

Relatório do Projeto 1

Vulnerabilidades

Diana Siso 98607

Miguel Ferreira 98599

Raquel Ferreira 98323

Sophie Pousinho 97814

Índice

Introdução	2
CWE-79: Improper Neutralization of Input During Web Page Generation('Cross-site Scripting')	3
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	5
CWE-862: Missing Authorization	9
CWE-20: Improper Input Validation	11
CWE-549: Missing Password Field Masking	12

Introdução

Para a realização deste projeto, decidimos desenvolver uma wiki sobre personagens do mundo de Harry Potter. Nela é possível obter informação sobre as personagens e também deixar comentários nas páginas das personagens. Para comentar é necessário realizar login.

Existem 2 tipos de utilizadores: os normais e os administradores. Os normais podem eliminar os seus próprios comentários e os administradores podem eliminar os comentários de qualquer utilizador.

Implementámos as seguintes vulnerabilidades:

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- CWE-862: Missing Authorization
- CWE-20: Improper Input Validation
- CWE-549: Missing Password Field Masking

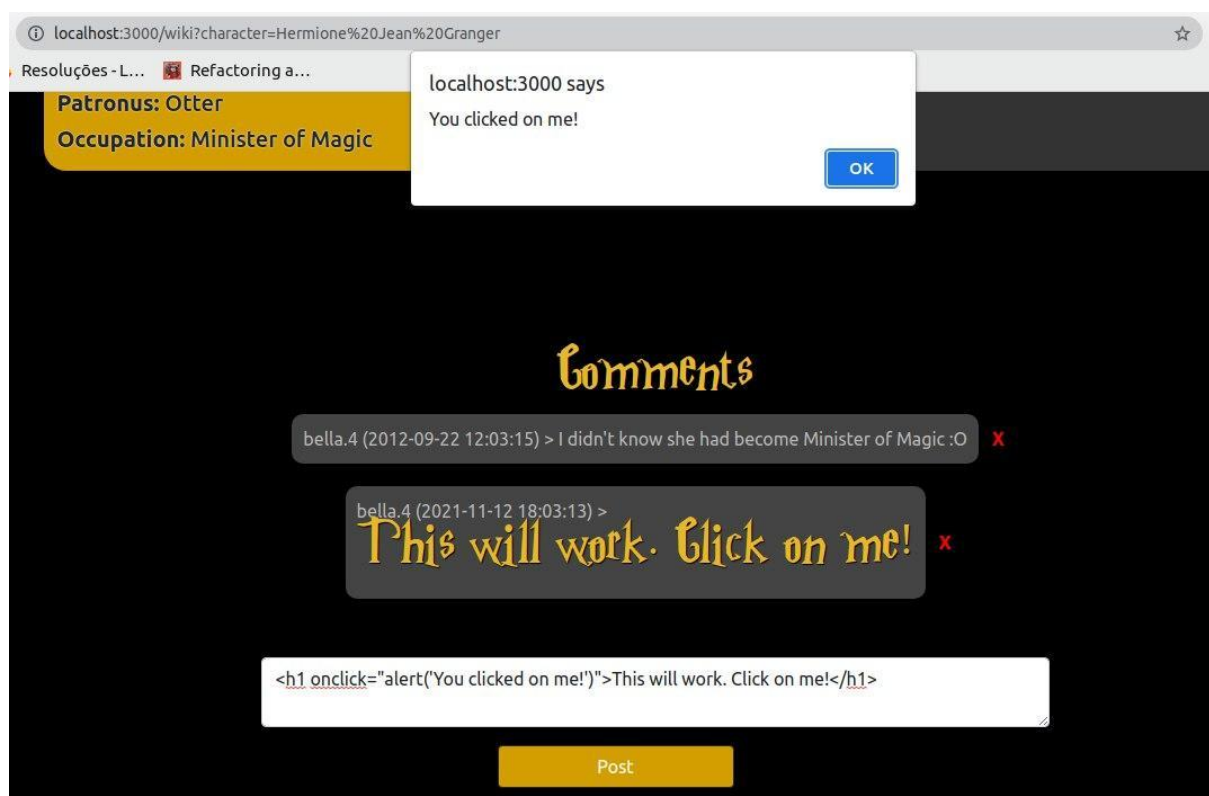
O projeto foi realizado em React para o front-end, php para o back-end, MySQL para a base de dados e xampp para o servidor.

