SQL Injection pode ocorrer em qualquer aplicação que interaja com um banco de dados, independentemente da linguagem ou framework utilizado. Vou te mostrar como o SQL Injection pode acontecer em uma aplicação **Spring Boot** (Java) e, em seguida, como preveni-lo usando boas práticas.

Exemplo de SQL Injection em Spring Boot

Código Vulnerável

Suponha que você tenha um repositório em Spring Boot que executa uma consulta SQL diretamente com concatenação de strings:

java
Copy
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

@Repository
public class UserRepository {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public String findUserByUsernameAndPassword(String username,
String password) {
        String query = "SELECT * FROM users WHERE username = '" +
username + "' AND password = '" + password + "'";
        return jdbcTemplate.queryForObject(query, String.class);
    }
}
```

Neste exemplo, se um invasor inserir o seguinte no campo `username`:

Copy
```
' OR '1'='1
```

E no campo `password`:

Copy
```
' OR '1'='1
```

A consulta SQL gerada será:

sql
Copy
```
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = ''
OR '1'='1';
```

Isso permitirá que o invasor faça login sem credenciais válidas.

Como Prevenir SQL Injection em Spring Boot

1. Use Prepared Statements com JdbcTemplate

O `JdbcTemplate` do Spring suporta prepared statements, que separam o código SQL dos dados do usuário, evitando SQL Injection.

Aqui está o código corrigido:

java
Copy
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

@Repository
public class UserRepository {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public String findUserByUsernameAndPassword(String username,
String password) {
        String query = "SELECT * FROM users WHERE username = ? AND
password = ?";
        return jdbcTemplate.queryForObject(query, new
Object[]{username, password}, String.class);
    }
}
```

Neste exemplo, os placeholders (`?`) são substituídos pelos valores de `username` e `password` de forma segura, evitando a injeção de SQL.

---

2. Use Spring Data JPA

O **Spring Data JPA** é uma camada de abstração que facilita o acesso ao banco de dados e já utiliza prepared statements internamente.

Aqui está um exemplo usando Spring Data JPA:

java
Copy
```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT u FROM User u WHERE u.username = :username AND
u.password = :password")
```

```
    User findByUsernameAndPassword(@Param("username") String username,
@Param("password") String password);
}
```

Neste exemplo, a consulta é escrita em JPQL (Java Persistence Query Language), que é automaticamente protegida contra SQL Injection.

---

3. Valide e Sanitize Entradas

Mesmo usando prepared statements ou JPA, é importante validar e sanitizar as entradas do usuário para garantir que estejam no formato esperado.

Exemplo de validação simples:

java
Copy
```
import org.springframework.util.StringUtils;

public class UserService {

    public boolean isValidUser(String username, String password) {
        if (!StringUtils.hasText(username) ||
!StringUtils.hasText(password)) {
            throw new IllegalArgumentException("Username and password
cannot be empty");
        }
        // Lógica adicional de validação
        return true;
    }
}
```

---

4. Use Hibernate com Criteria API

A **Criteria API** do Hibernate permite construir consultas de forma programática e segura.

Exemplo:

java
Copy
```
import org.hibernate.Session;
import org.hibernate.criterion.Restrictions;
import org.springframework.stereotype.Repository;

@Repository
public class UserRepository {

    @Autowired
    private SessionFactory sessionFactory;

    public User findByUsernameAndPassword(String username, String
password) {
```

```
        Session session = sessionFactory.getCurrentSession();
        return (User) session.createCriteria(User.class)
                .add(Restrictions.eq("username", username))
                .add(Restrictions.eq("password", password))
                .uniqueResult();
    }
}
```

---

5. Configure um Web Application Firewall (WAF)

Um WAF pode ajudar a detectar e bloquear tentativas de SQL Injection antes que elas cheguem à sua aplicação.

Exemplo Completo em Spring Boot

Aqui está um exemplo completo de uma aplicação Spring Boot segura contra SQL Injection:

Entidade `User`

java
Copy
```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String password;

    // Getters e Setters
}
```

Repositório `UserRepository`

java
Copy
```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
```

```
    @Query("SELECT u FROM User u WHERE u.username = :username AND
u.password = :password")
    User findByUsernameAndPassword(@Param("username") String username,
@Param("password") String password);
}
```

Serviço `UserService`

java
Copy
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User authenticate(String username, String password) {
        return userRepository.findByUsernameAndPassword(username,
password);
    }
}
```

Controlador `UserController`

java
Copy
```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @PostMapping("/login")
    public String login(@RequestParam String username, @RequestParam
String password) {
        User user = userService.authenticate(username, password);
        return user != null ? "Login successful" : "Invalid
credentials";
    }
}
```

Conclusão

Em Spring Boot, você pode evitar SQL Injection usando **prepared statements**, **Spring Data JPA**, **Hibernate Criteria API** e boas práticas de validação de entradas. Essas abordagens garantem que sua aplicação seja segura e resistente a ataques de injeção de SQL