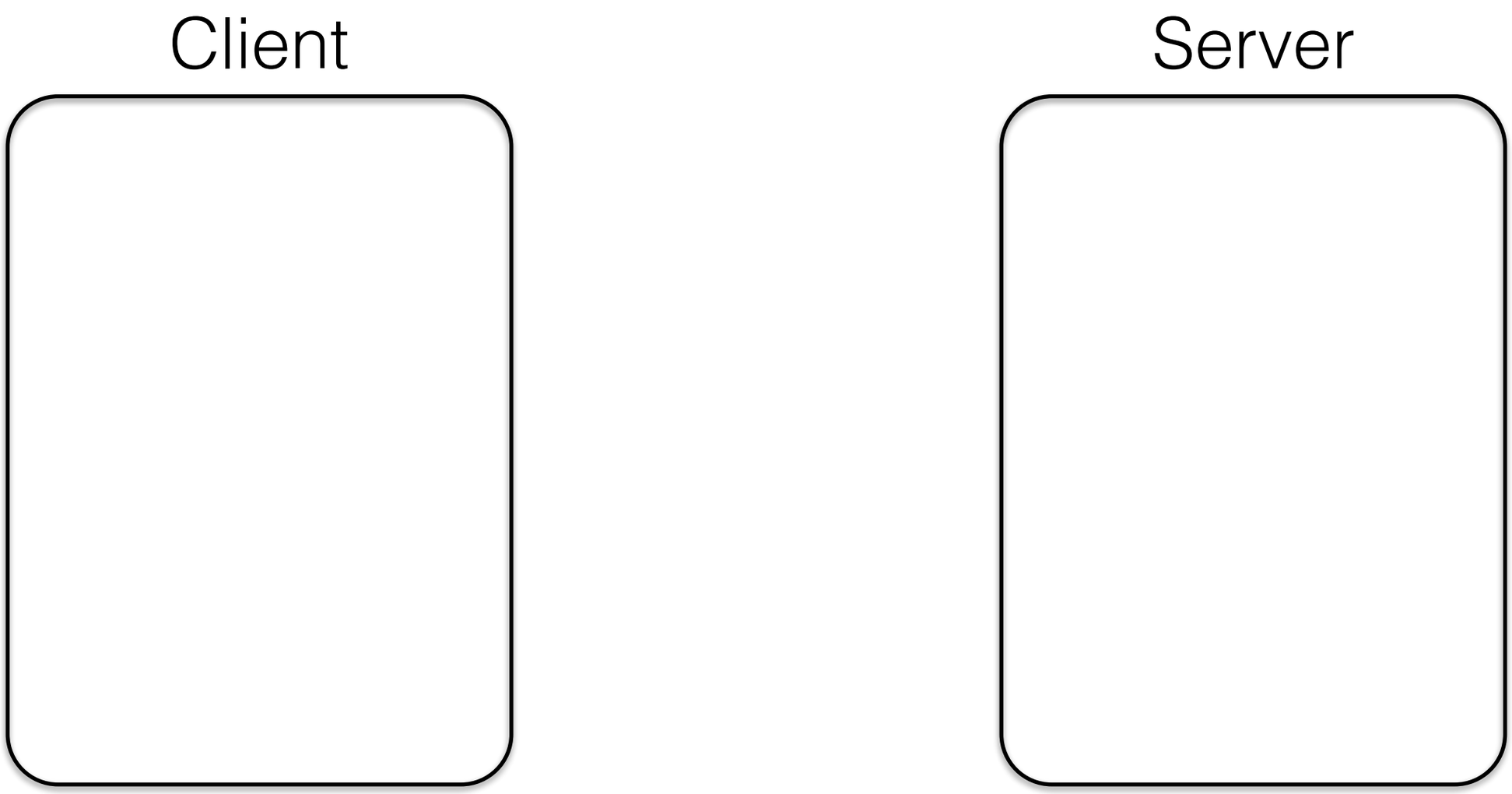
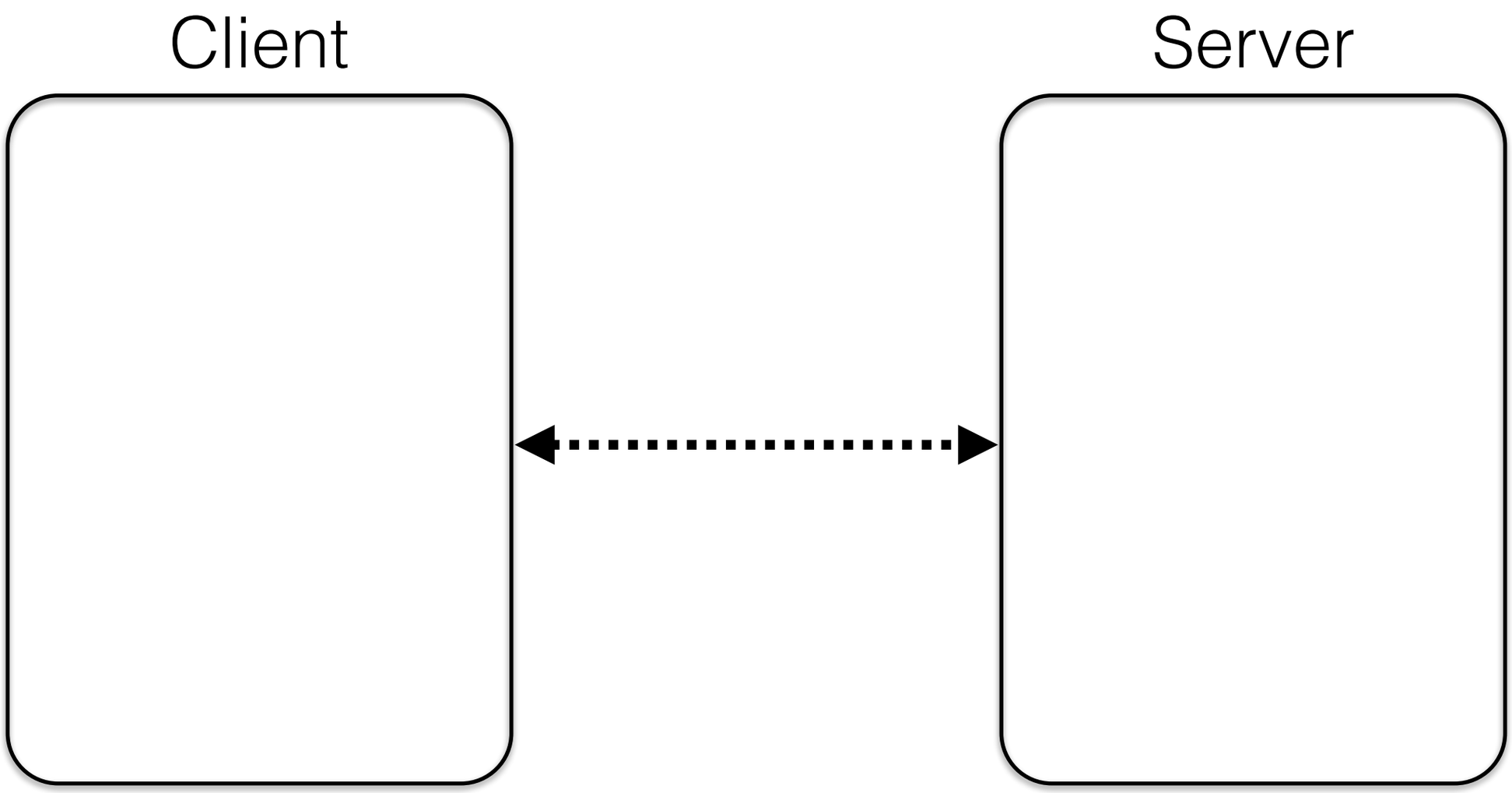
WEB SECURITY:
SQL INJECTION

