SQL Injection

Slide credit: York university

Agenda

- *Code injection* vulnerability - untrusted input inserted into query or command
  - Attack string alters intended semantics of command
  - Ex: ***SQL Injection***
    - unsanitized data used in query to back-end database
- SQL Injection Attack Scenarios
  - **First-order SQL Injection**
    - Type 1: compromises user data
    - Type 2: modifies critical data
  - **Second-order SQL Injection**
    - Two-phases attach (first store data, then exploit)

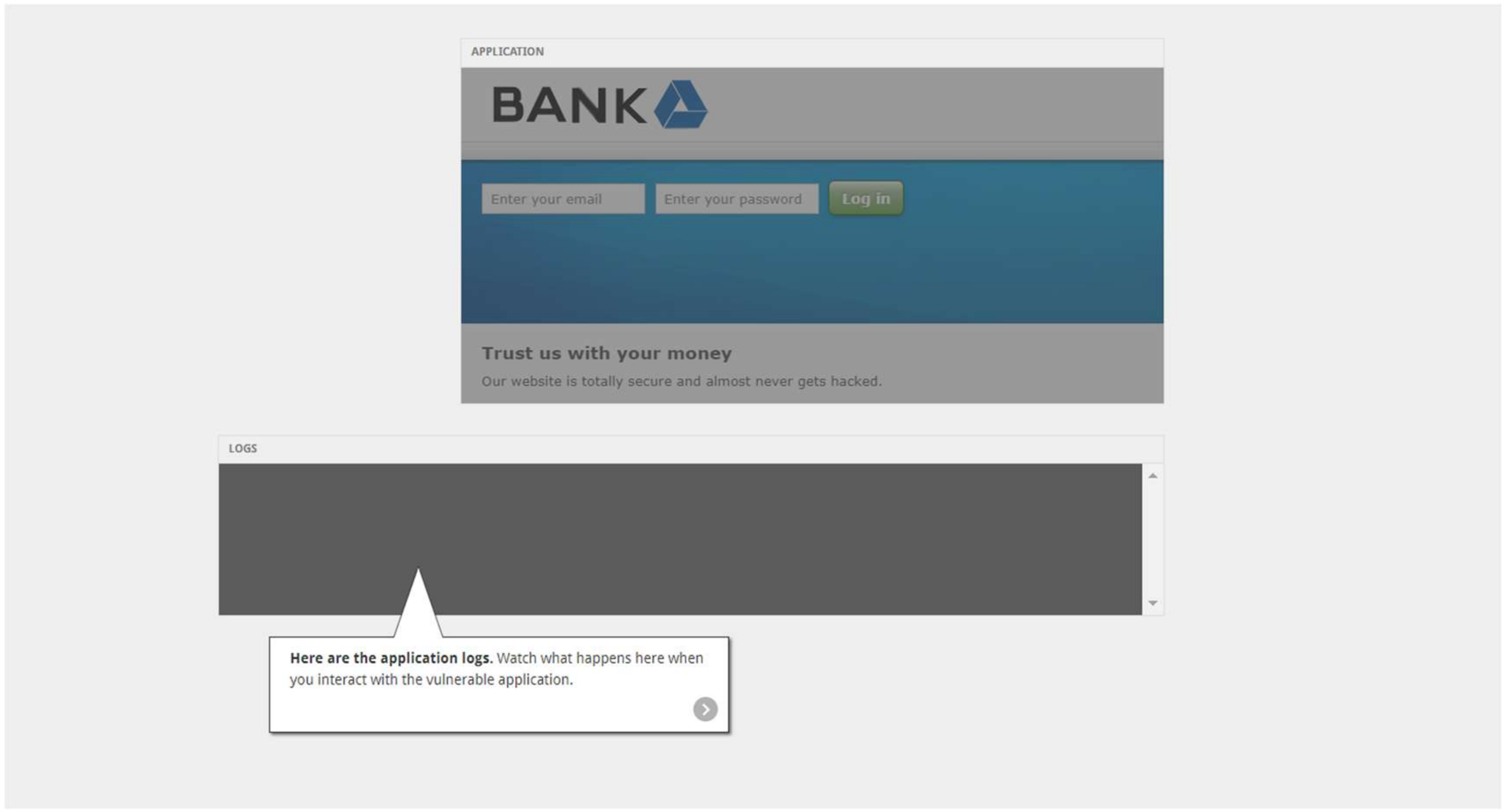
SQL Injection Impact

- CardSystems, credit card payment processing ruined by SQL Injection attack in June 2005
