SQL Injection: A Case Study
Stephen C. Bono and Ersin Domangue

01 October 2012

overview

We were recently engaged to perform a black-box security evaluation of a client’s web site that, in part, used SQL. We demonstrated the significance of how devastating a SQL injection attack can be. In order to combat the prevalence of this vulnerability, we strongly recommend that all developers follow the best practice guidelines we outline in this article.

The scenario described here is pedagogical and so some liberties were taken to gear this discussion strictly to the topic of SQL-injection. Here we describe a subset of the actions taken and the results obtained.

introduction

SQL injection and associated vulnerabilities are possible due to three common, yet critical design flaws. Lack of input sanitization, unnecessary construction of dynamic queries, and failure to adhere to the Principle of Least Privilege. Through our case study, we demonstrate how each of these design flaws can lead to information or system compromise.

Input sanitization refers to the removal of unwanted, unexpected or harmful data from application inputs. This can refer to the removal or reformatting of unwanted characters or keywords, the truncation of excessively long inputs, or the general restructuring of an input such that it is as it is expected to be. The need for input sanitization reaches far beyond the prevention of SQL injection attacks alone, and is the cause of cross-site scripting, buffer overflow, and a host of other injection vulnerabilities. With SQL injection, input sanitization is of particular importance, as many common characters are included in the syntax of SQL statements, including ones that you might anticipate to find within the user’s input. For example, the single quote character (‘) is used in SQL statements to designate the start and end of a string value, but the single quote character is also commonly found in proper names and sentence punctuation. The following dynamically generated SQL statement would then break if the last name “O’hara” was input:

```
SELECT * FROM users WHERE last_name='O'hara';
```

The syntax of this statement is incorrect and would result in an error, because it appears to the SQL interpreter that it should execute a statement selecting data from all users with the last name “O” followed by the unrecognized keyword “hara” and additional single quote and semi-colon.
The dynamic nature of the above example is also problematic, in the sense that the statement to be executed by the SQL interpreter is created on the fly as the input is entered. In the above case, the first part of the statement “SELECT * FROM users WHERE last_name='” is concatenated with a user-input value and a closing single quote character. Because of this, an attacker could input values such that the intended SQL statement becomes an entirely different statement. For example, entering the last name value “Jones' OR '1'='1” causes the application to create the following statement, which is markedly different than what was intended:

```
SELECT * FROM users WHERE last_name='Jones' OR '1'='1';
```

Rather than select the date of the user with last name “Jones,” this statement will select the data of all users.

As with a lack of input sanitization, the consequences of these dynamic statements are prevalent in other forms of injection attacks, such as XPATH and LDAP injection. Fortunately, there are elegant methods for avoiding dynamic statements, such as using prepared statements and stored procedures, which we discuss later in the mitigations section.

The third design flaw we mention is the failure to adhere to the Principle of Least Privilege, which bluntly asserts that a person, process or device should only have access to the minimum information or resources required to perform its duties. This could be the restriction of read, write or execute privileges, limitations on storage space, or restrictions on the time-availability of access to resources. With SQL injection vulnerabilities, our primary concern is with what access the calling service (most often a web server) has access to within the database. More often than one might expect, web services are granted full, administrative access to the database system. Read-only access, and more specifically, read-only access to the minimum set of information required for the application to function, should be enforced.

**ATTACK**

**attack plan**

Since this evaluation began as a black-box security assessment, we began by laying out our attack plan and including a number of reconnaissance steps. This is an excerpt of our plan pertaining to SQL injection vulnerabilities.

1) Identify inputs to the system.
2) Determine if a SQL server is running / handling these inputs.
3) Determine if SQL injection is possible.
4) Fingerprint the SQL server.
5) Determine if we can embed(concatenate) statements.
6) Determine if we can modify the database.
7) Determine if we can map the database.
8) Determine if we can harvest the database.
9) Determine if we can compromise the host server itself.

A good starting point for any black-box investigation is to run a web application scanner on the target application. These scanners can sometimes identify inputs that are not readily noticeable to the naked eye, and at the very least can quickly enumerate the inputs that make up the attack surface. However, as with any black-box assessment, the plan needed to be adapted and extended as we encountered various pitfalls. First and foremost, our initial set of inputs to the system was limited to the login interface. This included user name and password entry fields, as well as a field for submitting an email address if a password was forgotten.