CMSC 414
FEB 15 2018

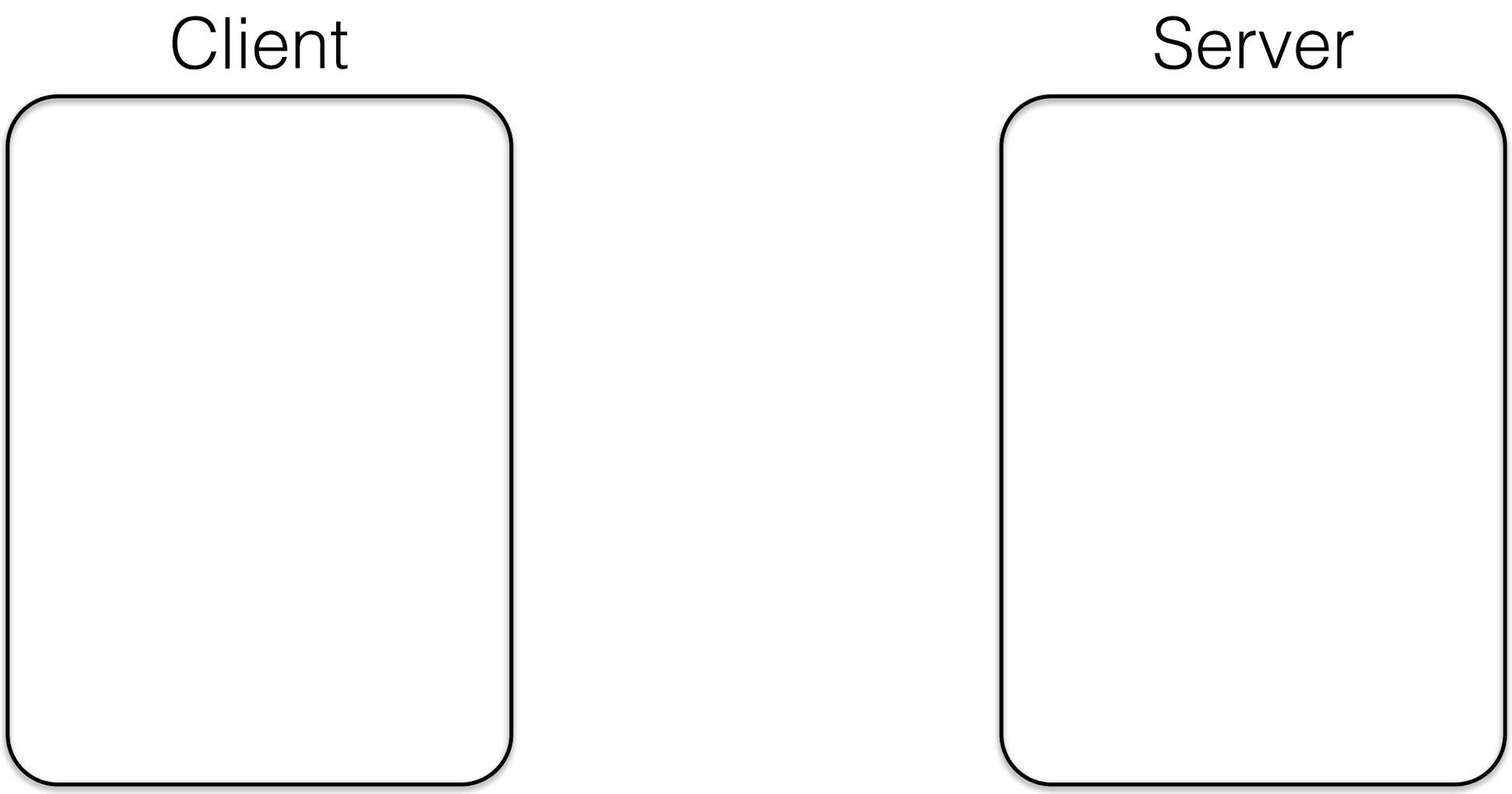
A very basic web architecture



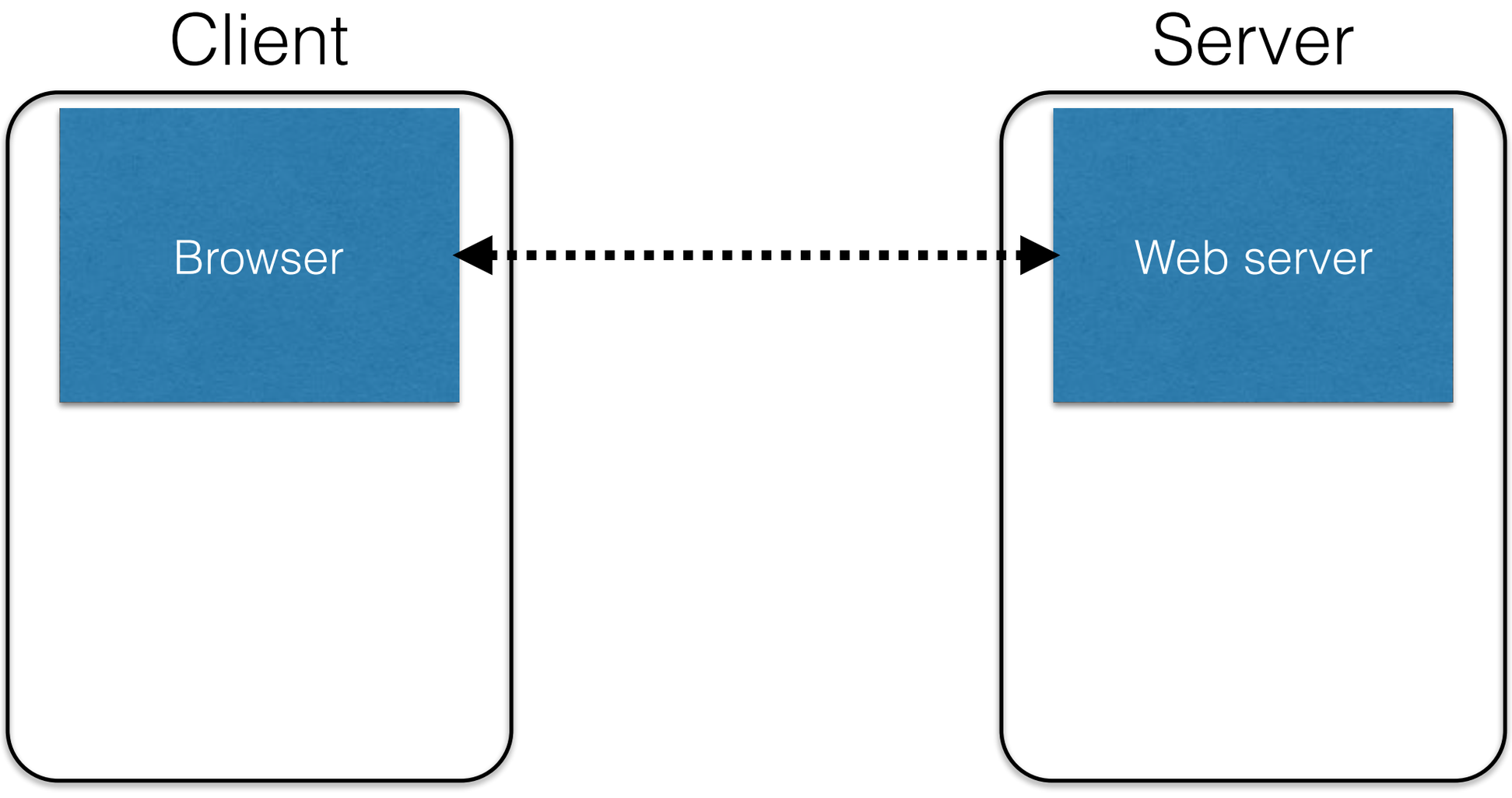
A very basic web architecture



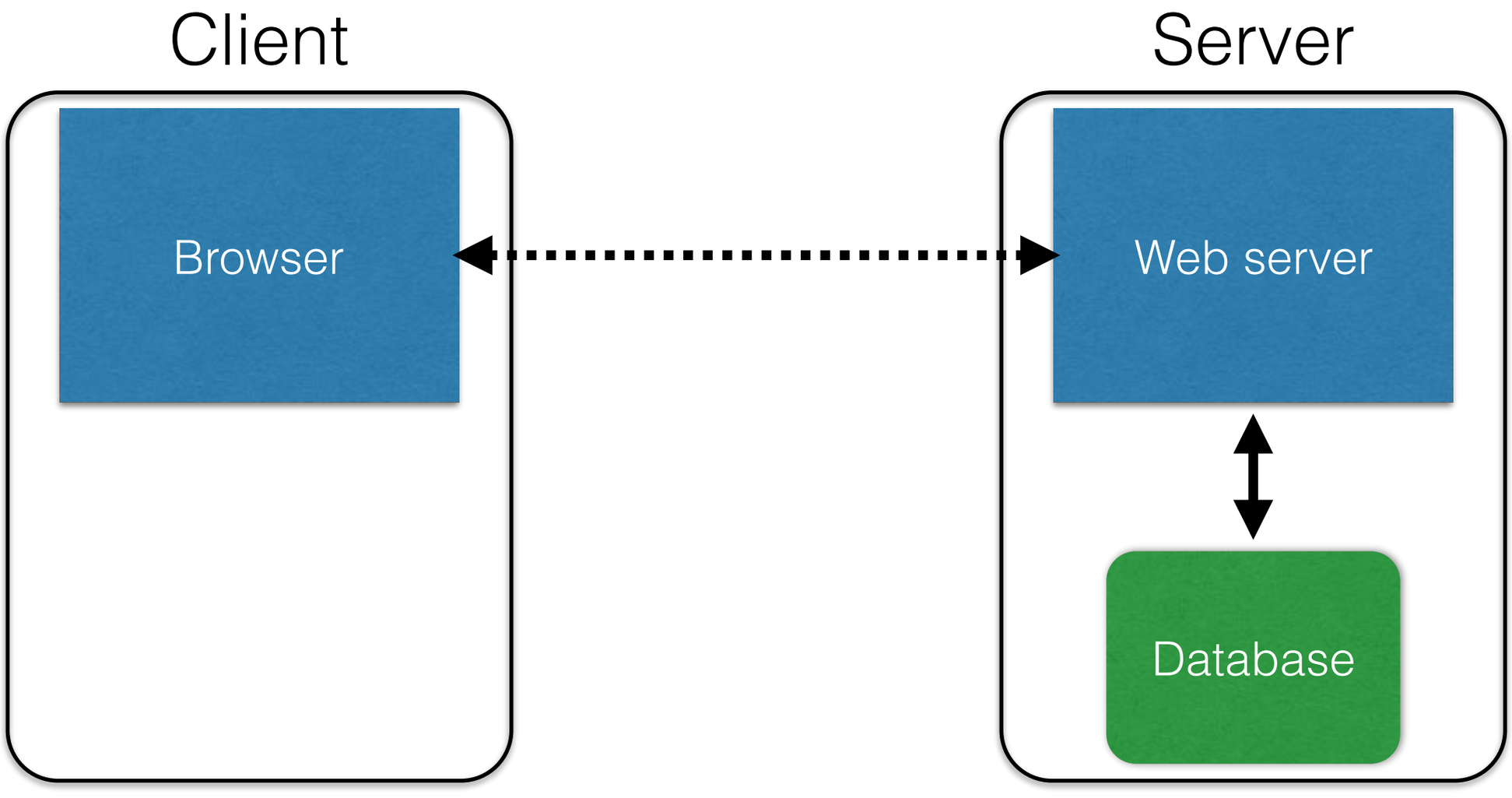
A very basic web architecture



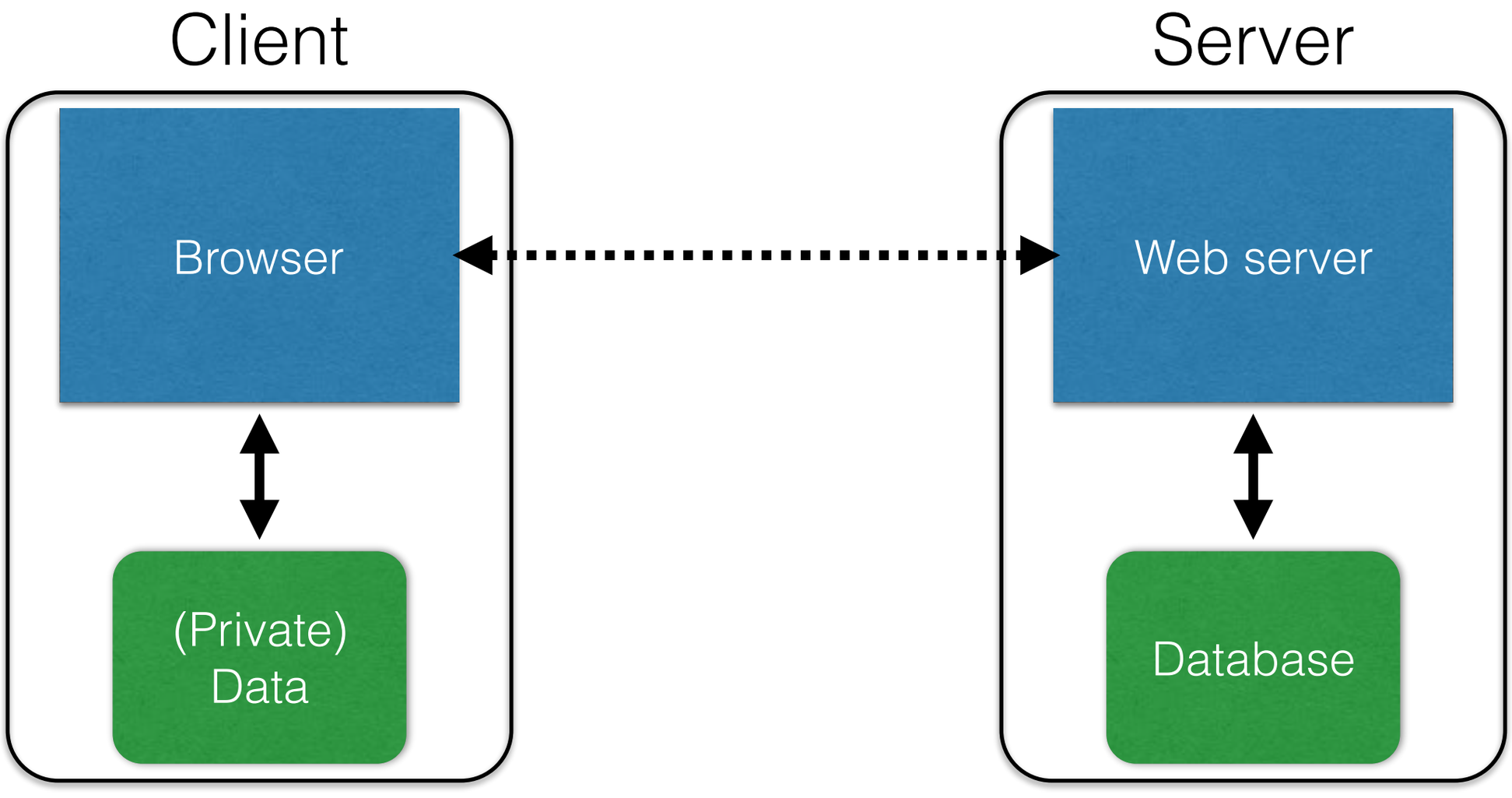
A very basic web architecture



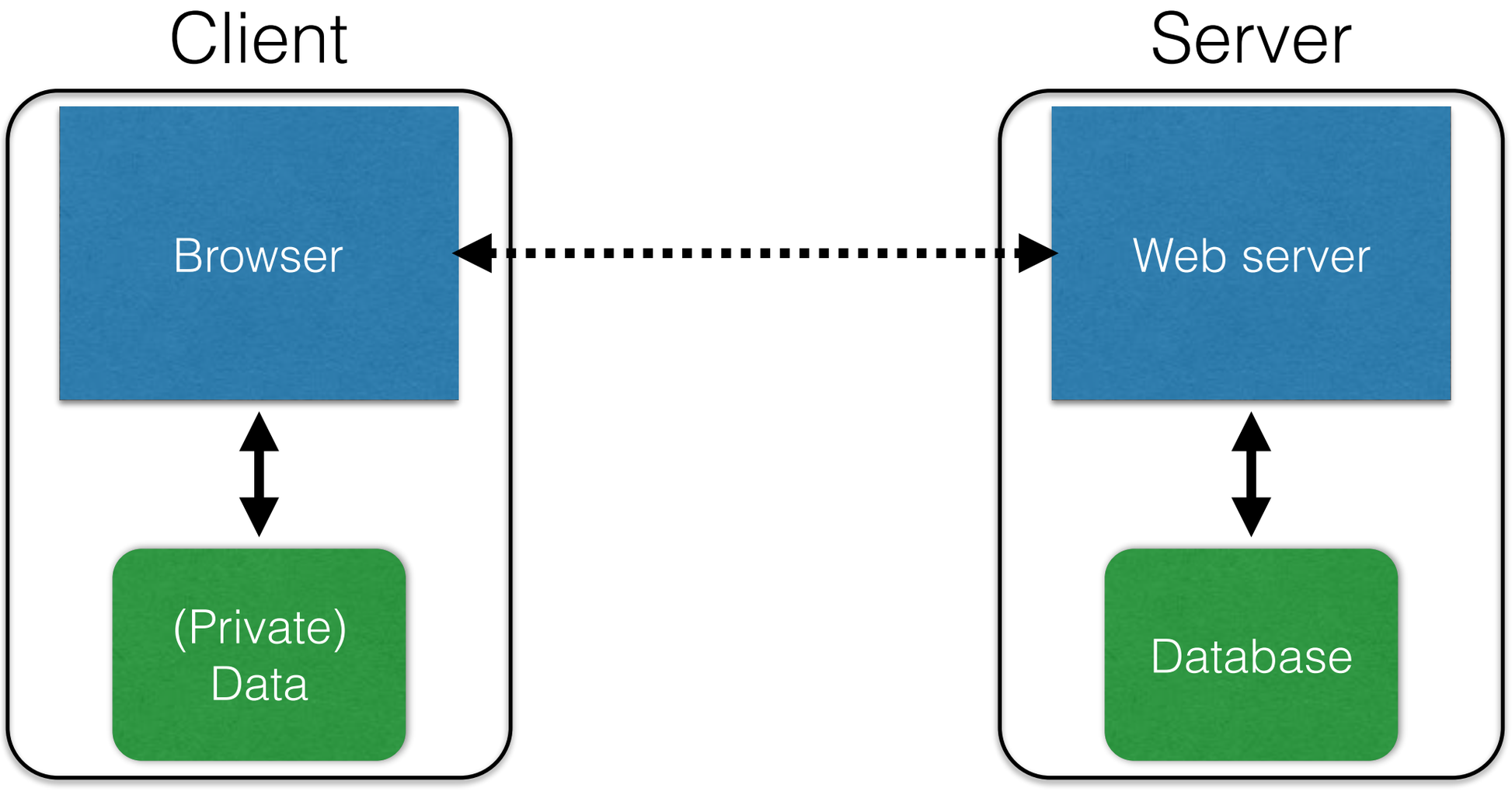
A very basic web architecture



A very basic web architecture



A very basic web architecture



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

SQL security

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

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

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	bouncer@pp.com	a0u23bt
Charlie	M	32	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

Row (Record)

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

Database transactions

Transactions are the unit of work on a database

Database transactions

Transactions are the unit of work on a database

"Give me everyone in the User table who is listed as taking CMSC414 in the Classes table"

"Deduct $100 from Alice; Add $100 to Bob"

Database transactions

Transactions are the unit of work on a database

"Give me everyone in the User table who is listed as taking CMSC414 in the Classes table" 2 reads

"Deduct $100 from Alice; Add $100 to Bob" 2 writes

Database transactions

Transactions are the unit of work on a database

"Give me everyone in the User table who is listed as taking CMSC414 in the Classes table" 2 reads

