A IMPORTÂNCIA DO TESTE DE SOFTWARE E SUAS APLICABILIDADES NAS DIFERENTES ÁREAS DO DESENVOLVIMENTO DE SISTEMAS

Hiago Fraga Duarte Salicio Silveira

Universidade de Cuiabá (Unic)

Av. Manoel José de Arruda, n° 3100 - Jardim Europa, Cuiabá - MT, 78065-900

1- Resumo

Este artigo discute a importância do teste de software no ciclo de vida do desenvolvimento de sistemas, explorando suas aplicabilidades nas diferentes áreas, como testes unitários, de integração, de sistema e de segurança. Foi desenvolvido um sistema de cadastro simples em C# com testes simulando ataques de força bruta e injeção de SQL. Também foi utilizado um framework para automação de testes, sendo MSTest a escolhida. A metodologia adotada foi exploratória e prática, permitindo análise dos resultados obtidos em cenários de falha, possibilitando atitudes preventivas contras tais falhas. Os resultados destacam a importância da prevenção de vulnerabilidades e da garantia da manutenibilidade do código.

Palavras-chave: teste de software, segurança, automação de testes, desenvolvimento de sistemas, qualidade de software

2- Abstract

This paper discusses the importance of software testing throughout the software development lifecycle, exploring its applicability in various areas such as unit, integration, system, and security testing. A simple registration system was developed in C# with tests simulating brute force and SQL injection attacks. Additionally, a testing framework was employed, with MSTest being used for automation. The adopted methodology was exploratory and practical, enabling the analysis of results in failure scenarios and facilitating proactive measures against such failures. The results highlight the significance of preventing vulnerabilities and ensuring code maintainability.

Keywords: software testing, security, test automation, software development, software quality

3- Introdução

Com o avanço contínuo das tecnologias de software, torna-se indispensável garantir a qualidade das aplicações por meio de testes bem estruturados. Este artigo tem como objetivo apresentar a importância dos testes de software e analisar suas aplicabilidades em diferentes áreas do desenvolvimento de sistemas. A pesquisa propõe um estudo prático por meio da construção de um sistema de cadastro básico em C#, desenvolvido como uma aplicação de console (sem interface gráfica), permitindo a interação por meio de entrada e saída. No sistema, foram realizados testes funcionais e simulações de ataques, como força bruta e injeção de SQL, com o objetivo de validar a segurança e a robustez da aplicação.

Além disso, o estudo reforça a importância da aplicação sistemática de testes desde as etapas iniciais do desenvolvimento e a organização prévia do **Software Quality Assurance (SQA)**. O uso de **frameworks de automação de testes** também foi adotado para garantir eficiência na execução repetitiva de testes, destacando a relevância dessas ferramentas para a melhoria contínua do processo. O escopo do projeto limita-se à construção de uma aplicação de console, com foco nos aspectos lógicos e estruturais do código, sem a implementação de interfaces gráficas, o que facilita a compreensão dos testes em um ambiente mais controlado. O artigo está estruturado em cinco seções: a introdução, a revisão da literatura sobre testes de software, a metodologia utilizada no desenvolvimento do sistema, a apresentação e discussão dos resultados obtidos e, por fim, a conclusão, com as considerações finais e sugestões para trabalhos futuros.

4- Revisão da Literatura

4.1. Definição e Importância

Teste de software é um processo crítico dentro do ciclo de vida de desenvolvimento de software, que envolve a execução planejada e sistemática de um programa ou aplicação com o objetivo de identificar defeitos, erros ou não conformidades com os requisitos especificados. O teste abrange diferentes níveis, desde a avaliação de componentes individuais (teste de unidade) até a verificação do sistema completo (teste de sistema e aceitação), e busca garantir que o software entregue atenda aos padrões de qualidade esperados, com o menor esforço possível.

4.2. Objetivos do Teste de Software

O teste de software emerge como uma fase crucial no ciclo de vida do desenvolvimento de software, desempenhando um papel multifacetado na garantia da qualidade e confiabilidade dos sistemas. Essa perspectiva inicial estabelece o teste como uma investigação abrangente, que vai além da simples detecção de defeitos.

Segundo Pressman e Maxim (2016, p. 481), “De forma simplificada, o objetivo do teste é encontrar o maior número possível de erros com um esforço gerenciável durante um intervalo de tempo realístico.”. Essa otimização é essencial, considerando as restrições de tempo e recursos inerentes aos projetos de software. Assim, o teste de software não se limita a uma atividade aleatória, mas sim a uma abordagem planejada e sistemática, que exige estratégias e técnicas bem definidas.

