Database Management Systems (DBMS)

preliminaries
- mainly relational databases (so pretty much just SQL)
- DBMS allow us to manage databases
- databases store data

relational databases
- store data in tables
- tables have columns (fields)
- rows are individual data items
- keys tag the data
- keys in one table can be paired or matched with keys in other tables (a relation!)
- usually the data is normalized (as many tables as are needed without too much detail)
  - with appropriate relations
- we try to ensure that major data items are never deleted
  - e.g., the last home in a zip code is removed (but we don't remove the zip code itself)

MySQL
- a popular open source DBMS

install MySQL:
```
sudo apt-get install mysql-server
```

try it out:
```
mysql -uroot -p
```

let's create a database, then use it:
```
CREATE DATABASE test;
USE test;
```

now, let's create a team table:
```
CREATE TABLE `teams` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(50) NOT NULL DEFAULT '',
  `score` int(10) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

what does it look like?
```
DESCRIBE teams;
```

and what does it contain?
```
SELECT * FROM teams;
```

so it's not a good idea to use the root user to do all of this
so let's undo by removing the database:
```
DROP DATABASE test;
```

let's create another user (with fewer privileges than root):

```
CREATE USER 'cyber'@'localhost' IDENTIFIED BY 'cyber';
```

so the user is cyber with password cyber (original!)
let's grant all privileges to this user on a new database that we will create
first, we need to create the database (we'll call it cyber):

```
CREATE DATABASE cyber;
```

next, we can grant the user cyber all privileges on this new database:

```
GRANT ALL PRIVILEGES ON cyber.* TO 'cyber'@'localhost';
```

note that localhost implies that cyber can only connect to the DBMS locally
the privileges ensure that cyber can only mess around with the database cyber
cyber.* implies all tables in the database cyber

logout of MySQL:
[Ctrl+D]

and let's login again as cyber:

```
mysql -ucyber -p cyber
```

note that the format is:

```
mysql -u<user> -p<password> <database>
```

therefore:

```
mysql -ucyber -p cyber
```

note that the password is left out (there's a space in between **-p** and **cyber**)
this logs in to MySQL as the user **cyber**
the password to be entered at the terminal (hidden)
the selected database to be **cyber**

let's see what databases we can see:

```
SHOW DATABASES;
```

since we're already using the database cyber, let's see what tables exist:

```
SHOW TABLES;
```

let's recreate the team table:

```
CREATE TABLE `teams` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(50) NOT NULL DEFAULT '',
  `score` int(10) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

let's create another table:

```
CREATE TABLE `acl` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `team_id` smallint(5) unsigned NOT NULL DEFAULT '0',
```

```
  `password` varchar(50) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `acl_team_id` (`team_id`),
  CONSTRAINT `acl_team_id` FOREIGN KEY (`team_id`) REFERENCES `teams`
(`id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

the CONSTRAINT clause means that the key **team_id** in the **acl** table
- will refer to the key **id** in the **teams** table
- this is a relation!
- if an **id** is deleted from the **teams** table
  - entries in the **acl** table with the same **team_id** will also be deleted
  - this is the CASCADE clause

now, let's add some data to the teams table:
```
INSERT INTO `teams` VALUES (1,'ATHENIANS',0);
INSERT INTO `teams` VALUES (2,'EGYPTIANS',0);
```

and what does it contain now?
```
SELECT * FROM teams;
```

let's also add some data to the acl table:
```
INSERT INTO `acl` VALUES (1,1,password('password'));
INSERT INTO `acl` VALUES (2,2,password('123456'));
```

and what does it contain now?
```
SELECT * FROM acl;
```

let's find the password (hash) of ATHENIANS:
```
SELECT `password` FROM acl WHERE `team_id`=(SELECT `id` FROM teams
WHERE `name`='ATHENIANS');
```

it's a query with multiple parts; the first part returns the ID 1:
```
(SELECT `id` FROM teams WHERE `name`='ATHENIANS');
```

the second part now uses this first value:
```
SELECT `password` FROM acl WHERE `team_id`=1;
```

we could get the same result with this query:
```
SELECT `password` FROM acl,teams WHERE acl.`team_id`=teams.`id` AND
teams.`name`='ATHENIANS';
```

we can even do a bit of shorthand:
```
SELECT `password` FROM acl a,teams t WHERE a.`team_id`=t.`id` AND
t.`name`='ATHENIANS';
```

here are some more SQL queries:
note that % in SQL behaves very much like * in bash
```
SELECT `score` FROM teams WHERE `name` LIKE 'ATH%';
```

it's also case insensitive:

```
SELECT `score` FROM teams WHERE `name` LIKE 'ath%';
```

ATHENIANS deserve 100 points for having one team member writing the code while most of the rest slack off (:P):

```
UPDATE teams SET `score`=`score`+100 WHERE `name` LIKE '%the%';
```

what does the teams table look like now?

