Desenvolvimento de Sistemas

Segurança da informação: tratamento da SQL *injection*, parametrização de consulta

Introdução

Segurança da informação é um assunto recorrente em qualquer área da tecnologia, seja desenvolvimento de *software*, seja infraestrutura, computação em nuvem etc. Segundo uma pesquisa realizada pela FortiGuard Labs , no Brasil, apenas no primeiro semestre do ano de 2021 foram registradas mais de 16 bilhões de tentativas de ataques cibernéticos. Com essas estatísticas, é crucial a dedicação a fim de implementar medidas que tornem os ambientes mais seguros. Em se tratando de desenvolvimento de *softwares*, é possível implementar diversas boas práticas para evitar que o *software* que está sendo desenvolvido esteja exposto a certas ameaças.

Neste conteúdo, serão abordadas as principais práticas para tornar seus projetos Java mais seguros. Você também conhecerá algumas das vulnerabilidades a que poderá estar exposto e como poderá lidar com elas em diferentes cenários possíveis. Além de muitos conceitos, haverá diversas práticas no decorrer deste conteúdo, para que você exercite tudo o que aprendeu, e também dicas para você construir códigos mais seguros.

Autenticação de usuários

A autenticação de usuários é uma medida de segurança adotada em sistemas para garantir que a pessoa é realmente autorizada e cadastrada para o uso. Essa autenticação costuma acontecer por meio do processo de *login*, no qual o usuário usa suas credenciais cadastradas no sistema como identificação. Uma vez verificadas essas credenciais, obtém-se informações sobre a identidade e o acesso do usuário.

Para entender na prática como é feita a autenticação de usuários, será construída uma aplicação Java do zero, aplicando tudo o que se aprendeu sobre programação de aplicativos *desktop* com integração de banco de dados com ênfase nos aspectos de segurança.

Para melhor aproveitamento do conteúdo, você utilizará esse mesmo projeto para realizar outras práticas em outros tópicos. Então, sugere-se que você crie um novo projeto no seu computador para acompanhar os exemplos e realizar as práticas que serão apresentadas no decorrer do conteúdo.

Criação do projeto de autenticação

Para esse exemplo, crie um novo projeto selecionando a opção **Java with Ant** no Apache NetBeans IDE. Você também utilizará a biblioteca MySQL Connector para comunicação com o banco de dados. O projeto será chamado de **LoginExemplo** e terá a seguinte estrutura:

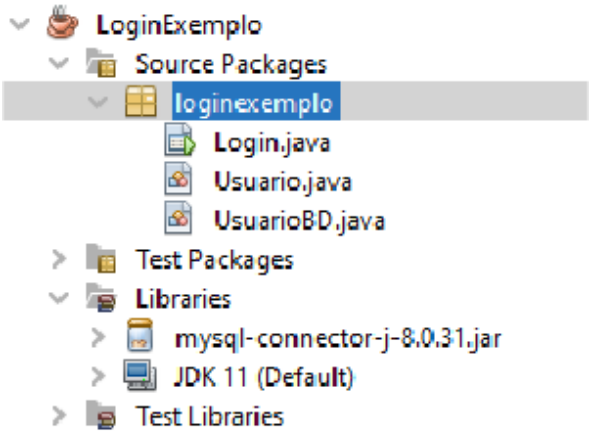


Figura 1 – Estrutura do projeto **LoginExemplo**

Fonte: Apache NetBeans IDE (2022)

Então, depois de criar o projeto, crie três arquivos dentro do pacote principal do projeto:

Usuario.java

Essa será a classe que permitirá criar objetos do tipo “Usuario”. Ela será composta pelos atributos “id”, “nome”, “login”, “senha” e “tipo” junto aos seus respectivos métodos **get** e **set**.

Crie esse arquivo usando a opção **New** > **Java Class** e cole o código a seguir:

```
public class Usuario {
  private int id;
  private String nome;
  private String login;
  private String senha;
  private String tipo;

  public int getId() {
      return id;
  }

  public void setId(int id) {
      this.id = id;
  }

  public String getNome() {
      return nome;
  }

  public void setNome(String nome) {
      this.nome = nome;
  }

  public String getLogin() {
      return login;
  }

  public void setLogin(String login) {
      this.login = login;
  }

  public String getSenha() {
      return senha;
  }

  public void setSenha(String senha) {
      this.senha = senha;
  }

  public String getTipo() {
      return tipo;
  }

  public void setTipo(String tipo) {
      this.tipo = tipo;
  }

}
```

UsuarioBD.java

Essa será a classe que você usará para que o seu sistema se comunique com o banco de dados MySQL. Ela será composta pelos métodos de manipulação de dados da tabela “Usuario”.

Crie esse arquivo usando a opção **New** > **Java Class** e cole este código:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class UsuarioBD {
    public static Usuario validarUsuarioSeguro(Usuario usuario) {
        // Criando consulta parametrizada
        String sql = "SELECT * FROM usuario WHERE login = ? AND senha
= ?";
        Usuario usuarioEncontrado = null;

        try {
            Connection conexao = DriverManager.getConnection("jdbc:my
sql://localhost:3306/login_exemplo", "root", "");
            PreparedStatement statement = conexao.prepareStatement(sq
l);

            // Atribuindo os valores na consulta
            statement.setString(1, usuario.getLogin());
            statement.setString(2, usuario.getSenha());
            ResultSet rs = statement.executeQuery();

            while (rs.next()) {
                usuarioEncontrado = new Usuario();
                usuarioEncontrado.setId(rs.getInt("id"));
                usuarioEncontrado.setNome(rs.getString("nome"));
                usuarioEncontrado.setLogin(rs.getString("login"));
                usuarioEncontrado.setSenha(rs.getString("senha"));
                usuarioEncontrado.setTipo(rs.getString("tipo"));
            }
        } catch (SQLException ex) {
            System.out.println("Sintaxe de comando invalida");
        }

        return usuarioEncontrado;
    }


    // MÉTODO INSEGURO!!!
    public static Usuario validarUsuarioInseguro(Usuario usuario) {
        String sql = "SELECT * FROM usuario WHERE login = "+ usuario.
getLogin() +" AND senha = " + usuario.getSenha();
        Usuario usuarioEncontrado = null;

        try {
            Connection conexao = DriverManager.getConnection("jdbc:my
sql://localhost:3306/login_exemplo", "root", "");
```

```
                        PreparedStatement statement = conexao.prepareStatement(sq
l);
                        ResultSet rs = statement.executeQuery();

