# Relatório de testes de segurança na aplicação PeçaAi

## Introdução

A segurança de aplicações é um pilar fundamental no desenvolvimento de software moderno, especialmente em um cenário onde a proteção de dados e a privacidade dos usuários são imperativos legais e éticos. A realização de testes de segurança abrangentes permite identificar e mitigar vulnerabilidades que poderiam ser exploradas por agentes mal-intencionados, resultando em perdas financeiras, danos à reputação e, crucialmente, na exposição de dados sensíveis.

Este documento detalha os diferentes tipos de testes executados, incluindo a verificação de dependências, o escaneamento de código estático, testes específicos no backend da aplicação (como autenticação, gerenciamento de variáveis de ambiente e segurança de uploads) e a utilização da ferramenta OWASP ZAP para uma análise dinâmica de segurança. Para cada tipo de teste, serão apresentadas as vulnerabilidades encontradas, os riscos associados, exemplos práticos (quando aplicável) e as recomendações para correção.

A análise dos resultados foi confrontada com os princípios e exigências da Lei Geral de Proteção de Dados (LGPD - Lei nº 13.709/2018), destacando a importância de cada teste na garantia da segurança, confidencialidade, integridade e na prevenção contra o vazamento de dados pessoais. O relatório visa fornecer uma visão clara e didática das fragilidades identificadas e das medidas necessárias para fortalecer a segurança do software, promovendo a conformidade com a legislação vigente e as melhores práticas de desenvolvimento seguro.

A metodologia adotada para a realização dos testes de segurança neste software buscou abranger diferentes facetas da segurança de aplicações, combinando análises estáticas e dinâmicas, bem como a verificação de configurações e componentes de backend. O objetivo foi identificar um espectro amplo de vulnerabilidades potenciais, desde falhas em dependências de terceiros até configurações inadequadas de segurança e brechas no código da aplicação.

Os testes foram divididos nas seguintes etapas principais:

1. **Verificação de Dependências:** Análise automatizada das bibliotecas e pacotes de terceiros utilizados no projeto, tanto no frontend quanto no backend. O objetivo desta etapa é identificar dependências com vulnerabilidades conhecidas (CVEs - Common Vulnerabilities and Exposures) que poderiam ser exploradas.
2. **Escaneamento de Código Estático (SAST - Static Application Security Testing):** Utilização de ferramentas para analisar o código-fonte da aplicação sem executá-lo. Neste caso, foi empregado o ESLint com plugins de segurança para identificar padrões de código que podem levar a vulnerabilidades comuns, como Cross-Site Scripting (XSS), injeção de NoSQL, e exposição de informações sensíveis.
3. **Testes de Backend:** Avaliação manual e automatizada de componentes críticos do backend, incluindo:
   * **Autenticação e Autorização:** Verificação da robustez dos mecanismos de login e controle de acesso, garantindo que usuários não autorizados não consigam acessar funcionalidades ou dados restritos.
   * **Gerenciamento de Variáveis de Ambiente:** Checagem da proteção de arquivos .env e outras formas de armazenamento de credenciais e chaves de API, prevenindo sua exposição acidental.
   * **Segurança de Uploads de Arquivos:** Análise da configuração do sistema de upload para prevenir o envio de arquivos maliciosos ou de tipos não permitidos, além de verificar limitações de tamanho.
   * **Configuração de Endpoints da API:** Teste para identificar endpoints inseguros, como aqueles que permitem acesso sem autenticação adequada ou que expõem dados sensíveis desnecessariamente.
4. **Testes Dinâmicos de Segurança de Aplicação (DAST - Dynamic Application Security Testing):** Utilização da ferramenta OWASP ZAP (Zed Attack Proxy) para realizar um escaneamento automatizado da aplicação em execução. O ZAP simula ataques comuns e inspeciona as respostas do servidor para identificar vulnerabilidades como:
   * Ausência ou má configuração de cabeçalhos de segurança (CSP, X-Frame-Options, X-Content-Type-Options).
   * Vulnerabilidades de Cross-Site Scripting (XSS) refletido e armazenado.
   * Divulgação de informações sensíveis em URLs ou comentários no código.

Para cada vulnerabilidade identificada, buscou-se documentar o problema, o risco associado, fornecer um exemplo (quando pertinente) e propor uma solução ou recomendação para mitigação. A análise dos resultados levou em consideração as boas práticas de desenvolvimento seguro e os requisitos da Lei Geral de Proteção de Dados (LGPD), focando na proteção da confidencialidade, integridade e disponibilidade dos dados processados pela aplicação.

**Verificação de Dependências**

A verificação de dependências é um processo crucial na manutenção da segurança de um software. Softwares modernos frequentemente utilizam um grande número de bibliotecas e pacotes de terceiros para acelerar o desenvolvimento e adicionar funcionalidades. No entanto, essas dependências podem, elas mesmas, conter vulnerabilidades que, se não identificadas e corrigidas, expõem a aplicação a riscos significativos.

**Problemas Identificados:**

Conforme a análise realizada, foram detectadas as seguintes vulnerabilidades nas dependências do frontend do projeto:

* **Vulnerabilidade Alta:**
  + **Axios:** Identificada uma vulnerabilidade de alta criticidade. Versões desatualizadas do Axios podem ser suscetíveis a ataques de Server-Side Request Forgery (SSRF) e vazamento de credenciais, especialmente quando URLs absolutas são utilizadas em requisições. Isso ocorre porque um atacante poderia induzir o servidor a realizar requisições para URLs externas controladas por ele.
* **Vulnerabilidades Moderadas:**
  + **Babel, Esbuild e Vite:** Foram encontradas três vulnerabilidades de criticidade moderada relacionadas a estes pacotes. Essas falhas geralmente estão associadas a problemas de segurança no código gerado ou no servidor de desenvolvimento, podendo levar a comportamentos inesperados ou brechas de segurança se não atualizadas.

No backend, a ferramenta npm audit também identificou uma vulnerabilidade de alta severidade no pacote axios, reiterando o risco de SSRF e vazamento de credenciais já mencionado para o frontend.

**Riscos Associados:**

As vulnerabilidades em dependências podem levar a uma variedade de ataques, incluindo:

* **Execução Remota de Código (RCE):** Algumas vulnerabilidades permitem que um atacante execute código arbitrário no servidor ou no cliente.
* **Cross-Site Scripting (XSS):** Falhas em bibliotecas de frontend podem permitir a injeção de scripts maliciosos nas páginas visualizadas pelos usuários.
* **Negação de Serviço (DoS):** Dependências vulneráveis podem ser exploradas para indisponibilizar a aplicação.
* **Vazamento de Dados:** Credenciais, chaves de API ou dados de usuários podem ser expostos através de falhas nas bibliotecas.
* **Server-Side Request Forgery (SSRF):** Como no caso do Axios, a aplicação pode ser forçada a fazer requisições não autorizadas a sistemas internos ou externos.

