CS 61:
Database Systems

Security

With great power comes great responsibility…

OR

William Lamb, 2nd Viscount Melbourne

Spider Man's uncle Ben

Source: Wikipedia

Agenda

1. MySQL permissions
2. Demo: SQL injection attacks
3. Password storage/salt and pepper
4. Password cracking

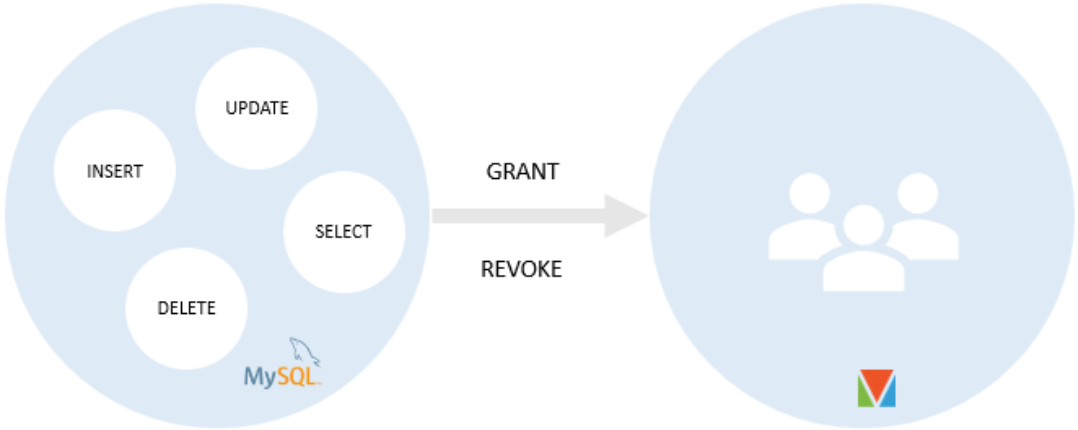
Show user permissions on sunapee

1. Connect to Sunapee
2. Click on Administration (upper left)
3. Click on Users and Privileges
4. Find cs61sp20
   - Show permissions grants
   - Show how to grant permission on a schema

Can assign rights to users individually or by role

Security authorization

Can assign rights to individual users



Can create roles, assign rights to roles, then assign users to roles

Benefits:

- Improved operational efficiency – new hires automatically get the rights they need
- Increased security – people do not get more rights that would typically need
- Increased visibility – easy to see what rights roles have

RBAC: Good idea in principle but has never worked for me!

- There is no generic person, each person has different responsibilities within dept
- People get temporary assignments with other departments, need different rights (creates a hybrid role)
- Assignment ends, but rights never changed (even if you set a calendar reminder and ask them if they still need the rights, they never say no!)

Adapted from: https://www.mysqltutorial.org/mysql-roles/

Agenda

1. MySQL permissions
2. Demo: SQL injection attacks
3. Password storage/salt and pepper
4. Password cracking

Do not trust user input

Consider the following Python code making a SQL call for restaurant details

What is wrong with this Python code?

Hint: CONCAT is ok, it combines attributes together

```
restaurant = “nobu” #user input from textbox, Nobu is a restaurant
cursor = cnx.cursor()
query = ("SELECT RestaurantName AS `Restaurant Name`,"
        +"CONCAT(TRIM(Building),' ',TRIM(Street)) AS Address, "
        + "Boro  "
        +"FROM Restaurants "
        +"WHERE RestaurantName LIKE '%" + restaurant +"%’) "
        +"LIMIT 20"
cursor.execute(query)
return cursor
```

Using Python as example rather than web API so I don’t leave vulnerable API running

Nothing is wrong with this query, provided we can trust the value in restaurant

Adding user input directly into command is a recipe for trouble!

What is wrong with this Python code?

