Pentesting (parte 2)

Pâmela Baron, Dereck Conink, Maria A. Giuliari e Mariele Vieira

SQL/NoSQL Injection

1) Implementar este Ataque:

• No campo de nome de usuário, insira um comando SQL/NoSQL malicioso para obter todas as informações da tabela de usuários.

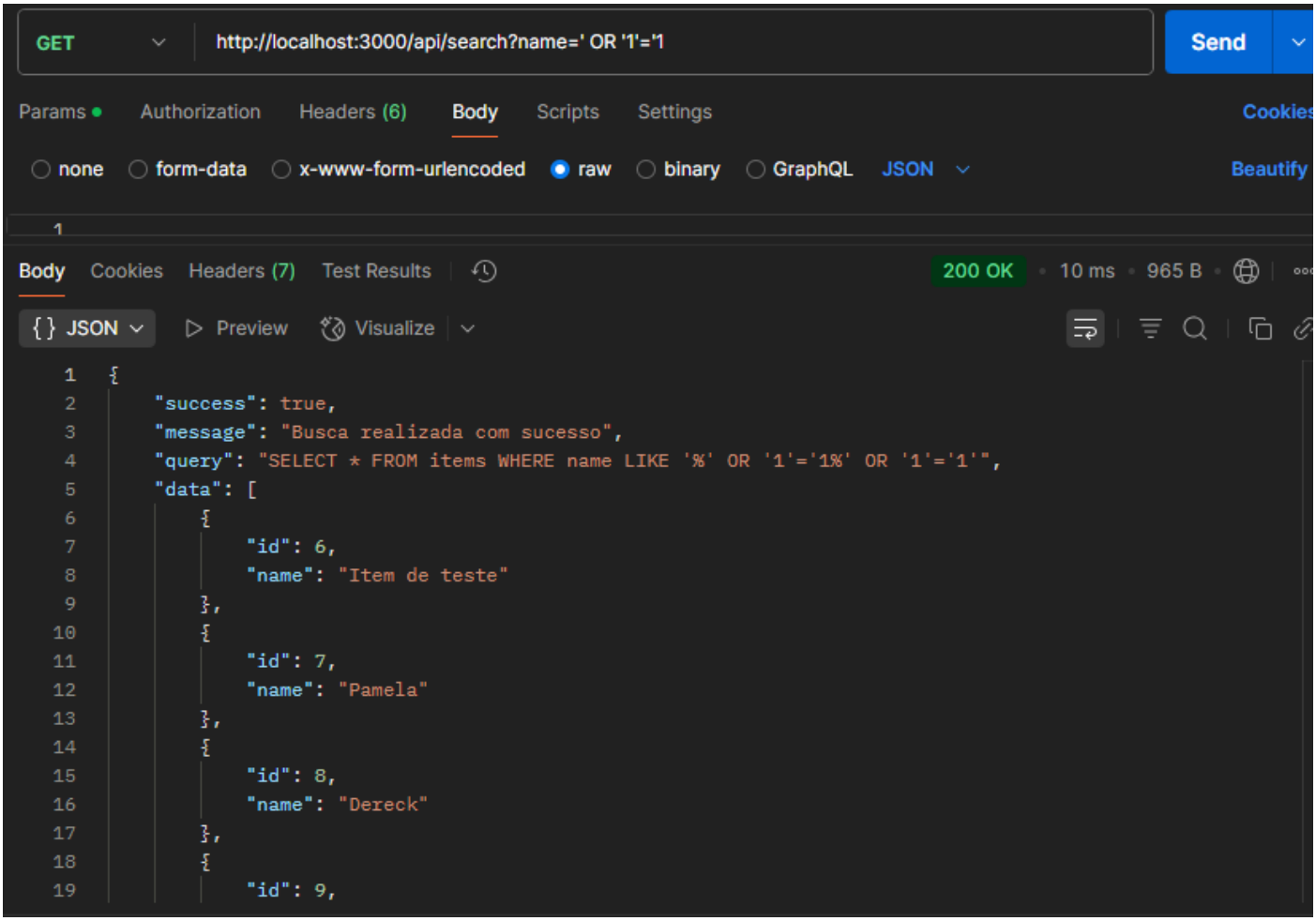
• Envie o formulário e observe o comportamento.

