NGWS runtime

Supporting Security Declarations and Annotations

Original: December 17, 1999

Last revision: June 2, 2000

This specification provides background information on how software development tools can support NGWS security declarations and annotations. For additional information on these topics see the related documents listed in Section 6.

The purpose of this document is to provide basic "how-to" information for software tools and reduce the amount of background reading necessary to understand how to support NGWS runtime security relevant information. There are four areas which compilers/software development tools are expected to support:

- Declarative security checks in code
- Security permission request sets in assembly manifests
- Marking of unverifiable code (not applicable for languages that generate verifiably type safe code such as VB 7.0 and C#)
- Insertion of custom evidence into an assembly manifest

Table of Contents

1 Supporting Declarative Security....3
1.1 Overview of Supporting Declarative Security....3
1.2 Implementation Discussion....4
1.2.1 Security Custom Attributes....4
1.2.2 API for Emitting Declarative Security Checks....6
2 Assembly Security Permission Requests....8
2.1 Overview of Assembly Security Permission Requests....8
2.2 Generating Assembly Permission Requests....9
2.2.1 Using XML-encoded Permission Sets....9
2.2.2 Using Custom Attributes....9
2.3 XML Permission Encoding....10
2.3.1 Permission Set Encoding....10
2.3.2 Permission Encoding....10
3 Marking Unverifiable Code Overview....12
3.1 Marking Unverifiable Code....12
3.2 Requesting the SkipVerification Permission....12
4 Inserting Evidence in Assemblies....14
5 Dynamic Assemblies....15
5.1 Declarations in dynamic assemblies....15
5.2 Asembly Permission Requests....15
5.3 Inserting Evidence....15
6 Relevant Documents....17

1 Supporting Declarative Security

1.1 Overview of Supporting Declarative Security

The NGWS runtime provides APIs to allow compilers to emit metadata information representing developer defined declarative security checks (see NGWS runtime Security Permissions in section 6). This process is supported using custom attributes, and is consistent with other uses of custom attributes. This should make it easier for tools to support this functionality based on common language syntax.

The NGWS runtime security team has defined security attribute classes corresponding to the default security permissions that ship with NGWS runtime. These security attributes derive from a special security attribute (System.Security.Permission.SecurityAttribute defined in Section 1.2.1) that is a subclass of the base custom attribute class (System.Attribute). The security attribute classes are packaged so that they are in the same namespace as the related permission classes. This insures that if the tool can resolve the custom attribute declaration to the associated attribute class, our supporting infrastructure can also resolve to the associated permission class. Custom Permissions are required to follow this basic pattern.

Tools are expected to parse security declarations, determine the appropriate security attribute class, and call a provided API providing the security attribute class and constructor values (see Section 1.2.2). Once the runtime has all declarations applicable to a given element (Class or Method[1]) they are post-processed to resolve any references (for example, to a Publisher identity certificate), union the declarations into a permission set, and generate a serialized object representation of the permission set. This is done to eliminate potential security holes that could arise if reference resolution were deferred until runtime and improves runtime efficiency.

These latter steps are handled transparently to tools. As a side effect however, the security declaration attributes are not visible via reflection as one might expect for custom attributes in general. This is not considered a major issue but is something software developers need to be aware of.

The subsequent sections review the API information tools must understand to handle security declarations (Section 1.2.2)

In summary, the NGWS runtime provides:

- Defined security permission attribute classes, all derived from SecurityAttribute so they are easily identified as security attribute classes.
- Basic syntax rules for declaring parameters on attribute declarations. In particular, where are references to external info allowed, can multiple references be provided, etc.
- As part of the API definition, we specify how descriptive error messages are returned to the compiler to indicate problems in the security declarations

Tools are expected to provide:

- A language syntax for expressing security declarations

[1] One should also be able to put declarative security checks on a property, but this should always be passed to the runtime metadata as though the declaration was on the individual get and/or set methods.

- Support for calling the provided API(s) to properly encode security declarations in the Metadata
- UI reporting mechanism to provide user feedback on security declaration errors

1.2 Implementation Discussion

This section describes support for tools implementing declarative security checks via unmanaged code interfaces. Tools using the reflection emit managed code interfaces use different API as described in Section 5.

