

INFORMÁTICA PARA INTERNET

Segurança da informação no desenvolvimento de aplicações *web*

Neste material, vamos aprofundar o estudo sobre a utilização das melhores práticas envolvendo aspectos de segurança e desenvolvimento de *sites* seguros, dando continuidade ao trabalho com a plataforma ASP.NET MVC da Microsoft. O uso extensivo da Internet aumenta diretamente a preocupação com a segurança em razão do imenso tráfego de informações sigilosas.

Possivelmente, sempre vai haver dúvida sobre se o emissor é realmente quem deveria ser, se os dados foram modificados durante o tráfego e se o receptor realmente é quem deveria ser.

Neste material, também vamos tratar da criação de *sites* mais seguros e ver como criar certificados para páginas seguras (HTTPS – *hypertext transfer protocol secure*), além de ter acesso a dicas usando bancos de dados.

Em nossos estudos com a utilização do VS Code, criamos até o momento *sites* do tipo HTTP (*hypertext transfer protocol*), ou seja, sem protocolos de segurança e sem o uso de certificados. Para isso, usamos a opção **-no-https** no comando para a criação dos exemplos anteriores. Agora, vamos experimentar criar um *site* seguro do tipo HTTPS, para dar início aos nossos estudos. Com o VS Code aberto, vamos criar uma nova pasta com o nome **SiteHttps**. Como já temos conhecimento sobre o uso da linha de comando (**dotnet new mvc**), nosso projeto ficará pronto em segundos, conforme a figura seguir.

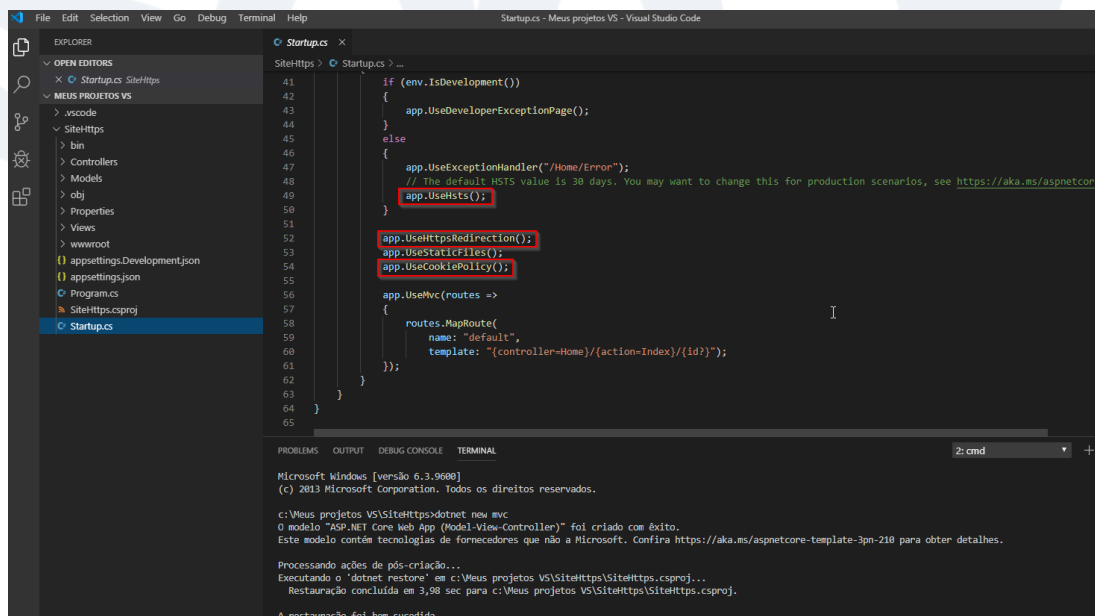


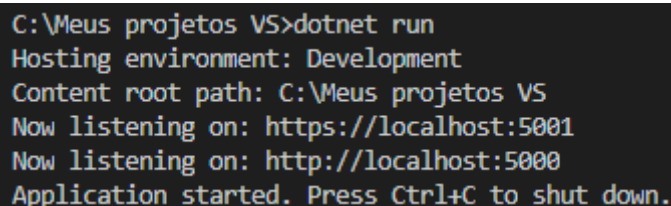
Figura 1 – Criando o projeto **SiteHttps**

Tela do programa VS Code mostrando o código fonte do arquivo Startup.cs, com retângulos vermelhos evidenciando as funções descritas no texto.

Após uma olhada no final do arquivo **Startup.cs**, é possível perceber algumas diferenças:

- ◆ O método **UseHsts()** ativa o uso do protocolo HSTS (http strict transport security), ou seja, será iniciada uma aplicação com o uso de segurança de transporte com segurança rigorosa, ou HTTPS.
- ◆ O método **UseHttpsRedirection()** permite o redirecionamento automático de HTTP para HTTPS.
- ◆ O método **UseCookiePolicy()** permite ativar rotinas de segurança no uso de *cookies*. A seguir, vamos estudar estes aspectos com mais detalhes.