CWE-79: Improper Neutralization of Input During Web Page Generation('Cross-site Scripting')

Nesta vulnerabilidade, de forma geral, algum código malicioso, normalmente em JavaScript, é inserido numa página web para que depois seja executado no lado do utilizador final. Isto pode acontecer caso seja exibido conteúdo dos utilizadores ou caso a aplicação permita a entrada de dados de fontes não confiáveis sem “escaping” ou validação.

Segundo o site do CWE, existem 3 tipos principais de XSS: reflected XSS, stored XSS e DOM-Based XSS.

Implementámos o stored XSS (inserção de código malicioso no servidor por parte do atacante) na página com as informações de cada personagem, na parte dos comentários. Desta forma, ao postar um comentário com código que permita correr um script, a versão vulnerável não faz a sua validação e qualquer utilizador que aceda àquela página pode correr o script. Podemos ver pelo exemplo que ao clicar no título, aparecerá o *alert* do JavaScript.



De forma a corrigirmos isto, usámos a função *htmlspecialchars*, que se encontra numa na função *verifyInput*, do php que transforma os caracteres do HTML com significado especial em entidades HTML, evitando que o script corra e apenas apareça o comentário com a transformação dos caracteres especiais, como podemos ver na última imagem.

```
function verifyInput($input) {  
    $input = trim($input);  
    $input = stripslashes($input);  
    return htmlspecialchars($input);  
}
```

(Função *verifyInput*, onde espaços e caracteres desnecessários são removidos)



Comments

bella.4 (2021-11-12 18:00:47) > <h1 onclick="alert(1)">wqdas<h1> x

Write a comment...

Post

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Quando esta vulnerabilidade está presente numa aplicação, o utilizador pode aceder ou modificar a base de dados da mesma através da inserção de comandos SQL, próprios para o efeito, visto que não há uma verificação que neutralize este tipo de comandos maliciosos.

Sem o correto tratamento do input feito pelo utilizador, a aplicação pode ser enganada, levando-a a usar o input inserido como código SQL, o que prejudica a integridade da aplicação.

Neste trabalho fizemos um basic SQL Injection em que conseguimos aceder à aplicação sem ter acesso ao nome de um utilizador e à respetiva password.

De forma a que esta vulnerabilidade existisse na nossa app vulnerável, construímos um query SQL em que o username e a password estivessem juntos. Isto causa problemas porque caso o utilizador consiga manipular corretamente o campo do username, vai conseguir ultrapassar a password.

O query utilizado foi o seguinte:

```
// get data from database
$sql = "SELECT * FROM users WHERE (((email='".$email."')) AND (((password='".$pass."')));
```

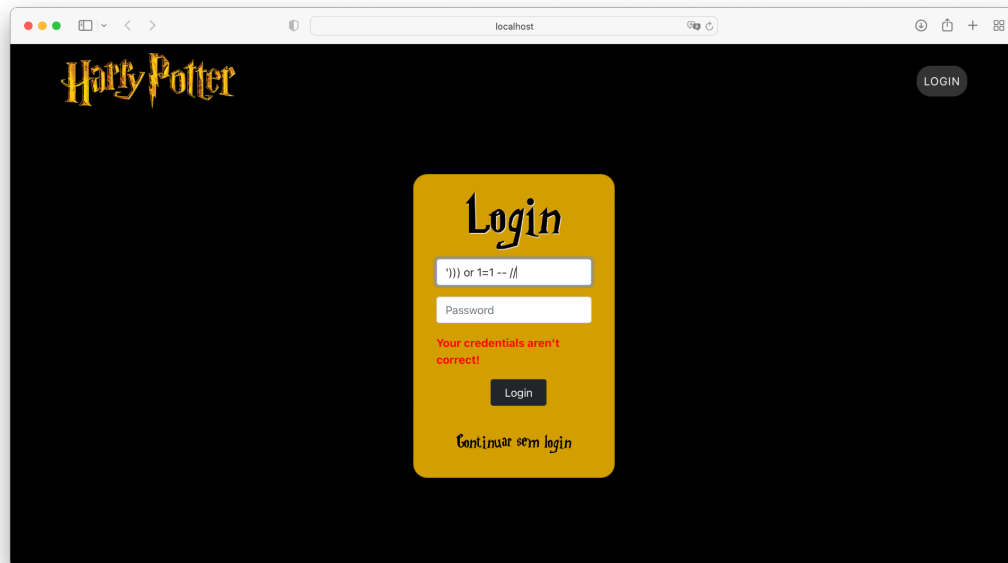
Utilizámos vários pares de parênteses de forma a que não fosse tão trivial de se fazer o SQL Injection.

