Web Applications Security: SQL Injection Attack

S. C. Kothari

CPRE 556: Lecture 8, February 2, 2006

Electrical and Computer Engineering Dept.
Iowa State University

SQL Injection: What is it

- A technique for exploiting web applications that use client-supplied data in SQL queries, but without first stripping potentially harmful special characters.
- Despite being simple to protect against, many web applications are vulnerable to the attack.
- Similar to the Buffer Overflow, the attack is possible because the input is not properly checked.
- While similar, the buffer overflow and SQL Injection require very different checks for inputs.

Rain Forest Puppy

- Description on Linux Security Site: He's nice-looking, polite and very intelligent. He goes by the name Rain Forest Puppy, RFP for short. It's a name that might suggest environmental leanings, but that would be a serious miscalculation. RFP may turn out to be the software industry's worst nightmare.
- He is credited with the earliest SQL Injection attack. He used a vulnerability in the wwwthreads package to gain administrative access and some 800 passwords to PacketStorm's discussion forum.
- RFP Exploits: http://lists.virus.org/isn-0012/msg00041.html
- Interview with RFP: http://www.safemode.org/rfp.html

SQL Injection by Example

- Next, we will show how an hacker uses SQL injection.
- The examples are from the paper: SQL Injection Attacks By Example, http://www.unixwiz.net/techtips/sql-injection.html

The Scenario

- Attacker has no prior knowledge of the application or access to the source code.
- The login page had a traditional username-and-password form, but also an email-me-my-password link.
- The attacker decided to exploit the latter link.
- Game plan: discover the internals of the system by submitting different inputs.
- It takes some intelligent guesswork.

Initial Guess

- The attacker speculates 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 quotation marks around it set it off as a literal string.

Step 1: check if input is sanitized

- Objective: check if the web application constructs an SQL string literally without sanitizing.
- The way to check: enter a single quote as part of the data. The attacker enters `steve@unixwiz.net'` - note the closing quote mark. This yields:

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

- The SQL parser find the extra quote mark and aborts with a syntax error.
- The error response is often a dead giveaway that user input is not being sanitized properly and that the application is ripe for exploitation.

Step 1: Output

- The web application responds with a 500 error (server failure).
- This suggests that the "broken" input is actually being parsed literally.
- The attacker knows that he/she has good opportunities ahead.

Step 2: exploit `WHERE` clause

- Objective: give legal input and learn more.
- Since the data appears to be going into the `WHERE` clause, change the nature of that clause *in an SQL legal way* and see what happens. Enter `anything' OR 'x'='x`

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

- 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.

Step 2: Output

- Unlike the "real" query, which should return only a single item each time, this version is likely to 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.
- In this case, the web application responds:

  Your login information has been mailed to

  random.person@example.com

- The attacker guesses that it's the *first* record returned by the query.

The attacker knows ..

- He can manipulate the query to his own ends.
- He has observed two different responses:
  - Server error
  - "Your login information has been mailed to *email*"
- The first response is for bad SQL while the latter is to a well-formed SQL. This distinction will be very useful when trying to guess the structure of the query.

Step 3: guessing `field` names

- The attacker guesses `email` as the name of the field.
- The way to check: This yields:

```
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.
- The attacker does not care about matching the email address (which is why he uses 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.

Step 3: Output

- If the server responds with an error message, it means the SQL is malformed and a syntax error was thrown. It's most likely due to a bad field name.
- Any kind of valid response implies that the field name is correct.
- This is the case whether we get the "email unknown" or "password was sent" response.
- The use of the `AND` conjunction instead of `OR` is intentional. The attacker does not want random users inundated with "here is your password" emails from the web application.
- Using the `AND` conjunction with an email address that couldn't ever be valid, the attacker makes sure that the query will always return zero rows.

Going Forward

- The attacker may have to try different `field` names `email_address` or `mail` or the like. This process could involve quite a lot of guessing.
- Let us suppose that the application responds: "email address unknown"
- Now the attacker knows that the email address is stored in a field `email`.

Step 4: guessing more `field` names

- Next the attacker guesses other field names like password, user ID, name and validates his guesses by submitting queries one at a time for each guess. For example:

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

- Suppose the attacker found other field names: email, passwd, login_id, full_name.

Step 5: guessing the `Table` name

- There are several approaches. This one relies on a `subselect`. For example, A standalone query: `SELECT COUNT(*) FROM tabname` returns the number of records in that table, and of course fails if the table name is unknown.

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

- The attacker does not care how many records are there, only whether the table name is valid or not.

Step 5: Output

- Let us suppose that by iterating over several guesses, the attacker 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 worksfor tables that are actually part of this query, not merely that the table exists. For example:

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

The attacker knows ..

- At this point the attacker has a partial idea of the structure of the **members** table.
- He knows of one username: the random member who got initial "Here is your password" email.
- Note that the attacker never received the message itself, only the address it was sent to.
- Next, the attacker wants to get some more names to work with, preferably those likely to have access to more data.

Step 6: Using the LIKE clause to get more names

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

Guessing the password

The attacker 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 log files, account lockouts, or other devices that would substantially impede such efforts, but because of the non-sanitized inputs, the attacker has another avenue that is much less likely to be so protected.

Step 7: guess the password

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

- The attacker knows he has found the password when he receives the "your password has been mailed to you" message. His target has now been tipped off, but he does have his password.

Important Discovery: The database isn't read only

- So far, the attacker has done nothing but **query** the database, and even though a SELECT is read only, that doesn't mean that SQL is.
- Drastic example:

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

- This one attempts to drop (delete) the entire **members** table.
- This shows that not only can the attacker run separate SQL commands, but he can also modify the database.

Step 8: add a new member

- The attacker adds a new record to that table and simply logs in directly with his newly-inserted credentials.

```
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');-- ';
```

Attacker faces roadblocks

1. There may not be enough room in the web form to enter this much text directly.
2. The web application user might not have **INSERT** permission on the **members** table.
3. There are other fields in the **members** table, and some may *require* initial values, causing the **INSERT** to fail.
4. The application itself might not behave well due to the auto-inserted NULL fields.
5. A valid "member" might require not only a record in the **members** table, but associated information in other tables (say, "access rights"), so adding to one table alone might not be sufficient.

Other Approaches

- 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 web server is inevitable.
- This particular application provided a rich post-login environment that was enough for the attacker. In other cases the attacker can 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.

What steps to take for ensuring security

- **Sanitize the input**
  - Insure that inputs do not contain dangerous codes, whether to the SQL server or to HTML itself.
  - Check: Input from users, parameters from URL, values from cookie
  - Strip out "bad stuff", such as quotes or semicolons or escapes. It is hard 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.
  - Rather than "remove known bad data", it's better to "remove everything but known good data“. Should consult internet message standard RFC2822.

What steps to take for ensuring security

- **Use bound parameters (the PREPARE statement)**
  - There may be fields that must be allowed to contain these "dangerous“ characters. Another approach is the use of **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. An example in perl:

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

What steps to take for ensuring security

- **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.
  - 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.

What steps to take for ensuring security

- **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).
  - 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 withbound variables that provide this protection.

What steps to take for ensuring security

- **Isolate the web server**
  - 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.

What steps to take for ensuring security

- **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 - ifpossible - 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.

Expected Work After the Lecture

- Learning more about one or more of the following:
  - Explores and report on tools for web application security such the SPI Dynamics' WebInspect™. How do they work? What are their capabilities? Refer to the guidelines on the course website for exploring tools. http://www.spidynamics.com/products/App_Dev/WI/index.html
  - Read and prepare a presentation on how to build secure web applications, refer to the free guide, http://unc.dl.sourceforge.net/sourceforge/owasp/guide_v2a1.pdf

Expected Work After the Lecture

- You should report your interesting findings in class or by sending me an email.
- If you send email, identify the lecture number and your last name in the subject line (e.g. Lecture 2-Smith) and also within your message. Give proper references for each of your findings.
- This will be considered as a part of class participation.

References

- The primary references are in the main body of the presentation.
- White paper on SQL Injection, http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf
- A SQL Injection Walkthrough, http://www.securiteam.com/securityreviews/5DP0N1P76E.html