Até aqui, além dos três métodos já mencionados, tudo parece igual. No entanto, após o comando “dotnet run”, vamos ver uma diferença, conforme a figura a seguir.



```
C:\Meus projetos VS>dotnet run
Hosting environment: Development
Content root path: C:\Meus projetos VS
Now listening on: https://localhost:5001
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Figura 2 – Diferença após o comando

Tela do programa VS Code mostrando a linha de comando com as respostas citadas no texto.

Além do *link* tradicional, apareceu outro *link* <https://localhost:5001>. Vamos clicar neste último *link*, sem esquecer da tecla **Ctrl**, para acessar nosso *site* seguro recém-criado.



Sua conexão não é particular

Invasores podem estar tentando roubar suas informações de **localhost** (por exemplo, senhas, mensagens ou cartões de crédito). [Saiba mais](#)

NET::ERR_CERT_AUTHORITY_INVALID

☐ Ajudar a melhorar o recurso "Navegação segura" enviando algumas [informações do sistema e conteúdo da página](#) para o Google. [Política de Privacidade](#)

Ocultar detalhes

Voltar à segurança

Este servidor não conseguiu provar que é **localhost**. O certificado de segurança não é confiável para o sistema operacional do seu computador. Isso pode ser causado por uma configuração incorreta ou pela interceptação da sua conexão por um invasor.

[Ir para localhost \(não seguro\)](#)

Figura 3 – Aviso de *site* sem certificado de segurança

Tela do navegador mostrando aviso de segurança: “Sua conexão não é particular”.

O aviso que aparece na figura anterior indica que estamos tentando acessar um *site* seguro sem o devido certificado de segurança. Além dessa mensagem, há o aviso ao lado do nome do *site* (URL – *uniform resource locator*), onde deveria aparecer o cadeado verde fechado, indicando que o *site* é seguro.


 Não seguro | localhost:5001

Figura 4 – URL não seguro

Barra de endereços do navegador em que aparece a mensagem “Não seguro” ao lado do endereço “localhost:5001”.



Os certificados SSL

Quando trabalhamos com *sites* que utilizam somente o protocolo HTTP, toda a informação que circula pela rede é facilmente visível, ou seja, é composta de texto puro com códigos específicos, dependendo da tecnologia utilizada na construção do *site*. Nesse sentido, se alguma informação sigilosa for acessada, ela pode ser visível facilmente. Por este motivo, é necessário utilizar criptografia e autenticação para proteger não só os dados, mas todo o sistema por trás do *site*.

Criptografia é o processo de codificar informação. Geralmente, este processo é feito com a utilização de uma chave, de modo que não seja possível ler a informação original sem a chave original (em vez disso, apenas informações desconexas estarão visíveis). Autenticação, por sua vez, é a verificação da identidade de um usuário de sistema (geralmente, acontece por meio de *log in*).

Aplicando corretamente essas técnicas, pode-se diminuir consideravelmente o risco de invasão e captura de dados sensíveis. Para que isso seja possível, vamos utilizar os certificados SSL (Secure Socket Layer – camada de soquete seguro, em português), associados à tecnologia de construção de *sites* seguros. Dependendo da complexidade e do tipo de *site*, basta um certificado para resolver o problema da segurança. Também pode-se optar pela autenticação logo no primeiro acesso (*log in*), tornando seguro todo o restante do acesso.

No fim do acesso, o usuário pode se desconectar com segurança (*log off*). Os certificados podem ser comprados de autoridades certificadoras com alcance mundial, com preço relativamente baixo, principalmente quando consideramos o prejuízo causado pela falta de segurança. Existem autoridades nacionais com preços atraentes, como Certisign, DigitalSign etc. Há também provedores de hospedagem que prestam este serviço aos seus clientes. Na contratação do serviço de hospedagem do *site*, muitos provedores já incorporam o serviço de certificação e as empresas certificadoras dispõem de várias modalidades de certificados.

