FUNDAÇÃO EDUCACIONAL DE FERNANDÓPOLIS
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

IMPLEMENTAÇÃO DE SEGURANÇA CONTRA SQL INJECTION EM APLICAÇÕES WEB

SECURITY IMPLEMENTATION AGAINST SQL INJECTION IN WEB APPLICATIONS

Ana Carolina de Souza Carvalho[1], João Gabriel Monteiro Cochut[2], Guilherme Morais[3]

[1]Fundação Educacional de Fernandópolis, anacarvalho@fef.edu.br
[2] Fundação Educacional de Fernandópolis, joaocochut@fef.edu.br
[3] Fundação Educacional de Fernandópolis, guilherme@fef.edu.br

Área: Segurança da Informação
Subárea: Segurança em Aplicações Web

RESUMO

O trabalho apresenta uma análise aprofundada sobre a implementação de mecanismos de segurança para prevenir ataques de injeção SQL em aplicações web, destacando o impacto dessa vulnerabilidade na integridade e segurança dos dados. Abordou-se a linguagem SQL, seus principais comandos e como falhas de segurança nesse contexto podem ser exploradas por atacantes para manipular o banco de dados. A pesquisa foi realizada utilizando ferramentas como XAMPP, Burp Suite e SQLMap, aplicadas em um ambiente de teste com vulnerabilidades intencionais, o que permitiu simular e analisar ataques. Constatou-se que práticas como a validação de entradas e a parametrização de consultas são essenciais para reduzir os riscos de ataques de injeção SQL. Conclui-se que a segurança de aplicações web exige não apenas ferramentas, mas também uma abordagem contínua e integrada de proteção durante o desenvolvimento.

Palavras-chave: segurança da informação, SQL *Injection*, segurança em aplicações web, SQLMap, Burp Suite.

ABSTRATC

This study provides an in-depth analysis of implementing security mechanisms to prevent SQL Injection attacks on web applications, emphasizing the impact of this vulnerability on data integrity and security. It explores SQL language, its key commands, and how security flaws in this context can be exploited by attackers to manipulate databases. The research was conducted using tools such as XAMPP, Burp Suite, and SQLMap within a deliberately vulnerable testing environment, allowing for the simulation and analysis of attacks. Findings indicate that practices such as input validation and query parameterization are essential to mitigating SQL Injection risks. The study concludes that web application security requires not only tools but also a continuous, integrated protection approach throughout development.

Keywords: information security, SQL Injection, web application security, SQLMap, Burp Suite.

1 INTRODUÇÃO

Hoje em dia, criar aplicações web é algo bem comum e vai desde plataformas de e-commerce e bancos digitais até serviços de comunicação e outros sistemas online. Com o crescimento dessas aplicações, a segurança acaba se tornando uma preocupação cada vez maior, já que a ameaça de ataques cibernéticos é constante. A arquitetura de muitas dessas aplicações web geralmente tem pontos vulneráveis, e garantir a segurança delas fica nas mãos dos desenvolvedores, que precisam adotar boas práticas de programação e proteção de dados.

Este trabalho busca explorar um dos ataques mais conhecidos e perigosos em ambientes web, o SQL *Injection*. Esse ataque acontece quando um invasor insere comandos SQL maliciosos nos campos de entrada de dados de um sistema, tentando acessar informações confidenciais, alterar ou até mesmo destruir dados importantes para o funcionamento da aplicação. Esse tipo de ataque pode colocar em risco a segurança e a integridade de uma organização, afetando tanto sua reputação quanto seu desempenho operacional. Vale destacar que a falta de validação correta dos dados de entrada e a ausência de práticas de segurança, como a parametrização de consultas SQL, são fatores que deixam o sistema mais vulnerável a esse tipo de ataque.

2 REFERENCIAL TEÓRICO

As próximas partes deste projeto são fundamentais para construir uma base sólida, ajudando no entendimento mais aprofundado dos conceitos essenciais para o avanço da pesquisa.

2.1 LINGUAGEM SQL

De acordo com Date (2000), a SQL (*Structured Query Language*) é uma linguagem amplamente usada para definir, manipular e gerenciar dados em sistemas de bancos de dados relacionais. Nesse tipo de sistema, as informações são organizadas em tabelas compostas por linhas e colunas, facilitando tanto a estruturação quanto a recuperação de dados. Cada tabela reflete uma entidade específica, e os vínculos entre essas entidades são estabelecidos por meio de valores comuns, como chaves primárias e estrangeiras. As operações de consulta e manipulação dos dados são feitas por meio de comandos SQL, que permitem realizar operações envolvendo uma ou mais tabelas, como filtrar, agrupar, ordenar ou combinar informações. Além disso, o SQL oferece recursos de controle de transações e integridade referencial, assegurando a consistência das operações realizadas nos dados.

A linguagem SQL possui diversos comandos principais, que são a base para a manipulação de dados em um banco de dados relacional, entre eles:

SELECT: para consultar informações no banco de dados.
INSERT: para incluir novos registros.
UPDATE: para alterar dados já existentes.
DELETE: para remover registros.

Nos sistemas de informação, que são alvos frequentes de ataques de injeção de SQL, essa linguagem é utilizada em chamadas feitas por aplicações para APIs específicas. A maneira como essa integração é projetada impacta diretamente a segurança do sistema. Além disso, muitos fabricantes de bancos de dados oferecem extensões proprietárias ao padrão SQL, com o objetivo de ampliar as funcionalidades de seus produtos. Embora essas extensões proporcionem mais recursos aos desenvolvedores, elas também podem introduzir novas vulnerabilidades, o

que aumenta a responsabilidade dos profissionais de TI em proteger os sistemas contra ataques de injeção de SQL.

Dessa forma, é essencial que os desenvolvedores não apenas conheçam a evolução da SQL, mas também estejam atentos às práticas recomendadas de segurança ao utilizá-la. Uma integração inadequada ou o uso imprudente de funcionalidades proprietárias podem criar brechas que facilitam ataques. Assim, adotar medidas de proteção robustas é crucial para garantir a segurança e integridade dos sistemas que fazem uso de SQL.

2.2 SQL *INJECTION*

Segundo PortSwigger (2024), a injeção SQL é uma das vulnerabilidades mais críticas e comuns em aplicações web, permitindo que atacantes interfiram nas consultas feitas ao banco de dados por meio da manipulação de entradas. Essa técnica pode resultar em acesso não autorizado a dados sensíveis, comprometendo a segurança e a integridade do sistema.

Aplicações que criam instruções SQL, especialmente aquelas que utilizam diretamente dados inseridos pelo usuário, devem ser rigorosamente revisadas para evitar vulnerabilidades de injeção de SQL. O uso de parâmetros em consultas é uma prática recomendada para aumentar a segurança, mas não garante proteção total; invasores experientes podem ainda manipular entradas para explorar brechas.

Qualquer sistema que interaja com um Sistema de Gerenciamento de Banco de Dados (SGBD) e utilize consultas SQL dinâmicas com dados fornecidos pelos usuários está suscetível a ataques de injeção de SQL. A vulnerabilidade ocorre principalmente quando essas entradas não são adequadamente validadas ou higienizadas, permitindo que o invasor altere a consulta e execute comandos maliciosos.

Adicionalmente, há fatores que podem intensificar o risco de exposição ao ataque. Um deles é a capacidade do SGBD de interpretar comentários dentro das instruções SQL, o que facilita a modificação da lógica das consultas. Outro fator de risco é a execução de comandos em lote, pois permite que várias instruções sejam processadas em uma única operação, ampliando o potencial de exploração.

Outro aspecto importante é o acesso a metadados e tabelas de sistema, o que pode fornecer ao atacante informações sobre a estrutura do banco de dados, como nomes de tabelas e colunas. O acesso a esses dados compromete a integridade, confidencialidade e disponibilidade do sistema, configurando um risco sério para a segurança.

2.3 XAMPP

Para Apache Friends (2024), o Xampp é uma distribuição gratuita e de código aberto que facilita a instalação de um servidor web local, incluindo componentes como Apache, MariaDB, PHP e Perl. Isso permite que desenvolvedores criem e testem aplicações web em seus próprios computadores antes de publicá-las na internet.