Criação do SQL injection:

```
// Rota para teste de SQL Injection
router.get('/search', (req, res) => {
  const searchName = req.query.name || '';
  const query = `SELECT * FROM items WHERE name LIKE '%${searchName}%' OR '1'='1'`;

  db.all(query, [], (err, rows) => {
    if (err) {
      return res.status(500).json({
        success: false,
        message: "Erro na busca",
        error: err.message
      });
    }
    res.json({
      success: true,
      message: "Busca realizada com sucesso",
      query: query,
      data: rows
    });
  });
});

module.exports = router;
```

Teste no POSTMAN:

Adicionar o URL: GET http://localhost:3000/api/search?name=' OR '1'='1



Resultado:
Ataque Realizado:
URL: http://localhost:3000/api/search?name=' OR '1'='1
Query SQL gerada: SELECT * FROM items WHERE name LIKE '%' OR '1'='1%' OR '1'='1'

Por que funcionou:
A injeção ' OR '1'='1 quebrou a lógica da query original.
'1'='1' é sempre verdadeiro (TRUE), isso fez com que a condição WHERE sempre retornasse TRUE para todas as linhas. Como resultado, TODOS os registros da tabela foram expostos.

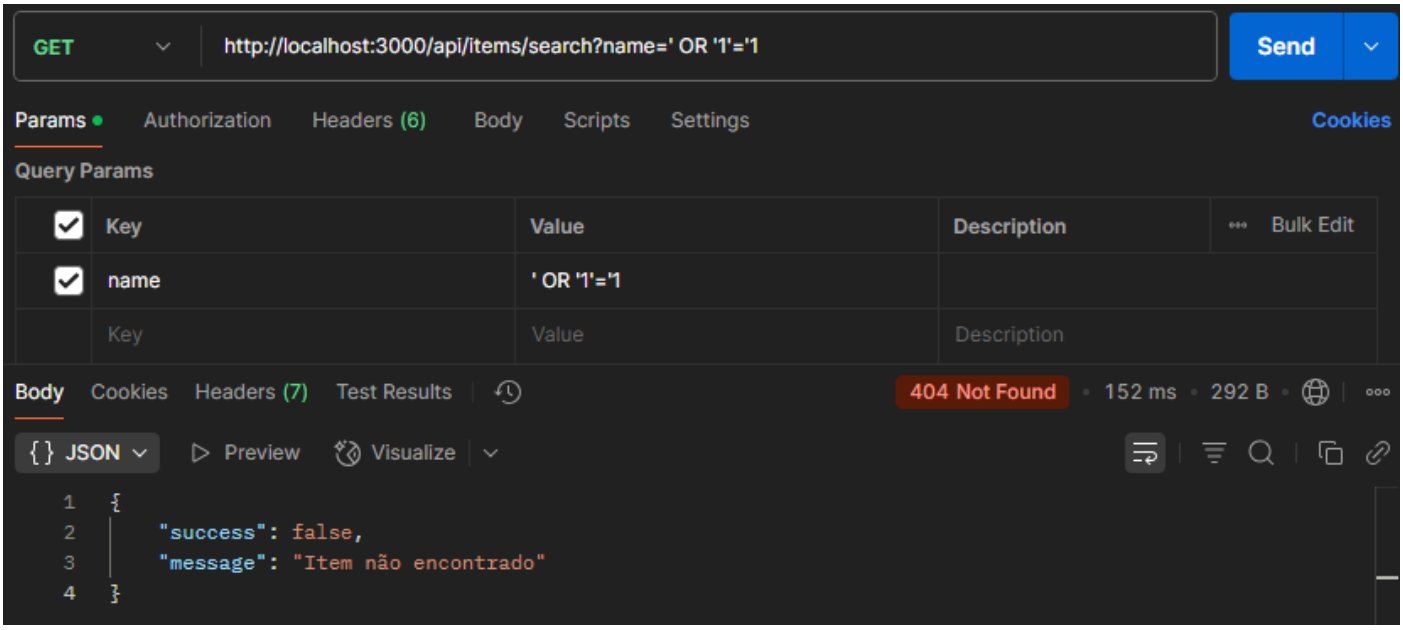
Impacto da Vulnerabilidade:

1. Vazamento de dados: Todos os registros foram expostos, mesmo aqueles que não deveriam ser visíveis
2. Nesse caso, expôs informações como:
   - IDs dos registros (6, 7, 8, 9)
   - Nomes ("Item de teste", "Pamela", "Dereck", etc)

Em um cenário real, poderia expor dados sensíveis como:
- Senhas
- Dados pessoais
- Informações financeiras
- Dados confidenciais da empresa

2) Ações para Correção e Prevenção:

• Use consultas parametrizadas ou prepared statements para evitar a inserção direta de dados do usuário em comandos SQL/NoSQL.
• Realize a validação de entrada para garantir que apenas dados válidos sejam inseridos nos comandos SQL/NoSQL.
• Limite as permissões de banco de dados para que o aplicativo da web tenha acesso apenas aos recursosnecessários.



O retorno "404 Not Found" com a mensagem "Item não encontrado" mostra que a implementação de segurança está funcionando corretamente.

Prevenção de SQL Injection

- A tentativa de injeção ' OR '1'='1 foi completamente bloqueada
- O sistema não permitiu a manipulação maliciosa da query SQL
- A validação removeu os caracteres especiais

```
if (!validateInput.name(searchTerm)) {
    return res.status(400).json({
        success: false,
        message: "Termo de busca inválido"
    });
}
```

```
  const stmt = db.prepare('SELECT * FROM items WHERE name LIKE ? LIMIT 50');

  try {
      const rows = stmt.all(`%${searchTerm}%`);
      res.json({
          success: true,
          message: "Busca realizada com sucesso",
          data: rows
      });
  } catch (error) {
      res.status(500).json({
          success: false,
          message: "Erro ao realizar busca"
      });
```

Resultado:

- Status: 404 Not Found
- Success: false
- Message: "Item não encontrado"

Indica que:

- A query foi executada de forma segura
- Nenhum dado foi exposto indevidamente
- O sistema manteve sua integridade

O sistema está seguro contra SQL Injection porque:

1. Rejeita entradas maliciosas
2. Usa prepared statements
3. Valida todas as entradas
4. Limita o acesso aos dados
5. Retorna mensagens seguras

Cross-Site Scripting (XSS)

1) Executar este Ataque na Versão 1 desenvolvida na parte 1:
• Injete um código JavaScript malicioso.
• Envie o formulário e observe o comportamento no navegador

Criamos o um formulário em http://localhost:3000/api/vulnerable-items

Ao inserir o código <img src="x" onerror="alert('Ataque XSS!')">
E enviar o formulário, uma mensagem aparece em tela:

Isso aconteceu porque:

1. O input não foi sanitizado
2. O HTML foi renderizado diretamente sem escape de caracteres especiais
3. Não há Content Security Policy (CSP) implementada
4. O navegador executou o JavaScript malicioso

Este é um exemplo claro de vulnerabilidade do tipo stored XSS, pois o código malicioso foi:

1. Armazenado no banco de dados
2. Será executado toda vez que alguém visitar a página
3. Poderia ser usado para roubar cookies, sessões ou executar outras ações maliciosas

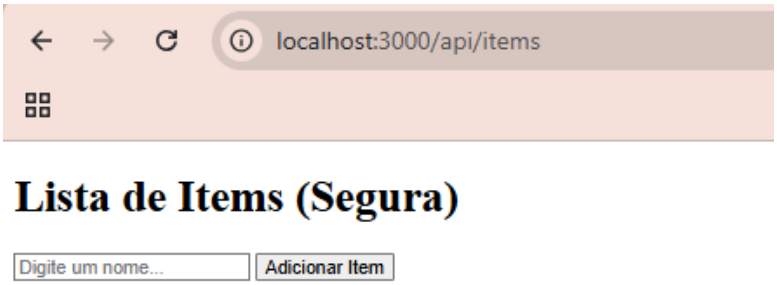
2) Ações para Correção e Prevenção:
• Valide e sanitize todos os dados de entrada do usuário.
• Use cabeçalhos de segurança, como Content Security Policy (CSP), para mitigar ataques XSS.
• Configure cookies de sessão com o atributo HTTPOnly para proteger contra roubo de cookies.

