SQL Injection Attacks

CMSC 414

September 18 & 20, 2017

What is SQL, and Why Do We Care?

Structured Query Language (pronounced "sequel")

- created at IBM in early '70s
- adopted by Relational Software, Inc. (now Oracle) in late '70s

De-facto standard language for structured databases

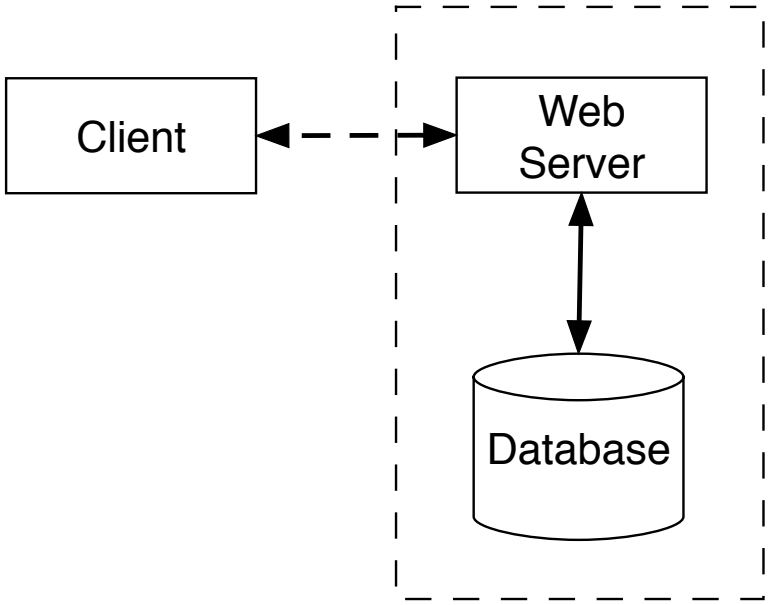
- Oracle dominated relational DB market for a long time
- different databases have slightly different *dialects*

Used for populating/modifying/querying databases

This is not a databases course, so ...

Backend Servers

Web servers often backed by a database



Can provide dynamic content

Data storage and manipulation

Queries come from web server, but sometimes passing through user-provided input

Database Overview

Structured data (also unstructured databases, but we're not worrying about them right now)

Table

Users ← Table Name

Name	**DOB**	**Email**	**Password**
Tom Servo	3/12/1988	tservo@gizmonic	OTYxMDgxMjM5
Crow T. Robot	4/1/1988	ctrobot@gizmonic	NDFlM2I2NzAw
Joel Robinson	2/20/1960	jrob@gizmonic	MzUyM2UxMDBi

Column

- names a variable/property
- typed (if you're lucky, not `VARCHAR(255)`)

Row

- individual record
- generally contains a *primary key*

Cell

- variable/property in an individual record

Database Management

Database Administrator (DBA)

- configures/populates database
- DB equivalent of root

Programmer

- implements queries
- interfaces with DB through its API

Database Management System (DBMS)

- semantics for DB design
- transactions for consistent data manipulation
- **query language** and APIs $\Leftarrow$ SQL
- user/permissions management

Group Exercise 1

Clone `sql-practice` using `get-assignment`.

This will give you some hands-on experience with using a SQL database. Work through steps 1–7.

Transactions

Unit of work

"Give me all of the records in the Users table where the name is registered as enrolled in CMSC414 in the Classes table"	2 reads
"Deduct 2 Hamdingers from Crow T. Robot; Add 2 Hamdingers to Tom Servo"	2 writes

ACID semantics:

- **Atomicity** — all-or-nothing completion
- **Consistency** — DB state is always *valid*, even if not *correct*
- **Isolation** — transaction results not visible until completed
- **Durability** — once committed, tx remains even after faults