No contexto do desenvolvimento, o teste idealmente inicia-se nas fases iniciais, com o projeto de testes ocorrendo concomitantemente ao projeto do software. Essa antecipação permite a identificação precoce de potenciais problemas, economizando tempo e recursos nas etapas subsequentes. Além disso, o teste abrange diferentes níveis e tipos, desde a avaliação de componentes individuais (teste de unidade) e a verificação da interação entre componentes (teste de integração) até a validação do sistema como um todo (teste de sistema e aceitação) e a garantia da proteção contra vulnerabilidades (teste de proteção/segurança). Essa amplitude garante que o produto final atenda aos requisitos especificados, às expectativas dos usuários e aos critérios de segurança.

4.3. Principais Tipos de Testes de Software

Existem diversas abordagens para testar o software, desde a verificação de unidades de código isoladas, passando pela integração entre módulos, até a validação do sistema como um todo. Além disso, é essencial garantir que o sistema seja seguro contra ataques, o que torna o **teste de segurança** uma parte crucial da estratégia de testes.

Nesta seção, exploraremos os principais tipos de testes de software, abordando suas características, objetivos e a importância de cada um para garantir a qualidade e o bom funcionamento do sistema ao longo de seu ciclo de vida.

4.3.1. Teste Unitário

O teste unitário é uma técnica de teste que foca na verificação de unidades individuais de código, como funções ou métodos, de forma isolada. Seu principal objetivo é garantir que cada parte do código esteja funcionando corretamente de maneira independente, o que facilita a identificação de falhas em estágios iniciais do desenvolvimento. Testes unitários bem feitos ajudam a manter o código modular e facilitam a manutenção, pois asseguram que as mudanças não quebrem funcionalidades existentes.

4.3.2. Teste de Integração

O **teste de integração** visa verificar a interação entre diferentes módulos ou componentes de um sistema. Após a validação das unidades de código, é essencial garantir que os módulos funcionem corretamente quando combinados. Esse tipo de teste foca na comunicação entre as partes do sistema, identificando falhas que podem surgir quando os módulos dependem uns dos outros para executar funções mais complexas. O teste de integração é fundamental para evitar erros de compatibilidade e garantir que o sistema como um todo funcione como esperado.

4.3.4. Teste de Sistema

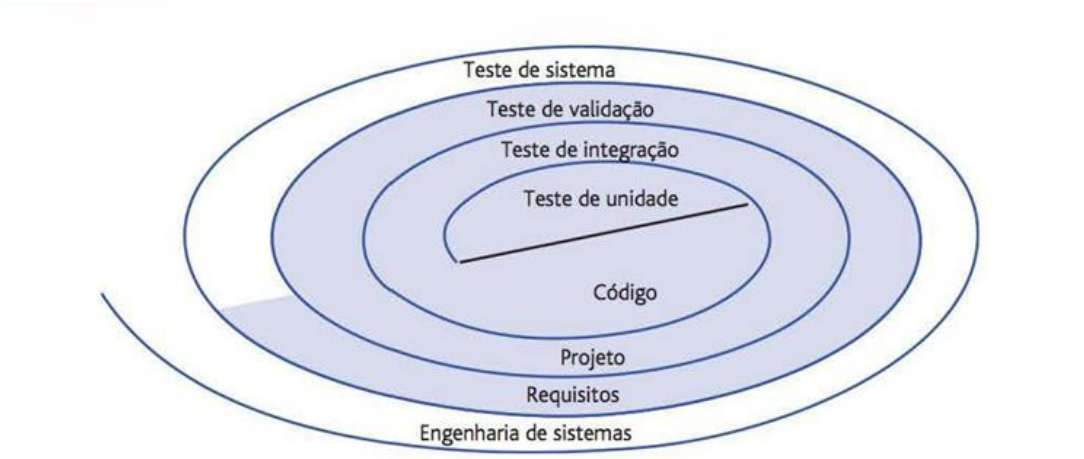
O **teste de sistema** envolve a validação do sistema completo, verificando se todas as partes do software, tanto individualmente quanto em conjunto, funcionam corretamente. Este teste busca garantir que o sistema atenda aos requisitos especificados, realizando uma verificação de ponta a ponta das funcionalidades. Ao testar o sistema como um todo, é possível identificar problemas de interação entre componentes e garantir que o software esteja pronto para ser entregue ao usuário final. Este é um teste crucial antes da liberação para produção.