O Xampp é amplamente utilizado para testes de páginas web devido à sua facilidade de uso e instalação. Ele simula um ambiente de servidor web, eliminando a necessidade de upload de arquivos para um servidor online, o que facilita o desenvolvimento e a verificação de funcionalidades localmente.

2.4 MYSQL

De acordo com a documentação oficial do MySQL (2024), o MySQL é um sistema de gerenciamento de banco de dados relacional de código aberto, conhecido por sua eficiência, confiabilidade e flexibilidade, sendo amplamente adotado em ambientes de desenvolvimento

web e aplicativos críticos de negócios. Ele suporta múltiplas plataformas, como Windows, Linux e Mac OS, e fornece uma interface SQL para manipulação de dados. As principais características incluem suporte a grandes volumes de dados, alta compatibilidade com padrões SQL, integração com várias linguagens de programação, e estabilidade ao lidar com operações intensas de leitura e escrita. O MySQL é também reconhecido por seu suporte a múltiplos usuários, controle de permissões e recursos de segurança, como criptografia de senhas. Além disso, oferece ferramentas para backups e recuperação de desastres, além de ser altamente configurável para diferentes necessidades de desempenho e armazenamento.

2.5 BURP SUITE

O Burp Suite é uma plataforma integrada e robusta, especialmente projetada para realizar testes de segurança em aplicações web. Desenvolvida pela PortSwigger, a ferramenta oferece um conjunto de funcionalidades que trabalham de maneira coordenada, permitindo desde a identificação precoce de vulnerabilidades até a exploração detalhada das falhas encontradas. Entre suas principais capacidades, destacam-se a varredura de segurança automatizada e a análise manual, possibilitando que os profissionais testem a resistência de uma aplicação contra diversos tipos de ataques, como injeção de SQL, XSS (*cross-site scripting*), ataques de força bruta, entre outros.

Uma das grandes vantagens do Burp Suite é sua flexibilidade, que permite a integração com outras ferramentas de segurança, como o SQLmap, aumentando significativamente a eficácia dos testes. Ao atuar como um proxy, o Burp Suite intercepta e filtra as requisições e respostas entre o navegador e a aplicação web, oferecendo ao testador um controle detalhado do tráfego. Essa funcionalidade é particularmente útil quando usada em conjunto com o SQLmap, que pode aproveitar os dados coletados pelo Burp Suite para executar ataques direcionados, como a identificação de nomes de tabelas, colunas e outros detalhes estruturais do banco de dados da aplicação. Essa integração fortalece o processo de teste, tornando a análise mais precisa e eficiente.

2.6 SQLMAP

De acordo com SQLMap (2024), o SQLMap é uma poderosa ferramenta de teste de segurança de código aberto, projetada para automatizar a detecção e exploração de vulnerabilidades de injeção SQL em bancos de dados, oferecendo recursos avançados que facilitam desde a extração de dados até o comprometimento de sistemas inteiros.

O SQLMap oferece uma interface de linha de comando que permite aos testadores de segurança realizar ataques complexos de SQL *Injection* de forma eficiente e automatizada. Ele suporta uma variedade de técnicas de injeção, como ataques baseados em tempo, erros, booleanos, entre outros, adaptando-se ao tipo de banco de dados e à complexidade do sistema alvo. Essa versatilidade torna o SQLMap uma escolha popular para avaliar a segurança de aplicações web e detectar pontos vulneráveis nos sistemas de armazenamento de dados.

Para rodar o SQLMap, é essencial ter o Python instalado, pois a ferramenta foi escrita nessa linguagem, o que a torna compatível com vários sistemas operacionais e fácil de modificar para necessidades específicas. O uso do Python permite ao SQLMap integrar-se com outras ferramentas de segurança e personalizar o fluxo de trabalho de testes, proporcionando maior controle e uma abordagem mais abrangente na detecção de vulnerabilidades de injeção SQL.

2.7 PYTHON

De acordo com a AWS (2024), Python é uma linguagem de programação versátil e acessível, amplamente utilizada para desenvolvimento web, ciência de dados, inteligência artificial e automação, devido à sua sintaxe simples e à vasta coleção de bibliotecas que facilitam a criação e execução de projetos complexos. Essa versatilidade do Python também o torna a base para ferramentas de segurança, como o SQLMap, que utiliza a linguagem para realizar ataques de injeção SQL automatizados.

2.8 PHP

O PHP é uma linguagem de programação amplamente utilizada para o desenvolvimento de sites dinâmicos, permitindo uma interação eficiente com o usuário por meio de formulários, parâmetros de URL e links. Ao contrário de linguagens como JavaScript, que são executadas diretamente no navegador do cliente, o PHP roda no lado do servidor. Isso significa que o servidor processa o código PHP e envia ao cliente apenas o HTML gerado, sem expor o código-fonte.

Essa característica oferece uma camada adicional de segurança, especialmente quando o site precisa lidar com informações sensíveis, como senhas ou dados pessoais, já que o código nunca é visível para o usuário final. O PHP também facilita a integração com bancos de dados e outras aplicações no servidor, tornando-o uma ferramenta poderosa para o desenvolvimento de aplicações web robustas e seguras.

Comparado a scripts CGI escritos em linguagens como C ou Perl, o PHP se destaca por ser incorporado diretamente no código HTML, simplificando o processo de desenvolvimento. Enquanto scripts CGI exigem que todo o HTML seja gerado separadamente pelo script, o PHP permite que o desenvolvedor mescle facilmente código PHP com HTML, o que torna o processo de criação de páginas dinâmicas mais ágil e intuitivo. Além disso, o PHP é altamente compatível com diversos servidores e bancos de dados, o que contribui para sua popularidade no desenvolvimento web.

3. METODOLOGIA

A linguagem de programação utilizada na infraestrutura deste projeto foi o PHP, na versão 8.3.6, escolhida pela sua robustez e ampla adoção no desenvolvimento de aplicações web. O Sistema de Gerenciamento de Banco de Dados (SGBD) empregado para o armazenamento dos dados foi o MySQL, versão 8.0, devido à sua confiabilidade e eficiência no gerenciamento de grandes volumes de informações. Para varredura da aplicação foi utilizar o Burp Suite versão v2024.8.5 e para coleta de dados para a utilização do SQL *Injection* foi utilizado o SQLmap versão 2.

O PHP foi utilizado para desenvolver um software web realizado pelos próprios autores deste artigo, cujo objetivo principal é realizar testes de vulnerabilidade focados na etapa de autenticação, especificamente no momento em que o usuário insere suas credenciais de login e após o login.

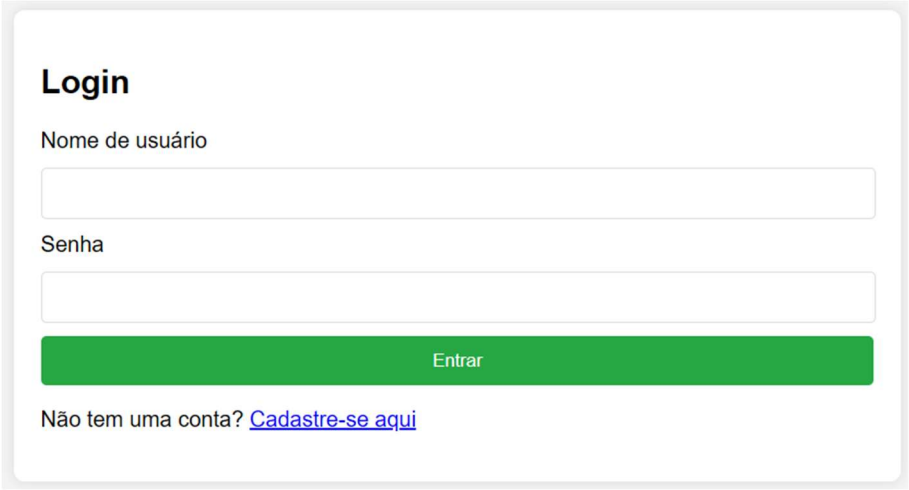
O software foi criado intencionalmente com vulnerabilidades para simular um site suscetível a ataques de SQL *Injection*. Nesse ambiente, é possível realizar tentativas de ataque cibernético por meio dos campos de senha, da tela de login, e pesquisa, da tela do *dashboard*. As simulações têm o objetivo de detectar falhas de segurança que possam comprometer o sistema, permitindo que sejam identificadas e corrigidas antes da implementação em produção. O sistema foi executado localmente na máquina através do XAMPP.