Desta forma, o login seria conseguido se o utilizador colocasse no campo do utilizador algo como: `))) or 1=1 -- //

No entanto, temos também uma parte no código que reforça um pouco mais a segurança da aplicação, que é apenas permitir um login quando o resultado do SQL apenas tem uma "row".

```
if(mysqli_num_rows($result) == 1) {
    // {loggedin:true, username:...}
    $res["loggedin"] = true;
    $res["username"] = $row["nickname"];
    $res["role"] = $row["role"];
} else {
    $res["loggedin"] = false;
    $res["erro"] = "More than one user on the table.";
}
```

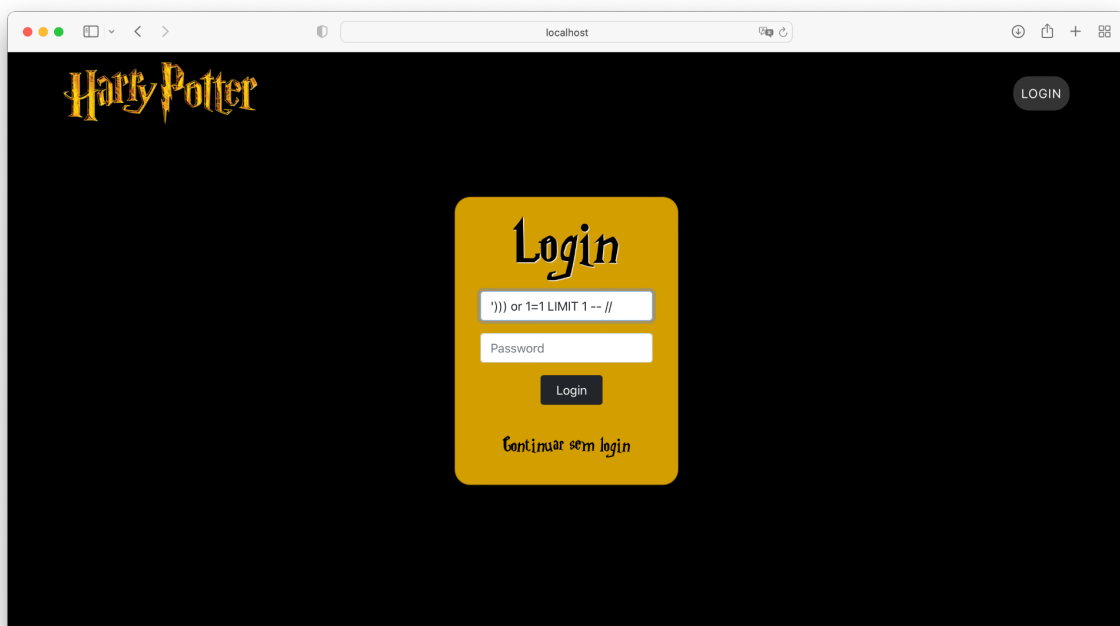
Sendo assim, ao executar o SQL Injection que referi anteriormente, o Login não vai ser efetuado porque ao fazermos “SELECT * “ vão ser devolvidas várias rows (de acordo com o número de users que existirem).



Sendo assim, para realmente conseguir fazer o verdadeiro SQL Injection, o utilizador terá que adicionar um ‘LIMIT 1’ ao código SQL, para apenas obter uma row de resultado.

Assim, o input a colocar no campo do username seria: ‘)) or 1=1 LIMIT 1 -- //

Desta forma, conseguiríamos ultrapassar o login e seríamos redirecionados para a página de escolha das personagens já com o login efetuado.



