SQL Injection Attacks by Example

A customer asked that we check out his intranet site, which was used by the company's employees and customers. This was part of a larger security review, and though we'd not actually used SQL injection to penetrate a network before, we were pretty familiar with the general concepts. We were completely successful in this engagement, and wanted to recount the steps taken as an illustration.

Table of Contents

- The Target Intranet
- Schema field mapping
- Finding the table name
- Finding some users
- Brute-force password guessing
- The database isn't readonly
- Adding a new member
- Mail me a password
- Other approaches
- Mitigations
- Other resources

"SQL Injection" is subset of the an unverified/unsanitized user input vulnerability ("buffer overflows" are a different subset), and the idea is to convince the application to run SQL code that was not intended. If the application is creating SQL strings naively on the fly and then running them, it's straightforward to create some real surprises.

We'll note that this was a somewhat winding road with more than one wrong turn, and others with more experience will certainly have different -- and better -- approaches. But the fact that we were successful does suggest that we were not entirely misguided.

There have been other papers on SQL injection, including some that are much more detailed, but this one shows the rationale of **discovery** as much as the process of **exploitation**.

The Target Intranet

This appeared to be an entirely custom application, and we had no prior knowledge of the application nor access to the source code: this was a "blind" attack. A bit of poking showed that this server ran Microsoft's IIS 6 along with ASP.NET, and this suggested that the database was Microsoft's SQL server: we believe that these techniques can apply to nearly any web application backed by any SQL server.

The login page had a traditional username-and-password form, but also an email-me-my-password link; the latter proved to be the downfall of the whole system.

When entering an email address, the system presumably looked in the user database for that email address, and mailed something to that address. Since **my** email address is not found, it wasn't going to send **me** anything.

So the first test in any SQL-ish form is to enter a single quote as part of the data: the intention is to see if they construct an SQL string literally without sanitizing. When submitting the form with a quote in the email address, we get a 500 error (server failure), and this suggests that the "broken" input is actually being parsed literally. Bingo.

We speculate that the underlying SQL code looks something like this:

```
SELECT fieldlist
  FROM table
 WHERE field = '$EMAIL';
```

Here, **$EMAIL** is the address submitted on the form by the user, and the larger query provides the quotation marks that set it off as a literal string. We don't know the specific *names* of the fields or table involved, but we do know their *nature*, and we'll make some good guesses later.

When we enter **steve@unixwiz.net'** - note the closing quote mark - this yields constructed SQL:

```
SELECT fieldlist
  FROM table
 WHERE field = 'steve@unixwiz.net';
```

when this is executed, the SQL parser find the extra quote mark and aborts with a syntax error. How this manifests itself to the user depends on the application's internal error-recovery procedures, but it's usually

different from "email address is unknown". This error response is a dead giveaway that user input is not being sanitized properly and that the application is ripe for exploitation.

Since the data we're filling in appears to be in the **WHERE** clause, let's change the nature of that clause *in an SQL legal way* and see what happens. By entering **anything' OR 'x'='x**, the resulting SQL is:

```
SELECT fieldlist
  FROM table
 WHERE field = 'anything' OR 'x'='x';
```

Because the application is not really thinking about the query - merely constructing a string - our use of quotes has turned a single-component **WHERE** clause into a two-component one, and the **'x'='x'** clause is **guaranteed to be true** no matter what the first clause is (there is a better approach for this "always true" part that we'll touch on later).

But unlike the "real" query, which should return only a single item each time, this version will essentially return every item in the members database. The only way to find out what the application will do in this circumstance is to try it. Doing so, we were greeted with:

> Your login information has been mailed to *random.person@example.com*.

Our best guess is that it's the *first* record returned by the query, effectively an entry taken at random. This person really did get this forgotten-password link via email, which will probably come as surprise to him and may raise warning flags somewhere.

We now know that we're able to manipulate the query to our own ends, though we still don't know much about the parts of it we cannot see. But we **have** observed three different responses to our various inputs:

- "Your login information has been mailed to *email*"
- "We don't recognize your email address"
- Server error

The first two are responses to well-formed SQL, while the latter is for bad SQL: this distinction will be very useful when trying to guess the structure of the query.

Schema field mapping

The first steps are to guess some field names: we're reasonably sure that the query includes "email address" and "password", and there may be things like "US Mail address" or "userid" or "phone number". We'd dearly love to perform a **SHOW TABLE**, but in addition to not knowing the name of the table, there is no obvious vehicle to get the output of this command routed to us.