4.3.5. Teste de Segurança

O **teste de segurança** tem como objetivo identificar vulnerabilidades no sistema que podem ser exploradas por atacantes. Esse tipo de teste simula diversas ameaças, como **ataques de força bruta**, **injeção de SQL**, e outros tipos de exploração, com o intuito de verificar a robustez da aplicação frente a possíveis falhas de segurança. A segurança é um aspecto crítico no desenvolvimento de sistemas, e a realização desses testes ajuda a proteger dados sensíveis e a garantir que a aplicação esteja segura contra ataques maliciosos.

4.3.6. Etapas de Teste

O processo de teste de software pode ser compreendido como uma sequência estruturada de etapas, cada uma focada em diferentes níveis de abstração e complexidade do sistema. Essa abordagem é bem representada na **Figura 4.3.6**, que ilustra o conceito da espiral de testes, em que se percorre do nível mais interno – o teste de unidade – até o nível mais externo – o teste de sistema –, à medida que se valida progressivamente o software.



Estratégia de teste representada na forma de espiral, demonstrando a progressão das etapas de validação no desenvolvimento de software. ***Fonte: Adaptado de PRESSMAN; MAXIM (2016, p. 476).***

A primeira etapa, o **teste de unidade**, concentra-se na validação de componentes individuais, como funções, métodos, classes ou objetos isolados. Seu principal objetivo é garantir que cada unidade do sistema opere conforme o esperado, de forma independente. A detecção precoce de defeitos nessa fase reduz custos e facilita a manutenção do software.

Prosseguindo para a próxima camada da espiral, temos o **teste de integração**, no qual o foco se desloca para a interação entre diferentes unidades ou módulos. Esta etapa verifica se as interfaces entre os componentes funcionam corretamente e se a comunicação entre eles é realizada sem erros, validando a estrutura de integração do sistema.

A terceira etapa é o **teste de validação**, que assegura que o sistema como um todo atende aos requisitos estabelecidos, especialmente aqueles provenientes da fase de modelagem. Aqui, os testes têm como objetivo confirmar se o software satisfaz as necessidades e expectativas do usuário final.

Por fim, o **teste de sistema** envolve a verificação do sistema completo em seu ambiente de operação real ou simulado. Essa etapa avalia se o software, integrado e validado, apresenta o comportamento correto diante de diferentes cenários de uso, desempenho e requisitos de segurança.

Essas etapas, quando seguidas de forma sistemática, proporcionam uma abordagem abrangente para a detecção de falhas, desde as mais simples até as mais complexas. Como destacado no conceito da espiral de testes, a evolução gradual de uma camada à outra permite o refinamento contínuo da qualidade do software, aumentando sua robustez e confiabilidade final.

4.4. Quality Assurance (QA)

Um dos grandes equívocos ainda presentes no desenvolvimento moderno é acreditar que as falhas e erros devem ser tratados apenas após a finalização da aplicação, promovendo correções de forma reativa e muitas vezes desorganizada. Essa abordagem "de trás para frente" ignora o potencial das estratégias conhecidas como *pre-fixing*, nas quais toda a equipe envolvida no projeto se compromete, desde as fases iniciais, com boas práticas de codificação, revisões constantes e testes rigorosos a cada nova funcionalidade adicionada ao sistema. Essa mentalidade preventiva permite identificar e corrigir falhas antes que elas se tornem problemas mais complexos.

Nesse contexto, o uso de metodologias como o **Quality Assurance (QA)** torna-se essencial. O QA atua de forma estruturada e contínua ao longo de todo o ciclo de vida do software, promovendo o planejamento, a verificação e a validação de cada etapa do desenvolvimento. Com isso, é possível escrever e depurar o software de forma integrada, evitando acúmulo de erros e promovendo entregas com maior qualidade e menor retrabalho. A integração incremental aliada ao QA representa, portanto, uma forma eficiente e moderna de construir sistemas robustos e confiáveis.

Quality Assurance (QA) é um processo sistemático que envolve a revisão e verificação da qualidade do software em todas as etapas de seu ciclo de vida, desde a concepção até a entrega. O objetivo do QA é garantir que o software atenda aos requisitos de qualidade estabelecidos, focando na **prevenção de falhas** e na melhoria contínua do produto. Embora o QA envolva várias práticas e metodologias, ele é essencial para criar uma base sólida de qualidade, assegurando que o software seja confiável, seguro e eficaz.

O **QA** impacta diretamente os testes de software, pois garante que as melhores práticas sejam seguidas ao longo do desenvolvimento e que o produto final seja testado adequadamente. Ele está intrinsecamente ligado à **qualidade** do processo de desenvolvimento, promovendo uma abordagem proativa para identificar e corrigir erros antes que eles impactem os usuários finais. Embora o conceito de QA englobue várias metodologias e técnicas, o foco principal é sempre a **prevenção** de defeitos e a **otimização** do processo de desenvolvimento.