Hint: CONCAT is ok, it combines attributes together

What if the user enters this instead?

```
restaurant = "nobu%' UNION SELECT 1,2,3 -- "
cursor = cnx.cursor()
query = ("SELECT RestaurantName AS `Restaurant Name`,"
         +"CONCAT(TRIM(Building),' ',TRIM(Street)) AS Address, "
         + "Boro  "
         +"FROM Restaurants "
         +"WHERE RestaurantName LIKE '%" + restaurant +"%')
         +"LIMIT 20"
cursor.execute(query)
return cursor
```

UNION adds rows from the following SELECT (number of attributes must match in each query)

LIMIT is commented out as a result of user input

Query is now:

… WHERE RestaurantName LIKE '%nobu%' UNION SELECT 1,2,3 -- LIMIT 20

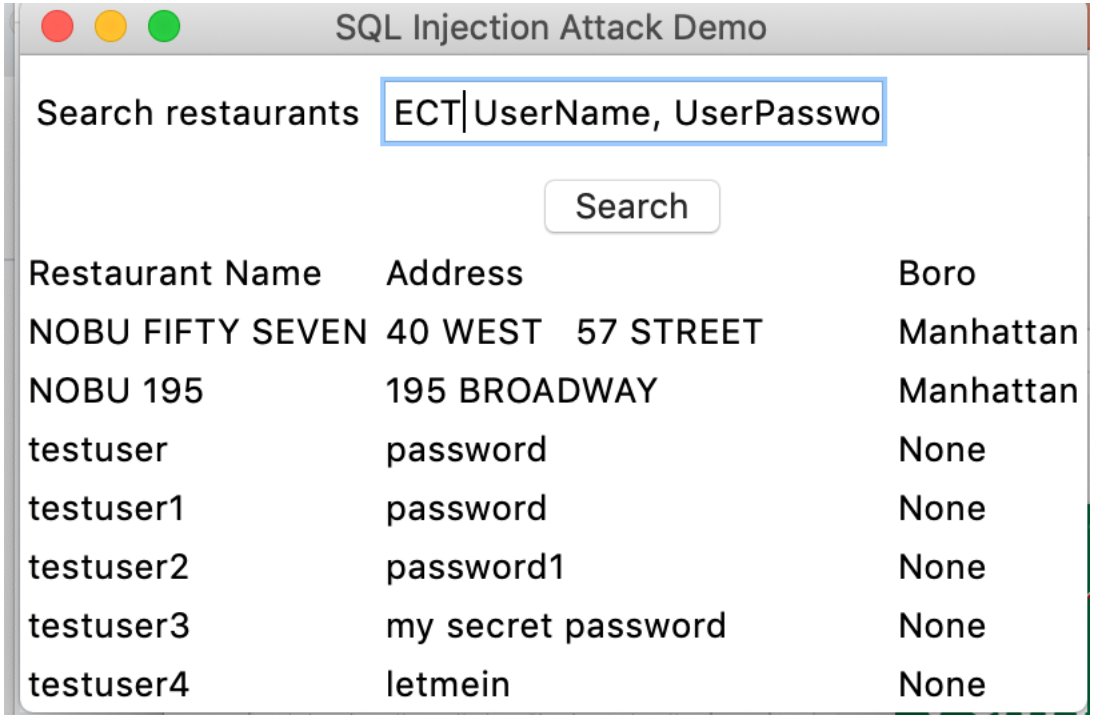
sql_injection.py demonstrates injection vulnerabilities

```
# test if user entry is vulnerable to injection, should see extra row with 1,2,3 if so
nobu%' UNION SELECT 1,2,3 --

#find out what schemas are on this database installation
nobu%' UNION SELECT schema_name, null, null from
information_schema.schemata --

#find tables in a schema
nobu%' UNION SELECT table_name, table_schema, null from
information_schema.tables where table_schema = 'nyc_inspections' --

#find all non-system tables on database
 nobu%' UNION (SELECT table_name, table_schema, null from
information_schema.tables where table_schema not like '%schema%' and
table_schema not like '%mysql%' and table_schema <> 'sys') --

#find attributes for restaurants table in nyc_inspections schema
 nobu%' UNION (SELECT `column_name`, data_type, character_maximum_length
from information_schema.`columns` where table_schema = 'nyc_inspections' and
table_name = 'Restaurants') --
```

Most sites have a Users table, let's steal all the username and passwords

#I've created a User's table in nyc_data
let's steal the username and passwords of all users!

nobu%' UNION SELECT UserName, UserPassword, null from nyc_data.Users --



You've been pwned!