Criamos o arquivo middleware/ security.js;
Criamos dependências adicionais

```
"cookie-parser": "^1.4.6",
"express-session": "^1.17.3",
"xss": "^1.0.14"
```

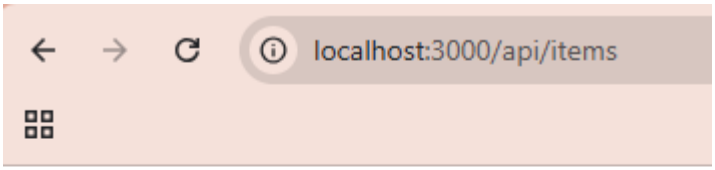
Resultado:

- Scripts são exibidos como texto
- HTML malicioso é escapado
- Cookies protegidos
- Proteção completa contra XSS



Lista de Items (Segura)

Digite um nome... Adicionar Item



Lista de Items (Segura)

Digite um nome... Adicionar Item

- <script>alert('XSS')</script>

Testes Realizados

- Injeção de scripts: <script>alert('XSS')</script>
- Resultado: Exibido como texto, sem execução
- Confirmação de sanitização funcionando

Benefícios da Implementação

1. Proteção contra diferentes tipos de XSS
2. Segurança de dados do usuário

3. Integridade da aplicação
4. Conformidade com práticas de segurança
5. Prevenção contra roubo de sessão

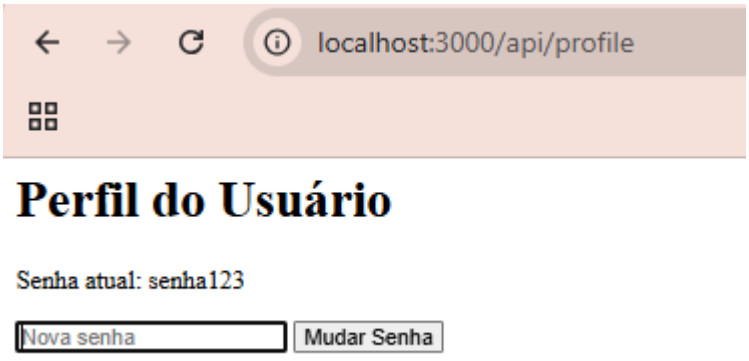
Cross-Site Request Forgery (CSRF)

1) Implementar este Ataque:
• Crie uma página HTML maliciosa em um servidor controlado pelo atacante.
• Nesta página, inclua um formulário com uma solicitação POST para realizar uma ação maliciosa em um aplicativo da web alvo. Por exemplo, enviar uma solicitação para alterar a senha do usuário.
• Engane um usuário autenticado para visitar essa página maliciosa.

Criamos um ataque CSRF que:
1. Explorou a confiança do servidor no navegador do usuário
2. Executou uma ação não autorizada (mudança de senha)
3. Aproveitou-se da autenticação existente do usuário