Os certificados SSL

Quando trabalhamos com *sites* que utilizam somente o protocolo HTTP, toda a informação que circula pela rede é facilmente visível, ou seja, é composta de texto puro com códigos específicos, dependendo da tecnologia utilizada na construção do *site*. Nesse sentido, se alguma informação sigilosa for acessada, ela pode ser visível facilmente. Por este motivo, é necessário utilizar criptografia e autenticação para proteger não só os dados, mas todo o sistema por trás do *site*.

Criptografia é o processo de codificar informação. Geralmente, este processo é feito com a utilização de uma chave, de modo que não seja possível ler a informação original sem a chave original (em vez disso,

apenas informações desconexas estarão visíveis). Autenticação, por sua vez, é a verificação da identidade de um usuário de sistema (geralmente, acontece por meio de *log in*).

Bloco 5

Aplicando corretamente essas técnicas, pode-se diminuir consideravelmente o risco de invasão e captura de dados sensíveis. Para que isso seja possível, vamos utilizar os certificados SSL (Secure Socket Layer – camada de soquete seguro, em português), associados à tecnologia de construção de *sites* seguros. Dependendo da complexidade e do tipo de *site*, basta um certificado para resolver o problema da segurança. Também pode-se optar pela autenticação logo no primeiro acesso (*log in*), tornando seguro todo o restante do acesso.

No fim do acesso, o usuário pode se desconectar com segurança (*log off*). Os certificados podem ser comprados de autoridades certificadoras com alcance mundial, com preço relativamente baixo, principalmente quando consideramos o prejuízo causado pela falta de segurança. Existem autoridades nacionais com preços atraentes, como Certisign, DigitalSign etc. Há também provedores de hospedagem que prestam este serviço aos seus clientes. Na contratação do serviço de hospedagem do *site*, muitos provedores já incorporam o serviço de certificação e as empresas certificadoras dispõem de várias modalidades de certificados.

Clique ou toque para visualizar o conteúdo.

O que é certificado SSL?

Embora pareça algo complexo, trata-se, na verdade, de um mecanismo de criptografia que, quando instalado no servidor que hospeda as páginas do seu *site*, permite que o protocolo passe de HTTP para HTTPS

Assim, o navegador que acessar o *site* vai exibir o cadeado verde indicando que o endereço é seguro. Mesmo que se digite apenas HTTP, o sistema direciona automaticamente para o HTTPS.

Os navegadores contam com chaves criptográficas das autoridades certificadoras, com a capacidade de reconhecer as outras chaves que estão instaladas devidamente nos servidores que hospedam os *sites*. Assim, um círculo criptográfico é fechado, adicionando uma camada extra de proteção que cobre desde o usuário até os dados internos do *site*.

Quando, por exemplo, um número de cartão de crédito é digitado pelo usuário no navegador, os números são criptografados antes de serem enviados. Se um *hacker* conseguir ter acesso à rede e tentar ver os dados, só vai visualizar um emaranhado de caracteres quase impossíveis de serem reconhecidos. No entanto, com o uso de supercomputadores e um tempo considerável (talvez anos), é possível quebrar a chave e acessar as informações.

Existem três tipos básicos de certificados baseados em SSL.

Clique ou toque para visualizar o conteúdo.

Validação de domínio

O primeiro certificado é chamado de “validação de domínio” e tem certificados apenas para uso em *sites*. Estes certificados são mais baratos, fornecem criptografia básica, têm emissão rápida e servem apenas para validar o domínio, garantindo a sua identidade. Com o uso destes certificados, o *site* já terá a indicação do cadeado verde fechado à esquerda da URL, conforme exemplo da figura a seguir.