                        while (rs.next()) {
                            usuarioEncontrado = new Usuario();
                            usuarioEncontrado.setId(rs.getInt("id"));
                            usuarioEncontrado.setNome(rs.getString("nome"));
                            usuarioEncontrado.setLogin(rs.getString("login"));
                            usuarioEncontrado.setSenha(rs.getString("senha"));
                            usuarioEncontrado.setTipo(rs.getString("tipo"));
                        }
                    } catch (SQLException ex) {
                        System.out.println("Sintaxe de comando invalida");
                    }

                    return usuarioEncontrado;
                }
            }
```

Login.java

Essa será a tela do seu sistema. Nela, haverá dois componentes para o usuário preencher com suas credenciais e um botão que, ao ser clicado, chamará os métodos da classe **UsuarioBD.java**.

Crie esse arquivo na opção **New** > **JFrame Form** com o nome **Login.java** e, utilizando o construtor de interface gráfica do Apache NetBeans IDE, monte a seguinte tela:

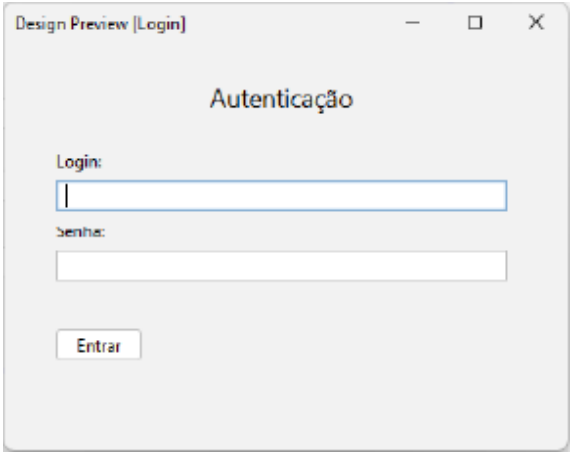


Figura 2 – Tela de autenticação

Fonte: Senac EAD (2022)

Lembre-se de que essa tela é apenas uma sugestão de *layout* para você ter como base. Caso deseje construir um *layout* diferente, fique à vontade. O importante é que o *layout* tenha dois componentes **JTextField** para informar os dados de autenticação (*login* e senha) e um componente **JButton** para acionar o método que validará o usuário.

Após concluir a construção da tela, clique com o botão direito do *mouse* sobre o componente **JButton** e selecione a opção **Events** > **Action** > **actionPerformed** para criar um evento quando o botão *for* clicado. Você será levado à aba **Source** do GUI Builder e terá o método **jButton1ActionPerformed()** focado. Lembre-se desse método, pois você voltará nele várias vezes. Por enquanto, ele pode ficar em branco.

Banco de dados

Como sua aplicação consultará no banco de dados as informações fornecidas pelo usuário, é essencial que se tenha uma base de dados construída para que essa consulta seja realizada.

Para esse exemplo, você construirá a base de dados **login_exemplo** e, dentro dela, criará a tabela “usuario”, que conterá os dados para validar a autenticação do sistema.

Então, execute no SGBD (Sistema de Gerenciamento de Banco de Dados) os seguintes comandos:

```
CREATE DATABASE login_exemplo;
USE login_exemplo;

CREATE TABLE usuario (
    id int PRIMARY KEY AUTO_INCREMENT,
    nome varchar(30),
    login varchar(100),
    senha text,
    tipo varchar(30)
);
```

Com a tabela criada, é possível começar a inserção de alguns dados. Considerando as práticas que serão abordadas nós próximos tópicos, alguns registros foram separados para você inserir na sua tabela. São eles:

```
INSERT INTO usuario (nome, login, senha) VALUES
('Teste 1', '123', '123'),
('Teste 2', '1234', '81dc9bdb52d04dc20036dbd8313ed055');
```

Um detalhe importante é que você nunca deve armazenar ou exibir senhas como texto simples, pois podem ser facilmente recuperadas por partes mal-intencionadas. Uma maneira típica de se evitar isso é usando um algoritmo de *hash* unidirecional para armazená-lo no banco de dados, como MD5 ou SHA1. Portanto, nesse exemplo, criou-se a coluna “senha” como “text” para que não se tenha nenhum problema de limite de caracteres ao registrar uma senha criptografada com um algoritmo de *hash*.

SQL injection

As injeções de SQL, também conhecidas como SQLi, acontecem quando um invasor altera com sucesso a entrada de um aplicativo da *web*, obtendo a capacidade de executar consultas SQL arbitrárias nesse aplicativo.

Em palavras simples, SQL *injection* significa injetar/inserir código SQL (*structured query language*) em uma consulta por meio de dados inseridos pelo usuário. É um ataque que pode ocorrer em qualquer aplicação que utilize bancos de dados relacionais, como Oracle, MySQL, PostgreSQL e SQL Server.

Um ataque de injeção de SQL bem-sucedido permite que seu autor altere dados no banco de dados de destino. Se o aplicativo de destino usar uma conexão de *string* de banco de dados, cujo usuário tenha privilégios de gravação no banco de dados, o ataque SQLi poderá causar danos devastadores. O invasor pode excluir grandes quantidades de dados ou até mesmo descartar as próprias tabelas.

Alternativamente, o invasor pode obter apenas acesso de leitura ao banco de dados. Se isso não soa tão ruim quanto o cenário anterior, pense novamente. A invasão que permite ao invasor obter acesso a informações não autorizadas seria uma situação extremamente problemática. Isso pode levar à exposição de dados confidenciais, incluindo não apenas informações confidenciais de negócios, mas também informações financeiras ou segredos industriais, além de informações pessoais sobre clientes.

Os ataques de injeção funcionam porque, para muitos aplicativos, a única maneira de executar uma determinada tarefa (por exemplo, buscar uma lista de produtos) é gerar dinamicamente um código que, por sua vez, é executado por outro sistema ou componente (no caso, um banco de dados). Usar dados não confiáveis sem a devida validação no processo de geração de código é deixar uma porta aberta para ser explorada.

Essa afirmação pode soar um pouco abstrata, então veja como isso acontece na prática em um exemplo com JDBC (Java Database Connectivity):

