STOP | THINK | CONNECT
www.stopthinkconnect.org
So you're telling me a Nigerian prince really didn't leave me any money?
STOP. THINK. CONNECT. When in doubt, throw it out! Avoid clicking on suspicious emails, links & social media posts - even if you know the source.

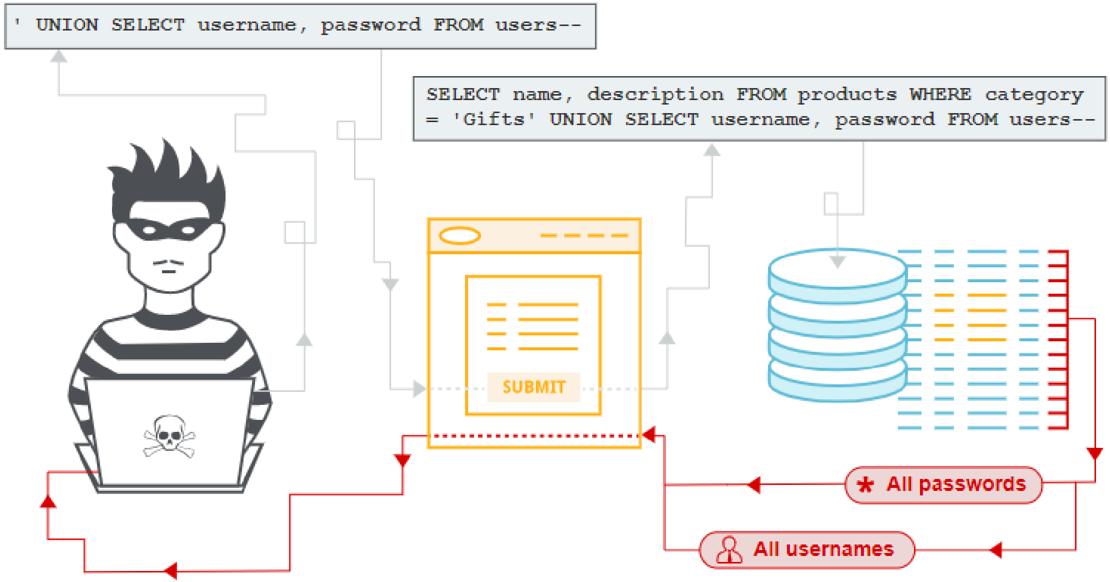
Module 2-8

Data Security

Objectives

- SQL Injection Attack
- Prepared statements
- Hashing
- Salt
- Encryption

SQL Injection attacks



```
' UNION SELECT username, password FROM users--
```

```
SELECT name, description FROM products WHERE category
= 'Gifts' UNION SELECT username, password FROM users--
```

SQL Injection attacks

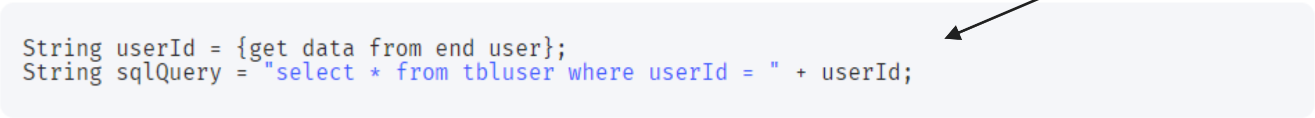
- Makes it possible to execute malicious SQL statements
  - SQL statements control database server
  - Attackers can bypass authentication and authorization
  - Can add modify and delete records in a database

Preventing SQL Injection

- Parameterized Queries
- Input Validation
- Limit Database User Privileges

Preventing SQL Injection

- Parameterized Queries



```
String userId = {get data from end user};
String sqlQuery = "select * from tbluser where userId = " + userId;
```

