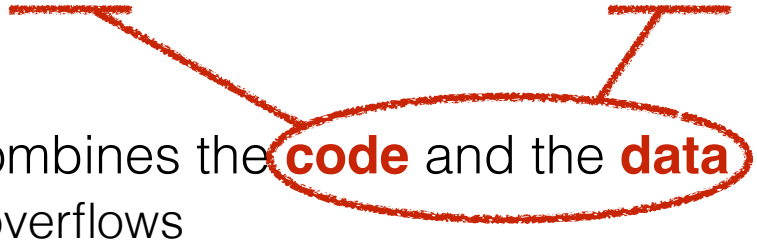
SQL injection countermeasures

The underlying issue

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

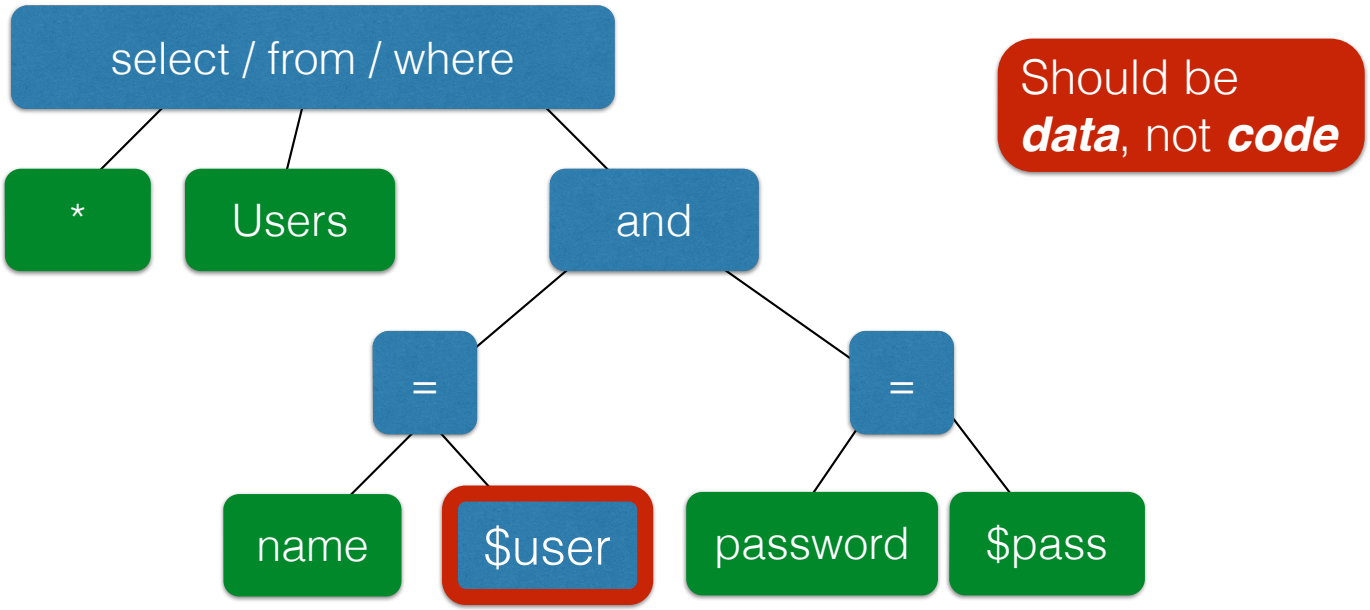


- This one string combines the **code** and the **data**
  - Similar to buffer overflows