**Exemplos e Recomendações de Correção:**

Para as vulnerabilidades identificadas, a principal recomendação é a atualização das dependências para versões corrigidas. Os seguintes comandos foram sugeridos para a atualização no ambiente do projeto (dentro da pasta client/ para o frontend e na raiz do backend para o axios do servidor):

* **Para o frontend:**
* npm update axios --save  
  npm update @babel/helpers --save  
  npm update esbuild --save  
  npm update vite --save
* Alternativamente, para tentar corrigir todas as vulnerabilidades de dependências automaticamente, pode-se usar:
* npm audit fix --force
* **Para o backend (especificamente o axios):**
* npm audit fix
* Caso o problema persista, forçar a atualização para a versão mais recente:
* npm install axios@latest

Após a execução desses comandos, é fundamental rodar novamente o comando npm audit para verificar se as vulnerabilidades foram efetivamente sanadas.

A manutenção regular das dependências, utilizando ferramentas como npm audit (para projetos Node.js/JavaScript) ou equivalentes em outras linguagens (como dependency-check do OWASP, Snyk, Dependabot do GitHub), é uma prática essencial de segurança. Essas ferramentas ajudam a identificar proativamente dependências desatualizadas ou com vulnerabilidades conhecidas, permitindo que as equipes de desenvolvimento tomem medidas corretivas antes que explorações ocorram. ### Escaneamento de Código com ESLint

O escaneamento de código estático (SAST) é uma técnica de teste de segurança que analisa o código-fonte de uma aplicação em busca de vulnerabilidades sem a necessidade de executá-lo. Para este projeto, foi utilizada a ferramenta ESLint, um popular linter para JavaScript, em conjunto com o plugin eslint-plugin-security. Esta combinação ajuda a identificar padrões de código que podem levar a falhas de segurança comuns, como NoSQL Injection, Cross-Site Scripting (XSS), e exposição de informações sensíveis.

**Configuração e Execução do ESLint:**

Para realizar o escaneamento, os seguintes passos foram recomendados:

1. **Instalação de Dependências (caso não existam):**

* npm install eslint eslint-plugin-security --save-dev

1. **Criação ou Atualização do Arquivo de Configuração do ESLint (.eslintrc.json):** Na raiz do projeto, o arquivo de configuração do ESLint deve incluir a extensão de segurança:

* {  
   "extends": ["plugin:security/recommended"]  
  }

1. **Execução do ESLint:** O comando para executar a análise no projeto, abrangendo arquivos JavaScript, JSX, TypeScript e TSX, é:

* npx eslint . --ext .js,.jsx,.ts,.tsx

**Resultados do Escaneamento com ESLint:**

O escaneamento realizado com o ESLint identificou 39 problemas no código. No entanto, a análise inicial indicou que a maioria desses problemas não se configurava como falhas de segurança críticas, mas sim como questões de qualidade de código ou potenciais bugs. Os principais tipos de erros encontrados foram:

* **Variáveis importadas e não utilizadas (no-unused-vars):** Por exemplo, a importação do React em arquivos onde ele não era explicitamente necessário (comum em versões mais recentes do React onde a importação global não é sempre exigida para JSX).
* **Propriedades de componentes não validadas (react/prop-types):** Componentes React recebendo propriedades (props) sem a devida validação utilizando PropTypes. Por exemplo, propriedades como \_id, name, onClose não estavam sendo validadas.
* **Dependências ausentes em useEffect hooks:** O hook useEffect do React não listava todas as suas dependências no array de dependências, o que pode levar a execuções inesperadas ou a não atualização correta do estado do componente.

**Falhas de Segurança Encontradas:**

É importante notar que, segundo o relatório dos testes, **o escaneamento com ESLint e eslint-plugin-security não apontou falhas graves de segurança como NoSQL Injection ou XSS diretamente**. Isso não significa que a aplicação esteja imune a esses ataques, mas que os padrões de código específicos que o plugin de segurança do ESLint procura não foram detectados em grande escala ou com criticidade alta.

Para uma análise de segurança de código estático mais profunda e abrangente, especialmente na detecção de vulnerabilidades complexas como injeções e XSS, seria recomendável o uso de ferramentas SAST mais especializadas, como SonarQube ou Snyk Code, que possuem motores de análise mais robustos e um conjunto maior de regras de detecção.

**Recomendações de Correção para os Problemas Apontados pelo ESLint:**

Embora não sejam vulnerabilidades de segurança diretas, a correção dos problemas apontados pelo ESLint melhora a qualidade, a manutenibilidade e a robustez do código, podendo prevenir bugs que indiretamente levem a problemas de segurança.

1. **Corrigir imports não usados:** Remover as linhas de importação de variáveis ou módulos que não estão sendo utilizados no arquivo. Por exemplo, remover import React from 'react'; se o React não for usado explicitamente.
2. **Adicionar validação de propriedades (prop-types):** Implementar a validação das props em todos os componentes React. Exemplo para um componente ConfirmDelete.jsx:

* import PropTypes from 'prop-types';  
    
  // ... código do componente ...  
    
  ConfirmDelete.propTypes = {  
   \_id: PropTypes.string.isRequired,  
   name: PropTypes.string.isRequired,  
   onClose: PropTypes.func.isRequired  
  };
* Esta prática garante que os componentes recebam os tipos de dados esperados, prevenindo erros em tempo de execução.

1. **Corrigir dependências do useEffect:** Revisar os hooks useEffect e garantir que todas as variáveis e funções externas referenciadas dentro do efeito estejam listadas em seu array de dependências. Por exemplo, se o ESLint sugere adicionar navigate ao array, a correção seria:

* useEffect(() => {  
   // código que usa navigate  
  }, [navigate]); // Adicionar navigate ao array de dependências

Em resumo, o ESLint com eslint-plugin-security serve como uma primeira linha de defesa na análise estática, ajudando a manter a higiene do código e a identificar alguns padrões problemáticos. No entanto, para uma cobertura de segurança SAST mais completa, deve ser complementado por ferramentas mais especializadas. ### Testes de Segurança do Backend

A segurança do backend é crítica para a proteção dos dados e da lógica de negócios de uma aplicação. Os testes realizados no backend focaram em identificar vulnerabilidades comuns que poderiam ser exploradas por atacantes para comprometer o sistema. As principais áreas investigadas foram:

**1. Dependências Desatualizadas:**

Conforme detalhado na seção “Verificação de Dependências”, foi identificada uma vulnerabilidade de alta severidade no pacote axios utilizado no backend. A correção envolve a atualização do pacote para a versão mais recente e segura, utilizando os comandos:

npm audit fix  
# ou, se necessário:  
npm install axios@latest

Manter as dependências do backend atualizadas é fundamental para evitar a exploração de falhas já conhecidas.

**2. Exposição de Variáveis Sensíveis (Proteção do arquivo .env):**

Arquivos .env frequentemente contêm informações críticas como chaves de API, credenciais de banco de dados e segredos de aplicação. É vital que este arquivo nunca seja versionado (enviado para repositórios Git) ou exposto publicamente.

