Instituto Federal de Educação, Ciência e Tecnologia de Brasília, Campus Taguatinga,
Campus Taguatinga

MITIGAÇÃO DE ATAQUES SQL INJECTION EM APLICAÇÕES WEB MYSQL: UMA ABORDAGEM PRÁTICA COM PREPARED STATEMENTS, FILTRAGEM DE DADOS E STORED PROCEDURES

Por

EMANUELLY PARREIRA DA SILVA E LUCAS BOMFIM FERNANDES

Trabalho de Graduação

BRASÍLIA/2025

Emanuelly Parreira da Silva e Lucas Bomfim Fernandes

MITIGAÇÃO DE ATAQUES SQL INJECTION EM APLICAÇÕES WEB MYSQL: UMA ABORDAGEM PRÁTICA COM PREPARED STATEMENTS, FILTRAGEM DE DADOS E STORED PROCEDURES

Trabalho apresentado ao Curso de Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia de Brasília, Campus Taguatinga, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Dr. Daniel Saad Nogueira Nunes

BRASÍLIA
2025

Ficha de identificação da obra elaborada pelo bibliotecário
Marcelo José Rodrigues da Conceição (CRB1-2323)

Silva, Emanuelly Parreira da

S586m Mitigação de ataques SQL injection em aplicações Web MySQL: uma abordagem prática com prepared statements, filtragem de dados e stored procedures / Emanuelly Parreira da Silva e Lucas Bomfim Fernandes, 2025.

84 f. : il.

Trabalho de Conclusão de Curso de Bacharelado em Ciência da Computação, Instituto Federal de Educação, Ciência e Tecnologia de Brasília, Campus Taguatinga, 2025.

Orientador: Dr. Daniel Saad Nogueira Nunes.

Inclui referências.

1. SQL (Linguagem de programação de computador). 2. Banco de dados - Programação. 3. Segurança da informação. I. Fernandes, Lucas Bomfim. II. Título. III. Nunes, Daniel Saad Nogueira. IV. Instituto Federal de Educação, Ciência e Tecnologia de Brasília.

CDU 004.655

Trabalho de Graduação apresentado por **Emanuelly Parreira da Silva e Lucas Bomfim Fernandes** ao curso de Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia de Brasília, *Campus* Taguatinga, sob o título **Mitigação de ataques SQL Injection em Aplicações Web MySQL: Uma abordagem prática com Prepared Statements, Filtragem de Dados e Stored Procedures**, orientado pelo **Prof. Dr. Daniel Saad Nogueira Nunes** e aprovado pela banca examinadora formada pelos professores:

gov.br
Documento assinado digitalmente
FABIANO CAVALCANTI FERNANDES
Data: 30/04/2025 17:40:45-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Fabiano Cavalcanti Fernandes
IFB

gov.br
Documento assinado digitalmente
EDWARD DE OLIVEIRA RIBEIRO
Data: 30/04/2025 13:43:41-0300
Verifique em https://validar.iti.gov.br

Me. Edward de Oliveira Ribeiro
Senado Federal

gov.br
Documento assinado digitalmente
DANIEL SAAD NOGUEIRA NUNES
Data: 23/04/2025 23:08:10-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Daniel Saad Nogueira Nunes
IFB

BRASÍLIA
2025

Agradecimentos

Chegar até aqui foi uma jornada desafiadora, mas repleta de aprendizados. Por isso, gostaria de dedicar este espaço para agradecer à minha avó Tania, que, embora não esteja mais aqui para compartilhar este momento comigo, deixou um impacto imensurável na minha vida. Com seu carinho e visão, ela me presenteou com meu primeiro desktop quando eu ainda era criança. Na época, era apenas um presente, mas acabou se tornando o primeiro passo da minha jornada no mundo da tecnologia. Seu apoio e incentivo foram essenciais para que eu me dedicasse a essa área, e por isso, serei eternamente grata.

Toda ciência seria supérflua se houvesse coincidência imediata entre a aparência e a essência das coisas.

—KARL MARX

Resumo

O banco de dados é um componente essencial de sistemas computacionais que interagem com usuários externos, armazenando e manipulando dados sensíveis. Nesse contexto, ataques de *SQL Injection* representam uma ameaça significativa, capazes de comprometer a confidencialidade, integridade e disponibilidade das informações. Este trabalho propõe boas práticas para mitigar ataques *SQL Injection In-band* (*Error-based*, *Union-based* e *Boolean-based*) em aplicações web *MySQL*, com foco em *Prepared Statements*, filtragem de dados e *Stored Procedures*. Para ilustrar essas técnicas, foi desenvolvida a aplicação *Bear Bank*, intencionalmente vulnerável, que permite simular ataques e testar as contramedidas propostas. A aplicação serve como um recurso educativo, permitindo que estudantes e profissionais pratiquem ataques e analisem o comportamento das contramedidas em tempo real. O trabalho destaca a importância de integrar práticas de segurança desde as etapas iniciais do desenvolvimento, seguindo princípios da cultura *DevSecOps*, e conclui que a combinação de técnicas como *Prepared Statements*, filtragem de entrada e *Stored Procedures* é essencial para fortalecer a segurança de aplicações web contra *SQL Injection*.

Palavras-chave: Mitigação de SQL Injection In-band, SQL Injection e Prepared Statements, SQL Injection e filtragem de dados, SQL Injection e Stored Procedures.

Abstract

Databases are an essential component of computational systems that interact with external users, storing and processing sensitive data. In this context, SQL Injection attacks pose a significant threat, potentially compromising the confidentiality, integrity, and availability of information. This study proposes best practices to mitigate SQL Injection In-band attacks (Error-based, Union-based, and Boolean-based) in MySQL web applications, focusing on Prepared Statements, data filtering, and Stored Procedures. To illustrate these techniques, the Bear Bank application was developed as an intentionally vulnerable system, allowing users to simulate attacks and test the proposed countermeasures. The application serves as an educational resource, enabling students and professionals to practice attacks and analyze the behavior of countermeasures in real time. The study highlights the importance of integrating security practices from the early stages of development, following the principles of the DevSecOps culture, and concludes that the combination of techniques such as Prepared Statements, input filtering, and Stored Procedures is essential to strengthening web application security against SQL Injection attacks.

Keywords: SQL Injection In-band mitigation, SQL Injection and Prepared Statements, SQL Injection and data filtering, SQL Injection and Stored Procedures.

Lista de Figuras

Lista de Blocos de Código

Lista de Tabelas

Lista de Acrônimos

Sumário

1

Introdução

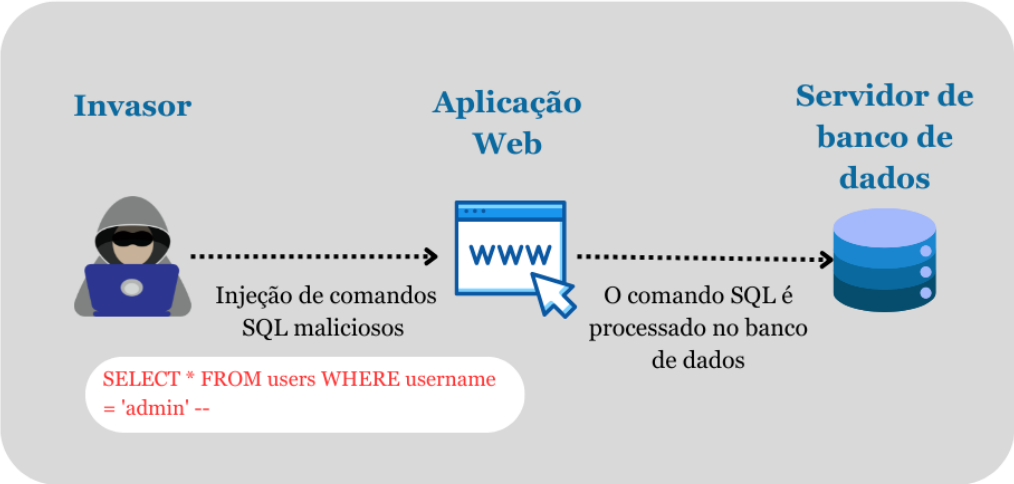
A Segurança da Informação é um campo de estudo amplo e de longa data, cujas técnicas e princípios antecedem o advento da informática, mas que existe e vem se desenvolvendo desde que a sociedade começou a valorizar a proteção de informações sensíveis, onde surgiu a necessidade de garantir a confidencialidade, integridade e disponibilidade desses dados. Esses três princípios, conforme descrito por Hintzbergen em Fundamentos de Segurança da Informação (HINTZBERGEN et al., 2018), são conhecidos como os "princípios críticos de segurança", tal que a confidencialidade diz respeito à proteção dos dados e ao controle de quem pode acessá-los; a integridade assegura que as informações permaneçam inalteradas e autênticas; e a disponibilidade garante que os dados estejam acessíveis quando necessário, mesmo diante de falhas ou ataques. Além desses pilares, destacam-se também os princípios de autenticidade e não repúdio, que asseguram a origem legítima das informações e a impossibilidade de negar sua autoria, respectivamente.

Nos dias de hoje, a segurança da informação se faz mais necessária do que nunca, tendo em vista que vivemos em um mundo cada vez mais digitalizado, onde a informação se tornou um dos ativos mais valiosos, e que por isso as ameaças cibernéticas evoluem constantemente na tentativa de obter dados confidenciais, prejudicar sistemas críticos etc. A Kaspersky, empresa famosa no ramo de desenvolvimento de software para segurança na internet, mantém um site de monitoramento em tempo real de ameaças cibernéticas, e atualmente encontramos o Brasil sendo o 3° país que mais recebeu ataques em 2024 (LAB, 2024). O Brasil também foi sede do maior ataque *hacker* direcionado a uma emissora de televisão, esse ataque consistiu em um ataque do tipo Ataque de Negação de Serviço Distribuído (DDoS) à rede televisiva Record TV, onde todos os dados armazenados no servidor da empresa foram criptografados e vazados para outros canais da internet (CISORADVISOR, 2022), o impacto disso, a princípio, foi o vazamento de planilhas de faturamento e documentos pessoais dos funcionários da empresa, porém é sabido que caso a equipe de TI da empresa não tivesse cópias dos dados armazenados todo o conteúdo do servidor teria sido perdido. Casos assim, servem de indicativo para demonstrar o quão frágil ainda estão nossos sistemas e que precisamos avançar em matéria de segurança da informação.

Entre as diversas ameaças cibernéticas, o SQL *Injection* (SQLi) se destaca pela sua frequência, gravidade e por ter o banco de dados das aplicações como alvo. Considere a Figura

1.1, que ilustra um ataque de Structured Query Language Injection (SQLi) realizado por um invasor em uma aplicação web. Esse ataque demonstra como o SQLi explora vulnerabilidades em aplicações web e bancos de dados, permitindo que invasores executem comandos maliciosos para obter acesso a informações confidenciais, modificar ou excluir dados e até mesmo comprometer a integridade de sistemas inteiros. Em síntese o ataque SQL *Injection* funciona na medida em que o invasor faz requisições indevidas ao Banco de Dados de uma Aplicação Web, usando a linguagem no gerenciamento de Banco de Dados, o Structured Query Language (SQL). Um exemplo relevante ocorreu em 2019, quando a Capital One, um dos maiores bancos dos Estados Unidos da América, sofreu um ataque que expôs dados sensíveis de cerca de 100 milhões de clientes, utilizando SQLi e outras técnicas (KHAN et al., 2023).

Figura 1.1: Exemplo de um fluxo padrão de ataque SQL *Injection In-Band*



Fonte:Autoria própria.

Outro ataque notório foi o comprometimento dos servidores do Yahoo Voices em 2013 (revelado apenas em 2016), quando 450 mil credenciais de usuários foram expostas através de técnicas de SQLi. O grupo *hacker* “D33Ds Company” assumiu a autoria do ataque, e o Yahoo foi multado em 35 milhões de dólares, além de pagar 117,5 milhões em uma ação coletiva (DASWANI, 2021).

Esses exemplos demonstram a urgência de que desenvolvedores compreendam os riscos associados à exposição de aplicações na web e adotem medidas proativas para mitigá-los. Além de conhecer as ameaças, é fundamental que desenvolvam contramedidas eficazes durante o processo de desenvolvimento e manutenção de sistemas, visando reduzir a vulnerabilidade a ataques de SQL *Injection* e outras ameaças cibernéticas.

1.1 Proposta

Este projeto teve como objetivo a elaboração de uma monografia que discute as principais estratégias e técnicas para mitigar ataques de SQL *Injection* do tipo *In-band*. Neste contexto, o termo *In-band* refere-se aos ataques de SQL *Injection* em que o invasor obtém o resultado do ataque pelo mesmo canal utilizado para a sua execução. Serão exploradas estratégias de mitigação aplicáveis tanto ao nível de banco de dados quanto da aplicação, com o intuito de promover a construção de sistemas mais seguros contra as ameaças abordadas.

A seleção das estratégias de segurança terá como foco demonstrar como o desenvolvimento seguro de um banco de dados MySQL, aliado à implementação de medidas de segurança na aplicação pode prevenir vulnerabilidades ao SQL Injection. Além disso, será discutido como uma abordagem de programação diferenciada pode contribuir para a construção de sistemas mais robustos contra esse tipo de ataque.

1.2 Objetivo Geral

O objetivo geral da presente monografia consiste em:

- Mapear um conjunto de boas práticas de Desenvolvimento de Banco de Dados MySQL, assim como técnicas de programação, com o intuito de mitigar ou reduzir os riscos de ataques de SQL *Injection* do tipo *In-band* em aplicações web.

1.3 Objetivos Específicos

Os objetivos específicos do presente trabalho serão:

- Identificar e descrever os principais métodos de SQL *Injection In-band*, elucidando seu funcionamento e analisando os aspectos estruturais das Aplicações Web que os tornam suscetíveis a tais ataques. Além disso, discutir as contramedidas correspondentes, evidenciando como essas técnicas podem fortalecer a segurança de Aplicações Web contra ameaças desse tipo.

- Desenvolver e disponibilizar uma aplicação propositalmente vulnerável, a fim de permitir que estudantes e entusiastas testem na prática as contramedidas descritas. Essa aplicação servirá como um ambiente de aprendizado e prática das técnicas de proteção contra ataques de SQL *Injection*.

2

Fundamentação Teórica

A segurança da informação é um campo multidisciplinar que abrange desde aspectos técnicos, como a configuração de *firewalls* e sistemas de detecção de intrusões, até aspectos humanos, como a conscientização e o treinamento dos usuários (CALDAS; FREIRE, 2013). Com a constante evolução das ameaças cibernéticas e o surgimento de técnicas de ataque cada vez mais sofisticadas, é essencial adotar uma abordagem proativa e integrada, incorporando práticas de segurança em todas as etapas do ciclo de vida dos sistemas de informação. Este capítulo aborda os conceitos fundamentais da segurança da informação, discute as principais ameaças que comprometem a integridade dos ativos informacionais e apresenta os mecanismos de defesa mais eficazes. Além disso, serão introduzidos conceitos críticos para a compreensão de vulnerabilidades específicas, essenciais para a implementação de medidas de proteção contra SQL *Injection*.

2.1 Importância da Segurança da Informação

Em uma era onde a informação é um ativo estratégico para organizações de todos os portes e setores, a segurança da informação desempenha um papel crucial na proteção dos negócios, da reputação e da continuidade das operações. E a falta de investimentos em segurança da informação pode ter consequências graves para as organizações, incluindo perdas de dados, interrupção dos negócios, danos à reputação e penalidades legais (CALDAS; FREIRE, 2013).

2.1.1 Ameaças à Segurança da Informação

As ameaças à segurança da informação são diversas, que podem ser ser classificadas com acidentais ou intencionais, estão em constante evolução, acompanhando o avanço tecnológico e as novas formas de interação com a tecnologia.

Dentre as ameças intencionais temos os ataques cibernéticos, os quais podem ser: **Malware**, que são *software*s maliciosos projetados para infiltrar sistemas e causar danos, roubar dados, espionar atividades ou controlar dispositivos, como vírus, *worms*, cavalos de Troia, *ransomware* e *spyware* (KASPERSKY, 2024); ***Phishing***, que utiliza mensagens fraudulentas, como e-mails ou mensagens instantâneas, para enganar os usuários e induzi-los a fornecer

informações confidenciais; **Ataque de Negação de Serviço (DoS)**, que visam tornar um sistema, servidor ou rede indisponível para os usuários legítimos; e a **Inserção de código malicioso (SQLi)**, que explora falhas em Aplicações Web para inserir código malicioso em bancos de dados (OWASP, 2023).

Já nas ameaças acidentais temos os sobretudo ameaças originadas por falhas humanas. Erros acidentais, como a exclusão ou modificação não autorizada de dados, podem ocorrer por falta de atenção, treinamento inadequado ou procedimentos mal definidos. A Engenharia Social, por sua vez, explora a vulnerabilidade humana através de técnicas de manipulação psicológica para induzir as pessoas a fornecer informações confidenciais, conceder acesso a sistemas ou realizar ações que comprometam a segurança (MITNICK; SIMON, 2003).

Por fim, até mesmo desastres naturais, como incêndios, inundações, terremotos e tempestades, podem representar uma ameaça à segurança da informação, causando danos físicos à infraestrutura de tecnologia da informação, comprometendo a integridade e a disponibilidade dos dados e sistemas.

2.1.2 Medidas de Segurança

Para proteger a informação contra as diversas ameaças, as organizações devem implementar uma combinação de medidas de segurança que abrangem diferentes camadas e aspectos. Essas medidas podem ser categorizadas em controles administrativos, que se referem a políticas, procedimentos e práticas para estabelecer um *framework* de segurança; controles técnicos, que utilizam tecnologias e ferramentas para proteger sistemas, redes e dados; e controles físicos, que visam proteger a infraestrutura física da organização (HINTZBERGEN et al., 2018).

