Please Complete The Socrative Pulse Survey

URL: gosocrative.com

Room Name: JAVAGOLD

What are the 7 things wrong with this code example from yesterday's exercise?

```
@Override
public Project createProject(Project newProject) {
  int newId = 0;
  String sql = "INSERT INTO project (name, from_date, to_date) VALUES (?, ?)+"
"RETURNING project_id;"
  try {
     newId = jdbcTemplate.update(sql, newProject.getName(),newProject.getFromDate());
  } catch (Exception e) {
     throw new DaoException("Unable to connect to server or database", e);
  } catch (BadSqlGrammarException e) {
     throw new DaoException("SQL syntax error", e);
  } catch (DataIntegrityViolationException e) {
     throw new DaoException("Data integrity violation", e);
  }
  return getProjectById(newId);
}
```

Integration Testing and Security

Today’s Objectives

1. Integration Testing
2. Testing DAOs
3. SQL Injection
4. Hashing
5. Encryption

Integration Testing

Integration Testing is a broad category of tests that validate the integration between units of code or code and outside dependencies such as databases or network resources.

Integration tests in Java

- Use the same tools as unit tests (i.e. JUnit)
- Usually slower than unit tests
- More complex to write and debug
- Can have dependencies on outside resources like files or a database

Test Database Approaches - Shared Database

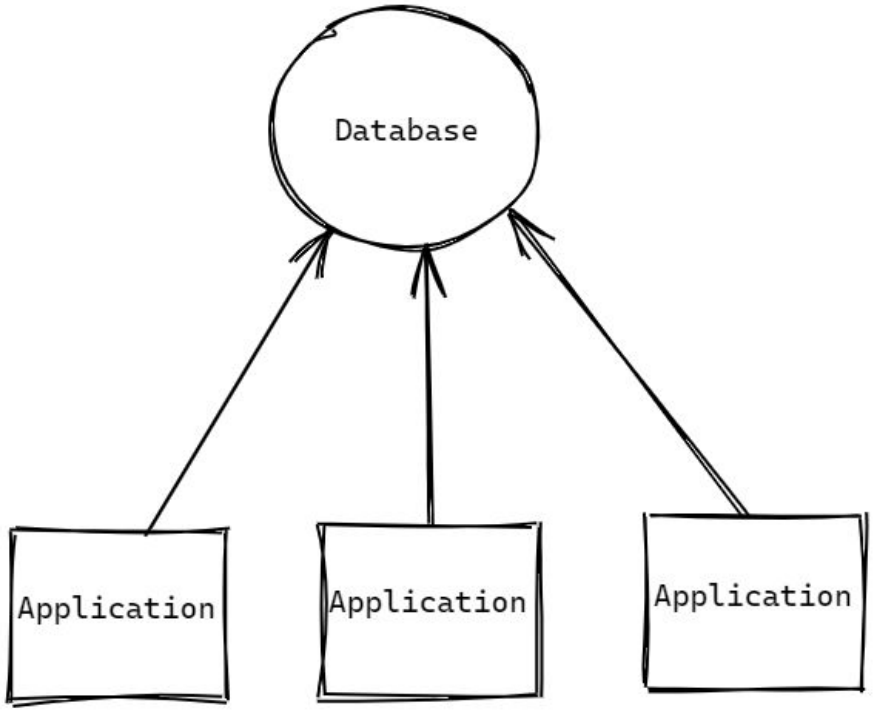
All Developers share a remote test database on the network.