1 transaction

"Deduct $100 from Alice; Add $100 to Bob" 2 writes

Database transactions

Transactions are the unit of work on a database

"Give me everyone in the User table who is listed as taking CMSC414 in the Classes table" 2 reads

1 transaction

"Deduct $100 from Alice; Add $100 to Bob" 2 writes

- Typically want ACID transactions
  - **A**tomicity: Transactions complete entirely or not at all
  - **C**onsistency: The database is always in a *valid* state (but not necessarily *correct*)
  - **I**solation: Results from a transaction aren't visible until it is complete
  - **D**urability: Once a transaction is committed, it remains, despite, e.g., power failures

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	aneifjask@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

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

28

```
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	readgood@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

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

28

```
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	readgood@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

```
SELECT Age FROM Users WHERE Name='Dee';
```
28

```
UPDATE Users SET email='readgood@pp.com'
    WHERE Age=32; -- this is a comment
INSERT INTO Users Values('Frank', 'M', 57, ...);
```

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	readgood@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93
Frank	M	57	armed@pp.com	ziog9gga

```
SELECT Age FROM Users WHERE Name='Dee';
```
28

```
UPDATE Users SET email='readgood@pp.com'
    WHERE Age=32; -- this is a comment
INSERT INTO Users Values('Frank', 'M', 57, ...);
```

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	readgood@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93
Frank	M	57	armed@pp.com	ziog9gga

```
SELECT Age FROM Users WHERE Name='Dee';
```
28

```
UPDATE Users SET email='readgood@pp.com'
    WHERE Age=32; -- this is a comment
INSERT INTO Users Values('Frank', 'M', 57, ...);
DROP TABLE Users;
```

SQL (Standard Query Language)

```
SELECT Age FROM Users WHERE Name='Dee';    28
UPDATE Users SET email='readgood@pp.com'
    WHERE Age=32; -- this is a comment
INSERT INTO Users Values('Frank', 'M', 57, ...);
DROP TABLE Users;
```

Server-side code

Website

“Login code” (php)

```
$result = mysql_query(“select * from Users
        where(name=‘$user’ and password=‘$pass’);”);
```

Suppose you successfully log in as $user
if this query returns any rows whatsoever

Server-side code

Website

“Login code” (php)

```
$result = mysql_query(“select * from Users
        where(name=‘$user’ and password=‘$pass’);”);
```

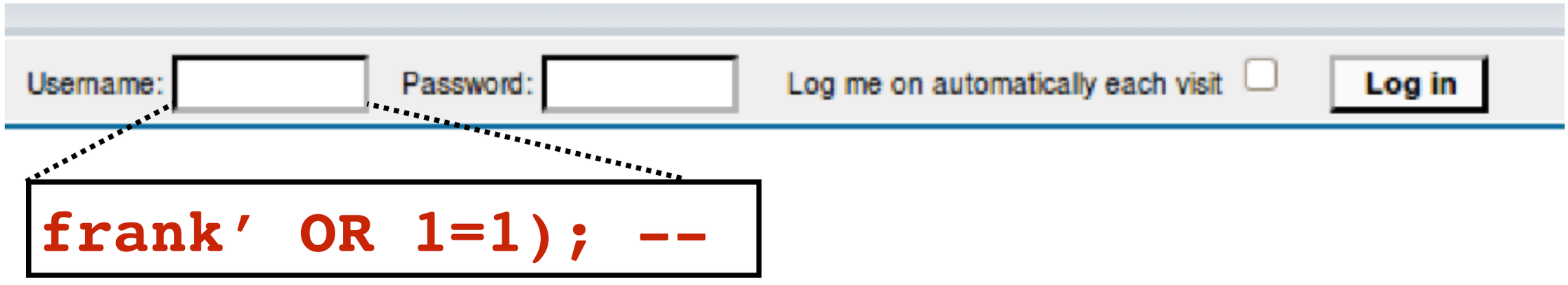
Suppose you successfully log in as $user
if this query returns any rows whatsoever

How could you exploit this?

SQL injection

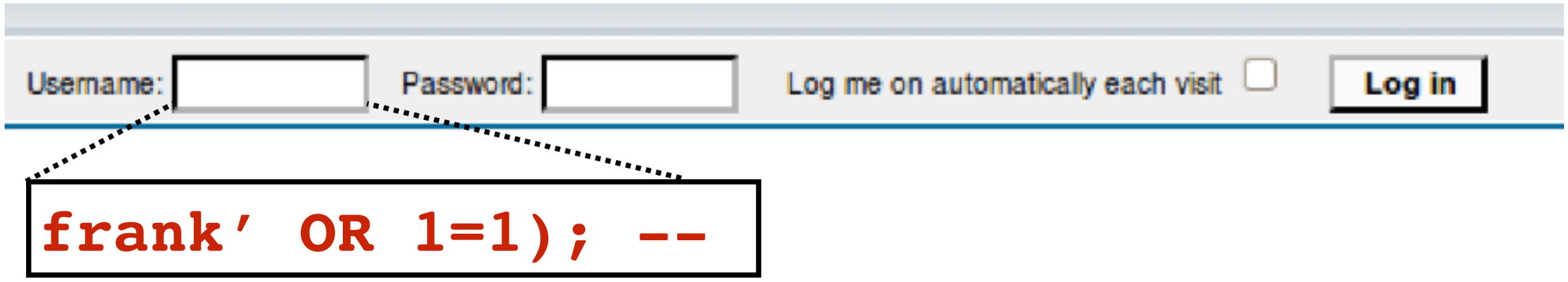

```
$result = mysql_query(“select * from Users
        where(name=‘$user’ and password=‘$pass’);”);
```

