## Cenário 1: IDOR - Insecure Direct Object Reference

**Explique o Cenário:** "O primeiro ataque é o IDOR. Acontece quando conseguimos acessar dados de outros usuários simplesmente adivinhando ou alterando o ID de um recurso. Aqui, estou 'logado' como o usuário comum, João."

**Execute o Ataque:** "O João pode ver seu próprio perfil, o de ID 1. Mas o que acontece se eu, como João, tentar ver o perfil de ID 2, que pertence à Maria, a administradora?" (Mude o ID no campo para 2 e clique em "Buscar Dados").

**Mostre o Resultado:** "Pronto. Sem nenhuma barreira, eu tenho acesso a todos os dados sensíveis da Maria, incluindo seu salário e CPF. Um vazamento de dados grave."

**Explique o Código Vulnerável:** (Mostre a parte do código vulnerável na tela). "Por que isso aconteceu? A falha está no backend. A rota da API simplesmente recebe um ID, encontra o usuário e devolve todos os dados. Não há nenhuma verificação para saber quem está fazendo o pedido."

**Mostre a Correção:** (Vá para a página /secure). "Agora, na versão segura. Vamos repetir o teste. Estou logado como João (user) e tento acessar o perfil de ID 2." (Faça o teste). "Como podem ver, a API agora retorna 'Acesso Proibido'. A defesa funcionou."

**Teste como Admin:** "Mas, se eu simular o login como Maria (admin), ela consegue ver o perfil de qualquer usuário, como esperado." (Troque o login para Maria e busque o ID 1 ou 3).

**Explique o Código Seguro:** (Mostre o código seguro na tela). "A correção tem duas partes essenciais no backend: primeiro, um middleware de autenticação que identifica quem é o usuário logado. Segundo, uma lógica de autorização que verifica: 'O ID que você está pedindo é o seu próprio? Ou você tem o cargo de 'admin'?'. Se a resposta para ambas for não, o acesso é negado."

Claro. Este código apresenta a **versão segura** da rota anterior, implementando as defesas necessárias para corrigir a vulnerabilidade de IDOR.

Vamos analisar as duas partes principais: o **Middleware** e a **Rota Segura**.

---

### 1. O Middleware: `authenticateUser`

```
JavaScript

const authenticateUser = (req, res, next) => {
  // ...lógica para verificar token e encontrar o usuário...
  req.user = user; // Anexa o usuário logado à requisição
  next();
};
```

Esta função é um **middleware** do Express. Pense nele como um "segurança" ou "porteiro" que fica entre o pedido do usuário e a rota final.

- **O que ele faz?** A principal função dele é a **autenticação**: descobrir *quem* está fazendo a requisição.
- **Como funciona?**
    - Em uma aplicação real (indicado pelo comentário `...lógica para verificar token...`), ele leria um token de autenticação (como um JWT) do cabeçalho da requisição para identificar o usuário.
    - `req.user = user;`: Esta é a parte crucial. Após identificar o usuário, o middleware anexa as informações dele (`user`) ao objeto da requisição (`req`). Isso torna os dados do usuário logado (como seu ID e seu cargo/role) disponíveis para todas as rotas que vierem a seguir.
    - `next();`: Esta função é chamada para "liberar a passagem", ou seja, para passar o controle para a próxima função na fila, que no nosso caso é a própria rota `app.get`. Se o `next()` não for chamado, a requisição fica "presa" no middleware.

---

## 2. A Rota Segura

```JavaScript
app.get('/api/secure/users/:id', authenticateUser, (req, res) => {
    // ...
});
```

- `app.get('/api/secure/users/:id', authenticateUser, ...)`: Note que `authenticateUser` é passado como um argumento antes da lógica principal da rota. Isso significa que, antes de qualquer código dentro da rota ser executado, o "porteiro" `authenticateUser` precisa fazer seu trabalho primeiro.

Agora, a lógica interna:

```JavaScript
const requestedUserId = parseInt(req.params.id);
const loggedInUser = req.user;
```

- Aqui, obtemos dois IDs importantes:
    - `requestedUserId`: O ID do recurso que o usuário *deseja* ver (vindo da URL).
    - `loggedInUser`: As informações de quem *está pedindo* (disponibilizadas pelo middleware `authenticateUser`).

```JavaScript
```

```
// ✅ DEFESA: Verifica se o usuário logado é o
// dono do recurso ou se é um admin.
if (loggedInUser.id !== requestedUserId && loggedInUser.role !== 'admin') {
  return res.status(403).json({ error: 'Acesso Proibido' });
}
```

Esta é a **defesa fundamental contra o IDOR** e a lógica de **autorização**. O código está fazendo duas perguntas essenciais:

1. O ID do usuário que está logado (`loggedInUser.id`) é o mesmo ID do recurso que ele está tentando acessar (`requestedUserId`)?
    - Se sim, ele é o "dono" do recurso e o acesso deve ser permitido.
2. Se não for o dono, o cargo (`role`) do usuário logado é 'admin'?
    - Se sim, ele tem um passe livre para ver os dados de outros usuários e o acesso também deve ser permitido.