A Little SQL

Users

Name	DOB	Email	Password
Tom Servo	3/12/1988	tservo@gizmonic	OTYxMDgxMjM5
Crow T. Robot	4/1/1988	golden@gizmonic	NDFlM2I2NzAw
Joel Robinson	2/20/1960	jrob@gizmonic	MzUyM2UxMDBi
Mike Nelson	10/11/1964	mike@gizmonic	YThhZWJjNTBj

```
SELECT Email FROM Users WHERE Name="Joel Robinson";
⇒ jrob@gizmonic

UPDATE Users SET Email="golden@gizmonic" WHERE
DOB="4/1/1988"; -- this is a comment

INSERT INTO Users Values('Mike Nelson', '10/11/1964',
'mike@gizmonic', ...);

DROP TABLE Users;
```

Group Exercise 2

Now proceed to step 8 of the README. See how many of the queries you can compose.

Interacting with Web Servers

Client specifies a **URL** (Uniform Resource Locator)

`http://www.cs.umd.edu/~mmarsh/index.html` ⇐ static content

`https://www.google.com/search?q=sql` ⇐ dynamic content

General form:
`<protocol>://<host>/<path to resource>[?<arguments>]`

Common protocols: `http`, `https`, `ftp`, `mailto`, `tor`

HTTP Requests

Most common requests: **GET** and **POST**

- GET: all data in URL, no request body, should be free of side effects
- POST: request body includes additional data, often has side effects

In addition to URL, **headers** provide extra information

- type of client
- cookies (if relevant)
- language/file encodings
- etc.

HTTP GET Request Example

http://cs.umd.edu/class/fall2017/cmsc414/

```
GET /class/fall2017/cmsc414/ HTTP/1.1
Host: cs.umd.edu
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0) \
   Gecko/20100101 Firefox/23.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\
   */*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

User-Agent can be an arbitrary string, but generally identifies the browser, command-line client (wget, curl, ...), software library (JDK, ...), etc.

HTTP POST Request Example

Headers:

```
POST /class HTTP/1.1
Host: piazza.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:23.0)\
   Gecko/20100101 Firefox/23.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\
   */*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://piazza.com/
Cookie: piazza_session=...
Connection: keep-alive
```

POST body:

```
email=mmarsh@cs.umd.edu
from=/signup
password=****
```

HTTP Responses

- Status code
- Headers
- Data
- Cookies (browser stores this on server's behalf)

```
HTTP/1.1 200 OK
Date: Sat, 16 Sep 2017 15:05:33 GMT
Server: Apache/2.4.6 (Red Hat\
   Enterprise Linux)
Last-Modified: Thu, 14 Sep 2017\
   13:58:48 GMT
Etag: "341f-55926aceb917a-gzip"
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Encoding: gzip
X-CSD-Backend: e
Content-Length: 3503
Connection: close
Content-Type: text/html
```