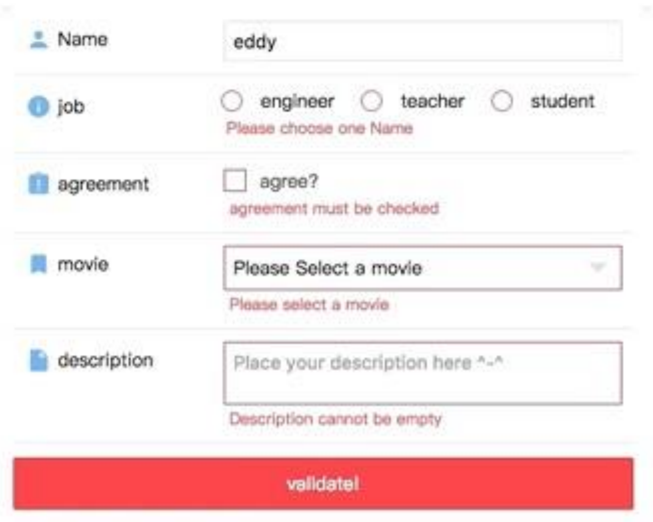
If this is executed with a userId of 132, it will look like this:
SELECT * FROM tbluser WHERE userId=132;

A hacker can alter a user request to send SQL code where the userId says 2 OR 1=1;
This will cause the sqlQuery to read:
SELECT * FROM tbluser WHERE userId=2 OR 1=1;

Because 1=1 is always true, it will return all data from the table!

Preventing SQL Injection

- Input Validation



Name: eddy

job: ○ engineer ○ teacher ○ student

Please choose one Name

agreement: □ agree?

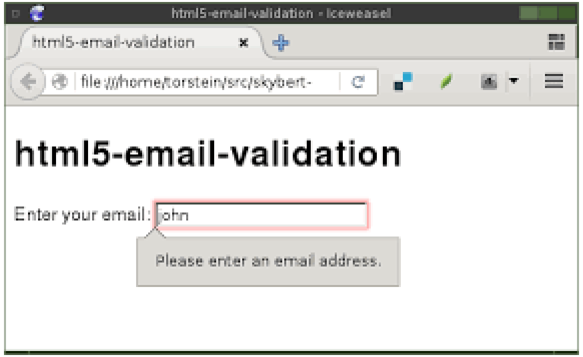
agreement must be checked

movie: Please Select a movie

Please select a movie

description: Place your description here ^-^

Description cannot be empty



file:///home/torstein/src/skybert-

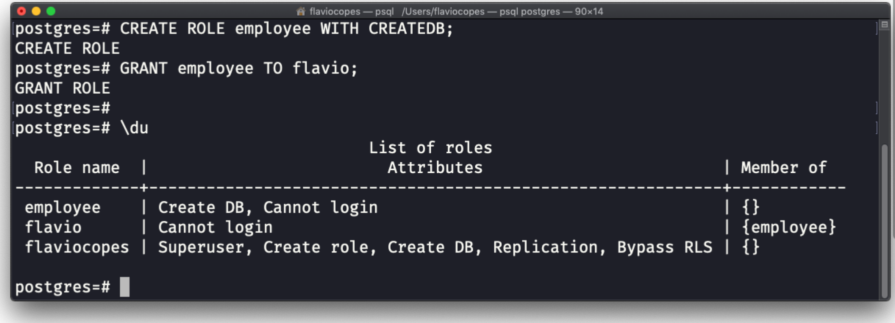
html5-email-validation

Enter your email: john

Please enter an email address.