1.2.1 Security Custom Attributes

Declarative security takes the form of one or more custom attribute class references preceding a class, or method declaration. The references contain constructor arguments that both specify the security action to take (assert, deny, demand etc) and any state data associated with the permission that the custom attribute maps to (e.g. allowed file name for a FileIOPermission).

All security attribute classes derive from a single well known class (System.Security.Permissions.SecurityAttribute). This will allow tools to detect that a given custom attribute class is a security attribute class without tying all security attribute classes to the same namespace (this is important to allow the extensibility of security permissions). COR_BASE_SECURITY_ATTRIBUTE_CLASS defines the SecurityAttribute class name in cor.h as an assist.

Code generation tools must detect security attribute classes prior to emitting it into metadata since only a typeref to a custom attribute class is stored. The typeref is insufficient to allow us to determine the parent class and the namespace of the attribute class itself.

The SecurityAttribute class definition is:

```
public abstract class SecurityAttribute : System.Attribute
{
    protected SecurityAction m_action;
    protected bool m_unrestricted;

    public SecurityAttribute( SecurityAction action )
    {
        m_action = action;
    }

    public SecurityAction Action
    {
        virtual get { return m_action; }
        virtual set { m_action = value; }
    }

    public bool Unrestricted
    {
```

```
            virtual get { return m_unrestricted; }
            virtual set { m_unrestricted = value; }
        }

        abstract public IPermission CreatePermission();
    }
```

As should be evident from this, all SecurityAttribute classes will support the ability to request the "unrestricted" version of a given permission plus an "action code" to indicate what type of check is desired. The action codes are defined by a SecurityAction enum (in namespace System.Security.Permissions):

```
public enum SecurityAction
{
    /**
     * Hint that permission may be required
     */
    Request = 1;

    /**
     * Demand permission of all caller
     */
    Demand = 2;

    /**
     * Assert permission so callers don't need
     */
    Assert = 3;

    /**
     * Deny permissions so checks will fail
     */
    Deny = 4;

    /**
     * Reduce permissions so check will fail
     */
    PermitOnly = 5;

    /**
     * Demand permission of caller
     */
    LinkDemand = 6;
```

```
    /**
     * Demand permission of a subclass
     */
    InheritanceDemand = 7;

    /**
     * Request minimum permissions to run
     */
    RequestMinimum = 8;

    /**
     * Request optional additional permissions
     */
    RequestOptional = 9;

    /**
     * Refuse to be granted these permissions
     */
    RequestRefuse = 10;

}
```

1.2.2 API for Emitting Declarative Security Checks

The unmanaged API below allows development tools to emit declarative security checks into metadata. As noted earlier, declarative security checks should be supported using a custom attribute syntax. Security custom attributes are always processed as a group on a per-element (class or method) basis, and error information is passed back to the calling code if a problem is detected.

```
HRESULT DefineSecurityAttributeSet(
   mdToken  tkObj,   // [IN] Class or method requiring security
attributes
   COR_SECATTR rSecAttrs[],// [IN] Array of security attribute
descriptions
   ULONG  cSecAttrs, // [IN] Count of elements in above array
   ULONG  *pulErrorAttr);  // [OUT] On error, index of attribute
causing problem
```

This requires use of the COR_SECATTR data structure, defined in cor.h, to express the entries in the attribute array:

```
typedef struct {
mdMemberRef  tkCtor; // Ref to constructor of security attribute
   const void *pCustomValue; // Blob describing ctor
args&field/property values
```

```
    ULONG   cbCustomValue;  // Length of the above blob
} COR_SECATTR;
```

To aid in the construction of the pCustomValue parameter, we provide an unmanaged action code enumeration, corresponding to System.Security.Permissions.SecurityAction.

```
typedef enum CorDeclSecurity
{
    dclActionMask       =   0x000f,     // Mask allows growth of enum.
    dclActionNil        =   0x0000,
    dclRequest          =   0x0001,     //
    dclDemand           =   0x0002,     //
    dclAssert           =   0x0003,     //
    dclDeny             =   0x0004,     //
    dclPermitOnly       =   0x0005,     //
    dclLinktimeCheck    =   0x0006,     //
    dclInheritanceCheck =   0x0007,     //
    dclRequestMinimum   =   0x0008,     //
    dclRequestOptional  =   0x0009,     //
    dclRequestRefuse    =   0x000a,     //
    dclMaximumValue     =   0x000a,     // Maximum legal value
} CorDeclSecurity;
```