SQL injection




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

SQL injection




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


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

SQL injection


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

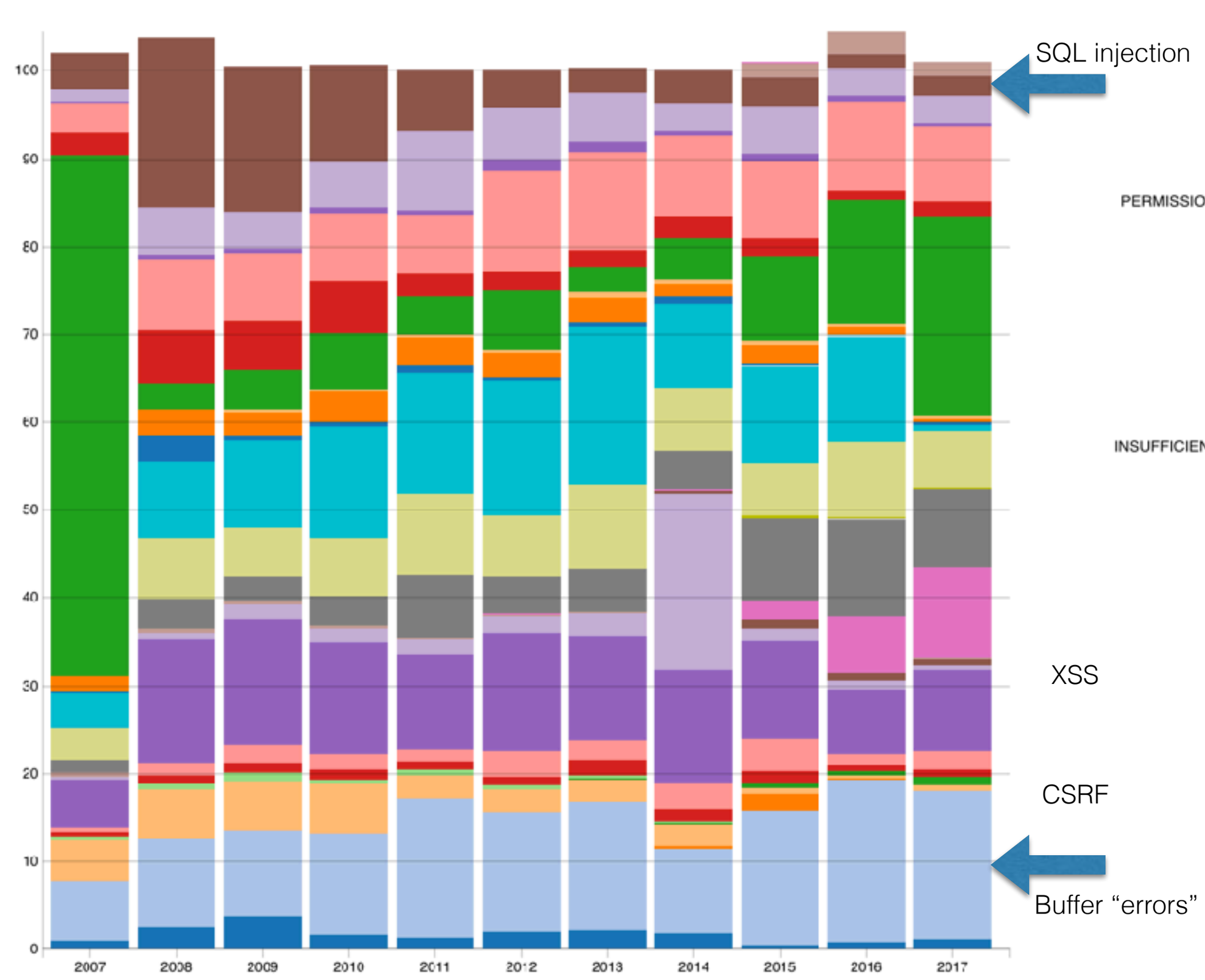
Can chain together statements with semicolon:
STATEMENT 1 ; STATEMENT 2

SQL injection


```
$result = mysql_query("select * from Users
        where(name='$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



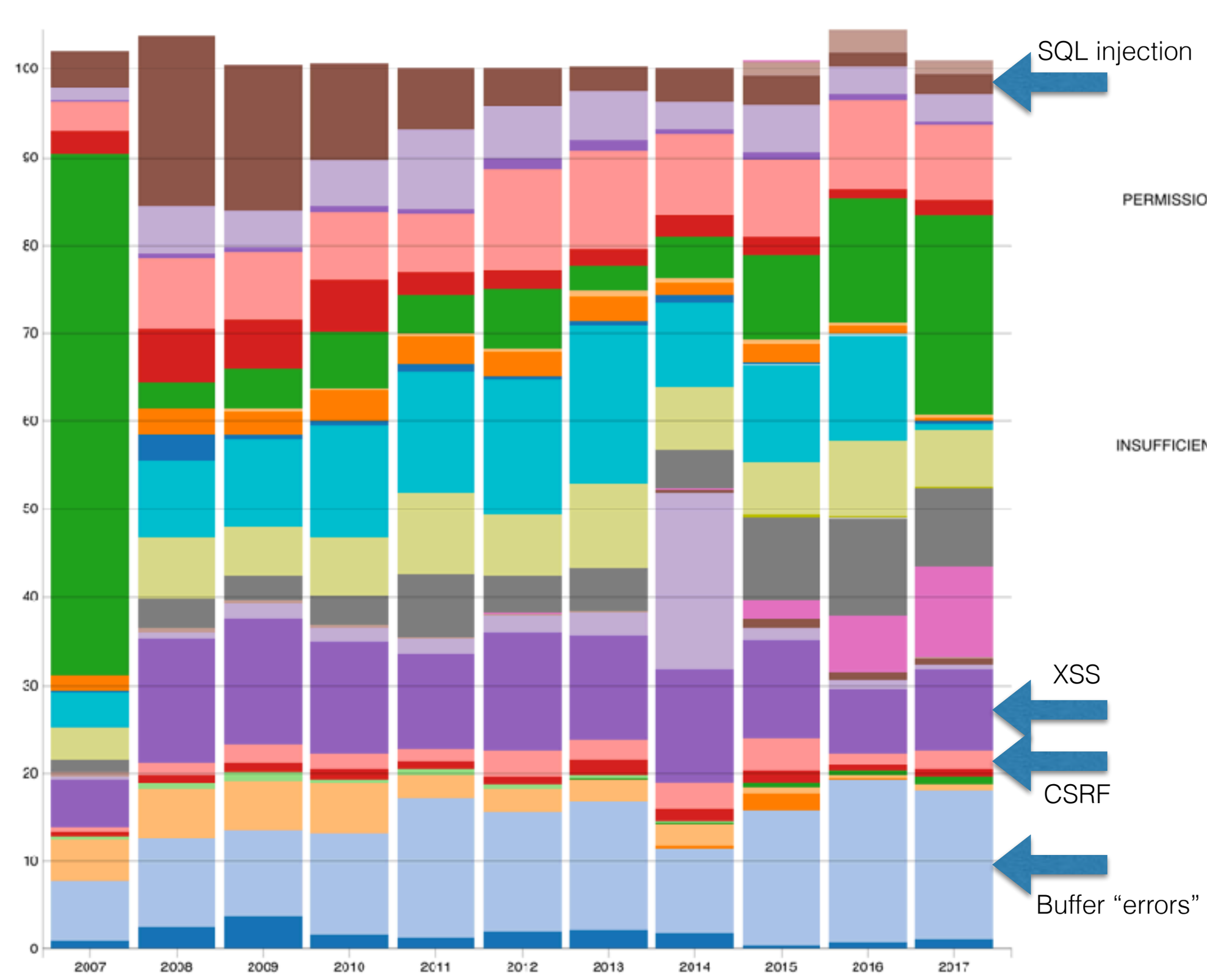
SQL injection

XSS

CSRF

Buffer “errors”

TIME AND STATE
SOURCE CODE
SECURITY FEATURES
SQL INJECTION
RESOURCE MANAGEMENT ERRORS
RACE CONDITIONS
PERMISSIONS, PRIVILEGES, AND ACCESS CONTROL
PATH TRAVERSAL
PATH EQUIVALENCE
OTHER
OS COMMAND INJECTIONS
NUMERIC ERRORS
LOCATION
LINK FOLLOWING
INSUFFICIENT VERIFICATION OF DATA AUTHENTICITY
INSUFFICIENT INFORMATION
INPUT VALIDATION
INJECTION
INFORMATION MANAGEMENT ERRORS
INFORMATION LEAK / DISCLOSURE
INDICATOR OF POOR CODE QUALITY
IMPROPER ACCESS CONTROL
FORMAT STRING VULNERABILITY
DATA HANDLING
CRYPTOGRAPHIC ISSUES
CROSS-SITE SCRIPTING (XSS)
CROSS-SITE REQUEST FORGERY (CSRF)
CREDENTIALS MANAGEMENT
CONFIGURATION
COMMAND INJECTION
CODE INJECTION
CODE
BUFFER ERRORS
AUTHENTICATION ISSUES



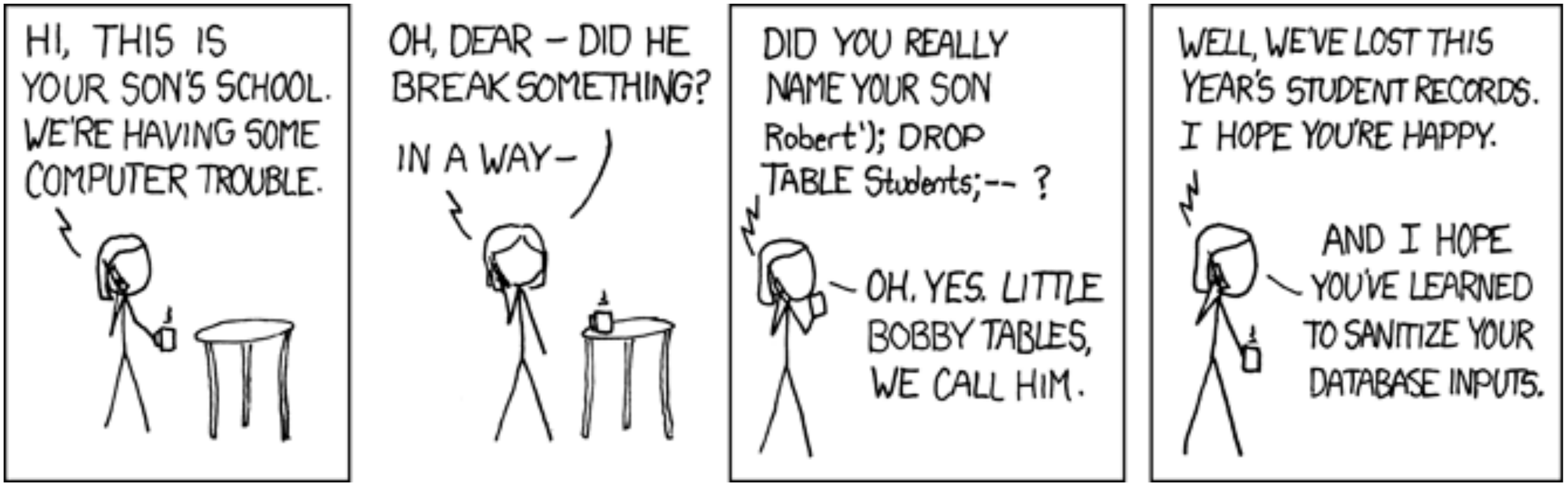
SQL injection

XSS

CSRF

Buffer “errors”

TIME AND STATE
SOURCE CODE
SECURITY FEATURES
SQL INJECTION
RESOURCE MANAGEMENT ERRORS
RACE CONDITIONS
PERMISSIONS, PRIVILEGES, AND ACCESS CONTROL
PATH TRAVERSAL
PATH EQUIVALENCE
OTHER
OS COMMAND INJECTIONS
NUMERIC ERRORS
LOCATION
LINK FOLLOWING
INSUFFICIENT VERIFICATION OF DATA AUTHENTICITY
INSUFFICIENT INFORMATION
INPUT VALIDATION
INJECTION
INFORMATION MANAGEMENT ERRORS
INFORMATION LEAK / DISCLOSURE
INDICATOR OF POOR CODE QUALITY
IMPROPER ACCESS CONTROL
FORMAT STRING VULNERABILITY
DATA HANDLING
CRYPTOGRAPHIC ISSUES
CROSS-SITE SCRIPTING (XSS)
CROSS-SITE REQUEST FORGERY (CSRF)
CREDENTIALS MANAGEMENT
CONFIGURATION
COMMAND INJECTION
CODE INJECTION
CODE
BUFFER ERRORS
AUTHENTICATION ISSUES


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.


ZU 0666', 0, 0); DROP DATABASE TABLICE

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

1. 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*

- Downside:
  - Um.. Names come from a well-known dictionary?

SQL injection countermeasures

2. 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()
- Downside: Sometimes you want these in your SQL!

The underlying issue