- **`&&` (E):** O acesso só será bloqueado se **ambas** as condições forem falsas. Ou seja, o usuário **não é o dono** E **não é um admin**.
- `return res.status(403).json({ error: 'Acesso Proibido' });`: Se o acesso for bloqueado, o servidor retorna o status **403 Forbidden**. Este é o código correto para indicar que o servidor entendeu o pedido, mas se recusa a autorizá-lo.

Em resumo, este código corrige a falha ao implementar uma camada de autenticação (para saber quem pede) e uma lógica de autorização (para decidir se quem pede tem permissão), garantindo que os usuários só possam acessar os recursos que lhes pertencem, com uma exceção para administradores.

**Cenário 2: Elevação de Privilégio**

**Explique o Cenário:** (Volte para a página /vulnerable). "O segundo ataque é ainda mais perigoso: a Elevação de Privilégio. É quando um usuário comum consegue se transformar em um administrador."

**Execute o Ataque:** "Nesta tela de edição de perfil, notem que o campo 'Role' é um menu onde posso escolher meu cargo. O sistema confia no que eu envio." (Mude o 'Role' de 'user' para 'admin' e clique em "Atualizar").

**Mostre o Resultado:** "E pronto. O sistema me responde que meu novo status é 'admin', e o painel administrativo foi desbloqueado. Eu agora tenho controle total sobre o sistema."

**Explique o Código Vulnerável:** (Mostre o código vulnerável na tela). "A falha aqui é a confiança cega no frontend. O backend recebe os dados do formulário, incluindo o campo 'role', e salva tudo no banco de dados sem questionar."

**Mostre a Correção:** (Vá para a página /secure). "Na versão segura, a primeira coisa que vemos é que o campo 'Role' não é mais editável. Ele apenas exibe minha permissão atual."

**Teste a Atualização:** "Se eu alterar meu nome e clicar em atualizar, a operação funciona. Mas meu cargo permanece 'user'."

**Explique o Código Seguro:** (Mostre o código seguro na tela). "A defesa no backend é simples e poderosa: a rota que atualiza o perfil do usuário ignora deliberadamente o campo 'role'. Ela só atualiza os campos permitidos, como nome e email. A alteração de cargos só pode ser feita por uma rota separada, acessível apenas por administradores."

Este código define uma rota segura para atualizar o perfil de um usuário, prevenindo a vulnerabilidade de **Elevação de Privilégio**.

## Detalhamento do Código

1. **`app.put('/api/secure/users/profile', authenticateUser, (req, res) => { ... });`**
    - **`app.put`**: Define que a rota responde a requisições HTTP do tipo **PUT**, usadas para atualizar um recurso existente.
    - **`authenticateUser`**: É um middleware que executa antes da lógica da rota. Ele identifica qual usuário está logado e anexa suas informações ao objeto `req` (ex: `req.user`).
2. **`const loggedInUser = req.user;`**
    - Acessa as informações do usuário que está fazendo a requisição, que foram previamente carregadas pelo middleware `authenticateUser`.
3. **`const userToUpdate = users.find(u => u.id === loggedInUser.id);`**
    - Busca no array `users` o objeto completo do usuário que está logado. A atualização será feita diretamente neste objeto, garantindo que usuários só possam modificar seu próprio perfil.
4. **`const { name, email } = req.body;`**
    - Esta é a **defesa principal**. O código extrai **apenas** os campos `name` e `email` do corpo da requisição (`req.body`).
    - Qualquer outro campo enviado pelo cliente, como `role`, é **deliberadamente ignorado** e não é armazenado em nenhuma variável.
5. **`if (userToUpdate) { ... }`**
    - Verifica se o usuário a ser atualizado foi encontrado.
    - `userToUpdate.name = name || userToUpdate.name;`: Atualiza o nome do usuário. A expressão `|| userToUpdate.name` garante que, se um novo nome não for enviado, o valor antigo seja mantido.
    - O campo `role` do objeto `userToUpdate` **não é alterado**, preservando o valor que já existe no servidor.
6. **`res.json(userToUpdate);`**
    - Envia de volta o objeto do usuário com os dados atualizados, confirmando que o `role` permaneceu inalterado.

A segurança deste código reside no fato de que o backend não confia nos dados enviados pelo cliente. Ele opera com base em uma "lista de permissão" (allowlist) de campos, aceitando apenas modificações em `name` e `email` e descartando qualquer tentativa de alterar campos críticos como `role`.