* **Verificação:** Foi recomendado verificar o arquivo .gitignore na raiz do projeto backend para garantir que a entrada .env esteja presente.
* cat .gitignore
* **Recomendação:** Se .env não estiver listado no .gitignore, deve ser adicionado manualmente para prevenir o versionamento acidental:
* .env

**3. Falhas de Autenticação e Autorização em Endpoints da API:**

A correta implementação de mecanismos de autenticação (quem é o usuário) e autorização (o que o usuário pode fazer) é essencial para proteger os recursos da API.

* **Problemas Identificados:**
  + **Rotas de Modificação/Exclusão sem Proteção:** As rotas PUT /item/:id e DELETE /item/:id no arquivo cardapioRoute.js não possuíam o authMiddleware. Isso permitiria que qualquer usuário, mesmo não autenticado ou sem as permissões adequadas, pudesse editar ou excluir itens do cardápio.
  + **Rotas de Cadastro sem Validação Adicional:** As rotas POST /cadastroEstabelecimento e POST /cadastroCliente em userRoute.js permitiam a criação de contas sem validações extras (como confirmação de e-mail ou CAPTCHA). Isso pode ser explorado por bots para criar contas em massa (spam).
* **Recomendações de Correção:**
  + **Aplicar authMiddleware:** Adicionar o middleware de autenticação (authMiddleware) às rotas PUT /item/:id e DELETE /item/:id para garantir que apenas usuários autenticados e autorizados possam realizar essas operações.
  + **Implementar Validações no Cadastro:** Introduzir mecanismos como verificação de e-mail por token ou a utilização de CAPTCHA nas rotas de cadastro para mitigar o risco de criação de contas por bots e spam.

**4. Exposição de Dados Sensíveis em Respostas da API:**

APIs podem inadvertidamente expor mais dados do que o necessário, incluindo informações sensíveis.

* **Problema Identificado:**
  + A rota GET / em userRoute.js, que retorna informações do usuário autenticado, não possuía um controle granular sobre quais dados eram expostos. Isso poderia levar ao vazamento de informações como senhas hasheadas (embora hasheadas, não deveriam ser expostas), tokens internos ou outros dados privados.
* **Recomendações de Correção:**
  + **Retornar Apenas Dados Essenciais:** Modificar a rota para retornar somente as informações estritamente necessárias para o cliente, evitando a exposição de senhas (mesmo hasheadas), tokens de sessão internos, ou outros campos sensíveis.
  + **Utilizar DTOs (Data Transfer Objects):** Implementar DTOs para modelar explicitamente os dados que são enviados nas respostas da API. Isso ajuda a filtrar e controlar a estrutura dos dados expostos.

**5. Upload de Arquivos sem Verificação Adequada:**

O upload de arquivos é uma funcionalidade comum, mas se não for devidamente securizada, pode ser um vetor para diversos ataques, como o upload de shells maliciosas ou arquivos que causem negação de serviço.

* **Problema Identificado:**
  + As rotas POST /item e PUT /item/:id em cardapioRoute.js, que lidam com upload de imagens, utilizavam uma configuração do multer (multerConfig.js) que poderia permitir o upload de arquivos inseguros ou de tipos não esperados.
* **Recomendações de Correção:**
  + **Filtragem de Tipos de Arquivo e Limitação de Tamanho:** No arquivo multerConfig.js, adicionar um filtro para permitir apenas tipos de arquivo específicos (ex: apenas imagens como JPEG, PNG) e impor um limite de tamanho razoável para os arquivos. Exemplo de configuração do multer com filtro e limite:
  + const multer = require('multer');  
      
    const storage = multer.diskStorage({ /\* ... sua configuração de storage ... \*/ });  
      
    const fileFilter = (req, file, cb) => {  
     if (file.mimetype.startsWith("image/")) { // Aceita qualquer tipo de imagem  
     cb(null, true);  
     } else {  
     cb(new Error("Apenas imagens são permitidas!"), false);  
     }  
    };  
      
    const upload = multer({  
     storage: storage,  
     fileFilter: fileFilter,  
     limits: { fileSize: 2 \* 1024 \* 1024 } // Limite de 2MB por arquivo  
    });

**6. Riscos de Injeção (SQL Injection, NoSQL Injection) e XSS no Backend:**

A falta de validação e sanitização adequadas das entradas do usuário é uma das causas mais comuns de vulnerabilidades de injeção e XSS.

* **Problema Identificado:**
  + Foi apontada a falta de validações e sanitizações robustas nas entradas dos usuários em diversas partes do backend. Isso abre brechas para que entradas maliciosas possam explorar vulnerabilidades de Injeção de SQL (se um banco SQL relacional for usado e as queries forem construídas por concatenação de strings) ou NoSQL Injection (para bancos NoSQL como MongoDB, se os inputs não forem devidamente tratados antes de serem usados em queries), além de XSS armazenado se o backend refletir dados não sanitizados.
* **Recomendações de Correção:**
  + **Sanitização e Validação Rigorosa:**
    - Para MongoDB (com Mongoose): Sempre sanitize os inputs que vêm do usuário antes de usá-los em queries. O Mongoose possui mecanismos de sanitização e validação de schema que devem ser utilizados.
    - Para SQL: Utilizar sempre queries parametrizadas (prepared statements) em vez de concatenar strings para montar as queries. Isso previne a maioria dos ataques de SQL Injection.
  + **Utilizar Bibliotecas de Validação:** Empregar bibliotecas como express-validator para validar e sanitizar todas as entradas do usuário de forma sistemática. Exemplo de uso com express-validator:
  + const { body, validationResult } = require("express-validator");  
      
    routeUser.post("/cadastroCliente", [  
     body("email").isEmail().withMessage("E-mail inválido").normalizeEmail(),  
     body("senha").isLength({ min: 6 }).withMessage("Senha deve ter pelo menos 6 caracteres")  
     // Adicionar mais validações conforme necessário  
    ], (req, res) => {  
     const errors = validationResult(req);  
     if (!errors.isEmpty()) {  
     return res.status(400).json({ errors: errors.array() });  
     }  
     // Prosseguir com a lógica de cadastro  
     cadastrarUsuarioCliente(req, res);  
    });

**7. Métodos HTTP Permitidos sem Restrição:**

Permitir que endpoints aceitem métodos HTTP desnecessários (ex: um endpoint que só precisa de GET aceitando POST, PUT, DELETE) pode aumentar a superfície de ataque.

* **Problema Identificado:**
  + Algumas rotas da API poderiam estar permitindo todos os métodos HTTP, sem uma restrição explícita aos métodos que são de fato utilizados e necessários para aquela funcionalidade.
* **Recomendação de Correção:**
  + **Middleware para Limitar Métodos:** No servidor principal da aplicação (ex: app.js ou index.js no Express), definir um middleware global ou específico por rota para limitar os métodos HTTP permitidos. Exemplo:
  + app.use((req, res, next) => {  
     const allowedMethods = ["GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS"]; // Ajustar conforme os métodos realmente usados  
     if (!allowedMethods.includes(req.method)) {  
     return res.status(405).json({ message: "Método não permitido" });  
     }  
     next();  
    });
  + Idealmente, cada rota deve definir explicitamente os métodos que suporta (e.g. router.get(...), router.post(...)) e o framework deve retornar 405 Method Not Allowed para outros métodos por padrão.