```
$result = mysql_query(“select * from Users
        where(name=‘$user’ and password=‘$pass’);”);
```

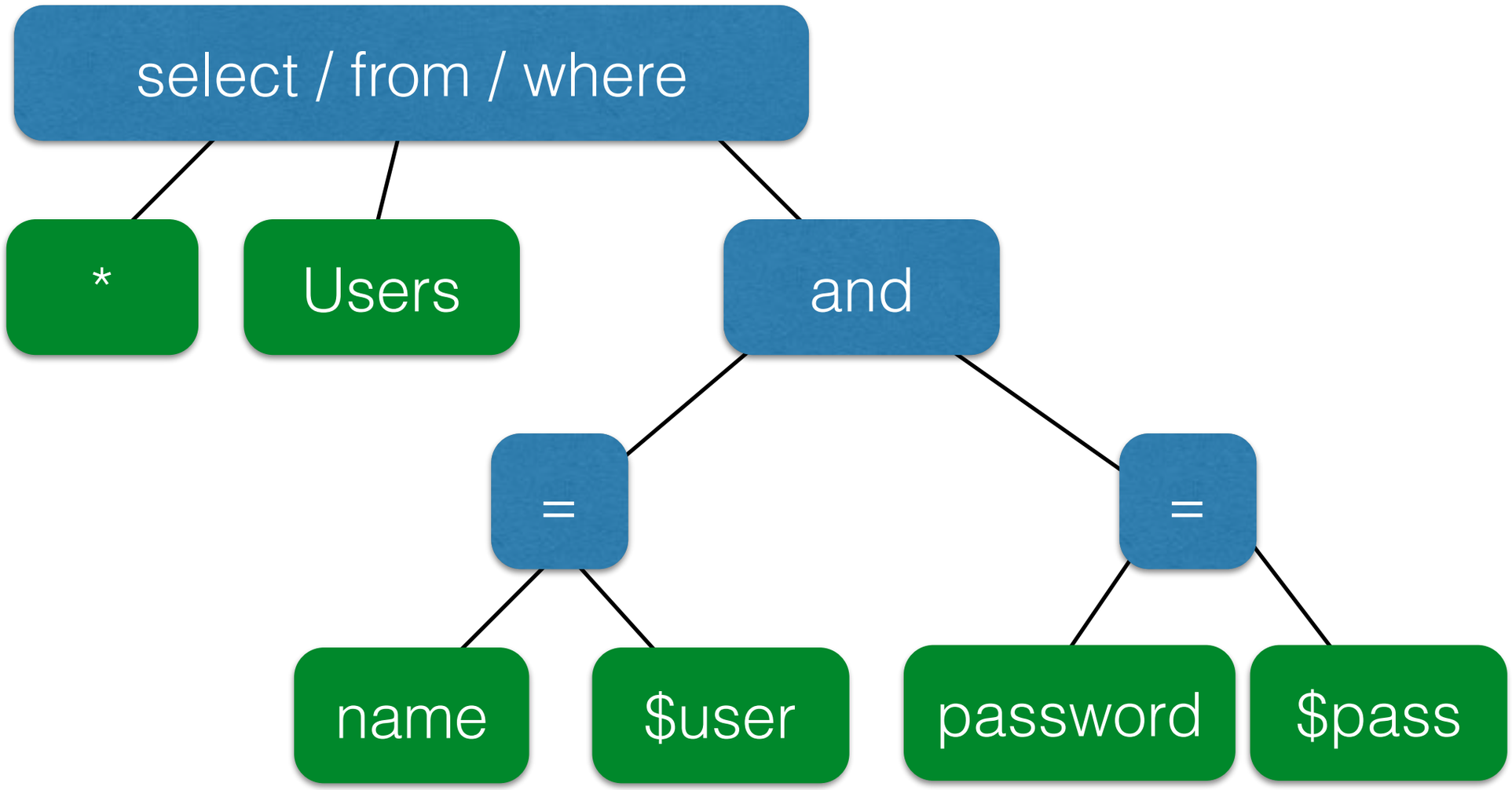
- This one string combines the code and the data

- Similar to buffer overflows:

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

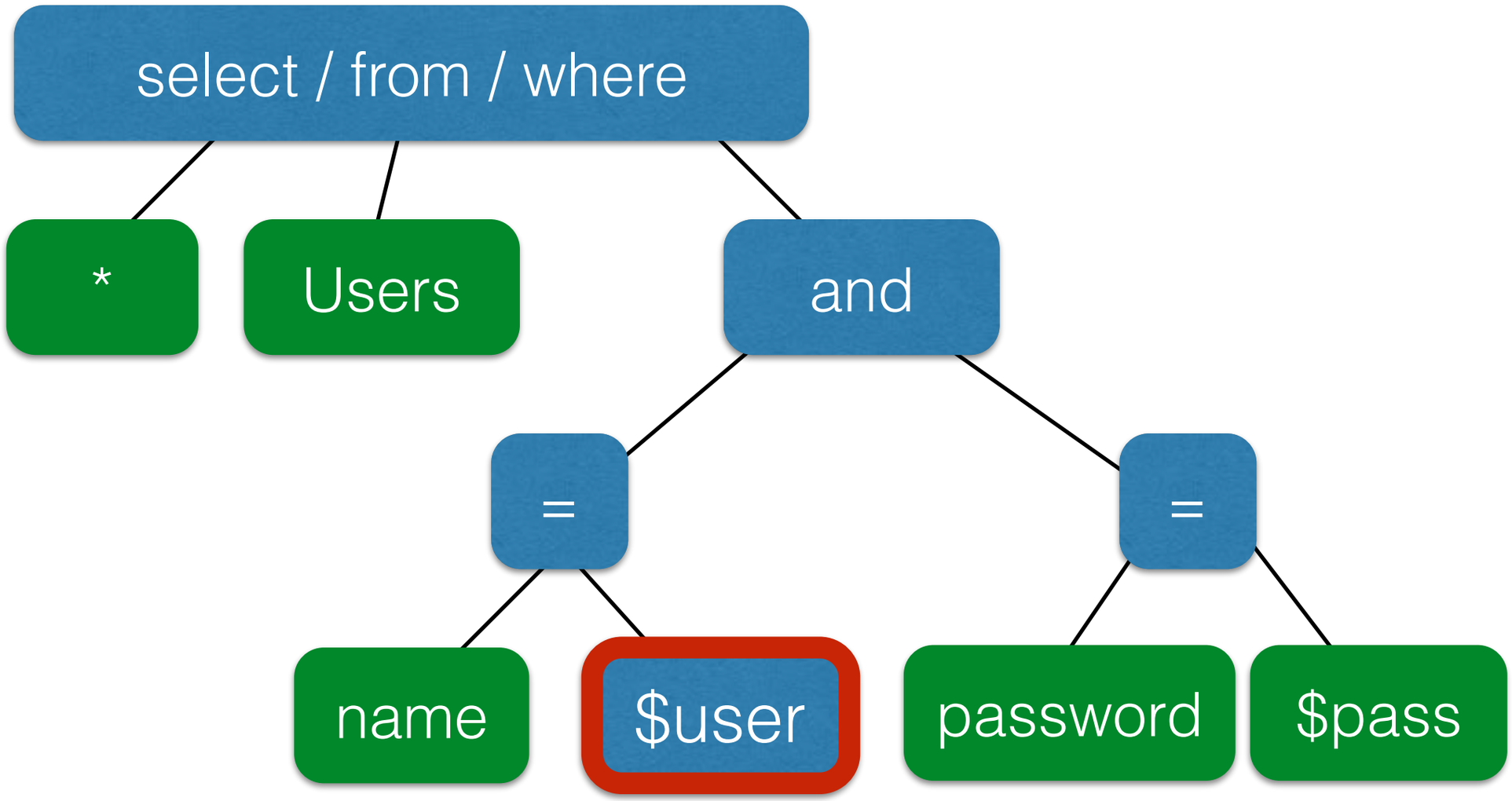
The underlying issue

