Capítulo 9
Utilizando Code Access Security – CAS

Introdução

Toda aplicação que utiliza o *Common Language Runtime* (CLR) obrigatoriamente deve interagir com o sistema de segurança do mesmo. Quando a aplicação é executada, automaticamente é avaliado se ela tem ou não determinados privilégios. Dependendo das permissões que a aplicação tem, ela poderá rodar perfeitamente ou gerar erros relacionados a segurança.

Como estamos diante de um ambiente cada vez mais conectado, é muito comum expor o código de diferentes formas e diferentes locais. Muitos mecanismos de segurança concedem direitos de acesso aos recursos (arquivos e pastas por exemplo) baseando-se nas credenciais (nome de usuário e *password*) do usuário. Só que isso acaba sendo uma técnica perigosa, já que os usuários podem obter o código de diversos locais, inclusive locais desconhecidos, que podem expor códigos maliciosos, códigos que contém vulnerabilidades, etc..

Code Access Security (também conhecido como CAS), é um mecanismo que ajuda limitar e proteger o acesso que o código que está querendo realizar, protegendo os recursos e operações.

Este capítulo abordará, em sua primeira parte, como utilizar o CAS, que é fornecido juntamente com o .NET Framework e, como configurar devidamente a aplicação para evitar problemas relacionados a segurança. Já a segunda parte, analisaremos a segunraça baseada em *roles*.

Conceitos básicos – *Code Access Security*

Toda aplicação que é executado sob a plataforma .NET interagi com o sistema de segurança – *Code Access Security*. De acordo com as permissões que ele recebe, ele pode executar de forma legal ou não, o que gerará uma exceção de segurança.

As configurações de segurança local em um computador particular é o que vai decidir, em última instância, que permissões o código irá receber. Como esses configurações podem varias de computador para computador você deve assegurar que seu código terá as permissões suficientes para ser executado. Sendo assim, todo o desenvolvedor deve estar familiarizado com os seguintes conceitos de segurança, que são necessários para todas as aplicações escritas em utilizando a plataforma .NET:

- **Escrever código *type-safe*:** para habilitar o benefício da utilização do *Code Access Security* você deve utilizar um compilador que gere o código *verifiably type-safe*.

- **Sintaxe imperativa e declarativa:** são as duas formas que temos para interagir com o *Code Access Security*. Com a primeira delas, utilizamos as classes de forma programática, ou seja, como já fazemos na maioria das vezes com outros objetos; a segunda forma, declarativa, permite decorarmos os tipos com atributos que definem a forma de segunraça a ser aplicada ao membro.
- **Requerimento de permissões:** é a forma que o teu código informa ao *runtime* as permissões que ele precisa para poder ser executada. Esses requerimentos são analisados pelo *runtime* durante a carga do código para a memória. Esse tipo de permissão é aplicado a nível de *Assembly*.
- ***Secure Class Libraries*:** uma biblioteca segura é uma biblioteca que utiliza a segurança sob demanda para assegurar que o chamador tem ou não permissão para acessar um determinado recurso.

Como já vimos, o *Code Access Security* são as permissões que concedemos ao código para que o mesmo pode ser executado. Mas a segurança não se resume somente a isso; existe uma porção de conceitos que precisamos analisar detalhadamente para, em seguida, aplicar efetivamente em nosso código. Cada um desses conceitos são analisados a seguir.

Evidence – Evidência

A evidência é o conjunto de informações sobre um determinado *Assembly*. Entre essas informações temos a identidade e origem do *Assembly*. O *Code Access Security* utiliza essa evidência do *Assembly* e a política de segurança corrente do computador onde o mesmo está sendo executado, para determinar se ele possui ou não permissões suficientes para acessar um determinado recurso.

Toda aplicação .NET roda em um *AppDomain*, sob o controle do *host* que cria o *AppDomain* e carrega os *Assemblies* para dentro do mesmo. O *host* tem acesso a evidência do(s) *Assembly(ies)* que está(ão) dentro deste *AppDomain*. Atualmente temos dois tipos principais de evidências: a nível de *Host* e a nível de *Assembly*. Por padrão, o .NET Framework utiliza somente a evidência a nível de *host*, que é a informação fornecida a partir do *AppDomain* onde a aplicação é executada. Tipicamente, a evidência do host vai informar a origem do *Assembly* e se ele está devidamente assinado (*strong-name*). Já a evidência a nível de *Assembly* é fornecida a partir do próprio *Assembly*, e pode ser definida a partir dos desenvolvedores ou administradores.

A evidência de um *Assembly* poderá incluir:

Evidência	Descrição	Condição (classe)
All Code		AllMembershipCondition
Diretório da Aplicação	O local físico onde a aplicação foi instalada.	ApplicationDirectoryMembershipCondition
Hash	Um código *hash* que é utilizado para algoritmos	HashMembershipCondition

	de *hashing*.	
Publicador	Assinatura do publicador do *Assembly*.	PublisherMembershipCondition
Site	Site de origem do *Assembly*.	SiteMembershipCondition
StrongName	Uma chave criptográfica, contendo o *StrongName* do *Assembly*.	StrongNameMembershipCondition
URL	URL de origem do *Assembly*.	UrlMembershipCondition
Zona	Zona de origem do *Assembly*, como por exemplo a *Internet*.	ZoneMembershipCondition

Para cada um dos itens acima, existe um classe que corresponde à uma condição, utilizada em um *code group*, que analisaremos mais tarde ainda neste capítulo. Essas classes estão informardas na terceira coluna da tabela acima.

Para manipularmos isso via código, temos uma classe chamada **Evidence** dentro do *namespace* **System.Security.Policy**. Essa classe nada mais é que uma coleção que armazena um conjunto de objetos que representam as evidências (*Host* ou *Assembly*).

O trecho de código abaixo exibe a forma de como devemos proceder para extrairmos as informações de evidência de um *Assembly*:

```
VB.NET
Imports System
Imports System.Collections
Imports System.Reflection
Imports System.Security.Policy

Dim a As Assembly = _
    [Assembly].GetAssembly(Type.GetType("System.String"))
Dim e As Evidence = a.Evidence
Dim i As IEnumerator = e.GetEnumerator()
While i.MoveNext()
    Console.WriteLine(i.Current)
End While

C#
using System;
using System.Collections;
using System.Reflection;
using System.Security.Policy;

Assembly a = Assembly.GetAssembly(Type.GetType("System.String"));
```

```
Evidence e = a.Evidence;
IEnumerator i = e.GetEnumerator();
while(i.MoveNext())
    Console.WriteLine(i.Current);
```

Permissions – Permissões

As permissões, que aqui também são conhecidas como *code access permissions*, são os direitos de acesso a determinado recursos do computador.

O .NET Framework possui muitas classes embutidas que foram desenhadas para proteger o acesso a determinados recursos do computador onde a aplicação é executada. Isso auxilia bastante, já que não precisamos escrever código necessário para efetuarmos a checagem de segurança; essas classes já tem essa finalidade. Para o exemplo, temos algumas permissões (em forma de classes) listadas abaixo:

Permissão	**O que protege**
DataProtectionPermission	Controla o acesso a dados criptografados em memória.
EnvioronmentPermission	Controla o acesso a variáveis de ambiente.
EventLogPermission	Controla o acesso ao *Event Log* do Windows.
FileIOPermission	Controla o acesso ao sistema do arquivos.
PrintingPermission	Controle o acesso as impressoras.
RegistryPermission	Controla o acesso ao *Registry* do Windows.
SqlClientPermission	Controla o acesso ao banco de dados SQL Server a partir do *provider*, também fornecido pela plataforma.
StorePermission	Controla o acesso aos repositórios de certificados X.509.
UIPermission	Controla o acesso a criação de elementos Windows.