3.1 TESTE DE VULNERABILIDADE UTILIZANDO INJEÇÃO SQL

3.1.1 TELA DE LOGIN E CADASTRO

Este é um exemplo de formulário básico de login (Figura 1), onde o usuário insere suas credenciais, como nome de usuário e senha, para acessar o sistema. Essas informações são verificadas com os dados armazenados no banco de dados MySQL para autenticar o usuário. No entanto, um dos campos de entrada esta suscetível a ataques de injeção de SQL (SQL *Injection*).

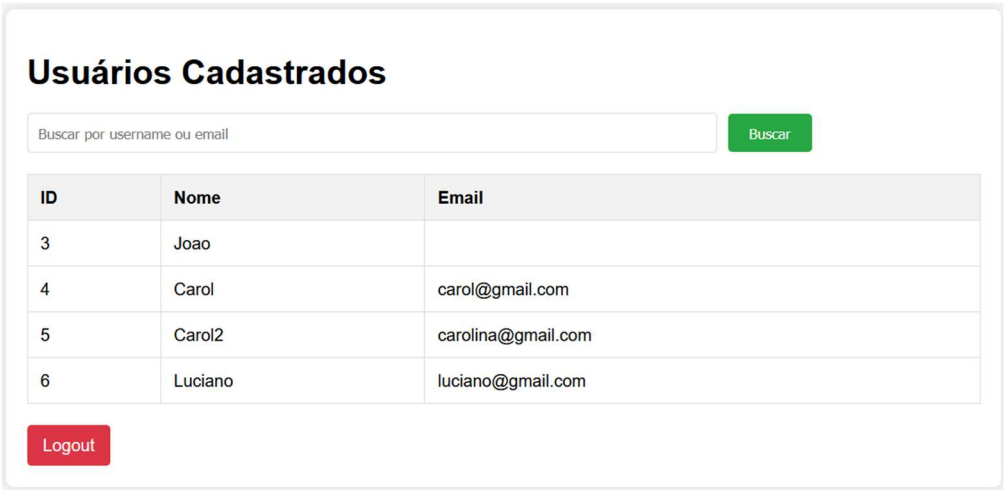
Figura 1 - Tela de Login



Fonte: Autores

Se o sistema identificar um usuário registrado com as credenciais fornecidas, ele automaticamente redirecionará o usuário para o *dashboard*, já autenticado. Nesse painel, o usuário logado poderá buscar por nome ou e-mail dos demais usuários cadastrados. Conforme ilustrado na Figura 2, o campo de busca também está suscetível a injeção SQL, por ele é possível realizar buscas, alterações ou até mesmo excluir algum dado, porém é preciso ter informações do nome da tabela.

Figura 2 - *Dashboard*

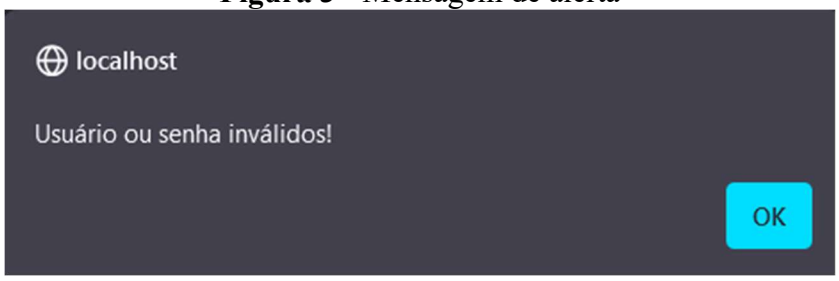


Fonte: Autores

Se as credenciais fornecidas não corresponderem a nenhum registro existente no sistema, aparecerá um pop-up. Nessa tela, será exibida uma mensagem de erro indicando que o

nome de usuário ou senha inseridos estão incorretos, impedindo o acesso ao sistema. Além disso, o sistema permitirá que o usuário tente novamente o login após clicar no OK da mensagem, garantindo uma nova oportunidade para corrigir as informações. Esse comportamento pode ser visualizado na Figura 3, que demonstra o fluxo de falha na autenticação.

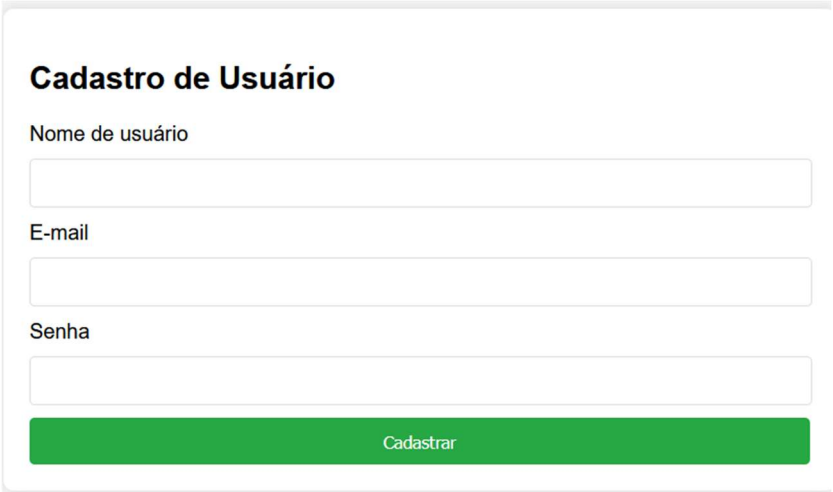
Figura 3 - Mensagem de alerta



Fonte: Autores

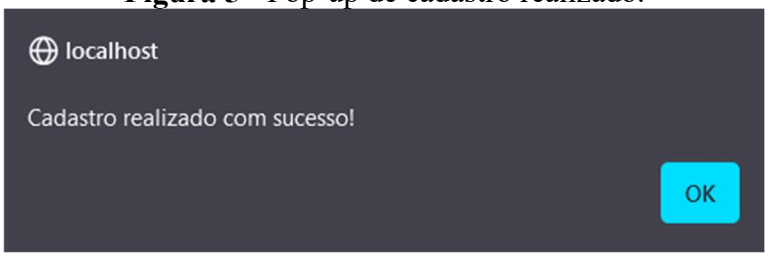
Caso o usuário não possua um cadastro prévio no sistema e deseje se registrar, poderá acessar a tela de cadastro ao clicar no link "Cadastre-se aqui", conforme ilustrado na Figura 1. Na tela de cadastro (Figura 4), o usuário deverá preencher os campos obrigatórios: nome de usuário, e-mail e senha. Após o preenchimento desses dados, será necessário clicar no botão "Cadastrar", momento em que aparecerá um pop-up com a mensagem “Cadastro realizado com sucesso!”, conforme demonstrado na Figura 5, e após clicar em OK o usuário será redirecionado automaticamente para a tela de login.

Figura 4 - Tela de Cadastro.



Fonte: Autores.

Figura 5 - Pop-up de cadastro realizado.



Fonte: Autores.

3.1.2 INJEÇÃO SQL LOGIN

Foi utilizado o comando ‘ or ‘’ = ‘ para acessar o sistema sem autorização. Esse comando deve ser inserido no campo de senha, que está vulnerável a ataques de SQL *Injection* devido à forma como o sistema foi implementado, o que cria brechas para invasões. O campo de usuário pode conter qualquer informação.