```
<html>
<head>
  <title>CMSC 414-0101/0301</title>
  <style>
  * { font-family: sans-serif; }
  dt { font-weight: bold; }
  table, th, td {
     border: 1px solid #AAAAAA;
     border-collapse: collapse;
     padding: 2pt; }
  h2 { font-size: large; }
  </style>
  ...
```

HTTP is Stateless

Session lifecycle:

1. Client connects to server
2. Client issues request
3. Server responds
4. Client issues additional requests
   (if `Connection: keep-alive` header present)
5. Client disconnects

HTTP does not have notion of a returning client

Would have to log in every time ⇒ *Cookies*

Group Exercise 3

The browser in your VM has an add-on called *Firebug* installed. Use this to examine requests and responses for various URLs. Get a sense of what headers you see, what they mean, and how queries change the headers as well as the bodies of messages (both requests and responses).

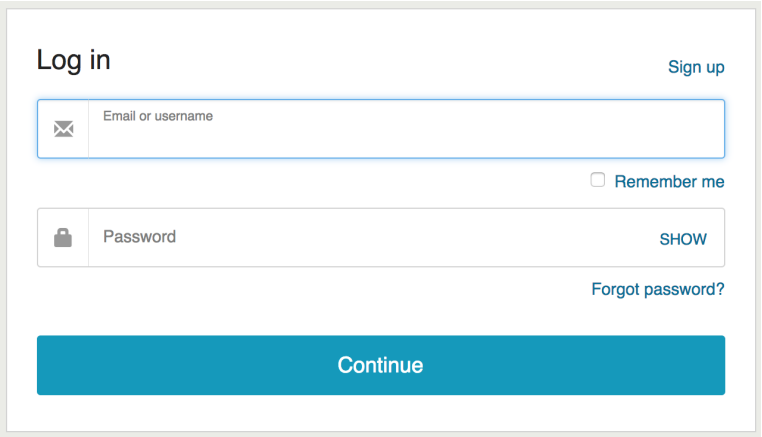
Look for this bug in the upper-right corner:

Try to find some interesting GET and POST requests. Be careful what you share with your group! Some servers (like Piazza) include your password as cleartext[1] in the login POST data.[2]

[1] We'll define this in a couple of weeks.

[2] Don't worry—your connection is encrypted.

Serving Requests



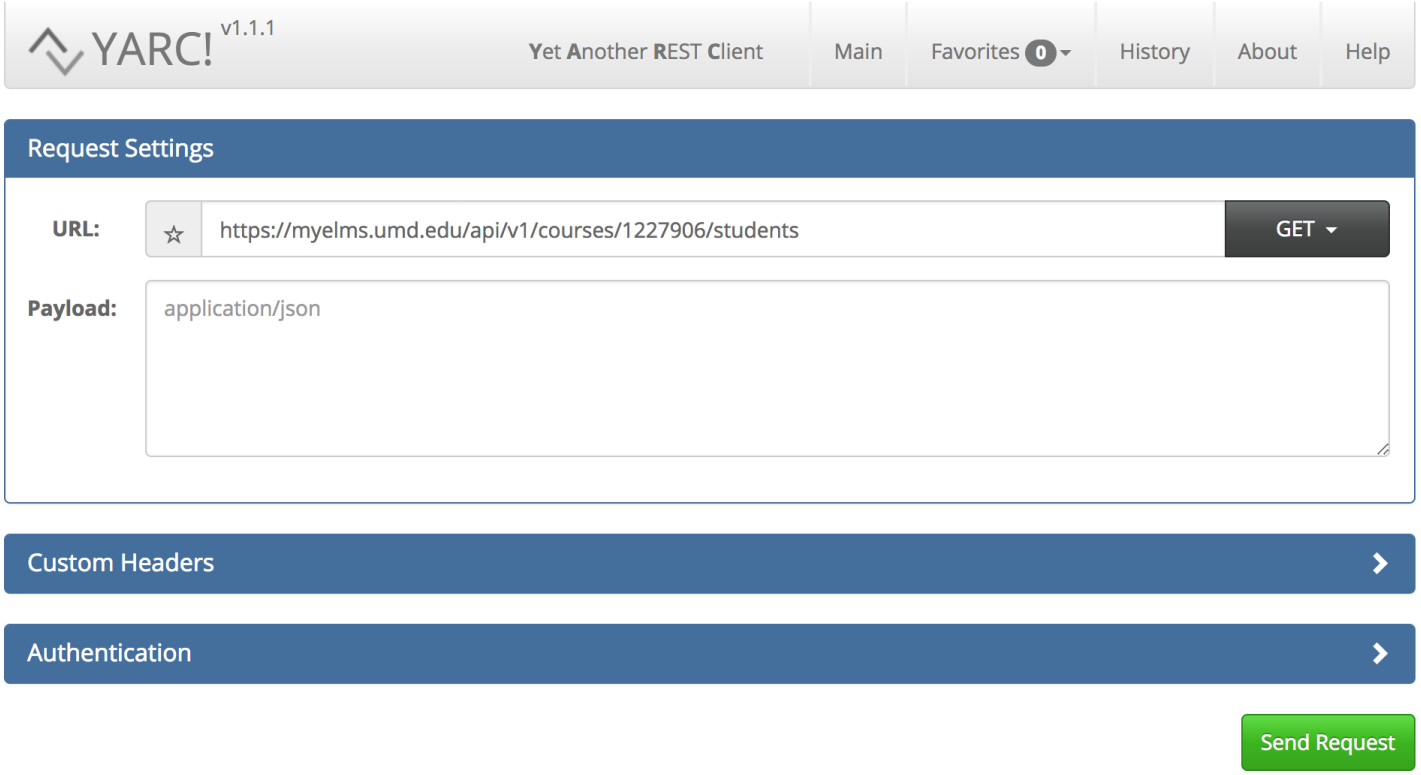