Figura 5 – Exemplo de *site* certificado

Barra de endereços do navegador mostrando a palavra “Seguro” ao lado do cadeado verde fechado citado no texto.

Validação de organizações

O certificado de validação de organizações tem um nível mais alto de segurança e também valida as informações de uma empresa por trás do domínio e até a sua presença física. Com este certificado, os clientes sabem que podem confiar nos seus serviços e enviar informações pessoais.

Validação estendida

Já os certificados de validação estendida são os mais completos. Eles prometem segurança mais profunda com relação à própria empresa e ao *site*, proporcionando o mais alto grau de confiança por parte do usuário. Com este tipo de certificado, a URL aparece com o nome da empresa em destaque. A seguir, veja um exemplo.

Figura 6 – *Site* com certificação de organização

Print da tela da barra de endereços do navegador mostrando, além do cadeado fechado, o nome da empresa cujo *site* seguro foi acessado.

O exemplo anterior incorpora métodos iniciais e necessários para que *site* seja seguro. Quando não se indica especificamente no comando `dotnet new` que o *site* não vai ser seguro (`--no-https`), será criado um *site* seguro, pois este é o padrão. Assim, o método `app.UseHsts()` vai ativar o uso do protocolo HTTPS com o uso do SSL. Já o método `app.UseHttpsRedirection()` permite o redirecionamento da URL, ou seja, se digitarmos **www.meusiteseguro.com**, seremos redirecionados automaticamente para **https://www.meusiteseguro.com**.

Quando nosso *site* é acessado pela primeira vez, o aviso de *site* não seguro aparece. Se clicarmos no botão **Avançado** e, depois, em **Ir para o localhost (não seguro)**, criaremos uma exceção que vai permitir o acesso ao nosso *site*, mesmo sem termos o certificado instalado. Este aviso pode aparecer quando acessamos *sites* na Internet que podem realmente podem ser falsificados (alguém mal-intencionado pode ter criado uma réplica de algum *site* conhecido). Como o provável fraudador não tem o certificado verdadeiro instalado, o navegador não reconhece a autenticidade do endereço e emite o aviso, mas podemos criar outra exceção por nossa própria conta e risco.

Uma boa dica para tornar os nossos *sites* locais certificados é utilizar uma ferramenta disponível no próprio .NET. O comando é bem simples: `dotnet dev-certs https --trust`. Na figura a seguir, veja um diálogo de confirmação.

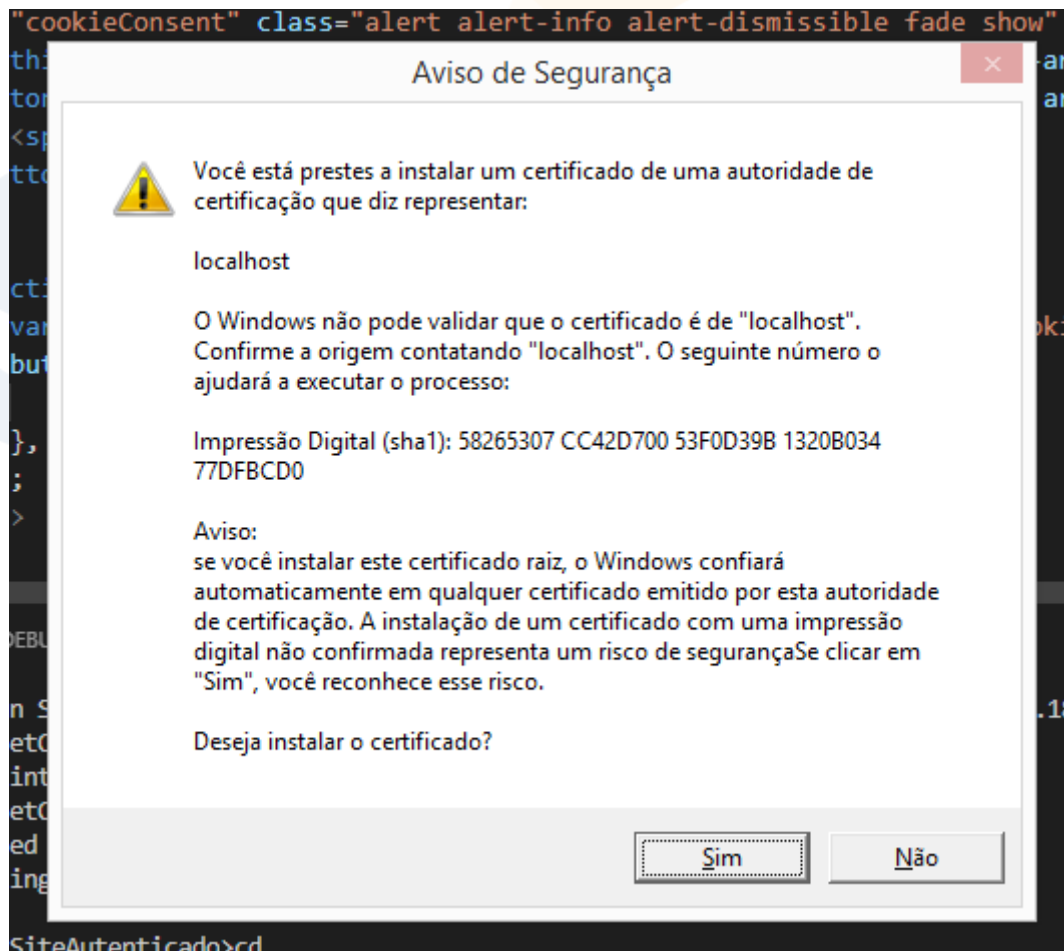


Figura 7 – Diálogo de confirmação de instalação de certificado

Após a confirmação e o comando `dotnet run`, finalmente teremos o nosso *site* local confirmado pelo cadeado fechado. No entanto, é importante lembrar que este certificado somente funciona para nossos *sites* locais (*localhost*).

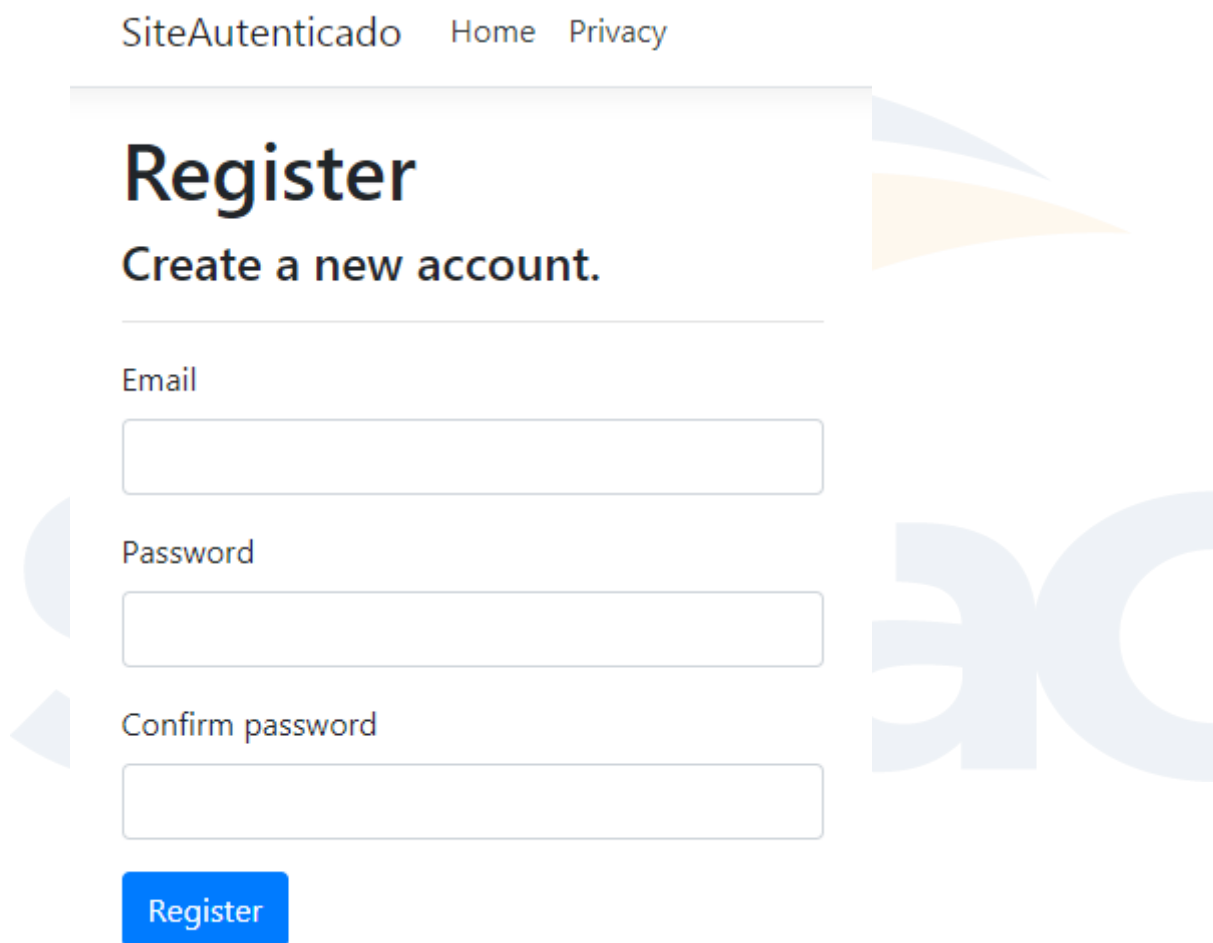


Figura 8 – Nosso *site* local certificado



Outra técnica de segurança é a autenticação feita pelo próprio usuário, por meio de *e-mail* e senha previamente cadastrados. Somente após esta verificação é que o *site* vai liberar as atividades que requerem mais segurança. Por exemplo, em um *site* de vendas *on-line*, qualquer um pode ver todas as ofertas, mas somente na hora de comprar é que páginas, métodos e autorizações para acesso à base de dados responsáveis pela oficialização da venda estarão acessíveis. A partir deste ponto é que trafegam os dados sigilosos do usuário, pois já foi feita a autenticação e foi criada uma linha de confiança entre o usuário e o sistema. Os dados já estavam trafegando por um canal criptografado, em virtude do uso da segurança SSL pelo protocolo HTTPS, ou seja, o acesso à página já estava seguro.