Do not store passwords in plain text!

Use prepared statement to avoid user input as part of SQL command

Vulnerable	Prepared statement
restaurant = “nobu” cursor = cnx.cursor() query = ("SELECT RestaurantName, " +"Building, " + "Boro " +"FROM Restaurants " +"WHERE RestaurantName LIKE" **+"'%" + restaurant +"%’) "** +"LIMIT 20”) **cursor.execute(query)** return cursor	restaurant = “nobu” cursor = cnx.cursor() query = ("SELECT RestaurantName, " +"Building, ", +"Boro, " +"FROM Restaurants " +"WHERE RestaurantName LIKE” **+" %s ”** +"LIMIT 20") **cursor.execute(query, ('%'+restaurant+'%’,))** return cursor

User input is included in the SQL query string

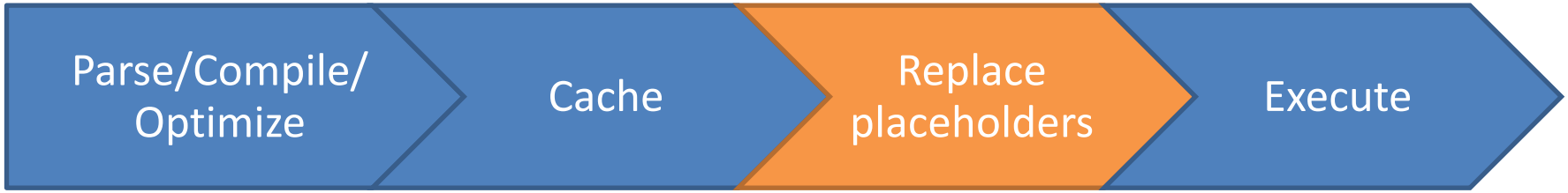
- **Can be abused!**

Prepared statement adds user input as a parameter after command is compiled

Prepared statements add data after compiling, optimizing, and caching

High-level overview of SQL execution process

UPDATE Users SET UserName = ? AND Password = ?



Parse
- Check syntax
- Check table and columns exist

Compile
- Convert query to machine code

Optimize
- Choose optimal execution plan

Cache
- Store optimized query plan in cache
- If command submitted again, skip prior steps (already done)

Replace placeholders
- Prepared statement are not complete statements
- Have placeholders for some values
- But, format of command is set now
- Placeholders filled with literal values
- Place holder data doesn't change command format

Execute
- Query is executed
- Data is returned
- Malicious data is stored in table, not executed