Essas classes estão contidas dentro do *namespace* **System.Security.Permissions** e, grande parte dessas classes, herdam direta ou indiretamente de uma classe abstrata chamada **CodeAccessPermission**, que define toda a estrutura para as permissões.

Security Policy – Políticas de Segurança

As políticas de segurança determinam o mapeamento entre a evidência do *Assembly* que o *host* fornece para o mesmo e o conjunto de permissões concedidas ao *Assembly*.

As políticas de segurança estão divididas em quatro níveis, quais estão abaixo descritos:

Nível de Segurança	**Descrição**
Enterprise	Especificada pelo administrador da rede. Contém a hierarquia de *code groups* que serão aplicados para todos os códigos gerenciados que serão executados dentro da rede.

Machine	Contém a hierarquia de *code groups* que serão aplicados para todos os códigos gerenciados que serão executados dentro de um determinado computador.
User	Contém a hierarquia de *code groups* que serão aplicados para todos os códigos gerenciados que serão executados dentro de um usuário.
Application Domain (opcional)	Este nível de segurança é opcional é fornece isolamento e limites de segurança para o código gerenciado que está sendo executado.

A permissão final concedida é definida uma por *Assembly* e, sendo assim, cada *Assembly* dentro da aplicação pode ter diferentes permissões. Finalmente, se desejarmos manipular as configurações de políticas de segurança via código, podemos utilizar a classe **SecurityManager**, que está contida dentro do *namespace* **System.Security**, que possui vários membros estáticos que permitem interagir com o sistema de segurança.

Através desta classe, utilizamos o método *PolicyHierarchy* que retorna um enumerador com os níveis em que as políticas de segurança se encontram:

VB.NET

```
Imports System
Imports System.Collections
Imports System.Security
Imports System.Security.Policy

Dim i As IEnumerator = SecurityManager.PolicyHierarchy()
While i.MoveNext()
    Dim p As PolicyLevel = DirectCast(i.Current, PolicyLevel)
    Console.WriteLine(p.Label)
End While
```

C#

```
using System;
using System.Collections;
using System.Security;
using System.Security.Policy;

IEnumerator i = SecurityManager.PolicyHierarchy();
while (i.MoveNext())
{
    PolicyLevel p = (PolicyLevel)i.Current;
    Console.WriteLine(p.Label);
}
```


Permission Sets – Conjunto de Permissões

Todas os níveis de segurança contém uma lista de conjunto de permissões. Cada um desses conjuntos representam uma espécie de conjunto de acesso confiável a determinados recursos do computador.

Para exemplificar, abaixo temos uma tabela com os conjuntos de permissões pré-definidos pelo .NET Framework:

Permission Set	**Descrição**
FullTrust	Fornece acesso a todos os recursos protegidos por permissões.
SkipVerification	Permite que a checagem de segurança não seja realizada.
Execution	Fornece permissão apenas para o código ser executado, não permitindo acesso à qualquer recurso protegido.
Nothing	Não fornece nenhum tipo de permissão, impedindo-o de ser executado.
LocalIntranet	Permite acesso para execução do código, criação de elementos a nível de *interface* sem qualquer restrição, acesso ao *isolated storage* sem limite de quota, utilizar serviços de DNS, ler algumas variáveis de ambiente, realizar conexões com a site de onde o *Assembly* se originou e ler arquivos que estão dentro da mesma pasta do *Assembly*.
Internet	Permite acesso para execução do código, criar janelas e caixas de diálogos, realizar conexões com a site de onde o *Assembly* se originou e acessar o *isolated storage* com quota.
Everything	Fornece todas as permissões padrões, exceto as permissões para a opção *SkipVerification*.

O trecho de código abaixo utiliza a classe **SecurityManager** para recuperar todos os níveis de políticas de segurança (*enterprise*, *machine* e *user*) e seus respectivos conjuntos de permissões da máquina em que o código é executado:

```
VB.NET
Imports System
Imports System.Collections
Imports System.Security
Imports System.Security.Policy

Dim i As IEnumerator = SecurityManager.PolicyHierarchy()
While i.MoveNext()
    Dim p As PolicyLevel = DirectCast(i.Current, PolicyLevel)
    Console.WriteLine(p.Label)
    Dim np As IEnumerator = p.NamedPermissionSets.GetEnumerator()
    While np.MoveNext()
        Dim pset As NamedPermissionSet = _
            DirectCast(np.Current, NamedPermissionSet)
        Console.WriteLine("\tPermission    Set:    \n\t\t    Name:
{0}\n\t\t Description: {1}", pset.Name, pset.Description)
    End While
```

```
End While

C#
using System;
using System.Collections;
using System.Security;
using System.Security.Policy;

IEnumerator i = SecurityManager.PolicyHierarchy();
while (i.MoveNext())
{
    PolicyLevel p = (PolicyLevel)i.Current;
    Console.WriteLine(p.Label);
    IEnumerator np = p.NamedPermissionSets.GetEnumerator();
    while (np.MoveNext())
    {
        NamedPermissionSet pset = (NamedPermissionSet)np.Current;
        Console.WriteLine("\tPermission    Set:    \n\t\t    Name:
{0}\n\t\t Description: {1}", pset.Name, pset.Description);
    }
}
```

Code Groups – Grupos de códigos

Os *code groups* são o coração do sistema de segurança do .NET Framework. Ele consiste em uma expressão condicional que, se for atendida, concede um determinado conjunto de permissões (*permission set*) que estão associadas ao *code group*. Em tempo de execução, a condição é avaliada comparando as informações do *code group* com o a evidência que foi extraída do *Assembly*.

Para exemplificar, podemos dizer que ele somente terá acesso ao sistema de arquivos, se o publicador do *Assembly* for a Empresa "ABC".

Os *code groups* são armazenados em um formato de uma "árvore" lógica e, se a condição "A" não for atendida, as condições abaixo dela não serão analisadas e, conseqüentemente, o código não terá acesso as permissões que a mesma poderia vir a conceder. A imagem abaixo ilustra de forma bem clara a hierarquia dos *code groups* e como eles são avaliados em tempo de execução:

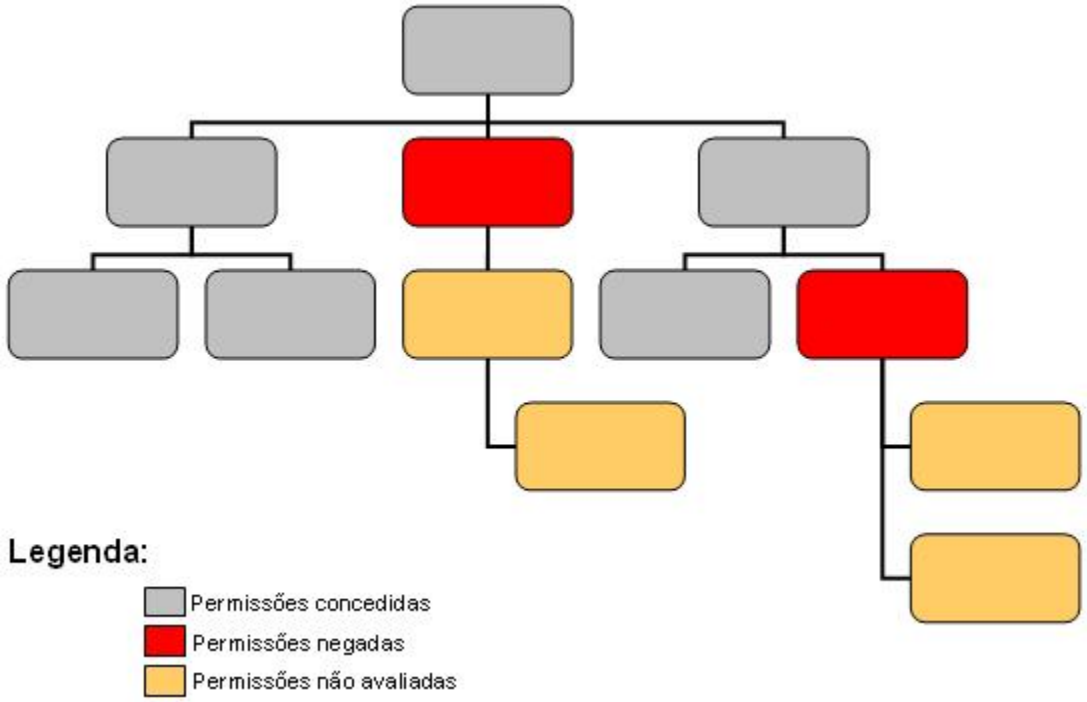


Imagem 9.1 – Avaliando as condições.

Como podemos analisar, quando um determinado *code group* não atende a um determinado critério (quadrados na cor vermelha), as permissões relacionadas a ele são negadas e, conseqüentemente, o que vem abaixo (quadrados na cor laranja) não é avaliado. Esse processo é repetido para todos os níveis das políticas de segurança (*enterprise*, *machine* e *user*).

Em tempo de execução, essas condições são transformadas em classes do tipo *xxxMembershipCondition* que vimos um pouco mais acima. Essas classes implementam a *Interface* **IMembershipCondition** que define um teste para determinar se o código do *Assembly* é membro de um determinado *code group*.

Stack Walk

Uma das partes essencias do sistema de segurança é o processo que chamamos de *stack walk*. Quando um determinado método é chamado, os dados referente ao mesmo (parâmetros que são passados para o método, o endereço de retorno quando o método retornar e variáveis locais) são colocados em uma espécie de pilha de chamadas, ***call stack***. Cada um desses "registros" são também chamados de ***stack frame***.

Em determinados estágios da execução do código, a *thread* que está executando pode precisar acessar um recurso protegido, como por exemplo, o sistema de arquivos. Antes de efetivamente conceder acesso a esse determinado recurso, sob demanda, uma verificação é efetuada em toda a *call stack*, analisando se todos os chamadores possuem

direitos ao recurso solicitado. Neste momento, se algum dos chamadores não tiver a permissão necessária, uma exceção é atirada. Esse processo é chamado de ***stack walk***.

Para fazer uma analogia em um mundo real, imagine que há uma pessoa que deseja alugar um livro em uma biblioteca mas ela não tem um cadastro na mesma. Imagine que essa pessoa pedi um favor para alguém que tenha esse cadastro, para que ele possa pegar o livro na biblioteca e, em seguida, o emprestar. Como o bibliotecário nada sabe sobre o que se passa, ele analisará o cadastro da pessoa que for diretamente até a biblioteca e, estando com o cadastro correto, alugará o livro sem maiores problemas. Esse processo é mostrado através da imagem abaixo:

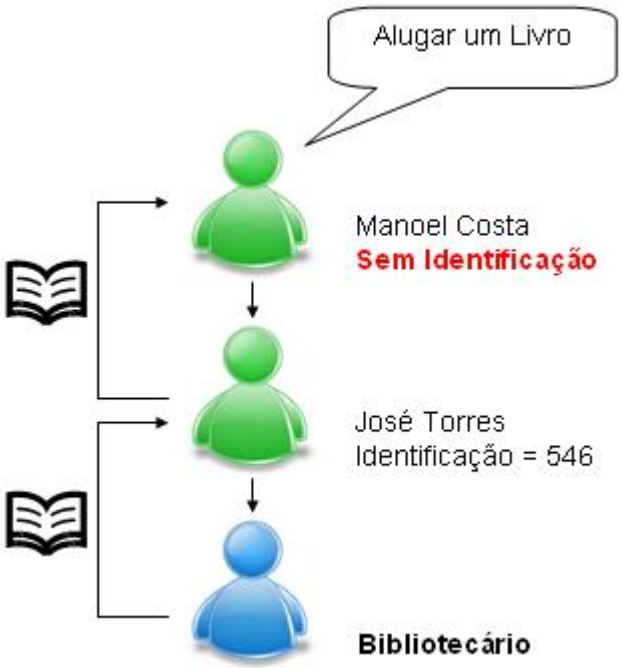


Imagem 9.2 – *Luring Attack*.

Apesar dessa forma ser executada sem maiores problemas, isso não traz uma segurança para a biblioteca. Trazendo para o mundo de desenvolvimento de software, entendemos isso como um ataque, chamado de *Luring Attack*. O *luring attack* é um tipo de ataque que eleva os privilégios de quem o executa, concedendo mais privilégios do que realmente ele possui.

Para evitar o *luring attack*, o .NET é capaz de analisar toda a cadeia de chamadores e analisar se todos eles possuem ou não os direitos necessários quando algum recurso protegido é requisitado. Ainda dentro do nosso exemplo, quando o *José Torres* chegar ao bibliotecário para alugar o livro, o bibliotecário irá analisar todos os chamadores que fazem parte do processo e, irá identificar que o *Manoel Costa* não tem cadastro na biblioteca, o que banirá o aluguel do livro. A imagem abaixo ilustra esse processo:

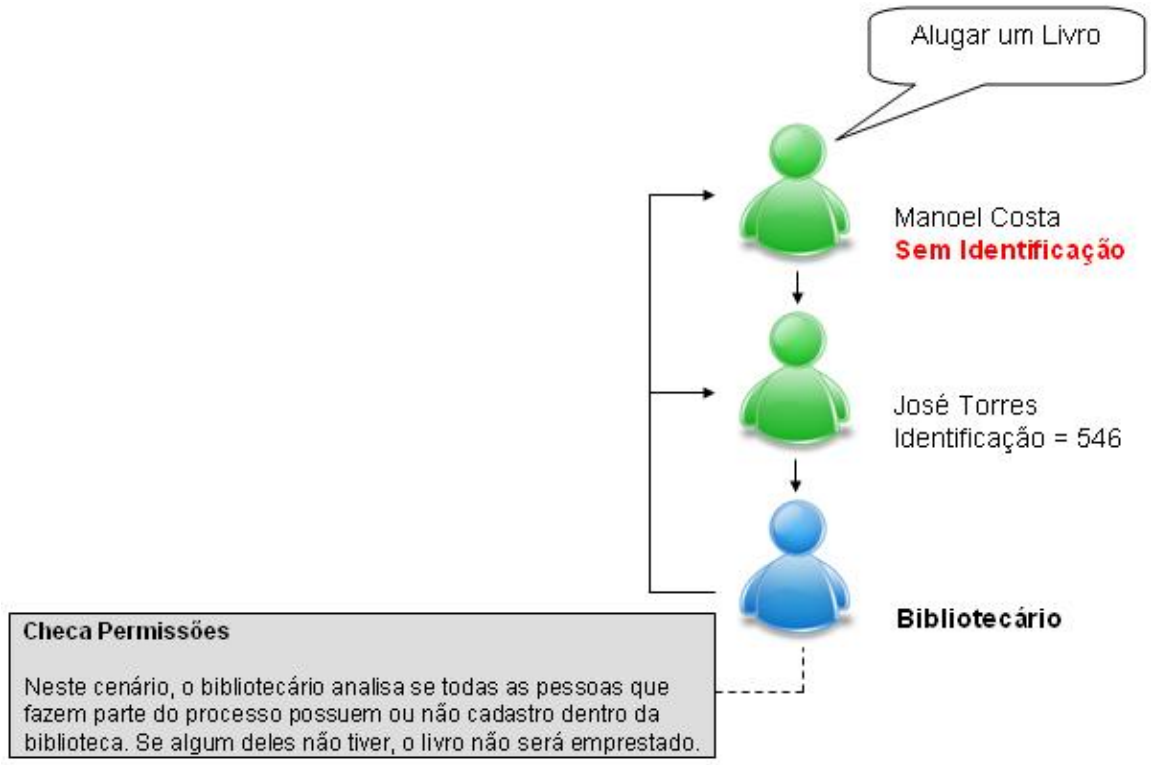
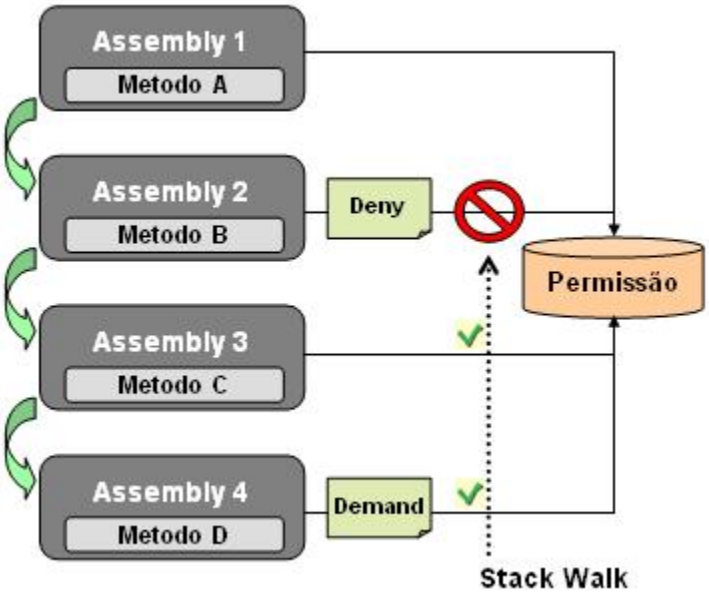


Imagem 9.3 – Evitando o *Luring Attack*.

Como vimos anteriormente, temos classes que são nomeadas com um sufixo *Permission*. Essas classes herdam da classe abstrata **CodeAccessPermission** e protegem determinados recursos de um computador. Essa classe abstrata fornece quatro principais métodos (de instância) que, são utilizados para determinar se o código possui ou não acesso ao recurso que a instância representa. Esses métodos são listados e descritos através da tabela abaixo:

Método	**Descrição**
Assert	Concede acesso ao recurso protegido pela instância mesmo que os chamadores não tenham permissão para isso. Esse método pode ser utilizado quando a sua biblioteca necessita acessar um recurso protegido de forma completamente oculta aos chamadores. De qualquer forma, utilize esse método com muito cuidado, porque ele pode deixar a sua aplicação/biblioteca vulnerável a ataques.
Demand	Ao chamar esse método, a checagem é realizada em toda a stack (de cima para baixo), forçando uma exceção do tipo **SecurityException** ser atirada, se algum dos chamadores dentro da *stack* não tiver a permissão para o recurso protegido pela instância. Esse método é geralmente utilizado por bibliotecas que precisam assegurar que o chamador tem acesso a um recurso protegido.

Deny	Previne os chamadores dentro da *stack* de acessar o recurso protegido pela instância, mesmo que eles tenham permissão para isso. A imagem abaixo ilustra o processo da utilização deste método:  **Imagem 9.4 –** Utilização do método *Deny*. Como sabemos, o método Demand executa a checagem dentro da stack para verificar se todos os chamadores possuem a permissão necessária para acessar o recurso. Na imagem acima, quando o *Método B* é avaliado, vemos que dentro dele, o método *Deny* foi chamado, o que evita que a permissão seja concedida. Note que, ao encontrar a chamada para o método *Deny*, o *Método A* não chega a ser avaliado, pois independente do resultado, a permissão não será concedida.
PermitOnly	Em essência, o método *PermitOnly* tem o mesmo efeito do método *Deny*, mas define uma condição diferente de como a segurança deve ser analisada para conceder ou negar acesso à um determinado recurso. Ao invés de dizer que um recurso específico não pode ser acessado (é o que o método *Deny* faz), o método *PermitOnly* informa somente os recursos que você quer conceder acesso. Se você chamar o método *PermitOnly* em uma permissão *X*, é o mesmo que chamar que chamar o método *Deny* para todas as permissões, com exceção da permissão *X*.

Além desses métodos, a classe **CodeAccessPermission** ainda fornece quatro métodos estáticos que são utilizados para reverter alguma das "ordens" acima, que foram aplicadas pelos métodos *Assert*, *Deny* ou *PermitOnly*, fazendo com que essas "ordens" sejam desfeitas. A tabela abaixo descreve cada um desses métodos:

Método	Descrição
RevertAll	Esse método desfaz todos as "ordens" realizadas pelos métodos *Assert*, *Deny* ou *PermitOnly* e, se nenhum desses métodos foi previamente invocado, uma exceção do tipo **ExecutionEngineException** será atirada.
RevertAssert	Remove qualquer instrução gerada pelo método *Assert* dentro de um mesmo *frame*. Se o método *Assert* não foi previamente invocado, uma exceção **ExecutionEngineException** do tipo será atirada.
RevertDeny	Remove qualquer instrução gerada pelo método *Deny* dentro de um mesmo *frame*. Se o método *Deny* não foi previamente invocado, uma exceção **ExecutionEngineException** do tipo será atirada.
RevertPermitOnly	Remove qualquer instrução gerada pelo método *PermitOnly* dentro de um mesmo *frame*. Se o método *PermitOnly* não foi previamente invocado, uma exceção **ExecutionEngineException** do tipo será atirada.

Você deve utilizar esses métodos com extremo cuidado porque eles modificam a forma com que o *Code Access Security* avalia a *stack walk*, o que pode possibilitar que sua aplicação sofra com ataques do tipo *luring attack*, como vimos nas imagens acima.

Geralmente, a checagem de segurança examina todos os chamadores que estão dentro da *stack* para assegurar que cada um deles possuem as devidas permissões para acessar o recurso protegido que está sendo solicitado. Entretanto, através dos métodos acima podemos sobrescrever esse comportamento e, conseqüentemente, customizar a forma com que o *Code Access Security* concede ou nega o acesso à um determinado recurso. Esse processo é conhecido como *"Overriding Security Checks"*.