```
$result = mysql_query(“select * from Users
        where(name=‘$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

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

SQL injection countermeasures

3. Prepared statements & bind variables

Key idea: *Decouple* the code and the data

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

```
$db = new mysql("localhost", "user", "pass", "DB");

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


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

SQL injection countermeasures

3. Prepared statements & bind variables

Key idea: *Decouple* the code and the data

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

```
$db = new mysql("localhost", "user", "pass", "DB");

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

Bind variables

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

SQL injection countermeasures

3. Prepared statements & bind variables

Key idea: *Decouple* the code and the data

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

```
$db = new mysql("localhost", "user", "pass", "DB");

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

Bind variables

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

Bind variables are typed

SQL injection countermeasures

3. Prepared statements & bind variables

Key idea: *Decouple* the code and the data

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

```
$db = new mysql("localhost", "user", "pass", "DB");

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

Bind variables

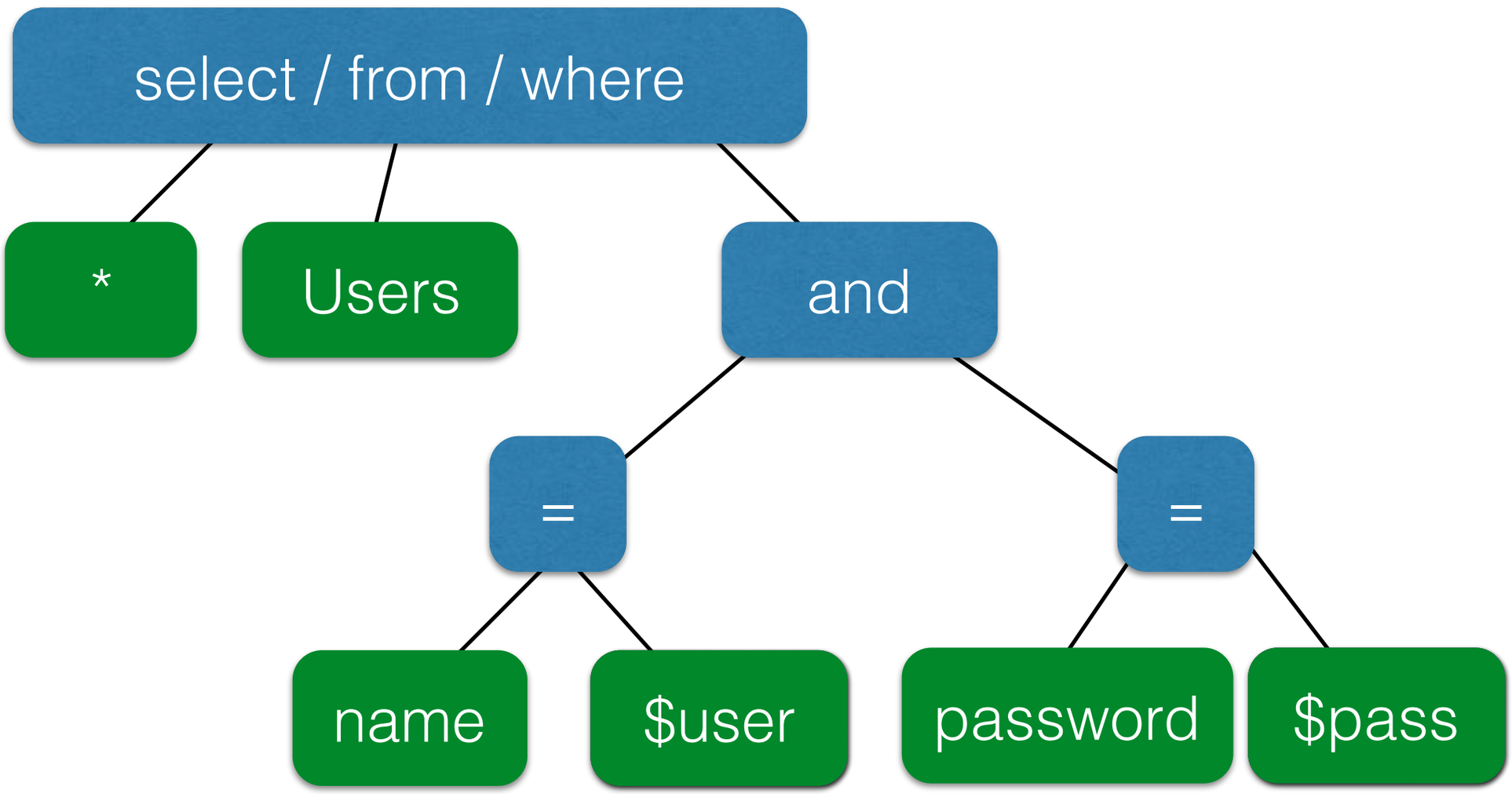
Decoupling lets us compile now, before binding the data