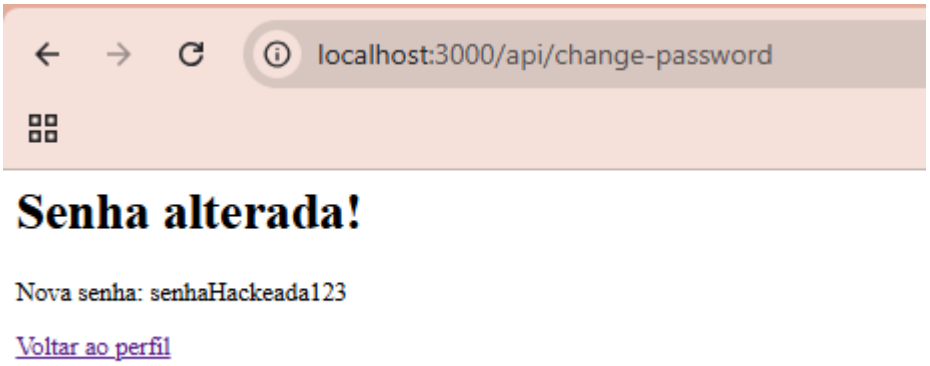
```
router.post('/change-password', (req, res) => {
    const newPassword = req.body.newPassword;
    userPassword = newPassword; // Atualiza a senha

    res.send(`
        <h1>Senha alterada!</h1>
        <p>Nova senha: ${newPassword}</p>
        <a href="/api/profile">Voltar ao perfil</a>
    `);
});
```

Página Vulnerável: http://localhost:3000/api/profile



Página Maliciosa: http://localhost:3000/pagina-maliciosa.html

Ao entrar em http://localhost:3000/pagina-maliciosa.html **a senha muda automaticamente**



Resultado:
Ausência de Proteções:

- Sem token CSRF
- Sem verificação de origem
- Sem validação adicional de autenticação

Comportamento do Navegador:

- Envia cookies automaticamente
- Permite requisições cross-origin
- Executa JavaScript automaticamente

Vulnerabilidades da Aplicação:

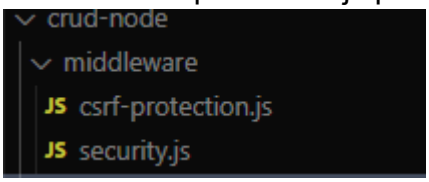
- Aceita POST de qualquer origem
- Não válida a fonte da requisição
- Executa ações sensíveis sem confirmação

Impacto do Ataque

1. Alteração não autorizada de senha
2. Comprometimento da conta do usuário
3. Possível perda de acesso pelo usuário legítimo
4. Violação de segurança sem detecção imediata

2) Ações para Correção e Prevenção:
• Use tokens CSRF (também conhecidos como tokens anti-CSRF) para proteger solicitações sensíveis contraataques CSRF.
• Certifique-se de que todas as operações sensíveis, como alterar a senha ou realizar transações financeiras, exijam um token CSRF válido.

Criamos csrf-protection.js para proteções de segurança implementadas



```
const csrfProtection = csrf({
    cookie: {
        httpOnly: true,
        secure: process.env.NODE_ENV === 'production',
        sameSite: 'lax',
        maxAge: 3600000 // 1 hora
    }
});
```

Gera tokens únicos para cada sessão
Valida tokens em todas as requisições POST
Cookies configurados como httpOnly para prevenir acesso via JavaScript

```
  res.setHeader('X-Content-Type-Options', 'nosniff');
  res.setHeader('X-Frame-Options', 'DENY');
  res.setHeader('X-XSS-Protection', '1; mode=block');
  res.setHeader('Referrer-Policy', 'same-origin');

  next();
```

Previne sniffing de MIME types
Impede que a página seja carregada em iframes
Ativa proteção contra XSS do navegador
Controla como o referer é enviado

Em resumo esta implementação forneceu várias camadas de segurança como:
Proteção contra ataques CSRF
Proteção contra XSS
Validação de entrada
Cookies seguros
Headers de segurança
Verificação de origem
Gerenciamento seguro de sessão

Resultado:
Página Inicial Segura:

- A página está protegida contra-ataques CSRF
- Usa cookies seguros para sessão
- Tem um link seguro para o perfil

Página de Perfil:

- Mostra que está protegida contra CSRF
- Exibe a senha atual (apenas para demonstração)
- Tem um formulário para mudar a senha
- Cada sessão gera um token único de proteção

Não é mais possível utilizar **http://localhost:3000/pagina-maliciosa.html**