Abaixo está o código SQL que é responsável pela verificação do usuário e senha, esse código é vulnerável ao comando apresentado acima:

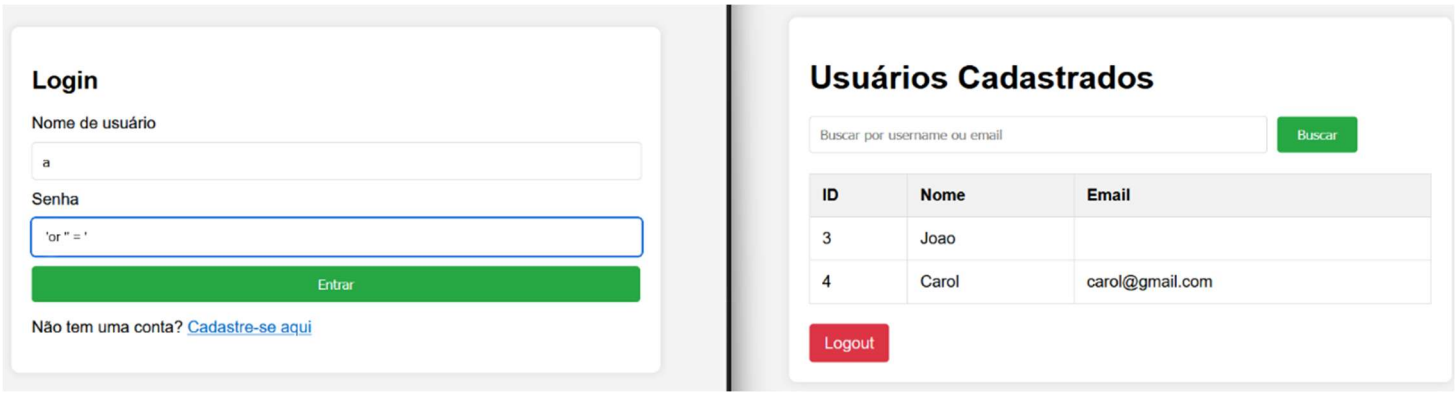
```
SELECT * FROM users WHERE username= '$username' AND password = '$password';
```

Pela falta de tratamento de segurança o código se torna vulnerável, pois as variáveis $username e $password estão inseridos diretamente na consulta SQL. Podemos exemplificar da seguinte maneira: a variável $username receberá o valor usuário e a variável $password receberá o comando ‘or ‘’ = ‘, como demonstrado abaixo:

```
SELECT * FROM users WHERE username = 'usuario' AND password = '' OR '' = '';
```

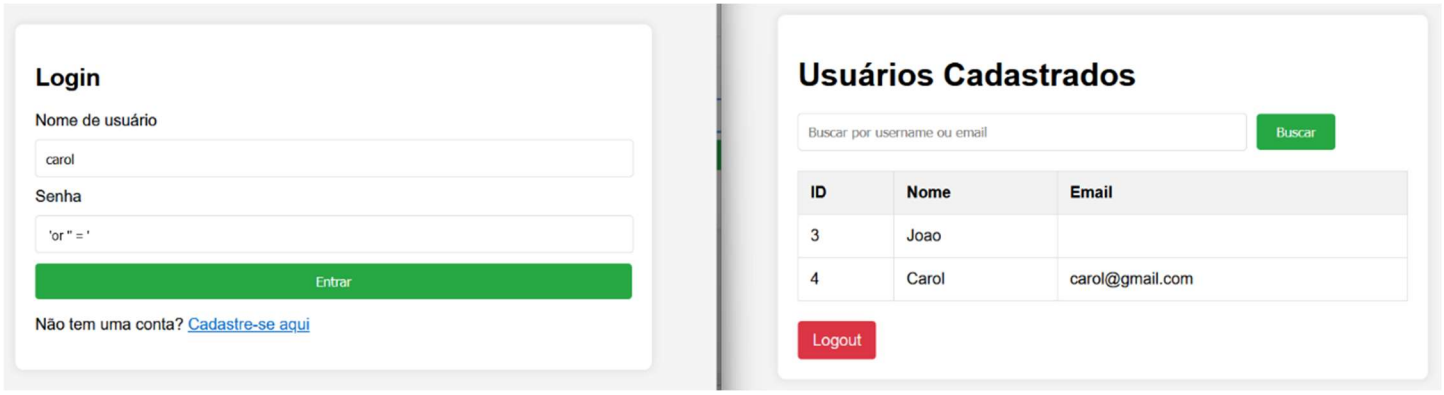
Neste caso, as condições para $username e $password deveriam verificar se as informações fornecidas estão corretas. Caso fossem falsas, o login não ocorreria. No entanto, há uma terceira condição ligada pelo operador OR, que faz com que a consulta retorne verdadeiro, independentemente dos valores de $username e $password. Como resultado, o código é executado e a tela do *dashboard* é acessada. Isso pode ser visto nas Figuras 6 e 7.

Figura 6 - Login sem nome de usuário cadastrado



Fonte: Autores

Figura 7 - Login com nome de usuário cadastrado



Fonte: Autores

Na Figura 8 é possível visualizar com mais clareza como a autenticação para o login foi escrita e tendo seu código exposto.

Figura 8 - Código fonte da autenticação do login sem proteção

authenticate.php > ...

```php
// Inicia a sessão
session_start();

// Configurações do banco de dados
$host = "localhost";  // Host do banco de dados
$dbname = "users";  // Nome do banco de dados
$db_username = "root";  // Usuário do banco de dados
$db_password = "";  // Senha do banco de dados

// Cria uma conexão com o MySQL
$conn = new mysqli($host, $db_username, $db_password, $dbname);

// Verifica se a conexão falhou
if ($conn->connect_error) {
    die("Conexão falhou: " . $conn->connect_error);
}

// Obtém dados do formulário
$username = $_POST['username'];
$password = $_POST['password'];

// Consulta o banco de dados para verificar as credenciais
$query = "SELECT * FROM users WHERE username = '$username' AND password = '$password'";
$result = $conn->query($query);

if ($result->num_rows > 0) {
    // Se o login for bem-sucedido
    $user = $result->fetch_assoc();

    // Armazena os dados do usuário na sessão
    $_SESSION['username'] = $user['username'];
    $_SESSION['user_id'] = $user['id'];

    // Redireciona para o dashboard
    header("Location: dashboard.php");
    exit();
} else {
    // Se o login falhar, redireciona de volta para a página de login com uma mensagem de erro
    echo "<script>alert('Invalid username or password!'); window.location.href='index.php';</script>";
    exit();
}

// Fecha a conexão com o banco de dados
$conn->close();
?>
```

Fonte: Autores

3.1.3 INJEÇÃO SQL *DASHBOARD*

Na tela do *dashboard* (Figura 2), como mencionado a cima, existe uma vulnerabilidade que permite fazer alterações nos usuários que listam na tela e em demais tabelas existentes no banco de dados, mas que não são listadas visualmente.

Na Figura 9, é possível observar o uso do comando *mysqli_multi_query*. O uso desse comando permite a execução de várias consultas de uma só vez, o que aumenta o risco de ataques, pois permite a adição de comandos extras que serão executados junto com a consulta original.