So we'll do it in steps. In each case, we'll show the whole query as we know it, with our own snippets shown specially. We know that the tail end of the query is a comparison with the email address, so let's guess **email** as the name of the field:

```
SELECT fieldlist
  FROM table
 WHERE field = 'x' AND email IS NULL; --';
```

The intent is to use a proposed field name (**email**) in the constructed query and find out if the SQL is valid or not. We don't care about matching the email address (which is why we use a dummy **'x'**), and the **--** marks the start of an SQL comment. This is an effective way to "consume" the final quote provided by application and not worry about matching them.

If we get a server error, it means our SQL is malformed and a syntax error was thrown: it's most likely due to a bad field name. If we get any kind of valid response, we guessed the name correctly. This is the case whether we get the "email unknown" or "password was sent" response.

Note, however, that we use the **AND** conjunction instead of **OR**: this is intentional. In the SQL schema mapping phase, we're not really concerned with guessing any particular email addresses, and we do not want random users inundated with "here is your password" emails from the application - this will surely raise suspicions to no good purpose. By using the **AND** conjunction with an email address that couldn't

ever be valid, we're sure that the query will always return zero rows and never generate a password-reminder email.

Submitting the above snippet indeed gave us the "email address unknown" response, so now we know that the email address is stored in a field **email**. If this hadn't worked, we'd have tried **email_address** or **mail** or the like. This process will involve quite a lot of guessing.

Next we'll guess some other obvious names: password, user ID, name, and the like. These are all done one at a time, and anything other than "server failure" means we guessed the name correctly.

```
SELECT fieldlist
  FROM table
 WHERE email = 'x' AND userid IS NULL; --';
```

As a result of this process, we found several valid field names:

- email
- passwd
- login_id
- full_name

There are certainly more (and a good source of clues is the names of the fields on **forms**), but a bit of digging did not discover any. But we still don't know the name of the **table** that these fields are found in - how to find out?

Finding the table name

The application's built-in query already has the table name built into it, but we don't know what that name is: there are several approaches for finding that (and other) table names. The one we took was to rely on a **subselect**.

A standalone query of

```
SELECT COUNT(*) FROM tabname
```

Returns the number of records in that table, and of course fails if the table name is unknown. We can build this into our string to probe for the table name:

```
SELECT email, passwd, login_id, full_name
  FROM table
 WHERE email = 'x' AND 1=(SELECT COUNT(*) FROM tabname); --';
```

We don't care how many records are there, of course, only whether the table name is valid or not. By iterating over several guesses, we eventually determined that **members** was a valid table in the database. But is it the table used in **this** query? For that we need yet another test using **table**.**field** notation: it only works for tables that are actually part of this query, not merely that the table exists.

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'x' AND members.email IS NULL; --';
```

When this returned "Email unknown", it confirmed that our SQL was well formed and that we had properly guessed the table name. This will be important later, but we instead took a different approach in the interim.

Finding some users

At this point we have a partial idea of the structure of the **members** table, but we only know of one username: the random member who got our initial "Here is your password" email. Recall that we never received the message itself, only the address it was sent to. We'd like to get some more names to work with, preferably those likely to have access to more data.

The first place to start, of course, is the company's website to find who is who: the "About us" or "Contact" pages often list who's running the place. Many of these contain email addresses, but even those that don't list them can give us some clues which allow us to find them with our tool.

The idea is to submit a query that uses the **LIKE** clause, allowing us to do partial matches of names or email addresses in the database, each time triggering the "We sent your password" message and email. **Warning**: though this reveals an email address each time we run it, it also actually sends that email, which may raise suspicions. This suggests that we take it easy.

We can do the query on email name or full name (or presumably other information), each time putting in the **%** wildcards that **LIKE** supports:

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'x' OR full_name LIKE '%Bob%';
```

Keep in mind that even though there may be more than one "Bob", we only get to see one of them: this suggests refining our **LIKE** clause narrowly.

Ultimately, we may only need one valid email address to leverage our way in.

Brute-force password guessing

One can certainly attempt brute-force guessing of passwords at the main login page, but many systems make an effort to detect or even prevent this. There could be logfiles, account lockouts, or other devices that would substantially impede our efforts, but because of the non-sanitized inputs, we have another avenue that is much less likely to be so protected.

We'll instead do actual password testing in our snippet by including the email name and password directly. In our example, we'll use our victim, **bob@example.com** and try multiple passwords.