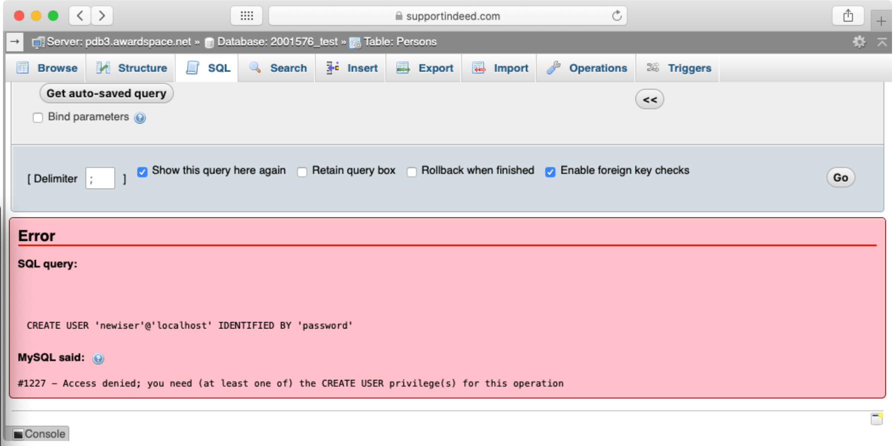
Preventing SQL Injection

- Limit Database User Privileges


flaviocopes — psql /Users/flaviocopes — psql postgres — 90×14

```
postgres=# CREATE ROLE employee WITH CREATEDB;
CREATE ROLE
postgres=# GRANT employee TO flavio;
GRANT ROLE
postgres=#
postgres=# \du
                                  List of roles
  Role name  |                         Attributes                         | Member of
-------------+------------------------------------------------------------+------------
 employee    | Create DB, Cannot login                                    | {}
 flavio      | Cannot login                                               | {employee}
 flaviocopes | Superuser, Create role, Create DB, Replication, Bypass RLS | {}

postgres=# 
```



□ Bind parameters

[Delimiter ;] ☑ Show this query here again □ Retain query box □ Rollback when finished ☑ Enable foreign key checks

Error

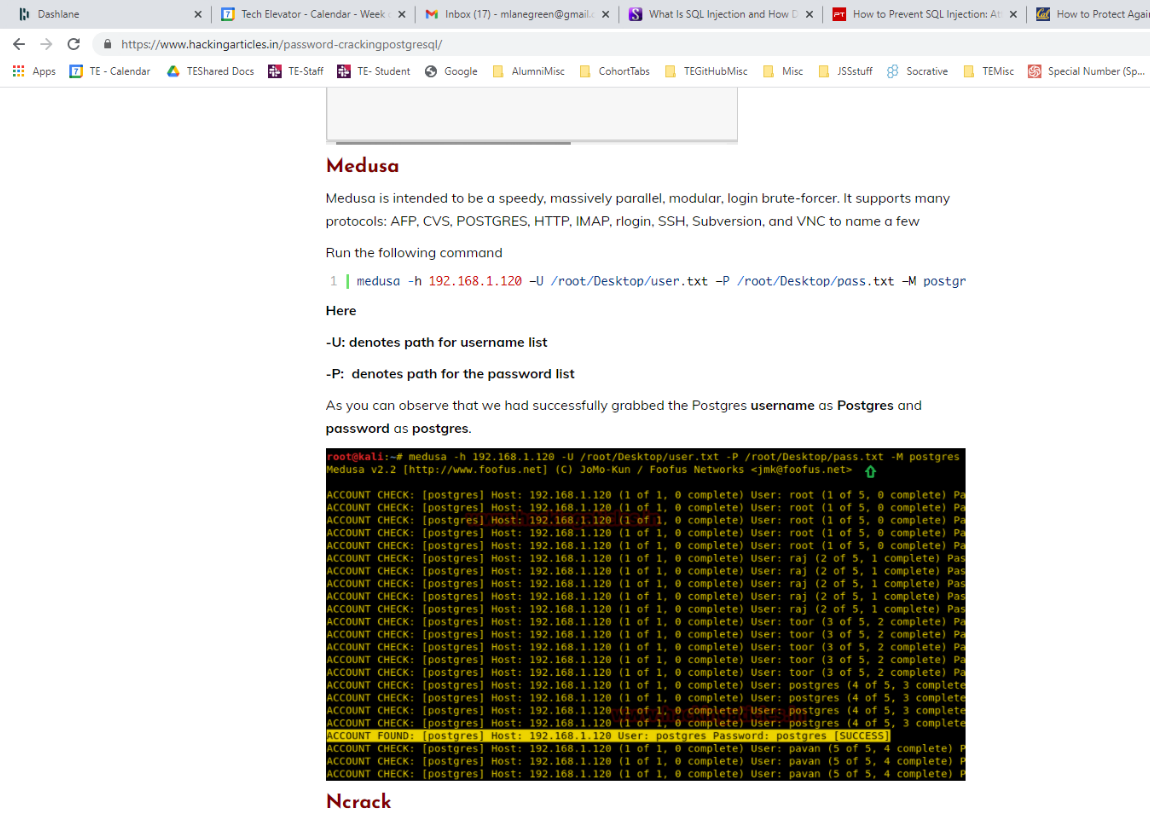
SQL query:

```
CREATE USER 'newiser'@'localhost' IDENTIFIED BY 'password'
```

