or anything that is certain to be invalid). At this point, an error should be generated, indicating that the enumeration is working.

The attacker now has a method of identifying which column numbers exist and which do not, and can easily identify the exact number of columns. The attacker simply needs to increase this number, one at a time, until an error is received (since some columns may be of a type that does not allow sorting, it is always advisable to test one or two additional numbers, and make sure that indeed errors are received). With this technique the number of fields is easily enumerated, and no error messages are required.

Identifying Columns Types

So, with the valid syntax already known, the Database vendor determined, and the number of fields enumerated, all that is left for the attacker is to identify the types of all the fields.

Getting the field types right can be tricky though. They must **all** match the original query. If it's just a few fields, this can easily be achieved by brute force, but if there are more, then a problem arises. As already mentioned, there are 3 possible main types (number, string, date), so having 10 fields means there are 3^{10} (nearly 60,000) combinations. At 20 requests per second, this will take almost an hour. Having more fields than that makes the entire process nearly impossible.

An easier technique must therefore be used when working in the dark. This comes in the form of the NULL keyword in SQL. Unlike injection of static fields, which are of a specific type (such as a string or an integer), NULL can match any type. It is therefore possible to inject a UNION SELECT statement where all fields are null, and there should be no type mismatch errors. Let us look at a query similar to the previous example:

```
SELECT CCNum,CCType,CCExp,CCName FROM CreditCards
WHERE (AccNum=11223344 AND CardState='Active')
       AND UserName='johndoe'
```

The only change is that the single CCNum field was replaced with several, so there are more fields. Assuming the attacker successfully counted the number of columns of the result of this query (4 in our example), it is now simple to inject a UNION statement with all NULLs, and having a FROM clause that does not generate permission errors (again, trying to isolate each problem, so the specific permissions issues will be handled later on). With MS SQL, the FROM clause can be simply omitted. This is valid syntax. With Oracle, using a table named *dual* can be useful. Adding a WHERE statement that always evaluates as false (such as WHERE 1=2) guarantees that no record-set containing only null values will be returned, eliminating further possible errors (some applications may not properly handle NULL values).

Let's now look at the MS SQL Server example, although the same applies to Oracle. Injecting `11223344) UNION SELECT NULL,NULL,NULL,NULL WHERE 1=2 --,` results in the following query:

```
SELECT CCNum,CCType,CCExp,CCName FROM CreditCards
WHERE (AccNum=11223344) UNION SELECT NULL,NULL,NULL,NULL
WHERE 1=2 --AND CardState='Active') AND UserName='johndoe'
```


This type of NULL injection serves two purposes. The main purpose is to get a working UNION statement that has no errors. Although this UNION still does not retrieve any real data, it provides an indication that the statement indeed works. Another purpose of this empty UNION is to get a 100% identification of the Database used (using a vendor-specific table name in the FROM clause).

Once the NULL-based UNION statement works, it is a trivial process to identify the types of each column. In each iteration a single field is tested for its type. All three types (number, integer, string) are tested for the field, one of them should work. This way, it takes up to three times the number of columns rather than three to the power of column numbers. Assuming that CCNum is an integer, and that all other fields are strings, the following flow of UNION will identify the valid types:

- `11223344) UNION SELECT NULL,NULL,NULL,NULL WHERE 1=2 --`
  No Error - Syntax is right. MS SQL Server Used. Proceeding.
- `11223344) UNION SELECT 1,NULL,NULL,NULL WHERE 1=2 --`
  No Error – First column is an integer.
- `11223344) UNION SELECT 1,2,NULL,NULL WHERE 1=2 --`
  Error! – Second column is not an integer.
- `11223344) UNION SELECT 1,'2',NULL,NULL WHERE 1=2 --`
  No Error – Second column is a string.
- `11223344) UNION SELECT 1,'2',3,NULL WHERE 1=2 --`
  Error! – Third column is not an integer.
- `11223344) UNION SELECT 1,'2','3',NULL WHERE 1=2 --`
  No Error – Third column is a string.
- `11223344) UNION SELECT 1,'2','3',4 WHERE 1=2 --`
  Error! – Fourth column is not an integer.
- `11223344) UNION SELECT 1,'2','3','4' WHERE 1=2 --`
  No Error – Fourth column is a string.

The attacker has now established a real, valid, UNION statement. Using the increasing numbers it is possible, of course, to identify which field is presented where. All that is left now is to actually exploit this for the purposes of the attack. For that, the injection can be used to retrieve data from the system tables (such as the list of tables and their columns), followed by retrieving actual application data. This document does not, however, go into these details, as they are thoroughly described in several existing SQL Injection white papers.

Summary

This document summarizes research done at WebCohort regarding the possibility of exploiting SQL Injection while having no detailed error messages. This research was performed in order to prove to customers that simply suppressing their error messages, going back to the "Security by Obscurity" approach, is just not enough. Using the techniques described in this document, many applications were proven to be exploitable, despite all attempts to disguise the information sent back to the client.

Hopefully, after reading this document the reader now understands as well, why SQL Injection is a real threat to any system, with or without detailed error messages, and why relying on suppressed error messages is not secure enough.

Most of all, this document shows why an infrastructure level solution (such as suppressing error messages, though any other infrastructure level solution would be similarly problematic) cannot provide a real solution to application level risks. The only way to provide protection to the application itself is by taking application level security measures.

For additional information regarding this whitepaper, please contact WebCohort Research at research@webcohort.com.

About WebCohort

WebCohort is the leader in next-generation web applications and database security solutions. WebCohort's SecureSphere family of products provides enterprises with a comprehensive and essential solution for detecting, reporting and preventing application attacks. Led by Shlomo Kramer, co-founder of Check Point Software Technologies, the company was founded in 2002 by a group of Internet security experts.

WebCohort is privately held company funded and backed by top-tier venture investors (Accel Partners, US Venture Partners and Venrock Associates).

About WebCohort SecureSphere

WebCohort SecureSphere is a revolutionary comprehensive security solution for protecting the Enterprise Application Sphere.

Using advanced anomaly detection, event correlation, and the broadest set of application level signature dictionaries, SecureSphere provides web applications and databases with comprehensive protection against both known and unknown attacks, thus preventing even the most sophisticated intrusion and attack attempts.

Using an in-depth rather then perimeter only approach, SecureSphere protects against both external and internal attacks. Using an innovative Sensor based architecture SecureSphere is a completely transparent, non-intrusive solution providing a scalable enterprise-ready solution that is free of any implementation-related risks.

For additional information visit us at http://www.webcohort.com/.

WebCohort Inc.
12 Hachilazon st. Ramat-Gan **52522**, Israel
Tel: **+972 3 6120133** | Fax: **+972 3 7511133**
U.S. Toll Free: **1-866-592-1289**

428 University Avenue
Palo Alto, CA **94301**
Toll Free: 1-866-926-4678

info@webcohort.com | www.webcohort.com

WEBcohort