Pros:

- Easy Developer setup
- 1 Setup for all developers
- Production-like software and hardware
- Can be managed by DBAs

Cons:

- Unreliable
- Brittle
- No Isolation
- Temptation to rely on existing data



Test Database Approaches - Local Database

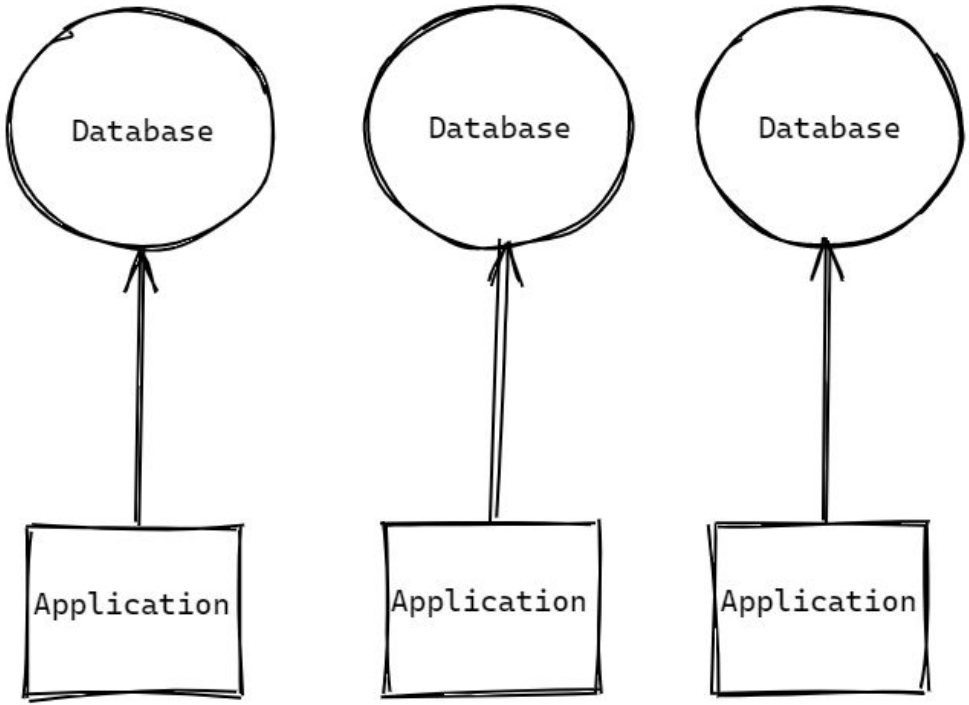
Each developer has their own copy of the database on their computer.

Pros:

- Production-like software
- Reliable
- Isolation

Cons:

- Requires developer to act as DBA
- RDBMS needs to be installed locally, requiring additional licences
- Hardware is not production like
- Production like data can be difficult
- Inconsistent across machines

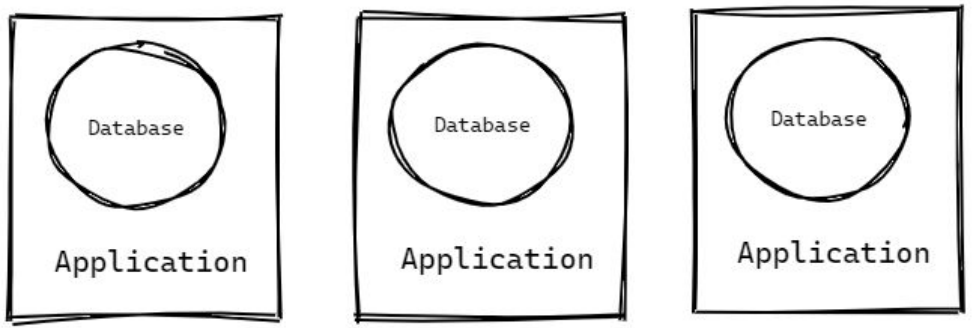


Test Database Approaches - Embedded Database

An in-memory database server is started and managed by test code and run inside the application

Pros:

- Very reliable
- Consistent across machines
- Lightweight
- Supports Continuous Integration



Cons:

- Software and hardware is not production like
- Can not use proprietary features of an RDBMS
- Production like data can be difficult

DAO Testing

Integration tests should be:

- *Repeatable:* If the test passes/fails on first execution, it should pass/fail on second execution if no code has changed.
- *Independent:* A test should be able to be run on it's own, independently of other tests, **OR** together with other tests and have the same result either way.
- *Obvious:* When a test fails, it should be as obvious as possible why it failed.

Integration Test should *never use existing data*.

They should always provide their own data.