Exemplos de controles administrativos incluem políticas de segurança, que definem regras e diretrizes, e treinamento de conscientização em segurança para educar os usuários (ROCHA LYRA, 2020). Controles técnicos englobam *firewalls* para bloquear acessos não autorizados, *software* antivírus para detectar e remover *malware*, criptografia para proteger dados confidenciais e autenticação forte para verificar a identidade dos usuários (JUNIOR; MOREIRA, 2020). Já os controles físicos incluem medidas como controle de acesso físico para restringir o acesso a áreas sensíveis e sistemas de vigilância para monitorar as instalações.

2.2 SQL (Structured Query Language)

O SQL é uma linguagem de programação de consulta amplamente utilizada para gerenciar e manipular bancos de dados relacionais. Ela foi desenvolvida na década de 1970 pela IBM, como parte do projeto *System R*, onde a intenção era definir um modelo que permitisse um bom desempenho no processamento de transações através da criação de um sistema de gerenciamento de Banco de Dados Relacional, baseado na linguagem Datalog, que na época da criação do SQL era uma linguagem acadêmica de consulta não procedural a banco de dados (Oracle Corporation, 2022). Desde então o SQL tornou-se a linguagem padrão para interação com sistemas de

gerenciamento de banco de dados (SGBDs) relacionais, tais como MySQL, PostgreSQL, SQL Server, Oracle, entre outros.

O sucesso da linguagem SQL se deve a um conjunto de fatores, entre os quais se destacam: sua sintaxe simples, que facilita o aprendizado e uso; a flexibilidade para combinar diferentes tarefas, permitindo a criação de rotinas que automatizam processos no banco de dados; e sua natureza declarativa, onde o programador apenas especifica o que deseja que seja feito, sem precisar se preocupar com os detalhes de como as operações serão executadas (TANIMURA, 2022). Essas características tornam o SQL a linguagem padrão para manipulação de dados em bancos de dados relacionais. Além disso, até mesmo em alguns sistemas NoSQL, o SQL é utilizado para realizar consultas, e em sistemas de Data Warehouse, que armazenam grandes volumes de dados voltados à análise e à tomada de decisões, é comum encontrar uma camada SQL que facilita o acesso aos dados e a geração de relatórios.

A principal funcionalidade do SQL é permitir que os usuários realizem consultas (*queries*) em bancos de dados para recuperar, inserir, atualizar e excluir dados. Essas consultas são executadas por meio de comandos específicos, como SELECT, INSERT, UPDATE e DELETE, que possibilitam a interação com as tabelas e outros objetos do banco de dados. Além das operações principais, o SQL também oferece recursos úteis para a segurança dos dados, como o controle de acesso, a integridade dos dados, transações, funções e procedimentos armazenados, criptografia de dados e controle de acesso baseado em papéis (roles), conforme sintetizado na Tabela 2.1.

Com o crescimento exponencial da utilização do SQL em uma ampla gama de aplicações e sistemas de informação, surgiram desafios significativos de segurança, com o SQL *Injection* destacando-se como um dos mais proeminentes (OWASP, 2021). Nesse contexto, é fundamental adotar medidas de proteção adequadas para garantir a segurança e a integridade dos dados. Assim, nesta monografia, serão discutidas as estratégias e práticas recomendadas para mitigar os riscos associados ao SQL *Injection* e proteger efetivamente os dados sensíveis.

2.3 Sistemas de Gerenciamento de Banco de Dados e MySQL

Um Sistema de Gerenciamento de Banco de Dados (SGBD) é uma peça fundamental no cenário da tecnologia da informação, proporcionando uma infraestrutura para armazenar, organizar e acessar dados de forma eficiente e segura. Ele serve como uma ponte entre os usuários e os dados armazenados, permitindo que os usuários realizem operações como inserção, consulta, atualização e exclusão de dados de maneira simplificada e controlada. Os SGBDs oferecem uma variedade de recursos e funcionalidades, incluindo linguagens de consulta como SQL para manipulação de dados, mecanismos de indexação para otimização de consultas e controle de transações para garantir a consistência dos dados em ambientes concorrentes (ELMASRI; NAVATHE, 2015).

O MySQL, por sua vez, é um dos SGBDs mais populares e amplamente utilizados no

Tabela 2.1: Recursos úteis na construção de um Banco de Dados com uso do SGBD MySQL

Funcionalidade	**Descrição**
Controle de acesso	Regula os níveis de acesso para visualização, exclusão ou modificação de dados em uma ou mais tabelas.
Integridade de dados	Garante a consistência e integridade dos dados armazenados no banco de dados, utilizando restrições como chaves estrangeiras, unicidade e gatilhos (*triggers*).
Transações	Agrupa um conjunto de operações em uma unidade lógica de trabalho, assegurando que todas as operações sejam concluídas com êxito ou revertidas em caso de falha.
Funções e procedimentos armazenados	Blocos de código SQL reutilizáveis ou executáveis no servidor, facilitando a execução de tarefas complexas.
Criptografia de dados	Protege os dados armazenados utilizando algoritmos de criptografia, garantindo a confidencialidade das informações.
Controle de acesso baseado em papéis (*roles*)	Permite atribuir diferentes níveis de acesso a usuários ou grupos de usuários com base em suas funções ou responsabilidades na organização.

mundo. Desenvolvido inicialmente pela MySQL AB e posteriormente adquirido pela *Oracle Corporation*, o MySQL é um SGBD relacional que organiza os dados em tabelas relacionadas. Cada tabela é composta por linhas (registros) e colunas (campos), proporcionando uma estrutura flexível e organizada para armazenar informações de diferentes tipos. A popularidade do MySQL se deve principalmente por ser um sistema de código fonte aberto, em grande parte à sua confiabilidade, desempenho e facilidade de uso. Ele é amplamente adotado em uma variedade de cenários, desde pequenos sites e aplicativos da web até sistemas corporativos de grande escala (ORACLE, 2024a), como Bradesco, HP, Sony e etc.

2.3.1 Funcionalidades e Benefícios do MySQL

O MySQL oferece uma série de funcionalidades que o tornam uma escolha robusta para diversos tipos de aplicações (CLOUD, 2024):

- **Desempenho:** MySQL é conhecido por seu alto desempenho, capaz de lidar com grandes volumes de dados e muitas transações simultâneas.

- **Escalabilidade:** Suporta desde pequenas aplicações até grandes sistemas corporativos, com capacidade de escalar horizontal e verticalmente.

- **Segurança:** Implementa robustos mecanismos de segurança, incluindo controle de acesso, criptografia e auditoria de logs.

- **Fácil Integração:** Compatível com diversas linguagens de programação e plataformas, facilitando a integração com outras ferramentas e sistemas.

- **Comunidade Ativa:** Uma grande comunidade de usuários e desenvolvedores contribui para o contínuo aprimoramento e suporte.

2.4 Inserção de Código Malicioso

A inserção de código malicioso, frequentemente referida pelo termo em inglês SQL *Injection*, é uma técnica de ataque cibernético amplamente explorada por invasores para comprometer a segurança de sistemas que interagem com bancos de dados (OWASP, 2021). Segundo a OWASP, ataques de SQL *Injection* estiveram em terceiro lugar no ranking dos principais riscos de segurança na Web, indicando a relevância e persistência dessa ameaça nos dias atuais.

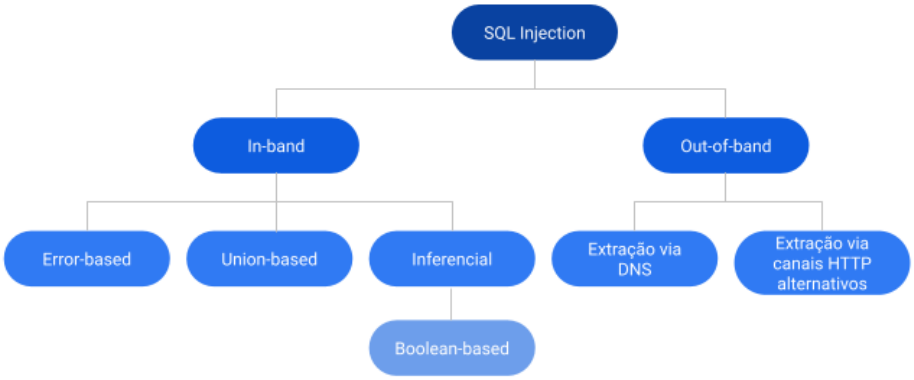
O objetivo principal dessa técnica é a inserção de código malicioso, geralmente na forma de comandos SQL, em campos de entrada de dados de um Aplicativo Web, com o intuito de manipular as consultas SQL executadas pelo sistema e, consequentemente, realizar ações não autorizadas no banco de dados (NASEREDDIN et al., 2023).

Existem pelo menos duas maneiras de classificar ataques de SQL *Injection*, que é o SQLi do tipo *In-band* e o do tipo *Out-of-Band*. No ataque *In-band*, a comunicação e a execução do ataque ocorrem através do mesmo canal, sem a necessidade do uso de um *software* ou plataforma intermediária. Já no ataque *Out-of-Band*, a comunicação e a execução ocorrem em canais diferentes, geralmente utilizando um canal secundário para extrair os dados resultantes do ataque (NASEREDDIN et al., 2023). Há também uma terceira classificação referente aos ataques do tipo Inferencial (ou *Blind SQL Injection*), que se caracteriza pela execução "às cegas". Nesses ataques, o invasor baseia-se em tentativa e erro, a partir de uma analise do comportamento da aplicação web para inferir informações. No entanto, conforme ilustrado na Figura 2.1, neste trabalho iremos tratar os ataques inferenciais como parte dos ataques *In-band*, uma vez que demonstraremos seu funcionamento utilizando técnicas *In-Band*.

2.4.1 Exemplo de Vulnerabilidade

Para ilustrar a vulnerabilidade explorada pela injeção de código malicioso, considere um sistema de *login* desenvolvido em Python. Esse sistema recebe o nome de usuário e a senha através de um formulário web e utiliza esses dados para construir uma consulta SQL que

Figura 2.1: Taxonomia de ataques SQL *Injection*, com a classificação dos tipos de ataques SQL *Injection* adotada nesta monografia.



Fonte: Autoria própria.

verifica a autenticidade do usuário. Um exemplo de código que implementa essa funcionalidade é apresentado a seguir:

```
nomeUsuario = request.form.get("nomeUsuario")
senha = request.form.get("senha")

sql = f"SELECT * FROM Usuarios WHERE nomeUsuario = '{nomeUsuario}'
AND senha = '{senha}'"

# Executa a consulta SQL
cursor.execute(sql)
```

Bloco de Código 2.1: Exemplo didático de Vulnerabilidade a SQLi

Em um cenário de ataque, um invasor pode explorar a falta de validação e tratamento adequados dos dados de entrada fornecidos pelo usuário. Se o invasor inserir no campo “nomeUsuario“ o valor “admin’–“, a consulta SQL resultante se torna:

```
sql = f"SELECT * FROM Usuarios
WHERE nomeUsuario = 'admin'--'
AND senha = '{senha}'"
```

Bloco de Código 2.2: Exemplo didático 1 de ataque SQLi Error-based

Neste caso, a sequência de caracteres “–“ representa um comentário em SQL, incluindo MySQL (ORACLE, 2024a), o que causa o descarte de todo o código subsequente na consulta. Consequentemente, a condição de verificação da senha se torna irrelevante, permitindo que o invasor se autentique como “admin“ sem o conhecimento da senha correta.

2.4.2 Impacto dos Ataques

As implicações de um ataque bem-sucedido de injeção de código malicioso podem ser devastadoras, comprometendo a confidencialidade, integridade e disponibilidade dos dados e sistemas afetados (BASTOS; CAUBIT, 2009). Alguns dos principais impactos incluem:

- **Vazamento de dados confidenciais:** Invasores podem obter acesso não autorizado a dados sensíveis, como informações pessoais de clientes, dados financeiros, senhas, propriedade intelectual e outras informações confidenciais (COELHO; ARAÚJO; BEZERRA, 2014).

- **Alteração indevida de informações:** A injeção de código malicioso pode permitir que invasores modifiquem ou excluam dados armazenados no banco de dados, comprometendo a integridade do sistema e a confiabilidade das informações (COSTA, 2020).

- **Interrupção dos serviços:** Ataques podem causar a indisponibilidade de sites, sistemas de missão crítica, Aplicações Web e outros serviços online, resultando em perdas financeiras, interrupção das operações e danos à reputação da organização (COSTA, 2020).

- **Comprometimento da integridade do sistema:** Em casos extremos, a injeção de código malicioso pode permitir que invasores assumam o controle total do sistema, instalem programas maliciosos *(malware)*, utilizem o sistema como plataforma para outros ataques ou até mesmo causem danos irreparáveis à infraestrutura de tecnologia da informação da organização (COSTA, 2020).

2.4.3 Cenários de Ataque

A injeção de código malicioso pode ser explorada em uma ampla variedade de cenários e aplicações que envolvam a interação com um banco de dados. Alguns dos cenários mais comuns incluem:

- **Formulários de *Login*:** Campos de entrada de dados para nome de usuário e senha são alvos frequentes de ataques de injeção de código malicioso, como demonstrado no exemplo anterior (INVICTI, 2024).

- **Campos de Pesquisa:** Invasores podem inserir código malicioso em campos de pesquisa de sites e Aplicações Web para manipular as consultas SQL e obter acesso a dados não autorizados, como listas de usuários, informações confidenciais ou registros do banco de dados (ATLASSIAN, 2024).

- **URLs:** Parâmetros presentes na URL, como aqueles utilizados para identificar recursos específicos, podem ser manipulados para realizar ataques de injeção de código malicioso (ORACLE, 2024b).

- **Campos de entrada de dados em geral:** Qualquer campo de entrada de dados que seja utilizado para construir ou influenciar a construção de consultas SQL pode ser um alvo em potencial para ataques de injeção de código malicioso.

2.4.4 Medidas de Prevenção

A prevenção de ataques de injeção de código malicioso é crucial para garantir a segurança e a integridade de sistemas que interagem com bancos de dados. A implementação de medidas de segurança abrangentes, desde o desenvolvimento até a implantação e manutenção dos sistemas, é essencial para reduzir o risco de ataques. Algumas das principais medidas de prevenção incluem:

- **Validação e sanitização rigorosas da entrada do usuário:** É fundamental validar e sanitizar todos os dados recebidos do usuário antes de utilizá-los em consultas SQL (OWASP, 2021). Isso inclui verificar o tipo de dado, tamanho, formato e remover caracteres especiais que possam ser interpretados como comandos SQL.

- **Uso de consultas parametrizadas ou declarações preparadas:** Consultas parametrizadas ou declarações preparadas são técnicas de programação que separam os dados das instruções SQL. (ELMASRI; NAVATHE, 2015). Essa separação impede que dados fornecidos pelo usuário sejam interpretados como comandos SQL, prevenindo a injeção de código malicioso.

- **Aplicação de práticas de segurança em todas as camadas da aplicação:** Assegurar que todas as camadas da aplicação, desde o *front-end*, passando pelas camadas de lógica de negócio até a camada de acesso a dados, estejam protegidas contra ataques é fundamental para garantir a segurança do sistema como um todo.

- **Atualização constante do sistema e *softwares*:** Manter o sistema operacional, o sistema de gerenciamento de banco de dados (SGBD) e todos os *softwares* relacionados atualizados com as últimas correções de segurança é crucial para mitigar vulnerabilidades conhecidas e reduzir o risco de exploração.

- **Firewall de Aplicação Web (WAF):** Um WAF atua como uma barreira de proteção entre a aplicação web e o mundo externo, filtrando tráfego malicioso, identificando padrões de ataque e bloqueando ataques de injeção de código malicioso, entre outras ameaças.

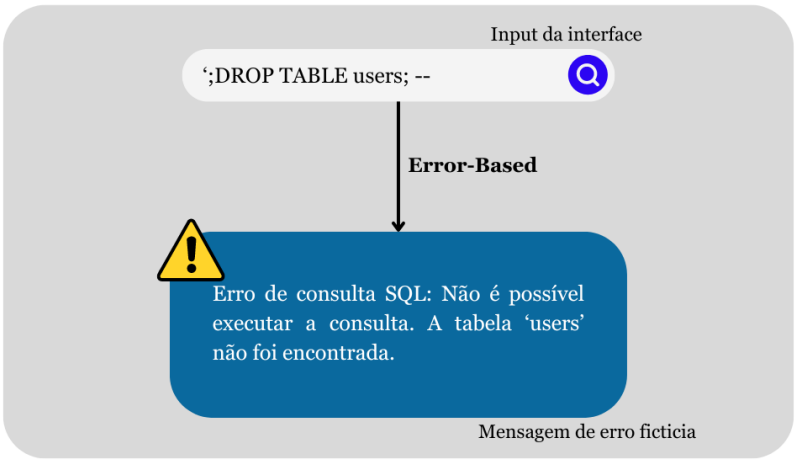
3

Ataques de Injeção de código SQL

3.1 Ataque SQLi *Error-based*

O ataque de Injeção SQL baseada em erro ou *Error-based* se caracteriza pela utilização de técnicas que exploram as mensagens de erro fornecidas pelo banco de dados. Em outras palavras, esse ataque busca inserir comandos SQL em uma interface de usuário que está comunicando com um servidor de Banco de Dados. Onde a intenção do invasor é fazer com que os comandos SQL inseridos gerem erros no Banco de Dados e esses erros sejam exibidos na interface. Dependendo de como foi feito o tratamento de erros do Banco de dados ou da ausência de tratamento, os erros exibidos podem ser reveladores acerca da estrutura do Banco de Dados, ou seja, de sua organização de tabelas, construção de consultas SQL, etc. É importante pontuar que o ataque *Error Based* por si só costuma não ser o suficiente para de fato obter informações privadas ou confidenciais do banco de dados, mas esse ataque é utilizado em conjunto com outros tipos de ataque (JAHANSHAHI; DOUPé; EGELE, 2020), pois ele pode proporcionar o conhecimento da estrutura do Banco de Dados, e pode servir para "refinar outros ataques SQLi"(OWASP, 2023), tendo em vista que é possível descobrir quais tabelas estão mais vulneráveis, qual é a sintaxe aceita pelas consultas SQL, a versão do Sistema de gerenciamento de banco de dados (SGBD), os tipos dos dados armazenados e afins, conforme a Imagem 3.1 exemplifica.