  - 263,000 credit card #s stolen from its DB
  - #s stored unencrypted, 40 million exposed
- Heartland Payment Systems (2005-2007)
  - 130 million cards were hacked
  - Hackers sentenced for SQL injections that cost $300 million
- More examples:
  - http://en.wikipedia.org/wiki/SQL_injection#Examples
  - https://moneywise.com/a/worst-data-breaches-of-the-century

SQL Injection
Attack Scenarios

First-order SQL Injection example

- https://www.hacksplaining.com/exercises/sql-injection#/start



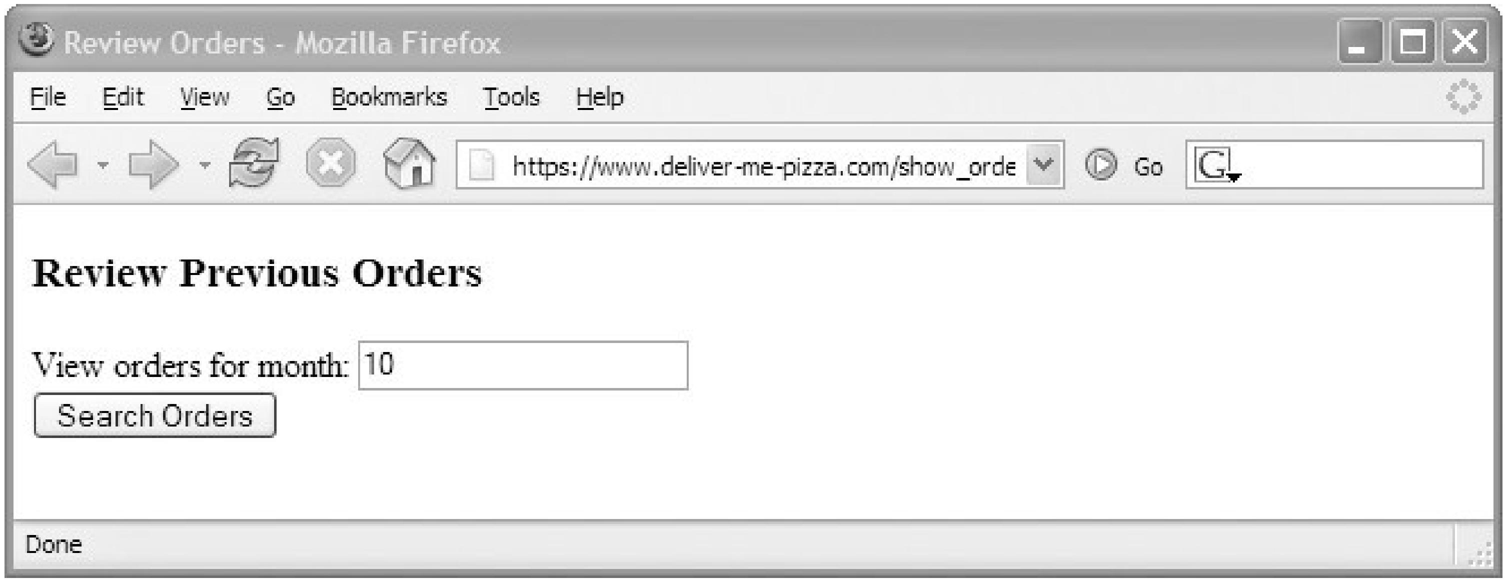
Trust us with your money

Our website is totally secure and almost never gets hacked.