DAO Integration Testing

Integration tests with a database should ensure that the DAO code functions correctly:

- SELECT statements are tested by inserting dummy data before the test
- INSERT statements are tested by searching for the data
- UPDATE statements are tested by verifying dummy data has been changed
- DELETE statements are tested by seeing if dummy data is missing

Mocking

- Make a replica or imitation
- Creating objects that simulate the behavior of real objects
- Typically used in unit testing, but we need to create fake data in order to test CRUD statements

@PostConstruct method

- Generally set up the data source in a @PostConstruct method:

```
 /* This method creates the temporary database to be used for the tests. */
    @PostConstruct
    public void setup() {
        if (System.getenv("DB_HOST") == null) {
            adminDataSource = new SingleConnectionDataSource();
            adminDataSource.setUrl("jdbc:postgresql://localhost:5432/postgres");
            adminDataSource.setUsername("postgres");
            adminDataSource.setPassword("postgres1");
            adminJdbcTemplate = new JdbcTemplate(adminDataSource);
            adminJdbcTemplate.update("DROP DATABASE IF EXISTS \"" + DB_NAME +
"\";");
            adminJdbcTemplate.update("CREATE DATABASE \"" + DB_NAME + "\";");
        }
    }
```

https://www.baeldung.com/spring-postconstruct-predestroy

@Before method

- Where we would insert mocked data into the database:

```
@Before
public void setup() {
    sut = new JdbcCityDao(dataSource);
    testCity = new City(0, "Test City", "CC", 99, 999);
}
```

@After method

- Want to rollback after each test method runs using the @After annotation:

```
/* After each test, we rollback any changes that were made to the database so that
 * everything is clean for the next test */
@After
public void rollback() throws SQLException {
    dataSource.getConnection().rollback();
}
```

Data Security

SQL Injection

```
"SELECT * FROM app_user WHERE UPPER(user_name) = '" +userName.toUpperCase()
        + "' "+ "AND password =  '" + password + "'"
```

What if I enter a valid username (Bill) and then the password as: `' OR 1=1--`

```
SELECT * FROM app_user WHERE UPPER(user_name) = 'BILL'
        AND password =  '' OR 1=1--'
```

What is the result of this query?

It returns the row of data were user_name = “Bill” regardless of the password, because OR 1=1 is always TRUE.

The trailing -- changes the remainder of the SQL statement into a comment, ending the query after OR 1=1.

Types of SQL Injection

1. Query Modification

   The attacker modifies the original query and then Ignores the rest of the original by adding -- at the end of their addition to comment it out.

2. Union Attack

   The attacker creates a UNION with an existing query that returns results from their query mixed with results of a legitimate query.

3. Stacked Queries

   The attacker ends the original query with a ; and then appends their own query onto the original..

4. Second Order Attack

   Parts of a SQL query are stored in related fields that execute together at a later time.