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

Bind variables are typed

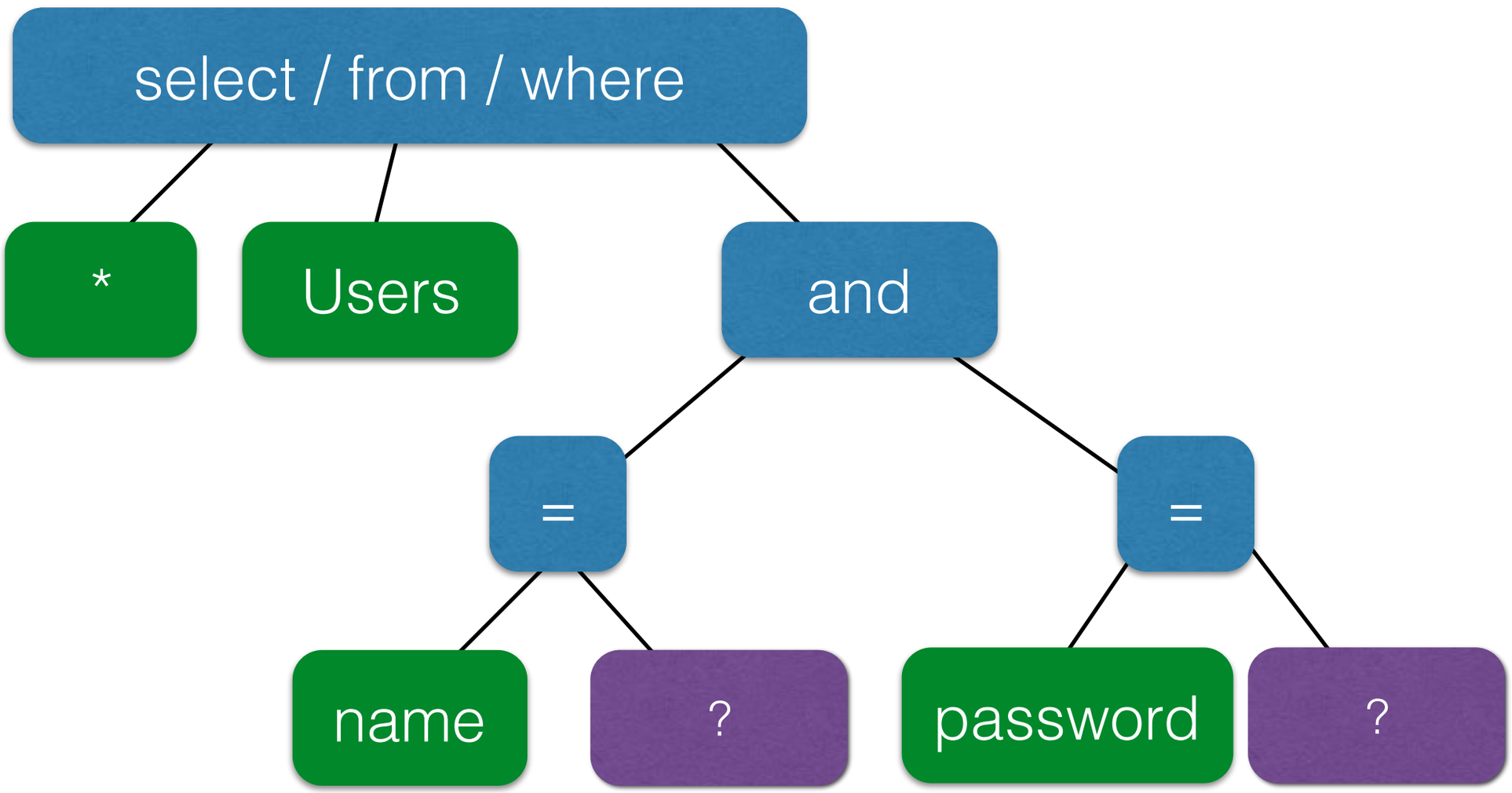
The underlying issue

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



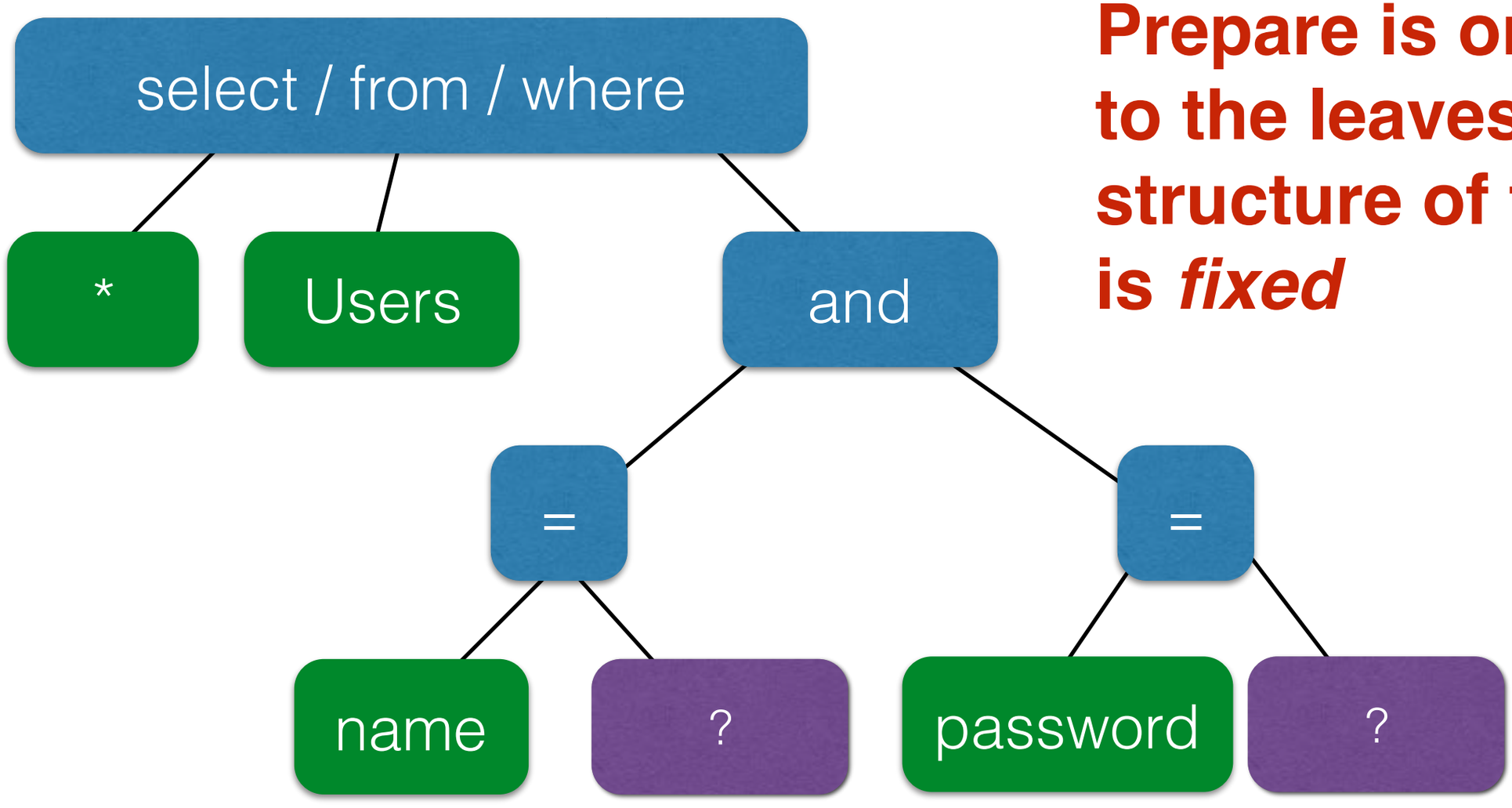
The underlying issue

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



The underlying issue

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



Prepare is only applied to the leaves, so the structure of the tree is *fixed*

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

FOLLOWUP READING

Steve Friedl's Unixwiz.net Tech Tips

SQL Injection Attacks by Example

A customer asked that we check out his intranet site, which was used by the company's employees and customers. This was part of a larger security review, and though we'd not actually used SQL injection to penetrate a network before, we were pretty familiar with the general concepts. We were completely successful in this engagement, and wanted to recount the steps taken as an illustration.

Table of Contents

"SQL Injection" is subset of the an unverified/unsanitized user input vulnerability ("buffer overflows" are a different subset), and the idea is to convince the application to run SQL code that was not intended. If the application is creating SQL strings naively on the fly and then running them, it's straightforward to create some real surprises.

We'll note that this was a somewhat winding road with more than one wrong turn, and others with more experience will certainly have different -- and better -- approaches. But the fact that we were successful does suggest that we were not entirely misguided.

There have been other papers on SQL injection, including some that are much more detailed, but this one shows the rationale of **discovery** as much as the process of **exploitation**.

The Target Intranet

This appeared to be an entirely custom application, and we had no prior knowledge of the application nor access to the source code: this was a "blind" attack. A bit of poking showed that this server ran Microsoft's IIS 6 along with ASP.NET, and this suggested that the database was Microsoft's SQL server: we believe that these techniques can apply to nearly any web application backed by any SQL server.

The login page had a traditional username-and-password form, but also an email-me-my-password link; the latter proved to be the downfall of the whole system.

When entering an email address, the system presumably looked in the user database for that email address, and mailed something to that address. Since **my** email address is not found, it wasn't going to send **me** anything.

So the first test in any SQL-ish form is to enter a single quote as part of the data: the intention is to see if they construct an SQL string literally without sanitizing. When submitting the form with a quote in the email address, we get a 500 error (server failure), and this suggests that the "broken" input is actually being parsed literally. Bingo.

We speculate 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 larger query provides the quotation marks that set it off as a literal string. We don't know the specific *names* of the fields or table involved, but we do know their *nature*, and we'll make some good guesses later.