In using this API, tools should follow the following scheme:

- For each custom attribute declaration on a class or method, determine whether it's a security custom attribute or not. That is, check whether the custom attribute class is derived from the SecurityAttribute class. For non-security custom attributes, call DefineCustomAttribute() to emit the attribute to metadata. For security custom attributes, remember the attribute definition (the information needed for a COR_SECATTR structure) and continue parsing.
- Once all custom attributes for a given class or method have been processed, pass all the security custom attributes in a single call to DefineSecurityAttributeSet(). If DefineSecurityAttributeSet() is subsequently called for the same class/method, the second call will overwrite the first (i.e. permission set merging is not done).
- If an error occurs, the ULONG pointed to by pulErrorAttr will be updated to indicate which attribute was at fault. If the error is general, *pulErrorAttr will be set to cSecAttrs.

The APIs EnumPermissionsSets and GetPermissionSetProps are used to retrieve permission set data that was actually persisted into the metadata.

2 Assembly Security Permission Requests

2.1 Overview of Assembly Security Permission Requests

The NGWS runtime security system grants permissions to managed code based on the evidence from the code, permissions requested by the code, and local security policy. The assembly is the basic packaging unit for application code and each assembly manifest may contain a declarative security request for the permissions it needs to run. This section discusses how tools can support the generation of assembly permission requests.

Code permission requests consist of three permission sets, as described below. These requests are only valid when attached to an assembly, i.e., are persisted in the assembly manifest. Any permission set requests attached to other security targets are ignored by the NGWS runtime. If a code permission request is omitted, then code will be granted exactly the permissions authorized by policy

Request Minimum

The "minimum" request set represents the permissions code must be granted in order to run (conversely, without policy sufficient to allow these permissions the code should not be run). This offers some benefit to developers. They can be assured that if the code runs it has at least these permissions, hence they can limit exception handling and error recovery logic necessary if lesser permissions were granted at run-time.

Request Optional

In addition to a minimum request, code may also request an "optional" set of permissions. These would also be granted if authorized by policy. If not granted, the code will still be allowed to run. This allows code to request permissions beyond the minimum required, but the developer should be prepared to gracefully handle security exceptions in the event it lacks these permissions.

Request Refuse

This represents a set of permissions the code is never to be granted, even if policy allows it. This feature is particularly useful in the case of an unrestricted optional request set, in effect it allows one to request all additional permissions *except* those listed here.

Applications can use request refuse to ensure they never get certain permission. For example, an application that browses data but never modifies it may refuse any file write permissions – doing so ensures that even in the event of a bug or malicious use the code will not be able to overwrite the data it operates on.

The following are the common permission request situations that arise:

- Code only needs a specific set of mandatory permissions, the optional request should be empty ("Nothing"); there is no need to explicitly refuse permissions
- Code only needs a specific set of mandatory and optional permissions, the requests should be for just those permissions only and there is no need to explicitly refuse permissions never requested

- Code wishes to get any unspecified permissions that policy allows, yet to never be granted certain specific permissions. To be granted unspecified permissions the optional request should be set to "Everything", in which case request refuse is necessary to refuse specific permissions

2.2 Generating Assembly Permission Requests

2.2.1 Using XML-encoded Permission Sets

One supported approach to inserting a permission request set into the assembly manifest is through an XML-encoded permission set declaration as described in Section 2.3.

Tools could either require the end-user to construct and supply the XML-encoded request set, or generate it after collecting user input via a UI mechanism. Once available, the tools use the custom security attribute

System.Security.Permission.PermissionSetAttribute

This attribute accepts either a reference to a file or a string object with the XML permission set. To indicate which permission request set is being provided, the appropriate action code is used. The relevant values are:

- RequestMinimum = 8,
- RequestOptional = 9,
- RequestRefuse = 10

To attach this to the assembly manifest, the API

DefineSecurityAttributeSet()

described in Section 1.2.2 is called with the mdToken parameter set to the assembly token. In this case, the input security attribute must contain a reference to a valid permission request set or an error will occur. Multiple PermissionSetAttributes are referenced in the call to DefineSecurityAttributeSet() to set all three permission request sets.

