**Komposition: A1 SQL Injektion**

**Agenda:**

1. Forklar SQL injection og vis et eksempel.
2. Forklar hvordan et prepared Statement kan forhindre SQL injection.
3. Forklar hvordan en placeholder, kan gøre arbejdet, hvor et prepared statement ikke kan forhindre SQL injection.
4. Forklar hvordan logging kan bruges til at monitorer SQL injections forsøg.

**Give an example of a SQL inject which will give all users in a user table**

* I tilfælde hvor userId = input fra brugeren og SQL query i backend:

” SELECT \* FROM users WHERE user\_id = ‘userId‘; ”

Ved at angive 1’ or 1=1; --, som input userId, vil overstående sql query eksekveres, som:

“SELECT \* FROM users WHERE user\_id = ‘1’ or 1=1; -- ‘; “

Alle brugere vil blive returneret, eftersom at 1=1 altid er sandt.

**Explain how prepared statements prevent SQL injection**

**Eksempler på Prepared Statement:**

* PreparedStatement statement = conn.prepareStatement("select banks from loan where loan\_type=?");  
  prestmt.setString(1,loanType);
* TypedQuery<Movie> query = em.createQuery("SELECT m FROM Movie m WHERE m.title = :title", Movie.class);

List<Movie> movies = query.setParameter("title", title).getResultList();

Prepared statements bliver håndteret i databasen og overordnet forhindre de SQL-injektions ved at lave bruger input til ikke-eksekverbar kode. Det vil sige at, hvis en bruger angiver sit brugernavn som ”drop table users; -- ” vil koden ikke blive eksekveret, men læst som en String.

**Prepared statements bliver håndteret på 3 overordnede måder, afhængig af databasen:**

1. **Luksus**
   * Prepared statement bliver sendt til databasen og lavet om til en funktion, samt en query plan, som benytter relationel algebra.
   * Syntakscheck udføres.
   * Den data der bliver sat ind i prepared statement bliver valideret.
   * Hurtig eksekvering.
2. **Normal**
   * Syntakscheck udføres, men benytter ikke relationel algebra.
   * Langsom eksekvering.
3. **Discount**
   * Minimal syntakscheck udføres.
   * Langsom eksekvering og dårlig data validering.

MySQL håndterer prepared statements, som en kombination af normal og luksus.

**Explain how to use placeholders in cases where prepared statements cannot do the job**

Man kan ikke bruge prepared statement, når input er en direkte del af syntaksen dette skyldes at input i dette tilfælde vil fortolkes som eksekverbar kode, dette kan udgøre en risiko for sql injektions. Derfor anvendes placeholders i stedet. Ved at anvende white- eller blacklisting i sammenspil med placeholders minimeres denne risiko.

* Et eksempel på dette tilfælde er: ORDER BY ?
* Her er spørgsmålstegnet placeholder for et givet kolonnenavn. Derfor anvendes ”white- eller blacklisting”.
* Whitelisting er en liste af godkendte inputs, som man gerne må søge efter, f.eks. kolonne navne
* Blacklisting er en liste af input som ikke må benyttes, f.eks. ”drop”, ”;” , ”--” ”or”.

**Eksempel på whitelist og placeholder:**

* Nedenstående er en funktion der går igennem en whitelist og returnere et input hvis det er whitelistet

public static String **field**(String name, String... whitelist) {

for (String option : whitelist) {

if(option.equals(name))

return name;

}

throw new IllegalArgumentException("Bad input bro..."); }

private static void **sortExample**(String sortKey) {

String sql = "select \* from junk order by "+field(sortKey, "name", "id")+";"; }

**Explain how logging could be used to monitor injection attempts**

* Log alle SQL exceptions, da disse kun forekommer når nogen forsøger at lave injektion eller ved forbindelses problemer.
* Forekommer en SQL exception burde en e-mail sendes direkte til systemadministratoren, så han/hun kan undersøge det.
* Forekommer mere end X antal exceptions fra samme IP-adresse, bør denne IP-adresse, der laver injektion forsøg, blacklistes.