SPI DYNAMICS

Start Secure. Stay Secure.™

Blind SQL Injection

Are your web applications vulnerable?

By SPI Labs

Blind SQL Injection

Table of Contents

Introduction

SQL Injection occurs when an application does not properly validate user-supplied input and then includes that input as part of a SQL statement. In a large measure, SQL Injection depends on an attacker discovering and verifying portions of the original SQL query using information gained from error messages. However, web applications can still be vulnerable to Blind SQL Injection attacks even when error messages are not presented, or when they only reveal generic information. By altering the input parameters, an attacker can pose various “true-false” statements to the application to gather information about the database and then ultimately reconstruct the SQL statement by gauging its behavior. Are different pages displayed as a result of changed input? Does an inserted “wait” command cause the application to pause before responding? The “blind” portion of this comes from the fact that no significant error was presented, yet the application is still vulnerable. Blind SQL Injection is no less dangerous than SQL Injection, and can have the same devastating consequences. The objective of this paper is to educate security professionals and developers on the techniques that can be used to take advantage of a web application that is vulnerable to Blind SQL Injection, and to make clear the correct mechanisms that should be put in place to protect against Blind SQL Injection and similar input validation problems.

What is Blind SQL Injection?

In many aspects, SQL Injection and Blind SQL Injection are exactly the same. Blind SQL and SQL Injection are both facilitated by a common coding error: the application accepts data from a client and executes SQL queries without first validating the client's input. The attacker is then free to extract, modify, add, or delete content from the database. In some circumstances, he may even penetrate past the database server and into the underlying operating system. An in-depth guide to SQL Injection can be found here: http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf.

A primary difference between the attacks is in its method of determination. Hackers typically test for SQL injection vulnerabilities by sending the application input that would cause the server to generate an invalid SQL query. If the server then returns an error message to the client, the attacker will attempt to reverse-engineer portions of the original SQL query using information gained from these error messages. A typical administrative safeguard is simply to prohibit the display of database server error messages. The absence of errors only means that the application is protected against one form of SQL Injection.

Since Blind SQL Injection attacks do not rely on error messages, there are no specific patterns or strings to look for in the web server's response. Instead, an attacker will look to see if two requests with different parameter values

will return the same information. In essence, Blind SQL Injection attacks are an attempt to recreate the query in a such a way that the meaning stays the same, but its content differs.

Detecting Blind SQL Injection

Web applications commonly use SQL queries with client-supplied input in the WHERE clause to retrieve data from a database. By adding additional conditions to the SQL statement and evaluating the web application's output, you can determine whether or not the application is vulnerable to Blind SQL injection.

For instance, many companies allow Internet access to archives of their press releases. A URL for accessing the company's fifth press release might look like this:

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5
```

The SQL statement the web application would use to retrieve the press release might look like this (client-supplied input is underlined):

```
SELECT title, description, releaseDate, body FROM pressReleases
WHERE pressReleaseID = 5
```

The database server responds by returning the data for the fifth press release. The web application will then format the press release data into an HTML page and send the response to the client.

Blind SQL Injection

To determine if the application is vulnerable to Blind SQL injection, try injecting an extra true condition into the WHERE clause. For example, if you request this URL . . .

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND
1=1
```

. . . and if the database server executes the following query . . .

```
SELECT title, description, releaseDate, body FROM pressReleases
WHERE pressReleaseID = 5 AND 1=1
```

. . . and if this query also returns the same press release, then the application is susceptible to Blind SQL injection. Part of the user's input was interpreted as SQL code.

A secure application would reject this request because it would treat the user's input as a value, and the value "5 AND 1=1" would cause a type mismatch error. The server would not display a press release.

Another method of testing for Blind SQL Injection vulnerabilities is to alter the "math" of the parameter. For instance, instead of submitting 5 as the value of PressReleaseID, an attacker could submit 3%2b3, which would equate to 3 + 2 if the raw string was passed verbatim to the database. The database would resolve the query because it conforms to a valid syntax. If the same press release is returned, the application is vulnerable to Blind SQL Injection.