**Conclusão e Recomendações Gerais para o Backend:**

Os riscos identificados no backend incluem endpoints desprotegidos, potencial para upload de arquivos maliciosos, exposição de dados sensíveis, e possíveis vulnerabilidades de injeção devido à falta de sanitização e validação robustas. As recomendações principais são:

* **Adotar authMiddleware** em todas as rotas que manipulam dados sensíveis ou realizam operações críticas.
* **Implementar validação rigorosa de arquivos** no processo de upload (tipo, tamanho, e possivelmente scan de malware).
* **Minimizar a exposição de dados** nas respostas da API, utilizando DTOs e retornando apenas o essencial.
* **Utilizar express-validator ou similar** para sanitizar e validar todas as entradas do usuário.
* **Restringir os métodos HTTP** aceitos pelo servidor e por cada endpoint específico.
* **Revisão contínua de código** com foco em segurança e **testes de penetração periódicos** são práticas recomendadas para manter a segurança do backend a longo prazo. ### Testes com a Ferramenta ZAP (OWASP Zed Attack Proxy)

O OWASP Zed Attack Proxy (ZAP) é uma ferramenta de Teste Dinâmico de Segurança de Aplicação (DAST) amplamente utilizada e de código aberto. Ela funciona interceptando e inspecionando as mensagens trocadas entre o navegador e a aplicação web, além de realizar ataques automatizados para identificar vulnerabilidades comuns. Os testes com ZAP forneceram insights valiosos sobre a postura de segurança da aplicação em tempo de execução.

**Alertas Detectados pelo ZAP e Recomendações:**

Os principais alertas identificados pelo ZAP foram:

**1. Content Security Policy (CSP) Header Not Set:**

* **O que é?** A ausência do cabeçalho HTTP Content-Security-Policy permite que o navegador carregue recursos (scripts, estilos, imagens, etc.) de qualquer origem. Isso aumenta significativamente o risco de ataques de Cross-Site Scripting (XSS) e injeção de dados, onde scripts maliciosos de fontes não confiáveis podem ser executados no contexto da aplicação.
* **Como funciona o CSP?** O CSP define uma lista de permissões (whitelist) de fontes confiáveis das quais o navegador pode carregar recursos. Um exemplo básico que permite recursos apenas do próprio domínio (self) é: http Content-Security-Policy: default-src 'self';
* **Recomendação de Correção (para Express.js com helmet):** A biblioteca helmet para Express.js facilita a configuração de cabeçalhos de segurança, incluindo o CSP.
* Instalação:
* npm install helmet
* Uso no backend (ex: server.js):
* const express = require('express');  
  const helmet = require('helmet');  
  const app = express();  
    
  // Aplica CSP e outros headers seguros (configuração padrão do helmet)  
  // app.use(helmet());   
    
  // OU: define um CSP personalizado e mais restritivo  
  app.use(  
   helmet.contentSecurityPolicy({  
   directives: {  
   defaultSrc: ["'self'"], // Padrão: apenas do próprio domínio  
   scriptSrc: ["'self'", "'unsafe-inline'"], // Permite scripts do próprio domínio e inline (ajustar se possível para remover 'unsafe-inline')  
   styleSrc: ["'self'", "'unsafe-inline'", 'https://fonts.googleapis.com'], // Permite estilos do próprio domínio, inline e Google Fonts  
   fontSrc: ["'self'", 'https://fonts.gstatic.com'], // Permite fontes do próprio domínio e Google Fonts  
   imgSrc: ["'self'", 'data:'], // Permite imagens do próprio domínio e data URIs  
   connectSrc: ["'self'"], // Permite conexões (XHR, WebSockets) para o próprio domínio. Adicionar outras APIs se necessário.  
   // Adicione outras diretivas conforme necessário (frame-src, media-src, etc.)  
   },  
   })  
  );  
  // ... resto da configuração do servidor ...
* É crucial ajustar as diretivas do CSP (scriptSrc, styleSrc, connectSrc, etc.) para incluir todas as fontes externas legítimas que a aplicação utiliza (ex: CDNs, APIs de terceiros, Google Analytics). O uso de 'unsafe-inline' e 'unsafe-eval' deve ser evitado ao máximo, pois reduz a eficácia do CSP.

**2. Missing Anti-clickjacking Header (X-Frame-Options):**

* **O que é?** A ausência do cabeçalho X-Frame-Options torna a aplicação vulnerável a ataques de *clickjacking*. Nesse tipo de ataque, um site malicioso embute a aplicação vulnerável em um <iframe> invisível ou disfarçado, enganando o usuário para que clique em elementos da aplicação embutida sem seu conhecimento, realizando ações indesejadas.
* **Recomendação de Correção (com helmet):** O helmet também configura este cabeçalho. A opção mais segura é DENY, que impede qualquer site de embutir a aplicação. javascript // Se helmet() já é usado, esta proteção pode estar inclusa por padrão. // Para configuração explícita: app.use(helmet.frameguard({ action: 'deny' })); // Ou 'sameorigin' se precisar embutir em domínios próprios.

**3. X-Content-Type-Options Header Missing:**

* **O que é?** Sem o cabeçalho X-Content-Type-Options: nosniff, alguns navegadores podem tentar “adivinhar” (MIME sniffing) o tipo de conteúdo de um recurso, ignorando o Content-Type declarado pelo servidor. Isso pode levar a problemas de segurança se, por exemplo, um arquivo de texto que contém código JavaScript for interpretado e executado pelo navegador como se fosse um script.
* **Recomendação de Correção (com helmet):** javascript // Se helmet() já é usado, esta proteção pode estar inclusa por padrão. // Para configuração explícita: app.use(helmet.noSniff());

**4. Divulgação de Informações - Comentários Suspeitos:**

* **O que é?** O ZAP identificou comentários no código HTML/JavaScript que podem expor informações sensíveis ou fornecer dicas para atacantes (ex: <!-- TODO: corrigir autenticação -->, // API\_KEY = 'abc123').
* **Recomendação de Correção:** Revisar todo o código frontend (HTML, JSX, componentes Vue/Angular, etc.) e backend para remover ou limpar quaisquer comentários que contenham informações sensíveis, detalhes de implementação interna que não deveriam ser públicos, ou TODOs que indiquem fraquezas. Comentários em código de produção devem ser apenas para clarificar lógica complexa, não para armazenar dados ou notas de desenvolvimento.