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'bob@example.com' AND passwd = 'hello123';
```

This is clearly well-formed SQL, so we don't expect to see any server errors, and we'll know we found the password when we receive the "your password has been mailed to you" message. Our mark has now been tipped off, but we do have his password.

This procedure can be automated with scripting in perl, and though we were in the process of creating this script, we ended up going down another road before actually trying it.

The database isn't readonly

So far, we have done nothing but **query** the database, and even though a **SELECT** is readonly, that doesn't mean that **SQL** is. SQL uses the semicolon for statement termination, and if the input is not sanitized properly, there may be nothing that prevents us from stringing our own unrelated command at the end of the query.

The most drastic example is:

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'x'; DROP TABLE members; --';  -- Boom!
```

The first part provides a dummy email address -- **'x'** -- and we don't care what this query returns: we're just getting it out of the way so we can introduce an unrelated SQL command. This one attempts to drop (delete) the entire **members** table, which really doesn't seem too sporting.

This shows that not only can we run separate SQL commands, but we can also modify the database. This is promising.

Adding a new member

Given that we know the partial structure of the **members** table, it seems like a plausible approach to attempt adding a new record to that table: if this works, we'll simply be able to login directly with our newly-inserted credentials.

This, not surprisingly, takes a bit more SQL, and we've wrapped it over several lines for ease of presentation, but our part is still one contiguous string:

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'x';
       INSERT INTO members ('email','passwd','login id','full name')
       VALUES ('steve@unixwiz.net','hello','steve','Steve Friedl');--';
```

Even if we have actually gotten our field and table names right, several things could get in our way of a successful attack:

1. We might not have enough room in the web form to enter this much text directly (though this can be worked around via scripting, it's much less convenient).
2. The web application user might not have **INSERT** permission on the **members** table.
3. There are undoubtedly other fields in the **members** table, and some may *require* initial values, causing the **INSERT** to fail.
4. Even if we manage to insert a new record, the application itself might not behave well due to the auto-inserted NULL fields that we didn't provide values for.
5. A valid "member" might require not only a record in the **members** table, but associated information in other tables (say, "accessrights"), so adding to one table alone might not be sufficient.

In the case at hand, we hit a roadblock on either #4 or #5 - we can't really be sure -- because when going to the main login page and entering in the above username + password, a server error was returned. This suggests that fields we did not populate were vital, but nevertheless not handled properly.

A possible approach here is attempting to guess the other fields, but this promises to be a long and laborious process: though we may be able to guess other "obvious" fields, it's very hard to imagine the bigger-picture organization of this application.

We ended up going down a different road.

Mail me a password

We then realized that though we are not able to add a new record to the **members** database, we can **modify** an existing one, and this proved to be the approach that gained us entry.

From a previous step, we knew that **bob@example.com** had an account on the system, and we used our SQL injection to update his database record with **our** email address:

```
SELECT email, passwd, login_id, full_name
  FROM members
 WHERE email = 'x';
      UPDATE members
      SET email = 'steve@unixwiz.net'
      WHERE email = 'bob@example.com';
```

After running this, we of course received the "we didn't know your email address", but this was expected due to the dummy email address provided. The **UPDATE** wouldn't have registered with the application, so it executed quietly.

We then used the regular "I lost my password" link - with the updated email address - and a minute later received this email:

```
From: system@example.com
To: steve@unixwiz.net
Subject: Intranet login

This email is in response to your request for your Intranet log in information.
Your User ID is: bob
Your password is: hello
```

Now it was now just a matter of following the standard login process to access the system as a high-ranked MIS staffer, and this was far superior to a perhaps-limited user that we might have created with our **INSERT** approach.

We found the intranet site to be quite comprehensive, and it included - among other things - a list of all the users. It's a fair bet that many Intranet sites also have accounts on the corporate Windows network, and perhaps some of them have used the same password in both places. Since it's clear that we have an easy way to retrieve any Intranet password, and since we had located an open PPTP VPN port on the corporate firewall, it should be straightforward to attempt this kind of access.

We had done a spot check on a few accounts without success, and we can't really know whether it's "bad password" or "the Intranet account name differs from the Windows account name". But we think that automated tools could make some of this easier.

Other Approaches

In this particular engagement, we obtained enough access that we did not feel the need to do much more, but other steps could have been taken. We'll touch on the ones that we can think of now, though we are quite certain that this is not comprehensive.

We are also aware that not all approaches work with all databases, and we can touch on some of them here.

Use xp_cmdshell

Microsoft's SQL Server supports a stored procedure xp_cmdshell that permits what amounts to arbitrary command execution, and if this is permitted to the web user, complete compromise of the webserver is inevitable.

What we had done so far was limited to the web application and the underlying database, but if we can run commands, the webserver itself cannot help but be compromised. Access to **xp_cmdshell** is usually limited to administrative accounts, but it's possible to grant it to lesser users.

Map out more database structure

Though this particular application provided such a rich post-login environment that it didn't really seem necessary to dig further, in other more limited environments this may not have been sufficient.

Being able to systematically map out the available schema, including tables and their field structure, can't help but provide more avenues for compromise of the application.

One could probably gather more hints about the structure from other aspects of the website (e.g., is there a "leave a comment" page? Are there "support forums"?). Clearly, this is highly dependent on the application and it relies very much on making good guesses.

Mitigations

We believe that web application developers often simply do not think about "surprise inputs", but security people do (including the bad guys), so there are three broad approaches that can be applied here.

Sanitize the input

It's absolutely vital to sanitize user inputs to insure that they do not contain dangerous codes, whether to the SQL server or to HTML itself. One's first idea is to strip out "bad stuff", such as quotes or semicolons or escapes, but this is a misguided attempt. Though it's easy to point out **some** dangerous characters, it's harder to point to **all** of them.

The language of the web is full of special characters and strange markup (including alternate ways of representing the same characters), and efforts to authoritatively identify all "bad stuff" are unlikely to be successful.

Instead, rather than "remove known bad data", it's better to "remove everything but known good data": this distinction is crucial. Since - in our example - an email address can contain only these characters:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
@.-_+
```