It is also important to make sure that inserting “1=1” does not yield results based on a flaw in the application as opposed to Blind SQL Injection. You can do this by inserting “1=2”, an untrue condition, into the SQL query. If the results for each query are the same , then SQL Injection has not been shown to exist.

Exploiting the Vulnerability

When testing for vulnerability to Blind SQL injection, the injected WHERE condition is completely predictable: 1=1 is always true. However, when we attempt to exploit this vulnerability, we don’t know whether the injected WHERE condition is true or false before sending it. If a record is returned, the injected condition must have been true. We can use this behavior to “ask” the database server true/false questions. For instance, the following request essentially asks the database server, “Is the current user dbo?”

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND
USER_NAME() = 'dbo'
```

USER_NAME() is a SQL Server function that returns the name of the current user. If the current user is dbo (administrator), the fifth press release will be returned. If not, the query will fail and no press release will be displayed. By combining subqueries and functions, more complex questions can be posed. The following example attempts to retrieve the name of a database table, one character at a time.

Blind SQL Injection

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND
ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE
xtype='U'), 1, 1))) > 109
```

The subquery (SELECT) is asking for the name of the first user table in the database (which is typically the first thing to do in SQL injection exploitation). The substring() function will return the first character of the query's result. The lower() function will simply convert that character to lower case. Finally, the ascii() function will return the ASCII value of this character.

If the server returns the fifth press release in response to this URL, we know that the first letter of the query's result comes after the letter "m" (ASCII character 109) in the alphabet. By making multiple requests, we can determine the precise ASCII value.

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND
ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE
xtype='U'), 1, 1))) > 116
```

If no press release is returned, the ASCII value is greater than 109 but not greater than 116. So, the letter is between "n" (110) and "t" (116).

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND
ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE
xtype='U'), 1, 1))) > 113
```

Another false statement. We now know that the letter is between 110 and 113.

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND
ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE
xtype='U'), 1, 1))) > 111
```

False again. The range is narrowed down to two letters: 'n' and 'o' (110 and 111).

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND
ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE
xtype='U'), 1, 1))) = 111
```

The server returns the press release, so the statement is true! The first letter of the query's result (and the table's name) is "o." To retrieve the second letter, repeat the process, but change the second argument in the substring() function so that the next character of the result is extracted: (change underlined)

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND
ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE
xtype='U'), 2, 1))) > 109
```

Repeat this process until the entire string is extracted. In this case, the result is "orders."

Solutions

Fundamentally, Blind SQL and SQL Injection are an attack upon the web application, not the web server or the operating system itself. In as such, any fixes other than those implemented in the application code will be stopgap measures and short term solutions, at best. Most methods of preventing Blind SQL and SQL Injection also have their own set of unique

limitations. Therefore, it is best to employ a layered approach to preventing these attacks, and implement several different measures to prevent unauthorized access to your backend database.

Parameterized Queries

SQL Injection arises from an attacker's manipulation of query data to modify query logic. Therefore, the best method of preventing both Blind SQL and SQL Injection attacks is to separate the logic of a query from its data. This will prevent commands inserted from user input from being executed. The downside of this approach, albeit slight, is that it can have an impact on performance, and that each query on the site must be structured in this method for it to be completely effective. If one query is inadvertently bypassed, that could be enough to leave the application vulnerable. The following code shows a sample SQL statement that is SQL injectable.

```
sSql = "SELECT LocationName FROM Locations ";
sSql = sSql + " WHERE LocationID = " + Request["LocationID"];
oCmd.CommandText = sSql;
```

The following example utilizes parameterized queries, and is safe from SQL Injection attacks.