Here are the application logs. Watch what happens here when you interact with the vulnerable application.

First-order SQL Injection (1/6)

- Ex: Pizza Site Reviewing Orders
  - Form requesting month # to view orders for


Review Orders - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
https://www.deliver-me-pizza.com/show_orde
Go
Search Orders
Done

Review Previous Orders

View orders for month: 10

  - HTTP request:

`https://www.deliver-me-pizza.com/show_orders?month=`**`10`**

First-order SQL Injection (2/6)

- App constructs SQL query from parameter:

```
sql_query = "SELECT pizza, toppings, quantity, order_day " +
            "FROM orders " +
            "WHERE userid=" + session.getCurrentUserId() + " " +
            "AND order_month=" + request.getParamenter("month");
```

Normal SQL Query

```
SELECT pizza, toppings, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=10
```

- Type 1 Attack: inputs `month='0 OR 1=1'` !
- Goes to encoded URL: (space -> `%20`, = -> `%3D`)

https://www.deliver-me-pizza.com/show_orders?**month=0%20OR%201%3D1**

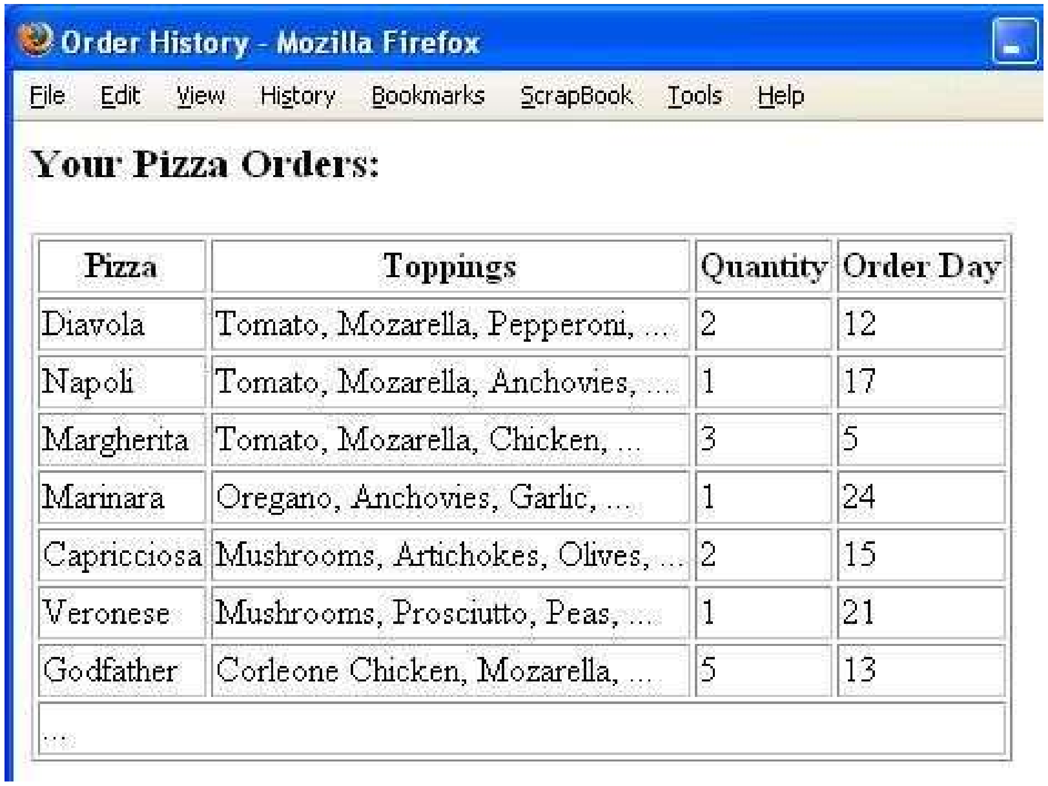
First-order SQL Injection (3/6)

Malicious Query

```
SELECT pizza, toppings, quantity, order_day
FROM orders
WHERE userid=4123 AND order_month=0 OR 1=1
```

- WHERE condition is always true!
  - AND precedes OR
  - Type 1 Attack: Gains access to other users' private data!