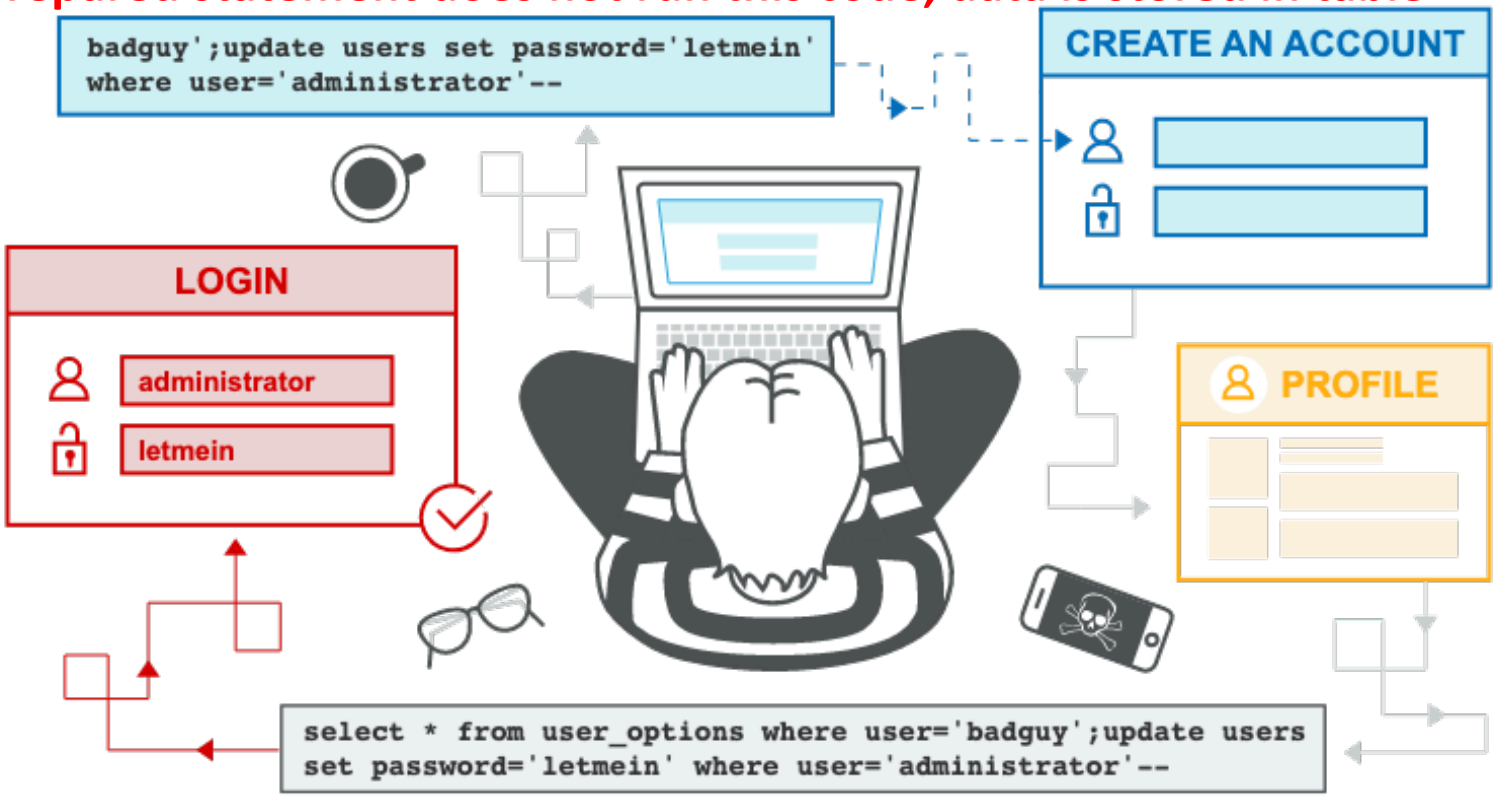
Adapted from: http://javabypatel.blogspot.com/2015/09/how-prepared-statement-in-java-prevents-sql-injection.html

Even if you use prepared statements, be wary of data in your database!

Second-order attack:

User enters data with SQL embedded

Prepared statement does not run this code, data is stored in table



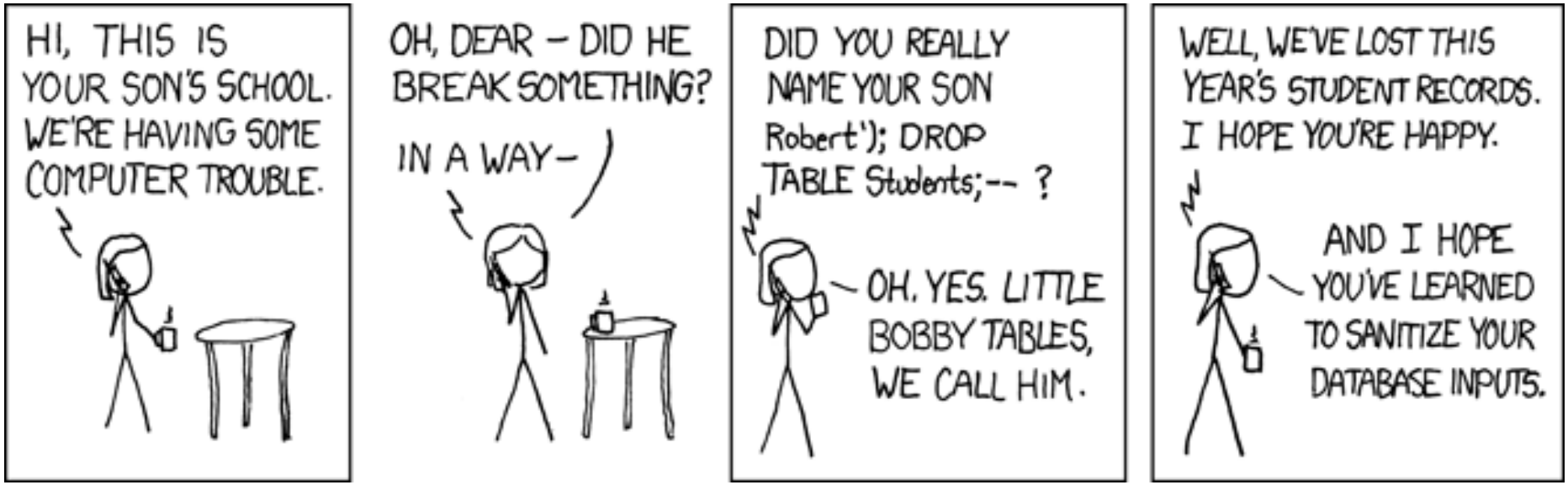
Can't trust data in database either!

Later someone runs a command where user = 'badguy'

Command executes; here resets admin password

Source: https://portswigger.net/web-security/sql-injection

Now we know why the comic on the course web site is funny!



Source: https://xkcd.com/327/

Practice

Assume a log in form issues the following SQL behind the scenes where user input is used directly in the SQL:

SELECT * **FROM** Users **WHERE** UserName = ‘username’ **AND** Password = ‘password’

The site then logs you in if one row is returned by the query

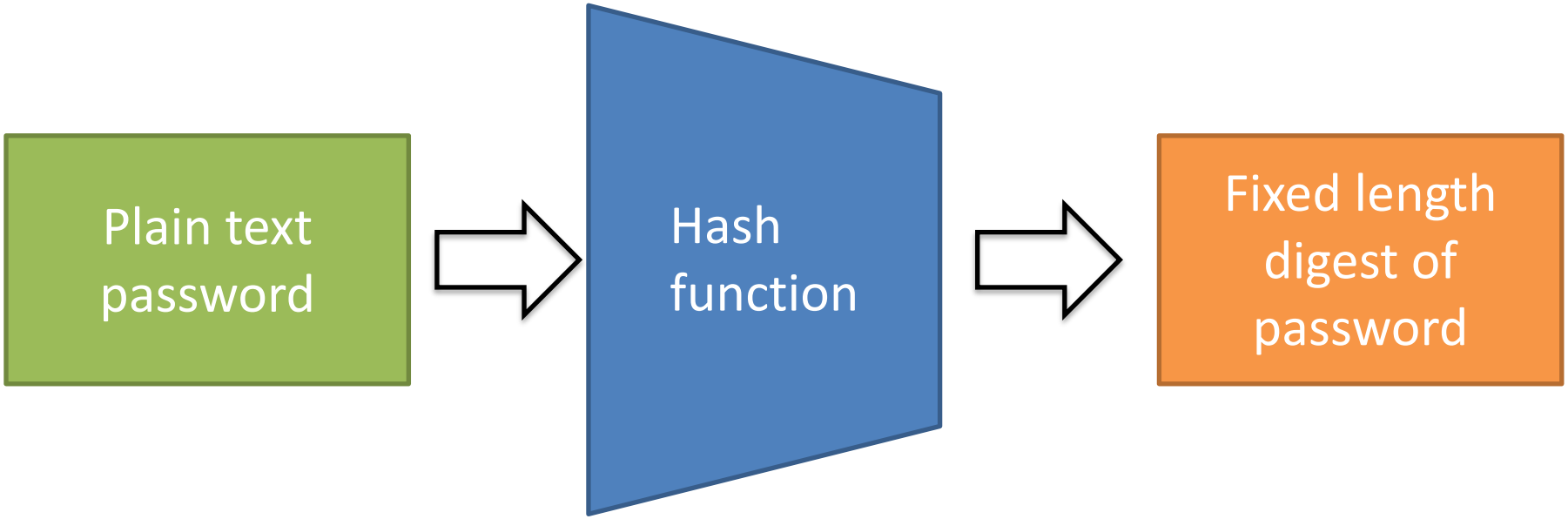
What could you enter in the username or password fields to log in as ‘administrator’ even if you do not know the password?