**5. Information Disclosure - Sensitive Information in URL:**

* **O que é?** O ZAP detectou URLs que pareciam conter informações sensíveis, como tokens, emails, ou IDs, diretamente nos parâmetros da query string (ex: http://localhost:5173/user?token=abc123).
* **Riscos:** Informações sensíveis em URLs podem ser logadas por servidores web, proxies, armazenadas no histórico do navegador, e facilmente compartilhadas (ex: ao copiar e colar a URL), levando à exposição desses dados.
* **Recomendação de Correção:** Nunca transmitir dados sensíveis (como tokens de sessão, senhas, chaves de API, informações pessoalmente identificáveis) através de parâmetros de URL (query string). Utilizar métodos HTTP como POST para enviar dados sensíveis no corpo da requisição e/ou cabeçalhos HTTP para tokens de autorização (ex: Authorization: Bearer <token>).

**Pacote de Segurança com Helmet (Exemplo Consolidado):**

O documento original forneceu um exemplo de configuração consolidada do helmet para o backend Express, que aborda várias dessas questões:

const express = require('express');  
const helmet = require('helmet');  
const app = express();  
  
// Helmet padrão com ajustes finos  
app.use(helmet()); // Aplica um conjunto de padrões de segurança  
  
// Content Security Policy (CSP) - Personalizar conforme as necessidades da aplicação  
app.use(  
 helmet.contentSecurityPolicy({  
 directives: {  
 defaultSrc: ["'self'"],  
 scriptSrc: ["'self'", "'unsafe-inline'"], // Tentar remover 'unsafe-inline' se possível  
 styleSrc: ["'self'", "'unsafe-inline'", 'https://fonts.googleapis.com'], // Tentar remover 'unsafe-inline'  
 fontSrc: ["'self'", 'https://fonts.gstatic.com'],  
 imgSrc: ["'self'", 'data:'],  
 connectSrc: ["'self'"], // Adicionar domínios de APIs externas aqui  
 },  
 })  
);  
  
// Anti-clickjacking (X-Frame-Options)  
// app.use(helmet.frameguard({ action: 'deny' })); // Já coberto pelo helmet() padrão, mas pode ser explícito  
  
// Evita MIME sniffing (X-Content-Type-Options)  
// app.use(helmet.noSniff()); // Já coberto pelo helmet() padrão  
  
// Proteção contra XSS (básica, fornecida pelo browser)  
// app.use(helmet.xssFilter()); // Obsoleto e removido do helmet; navegadores modernos têm suas próprias proteções XSS Auditor.  
  
// Esconde tecnologia do servidor (X-Powered-By)  
app.use(helmet.hidePoweredBy()); // Remove o cabeçalho X-Powered-By  
  
// HTTP Strict Transport Security (HSTS) - Força HTTPS  
// Importante: Usar HSTS requer que o site seja servido sobre HTTPS.  
// Não habilitar se o site não tiver HTTPS configurado e testado.  
/\*  
app.use(  
 helmet.hsts({  
 maxAge: 60 \* 60 \* 24 \* 60, // 60 dias em segundos  
 includeSubDomains: true, // Aplica a subdomínios  
 preload: true, // Permite submissão para listas de preload de HSTS  
 })  
);  
\*/  
  
// Exemplo de rota  
app.get('/', (req, res) => {  
 res.send('Segurança ativada com sucesso!');  
});  
  
app.listen(3000, () => {  
 console.log('Servidor rodando na porta 3000 🔐');  
});

**Observações Importantes:**

* **Personalização do CSP:** A política de CSP é altamente dependente dos recursos que a aplicação utiliza. A configuração fornecida é um ponto de partida e deve ser rigorosamente testada e ajustada para não quebrar funcionalidades legítimas da aplicação.
* **HTTPS para HSTS:** O cabeçalho HSTS (HTTP Strict Transport Security) só deve ser habilitado se a aplicação for servida exclusivamente sobre HTTPS. Ele instrui o navegador a se comunicar com o servidor apenas via HTTPS por um período especificado, o que é uma medida de segurança importante contra ataques de downgrade.
* **Testes Contínuos:** Após aplicar as correções, é fundamental executar novamente o ZAP (e outras ferramentas de teste) para verificar se as vulnerabilidades foram mitigadas e se nenhuma nova foi introduzida.

Os testes com ZAP são uma parte valiosa de uma estratégia de segurança em camadas, ajudando a identificar vulnerabilidades que podem não ser aparentes através de análise estática ou testes manuais focados apenas na lógica de negócios. ## Resultados e Análise de Vulnerabilidades

A compilação dos resultados obtidos através dos diversos testes de segurança (verificação de dependências, escaneamento de código com ESLint, testes de backend e análise dinâmica com OWASP ZAP) revela um panorama que exige atenção em múltiplas frentes para o fortalecimento da segurança do software. Embora algumas verificações, como o escaneamento com ESLint, não tenham apontado falhas de segurança críticas diretas, os demais testes identificaram vulnerabilidades significativas que, se exploradas, poderiam comprometer a integridade, confidencialidade e disponibilidade da aplicação e dos dados por ela processados.

**Principais Vulnerabilidades Identificadas em Conjunto:**

1. **Vulnerabilidades em Dependências (Frontend e Backend):**
   * **Risco Principal:** A presença do pacote axios desatualizado, tanto no frontend quanto no backend, com uma vulnerabilidade de alta severidade (risco de SSRF e vazamento de credenciais), é um ponto crítico. Dependências como Babel, Esbuild e Vite no frontend também apresentaram vulnerabilidades moderadas.
   * **Impacto Potencial:** Exploração dessas falhas pode levar a ataques de Server-Side Request Forgery, onde o servidor é induzido a fazer requisições maliciosas, e ao vazamento de credenciais ou outras informações sensíveis. Problemas em ferramentas de build (Vite, Esbuild) podem introduzir fragilidades no código final ou no ambiente de desenvolvimento.
2. **Falhas de Configuração de Segurança no Backend:**
   * **Endpoints Desprotegidos:** Rotas cruciais de modificação (PUT /item/:id) e exclusão (DELETE /item/:id) não possuíam authMiddleware, permitindo acesso não autenticado a funcionalidades críticas.
   * **Cadastro de Usuários sem Validação Robusta:** As rotas de cadastro (POST /cadastroEstabelecimento, POST /cadastroCliente) careciam de validações adicionais (ex: confirmação de e-mail), tornando-as suscetíveis a abusos por bots.
   * **Exposição de Dados Sensíveis:** A rota GET / em userRoute.js não filtrava adequadamente os dados do usuário autenticado, podendo expor informações desnecessárias.
   * **Upload de Arquivos Inseguro:** A configuração do multer para upload de imagens não restringia suficientemente os tipos de arquivo ou o tamanho, abrindo margem para o envio de arquivos maliciosos.
   * **Potencial para Injeção (SQL/NoSQL) e XSS:** A falta de sanitização e validação sistemática das entradas do usuário em várias partes do backend cria um risco de vulnerabilidades de injeção e XSS armazenado.
   * **Proteção do .env:** Embora a recomendação de verificar o .gitignore tenha sido feita, a ausência dessa verificação inicial representa um risco se o arquivo .env estiver acidentalmente versionado.
3. **Ausência e Má Configuração de Cabeçalhos de Segurança (Detectados pelo ZAP):**
   * **Content Security Policy (CSP) Não Configurado:** A falta do cabeçalho CSP é uma das vulnerabilidades mais impactantes identificadas pelo ZAP, pois deixa a aplicação aberta a ataques de XSS e injeção de dados de fontes não confiáveis.
   * **Missing Anti-clickjacking Header (X-Frame-Options):** A ausência deste cabeçalho torna a aplicação vulnerável a clickjacking, onde usuários podem ser enganados a realizar ações indesejadas.
   * **X-Content-Type-Options Header Missing:** Permite que navegadores tentem adivinhar o tipo MIME dos recursos, o que pode levar à execução de scripts em contextos inesperados.
4. **Divulgação de Informações (Detectadas pelo ZAP):**
   * **Comentários Suspeitos no Código:** Comentários contendo TODOs relacionados à segurança ou, pior, informações como chaves de API (mesmo que exemplos) podem fornecer informações valiosas para atacantes.
   * **Informações Sensíveis em URLs:** A passagem de tokens ou outros dados sensíveis via query string em URLs é uma prática insegura que expõe esses dados.