5- Automatização de Testes

Uma das ferramentas práticas mais eficientes dentro da abordagem de *pre-fixing* do código é a **automação de testes**, que visa antecipar e simplificar a detecção de falhas ao longo do desenvolvimento. Por meio da automação, é possível criar rotinas que testam o comportamento do sistema de forma sistemática e repetitiva, economizando tempo e reduzindo significativamente o esforço necessário para encontrar e corrigir erros. Neste projeto, optou-se pela utilização de um framework de testes automatizados com o objetivo de demonstrar, na prática, a confiabilidade e a eficácia dessa técnica. A linguagem adotada no desenvolvimento do sistema oferece diversas

alternativas de frameworks, sendo o **MSTest** o escolhido por sua simplicidade, clareza na escrita dos testes e integração nativa com o ambiente de desenvolvimento. Vale destacar que, embora o MSTest tenha sido a ferramenta adotada neste caso, cada linguagem de programação possui frameworks distintos mais adequados para diferentes contextos e tipos de aplicação.

A automação de testes consiste na criação de scripts e rotinas automatizadas que verificam o comportamento esperado de funcionalidades do sistema, simulando entradas e avaliando as saídas de forma precisa e constante. Essa abordagem permite validar múltiplos cenários com agilidade, garantindo que alterações no código não introduzam novos defeitos – um conceito conhecido como *regressão*. Um dos grandes diferenciais da automação em relação aos testes manuais é sua capacidade de ser executada repetidamente a qualquer momento, com mínima intervenção humana, promovendo maior cobertura de testes e reduzindo o tempo de validação de funcionalidades.

Dentro do ciclo de desenvolvimento ágil, a automação se mostra especialmente vantajosa ao ser integrada a pipelines de integração contínua (CI), onde cada alteração no código dispara automaticamente uma sequência de testes. Essa prática assegura que falhas sejam detectadas ainda nas fases iniciais, evitando o acúmulo de problemas nas etapas finais do projeto.

No caso específico do **MSTest**, a estrutura de automação é baseada em atributos e métodos de teste que permitem definir casos de uso com clareza e organização. A ferramenta permite a criação de testes unitários e de integração, além de possibilitar a parametrização de entradas, verificação de exceções e organização dos testes por categorias. Sua integração com o Visual Studio e com ferramentas como Azure DevOps facilita a visualização dos resultados, o rastreamento de falhas e a manutenção dos testes ao longo do tempo.

Além disso, a automação contribui diretamente para a manutenibilidade do sistema, promovendo um desenvolvimento mais seguro e estruturado. Ela permite que desenvolvedores realizem refatorações com maior confiança, já que os testes automatizados funcionam como uma rede de segurança, alertando sempre que uma funcionalidade for comprometida. Assim, a automação de testes não apenas acelera o processo de validação, mas também eleva significativamente o nível de qualidade do software entregue.

6- Metodologia

O objetivo deste trabalho é realizar uma análise de segurança e qualidade de software em um sistema protótipo de cadastro e login, desenvolvido utilizando a linguagem de programação **C#** e o banco de dados **MySQL**. O sistema foi submetido a uma série de testes, incluindo simulações de ataques de segurança (SQL Injection e Força Bruta), bem como testes unitários e de integração, para avaliar suas vulnerabilidades e a eficácia das correções implementadas.

6.1. Desenvolvimento do Protótipo

O protótipo foi desenvolvido utilizando a linguagem de programação **C#** e a framework **.NET**, com o banco de dados **MySQL** para o armazenamento das credenciais de usuários. O sistema implementa funcionalidades básicas de cadastro e login, as quais, inicialmente, apresentam vulnerabilidades de segurança que foram propositadamente inseridas para possibilitar a realização de testes de segurança.

Código 6.1 – Cadastro de Usuário Vulnerável

```
public void CadastrarUsuario(){

Console.WriteLine("----- Cadastro de Usuário -----");
        var usuario = new Usuario();

        Console.Write("Nome: ");
        usuario.Nome = Console.ReadLine();

        Console.Write("Usuário: ");
        usuario.UsuarioLogin = Console.ReadLine();

        Console.Write("Senha: ");
        usuario.Senha = Console.ReadLine();

        using var connection = _dbConnection.GetConnection();

        // Consulta SQL vulnerável: valores do usuário são concatenados diretamente
       var query = $"INSERT INTO usuarios (nome, usuario_login, senha) VALUES(
        '{usuario.Nome}', '{usuario.UsuarioLogin}', '{usuario.Senha}')";
        using var command = new MySqlCommand(query, connection);

        command.ExecuteNonQuery();

        Console.WriteLine("\nUsuário cadastrado com sucesso!");
}
```

Neste código, a concatenação direta de parâmetros na consulta SQL representa uma vulnerabilidade de **SQL Injection**, permitindo que um atacante possa manipular a consulta SQL e acessar o banco de dados de maneira não autorizada.