Figura 3.1: Exemplo ilustrativo de ataque SQLi *Error-Based*.



Fonte: Autoria própria

Há diversas técnicas que podem ser empregadas para forçar um erro na requisição feita pela aplicação ao seu Banco de Dados, tais como:

- **Injeção de caracteres reservados da linguagem:** Nessa técnica se utiliza de certos caracteres especiais da linguagem de programação do Banco de Dados para interromper a realização de consultas ou requisições ao Banco de Dados (BD). Isso ocorre porque na criação do banco de dados é usada uma estrutura especifica de construção de consultas SQL, e ao ser feita uma requisição que utiliza caracteres especiais que não eram esperados por aquela consulta SQL um erro é gerado.

  Os caracteres mais utilizados são aqueles reservados para a declaração de *strings* (por exemplo, as aspas simples (') ou aspas duplas (")), pois a sintaxe da linguagem SQL, utilizada por um conjunto específico de caracteres. A presença de um desses caracteres a mais pode gerar um erro de sintaxe (ORACLE, 2024a).

  Outros caracteres também são relevantes para essa técnica, como o ponto-e-vírgula (;), pois, no *MySQL* ele costuma representar o final de consultas SQL e instruções. Sua presença em requisições pode fazer fazer com que tudo o que vier após o ponto-e-vírgula seja ignorado em um primeiro momento, mas ainda assim tenham sua validade checada pelo Banco de Dados. Isso pode permitir que um comando SQL inserido após o ";"seja executado pelo BD.

  Veja o exemplo abaixo, onde o comando injetado seria capaz de verificar se existe um atributo específico em alguma tabela possivelmente existente e acessível. No comando é inserido uma aspa simples a mais em uma *string*, o que irá gerar um erro que pode informar a existência do atributo 'nome':

```
1    SELECT *
2    FROM Usuarios
```

```
3     WHERE nome = 'Lucas'
4     OR nome = 'Jose'';
```

Bloco de Código 3.1: Exemplo 2 de ataque SQLi Error-based, com uso de caracter reservado

- **Injeção de consultas booleanas utilizando um registro em específico:** Na linguagem SQL os comandos de consulta podem ser construídos de forma que, após uma busca nas tabelas determinadas, retornem NULL ou NOT NULL. Quando a pesquisa resulta em NOT NULL, o retorno pode ser a própria informação que estava sendo buscada, seja ela uma *string*, um número inteiro, etc.(ORACLE, 2024c).

  Por esse motivo, é possível realizar operações booleanas nas consultas SQL, ou seja relacionar logicamente duas ou mais consultas utilizando operadores como AND, NOT, OR ou XOR. Essas operações retornam TRUE ou FALSE, dependendo do resultado de cada consulta (NOT ou NOT NULL).

  Dessa forma, a técnica de injeção de consultas lógicas utilizando um registro específico que explora o comportamento booleano do Banco de Dados para confirmar a existência de um registro ou de uma tabela. Um invasor mal intencionado pode utilizar proposições lógicas que sempre resultam em verdadeiro (por exemplo, 1=1) em consultas SELECT, INSERT, UPDATE ou DELETE para verificar se uma outra proposição incluída na consulta é verdadeira ou falsa.

  Veja o exemplo de comando que pode ser injetado para descobrir se o registro "José"existe e se a tabela "Usuarios"é acessível e existente:

```
1     SELECT * FROM Usuarios
2     WHERE nome = 'Jose' OR 1=1;
```

Bloco de Código 3.2: Exemplo 3 de ataque SQLi Error-based, com uso de variáveis booleanas

3.2 Ataque SQLi *Union-based*

O método de Injeção SQL Union-based explora o operador UNION da linguagem MySQL, que permite combinar os resultados de duas ou mais consultas em um único conjunto de dados (ORACLE, 2024b). Essa técnica é especialmente perigosa porque possibilita a fusão de tabelas distintas, permitindo que um invasor recupere dados de diferentes fontes em uma única resposta. Por exemplo, ao unir uma tabela de usuários com uma tabela de informações financeiras, um ataque bem-sucedido pode revelar não apenas credenciais de login e senha, mas também dados como *e-mails*, números de telefone e até informações bancárias, aumentando significativamente o impacto da violação de segurança.

Dito isso, o ataque *Union-Based* atua principalmente em combinar consultas válidas com consultas feitas de maneira "cega", ou seja, através de inferências, para obter acesso a

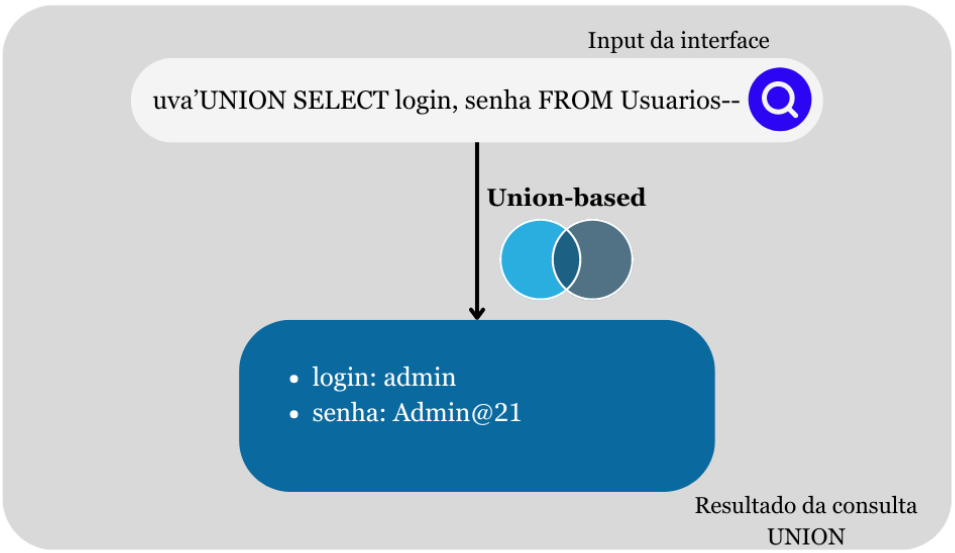
informações sensíveis ou executar ações não autorizadas. No exemplo hipotético abaixo seria possível obter o nome do usuário, telefone e endereço, bem como o número do cartão de crédito de um usuário específico:

```
1  SELECT nome, telefone, endereco FROM Usuarios
2  WHERE Id=1 UNION ALL SELECT numeroCartao,1,1
3  FROM InfosPagamento;
```

Bloco de Código 3.3: Exemplo de ataque SQLi Union-based

Perceba que essa técnica de ataque SQLi requer informações prévias para ser bem sucedida, a saber é necessário conhecer tanto o número de colunas das tabelas que farão parte da consulta quanto o tipo de dado a ser buscado (ATLASSIAN, 2024), isso porque o procedimento UNION exige que as duas tabelas a serem unidas tenham a mesma quantidade de colunas para poder combinar os resultados de cada consulta, bem como, os tipos de dados a serem consultados precisam ser compatíveis entre si, ou seja, os valores dos dados precisam ser representados de uma mesma maneira, valores inteiros, ou *strings*, ou datas. Por esse motivo, essa técnica também costuma ser usada em conjunto com outras técnicas SQLi que podem fornecer as informações necessárias para o sucesso do ataque baseado em UNION, como ilustrado na Imagem 3.2.

Figura 3.2: Exemplo ilustrativo de ataque *Union-Based* onde são combinadas uma consulta válida e outra inválida.



Fonte: Autoria própria

3.3 Ataque SQLi *Boolean-based*

Como explicado nos subtópicos anteriores, o ataque *Union-based* requer conhecimentos prévios a respeito de algumas especificações das tabelas do Banco de Dados alvo, e no caso

do ataque *Error-based* se faz necessário que não tenha sido feito nenhum tratamento de erros, de modo que a página web retorne os erros específicos gerados em cada tentativa de consulta. Por outro lado, no ataque *Boolean-based* poucas ou nenhuma informação prévia é necessária, pois este ataque funciona seguindo um comportamento inferencial (INVICTI, 2024), ou seja, o *hacker* mal intencionado faz uma sequência de tentativas de consultas, verificando se a consulta feita é válida (TRUE) ou não válida (FALSE), para isso os invasores exploram os operadores lógicos das consultas SQL.

É interessante apontar que esse comportamento inferencial é bastante comum em ataques do tipo SQLi, onde para que o ataque seja bem sucedido se faz necessário um processo de tentativa e erro para mapear os identificadores usados na criação do banco de dados que está sendo alvo de uma tentativa de ataque (SCHOENBORN; ALTHOFF, 2021).

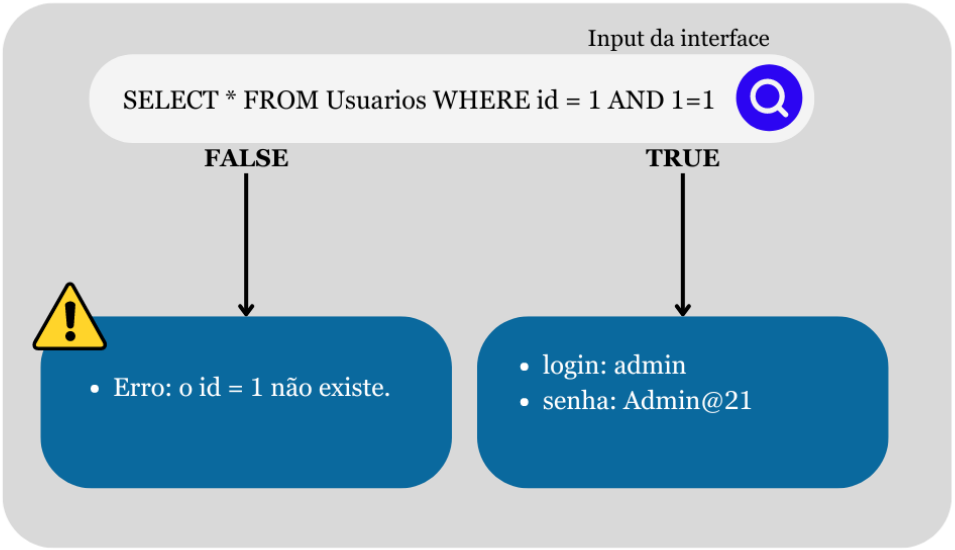
Como mencionado no Subtópico 3.1.1, os operadores AND, OR e XOR são alguns dos operadores lógicos do SQL, e eles se comportam fazendo comparações entre duas afirmações, retornando o valor booleano TRUE ou FALSE conforme for a relação existente entre as duas afirmações comparadas.

Tabela 3.1: Operadores lógicos do *MySQL*

Nome	**Descrição**
AND	Operador lógico que compara se a statement 1 **E** statement 2 são verdadeiras ou falsas.
OR	Operador lógico que compara se a statement 1 **OU** a statement 2 são verdadeiras ou falsas.
XOR	Operador lógico que compara se **apenas** a statement 1 **ou** a statement 2 é verdadeira (esse operador também é chamado de OU exclusivo).

Posto isso, o ataque *Boolean-based* consiste em inferências a respeito de dados e tabelas do Banco de Dados, a inferência feita pode ser verdadeira ou falsa, e guiado por esse retorno TRUE ou FALSE é possível descobrir informações relevantes que podem ser aplicadas em outros tipos de ataque SQLi e aumentar a probabilidade do ataque ser bem sucedido. Veja o exemplo da Figura 3.3, onde o usuário mal intencionado infere a existência de um usuário cujo número Identificador é armazenado em variáveis do tipo Inteiro e é igual a 1, bem como infere a existência de uma tabela de nome Usuário que está passível a consultas indevidas, um retorno TRUE, por exemplo, serviria para confirmar a existência de todas essas informações que podem ser usadas em outros ataques:

Figura 3.3: Exemplo ilustrativo de ataque *Boolean-Based*



Fonte: Autoria própria

4

Revisão da Literatura

Esta revisão da literatura tem como objetivo oferecer uma visão abrangente das pesquisas atuais sobre a prevenção de SQL *Injection*, abordando tanto estudos utilizados exclusivamente para compreensão do cenário acadêmico quanto trabalhos que contribuíram diretamente para o desenvolvimento deste estudo. Dessa forma, este capítulo explora os artigos que permitiram mapear as abordagens atuais, identificar desafios e entender os avanços na área, formando a base necessária para a análise e a construção de soluções mais robustas em segurança para Aplicações Web. No capítulo 5 de Materiais e Métodos, serão especificados os estudos que contribuíram diretamente para o desenvolvimento deste trabalho.

4.1 Técnicas e critérios para Revisão da Literatura

- **Fontes de pesquisa e palavras-chave:** A pesquisa foi realizada em bases de dados acadêmicas como IEEE Xplore, ACM Digital Library, ScienceDirect e Springer, com apoio do Google Scholar, utilizando as seguintes palavras-chave: "SQL injection survey"e “SQL Injection mitigation”.

- **Estratégia de busca e critérios de inclusão e exclusão:** Em um primeiro momento foram selecionados artigos publicados em periódicos acadêmicos e conferências de tecnologia, com foco em promover o conhecimento acerca dos tipos de ataque SQLi *In-Band*, e com foco na mitigação e prevenção desse ataque em aplicações web. Foram ignorados artigos que não estivessem escritos em português ou inglês, que tenham sido publicados antes de 2010 e que utilizem unicamente ferramentas de inteligência artificial para detectar e mitigar ataques de SQLi.

- **Extração de Dados:** A partir dos artigos selecionados, foram extraídos dados sobre os tipos de ataques, vulnerabilidades, técnicas de prevenção e suas eficácias, além de informações sobre as metodologias de pesquisa empregadas, utilizando a documentação oficial do MySQL e o site da OWASP como apoio. Bem como foram selecionados artigos que demonstrassem técnicas atuais de prevenção ao SQL *Injection*, de modo a tornar a pesquisa o mais completa possível acerca do assunto.

4.1.1 Análise dos dados

- **Análise dos Tipos de Ataques:** Foram analisados os mecanismos de funcionamento dos ataques *In-Band Error-based, Union-based e Boolean-based*, identificando suas diferenças e similaridades, além de seus impactos em sistemas MySQL.

- **Análise das Vulnerabilidades:** A pesquisa identificou as vulnerabilidades do MySQL que podem ser exploradas por ataques de Injeção SQL, incluindo a falta de validação de entrada, a utilização de consultas SQL dinâmicas e a ausência de mecanismos de segurança adequados.

- **Análise das Técnicas de Prevenção:** A revisão sistemática comparou diferentes técnicas de prevenção de injeção SQL, avaliando seus prós e contras, e sua aplicabilidade em bancos de dados MySQL.

4.1.2 Condução do mapeamento sistemático:

- **M**apeamento sob a palavra-chave “SQL Injection mitigation” : A partir das buscas no Google Scholar foram encontrados cerca de 50 artigos sobre técnicas de mitigação de ataques de SQL *Injection*, sob a palavra chave “SQL Injection mitigation”. A partir desse resultado ordenamos os resultados de acordo com sua relevância e 35 foram eliminados por conterem no título menções ao uso de outros SGBDs como Oracle DB, também foram eliminados artigos que não fossem escritos em inglês ou português. A partir da leitura da introdução dos 12 artigos restantes foram selecionados 5 artigos que serviram de fundamentação para as técnicas de mitigação abordadas no presente documento.

- **M**apeamento sob a palavra-chave "SQL injection survey": Foi feita uma busca no Google Scholar que resultou em 26.400 resultados. Inicialmente, foram ordenados por relevância e considerados apenas as 5 primeiras páginas, totalizando 50 resultados. Considerando a relevância e disponibilidade dos artigos, foram selecionados 4 artigos, a saber os documentos de título ’A survey on the detection of SQL injection attacks and their countermeasures’ e ’Detection and Prevention of SQL Injection Attack: A Survey’.

Por fim, no próximo tópico serão apresentadas as sínteses dos artigos selecionados, de modo a justificar de que modo os trabalhos selecionados contribuem para a presente monografia.

4.2 Síntese dos trabalhos selecionados

1. **"Identification and Mitigation tool for SQL Injection attacks (SQLia)":** Este trabalho apresenta a ferramenta WebVIM para identificar e mitigar vulnerabilidades

de Injeção SQL (SQLi) em aplicações web. A ferramenta usa parametrização de *queries* e sanitização do *input* do usuário.

- **Parametrização de *Queries*:** Separa comandos SQL dos dados, tratando-os como valores literais.
- **Sanitização do *Input*:** Modifica a entrada para remover ou alterar caracteres maliciosos, usando uma "base de padrões"(*pattern base*).