MySQL said:

```
#1227 - Access denied; you need (at least one of) the CREATE USER privilege(s) for this operation
```

Preventing SQL Injection


Dashlane
Tech Elevator - Calendar - Week
Inbox (17) - mlanegreen@gmail.
What Is SQL Injection and How D
How to Prevent SQL Injection: At
How to Protect Agai
https://www.hackingarticles.in/password-crackingpostgresql/
Apps
TE - Calendar
TEShared Docs
TE-Staff
TE- Student
Google
AlumniMisc
CohortTabs
TEGitHubMisc
Misc
JSSstuff
Socrative
TEMisc
Special Number (Sp...

Medusa

Medusa is intended to be a speedy, massively parallel, modular, login brute-forcer. It supports many protocols: AFP, CVS, POSTGRES, HTTP, IMAP, rlogin, SSH, Subversion, and VNC to name a few

Run the following command

```
medusa -h 192.168.1.120 -U /root/Desktop/user.txt -P /root/Desktop/pass.txt -M postgr
```

Here

-U: denotes path for username list

-P: denotes path for the password list

As you can observe that we had successfully grabbed the Postgres **username** as **Postgres** and **password** as **postgres**.

```
root@kali:~# medusa -h 192.168.1.120 -U /root/Desktop/user.txt -P /root/Desktop/pass.txt -M postgres
Medusa v2.2 [http://www.foofus.net] (C) JoMo-Kun / Foofus Networks <jmk@foofus.net>

ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: root (1 of 5, 0 complete) Pa
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: root (1 of 5, 0 complete) Pa
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: root (1 of 5, 0 complete) Pa
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: root (1 of 5, 0 complete) Pa
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: root (1 of 5, 0 complete) Pa
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: raj (2 of 5, 1 complete) Pas
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: raj (2 of 5, 1 complete) Pas
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: raj (2 of 5, 1 complete) Pas
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: raj (2 of 5, 1 complete) Pas
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: raj (2 of 5, 1 complete) Pas
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: toor (3 of 5, 2 complete) Pa
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: toor (3 of 5, 2 complete) Pa
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: toor (3 of 5, 2 complete) Pa
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: toor (3 of 5, 2 complete) Pa
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: toor (3 of 5, 2 complete) Pa
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: postgres (4 of 5, 3 complete
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: postgres (4 of 5, 3 complete
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: postgres (4 of 5, 3 complete
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: postgres (4 of 5, 3 complete
ACCOUNT FOUND: [postgres] Host: 192.168.1.120 User: postgres Password: postgres [SUCCESS]
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: pavan (5 of 5, 4 complete) P
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: pavan (5 of 5, 4 complete) P
ACCOUNT CHECK: [postgres] Host: 192.168.1.120 (1 of 1, 0 complete) User: pavan (5 of 5, 4 complete) P
```

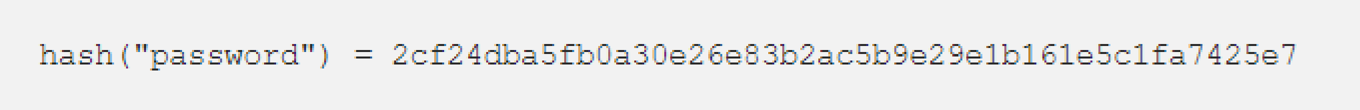
Ncrack

Protecting sensitive data

- How many stories have we heard regarding data breaches divulging sensitive information??
- Data stored in a database hacked
- To stop this, we need to have data stored in a database in such a way that it is not readable by unauthorized parties
- Data can be protected by either hashing or encryption

Hashing

- Using an algorithm to map data of any size to a fixed length.
  - Called a hash code or hash value
  - Many different algorithms (MD2, MD4, MD5, SHA, SHA1, SHA2)
- Is a one-way function
  - Technically it is possible to reverse-hash, would require immense computing power therefore unfeasible
- Meant to verify that a file or piece of data has not been altered



```
hash("password") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e7
```