Enter: administrator’ --
Command now:
SELECT * **FROM** Users **WHERE** UserName = ‘administrator’--
AND Password = ‘password’

Agenda

1. MySQL permissions
2. Demo: SQL injection attacks
3. Password storage/salt and pepper
4. Password cracking

Review: hashing takes plain text and outputs a fixed-length digest



Input:
"my secret password"

Output:
a7303f3eee5f3ff1942bfbb1797ea0af

Hash function is a mathematical one-way trap door

- **Cannot find plain text in "reasonable" amount of time given only the hash digest**
- **Or can we?**

DO NOT store user passwords in plain text!

UserID	UserName	UserPassword
1	testuser	password
2	testuser1	password
3	testuser2	password1
4	testuser3	my secret password

Do not store passwords in plain text

Hash Password

Note: same password results in same hash

If adversary steals passwords, cannot read plain-text password

UserID	UserName	HashedPassword
1	testuser	5f4dcc3b5aa765d61d8327deb882cf99
2	testuser1	5f4dcc3b5aa765d61d8327deb882cf99
3	testuser2	7c6a180b36896a0a8c02787eeafb0e4c
4	testuser3	a7303f3eee5f3ff1942bfbb1797ea0af

Instead store hash of password

On log in: hash plain text password and compare with database

Username: “testuser”
Password: “password”

Hash user’s plain text password and look for match in database

Because hash function is deterministic, same password will always result in same digest

Hashed password:
5f4dcc3b5aa765d61d8327deb882cf99

Hashes match for testuser

UserID	UserName	HashedPassword
1	testuser	5f4dcc3b5aa765d61d8327deb882cf99
2	testuser1	5f4dcc3b5aa765d61d8327deb882cf99
3	testuser2	7c6a180b36896a0a8c02787eeafb0e4c
4	testuser3	a7303f3eee5f3ff1942bfbb1797ea0af

User submitted valid password

Dictionary attack: try all words in a dictionary looking for a match

Username: “testuser”
Password: “password”

Password: “aardvark”
Password: “alice”
Password: “anteater”
…
Password: “password”

Change your password if in dictionary!

Hash Password

Assume adversary steals hashed passwords

Hashed password:
5f4dcc3b5aa765d61d8327deb882cf99

Dictionary attack:
Hash all words in a dictionary, if word hash matches database hash, password is “cracked”

UserID	UserName	HashedPassword
1	testuser	5f4dcc3b5aa765d61d8327deb882cf99
2	testuser1	5f4dcc3b5aa765d61d8327deb882cf99
3	testuser2	7c6a180b36896a0a8c02787eeafb0e4c
4	testuser3	a7303f3eee5f3ff1942bfbb1797ea0af

Will not crack if user’s password not in dictionary

Crack one, crack all with same password

Rainbow table attacks precompute all possible character combinations

Username: “testuser”
Password: “password”

Hash Password

Hashed password:
5f4dcc3b5aa765d61d8327deb882cf99

Password: “a”
Password: “aa”
Password: “aaa”
…
Password: “password”

Assume adversary steals hashed passwords

<u>Rainbow table attack:</u>
Precompute all character combinations up to certain length
Store resulting hash for each combo