```
sSql = "SELECT * FROM Locations ";
sSql = sSql + " WHERE LocationID = @LocationID";
oCmd.CommandText = sSql;
oCmd.Parameters.Add("@LocationID", Request["LocationID"]);
```

Blind SQL Injection

The application will send the SQL statement to the server without including the user's input. Instead, a parameter-@LocationID- is used as a placeholder for that input. In this way, user input never becomes part of the command that SQL executes. Any input that an attacker inserts will be effectively negated. An error would still be generated, but it would be a simple data-type conversion error, and not something which an attacker could exploit.

The following code samples show a product ID being obtained from an HTTP query string, and used in a SQL query. Note how the string containing the "SELECT" statement passed to SqlCommand is simply a static string, and is not concatenated from input. Also note how the input parameter is passed using a SqlParameter object, whose name ("@pid") matches the name used within the SQL query.

C# sample:

```
        string connString =
WebConfigurationManager.ConnectionStrings["myConn"].ConnectionString;
        using (SqlConnection conn = new SqlConnection(connString))
        {
            conn.Open();

            SqlCommand cmd = new SqlCommand("SELECT Count(*) FROM Products
WHERE ProdID=@pid", conn);

            SqlParameter prm = new SqlParameter("@pid", SqlDbType.VarChar,
50);
            prm.Value = Request.QueryString["pid"];
            cmd.Parameters.Add(prm);

            int recCount = (int)cmd.ExecuteScalar();
        }
```

VB.NET sample:

```
        Dim connString As String =
WebConfigurationManager.ConnectionStrings("myConn").ConnectionString
        Using conn As New SqlConnection(connString)
            conn.Open()

            Dim cmd As SqlCommand = New SqlCommand("SELECT Count(*)
FROM Products WHERE ProdID=@pid", conn)

            Dim prm As SqlParameter = New SqlParameter("@pid",
SqlDbType.VarChar, 50)
            prm.Value = Request.QueryString("pid")
            cmd.Parameters.Add(prm)

            Dim recCount As Integer = cmd.ExecuteScalar()
        End Using
```

Stored Procedures

Another method of separating query logic from its data is by using stored procedures to isolate the web application from SQL altogether. To secure an application against Blind SQL injection, developers must prevent client-supplied data from modifying the syntax of SQL statements. All SQL statements required by the application can be sequestered in stored procedures and kept on the database server. Be aware that simply moving all SQL statements into stored procedures will not solve Blind SQL and SQL Injection problems if you use input parameters without first validating the data. Stored procedures allow programmers to build dynamic SQL statements using string concatenation which can then be executed using EXEC commands. However, this will defeat using stored procedures for security purposes if you use input parameters without sanitizing the data

first. If arbitrary statements must be used, PreparedStatements can be utilized. Using PreparedStatements and stored procedures to compile the SQL statement before the user input is added makes it impossible for user input to modify the actual SQL statement. Finally, the application should execute the stored procedures using a safe interface such as JDBC's CallableStatement or ADO's Command Object.

Let's use pressRelease.jsp as an example. The relevant code would look something like this:

```
String query = "SELECT title, description, releaseDate, body
FROM pressReleases WHERE pressReleaseID = " +
request.getParameter("pressReleaseID");
Statement stmt = dbConnection.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

The first step toward securing this code is to take the SQL statement out of the web application and put it in a stored procedure on the database server.

```
CREATE PROCEDURE getPressRelease
@pressReleaseID integer
AS
SELECT title, description, releaseDate, body FROM pressReleases
WHERE pressReleaseID = @pressReleaseID
```

Now back to the application. Instead of string building a SQL statement to call the stored procedure, a CallableStatement is created to safely execute it.

```
CallableStatement cs = dbConnection.prepareCall("{call
getPressRelease(?)}");
cs.setInt(1,
```

```
Integer.parseInt(request.getParameter("pressReleaseID")));
ResultSet rs = cs.executeQuery();
```

In a .NET application, the change is similar. This ASP.NET code is vulnerable to Blind SQL injection:

```
String query = "SELECT title, description, releaseDate, body
FROM pressReleases WHERE pressReleaseID = " +
Request["pressReleaseID"];

SqlCommand command = new SqlCommand(query,connection);

command.CommandType = CommandType.Text;

SqlDataReader dataReader = command.ExecuteReader();
```