Código 6.1 – Login Vulnerável

```
public void Login() {

        Console.WriteLine("----- Login -----");

        Console.Write("Usuário: ");
        var usuarioLogin = Console.ReadLine();

        Console.Write("Senha: ");
        var senha = Console.ReadLine();

        using var connection = _dbConnection.GetConnection();
        var query = "SELECT * FROM usuarios WHERE usuario_login = @UsuarioLogin AND
        senha =    @Senha";
        using var command = new MySqlCommand(query, connection);

        command.Parameters.AddWithValue("@UsuarioLogin", usuarioLogin);
        command.Parameters.AddWithValue("@Senha", senha);
        using var reader = command.ExecuteReader();

        if (reader.Read()) {

        var nome = reader.GetString("nome");
        Console.WriteLine($"\nBem-vindo, {nome}!");

        } else {

        Console.WriteLine("\nUsuário ou senha inválidos.");
        } }
```

O código de login vulnerável permite que um atacante realize tentativas ilimitadas de login, sem qualquer restrição no número de tentativas. Essa falha pode ser explorada para ataques de **Brute Force**, onde o atacante tenta repetidamente diferentes combinações de senhas até conseguir acessar o sistema. A falta de limitação de tentativas de login é o principal fator que torna o sistema vulnerável a esse tipo de ataque.

6.2 Vulnerabilidades Introduzidas

Foram propositadamente introduzidas falhas no sistema para possibilitar a análise da eficácia das medidas de segurança. As vulnerabilidades inseridas incluem:

- **SQL Injection**: O uso da concatenação direta de parâmetros na consulta SQL;
- **Armazenamento de Senhas em Texto Claro**: As senhas dos usuários são armazenadas no banco de dados sem qualquer mecanismo de criptografia;
- **Ausência de Limitação de Tentativas de Login**: O sistema não impede que um atacante realize múltiplas tentativas de login com credenciais incorretas, tornando-o vulnerável a ataques de força bruta.

Essas falhas permitiram a realização de testes de segurança que avaliaram a robustez do sistema frente a tentativas de ataque.

6.3 Testes Realizados

A pasta ***Test*** do repositório contém arquivos de teste que simulam ataques de segurança, além de testar a funcionalidade do sistema por meio de testes unitários e de integração. A seguir, são descritos os testes realizados, com a explicação das soluções implementadas para corrigir as vulnerabilidades identificadas.

6.3.1 Teste de Segurança: SQL Injection

O arquivo ***SqlInjectionTest.cs*** realiza um ataque simulado de **SQL Injection**, com o objetivo de verificar se o sistema é vulnerável a manipulação da consulta SQL.

Código 6.3.1 – Teste de Ataque de SQL Injection

```
public class SqlInjectionTest {
        public static void Run() {
                var usuarioService = new UsuarioService();

                Console.WriteLine("----- Teste de SQL Injection -----");

                // Gerar um valor único para o login
                 var uniqueLogin = "' OR '1'='1_" + Guid.NewGuid().ToString("N");

                 // Exibir o login gerado
                 Console.WriteLine($"Usuário criado para teste de SQL Injection: {uniqueLogin}");

                 // Salvar o login gerado em um arquivo
                 File.WriteAllText("sql_injection_test_user.txt", uniqueLogin);

                // Teste de SQL Injection no cadastro
                Console.WriteLine("\nTentando SQL Injection no cadastro...");

                  usuarioService.CadastrarUsuarioSimulado("Teste", uniqueLogin, "senha123");

                 // Teste de SQL Injection no login
                Console.WriteLine("\nTentando SQL Injection no login...");

                usuarioService.LoginSimulado(uniqueLogin, "senha123");
        } }
```

Este código simula uma tentativa de login com um payload de **SQL Injection**. Se o sistema for vulnerável, o login será realizado com sucesso, ignorando a autenticação. Para corrigir a vulnerabilidade de SQL Injection, a solução adotada foi a utilização de **queries parametrizadas**, que asseguram que os parâmetros fornecidos pelo usuário sejam tratados de forma segura, evitando a execução de comandos maliciosos. Abaixo, segue a implementação corrigida.