Figura 9 - Código fonte *dashboard*

dashboard.php

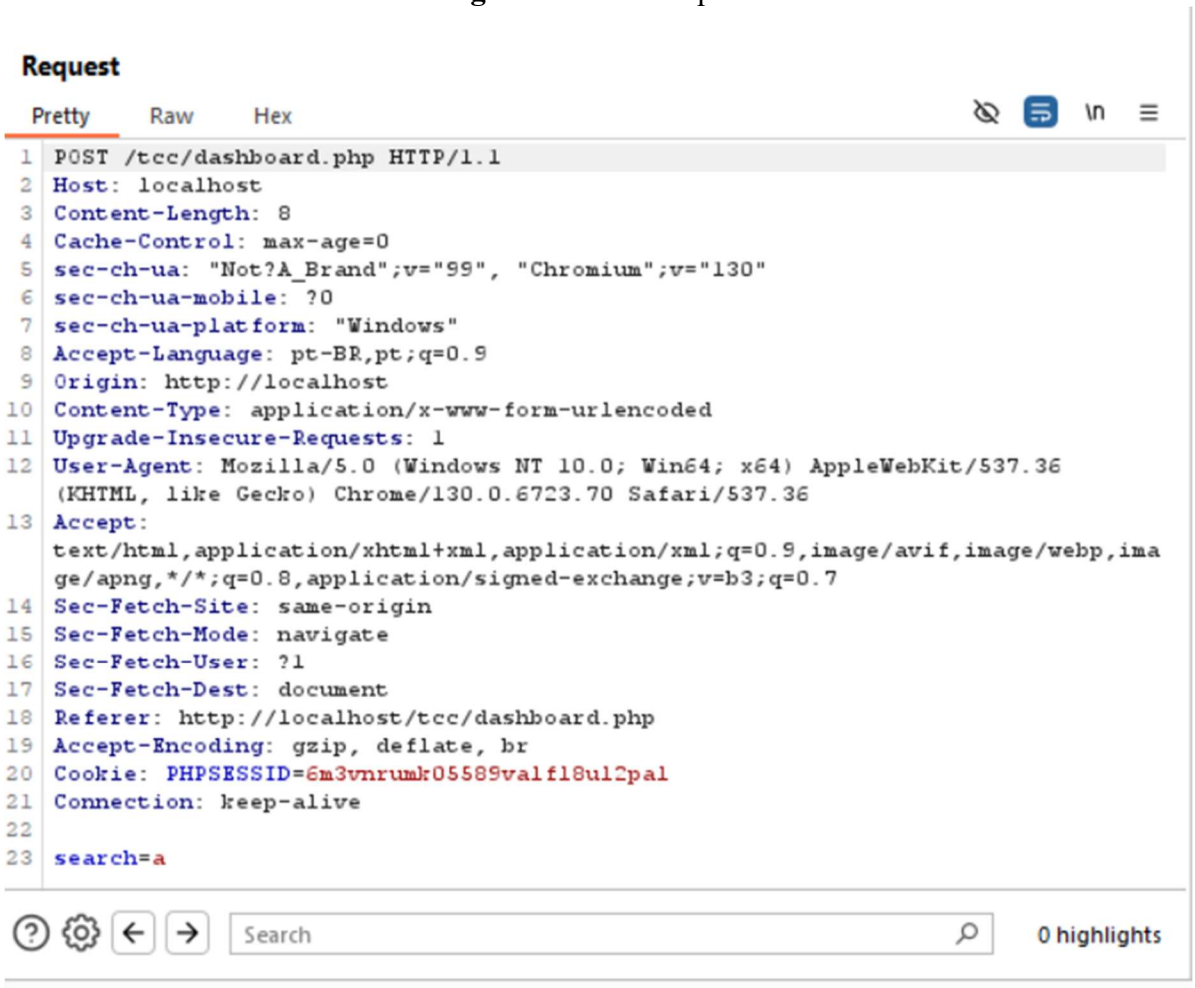
```php
// Verifica se o usuário está logado
if (!isset($_SESSION['username'])) {
    // Se não estiver logado, redireciona para a página de login
    header("Location: index.php");
    exit();
}

// Configurações do banco de dados
$host = "localhost";
$dbname = "users";
$db_username = "root";
$db_password = "";

// Cria uma conexão com o MySQL
$conn = new mysqli($host, $db_username, $db_password, $dbname);

// Verifica se a conexão falhou
if ($conn->connect_error) {
    die("Conexão falhou: " . $conn->connect_error);
}

// Inicializa a variável de busca
$search = "";

// Verifica se o formulário de busca foi enviado
if (isset($_POST['search'])) {
    $search = $_POST['search'];
}

// Consulta SQL para buscar usuários e executar outra query ao mesmo tempo
$query = "
    SELECT * FROM users WHERE username LIKE '%$search%' OR email LIKE '%$search%';
    UPDATE users SET last_search = NOW() WHERE username LIKE '%$search%' OR email LIKE '%$search%';
";

// Executa as múltiplas queries
if (mysqli_multi_query($conn, $query)) {
```

Fonte: Autores

Para ser possível realizar alguma modificação nos dados da aplicação será preciso descobrir o(s) nome(s) da(s) tabela(s) e suas colunas, para isso foi utilizado o Burp Suite e SQLmap. Pelo Burp Suite é possível escanear os dados de requisição da página, dessa forma conseguimos verificar quem é a variável que será usada no SQLmap.

Para que isso ocorra é preciso estar com o Burp Suite aberto, usar o navegador do mesmo e realizar o login com usuário cadastrado ou não. Na página *dashboard* será feito uma pesquisa no campo buscar, após realizar a pesquisa, será verificado no Burp Suite a requisição post, na Figura 10 conseguimos ver os dados capturados.

Figura 10 - Post request



Fonte: Autores

Após conseguir essas informações foi copiado o conteúdo da requisição para um bloco de notas, nomeado como "*dashboard*". Esse bloco de notas foi colocado dentro da pasta do SQLmap.

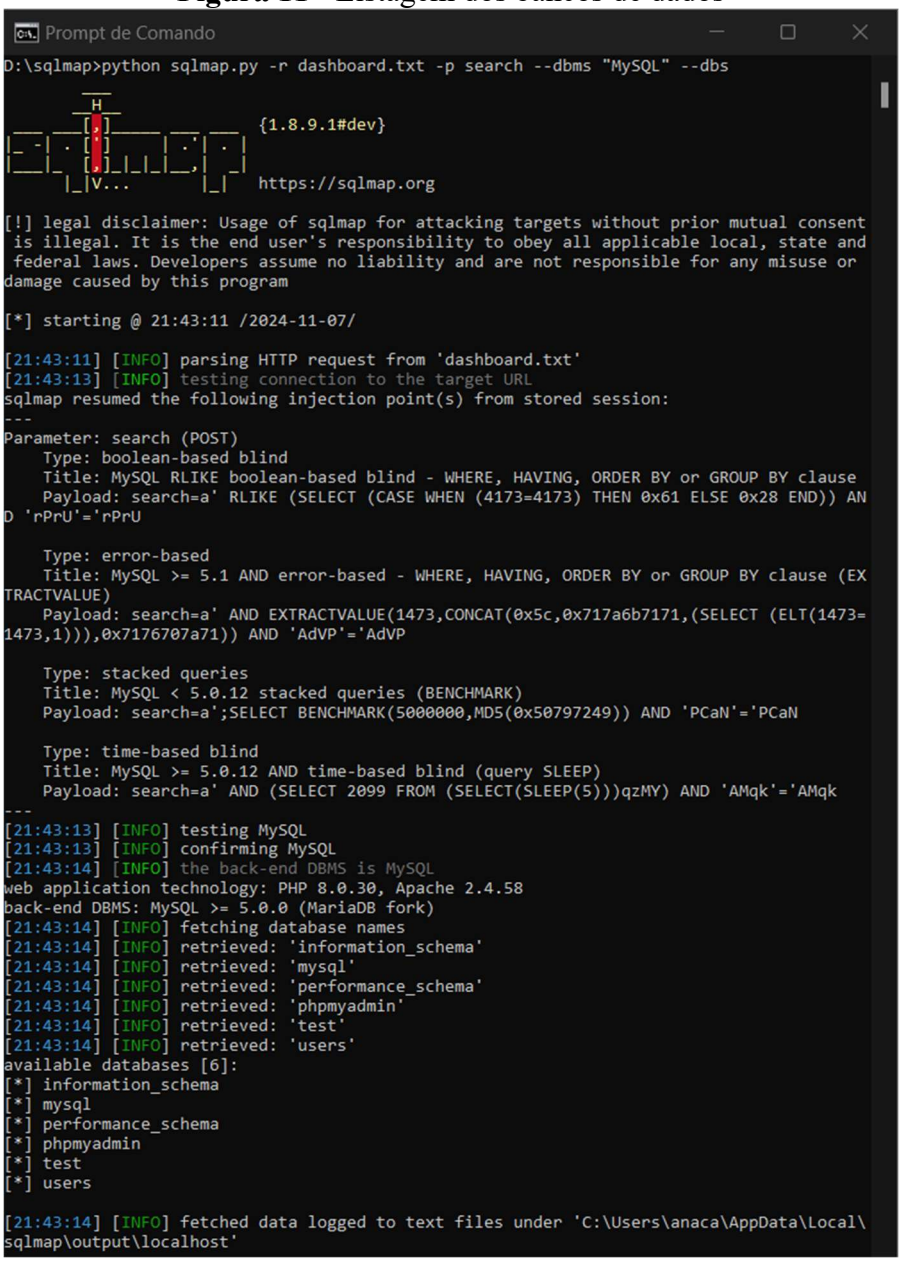
O Prompt de Comando foi aberto, e então foi acessada a unidade e a pasta onde o SQLmap está localizado para executar seus comandos.