```
$result = mysql_query("select * from Users
where(name='$user' and password='$pass');");
```

$result not empty ⇒ successful login

How might we exploit this?

Assumptions

1. Users will enter data into HTML form
2. Javascript may validate that all fields are filled in "appropriately"
3. Users will enter "normal" data



Look for REST clients in Firefox add-ons (YARC is for Chrome)

SQL Injection

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

Enter `frank' OR 1=1); --` for the username

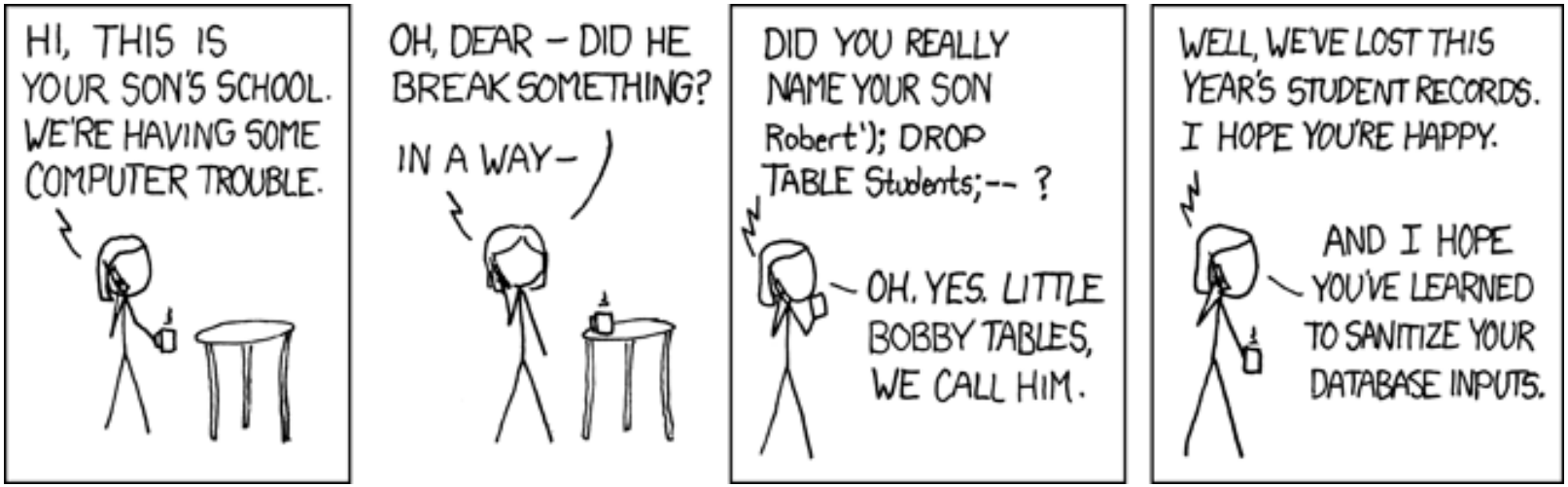
⇒ `$result = mysql_query("select * from Users where(name='frank' OR 1=1); -- ' and password='');");`

$result is then

