3
SQL Injection

SQL injection is yet another common vulnerability that is the result of lax input validation. Unlike cross-site scripting vulnerabilities that are ultimately directed at your site's visitors, SQL injection is an attack on the site itself—in particular its database.

The goal of SQL injection is to insert arbitrary data, most often a database query, into a string that's eventually executed by the database. The insidious query may attempt any number of actions, from retrieving alternate data, to modifying or removing information from the database.

To demonstrate the problem, consider this excerpt:

```
// supposed input
$name = "ilia'; DELETE FROM users;";

mysql_query("SELECT * FROM users WHERE name='{$name}'");
```

The function call is supposed to retrieve a record from the `users` table where the name column matches the name specified by the user. Under normal circumstances, `$name` would only contain alphanumeric characters and perhaps spaces, such as the string `ilia`. But here, by appending an entirely new query to `$name`, the call to the database turns into disaster: the injected `DELETE` query removes all records from `users`.

MySQL Exception
Fortunately, if you use MySQL, the `mysql_query()` function does not permit query stacking, or executing multiple queries in a single function call. If you try to stack queries, the call fails.

However, other PHP database extensions, such as SQLite and PostgreSQL, happily perform stacked queries, executing all of the queries provided in one string and creating a serious security problem.

Magic Quotes

Given the potential harm that can be caused by SQL injection, PHP's automatic input escape mechanism, `magic_quotes_gpc`, provides some rudimentary protection. If enabled, `magic_quotes_gpc`, or "magic quotes", adds a backslash in front of single-quotes, double-quotes, and other characters that could be used to break out of a value identifier. But, magic quotes is a generic solution that doesn't include all of the characters that require escaping, and the feature isn't always enabled (for reasons outlined in the first chapter). Ultimately, it's up to you to implement safeguards to protect against SQL injection.

To help, many of the database extensions available for PHP include dedicated, customized escape mechanisms. For example, the MySQL extension for PHP provides the function `mysql_real_escape_string()` to escape input characters that are special to MySQL:

```
if (get_magic_quotes_gpc()) {
  $name = stripslashes($name);
}
$name = mysql_real_escape_string($name);
mysql_query("SELECT * FROM users WHERE name='{$name}'");
```

However, before calling a database's own escaping mechanism, it's important to check the state of magic quotes. If magic quotes is enabled, remove any backslashes (\) it may have added; otherwise, the input will be doubly-escaped, effectively corrupting it (because it differs from the input supplied by the user).

In addition to securing input, a database-specific escape function prevents data corruption. For example, the escape function provided in the MySQL extension is aware of connection characters and encodes those (and others) to ensure that data isn't corrupted by the MySQL storage mechanism and vice versa.

Native escape functions are also invaluable for storing binary data: left "unescaped", some binary data may conflict with the database's own storage format, leading to the corruption or loss of a table or the entire database. Some database systems, such as PostgreSQL, offer a dedicated function to encode binary data. Rather than escape problematic characters, the function applies an internal encoding. For instance, PostgreSQL's `pg_escape_bytea()` function applies a Base64-like encoding to binary data:

```
// for plain-text data use:
pg_escape_string($regular_strings);

// for binary data use:
pg_escape_bytea($binary_data);
```

A binary data escaping mechanism should also be used to process multi-byte languages that aren't supported natively by the database system. (Multi-byte languages such as Japanese use multiple bytes to represent a single character; some of those bytes overlap with the ASCII range normally only used by binary data.)

There's a disadvantage to encoding binary data: it prevents persisted data from being searched other than by a direct match. This means that a partial match query such as `LIKE 'foo%'` won't work, since the encoded value stored in the database won't necessarily match the initial encoded portion looked for by the query.

For most applications, though, this limitation isn't a major problem, as partial searches are generally reserved for human readable data and not binary data, such as images and compressed files.

Prepared Statements

While database-specific escape functions are useful, not all databases provide such a feature. In fact, database-specific escape functions are relatively rare. (At the moment) only MySQL, PostgreSQL, SQLite, Sybase, and MaxDB extensions provide them. For other databases, including Oracle, Microsoft SQL Server, and others, an alternate solution is required.