**Análise Geral dos Riscos:**

As vulnerabilidades encontradas, quando analisadas em conjunto, indicam que a aplicação possui brechas que podem ser exploradas em diferentes vetores de ataque. A falta de authMiddleware em rotas críticas, combinada com a possibilidade de upload de arquivos maliciosos e a ausência de CSP, cria um ambiente de risco elevado. Um atacante poderia, por exemplo, explorar uma falha de upload para enviar um script malicioso e, devido à ausência de CSP, conseguir executá-lo no navegador de outros usuários, ou tentar escalar privilégios explorando endpoints desprotegidos.

A exposição de dados sensíveis, seja por meio de rotas de API mal configuradas ou por informações em URLs, aumenta o risco de vazamento de dados. As vulnerabilidades em dependências, especialmente a de SSRF no axios, podem permitir que atacantes usem o servidor da aplicação como um proxy para atacar outros sistemas internos ou externos.

Embora o ESLint não tenha detectado falhas de segurança graves, os problemas de qualidade de código que ele apontou (como props não validadas ou dependências ausentes em useEffect) podem, indiretamente, levar a bugs que tenham implicações de segurança ou dificultem a manutenção e a correção de vulnerabilidades reais.

É crucial que as recomendações fornecidas para cada tipo de teste sejam implementadas de forma prioritária. A correção dessas falhas não apenas reduzirá o risco de incidentes de segurança, mas também melhorará a robustez geral e a confiabilidade do software. A próxima seção detalhará como essas vulnerabilidades e a aplicação das correções se relacionam com a Lei Geral de Proteção de Dados (LGPD). ## Confronto dos Resultados com a Lei Geral de Proteção de Dados (LGPD)

A Lei Geral de Proteção de Dados (LGPD), Lei nº 13.709/2018, estabelece diretrizes claras para a coleta, tratamento, armazenamento e proteção de dados pessoais no Brasil. A segurança da informação é um dos pilares fundamentais da LGPD, exigindo que as organizações adotem medidas técnicas e administrativas aptas a proteger os dados pessoais de acessos não autorizados e de situações acidentais ou ilícitas de destruição, perda, alteração, comunicação ou qualquer forma de tratamento inadequado ou ilícito (Art. 46). Os testes de segurança realizados e as vulnerabilidades encontradas neste software têm implicações diretas no cumprimento desses requisitos.

### Importância dos Testes na Proteção de Dados Pessoais sob a Ótica da LGPD

Cada tipo de teste de segurança executado desempenha um papel vital na identificação de fragilidades que poderiam levar à violação da LGPD:

1. **Verificação de Dependências e a LGPD:**
   * **Risco à Proteção de Dados:** Dependências desatualizadas ou vulneráveis (como o axios com falha de SSRF) podem ser a porta de entrada para atacantes acessarem ou exfiltrarem dados pessoais processados pela aplicação. Um vazamento de credenciais, por exemplo, poderia dar acesso direto a bancos de dados contendo informações de clientes.
   * **Contribuição para a LGPD:** Manter as dependências atualizadas é uma medida técnica essencial para garantir a segurança dos dados desde a concepção (Privacy by Design) e por padrão (Privacy by Default), conforme preconizado pela LGPD. A falha em atualizar uma dependência com uma vulnerabilidade conhecida pode ser interpretada como negligência na adoção de medidas de segurança.
2. **Escaneamento de Código (ESLint) e a LGPD:**
   * **Risco à Proteção de Dados:** Embora o ESLint, no escopo dos testes, não tenha revelado falhas críticas diretas, a ausência de boas práticas de codificação (como validação de props ou tratamento de dependências em hooks) pode levar a bugs. Alguns desses bugs podem, indiretamente, criar brechas de segurança ou instabilidade que afetem a integridade ou disponibilidade dos dados pessoais.
   * **Contribuição para a LGPD:** A qualidade do código e a adesão a padrões seguros de desenvolvimento são fundamentais para construir sistemas robustos. A LGPD incentiva a adoção de boas práticas e governança em privacidade, o que inclui a qualidade e segurança do código que manipula dados pessoais. Ferramentas SAST, mesmo as mais básicas, auxiliam na manutenção dessa qualidade.
3. **Testes de Backend e a LGPD:**
   * **Risco à Proteção de Dados:** As vulnerabilidades encontradas no backend têm um impacto direto e severo na proteção de dados pessoais:
     + **Falhas de Autenticação/Autorização:** Endpoints desprotegidos (PUT/DELETE /item/:id) permitiriam que qualquer pessoa alterasse ou excluísse dados, potencialmente incluindo dados pessoais de clientes ou informações de estabelecimentos. Cadastros sem validação podem levar à criação de contas falsas para atividades ilícitas ou para inflar bases de dados com informações não verídicas.
     + **Exposição de Dados Sensíveis:** Rotas de API que retornam mais dados do que o necessário violam o princípio da minimização de dados (Art. 6º, III da LGPD) e aumentam o risco em caso de interceptação ou acesso indevido.
     + **Upload de Arquivos Inseguro:** Permitir o upload de arquivos maliciosos pode levar à execução de código no servidor, comprometendo toda a base de dados e os sistemas que processam dados pessoais.
     + **Risco de Injeção (SQL/NoSQL):** Vulnerabilidades de injeção podem permitir que atacantes leiam, modifiquem ou excluam todos os dados do banco, incluindo dados pessoais, violando a confidencialidade, integridade e disponibilidade.
     + **Proteção do .env:** O vazamento do arquivo .env exporia credenciais de acesso a bancos de dados e outros serviços, concedendo a um atacante acesso direto e irrestrito aos dados pessoais.
   * **Contribuição para a LGPD:** A correção dessas falhas é mandatória para o cumprimento da LGPD. Implementar autenticação e autorização robustas, garantir que apenas os dados necessários sejam expostos (minimização), proteger contra uploads maliciosos e prevenir injeções são medidas técnicas essenciais para assegurar a segurança dos dados pessoais (Art. 46) e atender aos princípios da lei, como segurança (Art. 6º, VII) e prevenção (Art. 6º, VIII).
4. **Testes com ZAP (Cabeçalhos de Segurança, XSS, etc.) e a LGPD:**
   * **Risco à Proteção de Dados:**
     + **Ausência de CSP:** Facilita ataques de XSS, onde scripts maliciosos podem ser injetados para roubar tokens de sessão, cookies contendo dados pessoais, ou redirecionar usuários para páginas falsas para coletar credenciais.
     + **Ausência de X-Frame-Options:** Permite clickjacking, que pode enganar usuários a realizar ações que comprometam seus dados pessoais (ex: autorizar o compartilhamento de dados, alterar configurações de privacidade sem consentimento).
     + **Ausência de X-Content-Type-Options:** Pode levar à execução de scripts maliciosos disfarçados de outros tipos de arquivo, potencialmente comprometendo dados no lado do cliente.
     + **Divulgação de Informações (Comentários, URLs):** A exposição de informações, mesmo que pareçam inócuas, pode auxiliar um atacante a entender a arquitetura do sistema e planejar ataques mais sofisticados, visando o acesso a dados pessoais.
   * **Contribuição para a LGPD:** A implementação correta de cabeçalhos de segurança como CSP, X-Frame-Options e X-Content-Type-Options são medidas técnicas preventivas que ajudam a proteger os dados pessoais contra uma variedade de ataques web. Essas medidas demonstram a devida diligência da organização em proteger os dados (accountability) e em prevenir incidentes de segurança.