Crie outra pasta no VS Code com o nome **SiteAutenticado**. Na linha de comando, digitamos `dotnet new mvc --auth Individual`. Esta última opção no comando informa ao VS Code que queremos um *site* que exige autenticação do usuário. Se ele não tiver uma conta, deve registrar um *e-mail* e uma senha. Este exemplo vai utilizar o banco de dados SQLite, mais simples e já disponível no VS Code. Assim, os usuários terão os seus dados permanentes no sistema. Após rodar o *site* no navegador e clicar em **Register**, teremos a tela inicial de cadastro de novo usuário, conforme a figura 9.



SiteAutenticado Home Privacy

Register

Create a new account.

Email

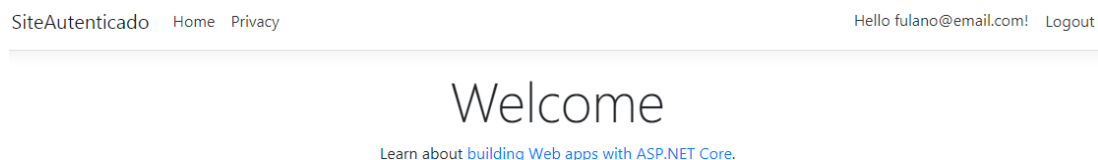
Password

Confirm password

Register

Figura 9 – Tela de cadastro de novo usuário

Para cadastrar novo usuário, você pode usar um *e-mail* qualquer. Em nosso exemplo, utilizamos **fulano@email.com**. A senha deve conter caracteres maiúsculos e minúsculos, além de número ou caractere de pontuação, como “Pa\$\$word1”. Após autenticar, veremos a mensagem “Hello fulano@email.com”, confirmando a autenticação.



SiteAutenticado Home Privacy

Welcome

Learn about [building Web apps with ASP.NET Core](#).

Hello fulano@email.com! Logout

Figura 10 – Autenticação reconhecida

Após este reconhecimento, podemos liberar métodos e páginas que vão fazer o trabalho sigiloso do nosso *site* com total segurança, pois estamos cadastrados no sistema.



O problema da injeção de SQL

A injeção de SQL é uma vulnerabilidade que permite que uma pessoa mal-intencionada consiga acesso a um banco de dados por meio de manipulação de uma consulta. Por exemplo, em um sistema de vendas *on-line*, no qual se preenche o nome de usuário e a senha, se não forem tomadas as devidas precauções, há a possibilidade de uso de comandos específicos de consulta que podem levar a um acesso indevido ao banco de dados que guarda os nomes e as senhas dos usuários.

Em primeiro lugar, com o XAMPP já instalado e tendo sido ativados (*start*) os serviços do Apache e MySQL, clicamos no botão **Admin** do MySQL para termos acesso ao phpMyAdmin, a fim de criarmos um banco de dados. Este banco vai se chamar **UsuariosDB** e vai conter apenas duas colunas, com o nome de usuários e as senhas.

No menu à esquerda, clicamos em **Novo** e inserimos o nome da base de dados (UsuariosDB). O resultado mostrado pode ser visto na figura 8.

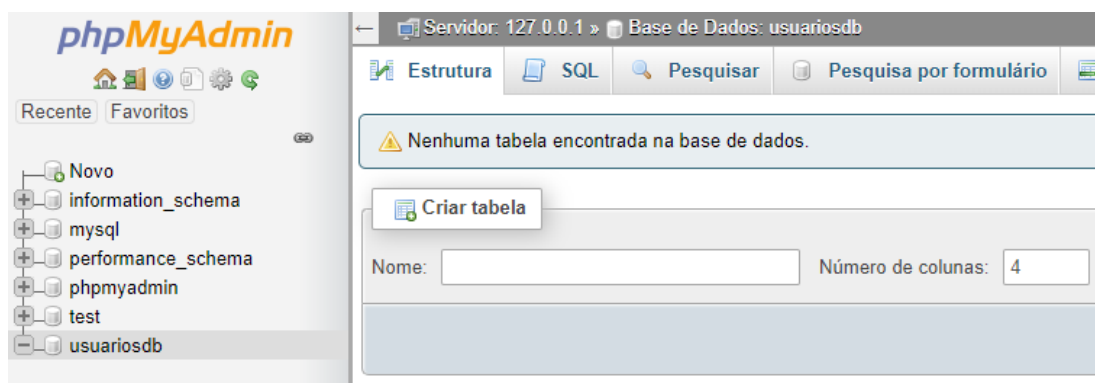


Figura 11 – Base de dados criada no MySQL com o auxílio do phpMyAdmin

Em seguida, clicamos acima, no menu SQL, para inserir um pequeno *script* com instruções SQL para criar a tabela **tabelausuarios**. Após digitar o *script*, tecele no botão **Executar** à direita.

Vamos dar uma olhada no que acabamos de fazer?

Na linha 1, vamos abrir nossa base de dados para podermos manipular. Na linha 3, vamos criar a tabela **tabelausuarios** com três campos: o primeiro é a indispensável chave primária e os outros dois campos são para os nomes de usuários e senhas, que podem conter qualquer tipo de caracteres com até 50 dígitos. Finalmente, na linha 10, vamos inserir os nomes e as senhas de dois usuários para podermos fazer nossos testes.

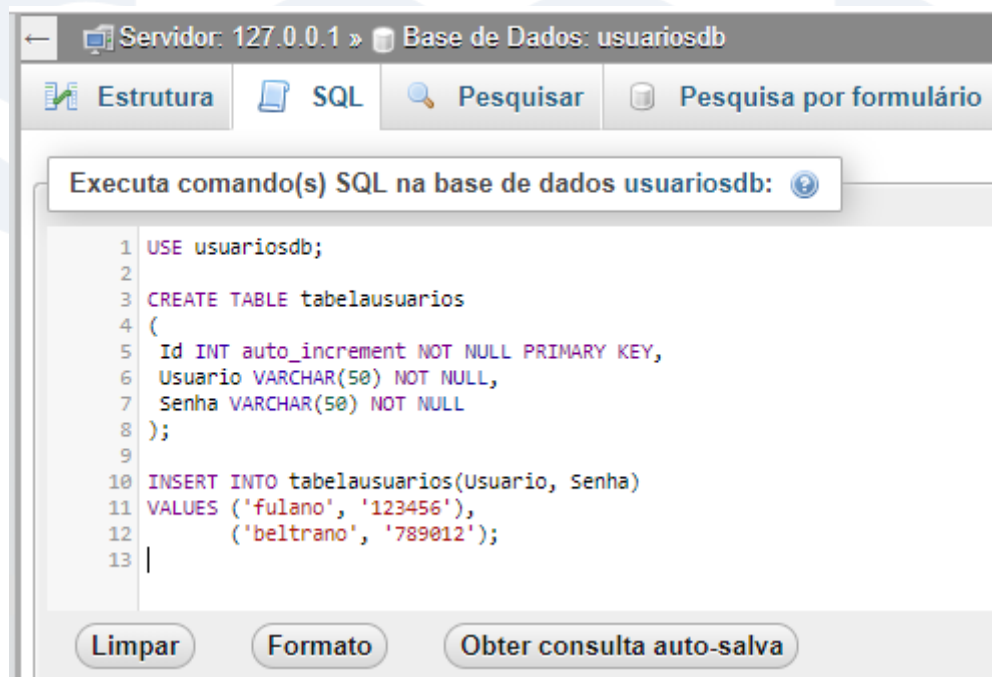


Figura 12 – *Script* que cria e preenche a base de dados

Agora vem a parte de testes do nosso *site*, então não esqueça de manter ativado o serviço MySQL no XAMPP. Vamos criar um novo projeto no VS Code. Crie uma pasta chamada SQLInjection para abrigar o teste. Vamos criar um projeto MVC no console com o comando `dotnet new mvc`. Antes de continuar, temos que acrescentar as bibliotecas do MySQL para que possamos utilizá-lo adequadamente. Para isso, digitamos no console `dotnet add package mysqlconnector`.

Logo após, crie um arquivo **EntradaController.cs** dentro da pasta **Controllers** e copie o conteúdo a seguir para dentro deste arquivo. Após a cópia, vamos salvar o arquivo teclando o comando **Ctrl + s**.