A WebVIM identifica vulnerabilidades injetando *payloads* SQLi em campos de entrada e, se bem-sucedida, aplica as técnicas de mitigação. O artigo afirma cem por cento de precisão, mas a análise crítica aponta limitações:

- **Falta de Detalhes Técnicos:** Não detalha como a "base de padrões"é construída e atualizada, como lida com ofuscação, nem como o código da aplicação é modificado.
- **Avaliação Limitada:** Não especifica quantas/quais aplicações foram testadas, quais tipos de ataque SQLi foram considerados, e a comparação com outras ferramentas é superficial.
- **Foco em PHP:** A ferramenta é testada apenas em aplicações PHP, limitando a generalização.

Embora a WebVIM mostre potencial, a falta de detalhes e a avaliação limitada impedem uma conclusão definitiva sobre sua eficácia.

2. **A Design Review:concepts for mitigating SQL injection attacks:** Consiste em uma revisão sistemática da literatura acerca de quatro métodos diferentes de mitigação para ataques SQL *Injection* a Aplicações Web, o documento destaca também a responsabilidade dos desenvolvedores de integrarem técnicas de segurança durante o processo de desenvolvimento da aplicação. As táticas abordadas no artigo são:

   - **Método Shell:** Foca em aprimorar a segurança das consultas SQL por meio da validação de caracteres e proteção do campo de dados (*input* do usuário).
   - **Método Prevenção Multi-Level:** Consiste em uma estratégia abrangente de medidas de segurança, incluindo análise estática para construir um fluxo de controle para determinar o *SQL-graph*, representação do *SQL-graph* para reduzir a sobrecarga de verificação em tempo de execução, instrumentalização para isolar a entrada do usuário das declarações SQL nativas, e análise em tempo de execução para comparar as entradas do usuário e garantir a validade da declaração SQL.

- **Proteção Automatizada com *Stored Procedures*:** Utiliza um sistema automatizado para comparar as entradas de usuário com as declarações SQL nativas.
- **Abordagem HCI-SQLIAM2:** Defende a integração de medidas de segurança ao longo do processo de desenvolvimento da aplicação.

Embora o artigo forneça um bom panorama das técnicas de mitigação, incluindo sua própria proposta (HCI-SQLIAM2), e reforce a importância do design seguro, ele também apresenta algumas limitações importantes:

- **Superficialidade na Descrição dos Métodos Existentes:** A descrição dos três primeiros métodos (*Shell*, Multinível e *Stored Procedures*) é bastante resumida. O artigo se beneficiaria de uma análise mais detalhada de cada um, incluindo exemplos concretos de como são implementados, suas vantagens e desvantagens específicas, e sua eficácia contra diferentes tipos de ataques SQLi.
- **Detalhes Insuficientes sobre a Abordagem HCI-SQLIAM2:** Embora a abordagem proposta (HCI-SQLIAM2) seja o foco principal do artigo, sua descrição é relativamente vaga. O artigo menciona o uso de *sketching*, prototipagem e avaliação cognitiva, mas não fornece detalhes concretos sobre como essas técnicas são aplicadas para prevenir SQLi. Seria útil apresentar exemplos de designs problemáticos e como a abordagem HCI-SQLIAM2 os corrigiria.

O artigo faz uma contribuição relevante ao propor uma abordagem focada no design para mitigar SQLi, mas a falta de detalhes sobre a implementação e a ausência de evidências empíricas limitam a avaliação de sua eficácia prática.

3. **"SQL Injection attacks countermeasures assessments":** Este trabalho também consiste em uma revisão sistemática da literatura que faz uma breve análise sobre os diferentes tipos de ataque de SQLi, bem como mapeia as principais abordagens usadas para mitigar e prevenir os ataques, apontando os pontos fracos e fortes das ferramentas que podem ser utilizadas no combate às vulnerabilidades das Aplicações Web. As técnicas mencionadas no artigo são:

   - Transformação de código
   - Randomização de Palavras chaves
   - Consulta parametrizada
   - Uso de sistemas de teste e ataque automatizado

- Testes confiáveis

Embora o artigo forneça uma visão geral útil das técnicas de mitigação de SQLi, é importante considerar algumas limitações e aprofundar a análise crítica:

- **Foco Limitado em Tipos de Sistemas:** A categorização dos sistemas ("*toy*", "*limited-scale*", "*large-scale*") é útil, mas simplifica a complexidade dos sistemas reais. Muitos sistemas modernos são híbridos e complexos, e a eficácia das técnicas pode variar significativamente nesses ambientes.
- **Dependência de Avaliações Existentes:** A análise se baseia em avaliações reportadas em outros estudos, o que pode introduzir vieses e imprecisões, já que cada estudo pode ter usado metodologias e conjuntos de testes diferentes.

4. **"A Survey on the Detection of SQL Injection Attacks and Their Countermeasures":** Neste artigo é apresentada uma análise abrangente sobre diferentes tipos de ataques de Injeção SQL e as principais abordagens para mitigar e prevenir tais ataques. Ele detalha métodos e ferramentas utilizados na detecção e prevenção de ataques SQLi, explorando as vantagens e desvantagens de cada técnica. Algumas das principais abordagens discutidas no *survey* incluem:

   - Defesa codificada: Esta abordagem envolve o uso de técnicas de programação segura para evitar que dados de entrada do usuário sejam incorporados diretamente em consultas SQL.
   - Criptografia e ofuscação: O objetivo aqui é proteger as consultas SQL com técnicas de criptografia e ofuscação para tornar mais difícil para os atacantes entenderem e explorarem as consultas.
   - Vinculação estática e dinâmica: Essas técnicas visam evitar que os atacantes manipulem a estrutura das consultas SQL, garantindo que os dados de entrada do usuário sejam tratados como valores literais.
   - Monitoramento de tempo de execução: Essa abordagem monitora a execução de aplicações web em tempo real para detectar atividades suspeitas que podem indicar um ataque de Injeção SQL.
   - Análise estática e dinâmica: Técnicas de análise estática e dinâmica são usadas para identificar vulnerabilidades de Injeção SQL no código fonte e em tempo de execução.

   O artigo também destaca várias ferramentas e abordagens específicas, como:

   - SQLIMW: Um middleware que protege contra ataques de injeção SQL usando uma barreira de segurança dupla.

- TransSQL: Uma solução que traduz e valida consultas SQL para detectar injeções maliciosas.
- CANDID: Um sistema que avalia dinamicamente a intenção do desenvolvedor ao construir consultas SQL.
- WAVES: Um *framework* de análise de segurança de aplicações web que usa técnicas de injeção de falhas e monitoramento de comportamento para detectar vulnerabilidades.

Em síntese, essa pesquisa fornece uma visão geral completa sobre as técnicas existentes para a detecção e prevenção de ataques de SQLi. No entanto, é importante ressaltar algumas limitações:

- **Falta de Comparação Quantitativa Rigorosa:** Embora a Tabela 4 forneça uma comparação qualitativa das diferentes abordagens, o artigo carece de uma análise quantitativa mais aprofundada. Não são apresentados dados concretos sobre o *overhead* (impacto no desempenho) de cada técnica, suas taxas de detecção (quantos ataques são corretamente identificados) e taxas de falsos positivos (quantas vezes a técnica identifica incorretamente uma operação legítima como um ataque). Essa análise quantitativa seria crucial para avaliar a eficácia prática das abordagens em cenários realistas e sob diferentes cargas de trabalho.
- **Evolução Constante das Ameaças:** O cenário de segurança cibernética é dinâmico e está em constante evolução. Os ataques de SQLi se tornam cada vez mais sofisticados, e novas vulnerabilidades são descobertas com frequência. Como o artigo foi publicado em 2017, é possível que ele não reflita completamente o estado da arte atual em termos de técnicas de ataque e defesa. Novas abordagens, como aquelas baseadas em aprendizado de máquina para detecção de anomalias, podem ter surgido ou se desenvolvido significativamente desde a publicação do artigo.

Apesar dessas observações, o artigo oferece um panorama valioso e informativo sobre as técnicas e ferramentas disponíveis para combater a Injeção SQL, destacando a importância de uma abordagem multicamadas para a segurança de aplicações web.

5. **"Detection and Prevention of SQL Injection Attack: A Survey":** O artigo realiza uma revisão sistemática da literatura sobre ataques de injeção SQL (SQLI), explorando diversos tipos de ataques e as principais abordagens para detectá-los e preveni-los. O estudo analisa as vantagens e desvantagens de diferentes técnicas, ferramentas e estratégias de proteção, incluindo:

- **Técnicas de programação segura:** O artigo destaca a importância da validação de entrada e outras práticas de codificação segura para evitar que dados de entrada do usuário sejam inseridos diretamente em consultas SQL.
- **Análise de código:** Diversas ferramentas e abordagens são apresentadas para identificar vulnerabilidades de SQLi no código fonte, utilizando técnicas de análise estática e dinâmica.
- **Consultas parametrizadas:** O artigo analisa a utilização de instruções preparadas (*Prepared Statements*) e o PDO (PHP Data Object) como mecanismos eficazes para evitar que os atacantes manipulem a estrutura das consultas SQL.
- **Monitoramento e detecção:** São exploradas técnicas de monitoramento em tempo de execução e sistemas de detecção de intrusões para identificar atividades suspeitas que podem indicar um ataque de Injeção SQL.
- **Sistemas de teste automatizados:** A pesquisa discute a aplicação de ferramentas de teste automatizadas para simular ataques de SQLi e avaliar a segurança de aplicações, ajudando a identificar e corrigir vulnerabilidades.

O artigo também aborda os tipos modernos de ataques de injeção SQL, incluindo ataques combinados com outras técnicas como XSS, DDoS e *Hijacking* de DNS, destacando as ferramentas e estratégias específicas para lidar com essas ameaças.

Além disso, é importante pontuar que o trabalho fornece uma visão abrangente sobre vulnerabilidades exploradas pelos ataques de SQL *Injection*, discutindo várias abordagens para prevenir o ataque, enfatizando o uso de boas práticas de programação como a Parametrização das consultas. Mas por outro lado, o trabalho não inclui nenhum estudo de caso, o que faz com que o trabalho deixe de demonstrar as consequências práticas das vulnerabilidades abordadas.

6. **"Defeating SQL Injection":** Nesta pesquisa, os autores apresentam uma análise sobre as técnicas de combate a SQL *Injection*, destacando a importância de uma abordagem combinada que envolve práticas de codificação segura, detecção de vulnerabilidades e prevenção de ataques em tempo de execução. O artigo destaca a importância de uma abordagem abrangente que combine práticas de codificação segura com ferramentas de detecção e prevenção de ataques.

   O artigo apresenta uma tabela de comparação que classifica diferentes métodos de defesa contra SQL *Injection* em três categorias:

   - **Codificação Defensiva:** Aborda práticas manuais e automatizadas, como a utilização de consultas parametrizadas, escape de caracteres, validação

de tipo de dados, *white list filtering*, e o uso de *frameworks* como SQL DOM.

- **Detecção de Vulnerabilidades:** Apresenta técnicas de análise de código estático, análise dinâmica, e *taint-based* e *data-mining* para a identificação de vulnerabilidades. Cita ferramentas como SQLUnitGen, MUSIC, SUSHI, Ardilla, PhpMinerl e AMNESIA.
- **Prevenção de Ataques em Tempo de Execução:** Discute métodos como randomização de palavras-chave, monitoramento de tempo de execução, e a especificação de consultas legítimas, alguns das ferramentas citadas são SQLrand, WASP, SQLProb, e CANDID.

O artigo destaca as vantagens e desvantagens de cada abordagem e de cada ferramenta. Ele também menciona algumas ferramentas populares disponíveis para detecção e prevenção de SQL *Injection*, como HP WebInspect, IBM Rational AppScan, Acunetix Web Vulnerability Scanner, SecuBat, Nikto2, sqlmap, e SQLIMW, TransSQL, CANDID e WAVES.

O artigo se destaca por apresentar uma comparação estruturada dos métodos de defesa contra SQL *Injection*, facilitando a análise das abordagens existentes. No entanto, a pesquisa não aborda o impacto dessas técnicas no desempenho dos sistemas, o que poderia fornecer uma visão mais prática sobre a viabilidade de cada solução em ambientes reais.

7. **"A Countermeasure to SQL Injection Attack for Cloud Environment":** Este artigo aborda a problemática dos ataques de injeção de SQL (SQLIA) no contexto da computação em nuvem e propõe um mecanismo de detecção chamado CCSD (Cloud Computing SQLIA Detection). Ao contrário de outras abordagens tradicionais que exigem acesso ao código fonte da aplicação, o CCSD é projetado para ser implementado diretamente em ambientes de nuvem, sem a necessidade de modificações no código existente.

   A proposta utiliza a análise estrutural de árvores de *parsing (parse trees)* para comparar a estrutura das consultas SQL com um repositório de consultas seguras pré-definidas. Esta metodologia permite a detecção de SQLIA com baixos índices de falsos positivos e negativos. Os principais componentes e etapas da metodologia CCSD incluem:

   - **Análise Estática e Dinâmica:** Combinando fases offline e online, o CCSD utiliza modelos de consulta pré-construídos para avaliar novas consultas em tempo real, validando-as contra a estrutura das consultas seguras armazenadas.

- **Rastreamento de Execução:** Utiliza pilhas de execução e árvores de *parsing* para capturar a estrutura exata das consultas SQL, facilitando a comparação entre consultas legítimas e suspeitas.
- **Hashing e Repositório:** Um repositório seguro com técnicas de hashing armazena as árvores de *parsing* de consultas seguras, permitindo a verificação rápida e eficiente de novas consultas sem o código-fonte.

Em síntese, o artigo demonstra que o CCSD oferece uma alternativa robusta e escalável para a detecção de SQLIA em ambientes de nuvem, apresentando alta precisão e baixo tempo de processamento. Comparado a métodos tradicionais, o CCSD se destaca por sua adaptabilidade ao contexto de computação em nuvem, sem dependência do código fonte da aplicação.

O mecanismo CCSD, proposto no trabalho *"A Countermeasure to SQL Injection Attack for Cloud Environment"*, tem como ponto positivo a facilidade de integração em ambientes de nuvem sem a necessidade de modificar o código da aplicação. Contudo, apresenta três desvantagens: (1) a necessidade de manter atualizada a base de dados das árvores de parse, (2) a complexidade na adaptação às frequentes mudanças nas aplicações e (3) a possibilidade de comprometimento da eficácia na detecção de SQLi se as atualizações não forem realizadas regularmente.

8. **"SQL Injection Attack Detection and Prevention Techniques Using Deep Learning":** Neste artigo é feito um estudo comparativo entre o uso de uma Rede Neural Convolucional (CNN) e uma ferramenta de Perceptron Multicamada (MLP), esses modelos são alimentados com um conjunto de dados divididos entre dados de treinamento e teste, com exemplos de injeção SQL e requisições HTTP normais e requisições HTTP maliciosas.

   Em síntese o trabalho feito inicia por um processo de limpeza de dados, onde foi realizada uma limpeza do tráfego HTTP selecionado, removendo da análise codificações que não são interessantes para identificar o ataque SQLi, como urlencode, JSON e outros. Após a limpeza é realizada uma análise lexical para gerar modelos de vetor de palavras, com os mais variados tipos de combinações utilizadas para efetuar a injeção de código. O treinamento do modelo é feito então a partir da análise semântica dos vetores de palavras criados, sendo que para a criação dessas entradas foram usadas ferramentas como ReLu e Softmax para classificar essas entradas como requisições HTTP maliciosas ou não.

   Por fim, temos que ambos os modelos passaram por um processo de *Deep Learning*, onde aprendem a distinguir os tipos de dados. Para concluir o autores do artigo fizeram uma série de testes para validar o desempenho dos modelos em detectar ataques, tendo como resultado uma acurácia de 98.2% e precisão de 47% a partir de

um conjunto de treinamento de 25.487 (vinte e cinco mil, quatrocentos e oitenta e sete) amostras de injeção SQL.

Além disso, o artigo apresenta um processo completo de pré-processamento de dados, incluindo limpeza e análise lexical, garantindo a qualidade dos dados utilizados no treinamento dos modelos de detecção de SQL *Injection*. No entanto, a pesquisa se limita ao uso de redes neurais e não compara seus resultados com técnicas tradicionais, o que dificulta avaliar se a abordagem realmente supera métodos já estabelecidos.

9. **"SQL Injection Attacks Prevention System Technology: Review":** O Presente trabalho consiste em uma revisão sistemática acerca do tema SQL *Injection*, abordando como o ataque funciona e alguns principais métodos de prevenção a essa ameaça. Além da revisão sistemática, o artigo destaca os ataques SQLi baseados em PHP, e apresenta duas medidas de prevenção que também são construídas em PHP. Sobre os ataques baseados em PHP, o texto traz informações sobre injeção de código SQL por meio do uso dos métodos GET e POST diretamente na URL de uma Aplicação Web.

    - **SQLi com método GET:** O método GET envia dados ao servidor através da URL. Isso significa que os dados que um usuário fornece (como um número de identificação de funcionário) são incorporados diretamente na URL da requisição. E tendo em vista que os parâmetros da URL são de fácil visualização, é possível que estes sejam modificados, bastando a alteração da parte da URL que contém os parâmetros.
    - **Injeção com o método POST:** Esté método é chamado após o recebimento dos dados, e semelhante ao método GET a Aplicação Web não realiza uma detecção de segurança, resultando na combinação imediata dos dados com comandos SQL. Essa característica em aplicações PHP criam uma vulnerabilidade para injeções SQL que podem funcionar para ignorar a parte de autenticação da consulta SQL, permitindo o acesso não autorizado a alguma informação do banco de dados.