### Prevenção de Incidentes de Segurança e Cumprimento da LGPD

A realização proativa e regular de testes de segurança, como os detalhados neste relatório, é uma das principais formas de prevenir incidentes de segurança que possam resultar em vazamento, perda ou acesso indevido a dados pessoais. A LGPD, em seu Art. 48, exige que o controlador comunique à Autoridade Nacional de Proteção de Dados (ANPD) e ao titular a ocorrência de incidente de segurança que possa acarretar risco ou dano relevante aos titulares.

Ao identificar e corrigir as vulnerabilidades apontadas, a organização não apenas reduz a probabilidade de ocorrência de tais incidentes, mas também demonstra conformidade com os deveres de segurança impostos pela LGPD. As recomendações fornecidas para cada vulnerabilidade visam:

* **Garantir a Confidencialidade:** Impedir que dados pessoais sejam acessados por indivíduos não autorizados (ex: corrigindo falhas de autenticação, protegendo o .env, implementando CSP para prevenir roubo de sessão).
* **Assegurar a Integridade:** Proteger os dados pessoais contra alterações não autorizadas ou indevidas (ex: prevenindo injeções, controlando acesso a rotas de modificação).
* **Manter a Disponibilidade:** Garantir que os dados pessoais e os sistemas que os processam estejam acessíveis aos usuários autorizados quando necessário (ex: prevenindo ataques de negação de serviço que poderiam ser facilitados por uploads maliciosos ou dependências vulneráveis).
* **Prevenir Vazamentos:** Adotar medidas técnicas que minimizem o risco de exposição acidental ou intencional de dados pessoais (ex: corrigindo exposição de dados em APIs, evitando informações sensíveis em URLs).

A adequação do software às recomendações deste relatório é, portanto, um passo crucial não apenas para a segurança técnica da aplicação, mas também para a conformidade legal com a LGPD, protegendo a organização de sanções administrativas, danos à reputação e, mais importante, protegendo os direitos e liberdades fundamentais dos titulares dos dados. ## Conclusão

Os testes de segurança abrangentes realizados neste software, cobrindo desde a análise de dependências e código estático até testes dinâmicos e de backend, revelaram um conjunto de vulnerabilidades que, se não tratadas, representam riscos significativos para a segurança da aplicação e, crucialmente, para a proteção dos dados pessoais por ela processados. As falhas identificadas, como dependências desatualizadas com vulnerabilidades críticas (notadamente o axios), endpoints de API desprotegidos, configurações inadequadas de upload de arquivos, ausência de cabeçalhos de segurança essenciais (como CSP e X-Frame-Options) e potencial para injeção de código, demonstram a necessidade de uma ação corretiva imediata e de um reforço nas práticas de desenvolvimento seguro.

A análise confrontada com a Lei Geral de Proteção de Dados (LGPD) ressalta a urgência dessas correções. A LGPD impõe às organizações o dever de adotar medidas técnicas e administrativas para proteger os dados pessoais, e a presença das vulnerabilidades listadas indica lacunas no cumprimento desses requisitos. A exploração de tais falhas poderia levar a incidentes de segurança com consequências graves, incluindo vazamento de dados, acessos não autorizados, perdas financeiras, danos à reputação da organização e sanções legais.

Ficou evidente que a segurança não é um estado a ser alcançado, mas um processo contínuo de avaliação, correção e melhoria. A verificação de dependências, o escaneamento de código, os testes de backend e as análises dinâmicas são ferramentas complementares que, juntas, oferecem uma visão mais completa da postura de segurança de uma aplicação. A implementação das recomendações detalhadas neste relatório não apenas mitigará os riscos imediatos, mas também contribuirá para a construção de uma cultura de segurança mais robusta dentro do ciclo de vida de desenvolvimento do software.

É imperativo que as equipes de desenvolvimento e gestão compreendam a seriedade das vulnerabilidades encontradas e priorizem sua correção. Além disso, a adoção de práticas de *Security by Design* e *Privacy by Design*, a realização de treinamentos contínuos em desenvolvimento seguro e a implementação de um ciclo de vida de desenvolvimento de software seguro (SSDLC) são passos fundamentais para prevenir a introdução de novas vulnerabilidades no futuro e garantir a conformidade contínua com a LGPD e outras regulamentações aplicáveis. A segurança do software e a proteção dos dados pessoais são responsabilidades compartilhadas que exigem comprometimento e vigilância constantes. ## Recomendações Práticas

Com base nas vulnerabilidades identificadas e na análise de conformidade com a LGPD, apresentamos as seguintes recomendações práticas, priorizadas para a correção das falhas e adequação do sistema:

**Ações Imediatas (Prioridade Alta):**

1. **Atualização de Dependências Críticas:**
   * **Ação:** Atualizar imediatamente o pacote axios para a versão mais recente e segura, tanto no frontend quanto no backend, para mitigar o risco de SSRF e vazamento de credenciais. Seguir com a atualização das demais dependências do frontend (Babel, Esbuild, Vite) que apresentaram vulnerabilidades moderadas.
   * **Comandos:** Utilizar npm update <pacote> --save ou npm audit fix --force conforme detalhado anteriormente. Validar as atualizações com npm audit.
2. **Correção de Falhas de Autenticação e Autorização no Backend:**
   * **Ação:** Implementar o authMiddleware nas rotas PUT /item/:id e DELETE /item/:id para garantir que apenas usuários autenticados e autorizados possam realizar estas operações.
   * **Ação:** Reforçar as rotas de cadastro (POST /cadastroEstabelecimento, POST /cadastroCliente) com validações adicionais, como confirmação de e-mail ou CAPTCHA, para prevenir abusos por bots.
3. **Implementação de Cabeçalhos de Segurança Essenciais (via Helmet ou similar):**
   * **Ação:** Configurar e habilitar o cabeçalho Content-Security-Policy (CSP) de forma restritiva, permitindo apenas as fontes de conteúdo estritamente necessárias. Evitar unsafe-inline e unsafe-eval sempre que possível.
   * **Ação:** Garantir que os cabeçalhos X-Frame-Options: DENY (ou SAMEORIGIN, se aplicável) e X-Content-Type-Options: nosniff estejam ativos para prevenir clickjacking e MIME sniffing, respectivamente.
   * **Ferramenta:** Utilizar a biblioteca helmet no Express.js, personalizando as diretivas do CSP conforme as necessidades da aplicação e testando exaustivamente.
4. **Proteção Contra Upload de Arquivos Inseguros:**
   * **Ação:** Modificar a configuração do multer (ou similar) para validar estritamente o tipo de arquivo (MIME type, extensão) e impor um limite de tamanho razoável para uploads. Permitir apenas os tipos de arquivo realmente necessários (ex: image/jpeg, image/png).
   * **Exemplo:** Implementar o fileFilter e limits no multerConfig.js conforme demonstrado anteriormente.
5. **Sanitização e Validação de Todas as Entradas do Usuário:**
   * **Ação:** Revisar todas as rotas de backend que recebem dados do usuário e implementar validação e sanitização robustas utilizando bibliotecas como express-validator. Isso é crucial para prevenir Injeção de SQL/NoSQL e XSS armazenado.
   * **Prática:** Adotar a validação no início de cada controller/handler de rota.
6. **Revisão e Correção da Exposição de Dados Sensíveis:**
   * **Ação:** Modificar as rotas de API (como GET / em userRoute.js) para retornar apenas os dados estritamente necessários (princípio da minimização). Utilizar DTOs para modelar as respostas.
   * **Ação:** Remover qualquer informação sensível de URLs (query strings). Utilizar o corpo da requisição (para POST/PUT) ou cabeçalhos HTTP para dados como tokens.
   * **Ação:** Verificar e garantir que o arquivo .env esteja listado no .gitignore e não tenha sido acidentalmente versionado.
7. **Remoção de Comentários Suspeitos:**
   * **Ação:** Realizar uma varredura completa no código-fonte (frontend e backend) para remover quaisquer comentários que exponham informações sensíveis, detalhes de implementação interna desnecessários ou TODOs relacionados à segurança.

**Ações de Médio Prazo (Melhoria Contínua):**

1. **Implementação de Ferramentas SAST Mais Robustas:**
   * **Ação:** Considerar a integração de ferramentas de Análise Estática de Segurança de Aplicações (SAST) mais especializadas, como SonarQube ou Snyk Code, no pipeline de CI/CD para uma detecção mais profunda e automatizada de vulnerabilidades no código.
2. **Testes de Penetração Periódicos:**
   * **Ação:** Planejar e executar testes de penetração (pentests) periódicos na aplicação, realizados por profissionais especializados. Esses testes simulam ataques reais e podem descobrir vulnerabilidades que ferramentas automatizadas não detectam.
3. **Treinamento em Desenvolvimento Seguro para a Equipe:**
   * **Ação:** Promover treinamentos regulares sobre práticas de desenvolvimento seguro para todos os desenvolvedores envolvidos no projeto. Tópicos devem incluir OWASP Top 10, desenvolvimento seguro para a stack tecnológica utilizada, e os princípios da LGPD.
4. **Adoção de um Ciclo de Vida de Desenvolvimento de Software Seguro (SSDLC):**
   * **Ação:** Integrar a segurança em todas as fases do ciclo de vida de desenvolvimento de software, desde os requisitos e design até os testes, implantação e manutenção (Security by Design).
5. **Monitoramento e Logging de Segurança:**
   * **Ação:** Implementar um sistema robusto de logging de segurança para registrar eventos relevantes (tentativas de login falhas, erros de aplicação, acessos suspeitos) e monitorar esses logs ativamente para detectar e responder a incidentes de segurança rapidamente.
6. **Configuração de HTTPS e HSTS:**
   * **Ação:** Garantir que toda a comunicação com a aplicação seja feita sobre HTTPS. Após a estabilização do HTTPS, configurar o cabeçalho HTTP Strict Transport Security (HSTS) para instruir os navegadores a se comunicarem apenas via HTTPS.

**Recomendações Específicas para Conformidade com a LGPD:**

1. **Revisão de Políticas de Privacidade e Termos de Uso:**
   * **Ação:** Garantir que as políticas de privacidade e os termos de uso da aplicação estejam alinhados com a LGPD, informando claramente aos usuários como seus dados são coletados, usados, armazenados e protegidos.
2. **Mapeamento de Dados Pessoais (Data Mapping):**
   * **Ação:** Realizar um mapeamento completo de todos os dados pessoais que a aplicação coleta e processa, entendendo seu ciclo de vida, quem tem acesso e como são protegidos. Isso é fundamental para a gestão da privacidade e para responder a requisições de titulares.
3. **Preparação para Atendimento aos Direitos dos Titulares:**
   * **Ação:** Desenvolver procedimentos e, se necessário, funcionalidades na aplicação para atender aos direitos dos titulares previstos na LGPD (confirmação de tratamento, acesso, correção, anonimização, portabilidade, eliminação, etc.).

A implementação dessas recomendações fortalecerá significativamente a segurança do software, protegerá os dados dos usuários e auxiliará na manutenção da conformidade com a LGPD. É crucial que essas ações sejam acompanhadas de um compromisso contínuo com a segurança da informação. ## Referências

1. Documentos Fornecidos:
   * “Testes de Segurança.docx” - Documento detalhando os testes de verificação de dependências, escaneamento de código, testes de backend e recomendações.
   * “Teste ZAP.docx” - Documento com os resultados dos testes realizados com a ferramenta OWASP ZAP, incluindo alertas detectados e recomendações.
   * “Testes de segurança de software.pdf” - Documento complementar sobre testes de segurança, incluindo verificação de dependências, escaneamento de código e testes de backend.
2. Referências Técnicas:
   * OWASP (Open Web Application Security Project). “OWASP Top Ten.” https://owasp.org/www-project-top-ten/
   * OWASP. “ZAP (Zed Attack Proxy).” https://www.zaproxy.org/
   * Node Security Project. “npm audit.” https://docs.npmjs.com/cli/v8/commands/npm-audit
3. Referências Legais:
   * Brasil. Lei nº 13.709, de 14 de agosto de 2018. Lei Geral de Proteção de Dados Pessoais (LGPD). Disponível em: http://www.planalto.gov.br/ccivil\_03/\_ato2015-2018/2018/lei/l13709.htm
   * Autoridade Nacional de Proteção de Dados (ANPD). “Guia Orientativo para Definições dos Agentes de Tratamento de Dados Pessoais e do Encarregado.” 2021.