Toda vez que um método chama outro, um novo *frame* é gerado dentro da *stack* para armazenar informações a respeito do método que está sendo invocado. Cada um desses *frames* contém informações sobre a chamada de qualquer um dos métodos *Assert*, *Deny* ou *PermitOnly* e, se o chamador utiliza mais que um desses métodos dentro do mesmo local (método), o *runtime* aplica as seguintes regras:

- Se, durante a checagem da *stack walk*, o *runtime* descobrir mais que uma chamada para um mesmo método (*Assert*, *Deny* ou *PermitOnly*) dentro do mesmo *frame*, o segundo causará uma exceção.
- Quando houver chamadas diferentes aos métodos *Assert*, *Deny* ou *PermitOnly* dentro do mesmo *frame*, o runtime os processará na seguinte ordem: *PermitOnly*, *Deny* e, finalmente, o *Assert*.

E como vimos na tabela dos métodos *Revert****, você utiliza-os quando desejar reverter qualquer uma das operações previamente executadas.

Ferramentas

Com exceção das classes que o .NET Framework fornece para customizarmos via código o que a nossa aplicação necessita para poder funcionar, ainda há duas ferramentas úteis, quais foram disponibilizadas com a instalaçã do .NET Framework, que permitem interagirmos com o sistema de segurança, modificando as políticas de segurança da máquina, do usuário ou da rede.

A primeira delas, é chamada de **.NET Framework 2.0 Configuration**. Essa ferramenta permite-nos configurarmos não somente a parte de segurança, mas o GAC e outras coisas que estão fora do escopo deste capítulo. Se reparar na imagem abaixo, temos uma opção chamada ***Runtime Security Policy***, onde podemos customizar os *code groups* e *permissions sets* para os níveis de segurança existentes. A imagem abaixo ilustra a *interface* desta ferramenta:

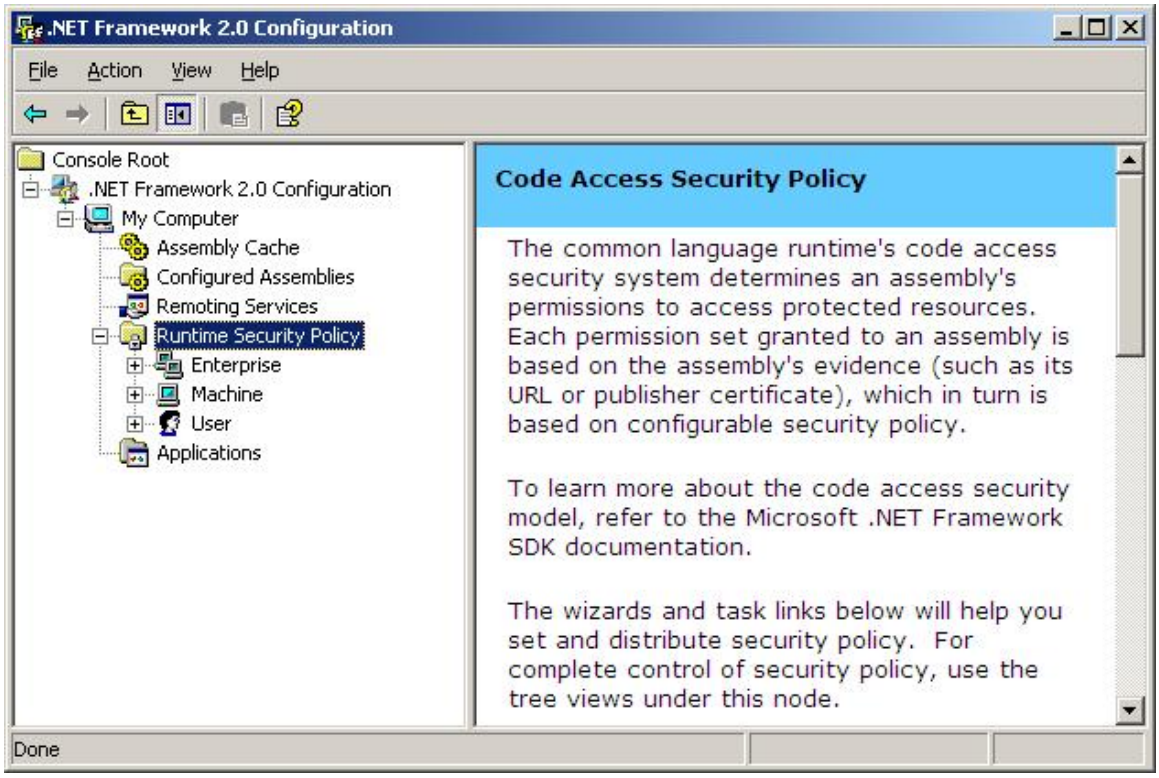


Imagem 9.5 - .NET Framework 2.0 Configuration.

Para acessar essa ferramenta, vá até as *Ferramentas Administrativas* que está dentro do *Painel de Controle* do Windows e clique em *Microsoft .NET Framework 2.0 Configuration*.

Além desta ferramenta gráfica, ainda existe uma outra chamada **CasPol.exe**. Trata-se de um utilitário de linha de comando que permite você interagir com o sistema de segurança do .NET Framework e, basicamente, fornece as mesmas funcionalidades que a *interface* acima.

Esse utilitário é disponibilizado junto com o .NET Framework e encontra-se localizado no seguinte endereço: **C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727**. Como esses utilitários são executados a partir de linha de comando e você optar por abrir o *Visual Studio 2005 Command Prompt*, não será necessário digitar o caminho todo até o executável para executá-lo. A sintaxe para a sua utilização é simples:

```
C:\caspol <opções> <argumentos>
```

Entre as opções que são aceitas, temos algumas delas listadas através da tabela abaixo:

Opção	Descrição
-l	Lista todas as informações disponíveis para a *policy level* padrão, que é o nível *Machine*.
-lg	Lista todos os *code groups* para a *policy level* padrão.
-lp	Lista todas as *permissions sets*.
-ag	Adiciona um novo *code group* na *policy level* padrão.
-url	Define a URL para um endereço HTTP ou FTP, indicando onde o *Assembly* se originou. Isso será adicionado em forma de uma condição (**UrlMembershipCondition**).
-n	O nome para o novo *code group*.
-exclusive	Definindo esta opção como "on" indica que qualquer *Assembly* que atender a condição definida pelo *code group* que você está criando, será associado com a *permission set* para esta *policy level*. Isso pode ser utilizado para nível de testes, pois irá garantir que o código que está rodando não recebe a permissão de ***FullTrust***.
-cg	Altera um *code group* existente.
-rg	Remove um *code group* existente.

Abaixo é exibido algumas possíveis formas de combinar essas opções e argumentos para a utilização deste utilitário:

```
C:\caspol -l

C:\caspol -lg

C:\caspol -ag 1 -url file:///C:/Teste/* Internet -n
Grupo_De_Teste -exclusive on

C:\caspol -rg Grupo_De_Teste
```

Com exceção da lista acima, temos outras opções e argumentos que podemos informar par ao utilitário **caspol.exe**. Para ter acessa a todas elas, vá até o *MSDN Library* ou, ainda no *prompt* de comando, digite:

```
C:\caspol -?
```

Avaliando as permissões

Quando a aplicação é inicializada, a mesma é carregada para dentro de um processo, que é chamado também de *host*. Esse *host* extrai e examina a evidência do *Assembly*, podendo diferentes informações serem coletadas de acordo com a origem do mesmo.