    No artigo são apresentadas diversas sugestões de contramedidas, como:

    - **Filtragem de palavra-chave:** Os dados do usuário são recebidos pelo sistema de Aplicação Web em PHP através das variáveis globais ($_GET, $_POST), portanto essa contramedida consiste em filtrar instruções SQL e caracteres especiais, antes de processar esses dados.
    - **PHP Data Object e Prepared Statements:** Ambas as funcionalidades são utilizadas juntas para conectar ao banco de dados e manipular dados em formulários web.

- **SQL Injection e Teste Black Box Automatizado:** É uma abordagem automatizada para avaliar a vulnerabilidade de aplicações web a ataques de injeção SQL. Utilizando um método *"black box"*, onde o teste simula tentativas de injeção SQL sem conhecimento interno do sistema, interagindo apenas através das interfaces públicas da aplicação.

O artigo apresenta uma análise detalhada sobre como os métodos GET e POST podem ser explorados em ataques SQL *Injection*, facilitando o entendimento das vulnerabilidades em aplicações PHP. No entanto, sua abordagem é limitada a essa linguagem, deixando de considerar como a ameaça se manifesta em outros ambientes, o que restringe a aplicabilidade das soluções propostas.

Por fim, conforme mencionado no início deste Capítulo, cinco dos nove artigos apresentados foram selecionados como base para esta monografia, pois forneceram respostas mais diretas e alinhadas às questões de pesquisa que orientam este estudo.

5

Material e Métodos

Este trabalho visa desenvolver uma monografia sobre a prevenção de ataques de injeção SQL em bancos de dados MySQL, com foco nos principais tipos de ataques *In-band*: *Error-based, Union-based e Boolean-based*, e dedicado a oferecer um conjunto de boas práticas no desenvolvimento do Banco de Dados de uma aplicação para diminuir as vulnerabilidades que possam resultar em um ataque de SQL *Injection*. A metodologia empregada para a construção da presente monografia buscou fundamentos na revisão sistemática da literatura realizada e sintetizada no capítulo anterior, com o objetivo de coletar e analisar informações relevantes sobre os seguintes aspectos:

- **Tipos de Ataques de Injeção SQL:** A pesquisa se concentra em compreender os mecanismos de funcionamento dos ataques de Injeção SQL *In-band*, especificamente *Error-based, Union-based e Boolean-based*, a partir de diferentes fontes teóricas.
- **Vulnerabilidades de Bancos de Dados MySQL:** A investigação se aprofunda nas vulnerabilidades do sistema de gerenciamento de banco de dados MySQL que podem ser exploradas por ataques de Injeção SQL.
- **Técnicas de Prevenção:** A pesquisa aborda diversas técnicas de prevenção de ataques de Injeção SQL, tanto a nível de aplicação quanto a nível de banco de dados, com destaque para *Prepared Statements*, *SQLrand*, filtros de *input* de usuário, tratamento de erros, funções de comparação seguras e segregação de dados.

5.1 Definição das Questões de Pesquisa

As questões desenvolvidas para orientar a seleção de artigos e monografias foram:

- **QP1:** Quais são os principais tipos de ataques de Injeção SQL *In-band* e seus mecanismos de funcionamento?
- **QP2:** Quais são as vulnerabilidades do MySQL que podem ser exploradas por ataques de Injeção SQL?

- **QP3:** Quais são as técnicas de prevenção de injeção SQL que podem ser aplicadas durante o desenvolvimento do Banco de Dados de uma Aplicação Web ou na manutenção de uma Aplicação Web já existente?

Abaixo se encontram os trabalhos acadêmicos e artigos selecionados da Revisão da Literatura feita no Capítulo 4, que contribuíram para o desenvolvimento da presente monografia.

Tabela 5.1: Síntese dos Trabalhos Selecionados que contribuíram com o desenvolvimento da presente monografia

Trabalho	Contribuição
Identification and Mitigation Tool for SQL Injection Attacks	Orientações práticas sobre o uso da ferramenta WebVIM e mitigação de ataques SQLi.
A Design Review: Concepts for Mitigating SQL Injection Attacks	Revisão de métodos de mitigação e responsabilidade dos desenvolvedores.
SQL Injection Attacks Countermeasures Assessments	Identificação dos tipos de ataques SQLi e principais contramedidas.
Detection and Prevention of SQL Injection Attack: A Survey	Comparação de metodologias conhecidas para detecção e prevenção de SQLi.
Defeating SQL Injection	Análise sobre diferentes práticas de codificação, técnicas de detecção e prevenção de SQL Injection, ferramentas e abordagens para mitigação.

5.2 Contribuição do Trabalho

A presente monografia visa contribuir para a literatura existente ao apresentar uma abordagem que reúne e consolida boas práticas e técnicas para a prevenção de SQL *Injection* em Aplicações Web. Diferentemente de outros trabalhos que se concentram em aspectos teóricos ou em técnicas inovadoras específicas, esta monografia adota uma perspectiva prática e aplicada, construída por meio de figuras ilustrativas e de um aplicativo web desenvolvido especificamente para este estudo, onde são realizados testes práticos de algumas das soluções mais eficazes identificadas na literatura. Essa abordagem prática facilita a compreensão e a aplicação das técnicas discutidas, oferecendo um recurso valioso tanto para profissionais quanto para acadêmicos.

Além disso, esta monografia se propõe a ser uma fonte de boas práticas e técnicas de mitigação de SQL *Injection*, posicionando-se como um referencial prático e acessível para desenvolvedores, engenheiros de segurança, estudantes e outros profissionais da área de TI. Ao

compilar e ilustrar as práticas recomendadas, este trabalho visa não apenas educar, mas também capacitar os leitores a implementar soluções a nível de programação.

Em resumo, este trabalho se diferencia por:

- Focar na consolidação e aplicação das boas práticas e técnicas existentes, ao invés de desenvolver novas metodologias.
- Utilizar figuras ilustrativas e um Aplicativo Web para realizar testes práticos das soluções discutidas.
- Fornecer uma base prática e acessível para profissionais da área de TI e estudantes implementarem técnicas eficazes de mitigação de SQL *Injection* em suas aplicações.

5.3 Desenvolvimento da Aplicação

Como mencionado no Capítulo 1 de Introdução esta monografia incluiu o desenvolvimento de uma aplicação propositalmente vulnerável (Aplicação do tipo Damn Vulnerable Web Application (DVWA)), projetada para enriquecer as explicações sobre ataques SQL *Injection* e suas respectivas contramedidas. Imagens dos ataques e da maneira com a qual as contramedidas foram implementadas serão copiadas no trabalho, bem como os leitores podem acessar a aplicação para por em prática os ataques e as contramedidas.

A aplicação, apelidada de Bear Bank, foi criada de tal forma que simula uma aplicação básica de banco, onde é possível realizar um *login* com as credenciais informadas no Subtópico 5.3.1. Após o *login* há uma página onde há informações do nome do usuário, saldo e transações fictícias associadas a esta conta. Por fim, a aplicação também conta com uma tela de investimento fictício, onde o usuário pode aplicar uma quantia para aumentar o valor do seu investimento.

Essa aplicação está parametrizada para que o usuário possa utilizar separadamente as contramedidas implementadas durante o desenvolvimento do programa, bem como o usuário poderá utilizar diversos campos de entrada, construídos de diferentes formas, para verificar como alguns ataques conhecidos de SQL *Injection* funcionam.

Também encorajamos o usuário a verificar os *prints* de *debug* emitidos no terminal de execução do programa, para acompanhar como as requisições ao banco de dados se comportam em face a Injeções SQL.

5.3.1 Execução e Uso da Aplicação

A Aplicação Beark Bank (EMANUELLY; LUCAS, 2025) desenvolvida está disponível no seguinte repositório do Github `https://github.com/ylleuname/Bear-Bank-DVWA`, onde se encontram também as instruções iniciais para instalação, execução e uso.

Na aplicação temos a princípio dois usuários fictícios criados. Neste contexto didático, esses dois usuários serão alvos dos ataques. E tendo em vista que alguns ataques são feitos

em campos de entrada disponíveis apenas após a autenticação do usuário, iremos informar as credenciais de acesso de cada um desses.

Usuário	Agência	Conta	Senha
Usuário 1	1234	56789	senha123
Usuário 2	1234	54123	senha456

Também é importante notar que a aplicação tem três opções iniciais de contramedidas já implementadas, caso nenhuma dessas opções seja marcada então a aplicação está no modo vulnerável, do contrário a contramedida correspondente será acionada para mitigar certos tipos de ataque. Em resumo, as contramedidas *Prepared Statement* e Filtragem de dados foram implementadas para mitigar ataques SQLi realizados na interface de *login*, enquanto a *Stored Procedure* foi criada para mitigar os ataques realizados na interface de transação e investimento.

5.3.2 Ferramentas Utilizadas

As ferramentas utilizadas para desenvolvimento e compartilhamento da Aplicação foram as seguintes:

- **Python:** É uma linguagem de programação de alto nível, interpretada e de propósito geral. Foi escolhida para este projeto devido à sua simplicidade e vasta gama de bibliotecas que facilitam o desenvolvimento rápido e eficiente de Aplicações Web (PYTHON, 2025).
- **Flask:** É um micro *framework* para desenvolvimento web em Python, conhecido por sua simplicidade e flexibilidade. Ele foi utilizado para construir o lado servidor da Aplicação Web, simplificando a criação de rotas, processamento de requisições e renderização de *templates* HTML (FLASK, 2025).
- **MySQL:** É um SGBD Relacional e de Código Aberto (Open Source). Seu lançamento ocorreu no dia 23 de maio de 1998, porém até os dias atuais ele permanece muito relevante, sendo um dos dos SGBDs mais famosos (MYSQL, 2025). A escolha do MySQL para este projeto se dá por dois principais fatores, estes são: a facilidade do seu uso e sua relevância no mercado de Desenvolvimento Web:
    - SGBD de Código Aberto, sob licença GPL, com uma versão comercial gerida pela Oracle: Conforme dito o MySQL é um software de código livre, o que implica uma vasta documentação pela internet, e também implica no SGBD possuir maior suporte a *plugins* e customizações.
    - SGBD bastante popular em projetos pequenos e também em ambientes reais de produção: Atualmente o MySQL é o 2° SGBD mais relevante no mundo, conforme o relatório DB-Engines (DB-ENGINES, 2024), bem

como ele é utilizado em empresas de grande porte como HP, Sony e Bradesco.

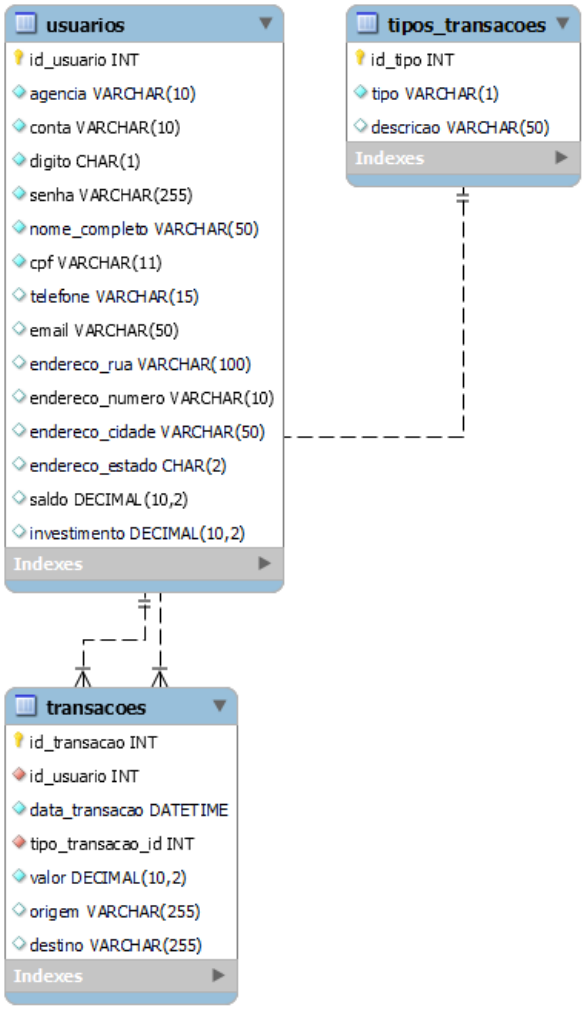
- Suporte a sistemas cliente/servidor: É um SGBD que utiliza um servidor multi-thread, permitindo múltiplas conexões simultâneas de diferentes clientes. Além disso, oferece compatibilidade com diversas linguagens de programação, facilitando sua integração em diferentes tipos de aplicações (ORACLE, 2024d).

- **Docker:** É uma ferramenta que possibilita a criação de *containers*. Os *containers* são unidades de virtualização a nível de sistema operacional, então em um único *containers* pode haver uma aplicação completa em execução. Neste projeto o Docker foi usado para fornecer o ambiente já configurado da aplicação para estudo de SQL *Injection* (DOCKER, 2025).

5.3.3 Modelo Entidade-Relacionamento

O Modelo Entidade-Relacionamento (MER) utilizado neste projeto visa ilustrar a estrutura do Banco de Dados construído para a Aplicação Bear Bank, mostrando como as entidades estão relacionadas entre si, conforme Imagem 5.1. Onde cada entidade representa uma tabela do Banco de Dados, e os relacionamentos definem a forma como essas tabelas se conectam para armazenar informações pertinentes à aplicação, como dados de usuários e transações financeiras.

Figura 5.1: Modelo Entidade Relacionamento do Banco de Dados da Aplicação DVWA Bear Bank



Fonte: Autoria própria.

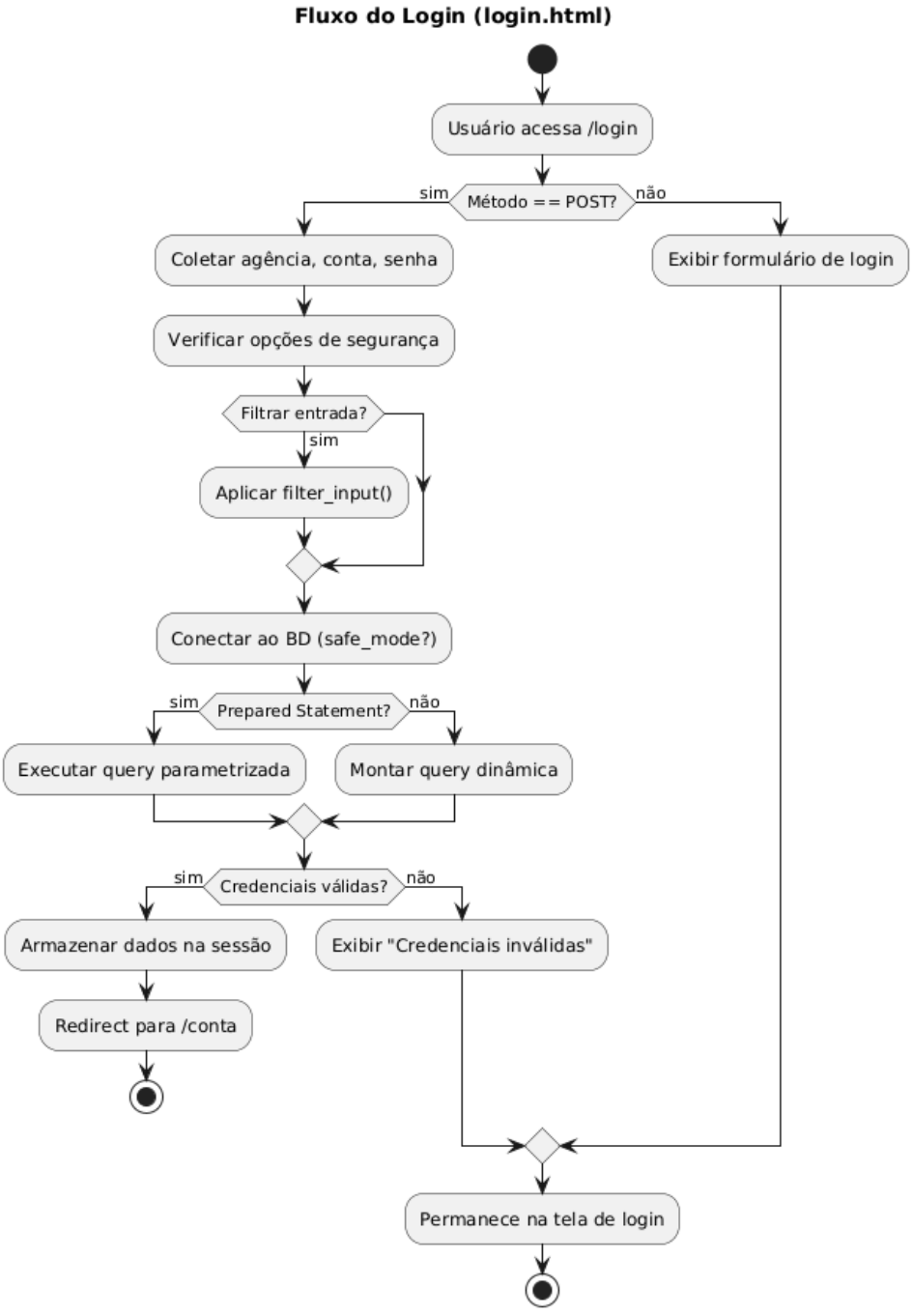
5.3.4 Fluxos da Aplicação

Os fluxos da aplicação foram modelados para representar as interações do usuário e as decisões lógicas em cada tela do sistema. Cada diagrama reflete funcionalidades específicas, mecanismos de segurança e comportamentos do sistema.

5.3.4.1 Fluxo de *Login*

Responsável pela autenticação do usuário, incorpora verificações de segurança como *Prepared Statements*, filtragem de entrada e seleção de modo de conexão ao banco de dados (seguro ou vulnerável). Redireciona para a tela de conta em caso de sucesso ou mantém o usuário na tela de *login* em caso de falha, conforme diagrama de fluxo da Figura 5.2.