```
                public static Usuario buscarUsuarioPorId(String idUsuario) throws
SQLException {
                  // INSEGURO!!! NÃO FAÇA ISSO!!!
                  String sql = "SELECT * FROM usuario WHERE id = " + idUsuario;
                  Connection conexao = DriverManager.getConnection("jdbc:mysql://
localhost:3306/login_exemplo", "root", "");
                  PreparedStatement statement = conexao.prepareStatement(sql);
                  ResultSet rs = statement.executeQuery();
              }
```

O problema aqui é que se está realizando a concatenação com um valor fornecido pelo usuário sem qualquer validação. Para executar uma SQL *injection*, um usuário mal-intencionado primeiro tenta encontrar um local na aplicação onde possa incorporar o código SQL junto aos dados – geralmente, a partir de componentes de entrada de dados. Assim, quando os dados incorporados com código SQL forem recebidos pelo aplicativo, o código SQL será executado junto à consulta do aplicativo. Infelizmente, *frameworks* e ORMs (*object relational mapper*) não são suficientes para proteger uma linguagem de ataques de SQL *injection*.

Tipos de SQL *injection*

Há uma grande variedade de vulnerabilidades, ataques e técnicas de SQL *injection* que surge em diferentes situações. Para apresentar de forma didática algumas dessas principais técnicas, serão realizadas algumas práticas com o sistema criado. Observe com atenção cada proposta para entender de fato onde se encontra a vulnerabilidade que se quer explorar e como se pode corrigi-la.

Para realizar as práticas a seguir no projeto Java que você criou, abra o arquivo **Login.java** e cole o seguinte trecho de código dentro do método **jButtonActionPerformed()**:

```
                private void jButton1ActionPerformed(java.awt.event.ActionEvent e
vt) {
                    // TODO add your handling code here:

                    // Criamos um objeto do tipo Usuario
                    Usuario usuario = new Usuario();

                    // Atribuimos os valores Login e Senha baseado nos dados dos co
mponentes JTextField
                    usuario.setLogin(jTextField1.getText());
                    usuario.setSenha(jTextField2.getText());

                    // Exemplo de SQL Injection
                    usuario = UsuarioBD.validarUsuarioInseguro(usuario);


                    // "Reescrevemos" os valores do objeto baseado na resposta do m
étodo
                    // Se nenhum registro for encontrado, teremos um usuário NULO
                    if (usuario != null) {
                        JOptionPane.showMessageDialog(null, "Você foi autenticado c
om sucesso!");
                    } else {
                        JOptionPane.showMessageDialog(null, "Erro de autenticação!
Verifique se os dados estão corretos.");
                    }
                }
```

SQL *injection* baseado em *boolean*

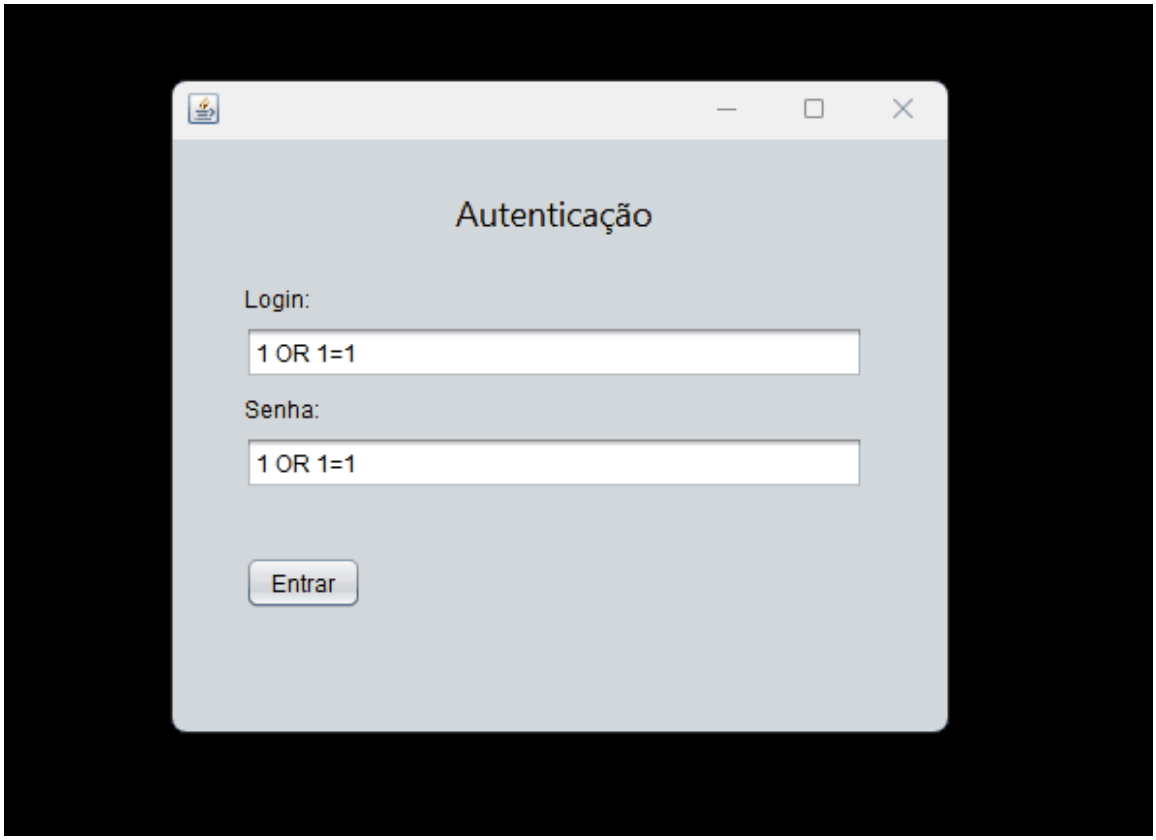
Nesse cenário, a SQL *injection* é explorada por meio da lógica booleana *true* (verdadeiro) ou *false* (falso). A forma mais simples para realizá-la é adicionando um **OR 1=1** na verificação **WHERE** da consulta do banco de dados. A comparação “1=1” é sempre verdadeira. Se o operador lógico OR for combinado na instrução SQL, significa que basta uma condição ser verdadeira para a consulta ser executada.

Após isso, execute o projeto e preencha o formulário de *login* com os seguintes dados:

- Login: **1 OR 1=1**
- Senha: **1 OR 1=1**

Observe a seguir a consulta que será executada no banco de dados e o GIF, para compreender melhor:

```
SELECT * FROM usuario WHERE login = 1 OR 1=1 AND senha = 1 OR 1=1
```



Perceba que, independentemente de a tabela ter um registro de **id** com valor 1 ou não, a consulta é realizada com sucesso por conta do **OR 1=1**.

Essa exploração costuma ser o ponto de início de um ataque de SQL *injection*. Se bem-sucedido, o invasor pode ter acesso a outras áreas do sistema, nas quais poderá realizar outras tentativas de manipulação de dados. Também é comum que essa injeção de código seja acompanhada de comandos de manipulação de dados (como os comandos **DELETE** e **DROP**, por exemplo), os quais serão executados logo em seguida.

SQL *injection* baseada em tempo

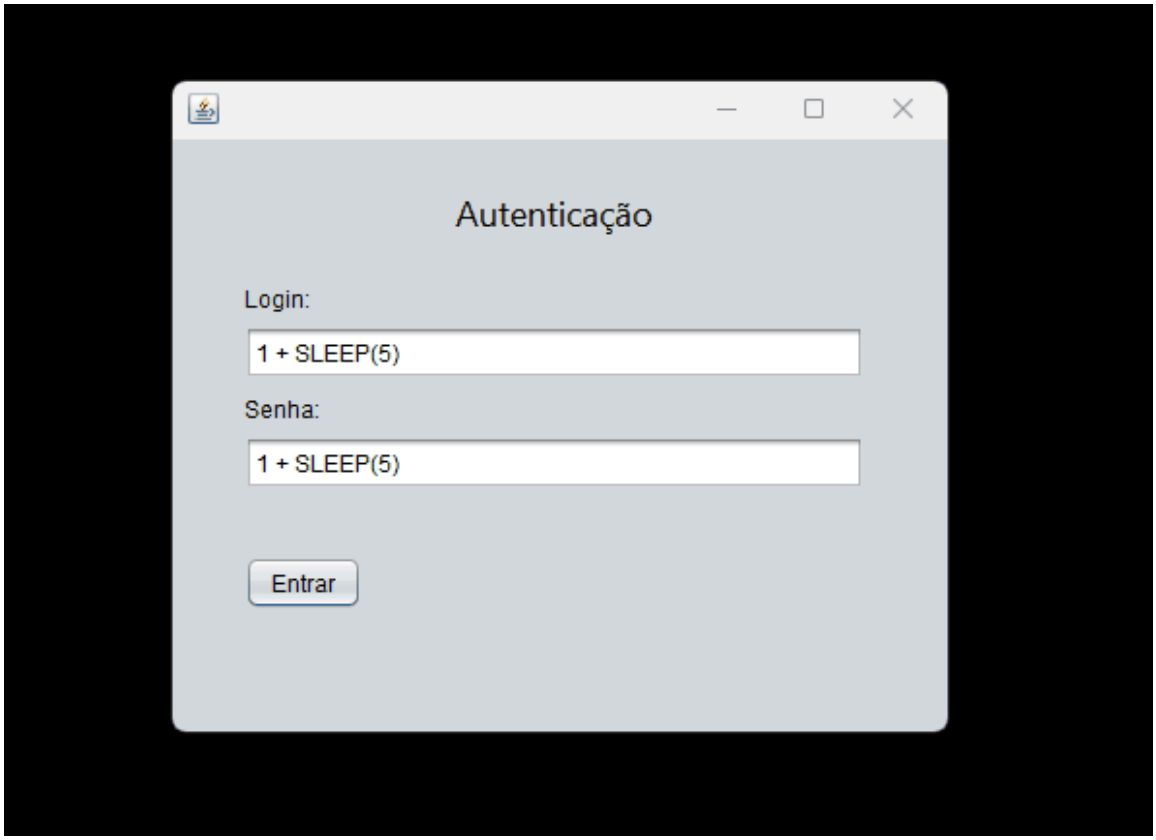
Em injeções SQL baseadas em tempo, funções especiais são injetadas na consulta e podem pausar a execução por um período de tempo específico. Esse ataque **diminui** a velocidade do servidor de banco de dados e pode derrubar sistemas afetando o desempenho desse servidor.

Para essa prática, execute o projeto e preencha o formulário de *login* com os seguintes dados:

- Login: **1 + SLEEP(5)**
- Senha: **1 + SLEEP(5)**

Confira a seguir a consulta que será executada no banco de dados e o GIF, para compreender melhor:

```
SELECT * FROM usuario WHERE login=1 + SLEEP(5) AND senha=1 + SLEEP(5)
```



Perceba que agora, antes de executar a consulta SQL, o sistema ficará “congelado”. Isso ocorre porque a consulta só será executada depois de se passarem dez segundos (cinco segundos do campo “login” e cinco segundos do campo “senha”), nos quais o sistema ficará aguardando uma resposta do servidor SQL.

SQL *injection* baseada em erro

Nessa variação, o invasor tenta obter informações com um código de erro e uma mensagem do banco de dados. O invasor injeta SQL que está sintaticamente incorreta para que o servidor de banco de dados retorne o código de erro e as mensagens que podem ser usadas para obter informações do banco de dados e do sistema.

Um exemplo simples em que esse tipo de situação pode ocorrer é quando se apresenta para o usuário explicitamente o conteúdo das exceções que são geradas na aplicação. Observe a imagem:

Figura 3 – Aplicação "Login Exemplo"

Fonte: Senac EAD (2022)

Para reproduzir esse exemplo, execute o projeto e clique no botão **Entrar** sem preencher nenhum campo. Uma nova janela se abrirá com esta mensagem:

Erro: java.sql.SQLSyntaxErrorException: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '' at line 1

Nesse exemplo, as informações podem parecer inofensivas, mas, em uma situação real, uma pessoa mal-intencionada já saberia a linguagem de programação utilizada para construir o sistema (Java), o banco de dados utilizado

(que, nesse exemplo, seria MariaDB) e em qual linha da instrução SQL ocorre a primeira manipulação de dados (nesse caso, na linha 1). Essas são informações suficientes para um invasor aplicar outras técnicas de exploração de vulnerabilidades.

Técnicas de prevenção

Apesar de os ataques de injeção de SQL serem comuns e potencialmente devastadores, eles são evitáveis. As vulnerabilidades que os ataques de injeção de SQL exploram se originam de erros de codificação. Portanto, aprender a evitar esses erros é sua primeira e mais importante linha de defesa contra esse tipo de ataque. Agora que você já sabe o que é uma injeção de SQL, veja como pode proteger seu código desse tipo de ataque. Neste momento, o foco são algumas técnicas muito eficazes disponíveis em Java e SQL.

Para quem procura uma lista completa de técnicas disponíveis, incluindo as específicas de banco de dados, o Projeto OWASP (Open Web Application Security Project) mantém uma lista de dicas de prevenção de injeção de SQL, conhecido como *SQL Injection Prevention Cheat Sheet* (ou, em tradução livre, “Folha de Dicas de Prevenção de Injeção SQL”), que é um ótimo recurso para se aprender mais sobre o assunto.

Parametrização de consulta

Um **PreparedStatement** é uma instrução SQL pré-compilada. É uma subinterface de **Statement**, na qual os objetos preparados têm alguns recursos adicionais úteis. Em vez de consultas codificadas, o objeto **PreparedStatement** fornece um recurso para executar uma consulta parametrizada.

Quando o **PreparedStatement** é criado, a consulta SQL é passada como parâmetro. Essa instrução preparada contém uma consulta SQL pré-compilada, portanto, assim que o **PreparedStatement** é executado, o SGBD pode simplesmente executar a consulta. Em outras palavras, primeiro a consulta é compilada e, se estiver tudo correto, então é executada no SGBD.

A técnica de parametrização de consultas consiste em usar declarações preparadas com o marcador de posição de ponto de interrogação (“**?**”) nas consultas sempre que for necessário inserir um valor fornecido pelo usuário. Isso é muito eficaz e, a menos que haja um *bug* na implementação do *driver* JDBC, imune a explorações.

Voltando para ao seu projeto, abrindo o arquivo **UsuarioBD.java**, você encontrará o seguinte trecho de código aplicando o uso da parametrização de consultas:

```
                public static Usuario validarUsuarioSeguro(Usuario usuario) {
                  // Criando consulta parametrizada
                  String sql = "SELECT * FROM usuario WHERE login = ? AND senha =
?";
                  Usuario usuarioEncontrado = null;

                  Connection conexao = DriverManager.getConnection("jdbc:mysql://
localhost:3306/login_exemplo", "root", "");
                  PreparedStatement statement = conexao.prepareStatement(sql);

                  // Atribuindo os valores na consulta
                  statement.setString(1, usuario.getLogin());
                  statement.setString(2, usuario.getSenha());
                  ResultSet rs = statement.executeQuery();

                  ...
              }
```

Aqui, utiliza-se o método **prepareStatement()** disponível na instância **Connection** para obter um **PreparedStatement**. Essa interface estende a interface **Statement** com vários métodos que permitem que você insira com segurança valores

fornecidos pelo usuário em uma consulta antes de executá-la. Agora, abra o arquivo **Login.java** e cole o seguinte trecho de código dentro do método **jButtonActionPerformed()**:

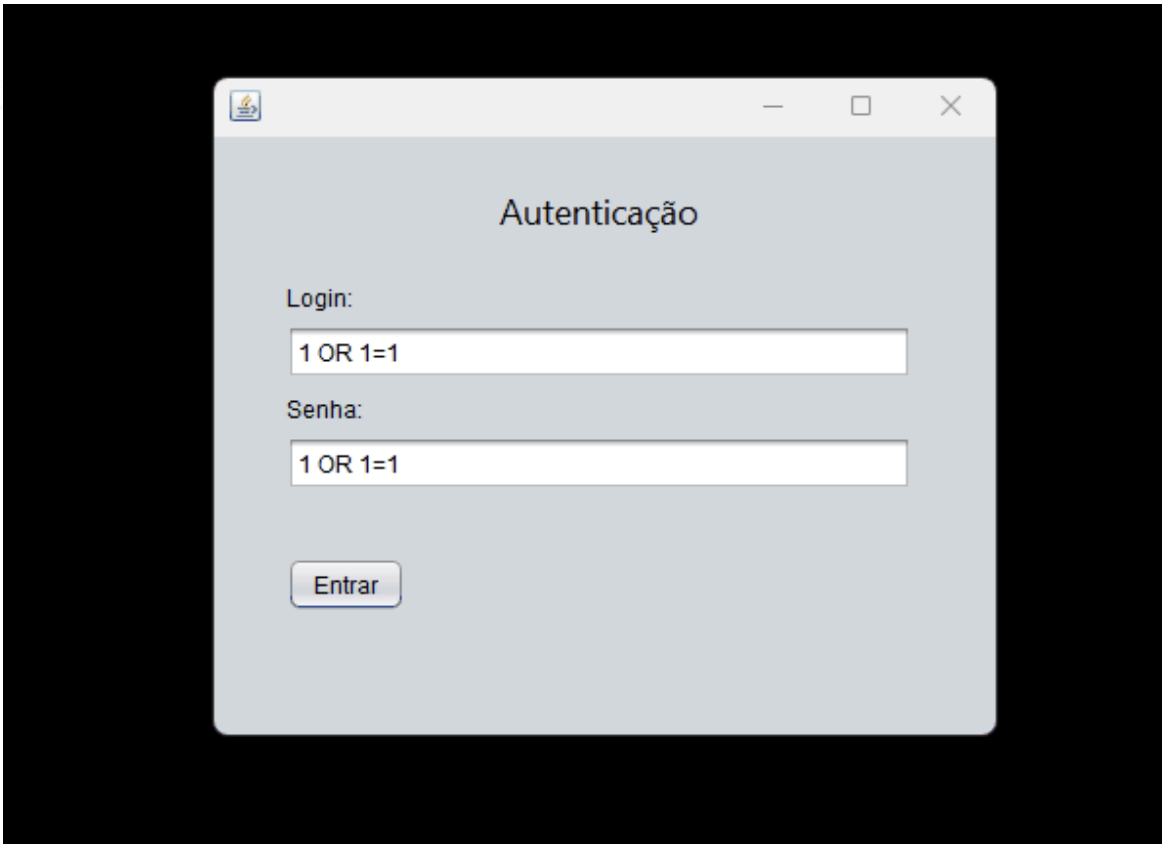
```
                private void jButton1ActionPerformed(java.awt.event.ActionEvent ev
t) {
                  // TODO add your handling code here:

                  // Criamos um objeto do tipo Usuario
                  Usuario usuario = new Usuario();

                  // Atribuimos os valores Login e Senha baseado nos dados dos comp
onentes JTextField
                  usuario.setLogin(jTextField1.getText());
                  usuario.setSenha(jTextField2.getText());