As with JSP code, the SQL statement must be converted to a stored procedure, which can then be accessed safely by a stored procedure SqlCommand:

```
SqlCommand command = new
SqlCommand("getPressRelease",connection);

command.CommandType = CommandType.StoredProcedure;

command.Parameters.Add("@PressReleaseID",SqlDbType.Int);

command.Parameters[0].Value =
Convert.ToInt32(Request["pressReleaseID"]);

SqlDataReader dataReader = command.ExecuteReader();
```

Data Sanitization

The vast majority of Blind SQL Injection vulnerabilities can be prevented by properly validating user input for both type and format. All client-supplied data needs to be cleansed of any characters or strings that could possibly be used maliciously. This should be done for all applications, not just those that use SQL queries. The best method of doing this is via “white listing”. This is defined as only accepting specific data for specific fields, such as limiting user input to account numbers or account types for those relevant fields, or only accepting integers or letters of the English alphabet for others. Many developers will try to validate input by “black listing” characters, or “escaping” them. Basically, this entails rejecting known bad data, such as a single quotation mark, by placing an “escape” character in front of it so that the item that follows will be treated as a literal value. Stripping quotes or putting backslashes in front of them is not enough, and is not as effective as white listing because it is impossible to know all forms of bad data ahead of time.

A good method of filtering data is by using a default-deny regular expression. Make it so that you include only the type of characters that you want. For instance, the following regular expression will return only letters and numbers:

```
s/[^0-9a-zA-Z]//\
```

Make your filter narrow and specific. Whenever possible, use only numbers. After that, numbers and letters only. If you need to include symbols or punctuation of any kind, make absolutely sure to convert them to HTML substitutes, such as `&quote;` or `>`. For instance, if the user is submitting an e-mail address, allow only the "at" sign, underscore, period, and hyphen in addition to numbers and letters, and allow them only after those characters have been converted to their HTML substitutes.

Database Considerations

Limit the rights of the database user. Any successful Blind SQL Injection attack would run in the context of the user's credential. While limiting privileges will not prevent SQL Injection attacks outright, it will make them significantly harder to enact. Don't give that user access to all of the system-stored procedures if that user needs access to only a handful of user-defined ones.

Have a strong SA password policy. Often, an attacker will need the functionality of the administrator account to utilize specific SQL commands. It is much easier to "brute force" the SA password when it is weak, and will increase the likelihood of a successful Blind SQL Injection attack. Another option is not to use the SA account at all, and instead create specific accounts for specific purposes. Also, if you have no need for them, delete SQL stored procedures such as master.Xp_cmdshell, xp_startmail, xp_sendmail, and sp_makewebtask.

About SPI Labs

SPI Labs is the dedicated application security research and testing team of S.P.I. Dynamics, Inc. (www.spidynamics.com). Composed of some of the industry's top security experts, SPI Labs is specifically focused on researching security vulnerabilities at the Web application layer. The SPI Labs mission is to provide objective research to the security community and give organizations concerned with their security practices a method of detecting, remediating, and preventing attacks upon the Web application layer.

SPI Labs' industry leading security expertise is evidenced via continuous support of a combination of assessment methodologies which are used in tandem to produce the most accurate web application vulnerability assessments available on the market. This direct research is utilized to provide daily updates to SPI Dynamics' suite of security assessment and testing software products. These updates include new intelligent engines capable of dynamically assessing web applications for security vulnerabilities by crafting highly accurate attacks unique to each application and situation, and daily additions to the world's largest database of more than 5,000 application layer vulnerability detection signatures and agents. SPI Labs engineers comply with the standards proposed by the Internet Engineering Task Force (IETF) for responsible security vulnerability disclosure. Information regarding SPI Labs policies and procedures for disclosure are outlined on the SPI Dynamics Web site at: http://www.spidynamics.com/spilabs.html.

Contact Information

S.P.I. Dynamics
115 Perimeter Center Place
Suite 1100
Atlanta, GA 30346

Telephone: (678) 781-4800
Fax: (678) 781-4850
Email: info@spidynamics.com
Web: www.spidynamics.com