```
SELECT * FROM teams;
```

let's interact with MySQL from a web page (using PHP)
we assume that Apache2 is already installed

first, we need to install PHP (including the Apache2 PHP module):

```
sudo apt-get install php libapache2-mod-php php-mysql
```

a simple web page has been created to interact with MySQL (see the code on the class web site)
specifically, the **cyber** database
the web page asks for a team name and returns that team's score

to connect to the database, we'll need two configuration files
one for useful database functions: **db.php**
one to access the **cyber** database: **config.php**
to allow users to interact, we'll need a main HTML file: **index.php**
place the three files in **/var/www/html**
and remove **index.html** (if it already exists)

try it out...
you can probably infer the query that it makes:

```
SELECT `name`,`score` FROM teams WHERE `name`='<whatever the user entered>'
```

so if we enter ATHENIANS:

```
SELECT `name`,`score` FROM teams WHERE `name`='ATHENIANS'
```

but what if we want to see the entire table and have no clue what the team names are?
how about if we enter name' or 'a'='a instead?

```
SELECT `name`,`score` FROM teams WHERE `name`='name' or 'a'='a'
```

`name`='name' will be false since no team in the table is called **name**
but 'a'='a' is always true!
yes, the character 'a' is always equal to the character 'a' (i.e., itself)!
so the entire WHERE clause will be true, causing the SELECT clause to be performed
the result is that `name` and `score` for all teams in the database will be returned!
yes! that's information that was probably not intended to be returned

this version of PHP prevents multiple SQL queries from being performed in a single submission
but old version did!

what would blah'; delete from teams; -- do?

```
SELECT `name`,`score` FROM teams WHERE `name`='blah'; delete from
teams; -- '
```

-- means that whatever follows is a comment
the SELECT query returns nothing since there is no team named **blah**
but the second SQL query is performed:

```
DELETE FROM teams;
```

which, of course (and unfortunately), deletes everything from the teams table!
oh noes! gah!

other stuff could be entered, like blah'; select * from acl; -- :

```
SELECT `name`,`score` FROM teams WHERE `name`='blah'; select * from
acl; -- '
```

this supposes that the attacker knows something about the database
or is just guessing at other table names (e.g., acl)
the first SQL query is not performed because the WHERE clause is false
but the second is performed:

```
SELECT * FROM acl;
```

the result is that the acl table is returned!
and this is BAD!

what about this: blah'; show tables; -- ?

```
SELECT `name`,`score` FROM teams WHERE `name`='blah'; show tables; --
'
```

yup, a listing of the tables in the **cyber** database will be returned!

or this: blah'; show databases; -- ?

```
SELECT `name`,`score` FROM teams WHERE `name`='blah'; show databases;
-- '
```

yup, a listing of the databases accessible by the user **cyber** will be returned!

...and we can keep going (like accessing other databases!)...
exploiting SQL vulnerabilities usually means just playing around and trying things
this is what a some folks do for a living!
SQL vulnerabilities are almost always the result of badly designed backend scripts (usually PHP)
it is always good practice to sanitize user input and check for invalid input
don't allow, for example, input that could alter SQL queries

you can export the contents of databases (do this outside of MySQL, at the terminal):

```
mysqldump -ucyber -p --skip-extended-insert --add-drop-database
--add-drop-table --databases cyber
```

you probably want to save this to a file:

```
mysqldump -ucyber -p --skip-extended-insert --add-drop-database
--add-drop-table --databases cyber > db.sql
```

let's drop the database to see how the file can be used to recreate it:

```
mysql -ucyber -p
DROP DATABASE cyber;
SHOW DATABASES;
```

[Ctrl+D]

and let's recreate it:

```
mysql -ucyber -p < db.sql
mysql -ucyber -p cyber
SHOW TABLES;
SELECT * FROM teams;
```

the script **user.sql** is provided on the web site to facilitate creating the **cyber** user (the first time)
execute it this way:

```
mysql -uroot -p < user.sql
```