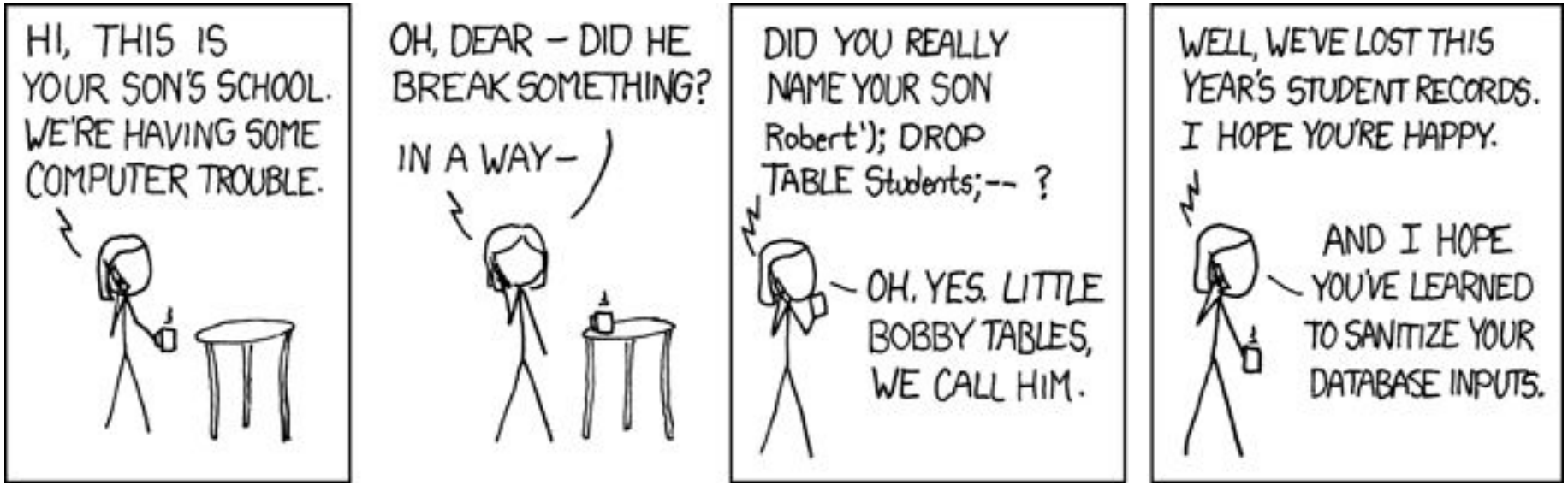
Preventing SQL Injection

1. **Parameterized Queries** - The single most effective thing you can do to prevent SQL injection is to use parameterized queries. ***If this is done consistently, First Order SQL injection will not be possible*****, however, second level attacks are still possible.**

2. **Input Validation** - Limiting the data that can be input by a user can certainly be helpful in preventing SQL Injection, but is by no means an effective prevention by itself. **If done consistently then Second Order SQL Injection will are also prevented.**

3. **Limit Database User Privileges** - A web application should always use a database user to connect to the database that has as few permissions as necessary.

SQL Injection/Security Resources
OWASP (Open Web Security Project) - SQL Injection
Hacksplaining
Past Student suggested Video on SQL Injection

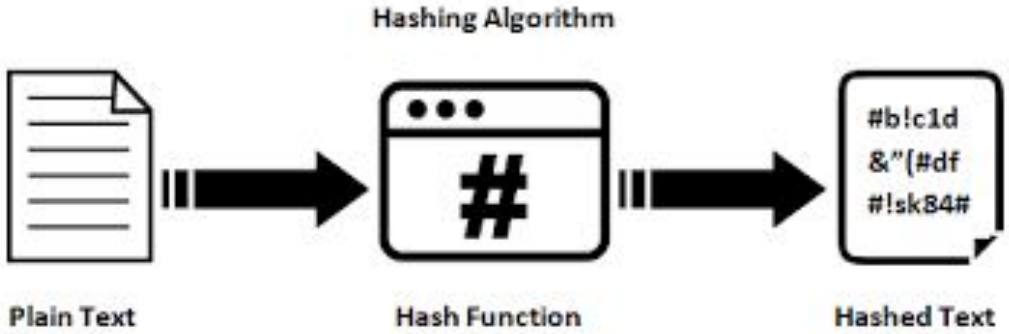


Hashing

A Hash Function is one that can map input data of arbitrary size to a fixed size output.

Hashing is 1-way, meaning that once data is hashed, the hash cannot be reversed back into the original data.

Commonly used to store passwords.