2.2.2 Using Custom Attributes

As an alternative to the XML-encoded permission set approach, tools may construct the permission request set(s) using security permission attributes in a manner analogous to inserting declarative security checks. In this case a set of security attributes for the desired permission are used, but the SecurityAction code indicates the permission request is part of a minimum, optional or refused permission request set. These relevant values are:

- RequestMinimum = 8,
- RequestOptional = 9,
- RequestRefuse = 10

Tools build up the assembly permission request set by creating a set of COR_SECATTR structures that represent the permissions desired in each of the three request sets. The API, DefineSecurityAttributeSet() is then called to insert the request set into the assembly manifest.

2.3 XML Permission Encoding

The examples in this section indicate the form of XML-encoded permission sets and individual permissions. The permission specification (see related documents in section 6) is the authoritative reference on permission implementation and should be reviewed if one intends to use this mechanism.

2.3.1 Permission Set Encoding

To build up a permission set including individual permissions one uses the format:

```
<PermissionSet>
   <Permission class="permission class 1">
   </Permission>
   <Permission class="permission class 2">
   </Permission>
</PermissionSet>
```

Alternately, one can reference pre-defined, named permission sets. The one of most interest in this context is the 'Everything' set represented by:

```
<PermissionSet class="System.Security.NamedPermissionSet">
   <Name>Everything</Name>
</PermissionSet>
```

2.3.2 Permission Encoding

Below are some examples of XML permission encoding.

```
<Permission class="System.Security.Permissions.EnvironmentPermission">
   <Read>{string of files}</Read>
   <Write>{string of files}</Write>
</Permission>

<Permission class="System.Security.Permissions.FileDialogPermission">
   <AllFiles/> | <ExceptSystem/> | <NoPresetFolder/> | <NoFiles/>
</Permission>

<Permission class="System.Security.Permissions.FileIOPermission">
   <Read>{string of files & folders}</Read>
   <Append>{string of files & folders }</Append>
   <Write>{string of files & folders }</Write>
</Permission>

<Permission class="System.Security.Permissions.RegistryPermission">
   <Read>{string of keys & values}</Read>
   <Append>{string of keys & values}</Append>
```

```
   <Write>{string of keys & values}</Write>
</Permission>


<Permission class="System.Security.Permission.UIPermission">
   <AllWindows/> | <SafeTopLevelWindows/> | <SafeSubwindows> |
<NoWindows/>
   <AllClipboard/> | <OwnClipboard/> | <NoClipboard/>
</Permission>
```

3 Marking Unverifiable Code Overview

The following describes the approach supported for marking unverifiable code modules and setting an associated assembly level request for the SkipVerification permission.

These markings are an important step in providing ways to know when code will fail verification and provide a basis for end-user trust decisions and associated policy control over verification requirements. Development tools generating type safe IL need not be concerned with this issue.

3.1 Marking Unverifiable Code

Tools should mark modules containing unverifiable code by calling DefineCustomAttribute() to insert a module level unverifiable mark. The custom attribute to use is:

System.Security.UnverifiableCodeAttribute

This derives from System.Attribute. That is, it is a 'normal' custom attribute not a security attribute. This is attached at the module level, i.e., the mdToken parameter is set to the module def. This custom attribute carries no internal state. Its presence in the module metadata is interpreted as indicating the module contains unverifiable code.

It is important that unverifiable code be marked as such in order that it may request the SkipVerification permission (see below). Otherwise, unverifiable code will fail verification and not be allowed to run.

3.2 Requesting the SkipVerification Permission

It is recommended that any development tools building assemblies be prepared to detect, and honor, module level unverifiable code markings. Obviously, tools that only create assemblies from type safe IL code they generate can safely ignore this issue.