A evidência consiste, entre outras inforamções, o *strong-name*, zona e publicador do *Assembly*. Depois de capturada, essa evidência é passada para o sistema de segurança do .NET Framework para que o mesmo avalie as políticas de seguranças.

A partir deste momento, baseando-se na evidência extraída do *Assembly*, o *runtime* começará a avaliar a "árvore" de *code groups*. Se a condição for atendida, dizemos que o *Assembly* é membro do *code group* e, além disso, o conjunto de permissões (*permission set*) vinculadas a essa condição será concedido ao *Assembly*. Caso contrário, possíveis *code groups* que estiverem abaixo da condição não atendida, não serão avaliados e, obviamente, não serão concedidos as possíveis permissões que ele poderia vir definir. Depois deste processo, a **união** dos conjuntos de permissões é computada e esse processo é repetido para cada nível das políticas de segurança (*policy levels*).

Depois de todos os níveis avaliados (*Enterprise*, *Machine*, *User* e *AppDomain*), o *Assembly* receberá a **interseção** de todos os grupos de permissões entre os níveis. A imagem abaixo ilustra esse todo esse processo que, a primeira vista, parece ser complicado:

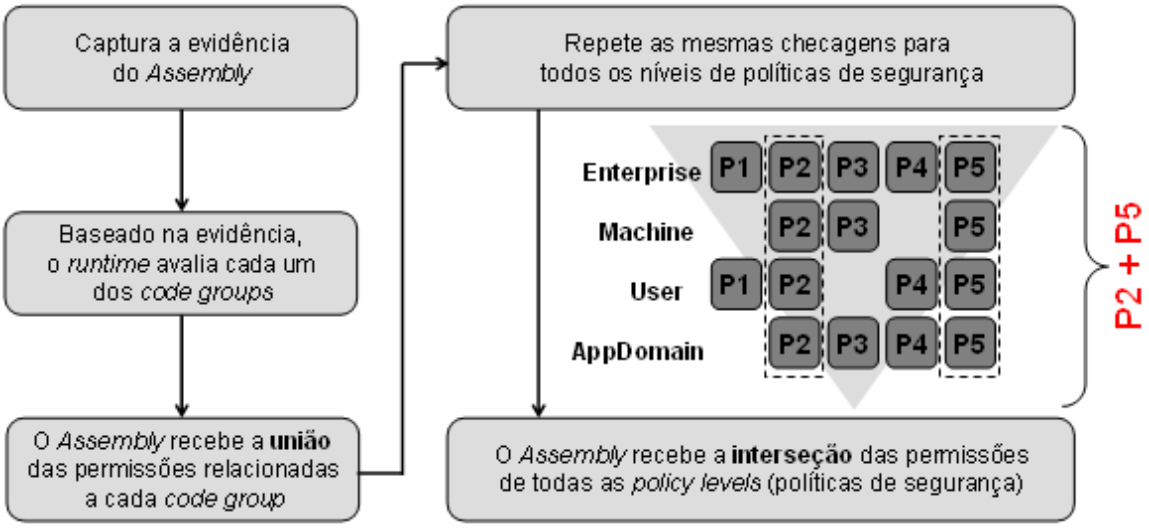


Imagem 9.6 – Avaliando as permissões de um *Assembly*

Como podemos notar, os níveis *Enterprise*, *Machine*, *User* e *AppDomain* são avaliados e somente são concedidos as permissões que estão contidas em todos os níveis que, no exemplo acima, são ***P2*** e ***P5***. Isto evitará que um usuário individual ou a aplicação conceda permissões adicionais que não foram concedidas pelo administrador (*Enterprise*).

Ainda é possível adicionar atributos a nível de *Assembly* que permite especificarmos as solicitações de permissões necessárias, mínimas e opcionais que o *Assembly* necessita para poder trabalhar. Abaixo veremos cada uma dessas opções:

- **Permissões Mínimas:** especificada a partir do atributo **RequestPermission**, permite especificar as permissões mínimas que são necessárias para que o *Assembly* para executar o seu trabalho. Se essas permissões não forem concedidas, ao carregar o *Assembly* o código não será executado e uma exceção do tipo **PolicyException** será atirada.
- **Permissões Opcionais:** especifica as permissões que seu código pode utilizar para executar mas, se não for concedida, o mesmo será capaz de executar. Neste caso, é extremamente importante que você utilize o tratamento de erros ou, de alguma outra forma, monitorar o trecho do código que exige essa permissão, pois, se não fizer isso, uma exceção pode ocorrer e danificar a sua aplicação.
- **Permissões Recusadas:** especifica as permissões que seu código jamais deverá receber, mesmo se as políticas de segurança as concedam.

O trecho de código abaixo exibe como configurar esses parâmetros dentro do arquivo *AssemblyInfo.vb* ou *AssemblyInfo.cs*:


```
VB.NET - (AssemblyInfo.vb)
Imports System.Security.Permissions
```

```
<Assembly:   EnvironmentPermission(SecurityAction.RequestMinimum,
Read := "windir")>
<Assembly: RegistryPermission(SecurityAction.RequestOptional)>
<Assembly: FileIOPermission(SecurityAction.RequestRefuse, Read :=
"C:\")>

C# - (AssemblyInfo.cs)
using System.Security.Permissions;

[assembly:   EnvironmentPermission(SecurityAction.RequestMinimum,
Read="windir")]
[assembly: RegistryPermission(SecurityAction.RequestOptional)]
[assembly:  FileIOPermission(SecurityAction.RequestRefuse,  Read  =
"C:\\")]
```

Como podemos notar no código acima, definimos que o *Assembly* precisa, no mínimo, de permissão suficiente para ler a variável de ambiente chamada *windir*; além disso, opcionalmente, pode ter acesso ao *registry* do Windows onde a aplicação estiver sendo executada e, finalmente, negamos o acesso a leitura da unidade *C* da máquina onde a aplicação estiver sendo executada.

Ao especificar a permissão, definimos se ela vai ser mínima, opcional ou recusada através de um enumerador chamado **SecurityAction**.

Para encerrar, depois de todas essas checagens, o *Assembly* recebe o que chamamos de **Final Permission Grant**. Essa permissão final é definida através do resultado da seguinte operação:

```
FG = SP ∩ ((M U O) – R)
```

Onde **FG** é a permissão final (*final grant*), **SP** é o grupo de permissões (*permission set*) que o *Assembly* recebe das políticas de segurança; depois disso, as permissões mínimas (**M**) unem-se as permissões opcionais (**O**), excluíndo as permissões recusadas (**R**). Com o resultado disso, é feito uma interseção com o grupo de permissões concedidos pelas políticas de segurança e, finalmente, atribuído a permissão final do *Assembly*.

Implementando a segurança no código

No código, o que precisamos fazer é instanciar a classe concreta da permissão, como por exemplo, a classe **FileIOPermission**, especificar os parâmetros necessários que a mesma exige e, através dos métodos que vimos acima (*Demand*, *Assert*, *Deny* ou *PermitOnly*) modificamos o *stack walk* e, conseqüentemente, customizamos o acesso ao recurso protegido.