**determine if SQL is used, injectable**

Our first, most basic test to determine if SQL is being used, is to try to cause an error by corrupting a SQL statement. We’ve already touched on how this can happen above, but now our attempt is more targeted. We enter a single quote character in each field, hoping to truncate or corrupt a dynamically generated SQL query, and cause an error. We are assuming here that the SQL statement might look something like the following:

```
SELECT <something> WHERE <something>='one of our inputs';
```

Adding a single quote in the first example would cause a SQL statement to be constructed with an extra quote, resulting in an invalid statement and error.

No luck. The error message returned immediately was “invalid username or password.” In many vulnerable cases, we would expect to see a detailed error message returned from the web server—a drastic information leakage vulnerability that would have immediately identified to us that yes, a SQL server is present, a fingerprint, and possibly even a version identifier of the server and information about the database such as table and field names which we would not ordinarily know.

Though we didn’t receive the detailed message we were hoping for, it did not mean our SQL injection attack did not cause an internal error. We may now have to rely on what is referred to as a “blind SQL injection” attack. This is an attack where the results may not be displayed to the user, at least not in the typical sense.

Blind attacks may be verified with the inclusion of always true or always false statements, such as appending “1=1” or “0=1” to a SQL query, in the hopes that the non-descriptive error message will be activated or deactivated depending on the Boolean choice. For example, if we had a legitimate user name and password (‘jdoe’ and ‘pw123’), we may be able to blindly test if a SQL
database is used. After a successful controlled login and failure, we would attempt to log in again with the strings “jdoe' AND '0'='1” and “jdoe' AND '1'='1” to verify that the former failed, and the later granted access, when it clearly should not have. Inserting these strings could look like the following:

```
SELECT id FROM users WHERE uname='jdoe' AND passwd='passwd';
SELECT id FROM users WHERE uname='jdoe' AND '0'='1' AND passwd='passwd';
SELECT id FROM users WHERE uname='jdoe' AND '1'='1' AND passwd='passwd';
```

The first and third statements would be successful, while the second would fail. We would have then verified that a blind SQL injection was possible in these fields.

However, as black-box adversaries, we did not start with legitimate credentials, and so we took a different approach. Rather than evaluate whether an injection was successful based on pass or fail results, we injected a SQL statement that would take a noticeably long time to process. If the login was rejected immediately, we wouldn’t have learned much, but if our login was rejected after a noticeable delay, it would be safe to assume our injection was successful. We set the username field to the following:

```
Jdoe' OR BENCHMARK(1000000, ENCODE('blah','nothing')) OR '1'='1
```

The injected command instructed the SQL database to use its pseudo-random number generator one million times to password-encrypt the phrase “blah” with the pass-phrase “nothing”. To our liking, this significantly delayed the return of the failed login page. We now knew that SQL injection was possible through the username field.

**bypassing login**

Even though we had discovered at this point that SQL injection was possible, we had very little access to what was returned from our injected SQL queries. Until we had additional access and additional inputs to manipulate, we may not have been able to fully compromise the database, and so our attack plan was updated to focus on leveraging the SQL injection vulnerability to bypass the application’s login page.

The means by which a web application developer could have implemented the login process to the system are potentially infinite, but in practice, the number of reasonable methods is an assessable value. It was possible that the SQL query we could inject could entirely subvert the login process, regardless of how it is structured. Two of the following methods were pulled from unnamed online tutorials, and the third was the implementation used by the site we were investigating. All are typical, and easily subverted.

```
SELECT id FROM users WHERE uname='<input>' AND passwd='<input>';
If results.count = 1 { grant access } else { error }
```
SELECT COUNT(id) AS count FROM users WHERE uname='<input>' AND pw='<input>';
If count == 1 { grant access } else { error }

SELECT passwd FROM students WHERE uname='<input>';  
if results.passwd = '<input>' { grant access } else {error }

Attacks on the first two login tests above can be launched by setting the password input field to “garbage' OR '1'='1” as in shown in the SQL statement it creates:

SELECT id FROM users WHERE uname='user_name' AND passwd='garbage' OR '1'='1';

These attempts failed during our evaluation, but we were successful in the following attack to gain access. By inputting the username field as “junk' OR passwd='password” we created the following SQL statement:

SELECT passwd FROM students WHERE uname='junk' OR passwd='password';

This attack tells the database to return any user who has the poorly chosen password “password.” When the subsequent check is made to see if our password field matches, we are granted access as whichever user happened to be returned first. We now had access to a user’s account in the system, availing us additional inputs and therefore additional SQL injection possibilities.

**fingerprinting the SQL server**

With our new found access, we were shown the personal information for a user of the system, including name, address, phone, email and social security number (Hooah!). For the remainder of our assessment, we really didn’t require much more than this page and the corresponding “update address” page. The following was the update address statement:

UPDATE students SET address='<input>', zip='<input>', state='<input>' WHERE id=id;

By setting the zip code field to “1234', zip='55555” we saw that the “55555” value was inserted in to the database in the zip code field, rather than resulting in an error. The query string was susceptible to SQL injection.

Our attack plan included tests to fingerprint the SQL server, that is, to uncover as much information about the database, host and running services as possible. This information allows an attacker to both eliminate tests and attacks that are not useful against a particular deployment, and to focus on attacks that would probably work.

Many versions of SQL have syntax for directly querying database and system information. For instance, consider the `VERSION()` command in MySQL and PostgreSQL, and the `SERVERPROP-`
ERTY('productversion') command for Microsoft SQL Server. Each of these commands may return a descriptive string that immediately fingerprints the server.

Even when these version commands are not accessible (perhaps the injection results are blind), it may still be possible to fingerprint a server. Depending on the server version, certain functions and syntactical conventions may or may not be permitted as part of a statement. These subtle differences could be enumerated in an injectable blind field to see what is and is not allowed. The version of SQL that supports the successful attacks and rejects the unsuccessful ones is then fingerprinted.

We were able to identify our victim server’s version and host system through two different methods. First, recall through our earlier attack that we were able to obtain the credentials for an account with the password, “password.” Going back to the login screen, we entered the correct user name “jdoe” and the following sequence of passwords, testing for correctness:

1. pass' + 'word
2. junk' OR passwd=concat('pass','word') OR '0'='1
3. pass' || 'word

Through attempts 1 and 2 we were successfully able to log in. This led us to believe that our server was MySQL, and strongly suggested that it was not an Oracle or PostgreSQL database.

A second attempt to fingerprint the database server left no guessing. We injected the following into the zip code field when updating our victim account’s address:

12345’, zip=VERSION(), state=’PA

Note that the additional “state=’PA” was required to cancel out the single quote added by the dynamic query after concatenating the zip code string.

To our delight, the value of the zip code field on our account information page then displayed “5.1.41-3ubuntu12.10” a version identifying this system as MySQL version 5.1 running on an Ubuntu host.

**concatenating or embedding statements**

Launching a SQL injection attack becomes much easier when concatenated or embedded statements are possible. It allows us to construct entirely standalone statements, and we no longer have to rely on the formatting of the surrounding dynamically generated SQL to assist us.

A concatenated statement is formed by separating two valid statements by a semicolon character. For example, the following SQL query (technically, two queries) performs an **UPDATE**, followed by a **SELECT**.
UPDATE users SET first='<input>', last='<input>' WHERE id=1234; SELECT * FROM users;

If concatenated statements are permitted by the server, and the above UPDATE statement was susceptible to injection, we could insert an entire statement by setting the first <input> field to

fake' WHERE id=1234; DROP TABLE users;

resulting in the following sequence of commands:

UPDATE users SET first='fake' WHERE id=1234;
DROP TABLE users;
', last='junk' WHERE id=1234; SELECT * FROM users;

The first command does a pointless set of the name field for the user with id “1234,” and the second command deletes the table named “users.” The third command is garbage, and will likely cause an error, but we don’t care at this point because our attack is complete.

Embedded statements are also very powerful tools for launching SQL injection attacks. An embedded statement is a statement nested within another statement, such that the result of the inner statement is used as input to the outer statement. This is demonstrated in the following example:

SELECT id FROM users WHERE name=(SELECT name FROM roster WHERE position='pitcher');

This statement does just what it sounds like when reading through it. First, select the name of the “pitcher” from the table “roster,” then select the id of the user with that same name.

If embedded statements are permitted by the server, and we have a dynamic query where we can insert our own injected code, such as with the above vulnerable UPDATE statement, we could inject an entire statement by setting the first <input> field to:

junk', first=(SELECT passwd FROM users WHERE id=3333), last=''

resulting in the following full command:

UPDATE users SET first='junk', first=(SELECT passwd FROM users WHERE id=3333), last='', last='fake' WHERE id=1234;

The above example attack will set the name field of our account to the password field of another account.

Knowing that the server in our assessment was MySQL version 5.1, we knew that embedded statements might be possible. By setting the zip code field to an additional SELECT statement that returned a scalar value, we saw that this was indeed possible. Consider the following input to the zip code field when updating the student’s information,
The result is that our zip code is not the “12345” that we first specified, but the second assignment of the value returned by the embedded SQL statement, which was our victim account’s id field. We were now assured that we would be able to query the database in nearly any fashion we chose.

**mapping and harvesting the database**

A database can be thought of as a series of tables, each with a series of columns, each with some set of attributes. As one might suspect, this information needs to be stored somewhere. What better place than the database itself? Lucky for most SQL injection practitioners, many database features, settings, table names and column names can be queried and listed through the same database connection that we pull data. Each database server provides its own mechanisms for querying this information, and we won’t enumerate all the possibilities here, but it suffices to show through the following example attacks that a database can be mapped given even a very limited portal to the information.

We leveraged our previously demonstrated ability to embed `SELECT` statements within the `UPDATE` statements to query the underlying database for its table and field names. Similar to the above attack, we injected the following in to our zip code field

', zip=(SELECT table_name FROM information_schema.tables LIMIT 1,1), state='PA

The final SQL statement set our zip code field to the name of the first table in the database’s list of tables, “CHARACTER_SETS.” Other table names can be retrieved by simply changing the value of the `LIMIT` condition. The following embedded statement,

```
SELECT table_name FROM information_schema.tables LIMIT 35,1
```

obtained the result “students” as the 35th table in the database, and the statement,

```
SELECT column_name FROM information_schema.columns WHERE table_name= 'students', LIMIT 2,1
```

obtained the result “pwd”, the second column in the “students” table. In a likewise manner, and in an automated fashion, we were able to deduce and map the remaining tables within the database.

Harvesting the database was now possible through the very same methods. Knowing all required table and column names, we could have easily automated the process of pulling all data from the database through a single output.
compromising the host

As mentioned at the onset of this article, the third design flaw we set out to demonstrate in our assessment was failure to adhere to the Principle of Least Privilege. Our target machine, running MySQL version 5.1 on an Ubuntu host, was apparently running the database service with elevated permissions. Furthermore, the database user “www” was allowed administrative access to the MySQL database. The combination of these gave us control over much of the system, as we will now demonstrate.

First, with administrative access to the database itself, we were able read the hashed, root password from the table, “mysql.user” and then replace it with our own password using the following injected and concatenated commands,

```
UPDATE students SET address='fake', zip='', zip=(SELECT password FROM mysql.user WHERE user = 'root'), state=' ', state='PA' WHERE id=id;

UPDATE mysql.user SET password = PASSWORD('gotcha') WHERE user='root';
```

Later, to cover our tracks we could reset the root password to the hashed value we copied out of the database. Alternatively, we could have added a new user of our own.

We then changed the domain access privileges for the user “root” to allow connections from anywhere, rather than only from localhost, with the following statement:

```
UPDATE mysql.user SET host='%' WHERE user='root';
```

Now, once the MySQL service was restarted, we had root access to the database from a MySQL prompt. This effectively gave us a shell on the host system, as any shell commands can be launched from the MySQL prompt with the syntax “\! <cmd>”. This gave us control over much of the file system, and ultimately full control of the host machine.