UserID	UserName	HashedPassword
1	testuser	5f4dcc3b5aa765d61d8327deb882cf99
2	testuser1	5f4dcc3b5aa765d61d8327deb882cf99
3	testuser2	7c6a180b36896a0a8c02787eeafb0e4c
4	testuser3	a7303f3eee5f3ff1942bfbb1797ea0af

Look up database password in rainbow table

Lots of time and storage needed

Length limited

Use salt to prevent attacks

Username: “testuser”
Password: “password”

Password

Password + **Salt**: “password**.ef_ob’3**”
Salted hashed password:
62c21dd30b2d7e6e6671628458aeaf1f

Salt:

- **Random string of characters appended (or prepended or both) to password before hash**
- **Each user gets unique salt**
- **Salt stored in plain text in database**
- **User need not know value of salt, it is added on server side**

If salt is long (say 64 characters) rainbow table is impractical

Dictionary attack still possible

- **Password plus salt unlikely to be in database**
- **But add salt to each word**
- **Slows adversary**

UserID	UserName	Salt	SaltedPassword
1	testuser	.ef_ob'3	62c21dd30b2d7e6e6671628458aeaf1f
2	testuser1	\s#>2!x}	a9055805cb7e588dc27945cb95067f6b
3	testuser2	as=8KlA=	19e3385ed6bfe321b36b6bc4290bea0b
4	testuser3	n%lzA7QQ	a6fc8f715df44839b50cf31b639b961c

Adding pepper is even better

Username: “testuser”
Password: “password”

Password

Password + Salt **+ Pepper**: “password.ef_ob’3**Secret**”
Salted hashed password:
2811922850bbcd79683b58e43d1ab76f

UserID	UserName	Salt	SaltedPassword
1	testuser	.ef_ob'3	62c21dd30b2d7e6e6671628458aeaf1f
2	testuser1	\s#>2!x}	a9055805cb7e588dc27945cb95067f6b
3	testuser2	as=8KlA=	19e3385ed6bfe321b36b6bc4290bea0b
4	testuser3	n%lzA7QQ	a6fc8f715df44839b50cf31b639b961c

Pepper:

- **Random string of characters appended to password + salt before hash**
- **Pepper kept secret, not stored in database**
- **One pepper for all users**

Another variant

- **Pepper is one character chosen at random for each user**
- **Not stored**
- **On log, try ‘a’, then ‘b’**
  - **Will eventually find match**
  - **Slows adversary**

Agenda

1. MySQL permissions
2. Demo: SQL injection attacks
3. Password storage/salt and pepper
4. Password cracking

Exercise

Enter username and password at phoney sign up site:
https://cs.dartmouth.edu/~tjp/cs61/saveUser.html

- Site stores entries into Users table on sunapee cs61 schema
- Table has unique constraint on UserName (so choose something else if what you enter is already taken)
- NOTE: for demonstration purposes only, it stores the password in plain text! *You would not do this is production*!
- Also stores hashed and salted hash passwords

Assume an adversary does a SQL injection attack (or otherwise steals Users table) and gets usernames with hashed and salted passwords

- What can they do? They do not have the users' passwords
- Enter hashcat!

Hashcat is a password hashing tool

1. **Download usertable.csv from Sunapee**

2. **Extract hashes from usertable.csv**

```
cat usertable.csv | awk "-F," '{print $4}' > unsalted.txt
cat usertable.csv | awk "-F," '{print $6 ":" $5}' > salted.txt
```

m is hash type:
- **0 = MD5**
- **10 = MD5(password+salt)**

a is attack mode
- **0 = dictionary**
- **Rockyou.txt is dictionary**
- **potfile-disable means restart**

3. **Crack passwords**

Unsalted

```
hashcat -m 0 -a 0 unsalted.txt ~/Downloads/rockyou.txt --potfile-disable
```

Salted

```
hashcat -m 10 -a 0 salted.txt ~/Downloads/rockyou.txt --potfile-disable
```