Versão Segura

Para corrigirmos este problema, começamos por verificar o input do email¹, retirando caracteres desnecessários ou que possam trazer alguma vulnerabilidade:

```
if(isset($dataObject->email))
{
    $email = verifyInput($dataObject->email);
}
```

E depois, fizemos o tratamento do SQL com um prepared statement, que é mais seguro do que da maneira anterior, uma vez que os parâmetros do query são enviados depois da query e com outro protocolo:

```
$sql = "SELECT * FROM users WHERE email=?";
$stmt = mysqli_prepare($conn, $sql);

mysqli_stmt_bind_param($stmt, 's', $email);

mysqli_stmt_execute($stmt);

$result = mysqli_stmt_get_result($stmt);
$result = mysqli_fetch_assoc($result);
```

A função *mysqli_prepare* prepara o sql para ser executado e devolve um statement, que vai permitir outras operações.

A função *mysqli_stmt_bind_param* vai fazer bind do valor da variável *\$email* (input do utilizador no campo email) e do statement devolvido na função anterior.

A *mysqli_stmt_execute* vai então executar o comando SQL completo já com o email incluído.

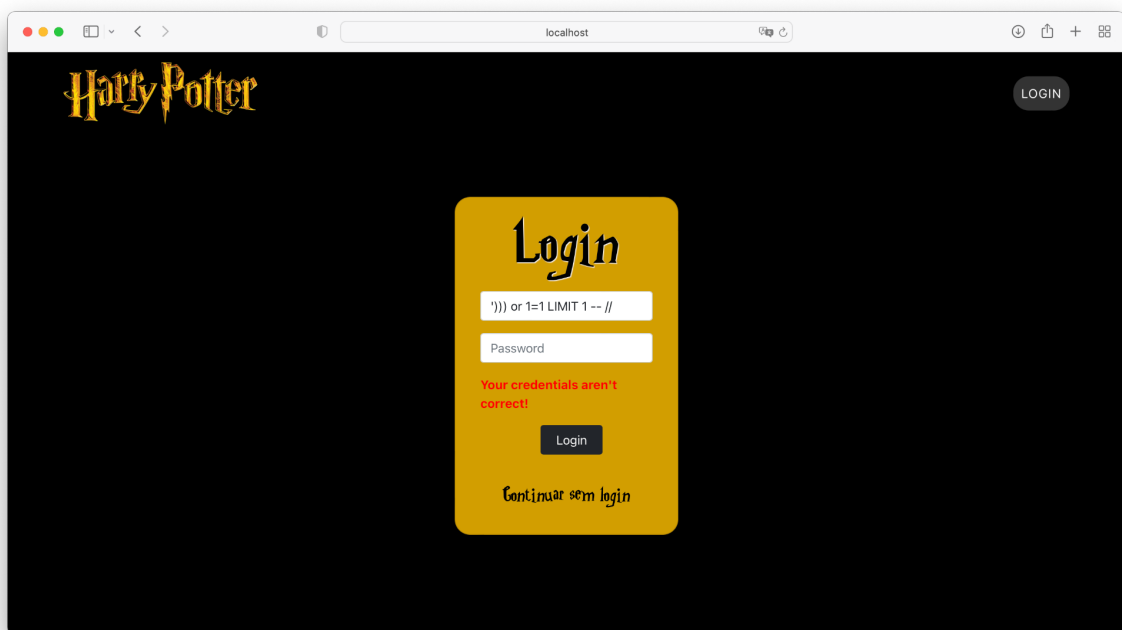
A *mysqli_stmt_get_result* vai buscar o resultado produzido pela última função, e a *mysqli_fetch_assoc* devolve os resultados em forma de array, ou seja, um array com os dados da row que o SQL devolveu.