There is really no benefit in allowing characters that could not be valid, and rejecting them early - presumably with an error message - not only helps forestall SQL Injection, but also catches mere typos early rather than stores them into the database.

> Sidebar on email addresses
>
> It's important to note here that email addresses *in particular* are troublesome to validate programmatically, because everybody seems to have his own idea about what makes one "valid", and it's a shame to exclude a good email address because it contains a character you didn't think about.
>
> The only real authority is RFC 2822 (which encompasses the more familiar RFC822), and it includes a fairly expansive definition of what's allowed. The truly pedantic may well wish to accept email addresses with ampersands and asterisks (among other things) as valid, but others - including this author - are satisfied with a reasonable subset that includes "most" email addresses.
>
> Those taking a more restrictive approach ought to be fully aware of the consequences of excluding these addresses, especially considering that better techniques (prepare/execute, stored procedures) obviate the security concerns which those "odd" characters present.

Be aware that "sanitizing the input" doesn't mean merely "remove the quotes", because even "regular" characters can be troublesome. In an example where an integer ID value is being compared against the user input (say, a numeric PIN):

```
SELECT fieldlist
  FROM table
 WHERE id = 23 OR 1=1;  -- Boom! Always matches!
```

In practice, however, this approach is highly limited because there are so few fields for which it's possible to outright exclude many of the dangerous characters. For "dates" or "email addresses" or "integers" it may have merit, but for any kind of real application, one simply cannot avoid the other mitigations.

Escape/Quotesafe the input

Even if one might be able to sanitize a phone number or email address, one cannot take this approach with a "name" field lest one wishes to exclude the likes of Bill **O'Reilly** from one's application: a quote is simply a valid character for this field.

One includes an actual single quote in an SQL string by putting two of them together, so this suggests the obvious - but wrong! - technique of preprocessing every string to replicate the single quotes:

```
SELECT fieldlist
  FROM customers
 WHERE name = 'Bill O''Reilly';  -- works OK
```

However, this naïve approach can be beaten because most databases support other string escape mechanisms. MySQL, for instance, also permits **\'** to escape a quote, so after input of **\'; DROP TABLE users; --** is "protected" by doubling the quotes, we get:

```
SELECT fieldlist
  FROM customers
 WHERE name = '\''; DROP TABLE users; --';  -- Boom!
```

The expression **'\''** is a complete string (containing just one single quote), and the usual SQL shenanigans follow. It doesn't stop with backslashes either: there is Unicode, other encodings, and parsing oddities all hiding in the weeds to trip up the application designer.

Getting quotes right is **notoriously** difficult, which is why many database interface languages provide a function that does it for you. When the same internal code is used for "string quoting" and "string parsing", it's much more likely that the process will be done properly and safely.

Some examples are the MySQL function **mysql_real_escape_string()** and perl DBD method **$dbh->quote($value)**.

These methods must be used.

- **Use bound parameters (the PREPARE statement)**

  Though quotesafing is a good mechanism, we're still in the area of "considering user input as SQL", and a much better approach exists: **bound parameters**, which are supported by essentially all database programming interfaces. In this technique, an SQL statement string is created with placeholders - a question mark for each parameter - and it's compiled ("prepared", in SQL parlance) into an internal form.