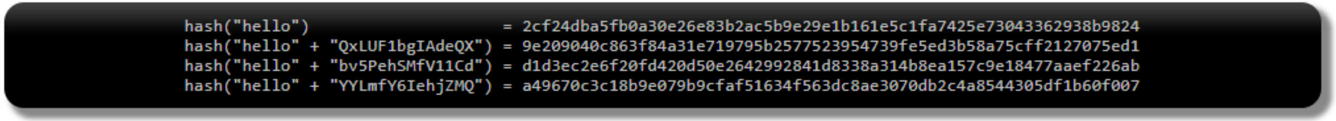
Hashing

- Hashed output of the same string will be the same.
- Hashed data conforms to algorithm in terms of storage size
- The stronger hash function used, the more storage required, the slower the performance but minimal chance of having collision
- Humans are predictable, passwords tend to be memorable keywords, phrases, or numbers
- Hackers create a “rainbow” table of possible passwords and run this through while trying to hack in

    - Salt

SALT

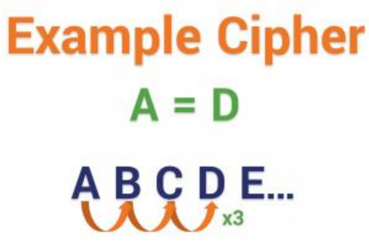
- Unique value added to end of password to create a different value.
- Adds layer of security to hashing process
    - Helps protect against brute force
- Because salt is unique, produced hash of same password will not be the same.

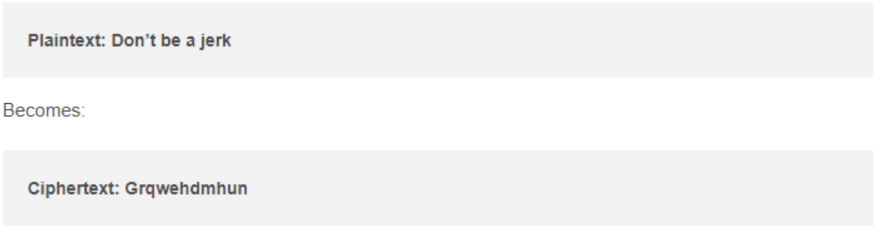


```
hash("hello")                    = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hello" + "QxLUF1bgIAdeQX") = 9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1
hash("hello" + "bv5PehSMfV11Cd") = d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aaef226ab
hash("hello" + "YYLmfY6IehjZMQ") = a49670c3c18b9e079b9cfaf51634f563dc8ae3070db2c4a8544305df1b60f007
```

Encryption

- Most effective way to achieve data security
- Practice of scrambling information
  - Needs a key to unscramble
- Two-way function





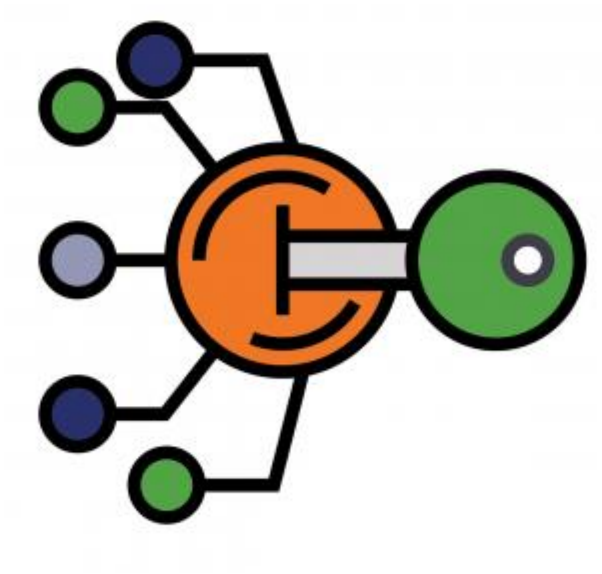
Plaintext: Don’t be a jerk

Becomes:

Ciphertext: Grqwehdmhun

Encryption algorithms

- Shift ciphers
- Substitution ciphers
- Transposition ciphers
- Polyalphabetic ciphers
- Nomenclature ciphers



Modern encryption algorithms

- Asymmetric Encryption
  - Public key example – 1 key encrypts, 1 key decrypts
  - Used in SSL/TLS transfer of data
- Symmetric Encryption
  - Closer to form of private key encryption
  - Each party has a key that encrypts and decrypts
  - After asymmetric encryption in SSL handshake, browser and server communicate with symmetric key that is passed along

Digital certificate

- Public key certificate
- Used for encryption and authentication
- Certificate authority (CA) is trusted third-party that provide certificate
    - Prevents attacker from impersonating a server

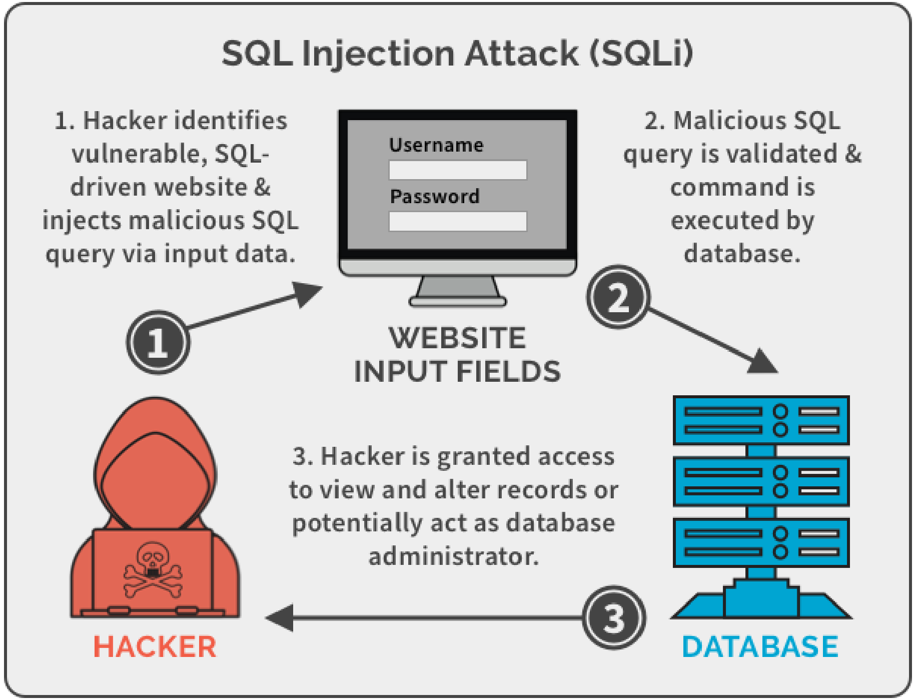
Man in the Middle Attack

- Attacker intercepts communications between two parties
    - Either to eavesdrop
    - Modify traffic
- Oldest form of cyber attacks
- Not as common as ransomware or phishing, still threat
- Encryption protocols (SSL/TLS) are best way to help protect against

Thinkstock

Objectives

- SQL Injection Attack



SQL Injection Attack (SQLi)