A common technique is to Base64-encode all values passed to the database, thus preventing any special characters from corrupting the underlying store or causing trouble. But Base64-encoding expands data roughly 33 percent, requiring larger columns and more storage space. Furthermore, Base64-encoded data has the same problem as binary encoded data in PostgreSQL: it cannot be searched with `LIKE`. Clearly a better solution is needed—something that prevents incoming data from affecting the syntax of the query.

Prepared queries (also called *prepared statements*) solve a great many of the aforementioned risks. Prepared queries are query "templates": the structure of the query is pre-defined and fixed and includes placeholders that stand-in for real data. The placeholders are typically type-specific—for example, `int` for integer data and `text` for strings—which allows the database to interpret the data strictly. For instance, a text placeholder is always interpreted as a literal, avoiding exploits such as the query stacking SQL injection. A mismatch between a placeholder's type and its incoming datum cause, execution errors, adding further validation to the query.

In addition to enhancing query safety, prepared queries improve performance. Each prepared query is parsed and compiled once, but can be re-used over and over. If you need to perform an `INSERT` en masse, a pre-compiled query can save valuable execution time.

Preparing a query is fairly simple. Here is an example:

```
pg_query($conn, "PREPARE stmt_name (text) AS SELECT * FROM users WHERE name=$1");
pg_query($conn, "EXECUTE stmt_name ({$name})");
pg_query($conn, "DEALLOCATE stmt_name");
```

`PREPARE stmt_name (text) AS` ... creates a prepared query named `stmt_name` that expects one `text` value. Everything following the keyword `AS` defines the actual query, except `$1` is the placeholder for the expected `text`.

If a prepared statement expects more than one value, list each type in order, separated by a comma, and use `$1`, `$2`, and so on for each placeholder, as in `PREPARE stmt_example (text, int) AS SELECT * FROM users WHERE name=$1 AND id=$2`.

Once compiled with `PREPARE`, you can run the prepared query with `EXECUTE`. Specify two arguments: the name of the prepared statement (such as `stmt_name`) to run and a list of actual values enclosed in parentheses.

Once you're finished with the prepared statement, dispose of it with `DEALLOCATE`. Forgetting to jettison prepared queries can cause future PREPARE queries to fail.This is a common error when persistent database connections are used, where a statement can persist across requests. For example, Given that there is no way to check if a statement exists or not, a blind attempt to create one anyway will trigger a query error if one is already present.

As nice as prepared queries are, not all databases support them; in those instances escaping mechanisms should be used.

No Means of Escape

Alas, escape functions do not always guarantee data safety. Certain queries can still permit SQL injection, even after escapes are applied.

Consider the following situation, where a query expects an integer value:

```
$id = "0; DELETE FROM users";
$id = mysql_real_escape_string($id); // 0; DELETE FROM users
mysql_query("SELECT * FROM users WHERE id={$id}");
```

When executing integer expressions, it's not necessary to enclose the value inside single quotes. Consequently, the semicolon character is sufficient to terminate the query and inject an additional query. Since the semicolon doesn't have any "special" meaning, it's left as-is by both the database escape function and addslashes().

There are two possible solutions to the problem.

The first requires you to quote *all* arguments. Since single quotes are always escaped, this technique prevents SQL injection. However, quoting still passes the user input to the database, which is likely to reject the query. Here is an illustrative example:

```
$id = "0; DELETE FROM users";
$id = pg_escape_string($id); // 0; DELETE FROM users
pg_query($conn, "SELECT * FROM users WHERE id='{$id}'")
                 or die(pg_last_error($conn));
// will print invalid input syntax for integer: "0; DELETE FROM users"
```

But query failures are easily avoided, especially when validation of the query arguments is so simple. Rather than pass bogus values to the database, use a PHP *cast* to ensure each datum converts successfully to the desired numeric form.

For example, if an integer is required, cast the incoming datum to an `int`; if a complex number is required, cast to a `float`.

```
$id = “123; DELETE FROM users”;
$id = (int) $id; // 123
pg_query($conn, “SELECT * FROM users WHERE id={$id}”); // safe
```

A cast forces PHP to perform a type conversion. If the input is not entirely numeric, only the leading numeric portion is used. If the input doesn't start with a numeric value or if the input is only alphabetic and punctuation characters, the result of the cast is 0. On the other hand, if the cast is successful, the input is a valid numeric value and no further escaping is needed.