MITIGATIONS

One should always keep in mind basic security practices when developing any application. Be it a SQL server, web server, or anything else, protect your network, protect your passwords, credentials and other sensitive data, and for goodness’ sake back up your systems. Though these security practices are outside the scope of this article, it is important to always reiterate them, as one can plainly see that each of these has played a part in our investigation. Beyond preventing SQL injection vulnerabilities, the system we tested should have hashed, or stored passwords and other sensitive information in some encrypted form, a firewall should have been in place to disallow MySQL access to the server from a remote site, web access should have been restricted to SSL,
and a regular back up policy should have been in place in case of malicious attack or accidental failure. Each of these simple security procedures would have closed the door on a wide attack surface, requiring the adversary to dig much deeper.

**non-descriptive messages**

When it comes to SQL injection vulnerabilities, a few general security best practices stand out as crucial best practices. One of these is to avoid descriptive error messages. With SQL database connections in particular, descriptive error messages can help guide an attacker as these messages leak information about the back-end database system, the structure of the tables, and other critical items. Websites should portray none but the most benign error messages, so as to maintain a positive user experience, while not giving an attacker an instruction manual.

**input sanitization**

Another security best practice critical to SQL injection vulnerabilities is input sanitization. Controlling user input is paramount, as this is precisely where an attack will originate. Special characters that could be used to manipulate the system should be rejected, discarded, or escaped, so that they become harmless. Regarding SQL injection, the single and double quote characters are hugely problematic if not sanitized properly. Other languages regard brackets, braces, parenthesis, semicolons, etc., as special characters, and all non-essential characters should be scrubbed from any input. MySQL offers an escape option through the function `mysql_real_escape_string()`, which can be used to sanitize input when building a dynamic query string.

**prepared statements**

Many SQL systems allow parameterized statements, a.k.a. “prepared” statements, to enable queries to be executed more efficiently. A prepared statement resembles a function declaration, taking user input as arguments, which are passed as parameters to the SQL statement at the time the query is run. The benefit here is that user input is always treated as such, and is never concatenated with, or allowed to be, executable query string data. We recommend using prepared statements whenever and wherever possible, above all other methods, in order to best prevent SQL injection attacks.

**stored procedures**

Stored procedures are SQL statements created inside the database itself, executed as subroutines. They take user input as parameters, can incorporate filtering or other access control methods, and centralize all SQL statement creation. The centralization of all statements allows for easy first or third-party review, as well as checking that proper sanitization methods are employed. Care
should be taken however, as poorly written stored procedures may still be vulnerable to SQL injection attacks.

**principle of least privilege**

The Principle of Least Privilege commands that a SQL server be run only with the minimum set of permissions needed to perform its function. In our case study, both the MySQL service and the user “www” with access to the database had unnecessary privileges, and we were able to leverage this. The web service access should have been restricted only to the tables it required, and if stored procedures were implemented, should have been restricted only to those specific queries. Furthermore, the MySQL service should only have had access to the directory with the database files.

**security audits**

Formal testing should be an integral part of all web application development. A strong test plan and methodology is important, and when possible it is recommended to have a third-party security audit performed at each stage of the development process to assure that design and implementation flaws such as the ones outlined in this article don’t surface after deployment. An outside resource that specializes in security audits may be useful in finding security vulnerabilities that the original developers did not consider, and periodic security audits should be conducted as code is changed during maintenance.

**CONCLUSION**

At its base, SQL injection works by inserting into web forms data that is not expected and that extends unto the SQL query that the back-end system expects. By carefully crafting the input an attacker may be able to gain information about the database (for later attacks), and get data from, as well as possibly modify, data in the database. In the perspective of CIA (Confidentiality, Integrity and Availability) by far the greatest loss seems to be confidentiality, viz. revealing of private, sensitive or secret data. However, integrity is of equal importance and one can envision how availability is affected when data is lost.

We have demonstrated through this case study how SQL injection can be devastating, even through a minimal set of inputs. We strongly urge all developers to follow the best practice guidelines described in this article, and when possible, have their projects audited for security vulnerabilities before deployment.