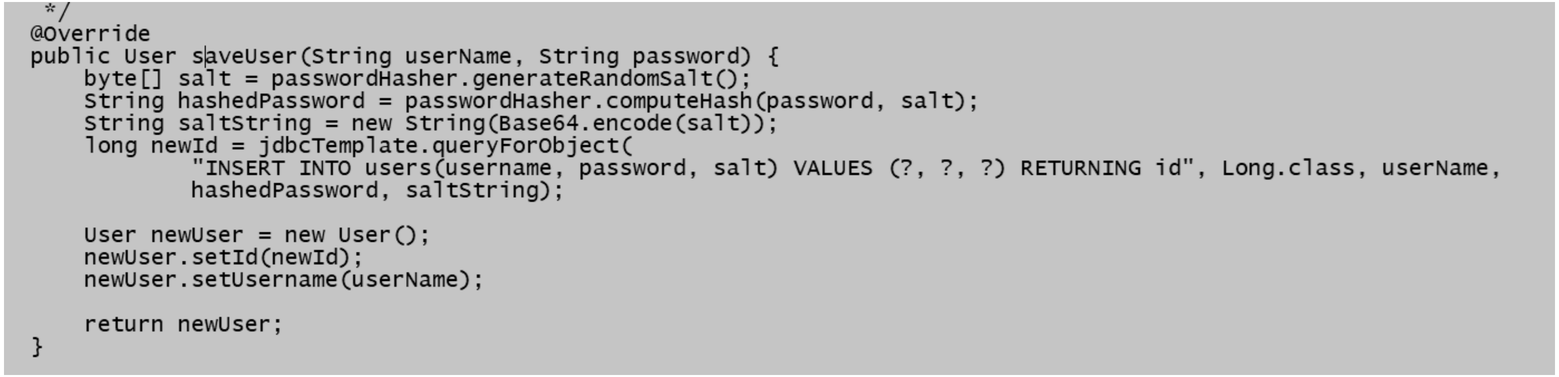
1. Hacker identifies vulnerable, SQL-driven website & injects malicious SQL query via input data.

2. Malicious SQL query is validated & command is executed by database.

3. Hacker is granted access to view and alter records or potentially act as database administrator.

Objectives

- SQL Injection Attack
- Prepared statements



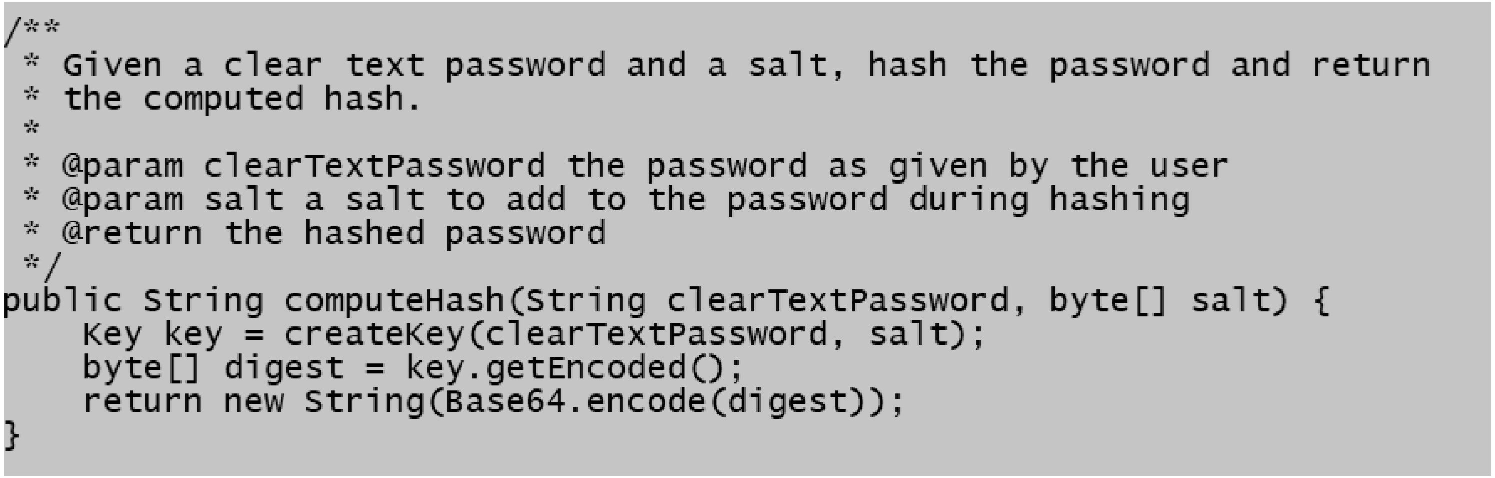
```
 */
@Override
public User saveUser(String userName, String password) {
    byte[] salt = passwordHasher.generateRandomSalt();
    String hashedPassword = passwordHasher.computeHash(password, salt);
    String saltString = new String(Base64.encode(salt));
    long newId = jdbcTemplate.queryForObject(
            "INSERT INTO users(username, password, salt) VALUES (?, ?, ?) RETURNING id", Long.class, userName,
            hashedPassword, saltString);

    User newUser = new User();
    newUser.setId(newId);
    newUser.setUsername(userName);

    return newUser;
}
```

