How PDO avoids SQL injection?

LAMP SitePoint Enthusiast **Mar '09 #1**

I have read in a number of places that PDO prepared statements provide the best form of defense against SQL injection? What does PDO provide that mysql_real_escape_string() doesn't, of course, assuming that MySQL is the underlying database.

elias SitePoint Enthusiast **Mar '09 #2**

afaik it doesn't do much more than mysql_real_escape_string,
the difference is that *you* don't have to do it.

sk89q SitePoint Wizard **Mar '09 #3**

Parametrized statements don't merely just escape the input. The parameters are transferred to the server in a less risky way.

However, as far as you are concerned, using prepared statements is a plus because you cannot accidentally forget to escape a piece of input. If you manually construct SQL statements, forgetting to surround a variable in a function call is pretty easy to do.

logic_earth ¬.¬ shoooo... **Mar '09 #4**

http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html

> Prepared statements can help increase security by separating SQL logic from the data being supplied. This separation of logic and data can help prevent a very common type of vulnerability called an SQL injection

> attack...

LAMP SitePoint Enthusiast **Mar '09 #5**

> sk89q said:
>
> Parametrized statements don't merely just escape the input. They parameters are transferred to the server in a less risky way.
>
> However, as far as you are concerned, using prepared statements is a plus because you cannot accidentally forget to escape a piece of input. If you manually construct SQL statements, forgetting to surround a variable in a function call is pretty easy to do.

does that mean that PDO prepared statements are superior not because of any technical merit but because of the fact that it eliminates the odds of the developer forgetting to escape the string ?

---

sk89q SitePoint Wizard **Mar '09 #6**

No. If you plan to execute a query repeatedly but with different parameters, using parametrized queries will give you a performance boost because the DBMS does not have to re-compile the query.

LAMP SitePoint Enthusiast **Mar '09 #7**

> sk89q said:
>
> No. If you plan to execute a query repeatedly but with different parameters, using parametrized queries will give you a performance boost because the DBMS does not have to re-compile the query.

that is besides the point. We aren't talking about how parameterized queries provide better performance. We are talking about how parameterized queries provide better security over the usual mysql_real_escape_string().

XtrEM3 SitePoint Guru **Mar '09 #8**

if used properly, mysql_real_escape_string() provides the same security level as prepared statements...but it's my opinion that it's easy to improperly (by forgetting to use it, or using it in the wrong spot) use it...so i feel safer using prepared statements because as you said, you don't have to worry about escaping your strings at all.

Jake_Arkinstall Theoretical Physics Student **Mar '09 #9**

It's not there incase you forget to escape the data; It's there so you DON'T need to.

This about it this way.

```
$Query = MySQL_Query("SELECT cols FROM table WHERE data1 = '$Data1'");
```

Ok, so as you probably know, if $Data1 contained any unescaped single quotes then the query can be changed, because it's converted into a string BEFORE being sent to MySQL.

But for PDO:

```
$Query = $Database->Prepare('SELECT cols FROM table WHERE data1 = :data1');
```

That SQL is sent to the database engine, so that the database engine knows what it's doing without needing the values yet.

```
$Query->BindValue(':data1', $Data1);
$Query->Execute();
```

That value is sent separately from the SQL string - it's sent on its own - the value you give it is taken literally, bit for bit.

So, whereas with MySQL_Query where your values are sent WITH the SQL, allowing changes in the SQL if unescaped, in PDO the values are sent separately, so it's impossible for it to affect the SQL.

kyberfabrikken SitePoint Wizard **Mar '09 #10**

> LAMP said:
>
> .. how parameterized queries provide better security over the usual mysql_real_escape_string().

With parameterised queries, the query and the data are sent separately to the database server. When the server receives the query, it parses it. Parsing a formal language is rather complex and has many edge-cases. If the query contains data embedded within, there is always the risk that there might be a loophole where this data is inadvertently interpreted as code. With parameterised queries this can never happen, because the data never ends up in the query-parser and thus is never evaluated.

crmalibu SitePoint Wizard **Mar '09 #11**

mysql_real_escape_string() may also not work properly if you change the character set of the connection after establishing the connection. If you use mysql_set_charset(), it will be ok. But a *SET NAMES* or *SET CHARACTER SET* query issued through mysql_query() won't change the charset mysql_real_escape_string() uses. There's some charsets which would make sql injection a possibility in this scenario.

Another drawback is you store this escaped value temporarily. In the meantime, the character set might be changed before you send the query(whether you use SET NAMES or mysql_set_charset() doesn't matter, you already escaped it and you have a string result). Now the values are no longer escaped using the proper charset for the connection, and again, the vulnerability exists.

Those definately aren't common scenarios. But they are real. I'm not sure how persistant connections play into this. I'm not sure if prepared statements would get the data to mysql correctly if you changed character sets at weird times, but at least there's no chance of sql injection.

Czaries PHP/Rails Developer **Mar '09 #12**

Prepared statements are 2 round-trips to the database for a single query.

As soon as you prepare a query, it's sent to the database with the placeholders you set. So the database engine takes that prepared statement and maps out the query and optimizes it for execution. Then when you call execute() only the values you give are sent to the database, with a reference to that query you just prepared. The database engine drops in the values and runs the query. *This is totally immune to SQL injection*, because the database engine already knows exactly where the values begin and end (the placeholder marker(s) you set), and therefore never need escaping.

The reason SQL injection exists in the first place is because the entire query is interpreted upon execution, values and all. So if anything interferes with the quotes surrounding your values, the engine thinks that value has ended earlier than it really should, and thus a security hole is introduced. That problem is avoided entirely with prepared statements by letting the database engine know ahead of time exactly where to put each value you pass to it later on. There is no need for escaping and there is no need to worry.

---

joebert Floridiot **Mar '09 #13**

Imagine a container full of cargo.

Now imagine that something in that container *might* be radioactive.

Prepared statements would be like taking that potentially radioactive item out of the container and transporting it in a hazmat container using full precautions, instead of just leaving everything in the normal container and letting the receiving party take their own precautions for the potential radiation.

World_Wide_Weird We're from teh basements. **Apr '09 #14**

> LAMP said:
>
> that is besides the point. We aren't talking about how parameterized queries provide better performance. We are talking about how parameterized queries provide better security over the usual mysql_real_escape_string().

It was a direct response to your question about whether prepared statements are superior due to technical merit, actually.

Be aware that, although the third argument to PDOStatement::bindParam is optional, it is necessary in some cases. For example, I have had PDO try to bind an IP address as a numeric type rather than a VARCHAR(15) when the datatype specification was omitted. One workaround is to quote the variable yourself, but that negates the main benefit of using PDO (automatic escaping) in the first place.

CLOSED OCT 7, '14