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');");
```



Prevention: Input Validation

- Since we require input of a certain form, but we cannot guarantee it has that form, we must **validate it before we trust it**
  - Just like we do to avoid buffer overflows
- **Making input trustworthy**
  - **Check it** has the expected form, and reject it if not
  - **Sanitize it** by modifying it or using it it in such a way that the result is correctly formed by construction

Sanitization: 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?

Sanitization: Escaping

- **Replace problematic characters with safe ones**
  - change `'` to `\'`
  - change `;` to `\;`
  - change `-` to `\-`
  - change `\` to `\\`
- 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!
  - And escaping still may not be enough

Checking: Whitelisting

- **Check that the user input is known to be safe**
  - E.g., integer within the right range
- **Rationale**: Given an invalid input, **safer to reject than to fix**
  - "Fixes" may result in wrong output, or vulnerabilities
  - ***Principle*** *of fail-safe defaults*
- **Downside**: Hard for rich input!
  - Um.. Names come from a well-known dictionary?

Sanitization: Prepared Statements

- **Treat user data according to its *type***
  - 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

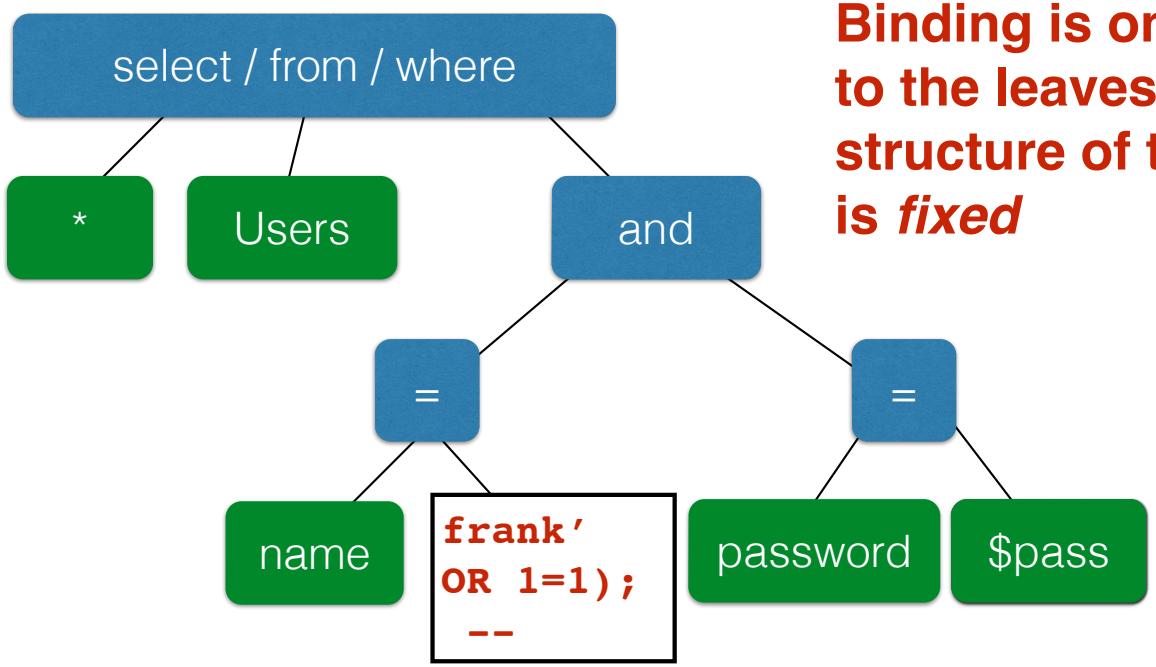
Decoupling lets us compile now, before binding the data

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

Bind variables are typed

Using prepared statements

```
$statement = $db->prepare("select * from Users
    where(name=?        and password=?);");
$stmt->bind_param("ss", $user, $pass);
```



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

Also: Mitigation

- For **defense in depth**, you might *also* attempt to mitigate the effects of an attack
  - But should **always do input validation** in any case!
- **Limit privileges**; reduces power of exploitation
  - Can limit commands and/or tables a user can access
    - Allow SELECT queries on Orders_Table but not on Creditcards_Table
- **Encrypt sensitive data** stored in the database; less useful if stolen
  - May not need to encrypt Orders_Table
  - But certainly encrypt Creditcards_Table.cc_numbers