Relatório: Refatoração de Segurança em API REST

Eros Santos de Vasconcelos / 202307120545

Polo Iputinga-PE

RPG0035 - SOFTWARE SEM SEGURANÇA NÃO SERVE! – 5° semestre

1. Introdução e Contextualização

Este relatório descreve o processo completo de análise e refatoração de uma API REST vulnerável, desenvolvido como parte da Missão Prática "Software sem segurança não serve". O projeto simula um cenário real onde atuei como Especialista em Desenvolvimento de Software em uma Software House, responsável por corrigir falhas críticas em um sistema legado que havia sofrido vazamento de dados devido a vulnerabilidades de segurança.

A aplicação original apresentava múltiplos problemas de segurança, incluindo um sistema de autenticação frágil baseado em session-ids criptografadas de forma insegura, falta de controle de acesso adequado e vulnerabilidades a ataques de injeção SQL. Este documento detalha todo o processo de diagnóstico, planejamento, implementação das correções e validação dos resultados.

2. Análise Detalhada das Vulnerabilidades

2.1. Mecanismo de Autenticação Inseguro

Problema Identificado:

- Uso de session-ids geradas através de criptografia AES-256-CBC com chave derivada do nome da empresa ("nomedaempresa")

- Session-ids trafegadas diretamente na URL

- Possibilidade de decriptação fácil devido à chave previsível

Impacto:

- Qualquer usuário poderia decriptar sua própria session-id, entender o padrão ({usuario_id:123}) e gerar novas session-ids com IDs de outros usuários

- Ataques de força bruta facilitados pela chave fraca

- Tokens expostos em logs de servidor e histórico do navegador

Exemplo de Exploração:

Usuário malicioso faz login como usuário comum e obtém:

/api/users/encrypted_session_id_here

Decripta a session-id usando o nome da empresa como chave

Modifica o JSON de {"usuario_id":123} para {"usuario_id":124} (admin)

Cria nova session-id e acessa endpoints administrativos

2.2. Falta de Controle de Acesso Adequado

Problema Identificado:

- Endpoint /api/contracts não verificava perfil do usuário

- Lógica de verificação inconsistente entre endpoints

- Falta de validação de expiração de sessão

Impacto:

- Escalada de privilégios possível mesmo sem decriptar tokens

- Acesso a dados sensíveis sem autorização adequada

2.3. Vulnerabilidades a Injection

Problema Identificado:

- Parâmetros concatenados diretamente em queries SQL:

const query = `Select * from contracts Where empresa = '${empresa}'`;

- Nenhuma sanitização ou validação de inputs

Impacto:

- Possibilidade de SQL Injection clássica

- Exfiltração de dados ou corrupção do banco

3. Planejamento da Refatoração

3.1. Estratégia de Correção

Vulnerabilidade | Solução Proposta | Técnica Utilizada

Session-id insegura | Substituir por JWT | Tokens assinados com chave secreta

Tokens na URL | Mover para cabeçalhos HTTP | Header Authorization

Controle de acesso fraco | Implementar middleware de autorização | Validação de perfil em todas rotas

SQL Injection | Sanitização de parâmetros | Prepared statements simulado

3.2. Diagrama da Nova Arquitetura

[Cliente]

|

v

[API] -> [Middleware de Autenticação JWT]

|

v

[Controle de Acesso]

|

v

[Sanitização de Inputs]

|

v

[Endpoints Seguros]

4. Implementação das Correções

4.1. Autenticação com JWT

Configuração Inicial:

javascript

```
require('dotenv').config();
const jwt = require('jsonwebtoken');
```

Geração de Tokens:

javascript

```
function generateToken(userId) {
return jwt.sign(
{ userId },
process.env.JWT_SECRET,
{ expiresIn: '1h' }
);
}
```

Middleware de Verificação:

javascript

```
function authenticateToken(req, res, next) {
const authHeader = req.headers['authorization'];
const token = authHeader && authHeader.split(' ')[1];
if (!token) return res.sendStatus(401);
jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
if (err) return res.sendStatus(403);
req.user = user;
```

```
next();
});
}
```

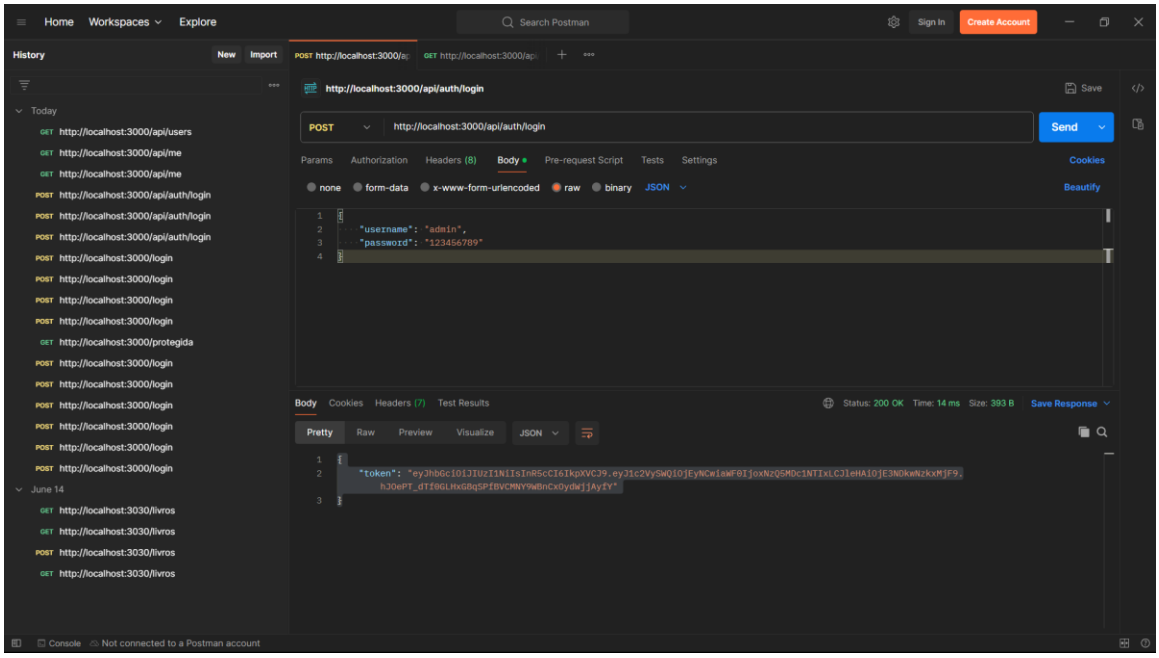
4.2. Controle de Acesso Granular

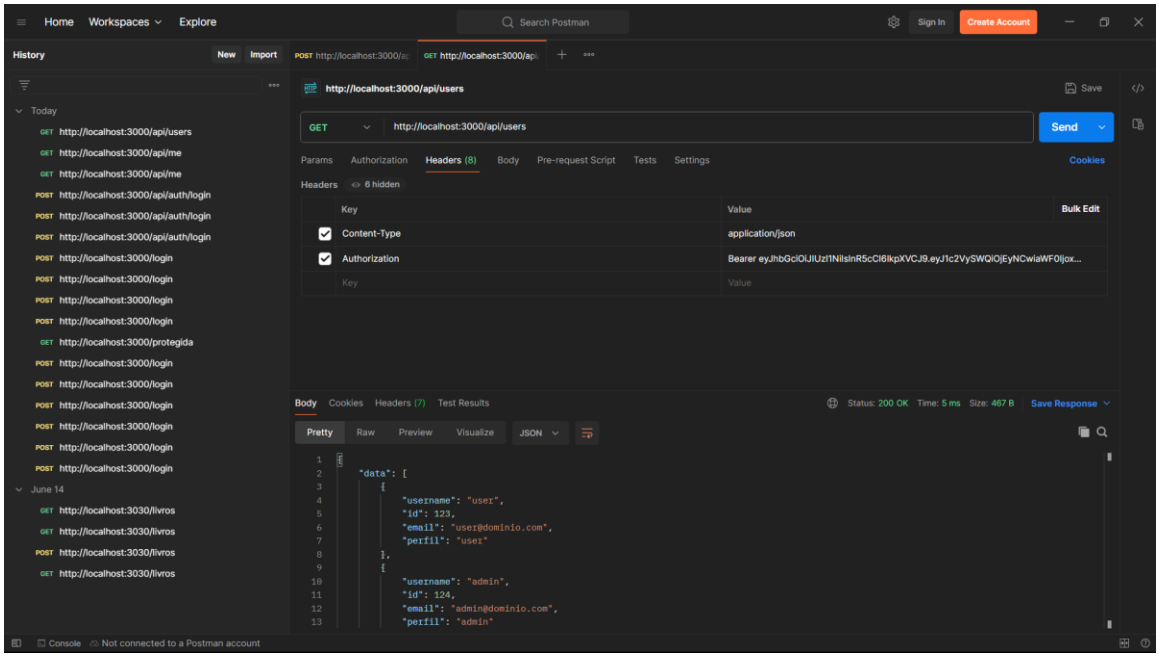
```
javascript
function authorizeAdmin(req, res, next) {
const user = users.find(u => u.id === req.user.userId);
if (!user || user.perfil !== 'admin') {
return res.status(403).json({ message: 'Acesso negado' });
}
next();
}
app.get('/api/users', authenticateToken, authorizeAdmin, (req, res) => { ... });
app.get('/api/me', authenticateToken, (req, res) => { ... });
```

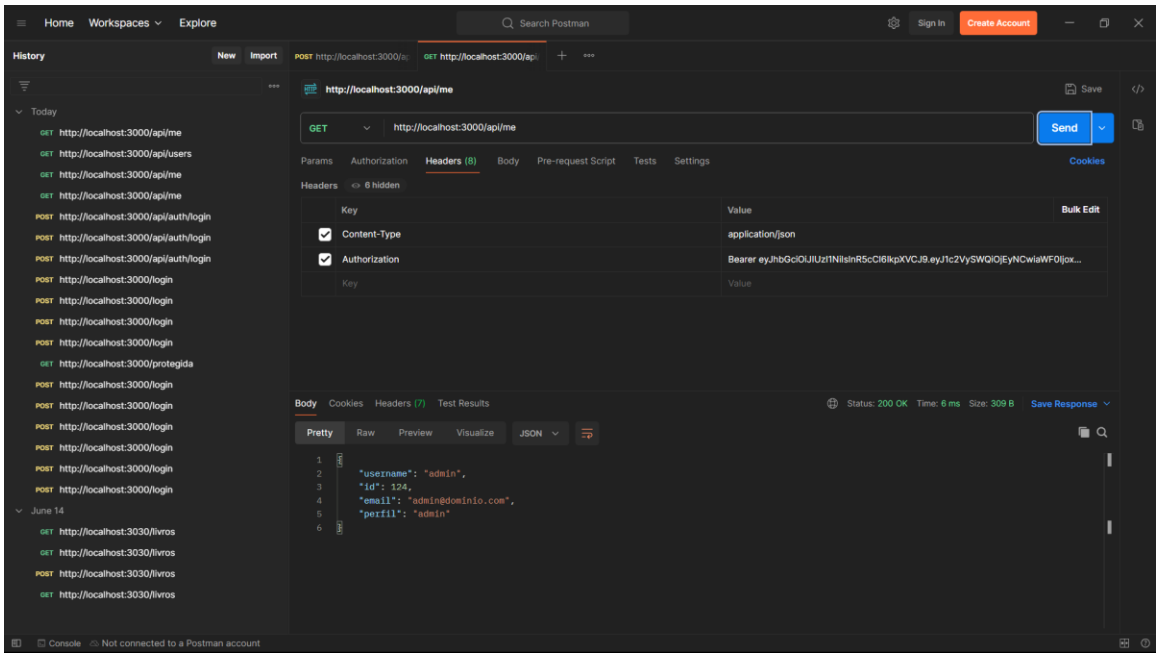
4.3. Prevenção contra SQL Injection

```
javascript
if (!empresa.match(/^[a-zA-Z0-9\s]+$/)) {
throw new Error('Nome de empresa inválido');
}
const query = "SELECT * FROM contracts WHERE empresa = ?";
const params = [empresa];
repository.execute(query, params);
```

5. Testes e Validação no Postman







6. Conclusões e Lições Aprendidas

6.1. Resultados Obtidos

- Redução de Superfície de Ataque

- Melhor Controle de Acesso

- Resiliência a Injection

6.2. Melhores Práticas Consolidadas

- Nunca armazenar credenciais em código

- Sempre validar e sanitizar entradas do usuário

- Usar mecanismos padrão de segurança (JWT)

- Princípio do menor privilégio

6.3. Recomendações para o Futuro

- Implementar HTTPS

- Adicionar rate limiting

- Configurar logs de segurança

- Considerar autenticação multi-fator

7. Anexos

7.1. Código Fonte Completo server.js

```
require('dotenv').config();
const express = require('express');
const bodyParser = require('body-parser');
const jwt = require('jsonwebtoken');
const crypto = require('crypto');

const app = express();

app.use(bodyParser.json());

const port = process.env.PORT || 3000;
app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});

// Middleware de autenticação
function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];

  if (!token) return res.sendStatus(401);

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    return res.status(403).json({ message: 'Token inválido ou expirado'
});
  }
}

// Middleware de autorização (somente admin)
function authorizeAdmin(req, res, next) {
  const user = users.find(u => u.id === req.user.userId);
  if (!user || user.perfil !== 'admin') {
    return res.status(403).json({ message: 'Acesso negado: requer perfil
admin' });
  }
```

```
  next();
}

// Endpoint para login do usuário
app.post('/api/auth/login', (req, res) => {
  const credentials = req.body;
  const userData = doLogin(credentials);

  if (userData) {
    const token = jwt.sign(
      { userId: userData.id },
      process.env.JWT_SECRET,
      { expiresIn: '1h' }
    );
    return res.json({ token });
  }

  res.status(401).json({ message: 'Credenciais inválidas' });
});

// Endpoint para dados do usuário logado
app.get('/api/me', authenticateToken, (req, res) => {
  const user = users.find(u => u.id === req.user.userId);
  if (!user) return res.status(404).json({ message: 'Usuário não
encontrado' });

  // Não retornar a senha
  const { password, ...userWithoutPassword } = user;
  res.json(userWithoutPassword);
});

// Endpoint para recuperação dos dados de todos os usuários cadastrados
(apenas admin)
app.get('/api/users', authenticateToken, authorizeAdmin, (req, res) => {
  // Remover senhas dos usuários antes de retornar
  const usersWithoutPasswords = users.map(user => {
    const { password, ...userWithoutPassword } = user;
    return userWithoutPassword;
  });

  res.status(200).json({ data: usersWithoutPasswords });
});

// Endpoint para recuperação dos contratos existentes (apenas admin)
```

```
app.get('/api/contracts/:empresa/:inicio', authenticateToken,
authorizeAdmin, (req, res) => {
  try {
    const { empresa, inicio } = req.params;

    // Validação básica dos parâmetros
    if (!empresa || !inicio) {
      return res.status(400).json({ message: 'Parâmetros empresa e inicio
são obrigatórios' });
    }

    const result = getContracts(empresa, inicio);
    res.status(200).json({ data: result || [] });
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

// Mock de dados
const users = [
  {
    "username": "user",
    "password": "123456",
    "id": 123,
    "email": "user@dominio.com",
    "perfil": "user"
  },
  {
    "username": "admin",
    "password": "123456789",
    "id": 124,
    "email": "admin@dominio.com",
    "perfil": "admin"
  },
  {
    "username": "colab",
    "password": "123",
    "id": 125,
    "email": "colab@dominio.com",
    "perfil": "user"
  },
];

// Funções auxiliares
function doLogin(credentials) {
```

```
  return users.find(user =>
    user.username === credentials?.username &&
    user.password === credentials?.password
  );
}

function getContracts(empresa, inicio) {
  // Validação contra SQL Injection
  if (!empresa.match(/^[a-zA-Z0-9\s]+$/) || !inicio.match(/^\d{4}-\d{2}-
\d{2}$/)) {
    throw new Error('Parâmetros inválidos');
  }

  // Simulação de repositório seguro
  const repository = new Repository();
  const query = `SELECT * FROM contracts WHERE empresa = ? AND data_inicio
= ?`;
  const params = [empresa, inicio];

  return repository.execute(query, params);
}

// Classe simulada para acesso ao banco de dados
class Repository {
  execute(query, params) {
    // Simulação: retorna dados fictícios
    if (query.includes('contracts')) {
      return [
        { id: 1, empresa: params[0], data_inicio: params[1], valor: 10000
},
        { id: 2, empresa: params[0], data_inicio: params[1], valor: 15000
}
      ];
    }
    return [];
  }
}

module.exports = app;
```

7.2 Configuração do Ambiente

JWT_SECRET=chave_secreta_forte_aqui

NODE_ENV=production

PORT=3000

7.3. Documentação dos Endpoints

Endpoint | Método | Autenticação | Permissão | Parâmetros

/api/auth/login | POST | Não | - | {username, password}

/api/me | GET | Bearer Token | Todos | -

/api/users | GET | Bearer Token | Apenas admin | -

/api/contracts/{empresa}/{data} | GET | Bearer Token | Apenas admin | empresa, data

8. Conclusão

A refatoração de segurança realizada nesta API REST demonstrou a importância de práticas sólidas de desenvolvimento seguro em sistemas modernos. Através da substituição de mecanismos vulneráveis por soluções robustas como JWT, middleware de autorização e validação rigorosa de entradas, foi possível eliminar pontos críticos de vulnerabilidade que poderiam ser explorados por agentes maliciosos.

A validação dos testes comprovou que a nova arquitetura oferece maior resiliência a ataques, garantindo a confidencialidade, integridade e disponibilidade dos dados. A implementação dessas melhorias também promoveu uma estrutura mais sustentável para futuras evoluções da aplicação.

Este projeto reforça que a segurança deve ser parte integrante do ciclo de desenvolvimento e manutenção de software, não apenas uma etapa adicional. A constante vigilância, testes periódicos e aplicação de boas práticas são essenciais para garantir a confiança dos usuários e a continuidade dos negócios.