Depois, vai ser feita uma verificação para ver se o email existe, e se existir e a password fornecida corresponder à password da base de dados, o login vai ser bem sucedido. Caso contrário, vai dar um erro:

¹ A verificação de se um email foi realmente inserido é realizada pelo React, o que impede a inserção de código SQL. Caso essa verificação não fosse feita, o código aqui descrito impediria o sql injection.


```
// user exists
if(isset($result["nickname"])) {
    // check if password is correct
    if($pass == $result["password"]){
        // {loggedin:true, username:...}
        $res["loggedin"] = true;
        $res["username"] = $result["nickname"];
        $res["role"] = $result["role"];
        $_SESSION["user"] = $result["nickname"];
        $_SESSION["role"] = $result["role"];
    } else {
        // {loggedin: false; erro: ...}
        $res["loggedin"] = false;
        $res["error"] = "Wrong password.";
    }
}
```

Com estas modificações, deixamos de conseguir fazer SQL Injection:



CWE-862: Missing Authorization

A seguinte vulnerabilidade está relacionada com a falta de verificação quando um utilizador realiza uma dada ação ou acede a um determinado recurso. No caso do nosso projeto, decidimos implementar o *cwe-862* da seguinte maneira: existem dois tipos de utilizadores do nosso site, utilizadores *admin* (administradores) e utilizadores comuns. Para acentuar essa diferença, foi atribuído um *role* com valor 0 para os utilizadores comuns e um *role* com valor 1 para os utilizadores *admin*.

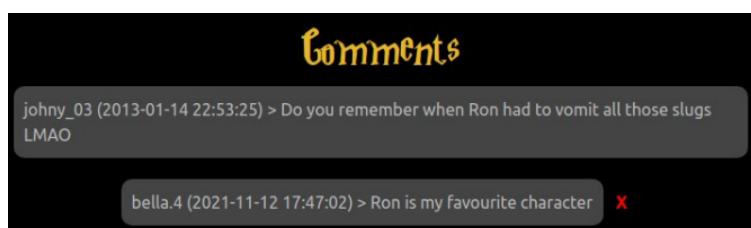
Verificando na base de dados, podemos concluir que existem 2 utilizadores comuns (isabella@hotmail.com e john@gmail.com) e um utilizador *admin* (lili_martinha@gmail.com)

```
INSERT INTO `users` (`name`, `nickname`, `email`, `password`, `role`) VALUES
('isabel', 'bella.4', 'isabella@hotmail.com', 'a4a2d4153f5bf93107c23e2750f11336', 0),
('joao', 'johny_03', 'john@gmail.com', '5788a97c728c710956e3d1f87c8429d4', 0),
('Liliana', 'lilimarta74', 'lili_martinha@gmail.com', 'e5862159d8c557f89a580e072b096f45', 1);
```

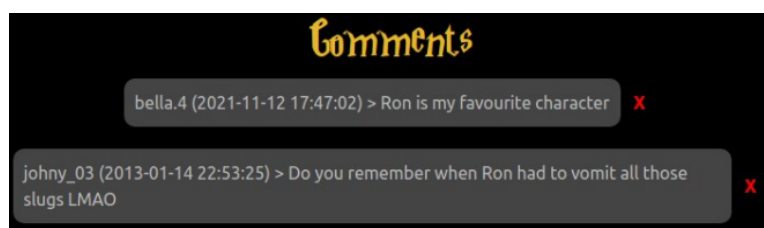
Os utilizadores comuns apenas podem apagar os seus próprios comentários na página de informação do personagem, enquanto que utilizadores *admin* podem apagar qualquer comentário.

```
{state.comments.map((comment, index) => {
  const condition =
    user?.role === 1 || comment.nickname === user?.username;
```

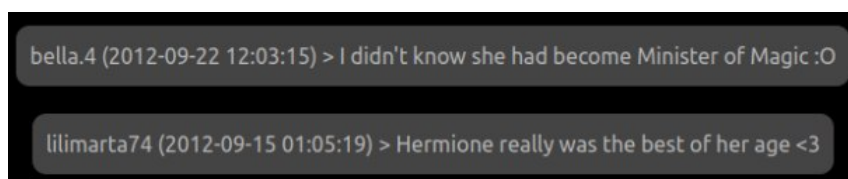
(Verificação da condição se o *user* é *admin* ou se o comentário é dele próprio)



(Ponto de vista da *bella.4* - *user* normal)

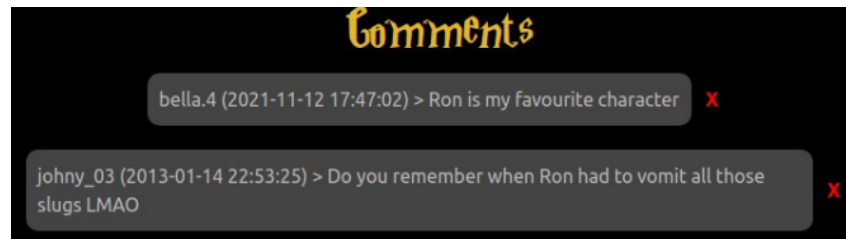


(Ponto de vista da *lilimarta74* - *user* admin)



(Ponto de vista de um utilizador sem *login*)

Todavia, na aplicação vulnerável, é possível que um utilizador comum ou até mesmo um utilizador que não tenha feito *login* apague o comentário de outro utilizador (seja *admin* ou comum).