                  // Usando a validação segura
                  usuario = UsuarioBD.validarUsuarioSeguro(usuario);
                  // "Reescrevemos" os valores do objeto baseado na resposta do mét
odo
                  // Se nenhum registro for encontrado, teremos um usuário NULO
                  if (usuario != null) {
                      JOptionPane.showMessageDialog(null, "Você foi autenticado com
sucesso!");
                  } else {
                      JOptionPane.showMessageDialog(null, "Erro de autenticação! Ve
rifique se os dados estão corretos.");
                  }
              }
```

Agora, se você tentar aplicar alguma das técnicas de SQL *injection* que aprendeu, verá que a prática não será bem-sucedida como antes devido à parametrização de consultas. Confira no GIF:



Para JPA (Java Persistence API), existe um recurso semelhante:

```
String jql = "SELECT u FROM Usuario u WHERE u.login = :login AND u.senha = :senha";
// Executar consulta e retornar resultados mapeados (omitido)
```

Nesse caso, a camada ORM cria um **Prepared Statement** usando um espaço reservado para os parâmetros ***login*** e ***senha***. Isso é o mesmo que foi feito no caso da JDBC, mas com algumas instruções a menos.

Como extra, essa abordagem geralmente resulta em uma consulta com melhor desempenho, pois a maioria dos bancos de dados pode armazenar em *cache* o plano de consulta associado a uma instrução preparada.

Procedimentos armazenados (*stored procedures*)

Os procedimentos armazenados nem sempre estão protegidos contra injeção de SQL. No entanto, certas construções de programação de procedimento armazenado padrão têm o mesmo efeito que o uso de consultas parametrizadas quando implementadas com segurança, o que é a norma para a maioria das linguagens de procedimento armazenado.

Esses procedimentos armazenados exigem que o desenvolvedor apenas construa instruções SQL com parâmetros que são parametrizados automaticamente, a menos que o desenvolvedor faça algo muito fora da norma. A diferença entre instruções preparadas e procedimentos armazenados é que o código SQL para um procedimento armazenado é definido e armazenado no próprio banco de dados e, em seguida, chamado no aplicativo. Ambas as técnicas têm a mesma eficácia na prevenção da injeção de SQL, portanto, sua organização deve escolher qual abordagem faz mais sentido para você.

“Implementado com segurança” significa que o procedimento armazenado não inclui nenhuma geração de SQL dinâmica insegura. Os desenvolvedores geralmente não geram SQL dinâmico dentro de procedimentos armazenados. No entanto, isso pode ser feito, mas deve ser evitado. Se não puder ser evitado, o procedimento armazenado deve usar validação de entrada ou escape adequado para garantir que nenhuma das entradas fornecidas pelo usuário para o procedimento armazenado possam ser usadas para injetar código SQL na consulta gerada dinamicamente.

Validação de entrada de dados da lista de permissões

Alguns trechos das consultas SQL não são locais válidos para o uso de variáveis de associação, como os nomes de tabelas ou colunas e o indicador de ordem de classificação (ASC ou DESC). Em tais situações, a validação de entrada ou o redesenho da consulta é a defesa mais adequada. Para os nomes de tabelas ou colunas, o ideal é que esses valores venham do código e não dos parâmetros do usuário.

Porém, se os valores dos parâmetros do usuário forem usados para direcionar nomes de tabelas e nomes de colunas diferentes, os valores dos parâmetros deverão ser mapeados para os nomes de colunas ou tabelas legais (esperados) para garantir que a entrada não validada do usuário não acabe na consulta. Observe que isso é um sintoma de *design* ruim e uma reescrita completa deve ser considerada, se o tempo permitir.

Confira um exemplo de validação de nome de tabela:

```
String nomeTabela;
switch(PARAMETRO):
case "Valor1": nomeTabela = "produto";
             break;
case "Valor2": nomeTabela = "pedido";
             break;
...
default      : throw new InputValidationException("valor inválido.");
```

O **nomeTabela** pode ser anexado diretamente à consulta SQL, pois agora é conhecido como um dos valores legais e esperados para um nome de tabela nessa consulta. Lembre-se de que as funções genéricas de validação de tabela podem levar à perda de dados, porque os nomes das tabelas são usados em consultas em que não são esperados.

Para algo simples, como uma ordem de classificação, seria melhor se a entrada fornecida pelo usuário fosse convertida em um booleano e, em seguida, esse booleano fosse usado para selecionar o valor seguro a ser anexado à consulta. Essa é uma necessidade padrão na criação de consultas dinâmicas.

Veja o exemplo:

```
public String metodoQualquer(boolean ordemDeClassificacao) {
  String query = "consulta SQL ... ORDER BY nome " + (ordemDeClassificacao ? "ASC"
: "DESC");`
  ...
```

Sempre que a entrada do usuário puder ser convertida em uma não *string*, como data, numérica, booleana, tipo enumerado, etc. antes de ser anexada a uma consulta ou usada para selecionar um valor para anexar à consulta, isso garante que é seguro fazê-lo.

A validação de entrada também é recomendada como defesa secundária em **todos** os casos, mesmo ao usar variáveis de associação.

Práticas recomendadas para evitar a SQL *injection*

Como uma boa prática de segurança, você deve sempre implementar várias camadas de defesa – um conceito conhecido como **defesa em profundidade**. A ideia principal é que, mesmo que você não consiga encontrar todas as vulnerabilidades possíveis em seu código – um cenário comum ao lidar com sistemas legados –, deverá pelo menos tentar limitar os danos que um ataque infligiria. As principais práticas de segurança são as seguintes:

Aplicar o princípio do privilégio mínimo

Restrinja o máximo possível os privilégios da conta usada para acessar o banco de dados. Durante o conteúdo, foi utilizado o usuário *root* em todos os exemplos.

Usar credenciais de curta duração

Faça com que o aplicativo alterne as credenciais do banco de dados com frequência.

Registrar tudo

Se o seu sistema armazena dados de clientes, isso é obrigatório! Existem muitas soluções disponíveis que se integram diretamente ao banco de dados ou funcionam como *proxy*, portanto, em caso de ataque, é possível pelo menos avaliar os danos.

Usar *firewalls* de aplicativos *web* (WAFs) ou soluções semelhantes de detecção de intrusão

Esses são os exemplos típicos de lista negra (*black list*) – geralmente, eles vêm com um banco de dados considerável de assinaturas de ataques conhecidos e acionarão uma ação programável após a detecção. Alguns também incluem agentes na JVM (Java Virtual Machine), que podem detectar intrusões aplicando alguma instrumentação. A principal vantagem dessa abordagem é que uma eventual vulnerabilidade se torna muito mais fácil de corrigir, pois haverá um rastreamento de pilha completo disponível.

Outras práticas importantes que devem ser citadas são:

1. Valide os dados antes de usá-los na consulta.
2. Não use palavras comuns como nome de tabela ou nome de coluna. Por exemplo, muitos aplicativos usam “usuário” ou “senha” para armazenar dados de *login* e esses nomes são considerados comuns.
3. Não concatene dados diretamente (recebidos como entrada do usuário) para criar consultas SQL.
4. Use estruturas como **Hibernate** e **Spring Data JPA** para a camada de dados de um aplicativo.
5. Use parâmetros posicionais na consulta. Se você estiver usando JDBC, use **PreparedStatement** para executar a consulta.
6. Limite o acesso do aplicativo ao banco de dados por meio de permissões e concessões.
7. Não devolva códigos de erro e mensagens confidenciais ao usuário final.
8. Faça uma revisão de código adequada, para que nenhum desenvolvedor escreva acidentalmente um código SQL inseguro.

Segurança em projetos Java

Além das técnicas de prevenção à SQL *injection*, existem outras boas práticas que você pode aplicar no desenvolvimento de projetos em Java para tornar suas aplicações mais seguras.

Hash MD5

Nunca se deve transmitir senhas de texto simples por um canal não criptografado, pois elas podem ser interceptadas por usuários não autorizados por meio de *sniffers* – *softwares* de monitoramento de tráfego de uma rede que captura dados dos computadores conectados. Considere como exemplo o sistema de *login* que você está construindo: quando o usuário clica no botão **Entrar**, uma solicitação é enviada para o servidor de banco de dados pela rede em forma de pacotes. É nesse momento que os pacotes podem ser "pegos" por um *sniffer* e os dados, caso não estejam criptografados, podem ser sequestrados. Sendo assim, é preciso garantir que as informações mais sensíveis do usuário estejam ilegíveis para o caso de um invasor recuperar essas informações. Por isso, você aplicará a *hash* MD5 neste projeto.

Crie um novo arquivo chamado **Criptografia.java** usando a opção **New** > **Java Class** e cole o seguinte código:

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class Criptografia {
    public static String getMD5(String texto) {
        try {

            // O método estático getInstance é
chamado com hash MD5
            MessageDigest md = MessageDigest.ge
tInstance("MD5");

            // O método digest() é chamado para
calcular a hash da mensagem
            // E enfim temos o vetor da mensage
m
            byte[] messageDigest = md.digest(te
xto.getBytes());

            // Convertemos o vetor de bytes em
um BigInt
            BigInteger no = new BigInteger(1, m
essageDigest);

            // A mensagem em BigInt é convertid
a para hexadecimal
            String hashtext = no.toString(16);
            while (hashtext.length() < 32) {
                hashtext = "0" + hashtext;
            }
            return hashtext;
        }

        // Em caso de erro, é lançada uma exceç
ão
        catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Em Java, utiliza-se a classe **MessageDigest** para calcular o valor *hash* de um texto. A classe fornece as funções de *hash* em MD5, SHA-1 e SHA-256.

Esses algoritmos são inicializados no método estático chamado **getInstance()**. Depois de selecionar o algoritmo, ele calcula o valor e retorna os resultados na matriz de *bytes*.

Agora que o método para gerar a hash MD5 está pronto, é possível fazer a chamada dele no arquivo para criptografar a senha do seu usuário no arquivo **Login.java**:

```
                            private void jButton1ActionPerformed(java.awt.e
vent.ActionEvent evt) {
                              // TODO add your handling code here:

                              // Criamos um objeto do tipo Usuario
                              Usuario usuario = new Usuario();

                              // Atribuimos os valores Login e Senha basead
o nos dados dos componentes JTextField
                              usuario.setLogin(jTextField1.getText());
                              usuario.setSenha(Criptografia.getMD5(jTextFie
ld2.getText()));

                              System.out.println("A senha em MD5 é: " + usu
ario.getSenha());

                              // Exemplo de SQL Injection baseada em boolea
no
                              usuario = UsuarioBD.validarUsuarioInseguro(us
uario);


                              // "Reescrevemos" os valores do objeto basead
o na resposta do método
                              // Se nenhum registro for encontrado, teremos
um usuário NULO
                              if (usuario != null) {
                                  JOptionPane.showMessageDialog(null, "Você
foi autenticado com sucesso!");
                              } else {
                                  JOptionPane.showMessageDialog(null, "Erro
de autenticação! Verifique se os dados estão corretos.");
                              }
                          }
```

Apenas para fins didáticos, foi inserido um **System.out.println()** para imprimir no terminal como ficou a *hash* da senha do usuário e para você conferir se está tudo funcionando. Confira no GIF:

É importante lembrar que agora há um valor criptografado sendo passado para a consulta do banco de dados. Então, se você digitar a senha **1234**, o valor que será atribuído à variável “senha” será **81dc9bdb52d04dc20036dbd8313ed055**. Portanto, é essencial que a senha do seu usuário também esteja em MD5 lá nos registros da tabela do banco de dados.

Privilégio de usuários

Definir privilégios para os usuários significa limitar o acesso a determinados recursos do sistema com base no tipo de usuário que está tentando acessá-lo.

Geralmente, o usuário “admin” é quem tem o acesso total ao sistema, enquanto outros usuários têm acessos limitados. Porém, essa abordagem pode significar uma brecha de segurança para um invasor explorar, visto que a credencial “admin” já é bastante conhecida e isso limitaria o ataque a apenas um alvo no sistema. Portanto, não é recomendado ter um usuário “admin” cadastrado, mas sim criar privilégios baseados no tipo de usuário.

O processo é bem simples e é uma ótima camada de proteção para o sistema Java. Primeiramente, deve haver uma coluna no banco de dados para se definir o tipo de cada usuário registrado. Como a tabela já foi criada com essa estrutura no começo desse conteúdo, basta popular a tabela de usuário com alguns registros:

```
                            INSERT INTO usuario (nome, login, senha, tipo)
VALUES
                            ('Lucas', 'lucas@email.com', md5('123'), 'Admi
n'),
                            ('Laura', 'laura@email.com', md5('123'), 'De
v'),
                            ('Luis', 'luis@email.com', md5('123'), 'Venda
s');
```

Agora, existem as seguintes credenciais para o acesso ao sistema:

Nome	Login	Senha	Tipo
Lucas	lucas@email.com	123	Admin
Laura	laura@email.com	123	Dev
Luis	luis@email.com	123	Vendas

Perceba que foram definidos três tipos diferentes de usuários: “admin”, “dev” e “vendas”.

Agora, em sua aplicação Java, considere mais uma vez o arquivo **Login.java** e atualize o método **jButton1ActionPerformed()** com o seguinte código:

```
                            private void jButton1ActionPerformed(java.awt.e
vent.ActionEvent evt) {
                                // TODO add your handling code here:

                                // Criamos um objeto do tipo Usuario
                                Usuario usuario = new Usuario();

                                // Atribuimos os valores Login e Senha basead
o nos dados dos componentes JTextField
                                usuario.setLogin(jTextField1.getText());
                                usuario.setSenha(Criptografia.getMD5(jTextFie
ld2.getText()));

                                // Exemplo de SQL Injection baseada em boolea
no
                                usuario = UsuarioBD.validarUsuarioSeguro(usua
rio);



                                // "Reescrevemos" os valores do objeto basead
o na resposta do método
                                // Se nenhum registro for encontrado, teremos
um usuário NULO
                                if (usuario != null) {
                                    // Dependendo do tipo de usuário, levamos
a uma página diferente
                                    if( usuario.getTipo().equalsIgnoreCase("A
dmin") ) {
                                        JOptionPane.showMessageDialog(null,
"Bem-vindo, Administrador " + usuario.getNome() + "");
                                    } else if ( usuario.getTipo().equalsIgnor
eCase("Dev") ) {
                                        JOptionPane.showMessageDialog(null,
"Olá, mundo! \nQuer dizer...\nOlá, " + usuario.getNome());
                                    } else {
                                        JOptionPane.showMessageDialog(null,
"Ops, " + usuario.getNome() + "\nParece que você não tem permissão acessar
o sistema :( ");
                                    }

                                } else {
                                    JOptionPane.showMessageDialog(null, "Erro
de autenticação! Verifique se os dados estão corretos.");
                                }
                        }
```

Se você testar a aplicação agora, perceberá que para cada usuário há uma mensagem diferente sendo apresentada.

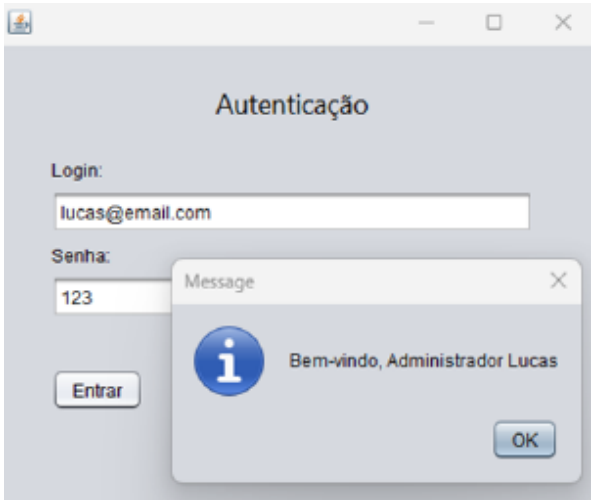


Figura 4 – Aplicação "Login Exemplo"

Fonte: Senac EAD (2022)

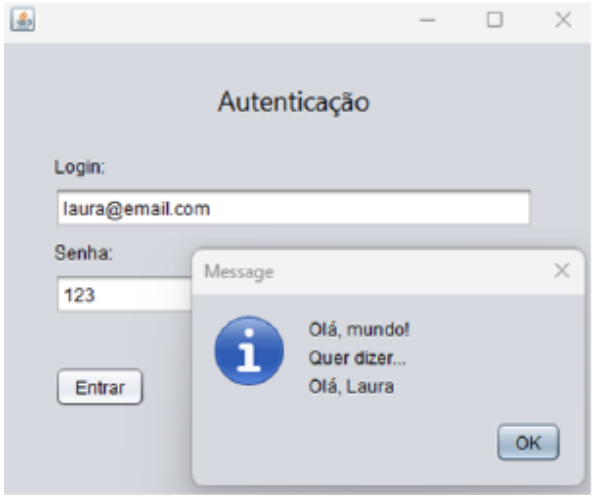


Figura 5 – Aplicação "Login Exemplo"

Fonte: Senac EAD (2022)

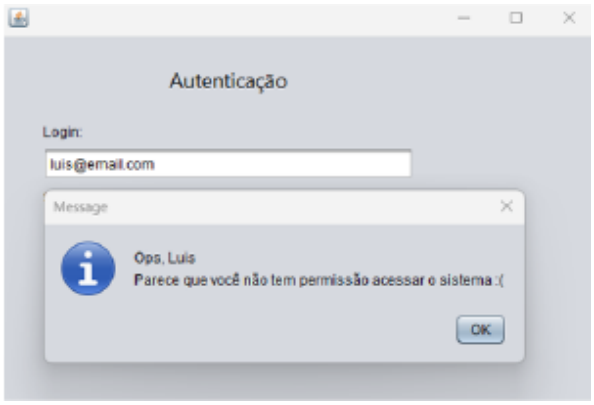


Figura 6 – Aplicação "Login Exemplo"

Fonte: Senac EAD (2022)

Para colocar em prática o uso dos privilégios de usuários em Java, crie duas novas telas usando o **JFrame**. Você pode personalizá-la como quiser, mas é importante que uma tela seja diferente da outra.

Depois isso, verifique o tipo do usuário e aplique a seguinte regra de negócio:

• Se o tipo de usuário for “Admin” ou “Dev”, abra a primeira tela criada.

• Se o tipo de usuário for “Vendas”, abra a segunda tela criada.

• Se o tipo de usuário for qualquer outro, exiba uma mensagem de erro.

Encerramento

As injeções de SQL não são ataques incrivelmente sofisticados. No entanto, como nesses ataques os invasores injetam códigos SQL maliciosos, essas injeções podem causar danos catastróficos.

Neste conteúdo, foram abordadas algumas técnicas de prevenção que você pode usar para evitar ser vítima de injeções de SQL em seus aplicativos Java. Só porque são ataques comuns, isso não significa que sejam inevitáveis. Além disso, foram estudadas as principais técnicas e boas práticas para que você as aplique e torne seu sistema mais seguro. Com a combinação certa de melhores práticas e ferramentas, você certamente poderá proteger seus aplicativos Java contra ataques de injeção de SQL e outras ameaças de segurança.