Objectives

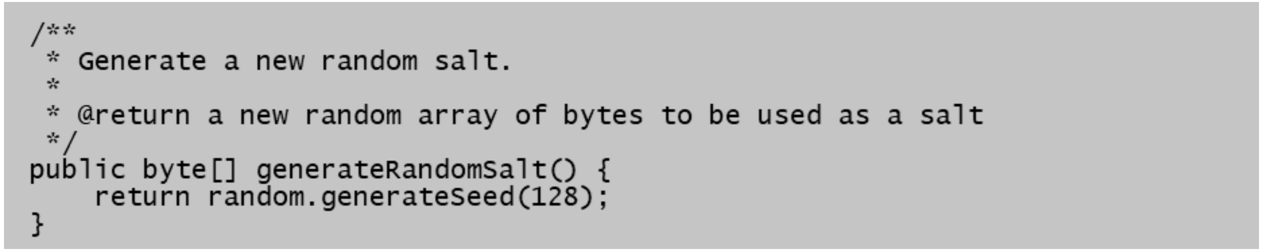
- SQL Injection Attack
- Prepared statements
- Hashing



```
/**
 * Given a clear text password and a salt, hash the password and return
 * the computed hash.
 *
 * @param clearTextPassword the password as given by the user
 * @param salt a salt to add to the password during hashing
 * @return the hashed password
 */
public String computeHash(String clearTextPassword, byte[] salt) {
    Key key = createKey(clearTextPassword, salt);
    byte[] digest = key.getEncoded();
    return new String(Base64.encode(digest));
}
```

Objectives

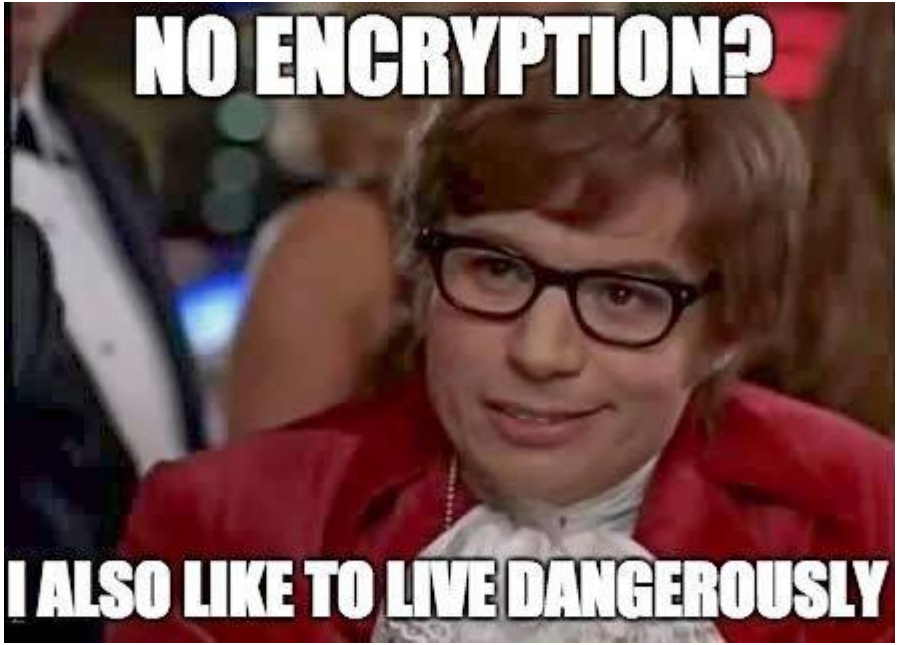
- SQL Injection Attack
- Prepared statements
- Hashing
- Salt



```
/**
 * Generate a new random salt.
 *
 * @return a new random array of bytes to be used as a salt
 */
public byte[] generateRandomSalt() {
    return random.generateSeed(128);
}
```

Objectives

- SQL Injection Attack
- Prepared statements
- Hashing
- Salt
- Encryption



Let’s Code!