Permissões Declarativas vs. Imperativas

Como vimos acima, há duas formas de aplicarmos a segurança em uma aplicação .NET. Essas formas são conhecidas como declarativa e imperativa. Abaixo há um exemplo de cada um destes estilos:

Forma Declarativa:

```
VB.NET
<FileIOPermission(SecurityAction.Assert, Read:="C:\")> _
Public Sub ReadFile()
     '...
End Sub

C#
[FileIOPermission(SecurityAction.Assert, Read:="C:\\")]
public void ReadFile()
{
     //...
}
```

Forma Imperativa:

```
VB.NET
Public Sub ReadFile()
     Dim f As New FileIOPermission(FileIOPermissionAccess.Read,
"C:\")
    f.Assert()
    '...
End Sub

C#
public void ReadFile()
{
     new            FileIOPermission(FileIOPermissionAccess.Read,
"C:\\").Assert();
    //...
}
```

Há algumas razões para escolher entre um estilo e outro. Um das grandes diferenças é que o estilo declarativo exige que todas as regras de segurança sejam definidas em tempo de compilação, permitindo apenas aplicar essas definições em métodos, classes ou *Assemblies*.

Já o estilo imperativo te dá uma maior flexibilidade ao estilo declarativo, pois permite que você manipule a segurança em tempo de execução, podendo assim, determinar o momento preciso de aplicar a checagem de segurança. Infelizmente, esse modo não permite extrairmos informações de metadados relacionadas a segurança. Essa ferramenta, chamada *Permission View* (**Permview.exe**), permite extrairmos essas informações de metadados somente a partir do estilo declarativo.

Esse utilitário é fornecido somente com a versão 1.x do .NET Framework. Para utilizá-lo, é necessário apenas informar caminho até o *Assembly* que deseja visualizar as informações. O código abaixo ilustra como devemos proceder para invocar esse utilitário:

```
C:\permview Aplicacao.exe
```

Segurança baseada em papéis (*roles*)

A segurança baseada em roles permite aos desenvolvedores controlarem o acesso das aplicações construídos sob a plataforma .NET baseando-se na identifidade do usuário. Neste caso, trabalhamos com dois conceitos chamados: *identity* e *principal*.

O primeiro deles, *identity*, encapsula informações a respeito da identificação do usuário que está sendo validado. Entre essas informações temos o nome do usuário e o tipo de autenticação. Basicamente, as *identities* são responsáveis pela autenticação do usuário. O .NET Framework fornece três tipos de objetos de identidade:

- **Windows Identity:** representa a identidade od usuário e o método de autenticação que é suportado pelo sistema operacional Windows. Além disso, esse tipo de identidade, fornece a possibilidade de personificarmos o usuário corrente para um outro usuário, talvez com mais ou menos privilégios para poder acessar um recurso protegido que, o usuário corrente talvez não tenha acesso. A classe que representa essa identidade é a **WindowsIdentity**.
- **Generic Identity:** representa a identidade do usuário baseando em uma autenticação customizada que é definida pela aplicação. A classe **GenericIdentity** implementa esse tipo de identidade.
- **Custom Identity:** representa uma identidade que encapsula as informações customizadas do usuário. Qualquer identidade customizada deve implementar a *Interface* **IIdentity** que, por sua vez, fornece três membros que são listados na tabela abaixo:

Membro	Descrição
Name	Retorna uma *string* contendo o nome corrente do usuário.
IsAuthenticated	Retorna um valor booleano indicando se o usuário está ou não autenticado.

AuthenticationType	Retorna o uma *string* contendo o tipo de autenticação do usuário corrente.

Já o principal representa o contexto de segurança em que o código está sendo executado. Basicamente, as *principals* são responsáveis pela autorização do usuário. O .NET Framework fornece três tipos de objetos relacionadas ao *principal*:

- **Windows Principal:** representa usuários do sistema operacional Windows e suas respectivas *roles*. Cada *role* representa um grupo que podem conter membros. A classe que representa essa *principal* é a **WindowsPrincipal**.
- **Generic Principal:** representa usuários e suas respectivas roles, mas de forma independente ao sistema operacional. A classe **GenericPrincipal** implementa este tipo de *principal*.
- **Custom Principal:** representa uma principal que encapsula a autorização de um determinado usuário. Qualquer *principal* customizada deve implementar a *Interface* **IPrincipal**.

A *Interface* **IPrincipal** possui apenas dois membros e, sendo assim, todas as classes relacionadas a *principal* também os possuem, já que implementam direta ou indiretamente a *Interface* **IPrincipal**. Os membros desta *Interface* são listados abaixo:

Membro	**Descrição**
Identity	Retorna um objeto que implementa a *Interface* **IIdentity** representando a identidade do usuário corrente.
IsInRole	Dado uma *string* contendo o nome da *role* (grupo), retorna um valor booleano indicando se o usuário corrente está ou não contido neste grupo.

Todas as Interfaces e classes que vimos acima estão contidas dentro do *namespace* **System.Security.Principal**. Abaixo é exibido uma imagem que ilustra as classes e *Interfaces* que vimos acima.

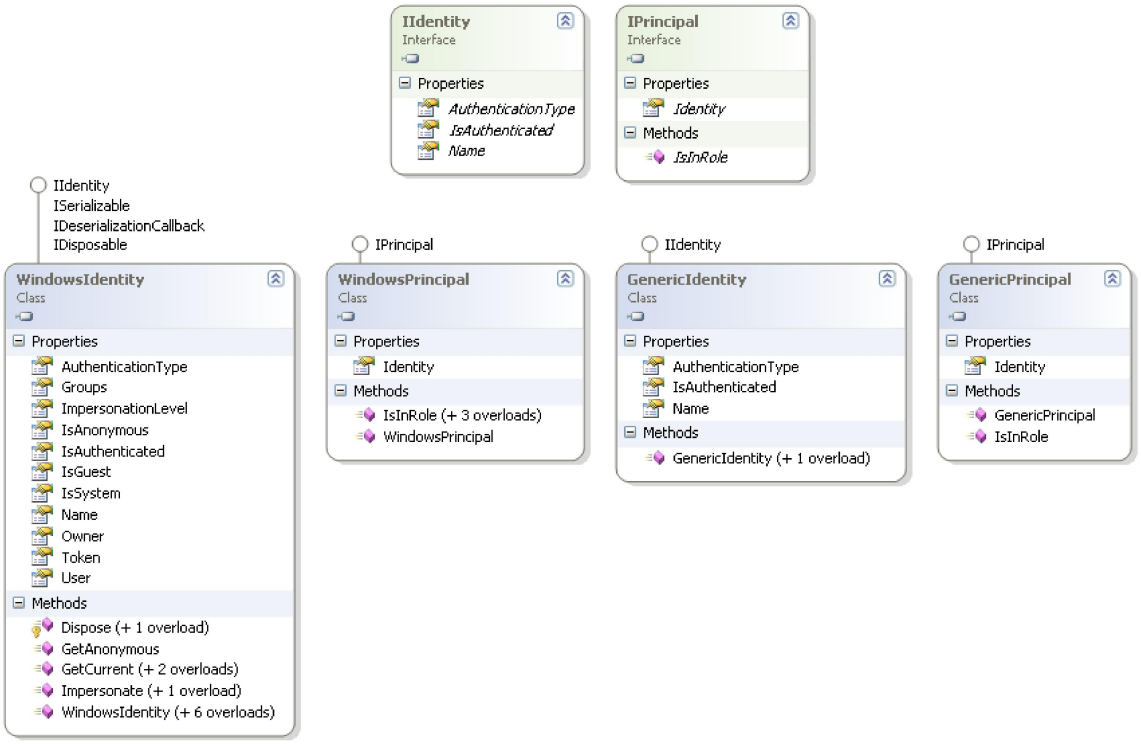


Imagem 9.7 – Estrutura das classes *identity* e *principal*.