(Ponto de vista de um utilizador sem *login*)

CWE-20: Improper Input Validation

A vulnerabilidade mencionada funciona na medida em que não ocorre uma verificação correta do *input*, permitindo assim que dados incorretos ou não seguros sejam processados.

No presente projeto, optámos por aplicar essa vulnerabilidade no campo referente ao *email* na página de *login*. No caso da aplicação com vulnerabilidades, o campo mencionado é do tipo *text*, enquanto que na aplicação segura esse mesmo campo é do tipo *email*. Optámos por essa abordagem, uma vez que a biblioteca *react* permite que a verificação do *email* seja feita automaticamente especificando o *type* do campo.

```
<Form.Group className="mb-3" controlId="formBasicEmail">
  <Form.Control
    type="text"
    placeholder="Email"
    autoComplete="off"
    onChange={handleChange}
  />
</Form.Group>
```

(Aplicação vulnerável)

```
<Form.Group className="mb-3" controlId="formBasicEmail">
  <Form.Control
    type="email"
    placeholder="Email"
    autoComplete="off"
    onChange={handleChange}
  />
</Form.Group>
```

(Aplicação não vulnerável)

A screenshot of a login form titled "Login" on a yellow background. The form has a text input field containing "isabel". Below the input field, a white error message box with a yellow exclamation mark icon displays the text: "Inclua um '@' no endereço de e-mail. 'isabel' está com um '@' faltando." Below the error message is a dark grey "Login" button. At the bottom of the form is a link that says "Continuar sem login".

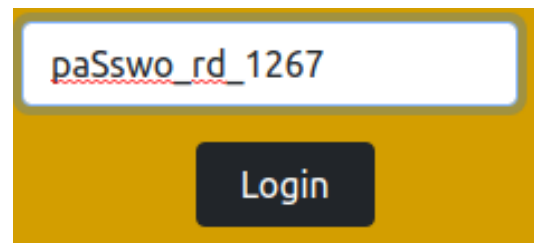
(Aplicação não vulnerável)

CWE-549: Missing Password Field Masking

A última vulnerabilidade diz respeito à falta de capacidade do *software* mascarar as *passwords* quando as mesmas estão a ser inseridas, aumentando assim a probabilidade de terceiros capturarem *passwords* alheias.

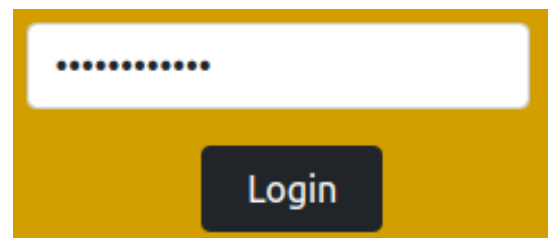
Seguindo o mesmo raciocínio que o ponto anterior, para aplicar a vulnerabilidade utilizámos o campo referente à *password*; este campo é do tipo *text* na aplicação com vulnerabilidades, enquanto que na aplicação segura ele é do tipo *password*. Escolhemos essa abordagem, uma vez que a biblioteca *react* permite que as *passwords* sejam mascaradas automaticamente especificando o *type* do campo.

```
<Form.Group className="mb-3" controlId="formBasicPassword">
  <Form.Control
    type="text"
    placeholder="Password"
    autoComplete="off"
    onChange={handleChange}
  />
</Form.Group>
```



(Aplicação vulnerável)

```
<Form.Group className="mb-3" controlId="formBasicPassword">
  <Form.Control
    type="password"
    placeholder="Password"
    autoComplete="off"
    onChange={handleChange}
  />
</Form.Group>
```



(Aplicação não vulnerável)