Para iniciar a análise da vulnerabilidade no campo "*search*" usamos o comando abaixo:

```
python sqlmap.py -r dashboard.txt -p search --dbms "MySQL" --dbs
```

O parâmetro python sqlmap.py executa o arquivo sqlmap.py, sendo o comando principal para o funcionamento do SQLmap. O parâmetro -r dashboard.txt define o caminho do arquivo que contém os dados da requisição HTTP, enquanto -p search especifica o parâmetro a ser testado para injeção SQL. Neste caso, o parâmetro *search* é o alvo, permitindo ao SQLmap focar nesse campo específico para verificar vulnerabilidades. O parâmetro --dbms "MySQL" informa ao SQLmap que o banco de dados alvo é MySQL, permitindo que adapte os testes a esse banco. Por fim, --dbs faz com que o SQLmap liste todos os bancos de dados (Figura 11) disponíveis no servidor.

Figura 11 - Listagem dos bancos de dados



Fonte: Autores

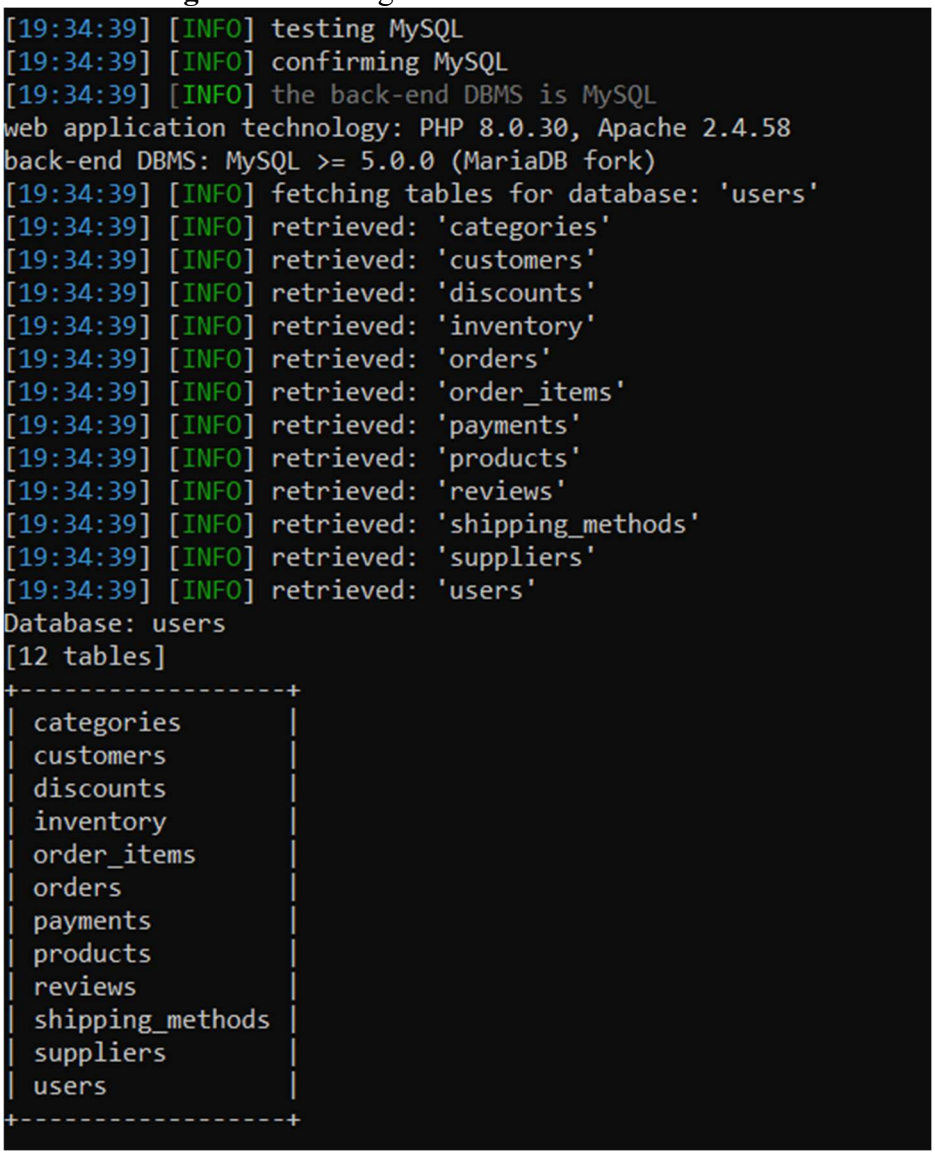
Após conseguir as informações de vulnerabilidades existentes no parâmetro *search* e também os nomes dos bancos de dados, será possível gerar um novo comando filtrando apenas o banco de dados especifico, nesse caso o "*users*".

Para que fosse possível conseguir os nomes das tabelas foi executado o seguinte comando no prompt de comando:

```
python sqlmap.py -r dashboard.txt -p search --dbms "MySQL" -D users --tables
```

A execução do comando anterior continha o --dbs, porém após execução dele o SQLmap consegue guardar em um log essas informações e, com isso, não é preciso mais utiliza-lo, mas para ter acesso as tabelas, é preciso substitui-lo por -D users --tables, nesse comando informamos que o banco de dados a ser utilizado é o "*users*" e queremos visualizar suas tabelas.

Figura 12 - Listagem das tabelas do banco *users*



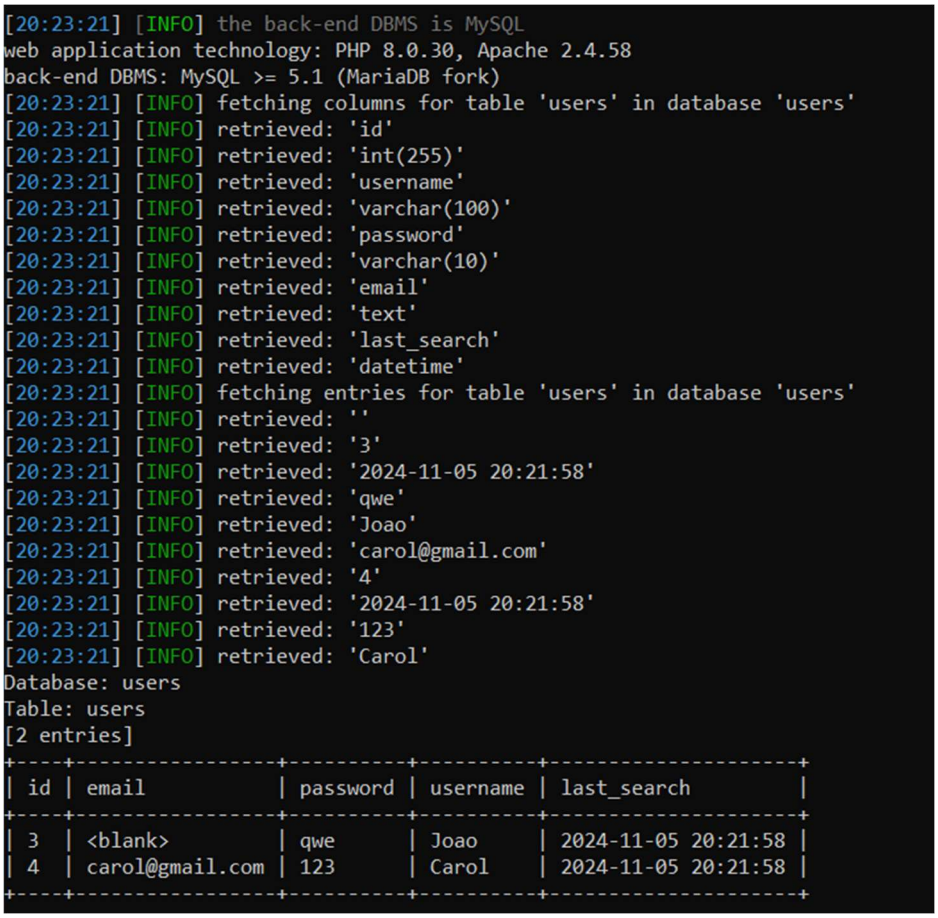
Fonte: Autores

A fim de descobrir os dados da tabela, foi usado o comando abaixo:

```
python sqlmap.py -r dashboard.txt -p search --dbms "MySQL" -D users -T users --dump
```