Figura 5.2: Diagrama do Fluxo da tela de *Login* da Aplicação Bear Bank

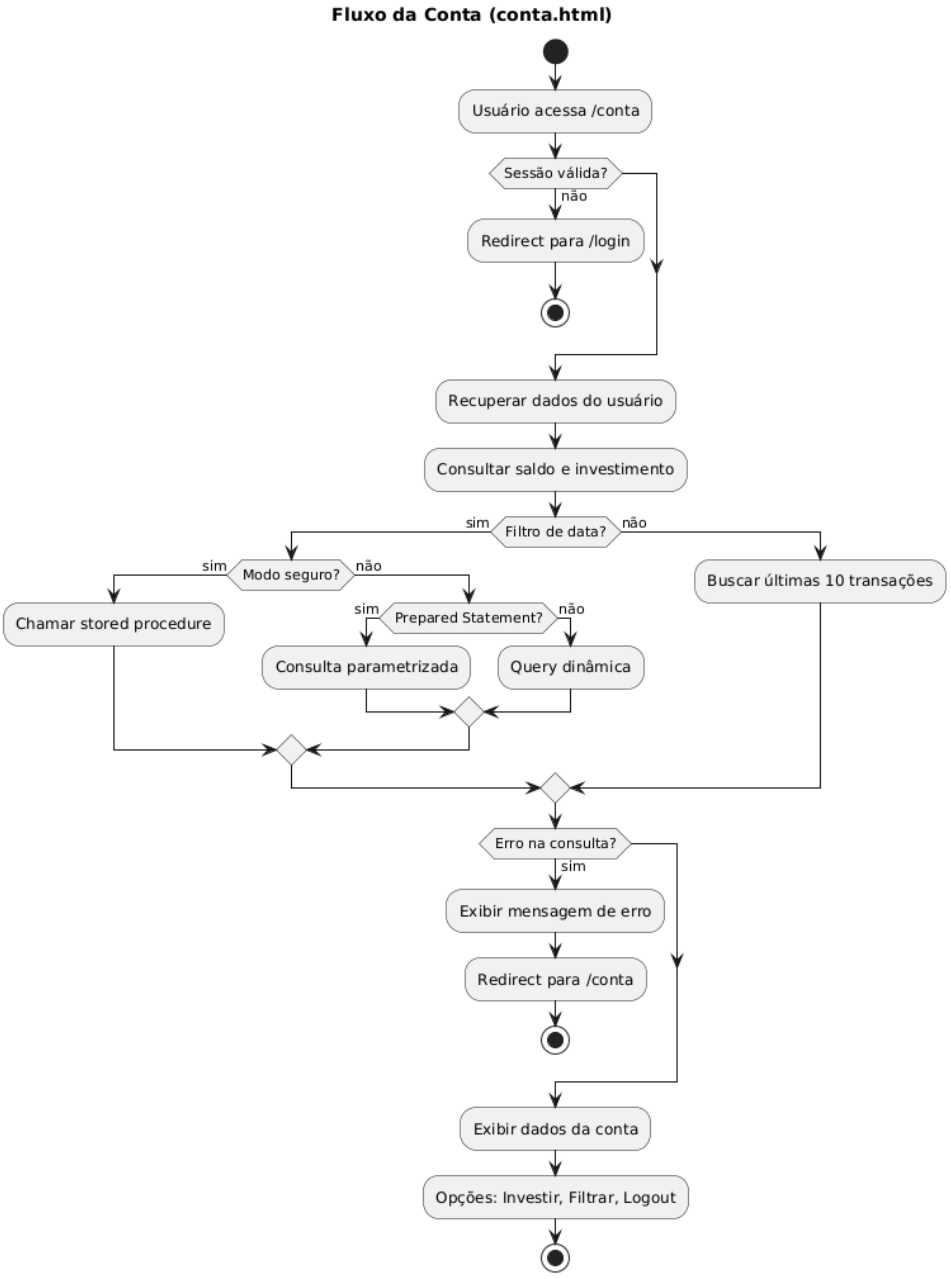


Fonte: Autoria própria.

5.3.4.2 Fluxo da Conta

Exibe informações do usuário (saldo, investimentos) e transações recentes. Inclui filtragem por data, utilizando *Stored Procedures* (modo seguro) e *Prepared Statements* ou consultas dinâmicas (modo vulnerável), conforme sintetizado no Diagrama de Fluxo da Figura 5.3.

Figura 5.3: Diagrama do Fluxo da tela de Conta da Aplicação Bear Bank.

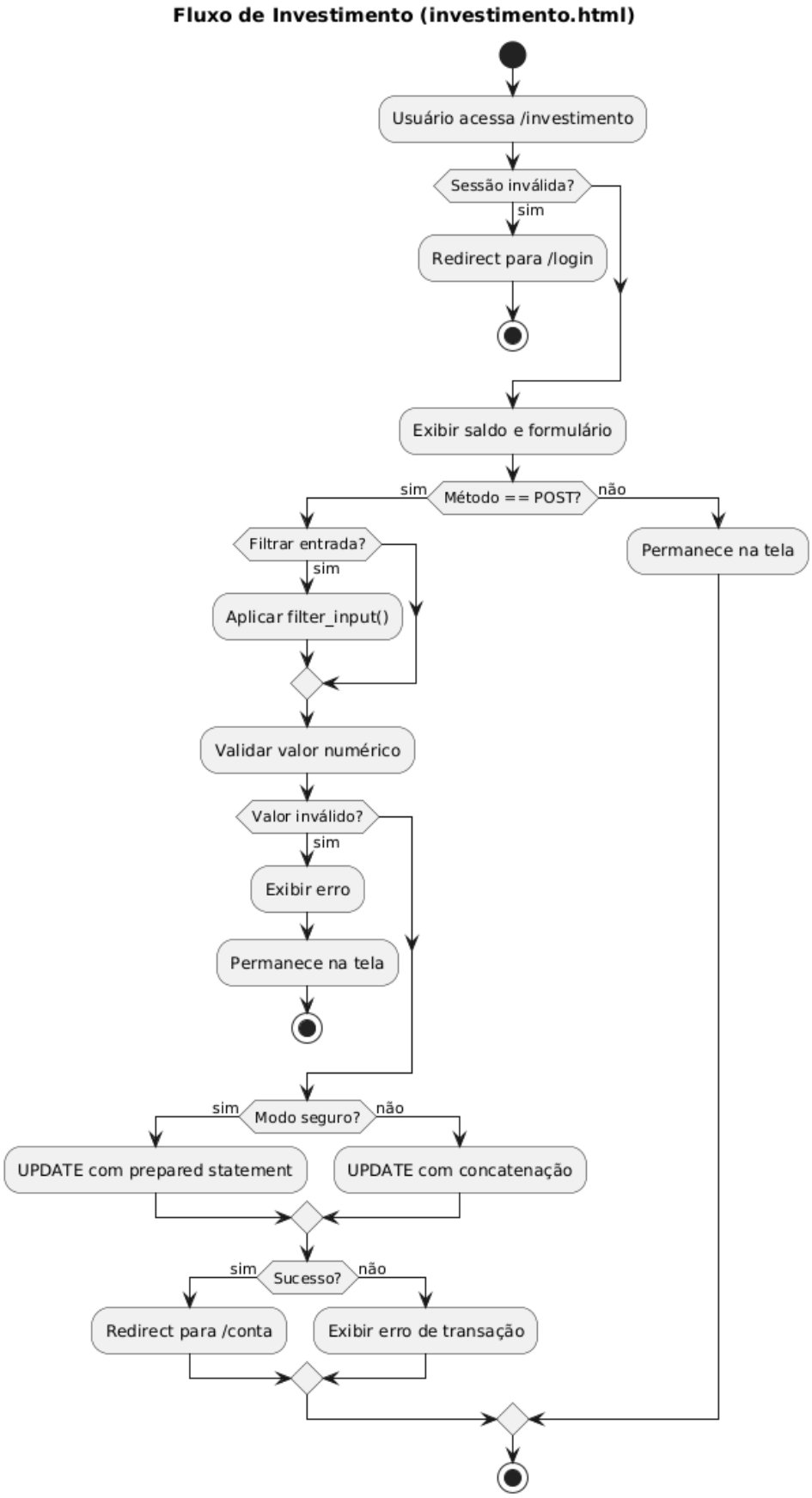


Fonte: Autoria própria.

5.3.4.3 Fluxo de Investimento

Gerencia a alocação de recursos financeiros, com validações de saldo e formatação de entrada. Oferece dois modos de operação: atualização segura via *Prepared Statements* e filtragem (Definidos na tela de *login*) ou atualização vulnerável por concatenação direta de valores na *query* SQL, conforme sintetizado na Figura 5.4.

Figura 5.4: Diagrama do Fluxo da tela de Investimento da Aplicação Bear Bank



Fonte: Autoria própria.

5.3.4.4 Fluxo de Logout

Encerra a sessão do usuário de forma segura, limpando todos os dados armazenados e redirecionando para a tela de *login*.

6

Boas Práticas de Programação e Desenvolvimento de Banco de Dados

As boas práticas de programação aplicadas à interação entre uma aplicação e seu Banco de Dados podem ser entendidas como contramedidas contra ataques cibernéticos. Contramedidas são técnicas e métodos empregados em sistemas computacionais para aumentar a segurança e reduzir vulnerabilidades SCHELDT (2023). Normalmente, um conjunto de contramedidas é adotado para prevenir ou mitigar ataques, sendo que a prevenção busca eliminar o risco, enquanto a mitigação reduz a chance de exploração ou minimiza o impacto do ataque.

Dessa forma, este capítulo apresenta um conjunto de contramedidas que consistem em boas práticas na programação de aplicações e na construção e administração de um Banco de Dados. Além disso, será utilizada a aplicação Bear Bank propositalmente vulnerável, para demonstrar a execução de ataques SQL *Injection* abordados neste trabalho, bem como a eficácia das contramedidas propostas. Usuários interessados em acessar a aplicação e realizar seus próprios testes podem seguir as instruções descritas no Capítulo 5 de Materiais e Métodos.

6.1 *Prepared Statements*

Prepared Statement é uma técnica capaz de atuar na proteção de SQL *Injection*, permitindo que uma consulta SQL seja compilada e armazenada no Banco de Dados antes da inserção dos valores dinâmicos, seu processo ocorre em duas etapas, a de Preparação e Execução, na fase de Preparação a consulta é SQL é enviada ao Banco de Dados com o uso de espaços reservados chamados de *Placeholders*, o Banco irá analisar e validar essa consulta. Já na fase de execução os valores são enviados separadamente e preenchidos nos *Placeholders*, garantindo que sejam tratados apenas como dados e não como parte da consulta SQL (KUMAR; RAJ; SRIVASTAVA, 2020).

Para exemplificar seu funcionamento, imagine que você tem um formulário de login. O usuário digita o nome de usuário e a senha, e a aplicação precisa verificar se essa combinação existe no banco de dados.

Com uma *Prepared Statement*, a aplicação envia primeiro a estrutura da consulta SQL para o banco de dados. Essa estrutura é algo como: ‘’Selecione todas as linhas da tabela de nome users onde o nome de usuário é igual a [*placeholder*] e a senha é igual a [*placeholder*]‘’.

Em seguida, os dados do usuário, ou seja, o nome de usuário e a senha, são enviados como parâmetros separados. O banco de dados processa a consulta utilizando os parâmetros fornecidos, garantindo que os dados não sejam interpretados como código SQL, conforme definição.

6.1.1 *Prepared Statements* na aplicação

Na aplicação o *Prepared Statement* foi construída para limitar os parâmetros inseridos pelo usuário no campo de *login* e no campo de busca de transação por data. Para isso definimos um *placeholder* que espera receber uma *string* em cada campo de entrada, no caso da área de login os valores inseridos no campos ‘’Agência‘’, ‘’Conta‘’ e ‘’Senha‘’ serão valores a serem comparados com valores armazenados no banco de dados. O *Prepared Statement* foi construída da seguinte forma:

```
    query = """
            SELECT * FROM usuarios
            WHERE agencia = %s AND conta = %s AND senha = %s
            """
            print(f"Consulta (Modo Seguro): {query}")
            cursor.execute(query, (agencia, conta, senha))
```

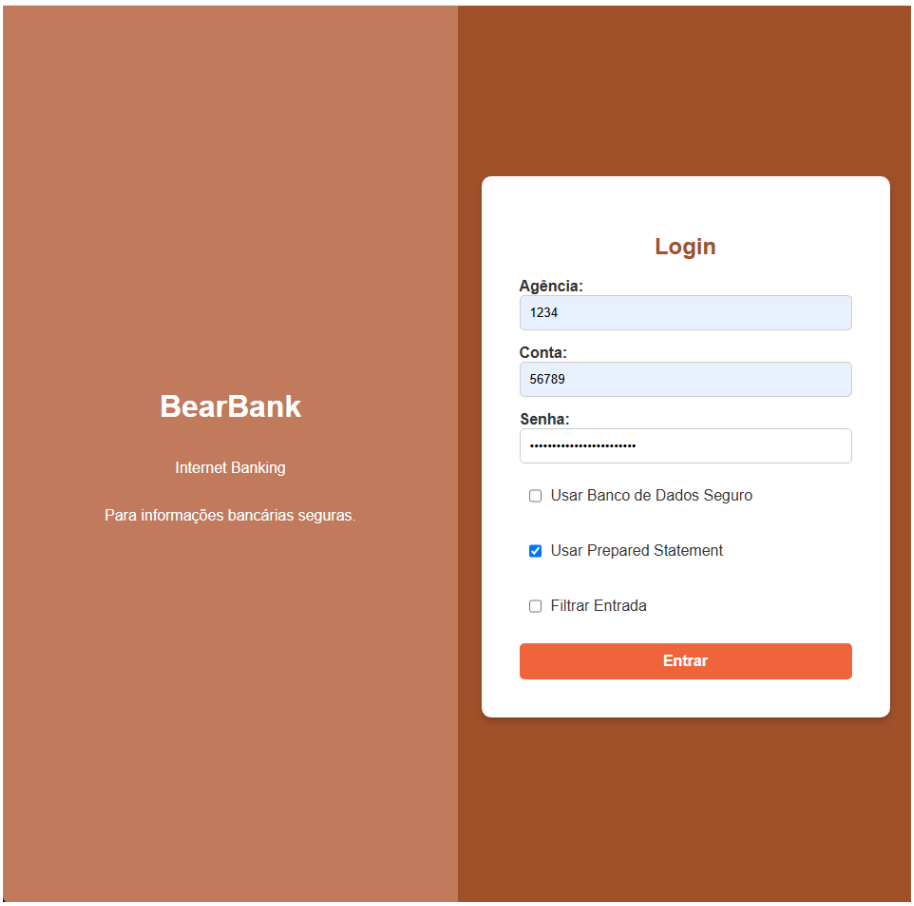
Bloco de Código 6.1: Prepared Statement implementado na Aplicação Bear Bank

Para testar essa contramedida, construímos um ataque de SQL *Injection* do tipo Boolean-based. Nesse ataque, o invasor precisa conhecer a Agência e a Conta de um usuário do banco fictício. Essas informações permitem validar corretamente o campo Agência e construir uma condição logicamente verdadeira no campo Senha. Como mostrado abaixo, a *query* resultante sempre retorna um usuário válido. Isso ocorre porque o primeiro termo (senha = ”) verifica uma senha vazia, enquanto o segundo (conta = ’54123’) garante que a condição seja verdadeira, independentemente da senha real. Além disso, o uso de comentários SQL (–) impede a execução do restante da consulta, permitindo o login indevido. Para reproduzir o ataque na aplicação, siga os comandos abaixo na tela de Login:

Agência: 1234
Conta: 54123
Senha: ’ OR conta = ’54123’ –
(adicione um Espaço ao fim deste comando)

Por outro lado, com o uso do *Prepared Statement*, a *query* não irá interpretar os valores inseridos no campo de Senha como comandos, e sim como um valor sem correspondência no banco de dados. O que impede que as verificações que tornam a *query* logicamente verdadeira não ocorram. Acompanhe as imagens da aplicação para verificar o ataque sendo realizado e as queries geradas, na Figura 6.1 e nas Listas 6.2 e 6.3.

Figura 6.1: Tela de login da aplicação com ataque Union Based no campo de senha.



Fonte: Autoria própria

```
Query Prepared Statement - login
SELECT * FROM usuarios
WHERE agencia = %s AND conta = %s AND senha = %s

127.0.0.1 - - [30/Jan/2025 12:13:18] "POST /login HTTP/1.1" 200 -
127.0.0.1 - - [30/Jan/2025 12:13:18] "GET /static/css/login.css HTTP
   /1.1" 304 -
```

Bloco de Código 6.2: Query segura gerada na Aplicação após requisição de login, com Prepared Statement

A termo de comparação, verifique a Lista 6.2, que indica os *placeholder*s sendo utilizados para receber os valores inseridos pelo usuário. Enquanto a Lista 6.3 demonstra os comandos SQL sendo recebidos pelos parâmetros do Banco de Dados, sem qualquer sanitização.

```
Query vulneravel - login
SELECT * FROM usuarios
WHERE agencia = '1234' AND conta = '56789' AND senha = '' OR conta =
   '54123' -- '
```

```
4
5 127.0.0.1 - - [30/Jan/2025 12:14:30] "POST /login HTTP/1.1" 302 -
6 127.0.0.1 - - [30/Jan/2025 12:14:30] "GET /conta HTTP/1.1" 200 -
7 127.0.0.1 - - [30/Jan/2025 12:14:30] "GET /static/css/conta.css HTTP
    /1.1" 304 -
```

Bloco de Código 6.3: Query vulnerável gerada na Aplicação após requisição de login, sem Prepared Statement

6.1.2 *Prepared Statements* e ataque SQL *Injection* de Manipulação de Dados

Até o presente momento, o nosso trabalho não dissertou diretamente os ataques SQL *Injection* que se propõem a alterar o estado do Banco de Dados da Aplicação. Porém ataques como estes podem ser construídos com os fundamentos dissertados aqui neste trabalho, e conhecimento básico de como *queries* são realizadas ao Banco de Dados MySQL. Na nossa aplicação demonstraremos como uma ataque desse tipo pode ser construído a partir da exploração dos comandos UPDATE e SET já existente. E em contrapartida demonstrar que a *Prepared Statement* é uma contramedida que mitiga algumas versões deste ataque.