```
using System;

using Microsoft.AspNetCore.Mvc;

using MySql.Data.MySqlClient;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Threading.Tasks;

using SQLInjection.Models;

namespace SQLInjection.Controllers

{

    public class EntradaController : Controller

    {

        [HttpGet]

        public ActionResult Login()

        {

            return View();

        }

    }

}
```

```
}

[HttpPost]

public ActionResult Login(string usuario, string senha)

{

    try

    {

        string stringConexao = "Database=Usuariodb;Data
Source=localhost;User Id=root;";

        using (var conexao = new MySqlConnection(stringConexao))

        {

            conexao.Open();

            var consulta = "SELECT COUNT(*) FROM tabelausuarios
WHERE usuario = '" + usuario + "' AND Senha = '" + senha + "'";

            MySqlCommand myCommand = new
MySqlCommand(consulta);

            myCommand.Connection = conexao;

            Int64 resultado = (Int64)myCommand.ExecuteScalar();

            if (resultado > 0)

                ViewBag.Mensagem = "Login efetuado com sucesso";

            else
```

```
        ViewBag.Mensagem = "Falha no login";

        myCommand.Connection.Close();

    }

}

catch (Exception e)

{

    ViewBag.Mensagem = "Erro: " + e.Message;

}

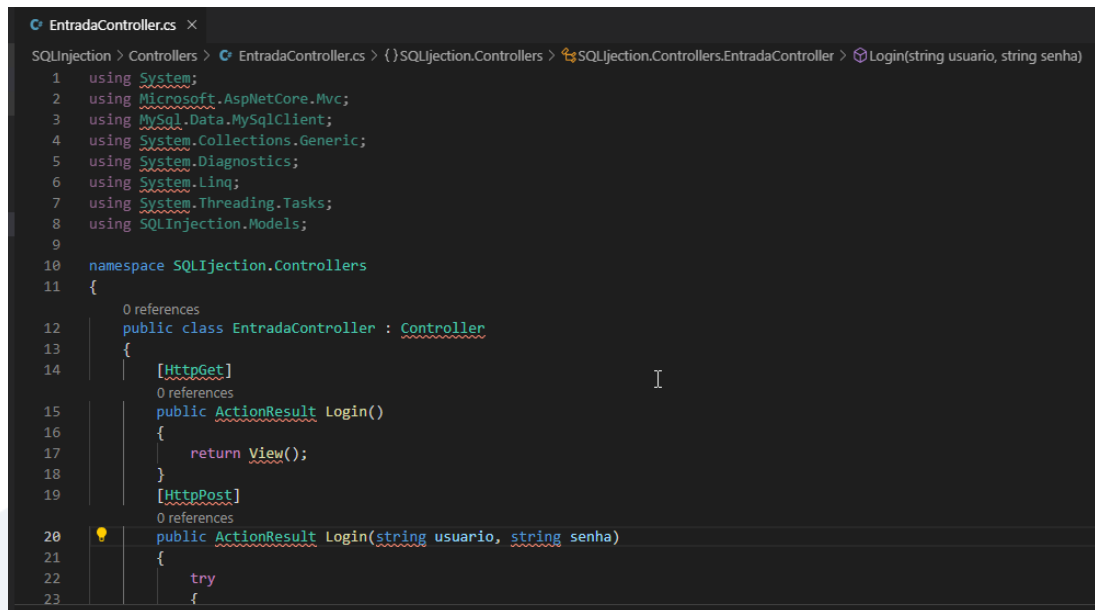
return View();

}

}

}
```

Após a cópia, o VS Code ficará da seguinte forma:



```

EntradaController.cs
SQLInjection > Controllers > EntradaController.cs > {} SQLInjection.Controllers > SQLInjection.Controllers.EntradaController > Login(string usuario, string senha)
1  using System;
2  using Microsoft.AspNetCore.Mvc;
3  using MySql.Data.MySqlClient;
4  using System.Collections.Generic;
5  using System.Diagnostics;
6  using System.Linq;
7  using System.Threading.Tasks;
8  using SQLInjection.Models;
9
10 namespace SQLInjection.Controllers
11 {
12     0 references
13     public class EntradaController : Controller
14     {
15         [HttpGet]
16         0 references
17         public ActionResult Login()
18         {
19             return View();
20         }
21         [HttpPost]
22         0 references
23         public ActionResult Login(string usuario, string senha)
24         {
25             try
26             {
27             }
28         }
29     }
30 }

```

Figura 13 – Arquivo **EntradaController.cs**

Para a nossa página inicial de *log in*, vamos criar uma pasta com o nome **Entrada**, dentro da pasta **Views**. Dentro daquela pasta, criamos um arquivo chamado **login.cshtml**. Cole o conteúdo a seguir dentro deste arquivo. Após a cópia, vamos salvar o arquivo teclando o comando **Ctrl + s**.

```
<h2>Login</h2>
```

```
@using (Html.BeginForm("Login", "Home", FormMethod.Post, new {
    enctype = "multipart/form-data" }))
```

```
{
```

```
<label>Usuario: </label>
```

```
<input type="text" id="usuario" name="usuario" /><br />
```

```
<label>Senha: </label>
```

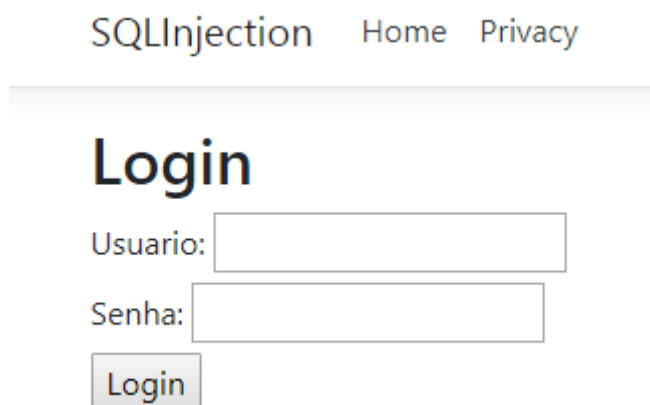
```
<input type="text" id="senha" name="senha" /><br />
```

```
<input type="submit" value="Login" /><br />
```

```
<label>@ViewBag.Mensagem</label>
```

```
}
```

Na próxima etapa, vamos compilar e montar o projeto com o dotnet build. Agora vem a melhor parte: dotnet run. Para ver o resultado no navegador, digite na barra de endereços: **localhost:5001/entrada/login**. Se tudo correu bem, teremos a tela inicial de *log in*, na qual vamos fazer o teste de injeção de SQL.



SQLInjection Home Privacy

Login

Usuario:

Senha:

Login

Figura 14 – Tela de *log in*

Nesta tela inicial, digitamos o usuário “fulano” e a senha “123456”. Após clicar no botão **Log in**, teremos acesso ao sistema, com a mensagem “Log in efetuado com sucesso!”. Tente digitar outro usuário e senhas erradas, para se certificar de que a autenticação realmente funciona.

Vamos direto ao assunto, dando uma olhada na linha 28 do arquivo **entradacontrollers.cs**:

```
var consulta = "SELECT COUNT(*) FROM tabelausuarios WHERE  
usuario = " + usuario + " AND Senha = " + senha + "";
```

Esta linha monta, de forma concatenada, a consulta SQL à base de dados pesquisando pelos nomes de usuários e senhas que foram digitados. Se os dados conferirem com a tabela, o resultado, após a sua execução na linha 32, será um número maior do que 0. A seguir, na linha 33, temos o teste condicional que informa o sucesso ou não da tentativa de *log in*.

Agora, vamos testar. No espaço da senha, digite a seguinte *string* de injeção: (' OR '1'='1'), sem os parênteses e cuidando com os caracteres de espaço antes e depois de “OR”. Clique no botão **Log in** e veja que você entrou no sistema sem sequer saber o nome do usuário. Isso é possível porque, ao digitar a *string* especial, você alterou a forma como é montada a consulta à base de dados, acrescentando o “OR 1=1” no final. Dessa forma, independentemente de o campo do usuário ou da senha serem verdadeiros na consulta, o resultado final será sempre verdadeiro e, assim, você terá o acesso. A seguir, veja como ficaria a consulta modificada pela injeção do *string* SQL:

```
SELECT COUNT(*) FROM tabelausuarios WHERE usuario = 'xxx'  
AND Senha = " OR '1'='1'
```

Esses tipos de ataques podem ocasionar entradas, consultas e alterações indevidas e até a eliminação de tabelas da base de dados.

Esse é um simples exemplo de como não se deve fazer uma consulta, mas serve para mostrar o conceito. Para evitar esse tipo de ataque, deve-se recorrer às boas práticas de programação. Uma maneira simples de resolver é utilizando a parametrização das variáveis envolvidas. Nesse sentido, sempre deve-se utilizar alguma camada de acesso às consultas SQL para evitar esse tipo de problema. O método `Parameters.AddWithValue()`, disponível na biblioteca do C#, filtra a *string* digitada, impedindo o uso de vários caracteres utilizados para injeção de

SQL, tais como ' – \ ; / ' entre outros. Para isso, substituímos a variável **senha**, que recebe o valor digitado, pelo parâmetro **@senha**. Depois, a variável **senha**, que recebe o valor da senha digitada, é filtrada pelo método já citado. Para ver na prática, substitua o seguinte trecho de código mostrado a seguir:

```
var consulta = "SELECT COUNT(*) FROM tabelausuarios WHERE  
usuario = " + usuario + " AND Senha = @Senha";
```

```
MySqlCommand myCommand = new  
MySqlCommand(consulta);  
  
myCommand.Connection = conexao;  
  
myCommand.Parameters.AddWithValue("@Senha",  
senha);
```

```
Int64 resultado = (Int64)myCommand.ExecuteScalar();
```

Agora, tente usar a *string* de injeção novamente para burlar o sistema. Se não estiver funcionando, então significa que nossa proteção foi efetiva e a tentativa de injetar a *string* especial foi filtrada adequadamente.

Se você se sente seguro, pode fazer um exercício. Protegemos apenas o campo da senha. Se você digitar a *string* (' OR '1'='1'--), sem os parênteses, e se atentando ao caractere de espaço após o último hífen e antes e depois do “OR” no campo **usuário**, também conseguirá acesso, seguindo a mesma lógica já descrita anteriormente. Tente usar o mesmo tipo de parametrização para proteger este campo.

Existem muitas outras maneiras de se proteger deste tipo de ataque. Para isso, dois fatores são importantes: devemos evitar este tipo de consulta dinâmica e devemos impedir que a entrada fornecida pelo usuário, que contém SQL mal-intencionado, afete a lógica da consulta executada. De acordo com a OWASP, a principal entidade mundial focada em segurança da informação, existem quatro técnicas principais de defesa.

Instruções já preparadas com consultas parametrizadas

Conforme demonstrado anteriormente, todos os dados são tratados antes de serem enviados às consultas SQL.

Uso de *stored procedures*, ou procedimentos armazenados

Esta técnica, embora não muito eficiente, se utiliza de rotinas armazenadas no próprio banco de dados, que são chamadas pelo aplicativo principal.

Validação de entrada com o uso de *whitelist*, ou lista branca

Este tipo de técnica usa a validação de todos os dados e nomes referentes aos componentes internos da base de dados, impedindo que qualquer entrada de dados feita pelo usuário acabe na consulta.

Caracteres de escape para todas as entradas de usuários

Esta técnica é a última linha de defesa que deve ser utilizada quando nenhuma das outras é viável. Depende de características muito particulares de cada tipo de banco de dados. O método é filtrar cada caractere em busca de caracteres especiais a fim de evitá-los, dificultando a injeção perniciosa.

Além desta primeira linha de defesa, o uso das restrições de acesso ao banco de dados é de suma importância e depende de um projeto bem feito, pois é comum o acesso sem restrições para “facilitar” a vida do desenvolvedor. Sem estas restrições, é muito mais fácil a programação, e o aplicativo simplesmente vai funcionar. A contrapartida desta política é o grande aumento da vulnerabilidade do sistema a vários tipos de ataques. Tentamos mostrar aqui, de maneira bem prática, os fundamentos da injeção de SQL, com o objetivo de apresentar um dos grandes problemas dos desenvolvedores de aplicativos que envolvem bancos de dados e as principais soluções adotadas atualmente.

Como usar a criptografia no C#

As bibliotecas do C# têm uma imensidade de códigos prontos para vários tipos de soluções. Com relação à criptografia, também temos muitos códigos disponíveis no namespace `System.Security.Cryptography`. Nesta biblioteca, existem vários serviços de criptografia que incluem operações de *hash*, geração de números aleatórios, autenticação de mensagens e codificação e decodificação segura de dados.

Uso de MD5

Veremos um exemplo do uso do MD5 por ser o mais simples. Este método criptográfico serve para criar um *hash*. Esta função serve para criar cadeias de caracteres binários de tamanho fixo, partindo de cadeias de caracteres de qualquer tipo e de tamanho variável. O uso mais comum é para proteger senhas. Os caracteres da senha (*string*), após passarem pelo algoritmo MD5, são transformados em uma cadeia de números binários de tamanho predeterminado, chamado de *hash*. Uma pequena alteração na senha vai causar uma grande alteração imprevisível no *hash*.



Por outro lado, de posse de um *hash*, necessitaríamos de um enorme poder computacional e de muito tempo para conseguir obter a *string* que o gerou. Em termos práticos, quando um usuário digita pela primeira vez uma senha em algum sistema, o algoritmo é aplicado imediatamente, criando o *hash*. Neste momento, a senha não está mais disponível, somente o *hash*, que fica armazenado em local seguro. Quando o usuário retorna e digita novamente a senha no seu próprio computador, o mesmo algoritmo é aplicado e gera o *hash*, que é enviado para comparação com o que já foi criado no cadastro inicial. Se os *hashes* forem iguais, significa que as senhas conferem.

Vamos na prática?

Crie uma nova pasta com o nome **TesteMD5** no VS Code. Abrindo o terminal, dentro da pasta, vamos criar uma aplicação de console usando o tradicional comando “dotnet new console”. Na figura 12, temos a tela com o código do famoso “Hello World”.

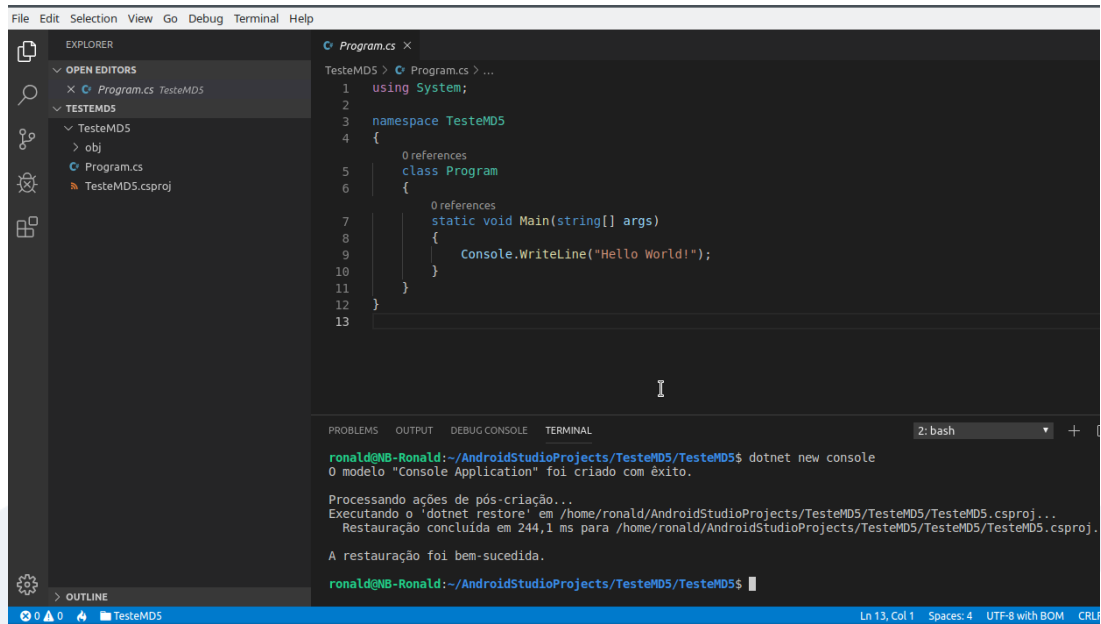


Figura 15 - Tela inicial do programa TesteMD5

Agora, vamos substituir todo o código criado pelo VS Code por este:

```
using System.Security.Cryptography;
```

```
using System.Text;
```

```
namespace TesteMD5
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

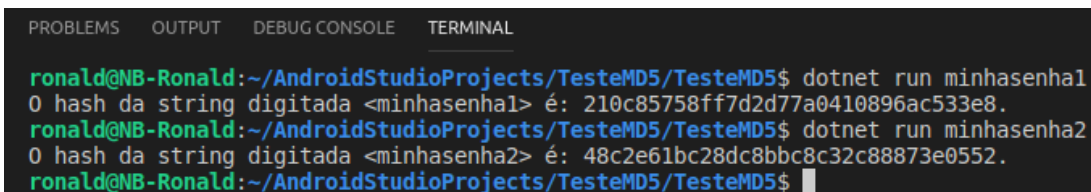
```
            string nome = args[0]; // Obtém o string digitado no console
```

```
            using (MD5 md5Hash = MD5.Create())
```

```
{  
  
    string hash = GetMd5Hash(md5Hash, nome);  
  
    Console.WriteLine("O hash da string digitada <" + nome + ">  
    é: " + hash + ".");  
  
}  
  
}  
  
static string GetMd5Hash(MD5 md5Hash, string input)  
{  
    // Converte a string de entrada em um array de bytes e calcula  
    o hash  
  
    byte[] dado =  
    md5Hash.ComputeHash(Encoding.UTF8.GetBytes(input));  
  
    // Cria um construtor de string para passar os bytes gerados  
  
    StringBuilder sBuilder = new StringBuilder();  
  
    // Converte cada byte do hash em uma string em hexadecimal  
  
    for (int i = 0; i < dado.Length; i++)  
  
    {  
  
        sBuilder.Append(dado[i].ToString("x2"));  
  
    }  
}
```

```
// Retorna a string já em hexadecimal  
  
return sBuilder.ToString();  
  
}  
  
}  
  
}
```

Uma boa dica para indentar o código a qualquer momento é a combinação de teclas **Shift + Alt + f** (no Linux, é **Ctrl + Shift + i**). Para rodar o programa, teremos de digitar a palavra da qual se quer conhecer o *hash* na própria linha de comando, por exemplo: `dotnet run minhasenha`. Veja na figura 13 a grande diferença nos *hashes* utilizando “minhasenha1” e “minhasenha2”. Perceba que a diferença entre estas duas *strings* é de apenas 1 *bit*.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
ronald@NB-Ronald:~/AndroidStudioProjects/TesteMD5/TesteMD5$ dotnet run minhasenha1  
O hash da string digitada <minhasenha1> é: 210c85758ff7d2d77a0410896ac533e8.  
ronald@NB-Ronald:~/AndroidStudioProjects/TesteMD5/TesteMD5$ dotnet run minhasenha2  
O hash da string digitada <minhasenha2> é: 48c2e61bc28dc8bbc8c32c88873e0552.  
ronald@NB-Ronald:~/AndroidStudioProjects/TesteMD5/TesteMD5$
```

Figura 16 – Programa que gera *hashes*

Vamos ver os principais métodos utilizados aqui. Na linha 1, estamos incluindo a biblioteca de criptografia: `using System.Security.Cryptography`. Na linha 9, vamos ler o texto digitado na linha de comando e salvar na variável do tipo *string* chamada “nome”: `string nome = args[0]`. Na linha 10, vamos criar um objeto, `md5Hash`, do tipo MD5, para calcular o *hash*. Na linha 16, temos o método que faz esta conversão: `static string GetMd5Hash (MD5 md5Hash, string input)`. Na linha 19, o cálculo do *hash* é realizado tendo o seu resultado binário salvo no *array* de *bytes* **dado**:


```
byte[] data =  
md5Hash.ComputeHash(Encoding.UTF8.GetBytes(input)).
```

As próximas instruções, a partir da linha 22, criam um *string sBuilder*, que vai receber o *array dado* com os números binários provenientes do cálculo já realizado e transformá-los em uma cadeia de 32 caracteres em hexadecimal, que é o padrão adotado em todo o mundo. Esta transformação ocorre no *loop for*.

O algoritmo MD5 já foi muito utilizado, mas após descobertas de falhas de desenvolvimento graves em 2004, ele não é mais considerado seguro. A recomendação atual é utilizar o algoritmo SHA-2, que usa números primos de 256, 384 e até 512 *bits* (chamados respectivamente de SHA-256, SHA-384 e SHA-512) para criar os *hashes*.

Existem outros algoritmos para outras funções que não foram abordados aqui, mas já iniciamos o estudo com o uso do MD5 em razão de sua simplicidade e porque ele serve para demonstrar o conceito inicial de segurança da informação.