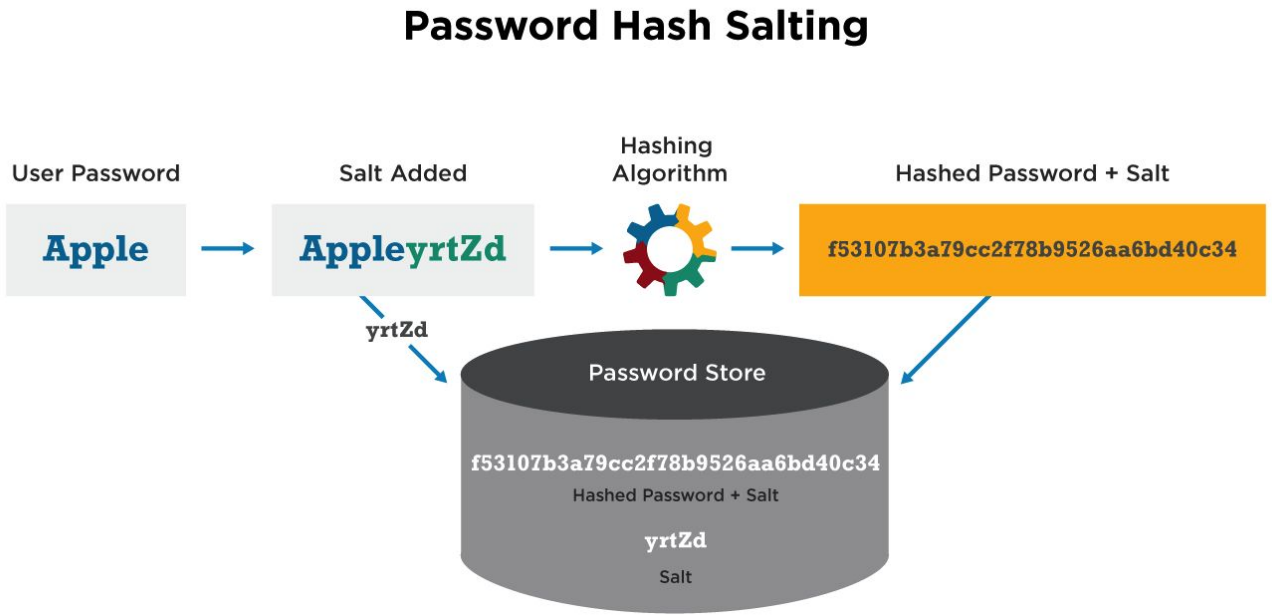


MD5 Hash Generator

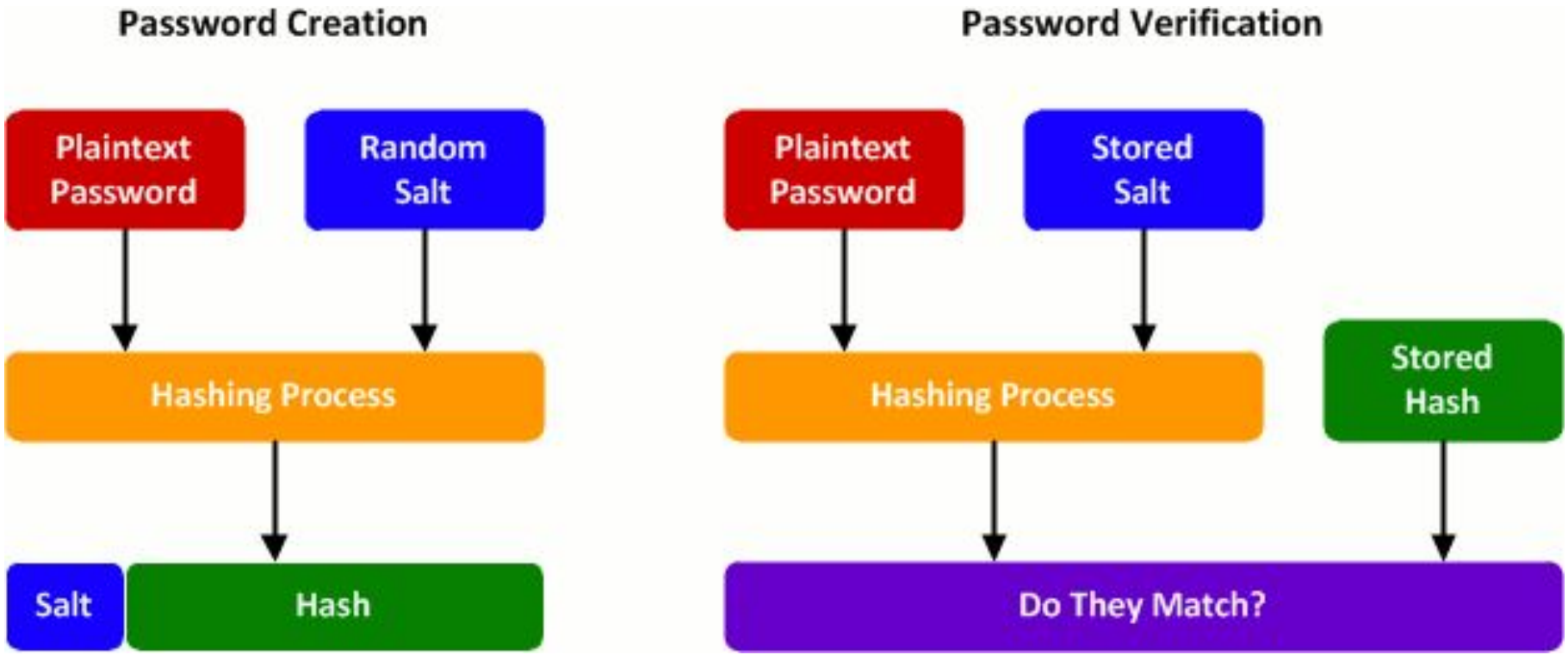
Salting

A Salt is a fixed-length cryptographically-strong random value that is added to a password as input to a hash function.

Dictionary attacks make passwords hashed with common algorithms vulnerable, salting reduces the effectiveness of dictionary attacks by making all input values for passwords unique.



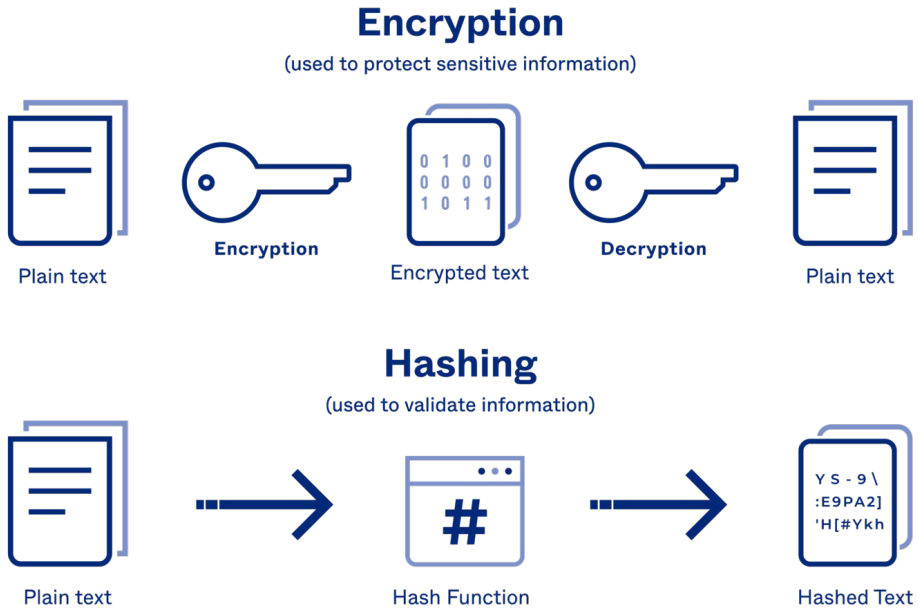
Password Salting



Encryption

Encryption is the most effective way to achieve data security. When data is sent between two parties or stored, it is stored in an encrypted non-human readable format that requires the key to properly decrypt and understand.

[OWASP Guide to Cryptography]



okta

Man In the Middle Attack

Performed by a local malicious network connection, for example, in a coffee shop or hotel.

1. Attacker provides a fake wifi connection
2. Victim connects and establishes a secure connection with the fake wifi connection.
3. The attacker establishes a secure connection on behalf of the victim to the intended destination.
4. Communication then transmits encrypted from the user to the attackers device and from the attackers device to the destination, but is unencrypted while on the attackers device.