Considere a *Prepared Statement* construída para proteger o campo de inserção de valores da tela de Investimentos:

```
1     query = """
2             SELECT * FROM usuarios
3             WHERE agencia = %s AND conta = %s AND senha = %s
4             """
5             print(f"Consulta (Modo Seguro): {query}")
6             cursor.execute(query, (agencia, conta, senha))
```

Bloco de Código 6.4: Prepared Statement implementada no Input de Usuário da Aplicação na tela de Investimento

Considere também o ataque selecionado para explorar o fato de que ao realizar um investimento o parâmetro de ‘’saldo‘’ e de ‘’valor‘’ do investimento irão mudar, ou seja, será necessário realizar um UPDATE no Banco de Dados. Sabe-se que o valor do parâmetro Saldo irá diminuir e o valor do campo de Investimento irá aumentar, portanto o ataque foi construído pensando em realizar uma manipulação aritmética que ao invés de subtrair o valor de saldo no momento do investimento, ocorra que o invasor insira o valor que ele quer atualizar o parâmetro Saldo. O ataque ficou da seguinte forma:

```
Valor para investir:  (SELECT saldo -
4521.40)
```

Ainda, acompanhe a Figura 6.2 e a Lista 6.5 abaixo para verificar como a *query* se comporta sem a *Prepared Statement*.

Figura 6.2: Ataque de SQL *Injection* de Manipulação de dados - Tela de Investimento

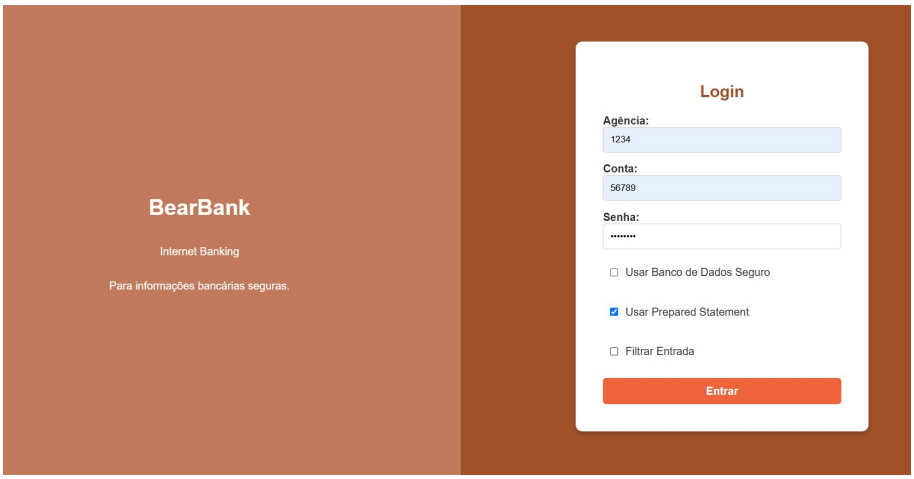
Fonte: Autoria própria

```
Modo vulneravel (concatenacao direta) ativado.
Query gerada:
UPDATE usuarios
SET saldo = saldo - (SELECT saldo - 4521.40),
investimento = investimento + (SELECT saldo - 4521.40)
WHERE id_usuario = 1
```

Bloco de Código 6.5: Query gerada na Aplicação devido o ataque de SQLi que altera o estado do Banco de Dados

A termo de comparação, verifique também que com o uso da *Prepared Statement*, a *query* sequer é gerada, devido à sanitização de entrada realizada pela contramedida, conforme a Lista 6.6.

Figura 6.3: Login seguro com uso de Prepared Statement



Fonte: Autoria própria

```
Modo seguro (prepared statements) ativado.
Query preparada: Erro durante a execucao: could not convert string to
    float: '(SELECT saldo - 4521.40)'
```

Bloco de Código 6.6: Resultado seguro após realização do ataque de SQLi que altera o estado do Banco de Dados, com Prepared Statement

6.2 Filtragem de Entrada de Dados

A filtragem de entrada de dados é uma técnica que que lidam com entrada do usuário. Ela consiste na validação e sanitização dos dados fornecidos antes que sejam processados pelo sistema, garantindo que atendam aos critérios esperados e evitando que insiram comandos inesperados ou maliciosos. Esse procedimento é essencial para prevenir erros, garantir a integridade dos dados e mitigar vulnerabilidades de segurança em Aplicações Web.

No contexto da proteção contra SQL *Injection*, a filtragem de entrada visa remover caracteres e padrões potencialmente perigosos antes que os dados sejam utilizados em consultas SQL. Essa técnica impede que comandos SQL maliciosos sejam injetados e executados indevidamente no banco de dados (ABDULQADER et al., 2017).

Imagine que você tem um campo de pesquisa em um site. O usuário pode digitar qualquer coisa nesse campo, incluindo caracteres especiais que podem ser interpretados como comandos SQL. Sem filtragem, um atacante poderia inserir algo como ’; DROP TABLE users; –, causando a exclusão da tabela de usuários.

Ao aplicar a filtragem corretamente, a aplicação garante que apenas dados válidos sejam inseridos na consulta, reduzindo significativamente os riscos de ataques. Por exemplo, se um campo de pesquisa de uma Aplicação Web for explorado para receber SQLi a filtragem de

entrada de dados pode remover não apenas caracteres especiais como aspas simples (’), aspas duplas (") e ponto e vírgula (;), mas também palavras-chave do SQL, como UNION, SELECT, DROP e INSERT. Dessa forma, evita-se que um invasor consiga modificar a estrutura da consulta e obter acesso indevido a informações do banco de dados, conforme definição.

6.2.1 Filtragem de Entrada de Dados na aplicação

Como dito, a técnica de Sanitização de Entrada exige do programador um bom conhecimento acerca dos ataques que ele pretende mitigar, pois é necessário informar exatamente quais caracteres ou palavras chaves o desenvolvedor ou administrador do sistema pretende banir dos campos de entrada da aplicação.

Na aplicação que desenvolvemos a Filtragem de Entrada consiste no escape de símbolos especiais conhecidos da linguagem SQL. como aspas simples, os símbolos de comentário do SQL, comandos que foram explorados aqui neste trabalho como o conectivo OR , AND, SELECT, UNION, e também comandos que podem estar presentes em ataques mais complexos, como UPDATE, SET, EXECUTE, e outros. Desse modo a função de sanitização foi construída da seguinte maneira:

```
1   def filter_input(user_input: str) -> str:
2   user_input = user_input.strip()
3
4   forbidden_patterns = [
5       r'(--|#)',
6       r'(/\*.*?\*/)',
7       r'(\bOR\b|\bAND\b)',
8       r'(\bUNION\b.*?\bSELECT\b)',
9       r'(\bINSERT\b|\bUPDATE\b|\bDELETE\b|\bDROP\b|\bALTER\b)',
10      r'(\bEXEC\b|\bEXECUTE\b)',
11      r'(\bSLEEP\b|\bBENCHMARK\b)',
12      r'([\'\";])',
13  ]
14
15  for pattern in forbidden_patterns:
16      user_input = re.sub(pattern, '', user_input, flags=re.
            IGNORECASE)
17
18  return user_input
```

Bloco de Código 6.7: Filtragem de Entrada implementada na Aplicação Bear Bank

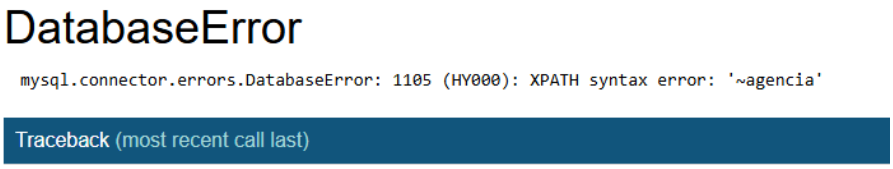
Para demonstrar o funcionamento dessa contramedida, considere o ataque *Error-based* injetado ainda no campo de login:

Agência: 1234
Conta: 54123
Senha: ' AND EXTRACTVALUE(1, CONCAT(0x7e,
(SELECT column_name FROM information_schema.columns
WHERE table_name='usuarios'
LIMIT 1 OFFSET 1))) -
(adicione um Espaço ao fim deste comando)

O ataque de SQL *Injection Error-based* foi construído usando um conjunto de comandos SQL para gerar um erro que mostrasse ao invasor informações relevantes sobre as tabelas criadas no Banco de dados. Em síntese, o ataque usa o comando EXTRACTVALUE que normalmente é usado para processar XML no MySQL, inserindo um valor inválido a esse comando, esse valor inválido será o responsável direto por expor os dados do SQL, veja no código abaixo que passamos uma *subquery* ao comando, onde tentamos acessar o *''information schema''*, que é um conjunto de visões de sistema que armazena metadados do banco de dados, permitindo que usuários consultem informações sobre tabelas, colunas, índices, permissões, restrições e outros objetos sem acessar diretamente as tabelas internas do SGBD. Ainda, são exploradas as instruções LIMIT e OFFSET, para navegar entre as tabelas do Esquema do Banco de Dados.

Realizando a injeção acima, sem marcar nenhuma *checkbox*, teremos a geração de um erro no MySQL, que nos redireciona a seguinte tela, com a informação da existência de um atributo chamado ''agencia'', na tabela ''usuarios'', conforme a Figura 6.4 abaixo:

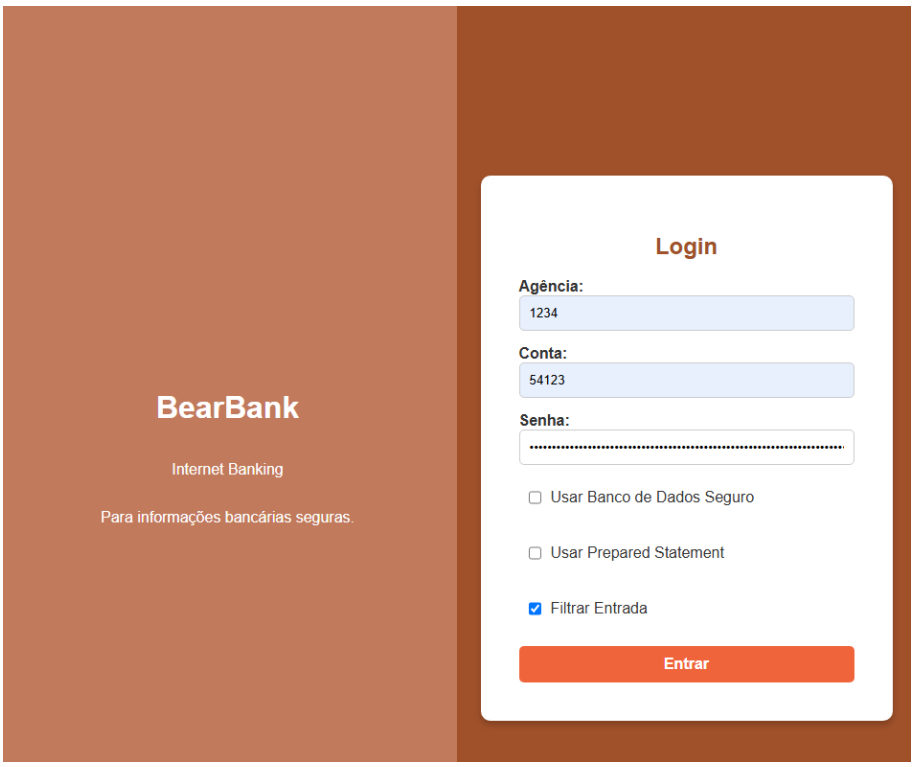
Figura 6.4: Erro do MySQL gerado pelo ataque *Error-based* aplicado no campo de login



Fonte: Autoria própria

Ainda, no *debug* da aplicação é possível vermos como cada campo de entrada do login é submetido à função de sanitização mencionada acima. Veja como existem algumas palavras-chave utilizadas no ataque que não foram banidas na filtragem; porém, ainda assim, a presença dos demais caracteres foi o suficiente para que o ataque *Error-based* não fosse bem-sucedido quando a *checkbox* de Filtragem de Entrada está ativada, conforme a Figura 6.5 e a Lista 6.8 abaixo:

Figura 6.5: Tela de Login da aplicação com a *checkbox* de Filtragem de Entrada



Fonte: Autoria própria

```
1 Input filtrado: 1234
2 Input filtrado: 56789
3 Input filtrado:  EXTRACTVALUE(1, CONCAT(0x7e, (SELECT column_name
    FROM information_schema.columns WHERE table_name=usuarios LIMIT 1
    OFFSET 1)))
```

Bloco de Código 6.8: Resultado de uma requisição indevida na tela de Login, com uso de Filtragem de Entrada

6.3 *Stored Procedures*

As *Stored Procedures* são blocos de código SQL armazenados e gerenciados diretamente no Banco de Dados. Elas permitem a execução de comandos SQL de forma reutilizável, reduzindo a redundância de código e otimizando o desempenho da aplicação. Além disso, seu uso facilita a manutenção do Banco de Dados, uma vez que as regras de negócio podem ser implementadas diretamente nas *Procedures*, evitando que lógicas complexas fiquem dispersas no código da aplicação.

No contexto de Segurança da Informação, as *Stored Procedures* podem atuar como uma barreira contra ataques cibernéticos, pois permitem que as consultas ao banco sejam pré-definidas

e controladas. Dessa forma, em vez de conceder acesso direto às tabelas, a aplicação executa **apenas** as operações permitidas dentro da *Procedure*.

Uma das principais vantagens das *Stored Procedures* contra *SQL Injection* é a possibilidade de restringir quais comandos podem ser executados no Banco de Dados. Como o usuário interage apenas com *procedures* predefinidas, e não diretamente com as tabelas, reduz-se a exposição de dados sensíveis. Além disso, é possível configurar permissões específicas, como *acesso apenas leitura*, impedindo alterações indevidas em caso de uma tentativa de ataque AHMAD; KARIM (2021).

Em *Stored Procedures*, também é possível implementar *Consultas Parametrizadas*, onde os valores passados são tratados como parâmetros e não como parte da *string* SQL. Isso impede que comandos maliciosos sejam executados, pois o Banco de Dados interpreta os dados do usuário como valores literais e não como parte da lógica da consulta OWASP (2024).

Veja no Bloco de Código abaixo, um exemplo didático de *Stored Procedure* que consulta dados sobre funcionários com base em um parâmetro específico:

```
DELIMITER //
CREATE PROCEDURE GetEmployeeDetails(IN emp_id INT)
BEGIN
    SELECT name, position, department FROM employees WHERE id =
       emp_id;
END //
DELIMITER ;
```

Bloco de Código 6.9: Exemplo de *Stored Procedure* para consulta de funcionários

Neste caso, um usuário pode fornecer um *ID* de funcionário, mas não pode modificar a estrutura da consulta SQL, reduzindo o risco de injeção de código malicioso.

6.3.1 Diferença entre *Prepared Statements* e *Stored Procedures*

Embora tanto as *Prepared Statements* quanto as *Stored Procedures* sejam técnicas interessantes para reduzir as vulnerabilidades ao SQLi, elas possuem diferenças fundamentais.

As *Prepared Statements* são consultas parametrizadas definidas no código da aplicação e enviadas ao banco de dados de maneira segura, garantindo que os valores inseridos pelo usuário sejam tratados como dados e não como comandos SQL. Esse método é mais flexível para consultas dinâmicas, pois permite a reutilização da mesma estrutura de consulta com diferentes valores de entrada.

Já as *Stored Procedures* são funções armazenadas diretamente no Banco de Dados, encapsulando consultas e operações SQL. Além de prevenir injeções SQL ao parametrizar os dados de entrada, elas permitem restringir o acesso às tabelas e definir regras específicas de execução. Seu uso é vantajoso quando se deseja centralizar a lógica de consultas dentro do banco, reduzindo a dependência da aplicação para processar dados.

Enquanto as *Prepared Statements* são mais utilizadas para consultas dinâmicas realizadas pela aplicação, as *Stored Procedures* são ideais para cenários onde se deseja definir e restringir previamente as operações no Banco de Dados, garantindo maior segurança e eficiência.