Numeric casting is not only very effective, it's also efficient, since a cast is a very fast, function-free operation that also obviates the need to call an escape routine.

The LIKE Quandary

The SQL `LIKE` operator is extremely valuable: its `%` and `_` (underscore) qualifiers match 0 or more characters and any single character, respectively, allowing for flexible partial and substring matches. However, both `LIKE` qualifiers are ignored by the database's own escape functions and PHP's magic quotes. Consequently, user input incorporated into a `LIKE` query parameter can subvert the query, complicate the `LIKE` match, and in many cases, prevent the use of indices, which slows a query substantially. With a few iterations, a compromised `LIKE` query could launch a Denial of Service attack by overloading the database.

Here's a simple yet effective attack:

```
$sub = mysql_real_escape_string(“%something”); // still %something
mysql_query(“SELECT * FROM messages WHERE subject LIKE ‘{$sub}%’”);
```

The intent of the `SELECT` above is to find those messages that *begin* with the user-specified string, `$sub`. Uncompromised, that `SELECT` query would be quite fast, because the index for `subject` facilitates the search. But if `$sub` is altered to include a leading `%` qualifier (for example), the query can't use the index and the query takes far longer to execute—indeed, the query gets progressively slower as the amount of data in the table grows.

The underscore qualifier presents both a similar and a different problem. A leading underscore in a search pattern, as in `_ish`, cannot be accelerated by the index, slowing the query. And a trailing underscore may substantially alter the results of the query. To complicate matters further, underscore is a very common character and is frequently found in perfectly valid input.

To address the `LIKE` quandary, a custom escaping mechanism must convert user-supplied `%` and `_` characters to literals. Use `addcslashes()`, a function that let's you specify a character range to escape.

```
$sub = addcslashes(mysql_real_escape_string("%something_"), "%_");
// $sub == \%something\_
 mysql_query("SELECT * FROM messages WHERE subject LIKE '{$sub}%'");
```

Here, the input is processed by the database's prescribed escape function and is then filtered through `addcslashes()` to escape all occurrences of `%` and `_`. `addcslashes()` works like a custom `addslashes()`, is fairly efficient, and much faster alternative that `str_replace()` or the equivalent regular expression.

Remember to apply manual filters after the SQL filters to avoid escaping the backslashes; otherwise, the escapes are escaped, rendering the backslashes as literals and causing special characters to re-acquire special meanings.

SQL Error Handling

One common way for hackers to spot code vulnerable to SQL injection is by using the developer's own tools against them. For example, to simplify debugging of failed SQL queries, many developers echo the failed query and the database error to the screen and terminate the script.

```
mysql_query($query) or die("Failed query: {$query}<br />".mysql_error());
```

While very convenient for spotting errors, this code can cause several problems when deployed in a production environment. (Yes, errors do occur in production code for any number of reasons.) Besides being embarrassing, the code may reveal a great deal of information about the application or the site. For instance, the end-user may be able discern the structure of the table and some of its fields and may be able to map `GET/POST` parameters to data to determine how to attempt a better SQL injection attack. In fact, the SQL error may have been caused by an inadvertent SQL injection. Hence, the generated error becomes a literal guideline to devising more tricky queries.

The best way to avoid revealing too much information is to devise a very simple SQL error handler to handle SQL failures:

```
function sql_failure_handler($query, $error) {
  $msg = htmlspecialchars("Failed Query: {$query}<br>SQL Error: {$error}");

  error_log($msg, 3, "/home/site/logs/sql_error_log");

  if (defined('debug')) {
          return $msg;
  }
  return "Requested page is temporarily unavailable, please try again later.";
}

mysql_query($query) or die(sql_failure_handler($query, mysql_error()));
```

The handler function takes the query and error message generated by the database and creates an error string based on that information. The error string is passed through `htmlspecialchars()` to ensure that none of the characters in the string are rendered as HTML, and the string is appended to a log file .

The next step depends on whether or not the script is working in debug mode or not. If in debug mode, the error message is returned and is likely displayed on-screen for the developer to read. In production, though, the specific message is replaced with a generic message, which hides the root cause of the problem from the visitor.

Authentication Data Storage

Perhaps the final issue to consider when working with databases is how to store your application's database credentials—the login and password that grant access to the database. Most applications use a small PHP configuration script to assign a login name and password to variables. This configuration file, more often than not (at least on shared hosts), is left world-readable to provide the web server user access to the file. But world-readable means just that: anyone on the same system or an exploited script can read the file and steal the authentication information stored within. Worse, many applications place this file inside web readable directories and give it a non-PHP extension—.inc is a popular choice. Since .inc is typically not configured to be interpreted as a PHP script, the web browser displays such a file as plain-text for all to see.

One solution to this problem uses the web server's own facilities, such as .htaccess in Apache, to deny access to certain files. As an example, this directive denies access to all files that end (notice the $) with the string .inc.

```
<Files ~ “\.inc$”>
    Order allow,deny
    Deny from all
</Files>
```

Alternatively, you can make PHP treat .inc files as scripts or simply change the extension of your configuration files to .php or, better yet, .inc.php, which denotes that the file is an include file.

However, renaming files may not always be the safest option, especially if the configuration files have some code aside from variable initialization in the main scope. The ideal and simplest solution is to simply not keep configuration and non-script files inside web server-accessible directories.

That still leaves world-readable files vulnerable to exploit by local users.

One seemingly effective solution is to encrypt the sensitive data. Database authentication credentials could be stored in encrypted form, and only the applications that know the secret key can decode them. But this use of encryption only makes theft slightly more difficult and merely shifts the problem instead of eliminating it. The secret key necessary to decrypt the credentials must still be accessible by PHP scripts running under the web server user, meaning that the key must remain world-readable. Back to square one…

A proper solution must ensure that other users on the system have no way of seeing authentication data. Fortunately, the Apache web server provides just such a mechanism. The Apache configuration file, httpd.conf can include arbitrary intermediate configuration files during start-up while Apache is still running as root. Since root can read any file, you can place sensitive information in a file in your home directory and change it to mode 0600, so only you and the superuser can read and write the file.

```
<VirtualHost ilia.ws>
Include /home/ilia/sql.cnf
</VirtualHost>
```

If you use the `Include` mechanism, be sure that your file is only loaded for a certain `VirtualHost` or a certain directory to prevent the data from being available to other hosts on the system.

The content of the configuration file is a series of `SetEnv` lines, defining all of the authentication parameters necessary to establish a database connection.

```
SetEnv DB_LOGIN “login”
SetEnv DB_PASSWD “password”
SetEnv DB_DB “my_database”
SetEnv DB_HOST “127.0.0.1”
```

After Apache starts, these environment variables are accessible to the PHP script via the `$_SERVER` super-global or the `getenv()` function if `$_SERVER` is unavailable.

```
echo $_SERVER[‘DB_LOGIN’]; // login
echo getenv(“DB_LOGIN”); // login
```

An even better variant of this trick is to hide the connection parameters altogether, hiding them even from the script that needs them. Use PHP’s `ini` directives to specify the default authentication information for the database extension. These directives can also be set inside the hidden Apache configuration file.

```
php_admin_value mysql.default_host “127.0.0.1”
php_admin_value mysql.default_user “login”
php_admin_value mysql.default_password “password”
```

Now, `mysql_connect()` works without any arguments, as the missing values are taken from PHP `ini` settings. The only information remaining exposed would be the name of the database.

Because the application is not aware of the database settings, it consequently cannot disclose them through a bug or a backdoor, unless code injection is possible. In fact, you can enforce that only an `ini`-based authentication procedure is used by enabling SQL safe mode in PHP via the `sql.safe_mode` directive. PHP then rejects any database connection attempts that use anything other than `ini` values for specifying authentication data.

This approach does have one weakness in older versions of PHP: up until PHP 4.3.5, there was a bug in the code that leaked INI settings from one virtual host to another. Under certain conditions, this bug could be triggered by a user, effectively providing other users on the system with a way to see the `ini` values of other users.

If you're using an older version of PHP, stick to the environment variables or upgrade to a newer version of PHP, which is a very good idea anyway, since older releases include many other security problems.