```
[['Tom Servo','3/12/1988','tservo@gizmonic','OTYxMDgxMjM5'],
 ['Crow T. Robot','4/1/1988','golden@gizmonic','NDFlM2I2NzAw'],
 ['Joel Robinson','2/20/1960','jrob@gizmonic','MzUyM2UxMDBi'],
 ['Mike Nelson','10/11/1964','mike@gizmonic','YThhZWJjNTBj']]
```

We have now authenticated to the server, and may even get to see the full Users table!

SQL Injection



https://xkcd.com/327/ "Exploits of a Mom"

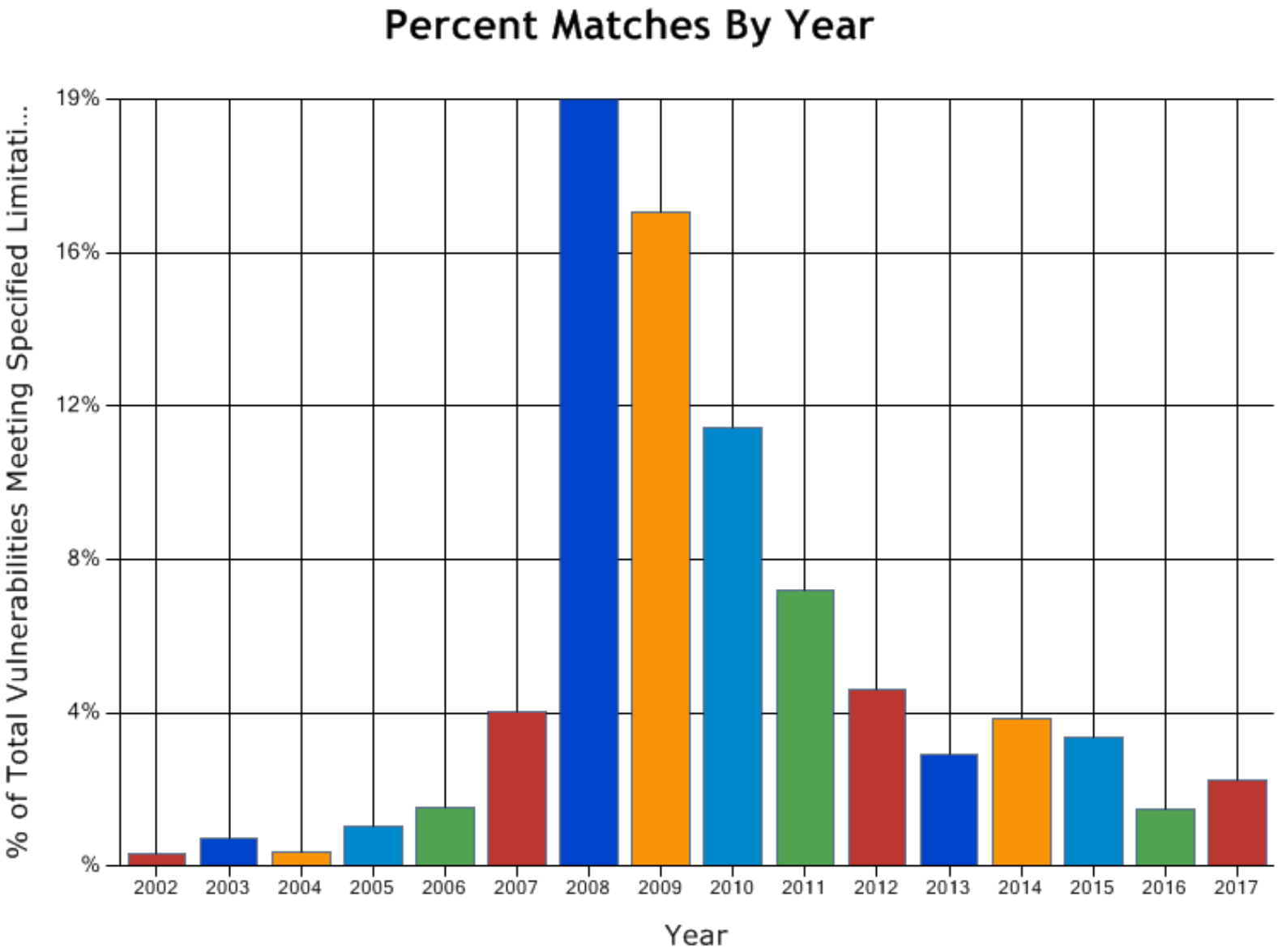
```
$result = mysql_query("select * from Users
where(name='Robert'); DROP TABLE Students; --' and
password='');");
```

SQL Injection Prevalence

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)

From MITRE's 2011 CWE/SANS Top 25 Most Dangerous Software Errors

SQL Injection Prevalence



From `https://nvd.nist.gov/vuln/search`

Group Exercise 4

`https://nvd.nist.gov/vuln/search` allows you to search for reported vulnerabilities. In the advanced search, you can search not just by keywords, but by time and category of vulnerability, of which "SQL Injection" is one option.

Take a look through some of the SQL Injection vulnerabilities. What problems seem to come up repeatedly? How many of these have a low attack complexity (that is, they're very easy to do) and don't require user privileges/authentication?

A term you will see frequently is *sanitization*. We'll talk about this next.

Countermeasures

How do we prevent SQL injection attacks?

Several approaches:

- Blacklisting
- Whitelisting
- Escape Characters
- Prepared Statements & Bind Variables

Blacklisting

There are a number of characters that cause problems

- '
- --
- ;

We can delete them — problem solved!

Well...

- Peter O'Toole
- *Wallace & Gromit: The Curse of the Were-Rabbit*
- "Avoid popularity; it has many snares, and no real benefit." (William Penn)

Determining when these characters are actually bad is *difficult*

Whitelisting

Verify that user-provided input is in a valid (safe) set

- Integers only contain digits
- Ranges might have a dash

Reject bad inputs, to fail safe

⇒ Names can be almost arbitrary, so can't validate against a known-safe set

⇒ Website designers act against their own best interest:

Escape Characters

Replace potentially harmful characters with escaped versions

- ' ⇒ \'
- ; ⇒ \;
- - ⇒ \-
- \ ⇒ \\

Can be automated by libraries

- `magic_quotes_gpc = On`
- `mysql_real_escape_string()`