All User Data Compromised


Order History - Mozilla Firefox
File Edit View History Bookmarks ScrapBook Tools Help

Your Pizza Orders:

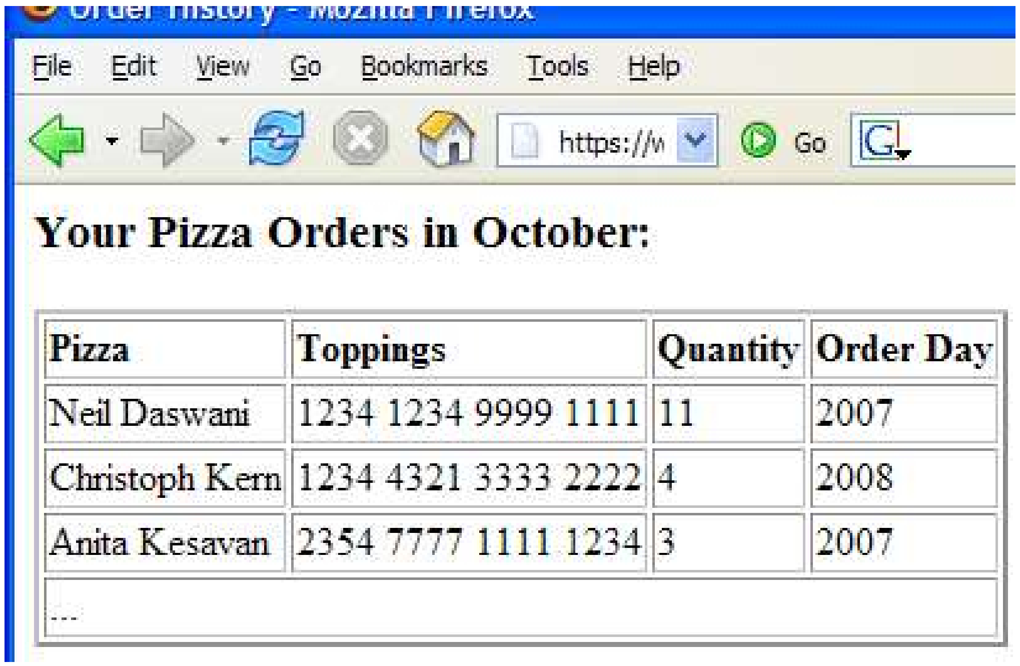
Pizza	Toppings	Quantity	Order Day
Diavola	Tomato, Mozarella, Pepperoni, ...	2	12
Napoli	Tomato, Mozarella, Anchovies, ...	1	17
Margherita	Tomato, Mozarella, Chicken, ...	3	5
Marinara	Oregano, Anchovies, Garlic, ...	1	24
Capricciosa	Mushrooms, Artichokes, Olives, ...	2	15
Veronese	Mushrooms, Prosciutto, Peas, ...	1	21
Godfather	Corleone Chicken, Mozarella, ...	5	13
...			

First-order SQL Injection (4/6)

More damaging attack: attacker sets

```
month='0 AND 1=0
UNION
SELECT cardholder, number, exp_month, exp_year
FROM creditcards'
```

- Attacker is able to
  - Combine 2 queries
  - 1st query: empty table (where fails)
  - 2nd query: credit card #s of all users


Order History - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
https://w
Go

Your Pizza Orders in October:

Pizza	Toppings	Quantity	Order Day
Neil Daswani	1234 1234 9999 1111	11	2007
Christoph Kern	1234 4321 3333 2222	4	2008
Anita Kesavan	2354 7777 1111 1234	3	2007
...			