Database Permissions

The last database security tip has nothing to do with PHP per se, but is sound advice that can be applied to every component in your system. In general, grant the fewest privileges possible.

For example, if a user only requires read-access to the database, don't permit the user to execute `UPDATE` or `INSERT` queries. Or more realistically, limit write access to those tables that are expected to change—perhaps the session table and the user accounts table.

By limiting what a user can do, you can detect, track, and defang many SQL injection attacks. Limiting access at the database level is supplemental: you should use it in addition to all of the database security mechanisms listed in this chapter.

Maintaining Performance

Speed isn't usually considered a security measure, but subverting your application's performance is tantamount to any other exploit. As was demonstrated by the `LIKE` attack, where `%` was injected to make a query very slow, enough costly iterations against the database could saturate the server and prevent further connections. Unoptimized queries present the same risk: if the attacker spots inefficiencies, your server can be exhausted and rendered useless just the same.

To prevent database overloading, there are a few simple rules to keep in mind.

Only retrieve the data you need and nothing more. Many developers take the "*" shortcut and fetch all columns, which may result in a lot of data, especially when joining multiple tables. More data means more information to retrieve, more memory for the database's temporary buffer for sorting, more time to transmit the results to PHP, and more memory and time to make the results available to your PHP application. In some cases, with large amounts of data, database sorting must be done within a search file instead of memory, adding to the overall time to process a request. Again, only retrieve the data you need, and name the columns to minimize size further.

To further accelerate a query, try using *unbuffered queries* that retrieve query results a small portion at a time. However, unbuffered queries must be used carefully: only one result cursor is active at any time, limiting you to work with one query at a time. (And in the case of MySQL, you cannot even perform `INSERT`, `UPDATE`, and other queries until all results from the result cursor have been fetched).

To work with a database, PHP must establish a connection to it, which in some cases can be a rather expensive option, especially when working with complex systems like Oracle, PostgreSQL, MSSQL, and so on. One trick that speeds up the connection process is to make a database connection *persistent*, which allows the database handle to remain valid even after the script is terminated. If a connection is persistent, each subsequent connection request from the same web server process reuses the connection rather than recreating it anew.

The code below creates a persistent MySQL database connection via the `mysql_pconnect()` function, which is syntactically identical to the regular `mysql_connect()` function.

```
mysql_pconnect("host", "login", "passwd");
```

Other databases typically offer a persistent connection variant, some as simple as adding the prefix "p" to the word "connect".

Anytime PHP tries to establish a persistent connection it first looks for an existing connection with the same authentication values; if such a connection is available, PHP returns that handle instead of making a new one.

Words of Caution

Persistent connections are not without drawbacks. For example, in PHP, connection pooling is done on a per-process basis rather than per-web server, giving *every* web-server process its own connection pool. So, 50 Apache processes result in 50 open database connections. If the database is not configured to allow at least that many connections, further connection requests are rejected, breaking your web pages.

In many cases, the database runs on the same machine as the web server, which allows data transmission to be optimized. Rather than using the slow and bulky TCP/IP, your application can use Unix Domain Sockets (UDG), the second fastest medium for Inter Process Communication (IPC). By switching to UDG, you can significantly improve the data transfer rates between the two servers.

To switch to UDG, change the host parameter of the connection. For example, in MySQL, set the host, followed by the path to the UDG.

```
mysql_connect(":/tmp/mysql.sock", "login", "passwd");
pg_connect("host=/tmp user=login password=passwd");
```

In PostgreSQL, where there's no need for a special host identifier, simply set the host parameter to the directory where the UDG can be found and enjoy the added performance.

Query Caching

In some instances, a query is as fast as it can be, yet still take significant time to execute. If you cannot throw hardware at the problem—which has its limits as well—try to use the *query cache*. A query cache retains a query's results for some period of time, short-circuiting the need to recreate the results from scratch each time the same query runs.

Each time there's a request for a page, the cache is checked; if the cache is empty, if the cache expired the previous results, or if the cache was invalidated (say, by an `UPDATE` or an `INSERT`), the query executes. Otherwise, the results saved in the cache are returned, saving time and effort.