Código 6.3.1 - Parte da implementação de Cadastro Corrigida

```
public void CadastrarUsuario( ){

        (...)

        using var connection = _dbConnection.GetConnection();
        var query = "INSERT INTO usuarios (nome, usuario_login, senha) VALUES (@Nome,
                @UsuarioLogin,@Senha)";

        using var command = new MySqlCommand(query, connection);

        command.Parameters.AddWithValue("@Nome", usuario.Nome);
        command.Parameters.AddWithValue("@UsuarioLogin", usuario.UsuarioLogin);
        command.Parameters.AddWithValue("@Senha", usuario.Senha);

        command.ExecuteNonQuery();

        Console.WriteLine("\nUsuário cadastrado com sucesso!");
        }
```

A solução consistiu na substituição da concatenação de strings por **parâmetros na query**, o que garante que os dados inseridos pelo usuário sejam tratados de forma segura.

6.3.2. Teste de Segurança: Força Bruta

O arquivo ***BruteForce.cs*** realiza um ataque simulado de **força bruta**, tentando diversas combinações de senhas para verificar a eficácia do sistema contra esse tipo de ataque.

Código 6.3.2 - Teste de Ataque de Força Bruta

```
public static void Run() {

                        var usuarioService = new UsuarioService();
                        Console.WriteLine("----- Teste de Brute Force -----");

                        // Simular tentativas de login com várias senhas
                        var usuario = "usuarioTeste";  // Nome de usuário correto
                        var senhaCorreta = "senhaCorreta"; // Senha correta

                        // Cadastrar o usuário com os dados corretos
                        usuarioService.CadastrarUsuarioSimulado("Teste", usuario,senhaCorreta);

                        for (int i = 0; i < 10; i++) {
                        string senha;

                        if (i < 9) {
                        // Gerar senhas incorretas para as primeiras 9 tentativas
                        senha = $"senha{i}";}
                        else
                        {
                         // Usar a senha correta na 10ª tentativa
                        senha = senhaCorreta;
                        }

                        Console.WriteLine($"Tentativa {i + 1}: Usuário: {usuario}, Senha:{senha}");
                        usuarioService.LoginSimulado(usuario, senha);
        }
        }
```

O código realiza tentativas automatizadas de login, utilizando diversas combinações de senhas, a fim de verificar se o sistema pode ser comprometido por um ataque de força bruta. Para prevenir

ataques de força bruta, foi implementada uma **limitação de tentativas de login**, permitindo apenas um número reduzido de tentativas consecutivas. Além disso, caso o usuário atinja o limite de tentativas falhas, o sistema **bloqueia o acesso ao login por um período de 15 minutos**, impedindo novas tentativas durante esse intervalo. Essa abordagem visa dificultar ataques automatizados, protegendo o sistema contra tentativas sucessivas de quebra de senha, conforme mostrado abaixo.

Código 6.3.2 - Parte da implementação de Login Corrigida

```
public void Login( ){

(...)
        tentativasFalhas++;
        var updateAttemptsQuery = "UPDATE usuarios SET tentativas_falhas = @TentativasFalhas
        WHERE usuario_login = @UsuarioLogin";

        if (tentativasFalhas >= 5) {

        updateAttemptsQuery = "UPDATE usuarios SET tentativas_falhas = @TentativasFalhas,
        bloqueado_ate = @BloqueadoAte WHERE usuario_login = @UsuarioLogin";
        }

        using var updateCommand = new MySqlCommand(updateAttemptsQuery, connection);
        updateCommand.Parameters.AddWithValue("@TentativasFalhas", tentativasFalhas);
        updateCommand.Parameters.AddWithValue("@UsuarioLogin", usuarioLogin);

        if (tentativasFalhas >= 5) {

        updateCommand.Parameters.AddWithValue("@BloqueadoAte",
        DateTime.Now.AddMinutes(15));  // Bloquear por 15 minutos
        }

        updateCommand.ExecuteNonQuery();
        (...)
}
```

Essa abordagem bloqueia a conta do usuário temporariamente após cinco tentativas de login inválidas, o que impede a realização de ataques automatizados, alem de bloquear o usuário durante 15 minutos, impedindo o acesso.

6.3.3 Teste Unitário

O teste unitário apresentado no arquivo ***UsuarioServiceTest.cs*** foi desenvolvido utilizando o framework **MSTest**, e tem como objetivo verificar a robustez do método de cadastro de usuário simulado. O teste assegura que o método de cadastro (CadastrarUsuarioSimulado) funcione corretamente e que não lance exceções inesperadas durante a operação, garantindo maior confiabilidade ao sistema.