But you might *want* these characters sometimes

Group Exercise 5

You've decided you want to write your own character-escaping library! In any language you like, write a simple program that takes a string as input and outputs an escaped string.

Try to make this as robust as possible to various inputs. Discuss your approach with your group. What issues must you consider? How do you avoid introducing errors or new vulnerabilities?

Interlude

There's one more countermeasure to discuss, but first...

Why Does SQL Injection Happen?

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

This has *code* and *data* in one string

We might suppose this to be a bad idea...

Buffer overflows *also* have code mingled with data

Code and data probably shouldn't be combined like this

Prepared Statements & Bind Variables

We can decouple code and data

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

becomes

```
$db = new mysqli("localhost","dbuser","dbpass","DB");
$statement = $db->prepare("select * from Users
   where(name=? and password=?);");
```

The ?'s denote **bind variables**

The **prepared statement** leaves placeholders for the bind variables

Binding Variables

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

`bind_param` takes a type string as the first argument

- one-character type specification
- expects number of following parameters to match length of type string

Character	Type
i	integer
d	double
s	string
b	blob

Binding Variables

By the time we call `bind_param`, the SQL statement has been parsed

We can't inject SQL through the variables

Also makes queries more efficient

? $\Leftrightarrow$ **variable** in order specified in `prepare` and `bind_param` statements

Must match in number

Mitigation

As always, we can't rely on programmers to do things right

$\Rightarrow$ *Defense in Depth*

Principle of Least Privilege

- Require user *Authentication*
- Limit commands/tables a user can access

Confidentiality

- Encrypt sensitive data, like credit card numbers
- DBMS might support table-by-table encryption, or column-by-column

Group Exercise 6

Consider a variety of businesses (stores, banks, etc.) that would use a structured database as part of their backend server. What sorts of roles would you expect, what kind of data, and how might you provide defense in depth against bad SQL? Discuss among your group.