No comando acima, foi necessário substituir o parâmetro --table por -T users e --dump. O parâmetro -T users define a tabela específica da qual os dados serão extraídos, enquanto --dump realiza a extração dos dados. Com esses dois novos parâmetros, o SQLmap extrai as informações da tabela, permitindo-nos obter todos os dados necessários para realizar alterações (Figura 13).

Figura 13 - Extração dos dados da tabela *users*



Fonte: Autores

Após ter verificado se há vulnerabilidade no site, conseguimos filtrar o nome de todas as tabelas e colunas existentes no banco de dados, dessa maneira teremos a oportunidade de realizar alguma alteração nelas através do campo de busca. Por mais que o banco de dados utilizado possui muitas tabelas, o foco dos autores é apenas realizar alterações na tabela visível de cadastro de usuários, dessa forma a tabela a ser utilizada será a "*users*".

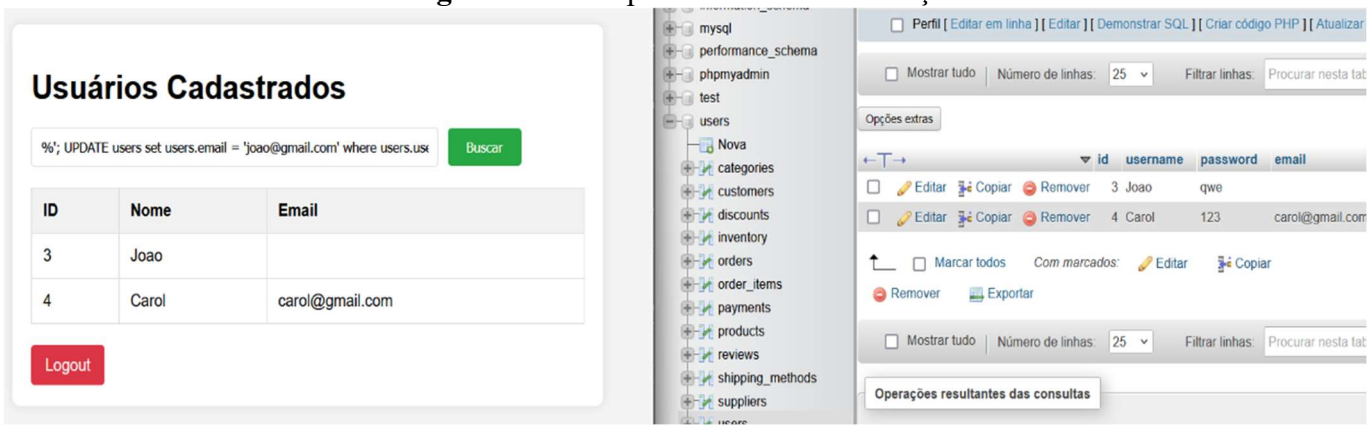
Para que fosse possível fazer a alteração em um dos usuários existentes, foi realizado o seguinte comando no campo de busca:

%'; UPDATE users SET users.email = 'joao@gmail.com' WHRE users.username='Joao';'%

Observa-se que o foco da atualização (update) foi no campo E-mail quando o nome de usuário era João. A informação sobre qual usuário precisávamos já estava visível, mas o que facilitou o processo foi obter o nome da tabela e o nome das colunas que armazenam os dados de usuário. Essa alteração também poderia ter sido feita utilizando o ID ou o E-mail.

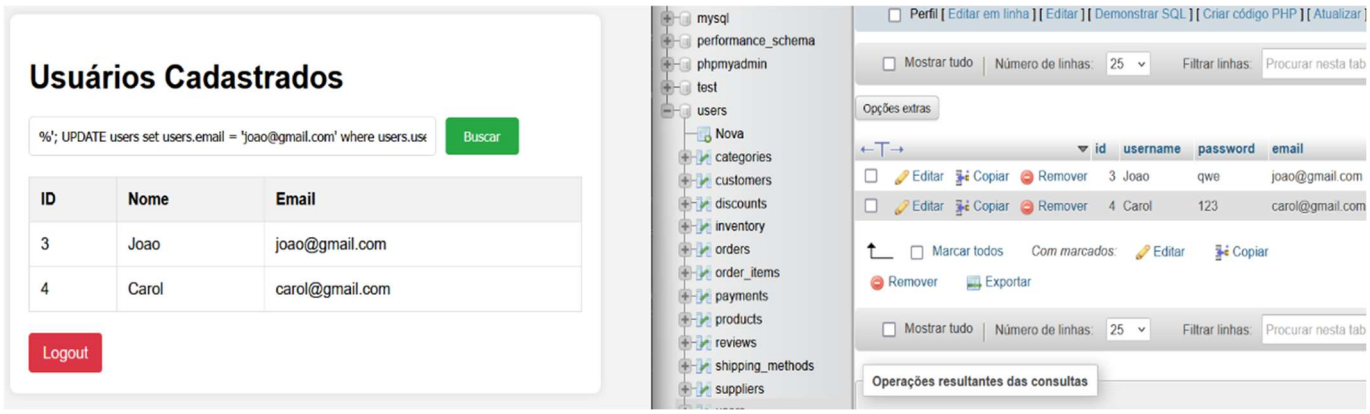
Na Figura 14, é possível observar que o campo E-mail do usuário João estava vazio tanto na visualização do painel quanto no banco de dados. Após a execução do comando citado acima, o campo passou a receber a informação atualizada no banco de dados e a ser visível no painel, conforme ilustrado na Figura 15.

Figura 14 - Campo e-mail sem informação



Fonte: Autores

Figura 15 - Campo e-mail com informação



Fonte: Autores

3.2 PROTEÇÃO CONTRA INJEÇÃO SQL

Afim de evitar a vulnerabilidade na aplicação, o código foi atualizado para utilizar *prepared statements*. Com essa técnica, a estrutura da consulta é separada dos dados inseridos pelo usuário, o que impede a execução de comandos maliciosos. Dessa forma, o banco de dados reconhece os valores dos parâmetros apenas como dados, e não como parte do código SQL, bloqueando tentativas de injeção e aumentando a segurança da aplicação.

Com base no código mostrado na Figura 16, é possível observar que o código do SELECT referente à autenticação de login agora utiliza pontos de interrogação (?) como espaços reservados, o que torna o código mais seguro e eficiente. Em vez de inserir diretamente os valores de username e password na consulta SQL, o que poderia deixar o sistema vulnerável a ataques de SQL *Injection*, o uso de espaços reservados permite que os dados do usuário sejam vinculados à consulta de maneira segura, separando os dados da lógica SQL. A função prepare cria uma versão pré-compilada da consulta SQL, e os valores de username e password são vinculados a essa consulta no momento da execução.