Código 6.3.3 - Teste Unitário com MSTest

```
[TestClass]
public class UsuarioServiceTests {

        private UsuarioService _usuarioService = null!;

        [TestInitialize]
        public void Setup()
        {
        usuarioService = new UsuarioService();
        }

        [TestMethod]
        public void CadastrarUsuarioSimulado_ShouldNotThrowException() {

                // Arrange
                var nome = "Teste";
                var usuarioLogin = "usuarioTeste_" + Guid.NewGuid().ToString("N");
                var senha = "senha123";

                // Act & Assert
                try {
                usuarioService.CadastrarUsuarioSimulado(nome, usuarioLogin, senha);
                Assert.IsTrue(true); // Passa se não lançar exceção
                } catch {
                Assert.Fail("O método lançou uma exceção inesperada.");
                }
        }
}
```

O teste ***CadastrarUsuarioSimulado_ShouldNotThrowException*** cria dinamicamente um usuário de teste e executa o método de cadastro, verificando se o processo ocorre sem gerar exceções. Caso alguma exceção inesperada seja lançada, o teste falha, indicando que o método de cadastro não está se comportando de forma confiável. Esse tipo de teste é fundamental para validar a estabilidade de funcionalidades críticas do sistema.

6.3.4 Teste de Integração

O arquivo ***IntegrationTest.cs*** realiza um teste de integração, simulando a interação real entre a aplicação e o banco de dados, verificando se o processo de cadastro e login funciona de maneira eficaz.

Código 6.3.4 - Teste de Integração com MSTest

```
namespace cadastro_test_analysis.Tests {

        [TestClass]
        public class IntegrationTests {
                private UsuarioService _usuarioService = null!;

                [TestInitialize]
                public void Setup()
                {
                usuarioService = new UsuarioService();
                }
                [TestMethod]
                public void CadastroELogin_DeveFuncionarCorretamente() {

                        // Arrange
                        var nome = "Teste";
```

```
                var usuarioLogin = "usuarioTeste_" + Guid.NewGuid().ToString("N");
                var senha = "senha123";

                // Act
                _usuarioService.CadastrarUsuarioSimulado(nome, usuarioLogin, senha);
                var output = new StringWriter();
                Console.SetOut(output);
                usuarioService.LoginSimulado(usuarioLogin, senha);

                // Assert
                var consoleOutput = output.ToString();
                Assert.IsTrue(consoleOutput.Contains("Bem-vindo"));
            }
        }
    }
```

Este teste de integração valida se o usuário, após ser cadastrado no banco de dados, consegue realizar o login corretamente. Esse teste assegura que a aplicação está interagindo de forma adequada com o banco de dados.

7- Resultados

Os testes realizados permitiram uma avaliação abrangente da segurança, estabilidade e funcionamento do sistema protótipo de cadastro e login.

Inicialmente, o teste de **segurança contra SQL Injection** demonstrou que, na versão vulnerável, era possível manipular as consultas SQL para autenticar usuários sem credenciais válidas. Após a implementação de consultas parametrizadas, o sistema passou a tratar adequadamente as entradas de dados, prevenindo ataques desse tipo. Caso essa vulnerabilidade não tivesse sido corrigida, um invasor poderia acessar informações confidenciais, alterar dados de usuários ou até mesmo derrubar o banco de dados, gerando perda de dados e exposição de informações sensíveis.

O teste de **força bruta** identificou que o sistema, antes da correção, permitia tentativas ilimitadas de login sem qualquer restrição. A solução adotada consistiu em limitar o número de tentativas consecutivas de autenticação e, ao atingir o limite permitido, o sistema bloqueia o acesso do usuário por um período de 15 minutos, dificultando ataques automatizados. Sem essa proteção, um atacante poderia comprometer contas de usuários legítimos utilizando softwares de ataque automático, resultando em acessos não autorizados, roubo de identidade e danos à reputação da aplicação.

Os **testes unitários**, desenvolvidos com o framework MSTest, confirmaram que os métodos de cadastro de usuários funcionam corretamente e que não ocorrem exceções inesperadas durante sua execução. Esses testes contribuíram para validar o comportamento isolado de componentes críticos do sistema, assegurando a estabilidade da aplicação em operações básicas. A ausência de testes unitários poderia levar à liberação de funcionalidades defeituosas, o que resultaria em falhas inesperadas durante o uso, impactando diretamente a experiência do usuário e aumentando custos de manutenção.