Dentro do *namespace* **System.Threading** existe uma classe chamada **Thread**. Essa classe determina como controlar uma *thread* dentro da aplicação. Essa classe, entre vários membros, possui uma propriedade estática chamada *CurrentPrincipal* que recebe e retorna uma instância de um objeto que implementa a *Interface* **IPrincipal**. É através desta propriedade que devemos definir qual será a *identity* e *principal* que irá representar o contexto do segurança para a *thread* atual.

Sendo assim, temos que, de acordo com o tipo de autenticação/autorização que iremos adotar na aplicação, devemos criar a instância de uma *identiy* e *principal* e, em seguida, definí-la na propriedade *CurrentPrincipal* da classe **Thread**. Abaixo temos um exemplo utilizando as classes identity e principal relacionados ao sistema operacional Windows e também a forma genérica:

Windows

```
VB.NET
Imports System.Threading
Imports System.Security.Principal

Dim identity As WindowsIdentity = WindowsIdentity.GetCurrent()
Thread.CurrentPrincipal = New WindowsPrincipal(identity)

Console.WriteLine(Thread.CurrentPrincipal.Identity.Name)
```

```
Console.WriteLine(Thread.CurrentPrincipal.IsInRole("Admin").ToStr
ing())

C#
using System.Threading;
using System.Security.Principal;

WindowsIdentity identity = WindowsIdentity.GetCurrent();
Thread.CurrentPrincipal = new WindowsPrincipal(identity);

Console.WriteLine(Thread.CurrentPrincipal.Identity.Name);
Console.WriteLine(Thread.CurrentPrincipal.IsInRole("Admin").ToStr
ing());
```

Genérica

```
VB.NET
Imports System.Threading
Imports System.Security.Principal

Dim identity As New GenericIdentity("Jose")

Dim roles() As String = { "Admin", "RH", "Financeiro" }
Thread.CurrentPrincipal = New GenericPrincipal(identity, roles)

Console.WriteLine(Thread.CurrentPrincipal.Identity.Name)
Console.WriteLine(Thread.CurrentPrincipal.IsInRole("Admin").ToStr
ing())

C#
using System.Threading;
using System.Security.Principal;

GenericIdentity identity = new GenericIdentity("Jose");

string[] roles = new string[] { "Admin", "RH", "Financeiro" };
Thread.CurrentPrincipal = new GenericPrincipal(identity, roles);

Console.WriteLine(Thread.CurrentPrincipal.Identity.Name);
Console.WriteLine(Thread.CurrentPrincipal.IsInRole("Admin").ToStr
ing());
```

Utilizando a forma genérica, você pode customizar o repositório de onde você pode recuperar os dados de acesso e também os grupos que o usuário está contido e, utilizá-los na aplicação. Um exemplo é buscar os dados em um banco de dados e, se encontrado, criar os objetos *identity* e *principal* com os dados do usuário.

Personificação

Personificar é a habilidade que a *thread* possui para executar uma determinada tarefa em um contexto de segurança que é diferente do contexto que foi criado para o processo da própria *thread*. Uma das principais razões para utilizar a personificação é quando você precisa acessar um determinado recurso que, o usuário corrente não possui privilégios e, neste caso, personificamos o mesmo para que a tarefa seja executada através de um outro usuário, com maiores privilégios e, conseqüentemente, que tenha acesso ao recurso necessitado.

A classe **WindowsIdentity** fornece um método que permite personificarmos o usuário. Esse método chama-se *Impersonate* que, através da instância da classe **WindowsIdentity**, irá personificar para o usuário que está definido nela. Se bem sucedido, esse método retorna um objeto do tipo **WindowsImpersonationContext** que representa o usuário do Windows já no contexto personificado. Essa classe também fornece um método chamado *Undo* que devemos utilizar para reverter a personificação, voltando ao contexto do usuário original.

O exemplo abaixo mostra uma função que, via **PInvoke**, verifica se existe ou não o usuário "*Teste*" e, se existir, cria um objeto do tipo **WindowsIdentity**, passando como parâmetro o *token* do usuário validado para efetuar a personificação para o mesmo. Repare também que o método *Undo* é chamado dentro do bloco *finally* para garantir que o mesmo será executado caso algum problema ocorra e assim, evitar com que o usuário fique personificado.

```
VB.NET
Imports System
Imports System.Security.Principal
Imports System.Runtime.InteropServices

Public Class Program
    <DllImport("advapi32.dll")> _
    Public Shared Function LogonUser( _
        ByVal lpszUsername As String, _
        ByVal lpszDomain As String, _
        ByVal lpszPassword As String, _
        ByVal dwLogonType As Integer, _
        ByVal dwLogonProvider As Integer, _
        ByRef phToken As IntPtr) As Boolean
    End Function

    Public Shared Sub Main()
        Impersonate()
    End Sub
```

```
    Public Shared Sub Impersonate()
        Dim token As IntPtr
        Try
            If LogonUser("Teste", String.Empty, "123456", 2, 0,
token) Then
                Dim identity As New WindowsIdentity(token)
                Dim impersonationContext As
WindowsImpersonationContext = Nothing

                Try
                    Console.WriteLine("1: " +
WindowsIdentity.GetCurrent().Name)
                    impersonationContext = identity.Impersonate()
                    Console.WriteLine("2: " +
WindowsIdentity.GetCurrent().Name)
                Finally
                    impersonationContext.Undo()
                    Console.WriteLine("3: " +
WindowsIdentity.GetCurrent().Name)
                End Try
            End If
        Finally
            token = IntPtr.Zero
        End Try
    End Sub
End Class
```

C#

```
using System;
using System.Security.Principal;
using System.Runtime.InteropServices;

class Program
{
    [DllImport("advapi32.dll")]
    private static extern bool LogonUser(
        String lpszUsername,
        String lpszDomain,
        String lpszPassword,
        int dwLogonType,
        int dwLogonProvider,
        out IntPtr phToken);

    static void Main()
    {
        WindowsIdentity identity = WindowsIdentity.GetCurrent();
        Impersonate();
    }

    private static void Impersonate()
```

```
    {
        IntPtr token;

        try
        {
            if (LogonUser("Teste", string.Empty, "123456", 2, 0,
out token))
            {
                WindowsIdentity       identity       =       new
WindowsIdentity(token);
                WindowsImpersonationContext   impersonationContext
= null;

                try
                {
                    Console.WriteLine("1:           "           +
WindowsIdentity.GetCurrent().Name);
                    impersonationContext                        =
identity.Impersonate();
                    Console.WriteLine("2:           "           +
WindowsIdentity.GetCurrent().Name);
                }
                finally
                {
                    impersonationContext.Undo();
                    Console.WriteLine("3:           "           +
WindowsIdentity.GetCurrent().Name);
                }
            }
        }
        finally
        {
            token = IntPtr.Zero;
        }
    }
}
```