6.3.2 *Stored Procedures* na aplicação

Como dito, a *Stored Procedure* é uma técnica implementada no código de criação do Banco de dados, para isso, a nossa aplicação conta com dois estados diferentes do banco de dados, todas as informações são iguais, com exceção de que no Banco de Dados Seguro implementamos uma *Stored Procedure* para cuidar da tela de transações, protegendo e predefinindo as *queries* relativas às transações realizadas e recebidas por um usuário credenciado no sistema. Sendo assim, a *Stored Procedure* foi construída da seguinte forma no Banco de Dados de nome *'bear_bank'*:

```
1   CREATE PROCEDURE sp_get_user_and_transactions (
2       IN p_user_id INT,
3       IN p_date_filter DATE
4   )
5   BEGIN
6
7       SELECT nome_completo, agencia, conta, digito
8       FROM usuarios
9       WHERE id_usuario = p_user_id;
10
11
12      IF p_date_filter IS NOT NULL THEN
13          SELECT DISTINCT t.data_transacao AS date,
14                 tt.descricao AS transaction_type,
15                 t.valor AS amount,
16                 t.origem AS origin,
17                  t.destino AS destination
18          FROM transacoes t
19          JOIN tipos_transacoes tt ON t.tipo_transacao_id = tt.
                id_tipo
20          WHERE t.id_usuario = p_user_id AND DATE(t.data_transacao)
                = p_date_filter
21          ORDER BY t.data_transacao DESC;
22      ELSE
23          SELECT DISTINCT t.data_transacao AS date,
24                 tt.descricao AS transaction_type,
25                 t.valor AS amount,
26                 t.origem AS origin,
```

```
27                      t.destino AS destination
28              FROM transacoes t
29              JOIN tipos_transacoes tt ON t.tipo_transacao_id = tt.
                    id_tipo
30              WHERE t.id_usuario = p_user_id
31              ORDER BY t.data_transacao DESC
32              LIMIT 10;
33          END IF;
34      END
```

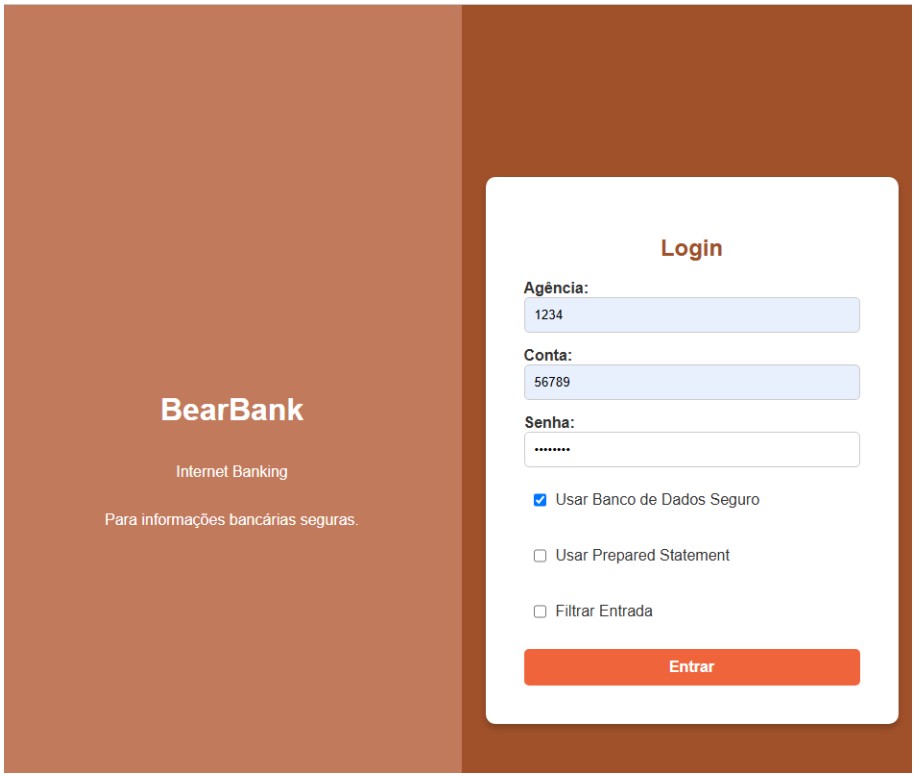
Bloco de Código 6.10: Stored Procedure implementado na Aplicação Bear Bank

Para demonstrar o funcionamento dessa contramedida, foi construído um ataque SQL *Injection Union-based*, onde o intuito é utilizar o campo de busca de transação por data para obter as credenciais de outros usuários da mesma agência, para tanto o ataque inicia informando uma data válida para fechar o campo de busca, e logo após inserindo a instrução UNION para unir os resultados da consulta original (data) com o resultado de um SELECT feito à tabela de usuários, informando os atributos com o nome completo e senha de um outro usuário do banco. O ataque ficou dessa forma:

Filtrar transação: `2024-10-15'`
`UNION SELECT NOW(), nome_completo, 0.00, 'null', senha`
`FROM usuarios -`
(adicione um Espaço ao fim deste comando)

Verifique na Lista 6.11 abaixo como o ataque gera um erro no banco de dados, pelo fato de o procedimento estar esperando um valor do tipo Data (YYYY-MM-DD).

Figura 6.6: Login com as credenciais válidas e *checkbox* de Modo Seguro marcada



Fonte: Autoria própria

```
Query Stored Procedure - Transacao [{'date': datetime.datetime(2024,
    10, 15, 9, 45), 'transaction_type': 'Recebida', 'amount': Decimal(
    '500.00'), 'origin': 'Caixa Eletronico', 'destination': 'Joao
    Silva'}]
127.0.0.1 - - [30/Jan/2025 15:48:37] "GET /conta?date=2024-10-15 HTTP
    /1.1" 200 -
127.0.0.1 - - [30/Jan/2025 15:48:37] "GET /static/css/conta.css HTTP
    /1.1" 304 -
Erro detectado: 1292 (22007): Incorrect date value: '2024-10-15'
    UNION SELECT NOW(), nome_completo, 0.00, 'Hacker', senha FROM
    usuarios -- ' for column 'p_date_filter' at row 1
```

Bloco de Código 6.11: Log de Transação e Erro SQL

A termo de comparação, veja como o ataque Union-based se comporta ao ser injetado em uma query padrão, sem medidas de segurança, conforme Figuras 6.7, 6.8 e a Lista 6.12.

Figura 6.7: Tela de transação com o ataque sendo injetado no campo de Filtro de transação

Fonte: Autoria própria

Figura 6.8: Tela de transação após o ataque *Union-based*

Fonte: Autoria própria

```
1    SELECT DISTINCT t.data_transacao AS date,
```

```
        titr.descricao AS transaction_type,
        t.valor AS amount,
        t.origem AS origin,
        t.destino AS destination
FROM transacoes t
JOIN tipos_transacoes titr ON t.tipo_transacao = titr.id_tipo
WHERE t.id_usuario = 1
  AND DATE(t.data_transacao) = '2024-10-15' UNION SELECT NOW(),
     nome_completo, 0.00, 'null', senha FROM usuarios -- '
```

Bloco de Código 6.12: Query gerada na Aplicação Bear Bank ao receber uma requisição maliciosa sem uso de Stored Procedure

Por fim, é importante salientar que diferentes técnicas podem e devem ser utilizadas juntas, para aumentar a segurança do sistema, isso porque nenhuma técnica por si só é suficiente para proteger um sistema com diferentes vulnerabilidades passíveis de serem exploradas, portanto organizamos uma Tabela 6.1 para sintetizar as contramedidas aqui apresentadas e apontar seus principais pontos positivos e negativos na proteção contra SQL *Injection*.

Tabela 6.1: Síntese das Boas Práticas de Programação e Desenvolvimento de Banco de Dados e respectivas vantagens e desvantagens.

Contramedida	**Vantagens**	**Desvantagens**
Prepared Statements	Impede a injeção de código ao separar a consulta dos dados. Melhora a segurança sem necessidade de filtragem manual. Permite reutilização eficiente de consultas.	Pode impactar o desempenho em consultas simples. Requer mudanças no código de aplicações legadas.
Stored Procedures	Restringe o acesso direto às tabelas do banco. Permite controle granular de permissões. Reduz carga de processamento na aplicação.	Pode tornar a manutenção complexa quando há muitas *procedures*. Menos flexível para consultas dinâmicas. Dependência do SGBD.
Filtragem de Input	Primeira camada de defesa contra dados maliciosos. Ajuda a prevenir outros ataques, como *XSS* e *Command Injection*.	Difícil de manter devido à evolução dos padrões de ataque. Pode causar falsos positivos e prejudicar a usabilidade.

7

Conclusão

Este trabalho teve como objetivo mapear um conjunto de boas práticas de Desenvolvimento de Banco de Dados e técnicas de programação para reduzir os riscos de ataques de SQL *Injection* do tipo *In-band*, mais especificamente os ataques do tipo *Error-based*, *Union-based* e *Boolean-based*. Para isso, exploramos os fundamentos da construção desses ataques, identificamos vulnerabilidades estruturais que os tornam possíveis e discutimos contramedidas eficazes para o fortalecimento da segurança de Aplicações Web.

Além da revisão teórica sobre os ataques e suas respectivas mitigações, um dos principais resultados desta pesquisa foi o desenvolvimento de uma aplicação propositalmente vulnerável. Esse ambiente experimental possibilitou a realização de testes práticos tanto dos ataques quanto das contramedidas propostas, permitiu validar a eficácia de técnicas como *Prepared Statements*, Filtragem de Entrada e *Stored Procedures*. Os experimentos demonstraram que a adoção dessas práticas contribui significativamente para a segurança da aplicação e que o uso combinado dessas estratégias pode oferecer uma proteção ainda mais robusta contra SQL *Injection*.

A experimentação prática foi fundamental para compreender como pequenas falhas na sanitização de entradas ou no tratamento de *queries* podem expor um sistema a ataques de SQL *Injection*, mesmo quando as vulnerabilidades não são imediatamente óbvias. Com isso, o trabalho não apenas reforça a importância das técnicas de defesa estudadas, mas também fornece um material de apoio para estudantes e profissionais que desejam aprimorar seus conhecimentos sobre Segurança de Aplicações Web, consolidando um aprendizado mais aplicado.

Por fim, reforçamos que a segurança de um sistema deve ser tratada de forma holística e multicamadas, combinando diferentes estratégias de mitigação que se complementam. A aplicação conjunta dessas medidas pode fortalecer significativamente a proteção da aplicação, tornando-a mais resiliente contra ameaças em constante evolução.

7.1 Trabalhos Futuros

Consideramos que, apesar das estratégias apresentadas demonstrarem eficácia na prevenção contra os ataques selecionados de SQL *Injection In-band*, é fundamental ressaltar que as técnicas de ataque cibernético estão em constante evolução. Isso exige que as abordagens de

contramedida também evoluam continuamente. Dessa forma, estudos futuros podem aprofundar a aplicação de medidas de segurança, explorando não apenas técnicas para a implementação segura do banco de dados e do código-fonte, mas também outras tecnologias e soluções, como detecção baseada em Inteligência Artificial, monitoramento de tráfego em tempo real e análise de logs.

Para trabalhos futuros, incentivamos uma investigação mais aprofundada sobre outros tipos de ataques de SQL *Injection*, como SQLi baseado em *cookies* de navegador, SQLi baseado em cabeçalhos HTTP ou a expansão dos ataques SQLi com o objetivo de modificar o Banco de Dados de uma aplicação, utilizando comandos como UPDATE e SET. Outra abordagem relevante seria o estudo sobre os impactos da combinação de *Stored Procedure* e *Prepared Statement* em um mesmo bloco de código, e também o estudo de mecanismos adicionais de segurança, incluindo o uso de *firewall* para Banco de Dados e sistemas de detecção de intrusão e algoritmos de *Machine Learning* aplicados à identificação de padrões de Injeção SQL.

ABDULQADER, F. et al. The impact of SQL injection attacks on the security of databases. **ResearchGate**, [S.l.], 2017.

AHMAD, K.; KARIM, M. A Method to Prevent SQL Injection Attack using an Improved Parameterized Stored Procedure. **(IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 12, No. 6, 2021**, [S.l.], 2021.

ATLASSIAN. **What is the difference between UNION and UNION ALL**. Acessado em 04 de Maio de 2024, https://www.atlassian.com/data/sql/what-is-the-difference-between-union-and-union-all.

BASTOS, A.; CAUBIT, R. **Gestão de segurança da informação**: iso 27001 e 27002: uma visão prática. Porto Alegre: Zouk, 2009.

CALDAS, A.; FREIRE, V. **Cibersegurança**: das preocupações à ação. [S.l.]: JSTOR, 2013.

CISORADVISOR. **Hackers pedem US 5 milhões de resgate à TV Record**. Acessado em 07 de Agosto de 2024, https://www.cisoadvisor.com.br/hackers-pedem-r-45-milhoes-de-resgate-a-tv-record/.

CLOUD, G. **MySQL**. Acessado em 04 de junho de 2024, https://cloud.google.com/mysql?hl=pt-br.

COELHO, F. E. S.; ARAÚJO, L. G. S. d.; BEZERRA, E. K. **Gestão da segurança da informação**: nbr 27001 e nbr 27002. Rio de Janeiro: Rede Nacional de Ensino e Pesquisa – RNP/ESR, 2014.

COSTA, S. E. d. **Conscientização em segurança da informação**. Indaial: UNIASSELVI, 2020.

DASWANI, N. A. e. a. **Big Breaches**: cybersecurity lessons for everyone. [S.l.]: Apress, 2021.

DB-ENGINES. **DB-Engines Ranking**. Acessado em 09 de outubro de 2024, https://db-engines.com/en/ranking.

DOCKER. **Docker**. Acessado em 03 de Março de 2025, https://www.docker.com/.

ELMASRI, R.; NAVATHE, S. B. **Sistemas de Banco de Dados**. [S.l.]: Pearson, 2015.

EMANUELLY, P.; LUCAS, B. **Aplicação Bear Bank**. Acessado em 25 de Fevereiro de 2025, https://github.com/ylleuname/Bear-Bank-DVWA.

FLASK. **Flask**. Acessado em 03 de Março de 2025, https://flask.palletsprojects.com/en/stable/.

HINTZBERGEN, J. et al. **Fundamentos de Segurança da Informação**: com base na iso 27001 e na iso 27002. [S.l.]: Brasport, 2018.

INVICTI. **Types of SQL Injection (SQLi)**. Acessado em 05 de Maio de 2024, https://www.acunetix.com/websitesecurity/sql-injection2/#:~:text=Boolean%2Dbased%20SQL%20Injection%20is,a%20TRUE%20or%20FALSE%20result.

JAHANSHAHI, R.; DOUPé, A.; EGELE, M. You shall not pass: mitigating sql injection attacks on legacy web applications. In: ACM ASIA CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 15. **Proceedings...** ACM, 2020. (ASIA CCS '20).

JUNIOR, M.; MOREIRA, D. **Segurança da informação**: uma abordagem sobre proteção da privacidade em internet das coisas. São Paulo, Brasil, 2020.

KASPERSKY. **O que é injeção de SQL? Definição e explicação**. Acessado em 14 de Março de 2024, https://www.kaspersky.com.br/resource-center/definitions/sql-injection.

KHAN, S. et al. A Systematic Analysis of the Capital One Data Breach: critical lessons learned. **ACM Trans. Priv. Secur.**, [S.l.], v.26, n.1, p.3:1–3:29, 2023.

KUMAR, S.; RAJ, J.; SRIVASTAVA, P. Vulnerability detection and prevention of SQL injection. In: INTERNATIONAL CONFERENCE ON EMERGING TRENDS IN INFORMATION TECHNOLOGY AND ENGINEERING (IC-ETITE), 2020. **Anais...** [S.l.: s.n.], 2020. p.1–5.

LAB, A. K. **Cyberthreat Real-Time Map**. Acessado em 14 de Março de 2024, https://cybermap.kaspersky.com.

MITNICK, K. D.; SIMON, W. L. **A arte de enganar**: controlando o fator humano na segurança da informação. [S.l.]: Pearson Universidades, 2003.

MYSQL. **MySQL**. Acessado em 03 de Março de 2025, https://www.mysql.com/.

NASEREDDIN, M. et al. A systematic review of detection and prevention techniques of SQL injection attacks. **Information Security Journal: A Global Perspective**, [S.l.], v.32, n.4, p.252–265, 2023.

ORACLE. **MySQL 8.3 Reference Manual**. Acessado em 26 de Abril de 2024, https://dev.mysql.com/doc/refman/8.3/en/string-literals.html.

ORACLE. **MySQL 8.3 Reference Manual**. Acessado em 04 de Maio de 2024, https://dev.mysql.com/doc/refman/8.4/en/set-operations.html.

ORACLE. **MySQL 8.3 Reference Manual**. Acessado em 28 de Abril de 2024, https://dev.mysql.com/doc/refman/8.3/en/query-attributes.html.

ORACLE. **O que é o MySQL?** Acessado em 09 de outubro de 2024, https://www.oracle.com/br/mysql/what-is-mysql/#:~:text=De%20acordo%20com%20o%20DB,%2C%20Shopify%20e%20Booking.com.

Oracle Corporation. **History of SQL**. Accessado em 30 de Abril de 2024, https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/History-of-SQL.html#GUID-4DD5E1B6-BEC7-4E9B-B369-1466F93ACA28.

OWASP. **OWASP Top Ten**. Acessado em 14 de Março de 2024, https://owasp.org/www-project-top-ten/.

OWASP. **Testing for SQL Injection**. Acessado em 24 de Março de 2024, https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection.

OWASP. **SQL Injection Prevention Cheat Sheet**. Acessado em 25 de Julho de 2024, https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html.

PYTHON. **Python**. Acessado em 03 de Março de 2025, https://www.python.org/.

ROCHA LYRA, M. **Governança e Segurança da Informação**. [S.l.]: Amazon, 2020. E-book.

SCHELDT, A. **What Is a Countermeasure in Computer Security?** Acessado em 24 de Julho de 2024, https://www.comptia.org/blog/what-is-a-countermeasure-in-computer-security.

SCHOENBORN, J. M.; ALTHOFF, K.-D. Detecting SQL-Injection and Cross-Site Scripting Attacks Using Case-Based Reasoning and SEASALT. In: **Anais...** [S.l.: s.n.], 2021. p.66–77.

TANIMURA, C. **SQL para Análise de Dados**: técnicas avançadas para transformar dados em insights. [S.l.]: Novatec Editora, 2022.