First-order SQL Injection (5/6)

- Even worse, attacker sets

```
month='0;
DROP TABLE creditcards;'
```

- Then DB executes
  - Type 2 Attack: Removes `creditcards` from schema!
  - Future orders fail: DoS!

```
SELECT pizza, toppings,
quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0;
DROP TABLE creditcards;
```

- Problematic Statements:
  - Modifiers: `INSERT INTO admin_users VALUES ('hacker',...)`
  - Administrative: shut down DB, control OS…

First-order SQL Injection (6/6)

- Injecting String Parameters: Topping Search

```
sql_query =
  "SELECT pizza, toppings, quantity, order_day " +
  "FROM orders " +
  "WHERE userid=" + session.getCurrentUserId() + " " +
  "AND topping LIKE '%" + request.getParamenter("topping") + "%' ";
```

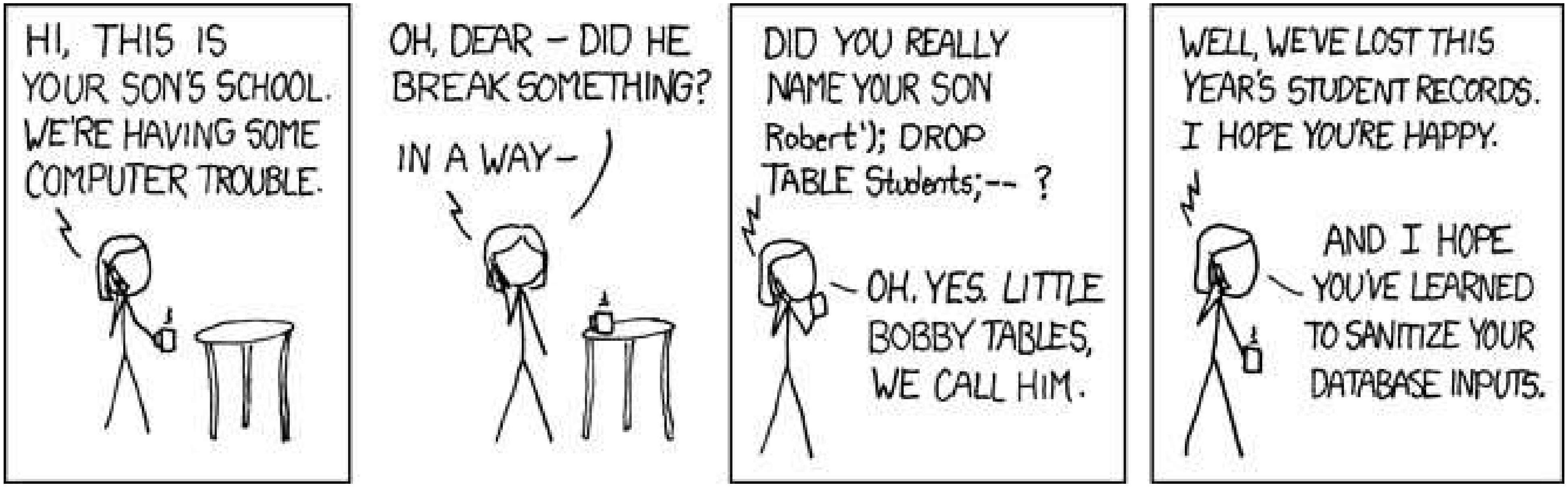
- Attack searches for:

  `brzfg%'; DROP table creditcards; --`

- Query evaluates as:
  - SELECT: empty table
  - -- comments out end
  - Credit card info dropped

```
SELECT pizza, toppings,
quantity, order_day
FROM orders
WHERE userid=4123
AND topping LIKE '%brzfg%';
DROP table creditcards; --%'
```

Sanitize your Database Inputs


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Source: http://xkcd.com/327/

Second-Order SQL Injection (1/2)

