02

Segurança: SQL Injection

Agora é hora de nos preocuparmos em tratar as informações que vem do usuário final. Por exemplo, usuários maliciosos podem ter bagunçar nossas informações. Geralmente a maneira mais comum disso acontecer é quando o usuário passa um dado malicioso, e nosso software acaba mandando isso para o banco de dados.

Hoje nossa listagem de vagas, representada pelo método `index`, nos devolve todas as vagas do sistema. Mas e se quisermos pegar vagas com ids maior que 3, por exemplo? Passaríamos algo assim na URL: `localhost:/8080/jobs?id_minimo=3`.

Em nosso controller então, pegaríamos esse número 3, e passaríamos essa informação para o banco de dados:

```
def index
  @id_minimo = params[:id_minimo]
  @jobs = Job.where("id > #{@id_minimo}").most_recent.includes(:company).all
  # ... codigo continua aqui
end
```

Se abrirmos o browser, ele trará somente os jobs com ids maior ou igual a 3, por exemplo, ou igual ao valor que passarmos pela querystring.

Precisamos fazer também com que isso fosse opcional. Ou seja, se não passarmos nada, a aplicação deveria funcionar do mesmo jeito. Para isso, fazemos um if:

```
def index
  @id_minimo = params[:id_minimo]
  if @id_minimo
    @jobs = Job.where("id > #{@id_minimo}").most_recent.includes(:company).all
  else
    @jobs = Job.most_recent.includes(:company).all
  end
  # ... codigo continua aqui
end
```

Pronto. Agora de ambas as maneiras, a aplicação funciona. Mas a pergunta é: o que acontece se, ao invés de passarmos um número, passarmos a letra "a"? Nossa consulta SQL fica inválida. Veja no browser: a SQL até aparece.

Sabendo disso, um usuário malicioso poderia tentar usar isso para fazer algum ataque ao nosso banco de dados. Por exemplo, se ele passar na URL: `/jobs?id_minimo=1 or company_id < 5`, o sistema pegará esse trecho de código e mandará direto para o banco de dados! O "or company_id < 5" será executado!

Isso não é legal - pelo contrário, é uma falha grave de segurança! Chamamos isso de **SQL Injection**. Muitos usuários mudam a SQL para conseguir o que querem. Um exemplo clássico é formulários de login. O usuário, se passar uma string bem pensada, consegue logar no sistema, mesmo sem ter um usuário válido.

Isso ocorreu justamente porque concatenamos strings na consulta SQL. Isso é perigoso demais! Precisamos tratar melhor os dados que chegam do usuário.

O Rails tem uma maneira bem elegante de resolver isso. Ao invés de fazermos a interpolação de strings, passaremos o dado que veio do usuário para que ele faça o tratamento e depois a concatenação. Para isso, basta passar outro parâmetro para o método `where`. Repare também no "?" que colocamos. O Rails o substituirá pelo valor passado no segundo parâmetro, mas de maneira inteligente, impossibilitando a injeção da SQL malvada:

```
def index
  @id_minimo = params[:id_minimo]
  if @id_minimo
    @jobs = Job.where("id > ?", @id_minimo).most_recent.includes(:company).all
  else
    @jobs = Job.most_recent.includes(:company).all
  end
  # ... codigo continua aqui
end
```

Agora se tentarmos passar aquela URL com informações maliciosas, o Rails não vai fazer a interpolação de uma vez, ele primeiro vai tratar o dado. Isso dá para ser visto no próprio log do Rails, onde a query não tem a sujeira.

Com isso funcionando, podemos agora melhorar um pouco nosso código. Repare que o trecho `most_recent.includes(:company).all` aparece nas duas consultas. Podemos melhorar esse código, reutilizando um pouco:

```
def index
  @id_minimo = params[:id_minimo]
  if @id_minimo
    list_parcial = Job.where("id > ", @id_minimo)
  else
    list_parcial = Job
  end

    @jobs = list_parcial.most_recent.includes(:company).all

  # ... codigo continua aqui
end
```

Evite SQL Injection. Nunca faça concatenação de Strings SQL, sempre faça uso do "?" e deixe o ActiveRecord montar a SQL para você.