Esse processo de vinculação é feito por meio do método *bind_param*, que também define os tipos de dados dos parâmetros (neste caso, "ss" indica que ambos os parâmetros são strings). Quando a consulta é executada com o método execute, os valores de username e password são inseridos nos espaços reservados e enviados para o banco de dados de forma segura. Assim, mesmo que um invasor tente injetar código SQL malicioso, ele será tratado apenas como um valor de texto, e não como parte da lógica SQL.

Por fim, o método *get_result* permite acessar o conjunto de resultados retornado pela consulta. Dessa forma, é possível verificar se existe algum registro correspondente no banco de dados, garantindo que o usuário forneceu credenciais válidas antes de prosseguir com o login. Caso o usuário seja um invasor e tente usar o código ' or '' = ', ele não conseguirá acessar o sistema e será direcionado ao pop-up ilustrado na Figura 3.

Figura 16 - Código fonte da autenticação protegido

```
19  // Obtém dados do formulário
20  $username = $_POST['username'];
21  $password = $_POST['password'];
22
23  // Consulta segura ao banco de dados para verificar as credenciais
24  $query = "SELECT * FROM users WHERE username = ? AND password = ?";
25  $stmt = $conn->prepare($query);
26  $stmt->bind_param("ss", $username, $password);
27  $stmt->execute();
28  $result = $stmt->get_result();
```

Fonte: Autores

Para corrigir o código do *dashboard*, usamos a mesma técnica aplicada no login. Antes, o *dashboard* rodava seu código usando *mysqli_multi_query*, mas agora, com essa nova abordagem, ele executa uma função por vez. Isso fica claro no código mostrado na Figura 17.

Figura 17 - Código fonte do *dashboard* protegido

```
32  // Consulta segura para buscar usuários
33  $query_select = "SELECT * FROM users WHERE username LIKE ? OR email LIKE ?";
34  $stmt_select = $conn->prepare($query_select);
35
36  // Preparando o parâmetro de pesquisa para a consulta
37  $search_param = "%$search%";
38  $stmt_select->bind_param("ss", $search_param, $search_param);
39  $stmt_select->execute();
40  $result = $stmt_select->get_result();
41
42  // Consulta segura para atualizar o campo last_search
43  $query_update = "UPDATE users SET last_search = NOW() WHERE username LIKE ? OR email LIKE ?";
44  $stmt_update = $conn->prepare($query_update);
45  $stmt_update->bind_param("ss", $search_param, $search_param);
46  $stmt_update->execute();
47  ?>
```

Fonte: Autores

Depois das alterações, o sistema foi testado e as vulnerabilidades foram eliminadas. Assim, não era mais possível fazer login sem usuário e senha cadastrados, e o SQLmap já não conseguia identificar falhas para extrair informações do banco de dados.

4 ANÁLISE E DISCUSSÃO DOS RESULTADOS

Na análise, foi identificado que a falta de tratamento adequado no código PHP permitia a exploração por meio de injeção SQL. Para reduzir esses riscos, foram aplicadas práticas como o uso de *prepared statements*, que separam os dados dos usuários da lógica SQL, evitando que

comandos maliciosos sejam executados. Com essa abordagem, a segurança da aplicação é reforçada, já que os dados de entrada são tratados como valores e não como código.

Os testes mostraram que vulnerabilidades de SQL *Injection* podem comprometer não só a integridade dos dados, mas também expor informações sensíveis e permitir manipulação do sistema sem autorização. Para corrigir o código, a solução proposta incluiu o uso de consultas parametrizadas e métodos que limitam a execução de múltiplas consultas, evitando que entradas maliciosas consigam explorar a aplicação.

Esses resultados destacam a importância de práticas de segurança no desenvolvimento web. Medidas simples, como o uso de *prepared statements* e a validação rigorosa das entradas, são eficazes para proteger contra ataques comuns e manter a integridade dos sistemas.

5 CONSIDERAÇÕES FINAIS

Este trabalho abordou a importância da implementação de mecanismos de segurança para mitigar ataques de injeção SQL em aplicações web, tema de alta relevância no contexto atual de cibersegurança. O SQL *Injection*, como discutido ao longo do trabalho, representa uma das mais comuns e perigosas vulnerabilidades, permitindo que invasores explorem falhas de segurança para acessar, modificar ou até destruir dados sensíveis.

A análise realizada destacou a necessidade de práticas robustas de segurança durante o desenvolvimento de software, enfatizando que a falta de validação de entradas e o uso inadequado de comandos SQL contribuem para a exposição a esses ataques. Com o estudo das ferramentas XAMPP, SQLMap e Burp Suite, o trabalho evidenciou como o uso de ferramentas de teste e simulação pode ser eficaz na identificação de vulnerabilidades e na proteção das aplicações contra possíveis ameaças.

O desenvolvimento de um ambiente vulnerável foi crucial para simular ataques e identificar os pontos frágeis no código. Esse ambiente possibilitou compreender a eficácia das práticas de proteção, como a parametrização de consultas SQL e o uso de mecanismos de autenticação mais seguros. Ao aplicar essas práticas, torna-se possível reduzir consideravelmente os riscos e aumentar a confiabilidade das aplicações web.

Portanto, o estudo conclui que a proteção contra injeção SQL não depende exclusivamente de ferramentas, mas de uma cultura de segurança que envolve conhecimento técnico, atualização constante sobre ameaças emergentes e uma abordagem preventiva no desenvolvimento. Assim, a pesquisa contribui para a conscientização sobre a importância de práticas de segurança eficazes, sugerindo a necessidade de uma abordagem contínua e integrada de proteção no desenvolvimento de aplicações web.

REFERÊNCIAS

BUCK DE GODOY, L. et al. **Segurança em Aplicações Web: Um estudo do SQL Injection**. [s.l: s.n.]. Disponível em: <https://ric.cps.sp.gov.br/bitstream/123456789/3987/1/20192S_GODOYLeonardoBuckde_OD0772.pdf>. Acesso em: 1 mai. 2024.

IBM i 7.5. Disponível em: <https://www.ibm.com/docs/pt-br/i/7.5?topic=concepts-structured-query-language>. Acesso em: 3 mai. 2024.

Manual de Referência do MySQL 4.1. [s.l: s.n.]. Disponível em: <https://downloads.mysql.com/docs/refman-4.1-pt.a4.pdf>. Acesso em: 11 ago. 2024.

HENRIQUE, J.; DE ALMEIDA, M. **Apostila de PHP com MySQL PHP com MySQL**. [s.l: s.n.]. Disponível em: <https://www.cin.ufpe.br/~ags/2464_php_com_mysql.pdf>. Acesso em 15 ago. 2024.

PORTSWIGGER. **SQL Injection**. Disponível em: <https://portswigger.net/web-security/sql-injection>. Acesso em: 9 nov. 2024.

CENTRO DE TRATAMENTO E RESPOSTA A INCIDENTES CIBERNÉTICOS DE GOVERNO – CTIR Gov. **Alerta CTIR Gov 2018-03: SQL Injection**. Disponível em: <https://www.gov.br/ctir/pt-br/centrais-de-conteudo/publicacoes/alertas/2018/alerta_ctirgov_2018_03_sql_injection.pdf>. Acesso em: 9 nov. 2024.

APACHE FRIENDS. **About XAMPP**. Disponível em: <https://www.apachefriends.org/pt_br/about.html>. Acesso em: 9 nov. 2024.

MYSQL. **What is MySQL?**. Disponível em: <https://dev.mysql.com/doc/refman/9.1/en/what-is-mysql.html>. Acesso em: 9 nov. 2024.

PORTSWIGGER. **Burp Suite Documentation**: <Desktop. Disponível em: https://portswigger.net/burp/documentation/desktop>. Acesso em: 9 nov. 2024.

SQLMAP PROJECT. **Introduction to SQLMap**. Disponível em: <https://github.com/sqlmapproject/sqlmap/wiki/Introduction>. Acesso em: 9 nov. 2024.

IBM. Structured Query Language (SQL) - Concepts. Disponível em: <https://www.ibm.com/docs/pt-br/i/7.5?topic=concepts-structured-query-language>. Acesso em: 9 nov. 2024.