  Later, this prepared query is "executed" with a list of parameters:

  Example in perl

```
$sth = $dbh->prepare("SELECT email, userid FROM members WHERE email = ?;");

$sth->execute($email);
```

  Thanks to Stefan Wagner, this demonstrates bound parameters in Java:

  Insecure version

```
Statement s = connection.createStatement();
ResultSet rs = s.executeQuery("SELECT email FROM member WHERE name = "
                              + formField); // *boom*
```

  Secure version

```
PreparedStatement ps = connection.prepareStatement(
    "SELECT email FROM member WHERE name = ?");
ps.setString(1, formField);
ResultSet rs = ps.executeQuery();
```

  Here, **$email** is the data obtained from the user's form, and it is passed as positional parameter #1 (the first question mark), and at no point do the contents of this variable have anything to do with SQL statement parsing. Quotes, semicolons, backslashes, SQL comment notation - none of this has any impact, because it's "just data". There simply is nothing to subvert, so the application is be largely immune to SQL injection attacks.

  There also may be some performance benefits if this prepared query is reused multiple times (it only has to be parsed *once*), but this is minor compared to the **enormous** security benefits. This is probably the single most important step one can take to secure a web application.

- **Limit database permissions and segregate users**

  In the case at hand, we observed just two interactions that are made not in the context of a logged-in user: "log in" and "send me password". The web application ought to use a database connection with the most limited rights possible: query-only access to the **members** table, and no access to any other table.

  The effect here is that even a "successful" SQL injection attack is going to have much more limited success. Here, we'd not have been able to do the **UPDATE** request that ultimately granted us access, so we'd have had to resort to other avenues.

  Once the web application determined that a set of valid credentials had been passed via the login form, it would then switch that session to a database connection with more rights.

  It should go almost without saying that **sa** rights should *never* be used for any web-based application.

- **Use stored procedures for database access**

  When the database server supports them, use stored procedures for performing access on the application's behalf, which can eliminate SQL entirely (assuming the stored procedures themselves are written properly).

  By encapsulating the rules for a certain action - query, update, delete, etc. - into a single procedure, it can be tested and documented on a standalone basis and business rules enforced (for instance, the "add new order" procedure might reject that order if the customer were over his credit limit).

  For simple queries this might be only a minor benefit, but as the operations become more complicated (or are used in more than one place), having a single definition for the operation means it's going to be more robust and easier to maintain.

Note: it's always possible to write a stored procedure that itself constructs a query dynamically: this provides **no** protection against SQL Injection - it's only proper binding with prepare/execute or direct SQL statements with bound variables that provide this protection.

- **Isolate the webserver**

  Even having taken all these mitigation steps, it's nevertheless still possible to miss something and leave the server open to compromise. One ought to design the network infrastructure to **assume** that the bad guy will have full administrator access to the machine, and then attempt to limit how that can be leveraged to compromise other things.

  For instance, putting the machine in a DMZ with extremely limited pinholes "inside" the network means that even getting complete control of the webserver doesn't automatically grant full access to everything else. This won't stop everything, of course, but it makes it a lot harder.

- **Configure error reporting**

  The default error reporting for some frameworks includes developer debugging information, and this **cannot** be shown to outside users. Imagine how much easier a time it makes for an attacker if the full query is shown, pointing to the syntax error involved.

  This information *is* useful to developers, but it should be restricted - if possible - to just internal users.

Note that not all databases are configured the same way, and not all even support the same dialect of SQL (the "S" stands for "Structured", not "Standard"). For instance, most versions of MySQL do not support subselects, nor do they usually allow multiple statements: these are substantially complicating factors when attempting to penetrate a network.

We'd like to emphasize that though we chose the "Forgotten password" link to attack in this particular case, it wasn't really because this particular web application feature is dangerous. It was simply one of several available features that might have been vulnerable, and it would be a mistake to focus on the "Forgotten password" aspect of the presentation.

This Tech Tip has not been intended to provide comprehensive coverage on SQL injection, or even a tutorial: it merely documents the process that evolved over several hours during a contracted engagement. We've seen other papers on SQL injection discuss the technical background, but still only provide the "money shot" that ultimately gained them access.

But that final statement required background knowledge to pull off, and the process of **gathering** that information has merit too. One doesn't always have access to source code for an application, and the ability to attack a custom application blindly has some value.

Thanks to David Litchfield and Randal Schwartz for their technical input to this paper, and to the great Chris Mospaw for graphic design (© 2005 by Chris Mospaw, used with permission).

Other resources

- SQL Injection walkthrough, SecuriTeam
- "Exploits of a Mom" — Very good xkcd cartoon about SQL injection

Last modified: Mon Mar 6 16:05:28 UTC 2017