- *Second-Order SQL Injection*: data stored in database is later used to conduct SQL injection
  - Common if string escaping is applied inconsistently
  - Ex: `o'connor` updates passwd to `SkYn3t`

```
new_passwd = request.getParameter("new_passwd");
uname = session.getUsername();
sql = "UPDATE USERS SET passwd='"+ escape(new_passwd) +
      "' WHERE uname='" + uname + "'";
```

  - `uname` not escaped, b/c originally escaped before entering into the DB, now inside our trust zone:

```
UPDATE USERS SET passwd='SkYn3t' WHERE uname='o'connor'
```

  - Query fails b/c **'** after **o** ends command prematurely

Second-Order SQL Injection (2/2)

- Even Worse: What if user set `uname=`**`admin'--`** !?

  ```
  UPDATE USERS SET passwd='cracked' WHERE uname='admin' --'
  ```

  – Attacker changes `admin`'s password to `cracked`
  – Has full access to `admin` account
  – Username avoids collision with real `admin`
  – `--` comments out trailing quote

- All parameters dangerous

Solutions

Slide credit: York university

Solutions

A. Blacklisting
B. Whitelisting over Blacklisting
C. Input Validation & Escaping
D. Use Prepared Statements & Bind Variables

A. Blacklisting

- Eliminating quotes enough (blacklist them)?

```
sql_query =
"SELECT pizza, toppings, quantity, order_day " +
"FROM orders " +
"WHERE userid=" + session.getCurrentUserId() + " " +
"AND topping LIKE
'kill_quotes(request.getParamenter("topping")) + "%'";
```

- `kill_quotes` (Java) removes single quotes:

```
String kill_quotes(String str) {
  StringBuffer result = new StringBuffer(str.length());
  for (int i = 0; i < str.length(); i++) {
    if (str.charAt(i) != '\'')
      result.append(str.charAt(i));
  }
  return result.toString();
}
```

A. Pitfalls of Blacklisting

- Filter quotes, semicolons, whitespace, and…?
  - Could always miss a dangerous character
  - Blacklisting not comprehensive solution
  - Ex: `kill_quotes()` can't prevent attacks against numeric parameters

- May conflict with functional requirements
  - Ex: How to store O'Brien in DB if quotes blacklisted?

B. Whitelisting

- *Whitelisting* – only allow input within well-defined set of safe values
  - set implicitly defined through *regular expressions*
  - *RegExp* – pattern to match strings against

- Ex: `month` parameter: non-negative integer
  - RegExp: `^[0-9]*$` - 0 or more digits, safe subset
    - The `^`, `$` match beginning and end of string
    - `[0-9]` matches a digit,
    - `*` specifies 0 or more

C. Input Validation and Escaping

- Could escape quotes instead of blacklisting
- Ex: insert user `o'connor`, password `terminator`

```
sql = "INSERT INTO USERS(uname,passwd) " +
      "VALUES (" + escape(uname)+ "," +
      escape(password) +")";
```

  - `escape(o'connor) = o''connor`

```
INSERT INTO USERS(uname,passwd) VALUES ('o''connor','terminator');
```

- Like `kill_quotes`, only works for string inputs
- Numeric parameters could still be vulnerable

D. Prepared Statements & Bind Variables

- Metachars (e.g. quotes) provide distinction between data & control in queries
  - most attacks: data interpreted as control
  - alters the semantics of a query
- *Bind Variables*: ? placeholders guaranteed to be data (not control)
- *Prepared Statements* allow creation of static queries with bind variables
  - Preserves the structure of intended query
  - Parameters not involved in query parsing/compiling

Java Prepared Statements

```
PreparedStatement ps =
db.prepareStatement("SELECT pizza, toppings, quantity, order_day "
                + "FROM orders WHERE userid=? AND order_month=?");
ps.setInt(1, session.getCurrentUserId());
ps.setInt(2, Integer.parseInt(request.getParamenter("month")));
ResultSet res = ps.executeQuery();
```

Bind Variable:
Data Placeholder

- Query parsed without parameters
- Bind variables are **typed**: input must be of expected type (e.g. int, string)