Assembly builders can discover if one or more modules contain unverifiable code by enumerating the custom attributes for each module using EnumCustomAttribute. If an UnverifiableCodeAttribute is present, then the assembly will contain unverifiable code. In this case, the assembly builder should insert the SkipVerification permission in the assembly manifest as part of the minimum permission request set (see Section 2.2). In effect, the SkipVerification permission should be unioned in with any developer specified security permission requests.

If an XML-encoded permission request set approach for attaching a permission request set is used (Section 2.2.1) then the SkipVerification permission, represented by

```
     <Permission
class="System.Security.Permissions.SecurityPermission">
      <SkipVerification\>
   </Permission>
```

needs to be merged in with any user supplied XML permission request set. This is fairly straightforward string manipulation and no runtime support is provided to assist in this operation.

If one is using the security attribute set approach (Section 2.2.2) then one merely adds the security attribute

System.Security.Permissions.SecurityPermissionAttribute

To the COR_SECATTR structures. SkipVerification must be set for this attribute by including it in the constructor arguments. The action code of SecurityAction.RequestMinimum set should be used.

4 Inserting Evidence in Assemblies

Compilers that generate assemblies may provide a means for developers to specify 'custom' evidence to be included into the assembly. This evidence can be used at runtime, in conjunction with the security policy system, to determine permission granted when the assembly is loaded.

Assemblies may provide evidence of their own by including it as a resource. The Security.Evidence resource contains a single Evidence object serialized in binary format. The evidence object inside the Evidence collection may be of any type but it may not be any of the standard evidence types supported by the NGWS runtime. This restriction insures inserted evidence cannot override the standard evidence types. For example, Publisher identity is always determined based on an Authenticode™ signature and is never determined by the Security.Evidence resource.

5 Dynamic Assemblies

Dynamic assemblies are created with the AssemblyBuilder class and are typically used by script engines or other hosts that generate IL at runtime. This section describes how support for declarative security and code requests differs for dynamic assemblies at the implementation level. Conceptually, the security features are similar but due to the nature of dynamic assemblies and the APIs used to create them differences exist.

5.1 Declarations in dynamic assemblies

Declarative security can be used on emitted types and methods; analogous to the custom attribute based declarations described earlier, using AddDeclarativeSecurity.

```
AddDeclarativeSecurity (SecurityAction action, PermissionSet pset)
```

There is no support for using the DefineSecurityAttributeSet method when creating dynamic assemblies.

5.2 Assembly Permission Requests

Code permission requests are specified as three permission sets at the time the dynamic assembly is created. System.AppDomain.DefineDynamicAssembly provides parameters for the code request permission sets to be given by the host. The method definition is:

```
AssemblyBuilder DefineDynamicAssembly (AssemblyName name,
   AssemblyBuilderAccess access, String dir, Evidence evidence,
   PermissionSet requiredPermissions, PermissionSet
optionalPermissions,    PermissionSet refusedPermissions)
```

The following pseudo-code fragment illustrates how to create a permission set object consisting of three permissions, P1, P2, P3. Details coding for of each permission depend on its type.

```
PermissionSet set = new PermissionSet();
set.AddPermission(new P1( … ));
set.AddPermission(new P2( … ));
set.AddPermission(new P3( … ));
```

5.3 Inserting Evidence

The host emitting a dynamic assembly can pass evidence into the assembly via the DefineDynamicAssembly API given above, but only if it has been granted the SecurityPermission(ControlEvidence) permission.

The first time code is executed in a dynamic assembly for which evidence was provided, security policy is applied using the evidence and code request supplied. This is used to determine the appropriate permissions to grant based on the system policy. If the dynamic assembly creator did not have permission to provide evidence

then the creator's permissions are applied to the dynamic assembly at creation time and no policy evaluation is needed before code can be executed.

6 Relevant Documents

NGWS runtime Security Permissions

Topic "NGWS Security Permissions Specification" in the SDK Developer's Specifications help (Docs\cpappendix.chm in the NGWS SDK directory)

NGWS runtime Security Policy

Topic "NGWS Security Policy Specification" in the SDK Developer's Specifications help (Docs\cpappendix.chm in the NGWS SDK directory)

NGWS runtime Metadata API

Document "COR Metadata Interfaces" in the SDK Tool Developers Guide (Tool Developers Guide\Docs\COR Metadata Interfaces.doc in the NGWS SDK directory)