O **teste de integração** validou a interação real entre o sistema e o banco de dados. Através dele, verificou-se que, após o cadastro de um usuário, era possível efetuar login utilizando as credenciais

criadas, evidenciando que os módulos de cadastro e autenticação estavam integrados e funcionando corretamente. Sem a realização de testes de integração, erros de comunicação entre diferentes partes do sistema poderiam passar despercebidos, causando falhas graves em ambientes de produção, como registros de usuários inconsistentes e impossibilidade de acesso ao sistema.

De forma geral, os resultados obtidos reforçam a importância da aplicação sistemática de diferentes tipos de testes, demonstrando que a detecção e correção de falhas em fases iniciais de desenvolvimento contribuem significativamente para o aumento da segurança, estabilidade e confiabilidade do sistema. Além disso, evidencia-se que a negligência na execução de práticas adequadas de teste pode resultar em prejuízos financeiros, danos à imagem da organização e comprometimento da confiança dos usuários.

A Tabela/Quadro a seguir resume os principais testes realizados, as falhas identificadas, as ações corretivas aplicadas e as potenciais consequências que seriam enfrentadas caso as correções não fossem implementadas.

Quadro 1 – Resumo dos Testes Realizados, Soluções Adotadas e Consequências Possíveis

Tipos de Teste	**Falha Encontradas**	***Solução Aplicada***	***Consequências***
Teste de SQL Injection	Concatenação direta de strings em consultas SQL	Uso de consultas parametrizadas com MySqlCommand	Acesso indevido a dados, alteração ou exclusão de informações
Teste de Força Bruta	Tentativas de login ilimitadas sem bloqueio	Limitação de tentativas e bloqueio do usuário por 15 minutos	Invasão de contas, roubo de identidade, exposição de dados
Testes Unitários	Verificação de métodos sem validação de exceções	Implementação de testes MSTest para detectar falhas	Liberação de funcionalidades instáveis ou defeituosas
Teste de Integração	Falha potencial de comunicação entre módulos	Teste de integração para validar cadastro e login reais	Erros operacionais, impossibilidade de cadastro ou login

8- Discussão

A análise aprofundada dos testes executados neste estudo revelou, de forma contundente, a indispensabilidade da implementação de práticas robustas de segurança desde as fases primordiais do desenvolvimento de software. Os resultados empíricos evidenciaram como a ausência de testes de segurança eficazes pode criar lacunas propícias a vulnerabilidades críticas, tais como ataques de SQL Injection e de força bruta, representando um risco substancial à integridade e à confidencialidade dos dados.

Ademais, a adoção estratégica de testes automatizados demonstrou ser um fator determinante para aprimorar a confiabilidade do software e otimizar os processos de manutenção. A capacidade de realizar testes de maneira repetida e sistemática, com mínima intervenção humana, não apenas proporciona economia de tempo e recursos, mas também contribui para a identificação precoce de defeitos e a prevenção eficaz de regressões.

Em consonância com pesquisas anteriores, este estudo corrobora a persistente preocupação com a negligência em relação aos testes de segurança no contexto do desenvolvimento de software. Tal constatação sublinha a necessidade premente de disseminar o conhecimento sobre testes de software e de promover uma cultura de conscientização entre os desenvolvedores acerca da importância crítica da segurança em seus projetos.

9- Conclusão

Este artigo apresentou uma investigação prática sobre a aplicação de testes de software, reiterando sua importância fundamental para assegurar a qualidade, a segurança e a manutenibilidade dos sistemas de software. A abordagem prática adotada neste estudo facilitou a visualização clara e objetiva dos impactos adversos decorrentes da ausência de testes adequados.

Como recomendação para pesquisas futuras, sugere-se a expansão deste estudo por meio da incorporação de ferramentas mais avançadas de automação e segurança, bem como a integração contínua e sistemática dos testes em pipelines de desenvolvimento.

10- Referências

PRESSMAN, Roger S.; MAXIM, Bruce R. Engenharia de Software: uma abordagem profissional. 8. ed. Porto Alegre: AMGH, 2016.

Galin, D. (2004). Software Quality Assurance: From Theory to Implementation. Boston: Addison-Wesley.

Hiago Fraga Duarte Salicio Silveira. Sistema de Cadastro e Login com Testes de Segurança. GitHub.https://www.google.com/searchq=https://github.com/duarteHiago/cadastro_test_analysis Acessado em 26 de abril de 2024.

Microsoft Docs. Unit testing C# with MSTest. Disponível em: https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest