O'REILLY®

Java vs .NET Security

Denis Piliptchouk

A practical guide for comparison and contrast of security features offered by Enterprise Java and .NET platforms

Introduction

An introduction to the history and approaches of the Java and .NET platforms, document's philosophy in comparing their respective approaches to security issues.

Security Configuration and Code Containment

A comparison of how the platforms' security features are configured, a survey of how they handle code verification and code containment.

Cryptography and Communication

An analysis of what cryptographic services are available to Java and .NET applications.

Code Protection and Code Access Security

A consideration of how application code is secured against misappropriation and misuse.

User Authentication and Authorization

A description of the platforms' approaches to authenticating users and granting or denying them permission to execute privileged actions.

Conclusion and Summary

Score for each platform's offerings in each area of interest.

Upcoming Security Features

A look ahead to the anticipated security offerings of J2SE 1.5 and the "Whidbey" release of .NET and how they address the shortcomings of previous offerings.

O'Reilly Network eDocument

Table of Contents

Introduction 3

An introduction to the history and approaches of the Java and .NET platforms, plus a statement of the document's philosophy in comparing their respective approaches to security issues.

Chapter 1 — Security Configuration and Code Containment 6

A comparison of how the platforms' security features are configured, followed by a survey of how they handle code verification and code containment issues, such as application isolation and limits imposed by the Java and C# programming languages.

Chapter 2 — Cryptography and Communication 15

An analysis of what cryptographic services are available to Java and .NET applications, how they are configured, and how they are used to secure communication.

Chapter 3 — Code Protection and Code Access Security (CAS) 28

A consideration of how application code is secured against misappropriation and misuse, as well as how cryptographic signing is used to verify that code is authentic and can be trusted.

Chapter 4 — User Authentication and Authorization 48

A description of the platforms' approaches to authenticating users and granting or denying them permission to execute privileged actions.

Conclusion and Summary 70

An overview of the security topics considered in the preceding chapters and a score for each platform's offerings in each area of interest.

Epilogue — Upcoming Security Features 73

A look ahead to the anticipated security offerings of J2SE 1.5 and the "Whidbey" release of .NET and how they address the shortcomings of previous offerings.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Introduction

Philosophy

This document reviews security features of two most popular modern development platforms — Java and .NET (Java v1.4.2/J2EE v1.4 and .NET v1.1). The platform choice is not random, because they represent, to a certain extent, competition between UNIX-like and Windows systems, which largely defined software evolution over the last decade. Although Java applications run on Windows, and there exist UNIX bridges for .NET, the Java/UNIX and .NET/Windows combinations are used for development of a significant portion (if not majority) of applications on their respective operating systems, so both platforms deserve a careful examination of their capabilities.

Such an examination is especially important since different aspects of UNIX/Windows and Java/.NET competition have been flaming endless heated debates between proponents of both camps, which often blindly deny merits of the opposite side while at the same time praising their preferred solution. The material here is purposely structured by general categories of protection mechanism and reviewing each platform's features in those areas. This allows starting each topic with a platform-neutral security concept and performing relatively deep drill-downs for each technology without losing track of the overall focus of providing an unbiased side-by-side comparison.

The document is based on the research material that was used as a foundation of the feature article, "Securing .NET and Enterprise Java: Side by Side", which was written by Vincent Dovydaitis and myself and appeared in Numbers 3-4 of *Computer Security Journal* in 2002. The following areas will be considered:

- Security Configuration and Code Containment
- Cryptography and Communication
- Code Protection and Code Access Security, or CAS
- Authentication and User Access Security, or UAS

It would be unrealistic to expect complete and detailed coverage of all aspects of platform security in the space available. Rather, the document attempts to highlight the most important issues and directions, leaving it up to the reader to check the appropriate manuals and reference literature. Another important aspect of platform security is that it does not attempt to deal with all possible types of threats, thus requiring the services of an OS and third-party software (such as using IIS in .NET). These services and applications will also be outside of the scope of this publication.

Historical Background

Although the foundations of modern computer security were laid in 1960s and 1970s, the scopes of tasks and challenges that needed to be addressed by the software systems of that time and today differ as much as medieval castles and sprawling megapolises of modern age.

Advances in computer hardware were one of the significant contributing factors — it is sufficient to remind that today's average PC workstations, sitting on office desks, easily surpass a super-computer Cray from late 1970s in terms of computing power. Such advances allowed development of new types of applications, unattainable twenty years ago, and making them available on people's desktops. While previously execution of software applications and results interpretation was an exclusive domain of bearded and gloomy half-gods —

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

inhabitants of computer rooms, which were taking up whole floors — now it has been pushed down to an average Joe. Correspondingly, an average user now has enough skills for powering up the computer and starting an application, but not nearly enough to protect himself from his own actions.

The Internet revolution represented another quantum leap of software evolution. As very close-knit communities of programming professionals from 60s-70s gave way to a myriad of loosely-coupled nodes on the world-wide Web, castle-like protection mechanisms started failing when faced with new, distributed paradigm. Once a computer was brought out of the isolated facility and connected to the global network, the old trusted and nurturing environment was gone, and the operating systems together with applications were confronted with hostile and aggressive environment.

The new realities required new types of platform support, as tightly bound, monolithic, and relatively small applications were being replaced by newer component-based distributed systems. These systems also required different types of protection: their modules were often supplied by different vendors, trusted and not so, direct authentication was no longer always possible, replaced by third-party and offline modes, encrypted data was not physically carried, but electronically transported via public networks, authorizations were issued by policy servers and had to be relayed via trusted mechanisms, etc. The emerging platforms had to take into account all these (and many other) requirements in order to become viable and attractive.

Platform Introduction – Java

The Java platform was brought to the world by a group of enthusiastic engineers from Sun, who had a nice technological concept and were trying to find an application for it in the first half of 1990s. The project, initially dubbed "The Green Project", changed its name several times, until the term Java emerged, allegedly as a tribute to the various coffee places where the group regularly held its meetings. The technology, initially slated to be incorporated into digital consumer devices like TV set-top boxes, was not accepted at that time because the industry was not yet ready for the concept. The onset of the Internet boom, however, was a fateful coincidence for the new platform — suddenly, it was perceived as the future global platform for Java, especially after it was endorsed by Netscape, "The Browser" of those days.

Java borrowed some concepts from SmallTalk and a number of earlier research environments to produce a platform based on a virtual machine (VM), which uses a single language (Java) to run on multiple operating systems... hence the famous catchy slogan "write once, run anywhere". However, the Internet-based distribution and execution model, intermediate bytecode language, interpreted by the JVM at run time, and other innovative features immediately presented Java applications (and their users) with new, unexpected ways to attack them. Correspondingly, the JVM had to incorporate more and more security features that are typical of operating systems. The application programmers, too, had to become aware of the Internet realities and could no longer assume that their applications are going to be running fully isolated behind concrete walls of corporate offices.

Platform Introduction – .NET

Microsoft went through several stages in its quest to come up with its own Internet platform. COM and COM+ were the early attempts to create loosely-coupled distributed applications, but they brought with them a very steep learning curve and still did not provide the desired solution. Attempts to fool around with "enhancing" Java ended in a lengthy and unpleasant court battle with Sun, which ended with a legal defeat for Microsoft, withdrawing the license,

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

and banning them from making modifications to the Java language.

Even though .NET has a number of strikingly similar characteristics to the JVM, it took its roots in a little-known *OmniVM*, developed by Colusa Software, which was acquired by Microsoft in 1996. That VM pioneered some of the features that later became the pillars of .NET technology: multiple languages running on the same VM, application isolation model (known in .NET as `AppDomains`), JIT to native code, as opposed to Java's interpreter-based approach. Of course, .NET architects carefully studied Java's features and made use of concepts like code verification, automatic memory management, garbage collection, etc.

In order to support its paradigm of "multiple languages — one platform", .NET had to define a minimalist subset of its functionality, that became known as *Common Language Infrastructure*, or CLI, and impose certain restrictions on the managed languages. The main rule: the language must produce a verifiable type-safe code that can passes bytecode verification. Since traditional C++, because of its immense flexibility and memory pointers, could not be made verifiably safe without severely restricting it, Microsoft came up with a clever idea of producing a new language that would combine the best features from the most influential modern languages. The language, called C#, was introduced with .NET 1.0, and it incorporates many familiar Java concepts, as well as borrowing some nice traits from C++ and a number of other programming languages.

As C# is designed to be the most feature-complete language in the .NET framework, it was chosen for all examples and discussions on the .NET side. In some cases, other languages may implement only subsets of its functionality — check the documentation for the appropriate language to see what it supports.

Although Java's success and its apparent threat to Microsoft's market dominance were one obvious reason for the .NET shift, business reasons also played a significant role in this move — the myriad of existing products and technologies did not have the unifying basis, besides the ever-growing Windows platform. Coincidentally, a number of security initiatives were launched in the same timeframe with .NET (with various degrees of success), and the company's leadership has been regularly bringing the terms "security" and ".NET" together in the same sentence ever since.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Chapter 1 – Security Configuration and Code Containment

Configuration

Configuration on both platforms is handled through XML or plain-text files, which can be modified in any text editor, or through the supplied tools. However, the platforms differ significantly in how they handle configuration hierarchies.

In the .NET world, tools like `Mscorcfg.msc` and `Caspol.exe` can be used to modify all aspects of security configuration. The former displays a GUI interface, shown in Figure 1-1, to perform GUI-based administration tasks.

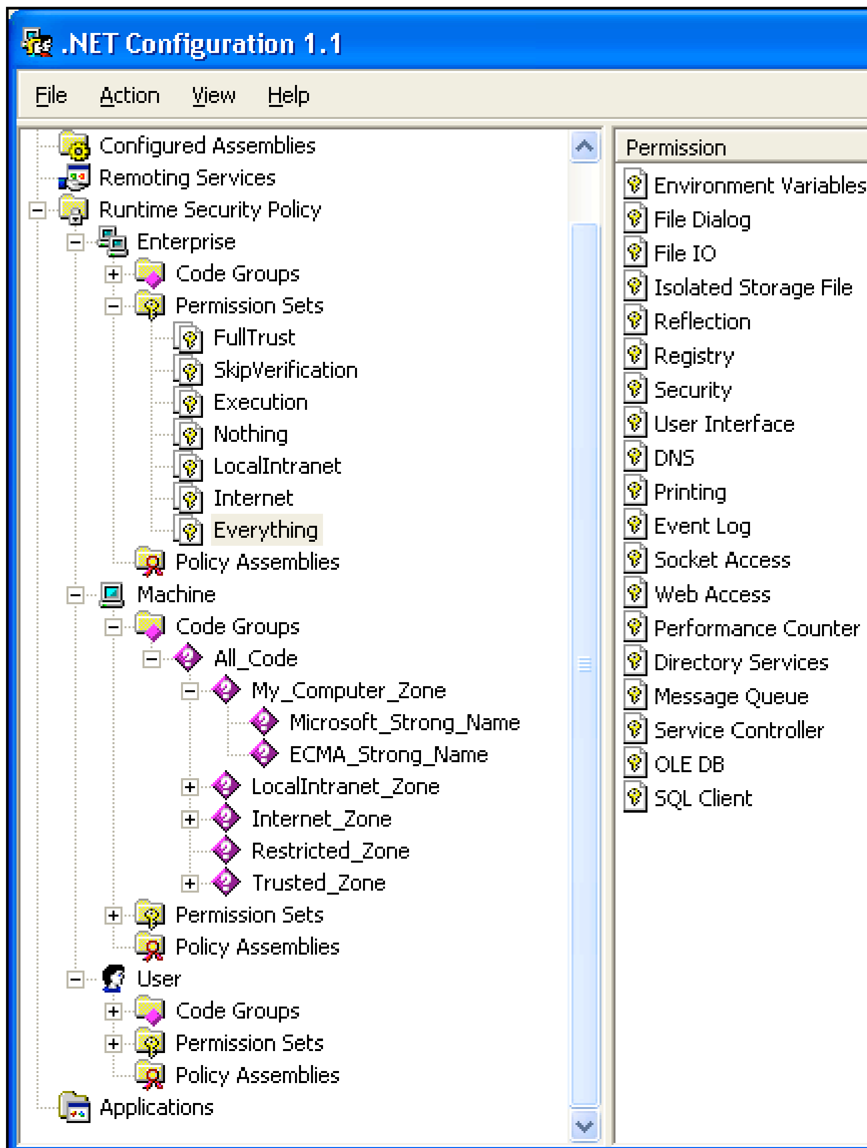

Figure 1-1. Mscorcfg.msc screen

On the other hand, `Caspol.exe` provides a number of command-line options, appropriate for use in scripts and batch routines. Here's how it would be used to add full trust to an

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

assembly: `caspol -af MyApp.exe`.

The Java platform provides a single GUI-based tool, `policytool.exe`, shown in Figure 1-2, for setting code- and Principal-based security policies. This tool works with arbitrary policy files (as long as they are in the proper format), as opposed to .NET, where names and locations of the configuration files are fixed (see below).

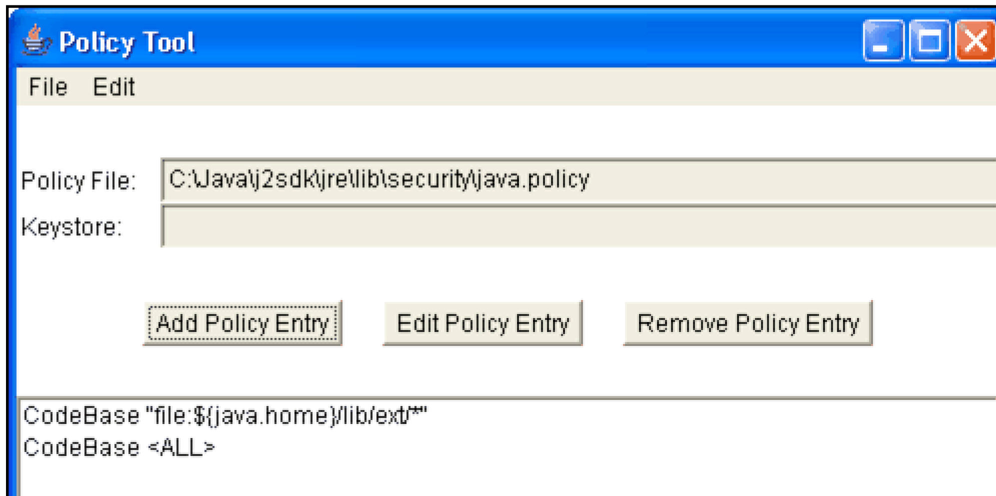

Figure 1-2. Policytool.exe screen

.NET defines machine-wide and application-specific configuration files, and allows for enterprise, machine, and user security policy configuration, whose intersection provides the effective policy for the executing user. These file have fixed names and locations, most of them residing under the Common Library Runtime (CLR) tree, at: `%CLR install path%\Config`

For .NET v1.1, the location is: `C:\WINNT\Microsoft.NET\Framework\v1.1.4322\CONFIG`.

Multiple versions of CLR may co-exist on the same computer, but each particular version can have only a single installation. Since security policy files cannot be specified as runtime parameters of .NET applications, this centralized approach hurts co-existence when applications require conflicting policy settings. For instance, if one attempts to strip locally installed code of `FullTrust` permission set in order to make Code Access Security features of his application work right, it will, most likely, break a number of existing programs which rely on this setting.

Three security configuration files (enterprise, machine, and user) contain information about configured zones, trusted assemblies, permission classes, and so on. Additionally, the general machine configuration file contains machine-wide settings for algorithms, credentials, timeouts, etc, and certain application-specific parameters (for instance, `ASP.NET` authentication/authorization parameters) can be configured or overriden in the application configuration file. The files' names and locations are listed below:

- User security configuration file: `%userprofile%\Application data\Microsoft\CLR security config\vxx.xx\Security.config`
- Machine security configuration file: `%CLR install path%\Config\Security.config`
- Enterprise security configuration file: `%CLR install path%\Config\Enterprisesec.config`
- Machine configuration file: `%CLR install path%\Config\machine.config`
- Application configuration files: `<AppName>.exe.config` (or `Web.config` for `ASP.NET`) in the application or web project's main directory

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Core Java and J2EE configuration files have specific locations, but locations of additional configuration files for extension and J2EE products vary by vendor. J2SE does provide significant runtime flexibility by using a number of command-line application parameters, which allow the caller, on a per-application basis, to set keyfiles, trust policy, and extend the security policy in effect:

```
java -Djava.security.manager
  -Djava.security.policy=Extend.policy Client1
```

or to completely replace it:

```
java -Djava.security.manager
  -Djava.security.policy==Replace.policy Client2
```

The Java platform's specifications require the following configuration files:

- J2SE: `$JAVA_HOME/lib/security/java.security`
  This file defines security properties for VM: security providers, policy providers, package access restrictions, keystore types, and so on.
- J2SE: `$JAVA_HOME/lib/security/java.policy, $HOME/.java.policy`
  Machine and user security policy that grants evidence- and Principal-based code permissions. Additional/alternative files may be specified on the command line or in the `java.security` file.
- J2EE: `%application dir%/WEB-INF/web.xml`, `%application dir%/META-INF/ejb-jar.xml`
  These files contain Servlet and EJB deployment instructions and include, among other parameters, authentication/delegation settings, security roles, role-based Access Control Lists (ACL), and transport security configuration. The "UAS" section in Part 4 will provide more detailed coverage of their elements.

Certain JVM parameters may be configured only in `$JAVA_HOME/lib/security/java.security`, as shown in the examples below:

- Adding third-party providers: `security.provider.<Number>=<ProviderClassName>`
- Configuring alternative policy providers: `policy.provider=<ProviderClassName>`
- Specifying multiple policy files: `policy.url.<Number>=file:<URL>`

Note: By allowing command-line JVM parameters, Java provides a significantly more flexible and configurable environment, without conflicts among multiple JVM installations.

Code Containment: Verification

In both environments, the respective VM starts out with bytecode, which it verifies and executes. The bytecode format is well known and can be easily checked for potential violations, either at loading or at execution time. Some of the checks include stack integrity, overflow and underflow, validity of bytecode structure, parameters' types and values, proper object initialization before usage, assignment semantics, array bounds, type conversions, and accessibility policies.

Both Java and CLS languages possess memory- (or type-) safety property; that is, applications written in those languages are verifiably safe, if they do not use unsafe constructs (like calling into unmanaged code).

In .NET, CLR always executes natively compiled code; it never interprets it. Before IL is

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

compiled to native code, it is subjected to validation and verification steps. The first step checks the overall file structure and code integrity. The second performs a series of extensive checks for memory safety, involving stack tracing, data-flow analysis, type checks, and so on. No verification is performed at runtime, but the Virtual Execution System (VES) is responsible for runtime checks that type signatures for methods are correct, and valid operations are performed on types, including array bounds checking. These runtime checks are accomplished by inserting additional code in the executing application, which is responsible for handling error conditions and raising appropriate exceptions. By default, verification is always turned on, unless `SkipVerification` permission is granted to the code.

The Java VM is responsible for loading, linking, verifying, and executing Java classes. In the HotSpot JVM, Java classes are always interpreted first, and then only certain, most frequently used sections of code are compiled and optimized. Thus, the level of security available with interpreted execution is preserved. Even for compiled and optimized code, the JVM maintains two call stacks, preserving original bytecode information. It uses the bytecode stack to perform runtime security checks and verifications, like proper variable assignments, certain type casts, and array bounds; that is, those checks that cannot be deduced from static analysis of Java bytecode.

Code verification in a JVM is a four-step process. It starts by looking at the overall class file format to check for specific tags, and ends up verifying opcodes and method arguments. The final pass is not performed until method invocation, and it verifies member access policies. By default, the last step of verification is run only on remotely loaded classes. The following switches can be passed to JVM to control verification:

- `-verifyremote`: verifies only classes from the network (default)
- `-verify`: verifies all classes
- `-noverify`: turns off verification completely

Starting with the initial releases of Java, there have been multiple verification problems reported, where invalid/malicious bytecode could sneak beyond the verifier. At the moment, there are no new reports about verification bugs, and Java 2 documentation does not list verification switches, which implies that the verification is always run in full.

However, the `-verify` switch is still required for local code to behave correctly, as the following example shows. Given class `Intruder`...

```
public class Intruder
{
   public static void main(String[] args)
   {
      Victim v = new Victim();
      System.out.println(
        "Intruder: calling victim's assault() method...");
      v.assault();
   }
}
```

A `Victim` class with a `public` method:

```
public class Victim
{
   public void assault()
   {
      System.out.println(
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
         "Victim: OK to access public method");
   }
}
```

And another version of the `Victim` class with a `private` method:

```
public class Victim
{
   private void assault()
   {
      System.out.println(
        "Victim: Private method assaulted!!!");
   }
}
```

We get the following output when we run a script to compile and run `Intruder` first against the `public` version of `Victim`, and then, without recompiling the `Intruder` class, against the `private` verison. Finally, it is run against the `private` version again, this time with `-verify` passed as a command-line argument to JVM:

```
********************************************
* Calling public version of Victim.assault()
********************************************
Intruder: calling victim's assault() method...
Victim: OK to access public method
*********************************************
* Calling private version of Victim.assault()
*********************************************
Intruder: calling victim's assault() method...
Victim: Private method assaulted!!!
****************************************************
* Calling private Victim.assault() with verification
****************************************************
Intruder: calling victim's assault() method...
java.lang.IllegalAccessError:
 tried to access method Victim.assault()V from class Intruder
        at Intruder.main(Intruder.java:7)
Exception in thread "main"
```

The sources and the `execute.bat` file are available as Java.I.NoVerification.zip for download.

Note: JVM, as opposed to .NET, does not verify local code by default. On the other hand, JVM always preserves the bytecode stack for runtime checks, while .NET relies on a combination of static analysis and injection of verification code at runtime.

Code Containment: Application Isolation

In effect, each VM represents a mini OS by replicating many of its essential features. Each platform provides application isolation for managed applications running side by side in the same VM, just as OSes do it. Automatic memory management is an important feature of both environments — it aids tremendously in writing stable, leak-free applications. The "CAS" section in Part 3 will provide detailed discussion about permissions, policies, and access checks.

Both environments do allow for exercising unsafe operations (`JNI` in Java; `unsafe` code and `P/`

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Invoke in .NET), but their use requires granting highly privileged code permissions.

Application Domains (AppDomains) represent separate .NET applications running inside the same CLR process. Domain isolation is based on the memory safety property because applications from different domains cannot directly access each other's address spaces, and they have to use the .NET Remoting infrastructure for communication.

Application security settings are determined by CLR on a per-domain basis, by default using host's security settings to determine those for loaded assemblies. The CLR receives information about the assembly's evidence from so-called trusted hosts:

- Browser host (Internet Explorer): Runs code within the context of a web site.
- Server host (ASP.NET): Runs code that handles requests submitted to a server.
- Shell host: Launches managed applications (.exe files) from the Windows shell.
- Custom hosts: An application that starts execution of CLR engine.

Domain security settings can be administered only programmatically; that is, there is no configuration file where those could be set. If the host process is granted a special SecurityPermission to control evidence, it is allowed to specify the AppDomain's policy at creation time. However, it can only reduce the compound set of permissions granted by the enterprise, machine, and user policies from security policy files. The following example, taken from MSDN documentation, illustrates using programmatic AppDomain policy administration to restrict permission set of the new domain to Execution only:

```
using System;
using System.Threading;
using System.Security;
using System.Security.Policy;
using System.Security.Permissions;

namespace AppDomainSnippets
{
   class ADSetAppDomainPolicy
   {
      static void Main(string[] args)
      {
         // Create a new application domain.
         AppDomain domain =
            System.AppDomain.CreateDomain("MyDomain");

         // Create a new AppDomain PolicyLevel.
         PolicyLevel polLevel =
            PolicyLevel.CreateAppDomainLevel();
         // Create a new, empty permission set.
         PermissionSet permSet =
            new PermissionSet(PermissionState.None);
         // Add permission to execute code to the
         // permission set.
         permSet.AddPermission
            (new SecurityPermission(
               SecurityPermissionFlag.Execution));
         // Give the policy level's root code group a
         // new policy statement based on the new
         // permission set.
         polLevel.RootCodeGroup.PolicyStatement =
            new PolicyStatement(permSet);
         // Give the new policy level to the
         // application domain.
         domain.SetAppDomainPolicy(polLevel);
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
         // Try to execute the assembly.
         try
         {
            // This will throw a PolicyException
            // if the executable tries to access
            // any resources like file I/O or tries
            // to create a window.
            domain.ExecuteAssembly(
               "Assemblies\\MyWindowsExe.exe");
         }
         catch(PolicyException e)
         {
            Console.WriteLine("PolicyException: {0}",
                              e.Message);
         }

         AppDomain.Unload(domain);
      }
   }
}
```

Application-style isolation is achieved in Java through a rather complicated combination of `ClassLoaders` and `ProtectionDomains`. The latter associates `CodeSource` (i.e., URL and code signers) with fixed sets of permissions, and is created by the appropriate class loaders (URL, `RMI`, custom). These domains may be created on demand to account for dynamic policies, provided by `JAAS` mechanism (to be covered in Part 4, in the "Authentication" section). Classes in different domains belong to separate namespaces, even if they have the same package names, and are prevented from communicating within the JVM space, thus isolating trusted programs from the untrusted ones. This measure works to preserve and prevent bogus code from being added to packages.

Secure class loading is the cornerstone of JVM security — a class loader is authorized to make decisions about which classes in which packages can be loaded, define its `CodeSource`, and even set any permissions of its choice. Consider the following implementation of `ClassLoader`, which undermines all of the access control settings provided by the policy:

```
protected PermissionCollection
    getPermissions (CodeSource src) {
        PermissionCollection coll =
            new Permissions();
        coll.add(new AllPermission());
        return coll;
}
```

Note: .NET's `AppDomains`, which are modeled as processes in an OS, are more straightforward and easier to use than Java's `ProtectionDomains`.

Code Containment: Language Features

Both platforms' languages have the following security features:

- Strong typing (a.k.a. statically computable property): all objects have a runtime type. There is no `void` type: a single-root class hierarchy exists, with all classes deriving implicitly from Object root.
- No direct memory access: therefore, it is impossible to penetrate other applications'

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

memory space from managed code.

- Accessibility/const modifiers (such as `private/protected/public`): The `const` (`final`) modifier's semantics, however, is quite different from that in C++; only the reference is constant, but the object's contents can be freely changed.
- Default objects initialization: "zero initialization" for heap-allocated objects. Proper initialization of stack objects is checked by the VM at runtime.
- Choice of serialization and transient options: controls contents of serialized objects that are outside of VM protection domains.
- Explicit coercion required: there are few well-defined cases when implicit coercion is used. In all other cases (and with custom objects) explicit conversion is required.

.NET defines the following accessibility modifiers: `public`, `internal` (only for the current assembly), `protected`, `protected internal` (union of protected and internal), and `private`. All properties are defined via Getters/Setters, and access to them is controlled at runtime by CLR.

In C#, there are two choices for declaring constant data values: `const` for built-in value types, whose value is known at compile time); and `readonly` for all others, whose value is set once at creation time:

```
public const string Name = "Const Example";
//to be set in the constructor
public readonly CustomObject readonlyObject;
```

A .NET class can be marked as serializable by specifying `[Serializable]` attribute on the class. By default, its full state is stored, including private information. If this is not desirable, a member can be excluded by specifying a `NonSerialized` attribute, or by implementing a `ISerializable` interface to control the serialization process.

```
[Serializable]
public struct Data
{
  //Ok to serialize this information
  private string publicData;
  //this member is not going to be serialized
  [NonSerialized] private string privateData;
}
```

Java language provides the following features to support writing secure applications:

- Accessibility modifiers: `public, protected, package protected, private`.
- Final classes and methods: `final` keyword can be applied to a class, method, or variable, and means that this entity cannot be changed or overridden.
- Serialization and transient options: for classes implementing a marker `Serializable` interface, the serialized object includes private members as well, unless they are decorated as static or transient. Use the `readObject/writeObject` pair to control the content of a serialized object. Alternatively, implementing the `Externalizable` interface's methods `readExternal/writeExternal` gives you complete control over the serialization process.

```
public class Person implements Serializable
{
  //get serialized by default
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
  private string name, address;
  //excluded from the default serialization
  transient int salary;
};
```

Note: In terms of protective language features, both platforms rate approximately equal, with .NET having a slight edge due to higher flexibility when it comes to constant modifiers.

Chapter 1 – Conclusions

This section covered security configuration issues and different aspects of code containment on .NET and Java platforms. Java offers a lot of advantages with its configurability. When it comes to code containment, both platforms have pretty strong offerings, with .NET having slightly more choices and being more straightforward to use.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Chapter 2 – Cryptography and Communication

In today's world, most of the means of secure data and code storage and distribution rely on using cryptographic schemes, such as certificates or encryption keys. Thus, cryptography mechanisms form a foundation upon which many important aspects of a solid security system are built, and it is crucial for a really secure platform to provide adequate support for these services.

Once an application steps out of the bounds of a single-computer box, its external communication is immediately exposed to a multitude of outside observers with various intentions, their interests ranging from employers scanning the list of web sites an employee visits to business spies looking for a company's "know-how". In order to protect sensitive data while it is en route, applications invoke different methods, most often with some kind of cryptographic protection applied to the data before transmitting it. Any respectable enterprise system has to demonstrate adequate protection measures in this area.

Cryptography: General

Cryptography in .NET is based to a large extent on the Windows `CryptoAPI` (CAPI) service, with some extensions. Many algorithms are implemented as managed wrappers on top of CAPI, and the key management system is based on CAPI key containers. Most cryptography-related classes reside in the `System.Security.Cryptography` namespace, with certificate classes separated into `X509Certificates` and XML digital signature functionality into `Xml` subpackages. `Web Service Extensions` (WSE; see Secure Communication section) provides its own set of certificate classes in the `Microsoft.Web.Services.Security.X509` package.

However, .NET's Cryptography service is more than just a managed wrapper — it extends the CAPI in a number of ways. First, it is highly configurable and allows adding custom algorithm implementations in the *machine.config* file. Second, .NET uses a stream-based model, where all cryptographic transformations (except for asymmetric algorithms) are always performed on streams. Third, the defaults for all algorithms are configured to the strongest and safest settings (subject to Windows OS encryption settings, though), so the default objects that the user receives are most secure from what his Windows encryption settings allow.

The cryptography model of .NET is horizontally organized into several layers, and vertically grouped by types. Each family of algorithms (symmetric, asymmetric, etc.) forms a vertical hierarchy, deriving from a single root class for that family, with (usually) two more levels beneath it: an abstract algorithm level, and its concrete implementation. Family root classes are sealed; i.e. they cannot be extended by applications. This means, for instance, that the family of asymmetric algorithms can not be extended beyond the provided RSA and DSA abstractions. By .NET's convention, the implementation class is called `Provider` if it is a wrapper around a CAPI object, or `Managed` if it is a completely new implementation. The (simplified) `System.Security.Cryptography` class hierarchy is shown in Figure 2-1:

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

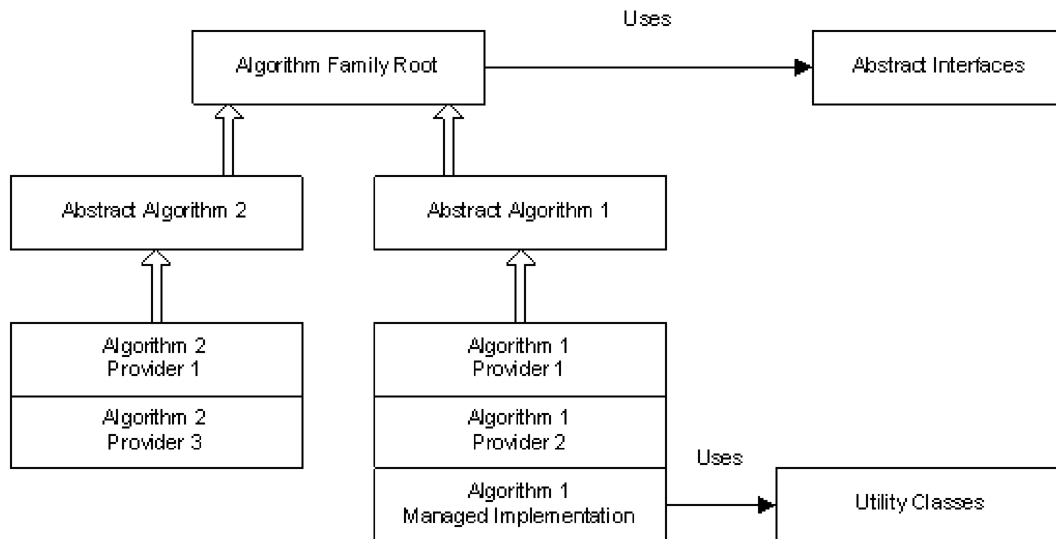

Figure 2-1. .NET cryptography class hierarchy

The Java platform's cryptography support has two parts to it: `Java Cryptography Architecture` (JCA) and `Java Cryptography Extension` (JCE), which were separated (due to US export restrictions) to gain exportability for the Java platform. All cryptography functions, which are subject to export laws, have been moved to JCE. In JDK 1.4, JCE became an internal part of the Java platform, instead of being an optional package, as it had been up to 1.4.

Both JCA and JCE have a similar *provider-based* architecture, which is widely employed in many of the Java platform's solutions. Those packages consist of so-called *frameworks*, which implement the required infrastructure, and a number of additional providers supply cryptography algorithms. JCA and JCE frameworks are internal Java packages, and cannot be replaced or bypassed. The JCE framework authenticates JCE providers, which should to be signed by a trusted `Certificate Authority` (Sun or IBM) — see the JCE Provider Reference for details.

Note that prior to v1.4, JCE was an extension and its framework classes could be supplied by a third-party vendor along with the provider itself, so the problem with signing could be avoided by removing Sun's JCE 1.2.2 provider and framework from the configuration. Since JCE has now become a standard Java package, the signing step poses an additional problem for independent vendors (although, according to representatives from Bouncy Castle, Sun is very cooperative in this matter, which significantly simplifies the involved procedure). Thus, with J2SE v1.4, vendors are forced to undertake the signing procedure, or begin developing proprietary solutions and abandon the JCE framework — see the JCE Reference Guide for further information.

The JCA Provider framework model, shown in Figure 2-2, consists of the following elements:

- `Service` (or `Engine`) abstract classes define types of functions available to developers, independent of particular algorithms: Asymmetric, Symmetric algorithms, Digests, etc.
- `Service Providers Interfaces` (SPI) for each of those services link the high-level abstract `Services` to the provided implementations.
- `Provider` is the central class that registers available implementations with the framework.
- `Security` is the class that handles all providers.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

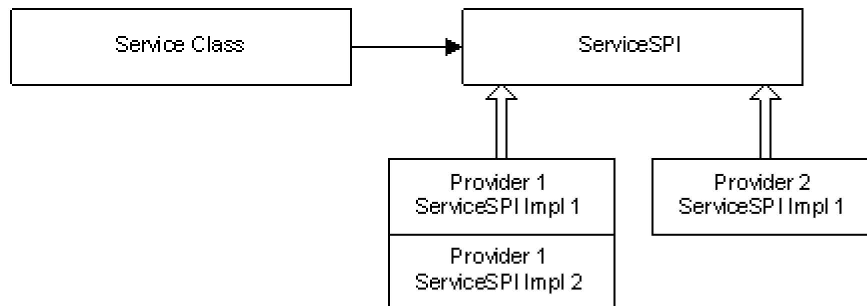

Figure 2-2. JCA class hierarchy

Note: Java requires its crypto providers to be signed by a trusted CA, which poses an obstacle for independent vendors.

Cryptography: Algorithms

Most common industry-standard cryptographic algorithms (Symmetric, Asymmetric, Hashes, Signatures, PBE) and stream/block ciphers are available out-of-the-box on both platforms.

The following families of algorithms are supplied in `System.Security.Cryptography` namespace of .NET:

- `AsymmetricAlgorithm`: digital signatures and key exchange functionality is also implemented by these family's providers (DSA,RSA).
- `HashAlgorithm`: `KeyedHashAlgorithm`, MD5, multiple SHA.
- `SymmetricAlgorithm` (DES, 3DES, RC2, Rijndael): additional parameters are specified by `PaddingMode` and `CipherMode` enumerations.
- `RandomNumberGenerator`.

These asymmetric algorithm helpers are used together with configured asymmetric providers to do their jobs:

- `AsymmetricKeyExchangeFormatter/Deformatter`: provides secure key exchange mechanism using OAEP or PKCS#1 masks.
- `AsymmetricSignatureFormatter/Deformatter`: creates/verifies PKCS#1 v1.5 digital signatures, using any configured by name hash algorithm.

The .NET Cryptography library provides `Password Based Encryption` (PBE) functionality through its `PasswordDeriveBytes` class. It uses the specified hashing algorithm to produce a secret key for the targeted symmetric encryption algorithm. A sample application that demonstrates symmetric encryption with PBE to encrypt/decrypt is available in the dotnet_encryption.zip example.

Symmetric and hash transforms in .NET are stream-based, so multiple transformations can be chained together without creating temporary buffer storage. The `CryptoStream` class derives from the `System.IO.Stream`, and plugs into any framework where stream interfaces are acceptable: memory, data, network, and other kinds of data. `CryptoStream` accepts the `ICryptoTransform` interface, which it then uses internally to transform the data block-by-block by calling `TransformBlock` repeatedly. This interface is implemented differently by symmetric and hash providers:

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

1. Using Streams with Symmetric Algorithms

In case of a symmetric algorithm, the top-level `SymmetricAlgorithm` class defines the abstract methods `CreateEncryptor/Decryptor`. These methods' implementations in derived classes (providers) create an instance of `CryptoAPITransform` class, appropriate for the particular algorithm, and return it to use with `CryptoStream`. The `CryptoAPITransform` class internally hooks to the CryptoAPI Windows service to do the job using the `_AcquireCSP` and `_EncryptData` private unmanaged functions, as shown in Figure 2-3:

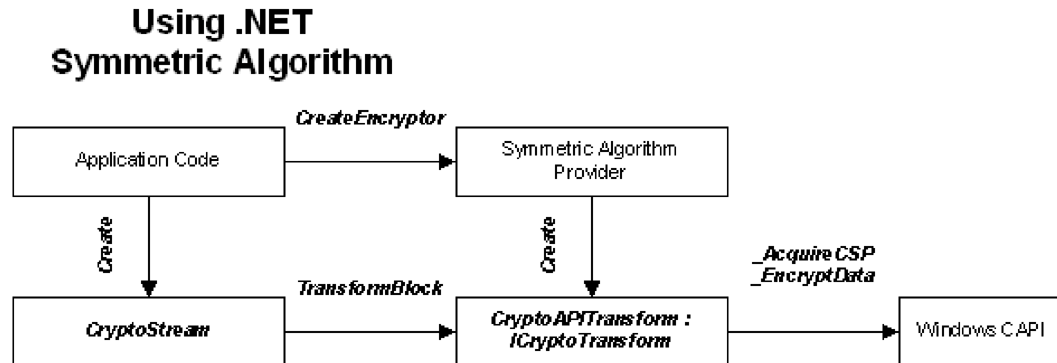

Figure 2-3. Streams with .NET symmetric algorithms

2. Using Streams with Hash Algorithms

The `HashAlgorithm` family root class itself implements the `ICryptoTransform` interface, so any derived object can be used directly with `CryptoStream`. Its implementation of the `TransformBlock` method simply delegates the call to the derived class' implementation of the abstract method `HashCore`, as demonstrated in Figure 2-4:

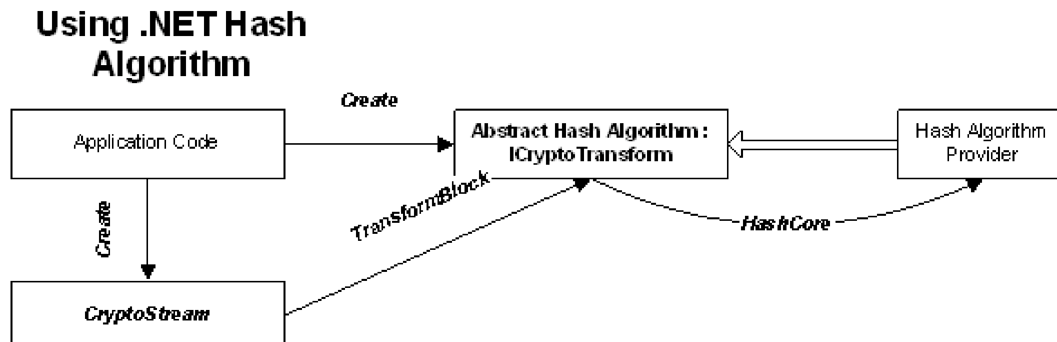

Figure 2-4. Streams with .NET hash algorithms

In Java, the following services are defined in the JCA framework (`java.security.*` packages), and Sun supplies two JCA providers ("SUN" and "RSAJCA") with J2SE v1.4.2:

- `MessageDigest`: data hashing algorithms (MD5, SHA-1).
- `Signature`: data signing and signature verification (DSA, RSAwithSHA1, RSAwithMD5).
- `KeyPairGenerator`: generation of public/private pair of keys for algorithms (DSA, RSA).
- `KeyFactory`: key conversions (DSA, RSA).
- `KeyStore`: managing keystores (JKS).
- `CertificateFactory`: certificate creation and CRL management (X.509).
- `AlgorithmParameters`: algorithms' parameter management, including their encoding (DSA).

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

- `AlgorithmParameterGenerator`: algorithms' parameter creation (DSA).
- `SecureRandom`: random numbers generators (SHA1PRNG).

As explained before, JCE has been separated out due to export restrictions. Its framework classes reside in `javax.crypto.*` packages and the Sun-supplied default provider "SunJCE" is shipped with J2SE v1.4.2:

- `Cipher`: objects carrying out encryption/decryption according to an algorithm, mode, or padding (AES, DES, 3DES, Blowfish, PBE). Java ciphers have the additional functionality of wrapping/unwrapping secret keys to make them suitable for transfer or storage. The implementation and algorithm varies by provider (this can be a PBE, for instance).
- `CipherStream`: combining input/output streams with a `Cipher` (`CipherInputStream`, `CipherOutputStream`).
- `KeyGenerator`: generating keys for symmetric algorithms and HMAC.
- `SecretKeyFactory`: conversions between key representations (AES, DES, 3DES, PBE).
- `SealedObject`: protecting a serialized object's confidentiality with a cryptographic algorithm.
- `KeyAgreement`: implementing Diffie-Hellman key agreement protocol (DH).
- `MAC`: producing cryptographically secured digests with secret keys (HMAC-MD5, HMAC-SHA1, PBE).

Additionally, Sun's provider supplies some JCA algorithms used by JCE: `KeyPairGenerator`; `AlgorithmParameterGenerator` for DH; `AlgorithmParameters` managers for DH, DES, 3DES, Blowfish, and PBE; and `KeyStore` implementation for "JCEKS".

The sample application java_encryption.zip demonstrates symmetric encryption and PBE to encrypt/decrypt a data file.

Surprisingly, however, many third-party providers (both commercial and free) provide a better selection of algorithms. For comparison, check the list of algorithms provided by an open source implementation from Bouncy Castle.

Note: Both platforms supply plenty of algorithms with default installations. There are quite a few independent Java vendors who offer even better selection than Sun's defaults.

Cryptography: Configuration

Cryptography systems on both platforms use configurable plug-in architectures — new algorithms, or updated implementations of existing ones can be added without code changes, by changing just few properties in the system configuration files.

A distinct feature of .NET's symmetric, asymmetric, and hash crypto family hierarchies (see the Algorithms section) is their configurability — all abstract classes in the hierarchies define static `Create` methods, allowing name-based lookup of the requested algorithm implementation in the *machine.config* file. New implementations may be mapped to existing (or new) names and will be picked up by the calls to the `Create` method, as explained later in this section. Classes of the `Cryptography` namespace that are not in those hierarchies do not follow this hierarchical approach and are not configurable by name.

At the heart of .NET Cryptography configuration lies the `CryptoConfig` utility class, which maps implementation classes to algorithm names, as configured in the *machine.config* file (or with the use of hardcoded defaults):

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
<cryptographySettings>
 <cryptoNameMapping>
  <cryptoClasses>
   <cryptoClass MySHA1Hash="MySHA1HashClass,
                MyAssembly Culture='en',
                PublicKeyToken=a5d015c7d5a0b012,
                Version=1.0.0.0"/>
  </cryptoClasses>
  <nameEntry
     name="SHA1" class="MySHA1Hash"/>
  <nameEntry
     name="System.Security.Cryptography.SHA1"
     class="MySHA1Hash"/>
  <nameEntry
name="System.Security.Cryptography.HashAlgorithm"
     class="MySHA1Hash"/>
 </cryptoNameMapping>
 <oidMap>
  <oidEntry OID="1.3.14.33.42.46" name="SHA1"/>
 </oidMap>
</cryptographySettings>
```

Application developers have the following choices when creating a configurable algorithm object:

- Invoke the `new` operator on the specific implementation class. This approach completely bypasses the .NET configuration mechanism.
- Call the `CryptoConfig.CreateFromName` method to map an abstract name to a specific algorithm implementation class.
- Using the factory pattern, call an overloaded static `Create` method on one of the abstract classes in the algorithm's family hierarchy (family root, or algorithm abstraction). Both overloads of `Create` will end up calling `CryptoConfig.CreateFromName` to retrieve the implementation class.

Continuing with the previous configuration example:

```
//all calls return an instance of MySHA1HashClass

HashAlgorithm sha1 =
  System.Security.Cryptography.SHA1.Create();

HashAlgorithm sha1 =
  System.Security.CryptoConfig.CreateFromName("SHA1");

HashAlgorithm sha1 =
  System.Security.Cryptography.HashAlgorithm.Create();
```

Configuration's `nameEntry` tags form a lookup table, which is consulted when `CryptoConfig.CreateFromName` is called. Any string can be used as a name, as long as it is unique (see "Specifying Fully Qualified Type Names" in the MSDN documentation for character restrictions). The OID mapping is optional; it allows mapping ASN.1 Object Identifiers to an algorithm implementation. If no algorithm-name configuration is specified, the following defaults are used. Note the following strong defaults for algorithm families:

- `System.Security.Cryptography.HashAlgorithm: SHA1CryptoServiceProvider`
- `System.Security.Cryptography.AsymmetricAlgorithm: RSACryptoServiceProvider`

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

- `System.Security.Cryptography.SymmetricAlgorithm: TripleDESCryptoServiceProvider`

In order to be usable after having been installed, Java's JCE providers should be made known to the runtime system. A `Provider` can be configured either declaratively in the `java.security` file:

```
// adding a crypto provider at the third position
security.provider.3=com.MyCompany.ProviderClassName
```

or programmatically by the code at runtime:

```
// appending a provider to the list
Security.addProvider(
          new com.MyCompany.ProviderClassName());
// adding a crypto provider at the third position
Security.insertProviderAt(
        new com.MyCompany.ProviderClassName(),3);
```

Programmatic runtime configuration assumes that the necessary permissions are granted to the executing code by the security policy (note that the providers themselves may require additional permissions to be specified):

```
// java.policy
// granting permissions for programmatic configuration
grant codeBase "file:/home/mydir/*" {
        permission java.security.SecurityPermission
                        "Security.setProperty.*";
        permission java.security.SecurityPermission
                     "Security.insertProvider.*";
        permission java.security.SecurityPermission
                     "Security.removeProvider.*";
}
```

Whether they were added declaratively or programmatically, all `Providers` end up in a single list and are queried for the requested algorithms (with optional parameters like mode and padding) according to their positions in the list (one being the highest) until finding a match. This process is shown in Figure 2-5. Algorithm and parameter names are hardcoded inside of the providers and cannot be changed. Developers can optionally request using only a particular provider, when they create an instance of an algorithm. This can be used, for example, when the developers want to use only particularly certified providers (for instance, DoD):

```
//requesting an implementation
//from only a single provider
Signature sigAlg1 = Signature.getInstance(
                "SHA1withDSA","MyGreatProvider");

//requesting the first matching implementation
Signature sigAlg2 = Signature.getInstance(
                "SHA1withDSA");
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

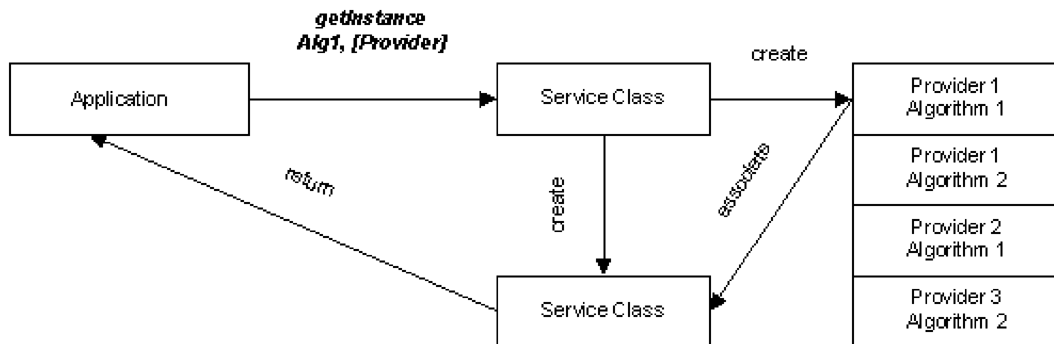

Figure 2-5. JCA collaborations

Note: Overall, both platforms are pretty even when it comes to configurability. Defaults for algorithm names are a convenient feature in .NET. Java, on the other hand, allows specifying additional algorithm details besides the name.

Secure Communication

During transmission, data can be protected on three levels: hardware, platform, and application. These can be used independently, or combined for better results. In all cases, there is some kind of cryptographic protection applied to the data prior to communication, but the amount of required application code and its complexity increases, with application-level solution being the most involved. While wire-level protocols (IPSec, for instance) may be implemented at the hardware level for speed and efficiency, they are not discussed here in order to keep the primary focus on the platforms themselves.

At the platform level, SSL is the de facto industry standard of transport protection. Both platforms support (to some extent) the latest SSL 3.0 specification that allows mutual authentication of both client and server. Recently, TLS 1.0 specifications were released by IETF (RFC 2246) as a new standard for Internet communication security, which is supposed to gradually replace SSL.

Additionally, both platforms expose — albeit at different levels — implementations of the `Generic Security Service API` (GSSAPI) (RFC 1508, 1509) common standard, which defines a generic programming interface for different authentication and communication protocols.

Secure Communication: Platform

Windows OS implements GSSAPI in the so-called `Security Support Provider Interface` (SSPI) to select one of the configured providers for securing data exchange over networked connections, which is used internally by .NET itself. However, as ridiculous as it sounds, .NET applications have only SSL configuration in IIS at their disposal for protection of HTTP-based traffic, while non-IIS based applications, such as standalone Remoting (the successor to DCOM) or HTTP servers, have no means of protecting their data en route. Not surprisingly, during the first year after .NET 1.0's release, protection of Remoting communication was one of the most frequently asked questions on the web forums.

There still exists no officially supported solution for securing Remoting communication, but fortunately, its highly flexible sink architecture allowed for the development of a number of low-level solutions that can be plugged into the infrastructure and server as a substitute for platform-level protection. Microsoft also apparently realized its omission, and released a fix in

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

the form of two assemblies in the samples namespace, `Microsoft.Samples`: `Security.SSPI` and `Runtime.Remoting.Security`. The former exposes a managed SSPI wrapper, and the latter uses it to implement a Remoting sink featuring symmetric encryption. Another article, which appeared at MSDN Magazine, outlined an alternative approach to Remoting security using asymmetric encryption.

The Java platform offers `Java Secure Socket Extensions` (JSSE) as a platform-level service for securing TCP/IP-based communication in vanilla J2SE applications, and J2EE's servlet specifications declare options for configuring SSL protection and refusing unprotected connection attempts.

Additionally, application servers from various vendors usually include some means to configure the SSL protocol for their HTTP servers. Since these are proprietary solutions, they are not going to be further pursued in this document.

JSSE, originally an extension to J2SE, was incorporated as a standard package as of version 1.4, so any Java application may count on using its services. The standard JSSE API is located in the `javax.NET.*` packages (`javax.security.cert` is obsolete and should not be used). It is quite rich; readers should consult the Javadocs for the specified packages and the online documentation for the class model and operation overview.

The example below shows a simple scenario of a client/server application, which will be satisfactory in most cases. Normal sockets are replaced with SSL ones by specifying different factory implementations, which are consequently used to obtain input/output streams:

```
//client establishing a connection
SSLSocketFactory clientFactory =
 (SSLSocketFactory)SSLSocketFactory.getDefault();
SSLSocket sslSocket = (SSLSocket)
clientFactory.createSocket(host,port);

//use as a normal socket
OutputStream out = sslSocket.getOutputStream();
InputStream in = sslSocket.getInputStream();

...

//server accepting a connection,
//requesting mutual authentication
SSLServerSocketFactory serverFactory =
  (SSLServerSocketFactory)
   SSLServerSocketFactory.getDefault();
SSLServerSocket ss = (SSLServerSocket)
serverFactory.createServerSocket(port);
ss.setNeedClientAuth(true);

//use as a normal socket
SSLSocket socket = ss.accept();
OutputStream out = socket.getOutputStream();
InputStream in = socket.getInputStream();

...
```

A connection between two peers in JSSE is represented by a `javax.NET.ssl.SSLSession` object. Among other things, this session contains negotiated shared secrets and information about ciphers used in the session. The master shared secret keys are not exposed through the JSSE API, and remain known only to the underlying implementation classes. Cipher

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

information, however, is available for analysis, and the server may refuse a connection if the client does not use strong enough ciphers:

```
SSLSocket socket = ss.accept();
SSLSession session = socket.getSession();
String cipher = session.getCipherSuite();
if (cipher.equals("SSL_RSA_WITH_RC4_128_SHA") ||
cipher.equals("SSL_RSA_WITH_RC4_128_MD5")) {
        //sufficient strength, may continue
        ...
} else {
        throw new SSLException(
            "Insufficient cipher strength!");
}
```

JSSE providers follow general JCE guidelines, and are pluggable into the provider-based JCA architecture. As a result, they may be configured in the *java.security* file, or added in code just like other security providers. Consequently, if a JCE provider, implementing the same algorithms as a JSSE one, is configured higher (i.e. it has a lower ordinal number — see JCE Providers Configuration) in the crypto providers list than the JSSE provider, JSSE operations will use the JCE provider's implementations instead of the built-in ones. Note, however, that the JSSE cryptography algorithm implementations are private and not available for public usage. Also, as a departure from the usual provider model due to export restrictions, the default `SSLSocketFactory` and `SSLServerSocketFactory` cannot be replaced.

In JDK 1.4.2, Sun provides a reasonably good reference JSSE implementation named "SunJSSE," whose features are highlighted below. For the complete list, check the JSSE guide.

- API and implementations of SSL 3.0 and TLS 1.0 algorithms.
- Stream-based I/O classes: `SSLSocket` and `SSLServerSocket`.
- One-way and mutual authentication. Certificate management is required and key and trust stores on both client and server should be set up appropriately.
- Implementation of HTTPS. Actually, JSSE services can be applied to many application-level protocols, such as RMI, FTP, LDAP, etc.
- Internal implementations for some cryptography algorithms.
- Read-only implementation of PKCS#12 keystore, in addition to the default `Java KeyStore` (JKS).
- Key and trust store management. For easier control, JSSE defines several system properties to control behaviors of appropriate classes from the command line.

The following properties (all starting with `javax.NET.ssl`) may be specified on the command line or set in code: `keyStore`, `keyStorePassword`, `keyStoreType`, `trustStore`, `trustStorePassword`, and `trustStoreType`. For example:

```
java -Djavax.NET.ssl.trustStore=AppTrustStore SecureApp
```

Java applications using RMI communication are pretty much limited to using JSSE for protection. Sun is working on separate specifications for secure RMI, which will include authentication, confidentiality, or integrity mechanisms, but they are not going to be available any time soon — the JSR 76 "RMI Security for J2SE" was rejected in February 2001. Custom solutions are possible (such as subclassing `SSLServerSocket`), but they are non-trivial.

The J2EE specification promotes usage of SSL/TLS across all of its components by mandating

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

support for the following ciphers:

- `TLS_RSA_WITH_RC4_128_MD5`
- `SSL_RSA_WITH_RC4_128_MD5`
- `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
- `SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
- `TLS_RSA_EXPORT_WITH_RC4_40_MD5`
- `SSL_RSA_EXPORT_WITH_RC4_40_MD5`
- `TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`

In most cases, cipher suites will be either `SSL_RSA_WITH_RC4_128_SHA` or `SSL_RSA_WITH_RC4_128_MD5` (or their TLS equivalents), as they are currently the strongest commonly used SSL ciphers.

The servlet specification defines the `transport-guarantee` element in the deployment descriptor, which is used to require a certain level of call protection from the servlet container. Possible values are: `NONE`, `INTEGRAL`, and `CONFIDENTIAL`, and can be specified in the */WEB-INF/web.xml* file. The names for the constraints are pretty self-descriptive, and implementation interpretation is left at the vendor's discretion. However, the servlets have an option to programmatically reject a HTTP connection if the `HttpRequest.isSecure` method shows that the connection is not secure. Below is an example of specifying the transport guarantee element:

```
<security-constraint>
  <user-data-constraint>
    <transport-guarantee>
      CONFIDENTIAL
    </transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

EJBs do not have an option to determine connection's security settings. EJB specifications, however, require passive support for remote calls' security; i.e., if a call is made using a secure connection, EJB server also uses a secure connection for further calls. Remote EJB calls use the IIOP protocol, with support for SSL 3.0 and TLS 1.0 support mandated by EJB 2.0 and J2EE specifications.

Note: Besides IIS, .NET does not offer any standard means for communication protection at the platform level, while Java has a complete solution in this space.

Secure Communication: Application

For finer control over applied security mechanisms, an application can use an application-level, token-based protection mechanism, abstract from the underlying transmission protocol. This approach has an advantage over channel blanket encryption by being smarter and protecting only sensitive data. For instance, web services (see later in this section) use this paradigm for message protection, where only particular details of messages are signed and encrypted.

As already explained, J2SE includes GSSAPI, which may be utilized on the application level to provide token-based protection using the Kerberos V 5. GSSAPI framework, is quite a thin wrapper, delegating all requests to the underlying mechanism providers. The Java GSS mechanisms do not perform user logins themselves — they should be done using JAAS prior

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

to invoking GSSAPI services, and the credentials should be stored in some cache accessible to the GSS mechanism provider. Using JAAS and Kerberos tickets in GSS provides not only transport-level security protection, but also a principal delegation mechanism over the network. See "Authentication" in Part 4 for more information about JAAS and delegation.

GSS classes reside in the `org.ietf.jgss` package; check the online documentation for details and code examples.

Overall, Java offers a choice of platform-level (JSSE) and application-level (GSS) protection services with similar security goals: client-server authentication and protection of transmitted data. Listed below are a few criteria that can help to decide which service is more appropriate for a particular application:

- JSSE is very easy to use from client's code — no action besides establishing the proper socket factory is needed. GSS setup and coding are significantly more involved.
- Java's "Single Sign-On" mechanism is based on Kerberos V5, which is supported only by GSS.
- JSSE implementations are socket-based and typically use TCP as the underlying protocol. GSS is token-based and can use any communication channel for token transmission — the code is responsible for establishing the channel, though.
- GSS is capable of client credential delegation.
- JSSE encrypts all data sent through the socket. GSS, being token-based, can encrypt tokens selectively, thus significantly lowering computational load.
- JSSE implements TLS 1.0 and SSL 3.0 communication protocols. GSS supports only the Kerberos V5 protocol (known as "SSPI with Kerberos" on Win32), and provides implementation of IETF's generic GSS-API framework.

Web services security specifications and toolkits also look at protecting individual messages, or tokens. This area has been rapidly evolving, and has not yet been fully standardized. Because of this lack of standards, both platforms provide only partial support for it via extensions or external products. However, since the topic of web services security alone warrants a whole separate book, it is not addressed here in any significant detail. Of all competing standards in this area, only SAML and WS-Security have been so far accepted for standardization by OASIS, with the latter still undergoing committee reviews.

For web services security, Microsoft has been actively promoting its `Web Service Architecture` (WSA, formerly GXA), and adding support for all released up-to-date specifications via its `Web Services Extension` (WSE) pack for .NET. WSE is currently at 1.0 release, with 2.0 coming soon — check the MSDN documentation for updates and new releases. Notably, WSE (and .NET in general) lacks support for SAML, even though the WS-Security specification does define binding for SAML assertions as one of the supported token types. In other areas, WSE provides relatively complete support of WS-Security and a number of other specifications. Additionally, WSE's certificate classes (located in the `Microsoft.Web.Services.Security.X509` package) are much more convenient to deal with than .NET's original ones. The code sample below shows how to sign a request using WSE:

```
// Get SOAP context from the Web service proxy
SoapContext reqCxt =
               serviceProxy.RequestSoapContext;

// Retrieve the certificate to be used for signing
Microsoft.Web.Services.Security.X509.X509Certificate crt = ...;
// Create a X509 security token
X509SecurityToken token =
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
                    new X509SecurityToken(crt);

// Sign the request by adding
//a signature to the request
reqCxt.Security.Tokens.Add(token);
reqCxt.Security.Elements.Add(
                      new Signature(token));

// Use the signed request to call the service...
serviceProxy.Hello();
```

The extensible architecture of .NET's Remoting has allowed for the development of quite interesting approaches to transport security, whereas Remoting's RPC-style invocations are transmitted and protected by the means of SOAP-based web service messages. In principle, this is not very different from the Microsoft solution described earlier, but it allows applying WSA-family protection (in particular, WS-Security) to individual messages, which ensures standard-based authentication, integrity, and authorization at the message level, as opposed to the non-standard approach of blank encryption of the former solution. For explanations and code samples, read the excellent publications at the CodeProject web site, in particular "Remoting over Internet" and related articles.

The Java platform does not provide direct support for web services security yet. Currently, there are two web services security-related JSRs at work: JSR 155 "Web Services Security Assertions", and JSR 183 "Web Services Message Security APIs". When accepted (although they have been in review stage for over a year now), these specifications should provide assertions support and transport-level security to web services written in Java. Although not a standard part of Java platform, IBM's Emerging Technologies Toolkit v1.2 (ETTK), formerly known as "Web Services Toolkit", or WSTK, adds support for the current draft of WS-Security and some other specifications from the WSA family, of which IBM is a co-author.

Note: The .NET platform stays very current with the latest developments in web services security, while their support in Java is not standardized and is limited to offerings from individual vendors.

Chapter 2 – Conclusions

In this section, cryptography and communication protection on Java and .NET platforms were reviewed. Both platforms come out pretty even in terms of cryptographic features, although Java has a more complicated solution due to the obsolete US export restrictions. The picture becomes muddier when it comes to communication protection — while Java fares much better by providing a choice of both platform and application-level solutions, it clearly lags behind .NET when it comes to support for web services security. Here, Java developers would have to turn to independent vendors for the desired features.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Chapter 3 – Code Protection and Code Access Security (CAS)

Once code or an algorithm has been written, it becomes an asset that requires protection. Such a protection is needed not only against theft, but also against unauthorized or unintended use. On the other hand, when somebody purchases a software package, he wants to be confident that he is not dealing with a counterfeit product. To answer all of these challenges, various techniques, broadly divided into cryptography-based and everything else, are employed for code protection and verification.

Code-access security is also known as policy-based security. It allows minimizing the risks of executing certain application code by providing policies restricting it to only a particular, well-defined set of operations that the code is permitted to execute. Of course, the underlying services (platform or application) have to actually carry out those checks for the policy to become effective.

Code Protection: General

Issues discussed in this section are applicable, to a certain degree, to both platforms. They also employ similar protection mechanisms to combat those problems.

The possibility of the reverse engineering of distributed bytecodes needs to be taken into account when planning the security aspects of an application, because bytecode formats are well-documented for both Java and .NET (see also GotDotNet), so any hardcoded data or algorithms may be easily restored with readily obtainable decompiling tools. This point is especially important for avoiding hardcoding user credentials or non-patented algorithms in client-based application modules.

While there is no ideal way around this issue, short of shipping encrypted code and providing just-in-time decryption, an average perpetrator's task may be made harder by using so called *obfuscators*; i.e., tools that intentionally scramble bytecodes by using unintelligible names and moving entry points around. In addition to the obfuscator tool available with VS.2003, a number of decompiling/obfuscating tools can be found at the Microsoft web site. For Java, a great number of commercial or free Java decompilers and obfuscators can be found by running a simple search on the Web.

Finally, OS-level protection mechanisms need to be utilized, along with the platform ones, in order to ensure good protection of the stored data. All of the hard work at the platform level is useless if code or data can be obtained and inspected as raw binary files. Therefore, any normal OS operation security rules (like `ACL`, minimizing attack surface, the principle of "least privilege," etc.) should be employed in addition to the platform-specific ones, in order to ensure blanket protection.

Certificate Management

Before addressing cryptography-based code protection features, the issue of certificate management in general needs to be covered, because all cryptography-based solutions deal, in one way or another, with certificates or keys. First of all, certificates need to be created and stored, and then accessed from the applications. Both platforms supply tools to issue certificate requests, as well as APIs for accessing the stored certificates.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

.NET, as usual, heavily relies on Windows certificate stores to deal with certificates they are used to store X509 certificates and certificate chains of trusted signers. There are a number of tools included with the .NET SDK to help accessing certificate stores, manage certificates, and sign assemblies with publisher certificates.

.NET's Certificate API is represented in its `System.Security.Cryptography.X509Certificates` namespace, where the `X509Certificate` class is of particular interest to us. Unfortunately, this class is rather poorly designed; it does not support accessing certificate stores, but works only with certificates in binary *ASN.1 DER* format, and does not provide any way to use certificates in asymmetrical encryption. The official suggestion from Microsoft is to stick with using unmanaged `CryptoAPI` (CAPI) functions, in particular `CryptExportKey/CryptImportKey`. See MSDN articles for details of bridging .NET's managed certificate implementation with CAPI.

Another, much better alternative is using WSE (already covered in Part 2). It provides the `Microsoft.Web.Services.Security.X509` namespace with several useful classes, among them another version of `X509Certificate`, derived from the .NET-supplied one. This class recognizes the shortcomings of its predecessor and provides a very convenient interface for accessing certificate stores, as well as extracting public/private key information in a format appropriate for asymmetric encryption. As an added benefit, it can read certificates stored in `Base64` text format. Together with the `X509CertificateStore` class, they make .NET's certificate API pretty well rounded. The following MSDN example shows how they can be used together:

```
// Open and read the Personal certificate store for
// the local machine account.
X509CertificateStore myStore =
  X509CertificateStore.LocalMachineStore(
  X509CertificateStore.MyStore);
myStore.OpenRead();

// Search for all certificates named "My Certificate"
// add all matching certificates
// to the certificate collection.
X509CertificateCollection myCerts =
  myStore.FindCertificateBySubjectString(
          "My Certificate");
X509Certificate myCert = null;

// Find the first certificate in the collection
// that matches the supplied name, if any.
if (myCerts.Count > 0)
{
  myCert = myCerts[0];
}

// Make sure that we have a certificate
// that can be used for encryption.
if (myCert == null ||
    !myCert.SupportsDataEncryption)
{
  throw new ApplicationException(
    "Service is not able to encrypt the response");
  return null;
}
```

The Java platform implements RFC 3280 in the `Certification Path API`, which is supplied

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

in the default “SUN” provider. This API, however, allows read-only retrieving, accessing attributes, etc. access to certificates, because they are considered to be immutable entities in Java. Classes implementing `Certification Path API` belong to the `JCA` framework and can be found in the `java.security.cert` package. There are three classes of interest there:

- `Certificate`: An abstract class for dealing with certificates.
- `X509Certificate`: An abstract class for dealing specifically with X.509 certificates, stored using `Base64` encoding, with `BEGIN CERTIFICATE/END CERTIFICATE` markers serving as delimiters.
- `CertificateFactory`: A factory for generating certificate objects from their encoded formats.

Java uses so-called *keystores* for storing certificates. They can have different formats, as supplied by JCA providers (see Part 2); the default is Sun’s proprietary JKS format. There are common *keystores* that contain keys, both public and private (or symmetric, if desired), and *truststores*, which are used to establish certificate chains. The JVM uses truststores (*lib/security/cacert* by default) to store the trusted certificates, and keystores for accessing key information. Having keystores as separate files is a nice feature in Java, as it is easy to move them around and manage them (compared to .NET’s reliance on CAPI containers). Both stores can be specified as parameters on the command line, or accessed directly from code:

```
java -Djavax.NET.ssl.keyStore=MyKeyStore
    -Djavax.NET.ssl.keyStorePassword=password
    -Djavax.NET.ssl.trustStore=MyTrust MyClass
```

Compared to the standard .NET certificate implementation, Java provides very convenient facilities of working with certificates. The examples below demonstrate how easy it is to obtain certificates from a keystore:

```
FileInputStream fin = new FileInputStream("MyKeyStore");
KeyStore ks = KeyStore.getInstance("JKS");
ks.load(fin,"password");
Certificate cert = ks.getCertificate("MyEntry");
```

or from a file:

```
FileInputStream fin =
            new FileInputStream("MyCert.cer");
CertificateFactory factory =
            CertificateFactory.getInstance("X.509");
X509Certificate cert = (X509Certificate)
            factory.generateCertificate(fin);
```

Java provides a tool for generating keys, `keytool`, which has a number of options. Among them are importing/exporting certificates, creating test certificates, creating certificate signing requests, etc. This tool picks up the keystore types (*JKS, PKCS#12*, etc.) defined in the *java.security* configuration file, and can use plugged-in provider types to operate on various types of *keystores*. By default, `keytool` generates only *X.509 v1* certificates, which may be restricting for some applications.

Becoming a *Certificate Authority* (CA) on your own is problematic, but not impossible, with Java. One can either purchase a commercial library, or build his or her own CA using `sun.security.x509` classes, although they only work with JKS keystores. However, the latter

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

solution is neither portable nor documented. There are also a number of open source libraries that allow you to deal with certificate management, including CA functionality. A good free implementation is OpenSSL.

Note: Java provides a solid API for dealing with certificates. .NET programmers have to turn to unmanaged CAPI functions to access certificates, unless they use WSE, which adds a lot of useful functionality.

Code Protection: Cryptographic

Cryptography-based mechanisms include certificates, digital signatures, and message digests, which are used to “shrink-wrap” distributed software and data. They establish software origin and verify its integrity with a high degree of reliability, subject to the strength of the underlying cryptography algorithm.

In their simplest forms, CRC or digests are used to verify software integrity. For more involved implementations, *Message Authentication Code* (MAC), or *Hash-based MAC* (HMAC), specified in RFC 2104, may be applied, which add cryptographic protection (using symmetric secret keys) for improved protection. Both platforms support most common digest, MAC, and HMAC functions in their respective cryptography namespaces. See Part 2 for details of supported algorithms. Numerous code samples are available on the Web for both Java and .NET (also see MSDN).

Sample applications for .NET and Java are provided as NET.III.DataSigning.zip and Java.III.DataSigning.zip respectively.

For application distribution, .NET supports software signing to prove application or publisher identities. For the first task, it provides so-called *strong names*, and for the second, signing with publisher certificates. These approaches are complementary and independent; they can be used individually or jointly, thus proving the identities of both the application and publisher. The users can configure .NET CAS policy based either on strong names, or on the software publisher, because they both provide strong assurances about the signed code. Because of their high levels of trust, strong-named assemblies can call only strong-named assemblies.

Strong names are used to prove authenticity of the assembly itself, but they have no ties to the author, as it is not required to use the developer’s certificate to sign an assembly. A strong name can be viewed as a descendent of GUID mechanism, applied to the assembly names, and composed of text name, version number, culture information, plus the assembly’s digital signature and a unique public key, all stored within the assembly’s manifest, as shown in Figure 3-1:

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
MANIFEST
  //                                                                         boo
  .publickey = (00 24 00 00 04 80 00 00 94 00 00 00 06 02 00 00   // .$..............
                00 24 00 00 52 53 41 31 00 04 00 00 01 00 01 00   // .$..RSA1........
                BB CC 2B 24 5D AD 8E 7D 2F 1D 1D F2 41 27 A7 49   // ..+$]..}/...A'.I
                83 A2 F0 A9 76 90 97 DA 5E 93 C8 CC 08 7E 76 57   // ....v...^....~vW
                CE 1B 25 82 82 69 E3 05 8D 4D 19 03 2F AF 48 F9   // ..%..i...M../.H.
                00 03 ED 0F 74 F5 E3 8C B3 31 C0 B3 D0 DD 41 F9   // ....t....1....A.
                23 38 63 E7 33 6F D8 D5 85 5B 1D 76 30 FB 01 9F   // #8c.3o...[.v0...
                61 CF DD E9 30 6C 1E C5 F2 CB FC FE F4 1B C8 C2   // a...0l..........
                E0 D0 38 6E 0D 8A 5F 33 1F 85 F1 A8 A1 41 E2 35   // ..8n.._3.....A.5
                4A E9 39 CE 5D 53 AD FC D3 49 DE 32 ED 01 76 DD ) // J.9.]S...I.2..v.
  .hash algorithm 0x00008004
  .ver 1:0:1382:15506
}
.module CommonLibrary.dll
// MVID: {9DB58A5A-EEDE-4C71-AE36-AFC8F6B84E97}
.imagebase 0x11000000
.subsystem 0x00000003
.file alignment 4096
.corflags 0x00000009
// Image base: 0x06d00000
```

Figure 3-1. *Manifest of a Strong-Named Assembly*

The same key pair should not be reused for signing multiple assemblies unique key pairs should be generated for signing different assemblies. Note that a version is not guaranteed by the strong name due to applying CLR versioning policy, as the same key will be reused to sign a newer version of the particular assembly. .NET provides a tool named `sn.exe` to generate key pairs and perform a number of validation steps related to strong names. Generated keys can either be picked up by `AssemblyLinker`, or added to the assembly declaratively by using an attribute:

```
[assembly: AssemblyKeyFile(@"CommonLib.snk")]
```

Clients, linked against a strong-named assembly, store a `PublicKeyToken` representing the assembly, an eight-byte hash of the full public key used to create its strong name, as shown in Figure 3-2. This is done transparently by the compiler when a strong-named assembly is referenced. This is an example of *early binding*.

```
MANIFEST
  .ver 1:0:3300:0
}
.assembly extern System
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:3300:0
}
.assembly extern CommonLibrary
{
  .publickeytoken = (F5 BE 3B 7C 2C 52 3B 5D )
  .ver 1:0:1385:17444
}
.assembly extern KeyMgmtLib
{
  .publickeytoken = (51 E0 EC D2 A6 DC 48 02 )
  .ver 1:0:1441:18468
}
.assembly extern ZipEncryptLib
{
  .publickeytoken = (8C 57 60 82 7B CD F2 8A )
  .ver 1:0:1441:18468
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Figure 3-2. Manifest with Public Key Token

Late binding can be achieved by using `Assembly.Load` and calling the full *display name* of the assembly, including its token:

```
Assembly.Load("CommonLibrary,
   Version= 1:0:1385:17444,
   Culture=neutral,
   PublicKeyToken=F5BE3B7C2C523B5D");
```

The CLR always verifies an assembly's key token at runtime, when it loads the referenced assembly. Two strong names are configured in security policy by default: one for Microsoft code, another for .NET's components submitted to ECMA for standardization.

Publisher certificates establish the identity of the code distributor by requiring him to use a personal certificate to sign a whole assembly or individual files in an assembly. The signature is then stored inside of the file and verified by the CLR at runtime. .NET provides the `Signcode.exe` tool to perform the publisher signing operation. To do its job, it should have access to both the publisher certificate (with a valid trust chain), and the private key for that certificate, which will be used to sign the file(s).

Publisher certificates, as opposed to the strong names concept, are used to sign multiple applications and cannot uniquely identify a particular code module, nor are they intended to. Their intent is to identify a broad set of applications as originating from a particular vendor, in order to assign appropriate permissions based on the level of trust in this company.

As far as signing distributives goes, Java offers a single route that is similar the publisher-signing paradigm in .NET. However, there are significant differences in the approaches, since JAR specifications permit multiple signers and signing of a subset of the JAR's content.

Quite surprisingly, in Sun's default distribution of JDK 1.4.2, only `jce.jar` is signed — all of the other libraries do not have any signature. As a standard part of the JDK, Sun ships the `jarsigner` tool, which works with Java keystores to obtain private keys and certificates for signing and verification to validate certification chains. This tool operates on existing JAR files (it does not create them), which are a standard distribution format in Java.

```
jarsigner -keystore PrivateKeystore.jks
          -sigfile DP -signedjar DemoApp_Sig.jar
          DemoApp.jar denis
```

When a JAR file is signed, its *manifest* is updated and two new files are added to the *META-INF* directory of the archive (see the JAR Guide for details): *class signature* and *signature block*.

The *manifest* file is called *MANIFEST.MF* and contains digest records of all signed files in the archive (which may be a subset of the archive!). Those records conform to the RFC 822 header specification and consist of a file name and one or more tuples (digest algorithm, file digest). As a rule, either `SHA1` or `MD5` digest algorithms are used. It is the manifest itself, not the physical JAR file, that is signed, so it is important to understand that once a JAR is signed, its manifest should not change -- otherwise, all signatures will be invalidated.

```
Manifest-Version: 1.0
Created-By: 1.4.2-beta (Sun Microsystems Inc.)

Name: javax/crypto/KeyGeneratorSpi.class
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
SHA1-Digest: HxiOMRd8iUmo2/fulEI1QH7I2Do=

Name: javax/crypto/spec/DHGenParameterSpec.class
SHA1-Digest: zU+QpzVweIcLXLjmHLKpVo55k0Q=
```

A *signature* file represents a signer, and an archive contains as many of these files as there are signatures on it. File names vary, but they all have the same extension, so they looks like `<Name>.SF`. Signature files contain "digests of digests" they consist of entries with digests of all digest records in the manifest file at the time of signing. Those records conform to RFC 822 header specification and have the same format as the manifest's ones. Additionally, this file also contains a digest for the entire manifest, which implies that the JAR manifest may not be changed once signed. Incidentally, this means that all signers have to sign the same set of files in the JAR otherwise, if new files have been added to the JAR prior to generating another signature, their digests will be appended to the manifest and invalidate already existing signatures.

An important point to observe is that when a JAR file is signed, all of the files inside it are signed, not only the JAR itself. Up until JDK 1.2.1, signed code had a serious bug: it was possible to alter or replace the contents of a signed JAR, and the altered class was still allowed to run. This problem was rectified starting with version 1.2.1 by signing each class inside of a signed JAR.

```
Signature-Version: 1.0
Created-By: 1.4.2-beta (Sun Microsystems Inc.)
SHA1-Digest-Manifest: qo3ltsjRkMm/qPyC8xrJ9BN/+pY=

Name: javax/crypto/KeyGeneratorSpi.class
SHA1-Digest: FkNlQ5G8vkiE8KZ8OMjP+Jogq9g=

Name: javax/crypto/spec/DHGenParameterSpec.class
SHA1-Digest: d/WLNnbH9jJWc1NnZ7s8ByAOS6M=
```

A *block signature* file contains the binary signature of the *SF* file and all public certificates needed for verification. This file is always created along with the *SF* one, and they are added to the archive in pairs. The file name is borrowed from the *signature* file, and the extension reflects the signature algorithm (RSA|DSA|PGP), so the whole name looks like `<Name>.RSA`.

The JAR-signing flexibility comes from separating digest and signature generation, which adds a level of indirection to the whole process. When signing or verifying, individual signers operate on the manifest file, not the physical JAR archive, since it is the manifest entries that are signed. This allows for an archive to be signed by multiple entities and to add/delete/modify additional files in the signed JAR, as long as it does not affect the manifest (see the explanations in the *signature* file paragraph).

Note: Strong names in .NET offer an improved approach to versioning. JAR files, on the other hand, have more options for signing, so this category is a draw.

Code Protection: Non-Cryptographic

Once a piece of software's origin and integrity have been established, non-cryptographic approaches may be used to ensure that the code can not be used in an unintended manner. In particular, this implies that the platform's package- and class-protection mechanisms cannot be subverted by illegally joining those packages or using class derivation to gain access to protected or internal members. These types of protection are generally used to supplement

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

CAS and aimed at preventing unauthorized execution or source code exposure.

Reflection mechanisms on both platforms allow for easy programmatic access to code details and very late binding of arbitrary code, or even utilize code generation capabilities -- see, for instance,.NET technology samples, or the Java Reflection tutorial. One common example of such a threat would be a spyware application, which secretly opens installed applications and inspects/executes their functionality in addition to its officially advertised function. To prevent such code abuse, granting reflection permissions (`System.Security.Permissions.ReflectionPermission` in .NET, `java.lang.reflect.ReflectPermission` in Java) in CAS policy should be done sparingly and only to highly trusted code, in order to restrict capabilities for unauthorized code inspection and execution.

In .NET, application modules are called *assemblies*, and located at runtime by a so-called *probing algorithm*. By default, this algorithm searches for dependent assemblies only in the main assembly's directory, its subdirectories, and the `Global Assembly Cache` (GAC). Such a design is used to guard against possible attempts to access code outside of the assembly's "home." Note that it does not prevent loading and executing external assemblies via reflection, so CAS permissions should be applied as well.

Types in .NET are organized into *namespaces*. One may extend an already established namespace in his own assemblies, but will not gain any additional information by doing so, since the keyword `internal` is applied at the assembly, and not namespace, level. *Strong names* are used as a cryptographically strong measure against replacement of the existing types.

If the designer wants to completely prohibit inheritance from a class or method overloading, the class or method may be declared `sealed`. As an additional means of protection against source browsing, the C# language defines a `#line hidden` directive to protect against stepping into the code with a debugger. This directive instructs the compiler to avoid generating debugging information for the affected area of code.

During execution of a Java application, *class loaders* are responsible for checking at loading/verification time that the loaded class is not going to violate any package protection rules (i.e., does not try to join a `sealed` or `protected` package). Particular attention is paid to the integrity of system classes in `java.*` packages — starting with version 1.3, the class-loading delegation model ensures that these are always loaded by the *null*, or *primordial class loader* (see "Secure Class Loading" for details).

The Java platform defines the following options for protecting packages from joining:

- Sealed JAR files

  These are used to prevent other classes from "joining" a package inside of that JAR, and thus obtaining access to the protected members. A package, `com.MyCompany.MyPackage.*`, may be sealed by adding a `Sealed` entry for that package to the JAR manifest file before signing it:

  ```
  Name: com/MyCompany/MyPackage/
  Sealed: true
  ```

- Configuration restrictions for joining packages

  These can be used to control which classes can be added to a restricted package by adding them to the *java.security* file. Note that none of Sun-supplied class loaders performs this check (due to historical reasons), which means that this protection is

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

only effective with a custom class loader installed:

```
# List of comma-separated packages that start
# with or equal this string will cause a security
# exception to be thrown when passed to
# checkPackageDefinition unless the corresponding
# RuntimePermission ("defineClassInPackage."+package)
# has been granted.

#

# by default, no packages are restricted for
# definition, and none of the class loaders
# supplied with the JDK call checkPackageDefinition.

package.definition=com.MyCompany.MyPackage.private
```

- Configuration restrictions for accessing packages

  If `SecurityManager` is installed, it checks the package-access policies defined in `java.security` file. A package can have restricted access so that only classes with appropriate permissions can access it. For instance, all `sun.*` packages are restricted in the default installation:

```
# List of comma-separated packages that start
# with or equal this string will cause a security
# exception to be thrown when passed to
# checkPackageAccess unless the corresponding
# RuntimePermission ("accessClassInPackage."+package)
# has been granted.

package.access=sun.
```

A sample Java application, demonstrating declarative access control to packages, is provided as Java.III.PackageChecks.zip.

Note: Configuration options in Java add a convenient method for declarative code protection, which gives it a slight edge over .NET in this category.

Code Access Security: Permissions

Code-access permissions represent authorization to access a protected resource or perform a dangerous operation, and form a foundation of CAS. They have to be explicitly requested from the caller either by the system or by application code, and their presence or absence determines the appropriate course of action.

Both Java and .NET supply an ample choice of permissions for a variety of system operations. The runtime systems carry out appropriate checks when a resource is accessed or an operation is requested. Additionally, both platforms provide the ability to augment those standard permission sets with custom permissions for protection of application-specific resources. Once developed, custom permissions have to be explicitly checked for (demanded) by the application's code, because the platform's libraries are not going to check for them.

.NET defines a richer selection here, providing permissions for role-based checks (to be covered in the "User Access Security" section of Part 4) and evidence-based checks. An interesting feature of the latter is the family of *Identity* permissions, which are used to identify

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

an assembly by one of its traits -- for instance, its *strong name* (`StrongNameIdentityPermission`). Also, some of its permissions reflect close binding between the .NET platform and the underlying Windows OS (`EventLogPermission, RegistryPermission`). `IsolatedStoragePermission` is unique to .NET, and it allows low-trust code (Internet controls, for instance) to save and load a persistent state without revealing details of a computer's file and directory structure. Refer to MSDN documentation for the list of .NET Code Access and Identity permissions.

Adding a custom code access permission requires several steps. Note that if a custom permission is not designed for code access, it will not trigger a stack walk. The steps are:

- Optionally, inherit from `CodeAccessPermission` (to trigger a stack walk).
- Implement `IPermission` and `IUnrestrictedPermission`.
- Optionally, implement `ISerializable`.
- Implement XML encoding and decoding.
- Optionally, add declarative security support through an `Attribute` class.
- Add the new permission to CAS Policy by assigning it to a code group.
- Make the permission's assembly trusted by .NET framework.

A sample of custom code-access permission can be found in the NET.III.CodePermissions.zipdemo application. Also, check MSDN for additional examples of building and registering a custom permission with declarative support.

.NET permissions are grouped into `NamedPermissionSets`. The platform includes the following non-modifiable built-in sets: `Nothing, Execution, FullTrust, Internet, LocalIntranet, SkipVerification`. The `FullTrust` set is a special case, as it declares that this code does not have any restrictions and passes **any** permission check, even for custom permissions. By default, all local code (found in the local computer directories) is granted this privilege.

The above fixed permission sets can be demanded instead of regular permissions:

```
[assembly:PermissionSetAttribute(
         SecurityAction.RequestMinimum,
         Name="LocalIntranet")]
```

In addition to those, custom permission sets may be defined, and a built-in `Everything` set can be modified. However, imperative code-access checks cannot be applied to varying permission sets (i.e., custom ones and `Everything`). This restriction is present because they may represent different permissions at different times, and .NET does not support dynamic policies, as it would require re-evaluation of the granted permissions.

Permissions, defined in Java, cover all important system features: file access, socket, display, reflection, security policy, etc. While the list is not as exhaustive as in .NET, it is complete enough to protect the underlying system from the ill-behaving code. See the JDK documentation for the complete list, and the Java permissions guide for more detailed discussions of their meaning and associated risks.

Developing a custom permission in Java is not a complicated process at all. The following steps are required:

- Extend `java.security.Permission` or `java.security.BasicPermission`.
- Add new permission to the JVM's policy by creating a `grant` entry.

Obviously, the custom permission's class or JAR file must be in the `CLASSPATH` (or in one of

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

the standard JVM directories), so that JVM can locate it.

Below is a simple example of defining a custom permission. More examples can be found in the Java.III.CodePermissions.zip demo application or in the Java tutorial:

```
//permission class
public class CustomResPermission extends Permission {
  public CustomResPermission (String name,
                              String action) {
  super(name,action);
  }
}

//library class
public class AccessCustomResource {
  public String getCustomRes() {
    SecurityManager mgr =
         System.getSecurityManager();
    if (mgr == null) {
          //shouldn't run without security!!!
          throw new SecurityException();
    } else {
          //see if read access to the resource
          //was granted
          mgr.checkPermission(
        new CustomResPermission("ResAccess","read"));
    }
   //access the resource here
   String res = "Resource";
   return res;
  }
}

//client class
public class CustomResourceClient {
  public void useCustomRes() {
    AccessCustomResource accessor =
     new AccessCustomResource();
    try {
      //assuming a SecurityManager has been
      //installed earlier
      String res = accessor.getCustomRes();
    } catch(SecurityException ex) {
      //insufficient access rights
    }
  }
}
```

J2EE reuses Java's permissions mechanism for code-access security. Its specification defines a minimal subset of permissions, the so-called *J2EE Security Permissions Set* (see section 6.2 of the J2EE.1.4 specification). This is the minimal subset of permissions that a J2EE-compliant application might expect from a J2EE container (i.e., the application does not attempt to call functions requiring other permissions). Of course, it is up to individual vendors to extend it, and most commercially available J2EE application servers allow for much more extensive application security sets.

Note: .NET defines a richer sets-based permission structure than Java. On the other hand, .NET blankly grants `FullTrust` to all locally installed code, and its permissions structure reflects the platform's close binding to Windows OS.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Code Access Security: Policies

Code Access Security is *evidence*-based. Each application carries some *evidence* about its origin: location, signer, etc. This *evidence* can be discovered either by examining the application itself, or by a trusted entity: a class loader or a trusted host. Note that some forms of evidence are weaker than others, and, correspondingly, should be less trusted -- for instance, *URL evidence*, which can be susceptible to a number of attacks. *Publisher evidence*, on the other hand, is `PKI`-based and very robust, and it is not a likely target of an attack, unless the publisher's key has been compromised. A *policy*, maintained by a system administrator, groups applications based on their evidence, and assigns appropriate permissions to each group of applications.

Evidence for the .NET platform consists of various assembly properties. The set of assembly evidences, which CLR can obtain, defines its group memberships. Usually, each evidence corresponds to a unique `MembershipCondition`, which are represented by .NET classes. See MSDN for the complete listing of standard conditions. They all represent types of evidence acceptable by CLR. For completeness, here is the list of the standard evidences for the initial release: `AppDirectory, Hash, Publisher, Site, Strong Name, URL,` and `Zone`.

.NET's policy is hierarchical: it groups all applications into so-called *Code Groups*. An application is placed into a group by matching its *Membership Condition* (one per code group) with the *evidence* about the application's assembly. Those conditions are either derived from the evidence or custom-defined. Each group is assigned one of the pre-defined (standard or custom) `NamedPermissionSet`. Since an assembly can possess more than one type of *evidence*, it can be a member of multiple code groups. In this case, its total permission set will be a union of the sets from all groups (of a particular level) for which this assembly qualifies. Figure 3-3 depicts code-group hierarchy in the default machine policy (also see MSDN):

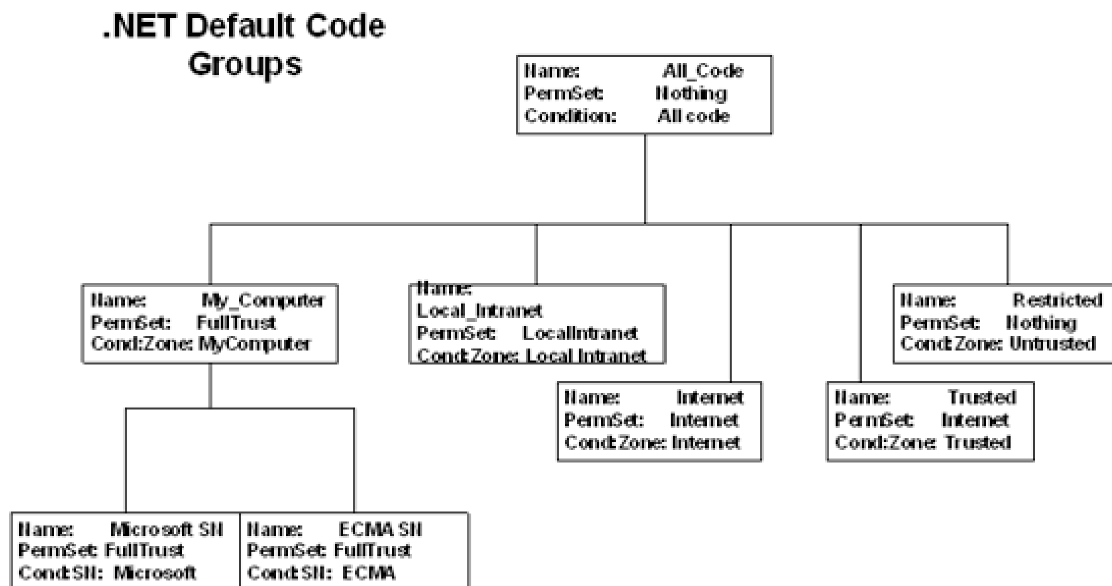

Figure 3-3. NET Default Code Groups

Custom groups may be added under any existing group (there is always a single root). Properly choosing the parent is an important task, because due to its hierarchical nature, the policy is navigated top-down, and the search never reaches a descendent node if its parents' `MembershipCondition` was not satisfied. In Figure 3-3 above, the *Microsoft* and *ECMA* nodes are not evaluated at all for non-local assemblies.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Built-in nodes can be modified or even deleted, but this should be done with care, as this may lead to the system's destabilization. Zones, identifying code, are defined by Windows and managed from Internet Explorer, which allows adding to or removing whole sites or directories from the groups. All code in non-local groups have special access rights back to the originating site, and assemblies from the intranet zone can also access their originating directory shares.

To add a custom code group using an existing `NamedPermissionSet` with an associated `MembershipCondition`, one only needs to run the `caspol.exe` tool. Note that this tool operates on groups' ordinal numbers rather than names:

```
caspol -addgroup 1.3 -site
   www.MySite.com LocalIntranet
```

Actually, .NET has three independent policies, called *Levels*: `Enterprise, Machine,` and `User`. As a rule, a majority of the configuration process takes place on the `Machine` level the other two levels grant `FullTrust` to everybody by default. An application can be a member of several groups on each level, depending on its *evidence*. As a minimum, all assemblies are member of the *AllCode* root group.

Policy traversal is performed in the following order: Enterprise, Machine, and then User, and from the top down. On each level, granted permission sets are determined as follows:

Level Set = Set1 U Set2 U ... U SetN

where *1..N* — code groups matching assembly's *evidence*. System configuration determines whether union or intersection operation is used on the sets.

The final set of permissions is calculated as follows:

Final Set = Enterprise X Machine X User

Effectively, this is the least common denominator of all involved sets. However, the traversal order can be altered by using `Exclusive` and `LevelFinal` policy attributes. The former allows stopping intra-level traversal for an assembly; the latter, inter-level traversal. For instance, this can be used to ensure, on the `Enterprise` level, that a particular assembly always has enough rights to execute.

The NET.III.CodePermissions.zip demo application demonstrates the tasks involved in granting custom permissions in the policy and executing code that requires those permissions.

Each policy maintains a special list of assemblies, called *trusted assemblies* -- they have `FullTrust` for that policy level. Those assemblies are either part of CLR, or are specified in the CLR configuration entries, so CLR will try to use them. They all must have strong names, and have to be placed into the `Global Assembly Cache` (GAC) and explicitly added to the policy (GAC can be found in the *%WINDIR%\assembly* directory):

```
gacutil /i MyGreatAssembly.dll

caspol -user -addfulltrust MyGreatAssembly.dll
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Figure 3-4 shows the `Machine`-level trusted assemblies:

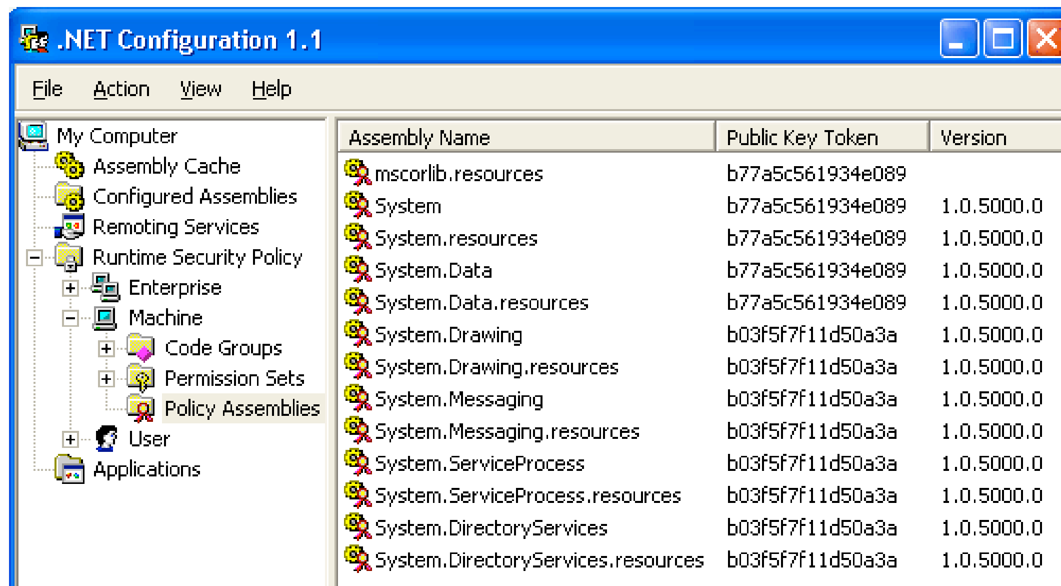

Figure 3-4. NET Trusted Assemblies

For Java, two types of code evidence are accepted by the JVM -- `codebase` (URL, either web or local), from where it is accessed, and `signer` (effectively, the publisher of the code). Both evidences are optional: if omitted, all code is implied. Again, publisher evidence is more reliable, as it less prone to attacks. However, up until JDK 1.2.1, there was a bug in the `SecurityManager`'s implementation that allowed replacing classes in a signed JAR file and then continuing to execute it, thus effectively stealing the signer's permissions.

Policy links together *permissions* and *evidence* by assigning proper rights to code, grouped by similar criteria. A JVM can use multiple policy files; two are defined in the default `java.security`:

```
policy.url.1=file:${java.home}/lib/security/java.policy

policy.url.2=file:${user.home}/.java.policy
```

This structure allows creating multi-level policy sets: network, machine, user. The resulting policy is computed as follows: *Policy = Policy.1 U Policy.2 U ... U Policy.N* JVM uses an extremely flexible approach to providing policy: the default setting can be overwritten by specifying a command-line parameter to the JVM:

```
//adds MyPolicyFile to the list of policies

java -Djava.security.policy=MyPolicyFile.txt

// replaces the existing policy with MyPolicyFile

java -Djava.security.policy==MyPolicyFile.txt
```

Java policy has a flat structure: it is a series of `grant` statements, optionally followed by *evidence*, and a list of granted permissions. A piece of code may satisfy more than one clause's condition — the final set of granted permissions is a union of all matches:

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
grant [signedBy "signer1", ..., "signerN"] [codeBase "URL"] {

        permission <PermissionClassName> "TargetName", "Action"
            [signedBy "signer1", ..., "signerN"];
        ...

}
```

The Java.III.CodePermissions.zip demo application defines a custom permission in the policy and executes applications requiring that permission.

Even locally installed classes are granted different trust levels, depending on their location:

- *Boot classpath*: `$JAVA_HOME/lib, $JAVA_HOME/classes`

  These classes automatically have the full trust and no security restrictions. Boot classpath can be changed both for compilation and runtime, using command-line parameters: `-bootclasspath` and `-Xbootclasspath`, respectively.

- *Extensions*: `$JAVA_HOME/lib/ext`

  Any code (JAR or class files) in that directory is given full trust in the default `java.policy`:

  ```
  grant codeBase "file:{java.home}/lib/ext/*" {
          permission java.security.AllPermission;
  }
  ```

- *Standard classpath:* `$CLASSPATH (".") by default)`

  By default, have only few permissions to establish certain network connections and read environment properties. Again, the `SecurityManager` has to be installed (either from command line using the `-Djava.security.manager` switch, or by calling `System.setSecurityManager`) in order to execute those permissions.

Policy-based security causes problems for applets. It's unlikely that a web site's users will be editing their policy files before accessing a site. Java does not allow runtime modification to the policy, so the code writers (especially applets) simply cannot obtain the required execution permissions. IE and Netscape have incompatible (with Sun's JVM, too!) approaches to handling applet security. JavaSoft's Java plug-in is supposed to solve this problem by providing a common JRE, instead of the browser-provided VM.

If the applet code needs to step outside of the sandbox, the policy file has to be edited anyway, unless it is an RSA-signed applet. Those applets will either be given full trust (with user's blessing, or course), or if policy has an entry for the signer, it will be used. The following clause in the policy file will always prevent granting full trust to any RSA-signed applet:

```
grant {
  permission java.lang.RuntimePermission "usePolicy";
}
```

Note: Policy in .NET has a much more sophisticated structure than in Java, and it also works with many more types of evidences. Java defines very flexible approach to adding and overriding default policies -- something that .NET lacks completely.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Code Access Security: Access Checks

Code access checks are performed explicitly; the code (either an application, or a system library acting on its behalf) calls the appropriate Security Manager to verify that the application has the required permissions to perform an operation. This check results in an operation known as a *stack walk*: the Runtime verifies that each caller up the call tree has the required permissions to execute the requested operation. This operation is aimed to protect against a *luring attack*, where a privileged component is misled by a caller into executing dangerous operations on its behalf. When a stack walk is performed prior to executing an operation, the system can detect that the caller is not allowed to do what it is requesting, and abort the execution with an exception.

Privileged code may be used to deal with luring attacks without compromising overall system security, and yet provide useful functionality. Normally, the most restrictive set of permissions for all of the code on the current thread stack determines the effective permission set. To bypass this restriction, a special permission can be assigned to a small portion of code to perform a reduced set of restricted actions on behalf of under-trusted callers. All of the clients can now access the protected resource in a safe manner using that privileged component, without compromising security. For instance, an application may be using fonts, which requires opening font files in protected system areas. Only trusted code has to be given permissions for file I/O, but any caller, even without this permission, can safely access the component itself and use fonts.

Finally, one has to keep in mind that code access security mechanisms of both platforms sit on top of the corresponding OS access controls, which are usually role or identity-based. So, for example, even if Java/.NET's access control allows a particular component to read all of the files on the system drive, the requests might still be denied at the OS level.

A .NET assembly has a choice of using either imperative or declarative checks (demands) for individual permissions. Declarative (attribute) checks have the added benefit of being stored in metadata, and thus are available for analyzing and reporting by .NET tools like `permview.exe`. In either case, the check results in a stack walk. Declarative checks can be used from an assembly down to an individual properties level.

```
//this declaration demands FullTrust
//from the caller of this assembly

[assembly:PermissionSetAttribute(
  SecurityAction.RequestMinimum,
  Name = "FullTrust")]


//An example of a declarative permission
//demand on a method

[CustomPermissionAttribute(SecurityAction.Demand,
  Unrestricted = true)]
public static string ReadData()
{ //Read from a custom resource. }


//Performing the same check imperatively

public static void ReadData()
{
  CustomPermission MyPermission = new
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
    CustomPermission(PermissionState.Unrestricted);
  MyPermission.Demand();
  //Read from a custom resource.
}
```

In addition to ordinary code access checks, an application can declaratively specify `LinkDemand` or `InheritanceDemand` actions, which allow a type to require that anybody trying to reference it or inherit from it possess particular permission(s). The former applies to the immediate requestor only, while the latter applies to all inheritance chain. Presence of those demands in the managed code triggers a check for the appropriate permission(s) at JIT time.

`LinkDemand` has a special application with strong-named assemblies in .NET, because such assemblies may have a higher level of trust from the user. To avoid their unintended malicious usage, .NET places an implicit `LinkDemand` for their callers to have been granted `FullTrust`; otherwise, a `SecurityException` is thrown during JIT compilation, when an under-privileged assembly tries to reference the strong-named assembly. The following implicit declarations are inserted by CLR:

```
[PermissionSet(SecurityAction.LinkDemand,
   Name="FullTrust")]

[PermissionSet(SecurityAction.InheritanceDemand,
   Name="FullTrust")]
```

Consequently, if a strong-named assembly is intended for use by partially trusted assemblies (i.e., from code without `FullTrust`), it has to be marked by a special attribute, `[assembly: AllowPartiallyTrustedCallers]`, which effectively removes implicit `LinkDemand` checks for `FullTrust`. All other assembly/class/method level security checks are still in place and enforceable, so it is possible that a caller may still not possess enough privileges to utilize a strong-named assembly decorated with this attribute.

.NET assemblies have an option to specify their security requirements at the assembly load time. Here, in addition to individual permissions, they can operate on one of the built-in non-modifiable `PermissionSets`. There are three options for those requirements: `RequestMinimum, RequestOptional,` and `RequestRefuse`.

If the `Minimum` requirement cannot be satisfied, the assembly will not load at all. Optional permissions may enable certain features. Application of the `RequestOptional` modifier limits the permission set granted to the assembly to only optional and minimal permissions (see the formula in the following paragraph). `RequestRefuse` explicitly deprives the assembly of certain permissions (in case they were granted) in order to minimize chances that an assembly can be tricked into doing something harmful.

```
//Requesting minimum assembly permissions
//The request is placed on the assembly level.

using System.Security.Permissions;
[assembly:SecurityPermissionAttribute(
   SecurityAction.RequestMinimum,
   Flags = SecurityPermissionFlag.UnmanagedCode)]
namespace MyNamespace
{
        ...
}
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

CLR determines the final set of assembly permissions using the granted permissions, as specified in .NET CAS policy, plus the load-time modifiers described earlier. The formula applied is (P - Policy-derived permissions): $G = M + (O<<P) - R$, where M and O default to P, and R to Nothing.

Applications on .NET platform may affect the stack-walking process by executing special operations on individual permissions or permission sets: `Assert, Deny, PermitOnly`. The application itself should be granted the affected permissions, as well as the `SecurityPermission` that grants the rights to make assertions.

The `Assert` option explicitly succeeds the stack walk (for the given `PermissionSet` or any subset of it, as determined by the `Intersect` function), even if the upstream callers do not have the required permissions (it fails if sets intersections are not empty). `Deny` and `PermitOnly` effectively restrict the available permission sets for the downstream callers.

The NET.III.PrivilegedCode.zip demo application demonstrates the effects of applying stack-walk modifications. Figure 3-5 represents an overview of the Code Access Security permission grants and checks in .NET:

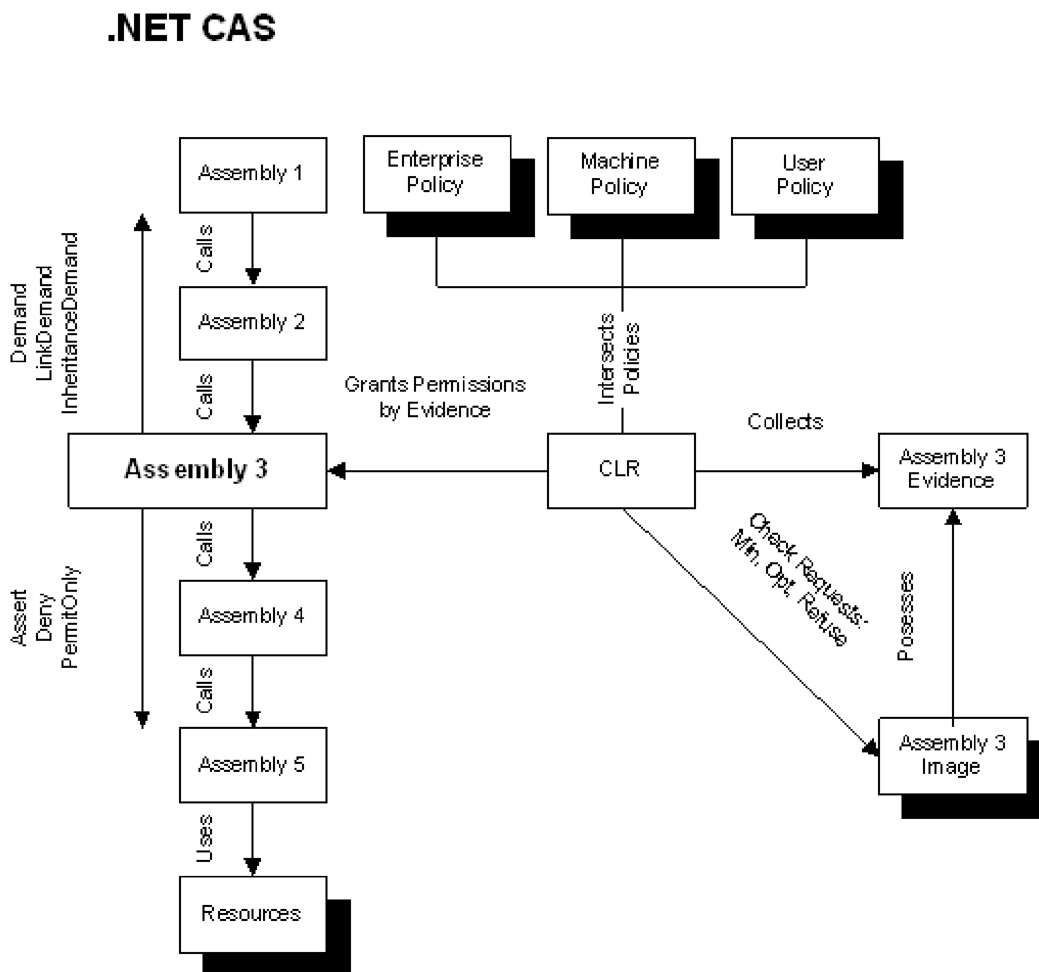

Figure 3-5. NET CAS Operation

In Java, permissions are normally checked by the `SecurityManager` (or installed derivative), by using the `checkPermission` function. It defines a helper for each major group of permissions, such as `checkWrite` for the `write` action of `FilePermission`. All checks are imperative; there are no declarative code access checks in Java language. Each JVM can have at most one `SecurityManager` (or derivative) installed — once set, they cannot be replaced,

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

for security reasons. Browsers always start `SecurityManager`, so any Internet Java application executes with enabled security. Locally started JVMs have to install a `SecurityManager` before exercising the first sensitive operation; this can also be done programmatically:

```
System.setSecurityManager(new SecurityManager());
```

or using a command-line option:

```
java -Djava.security.manager MyClass
```

In Java 2, when determining application permissions, `SecurityManager` delegates the call to `java.security.AccessController`, which obtains current snapshot of `AccessControllerContext` to determine which permissions are present. `SecurityManager`'s operations may be influenced by the `java.security.DomainController` implementation, if one exists. It instructs an existing `SecurityManager` to perform additional operations before security checks, thus allowing security system extensibility without re-implementing its core classes. JAAS uses this functionality to add principal-based security checks to the original code-based Java security (see section "User Access Security" in Part 4).

When making access control decisions, the `checkPermission` method stops checking if it reaches a caller that was marked as "privileged" via a `doPrivileged` call without a context argument. If that caller's domain has the specified permission, no further checking is done and `checkPermission` returns quietly, indicating that the requested access is allowed. If that domain does not have the specified permission, an exception is thrown, as usual.

Writing privileged code in Java is achieved by implementing the `java.security.PrivilegedAction` or `PrivilegedExceptionAction` interfaces. This approach is somewhat limiting, as it does not allow specifying the exact permissions to be asserted, while still requiring the callers to possess others — it is an "all or nothing" proposition.

```
public class PrivilegedClass implements PrivilegedAction {
    public Object run() {
        //perform privileged operation
        ...
        return null;
    }
}
```

Suppose the current thread traversed *m* callers, in the order of caller 1 to caller 2 to caller *M*, which invoked the `checkPermission` method. This method determines whether access is granted or denied based on the following algorithm:

```
i = m;
while (i > 0) {

  if (caller i's domain
      does not have the permission)
          throw AccessControlException

  else if (caller i is marked as privileged) {
          if (a context was specified
              in the call to doPrivileged)
             context.checkPermission(permission)
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
            return;
    }
    i = i - 1;
}

// Next, check the context inherited when
// the thread was created. Whenever a new thread
// is created, the AccessControlContext at that
// time is stored and associated with the new
// thread, as the "inherited" context.

inheritedContext.checkPermission(permission);
```

A complete application demonstrating privileged code in Java can be found in the Java.III.PrivilegedCode.zip demo.

Note: .NET arms developers with an impressive arsenal of various features for access checks, easily surpassing Java in this respect.

Chapter 3 – Conclusions

In this section, code protection and Code Access Security features of Java and .NET platforms were reviewed. While code protection came out more or less even, CAS features in .NET are significantly better than the ones Java can offer, with a single exception: flexibility. Java, as it is often the case, offers ease and configurability in policy handling that .NET cannot match.

Demo Applications

- Java.III.CodePermissions.zip
- Java.III.DataSigning.zip
- Java.III.PackageChecks.zip
- Java.III.PrivilegedCode.zip
- NET.III.CodePermissions.zip
- NET.III.DataSigning.zip
- NET.III.PrivilegedCode.zip

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Chapter 4 – Authentication and User Access Security (UAS)

When authentication comes into play, the system should already have a strong foundation, defined by the features discussed in previous sections. Authentication adds to that bag an ability to determine whether the user is the person he claims to be. Results of the authentication process are usually passed on to the authorization step.

The issue of user authorization (a.k.a. Role-based security) comes after solidifying the platform's base. At that point, in any more or less advanced system the administrator is left to be the judge, determining who is allowed to do what. This is traditionally done in two ways: using ACL to protect a particular resource (this is known as *Discretionary Access Control*, or DAC), or checking a user's group (or role) membership and allowing/denying him an operation based on the results of this check (a variation of *Mandatory Access Control*, or MAC).

User Authentication – General

The process of *authentication* starts right after *identification* by collecting caller credentials, confirming the identity claim, and securely communicating them to the server. Those credentials (possibly, several types of them, so it's called *multi-factor authentication*) are matched against the registered account information and a positive or negative answer is returned regarding the claimed identity.

To do this work, application developers can either utilize standard platform facilities, as described below, or roll out some custom authentication solution (for instance, biometric readers), which is outside of the scope of this publication. We also do not cover RMI and Remoting authentication, since their status was already discussed in Part 2.

.NET includes a web solution for the server side: ASP.NET, which is coupled with IIS for processing HTTP requests. It is also possible to attach it to a different Web Server, if an appropriate server extension is supplied. The IIS extension is called `aspnet_isapi.dll`, and handles all requests directed to ASP.NET (suffixes `ASPX`, `ASMX`, etc). ASP.NET itself, however, runs separately from IIS, in the `aspnet_wp.exe` process, so process isolation settings in IIS do not matter much. All managed code is executed in the worker process, and requests are forwarded there from the `aspnet_isapi.dll` extension through a named pipe.

Based on its configuration, IIS can either authenticate the requestor against a Windows account, using one of its standard methods (NTLM, Kerberos, Basic, Digest, Certificates) before forwarding the request, or forward the unauthenticated call to the ASP.NET handler. It is important to remember that security settings of IIS and ASP.NET are unrelated, although the latter uses IIS services for particular kinds of authentication.

ASP.NET handles authentication via so called *Authentication Modules*, one per each authentication type that APS.NET supports, which reside in `System.Web.Security` namespace. They all provide an `OnAuthenticate` event handler, which can be used to create a custom authentication/authorization schema by using different user account mapping and attaching new custom principals to the thread context.

The Java platform defines two solutions for user authentication to the servers: JAAS and servlets. Although EJB does not have its own authentication facilities and its 2.1 specification does not require any particular authentication mechanism from vendors, it does mention the

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

requirements for propagating the `Principal` object (see Identities section), created in a server-specific manner.

Java Servlets is the Java platform's HTTP-oriented server layer, which performs HTTP processing functions, similar to those of ASP.NET layer. Correspondingly, the Servlet's security model is intended specifically to handle the requirements of web applications.

JAAS may be used to add authentication and authorization to any Java-based application (executable, bean, applet, etc). It defines an API-based configurable generic authentication mechanism, independent of the underlying methods. The power of this approach lies in the clear separation of application and authentication code, allowing transparent replacement or alteration of authentication mechanisms.

User Authentication – Identities

In both systems, a principal and his identity (or identities) are established as a result of the authentication process, which serve to represent the user in the application during his further requests.

A user and his roles are represented in .NET via objects, implementing `IIdentity`, and `IPrincipal` interfaces, attached to the current thread context. `IIdentity` provides access to name and authentication type information, `IPrincipal` provides access to the contained user identity (one-to-one relationship) and role membership information. .NET provides two sets of implementations of those interfaces — `WindowsPrincipal` with `WindowsIdentity`, and `GenericPrincipal` with `GenericIdentity`. If the user does represent a Windows authorized account, he may use a `WindowsIdentity`, and this object represents a Windows security token, with role membership and authorization type derived from the Windows token. Generic versions of interfaces are used to implement any additional type of principal, unrelated to Windows accounts.

```
WindowsPrincipal principal =
      (WindowsPrincipal)Thread.CurrentPrincipal;
WindowsIdentity identity =
      (WindowsIdentity)principal.Identity;
```

There is no required relationship between the identities used by CLR and the current Windows process token, because CLR has a separate security context from that of Windows. In fact, CLR thread might not have an associated identity at all (or, rather, an empty one), while Windows threads always have one. In order for the CLR thread to take on the Windows thread's `WindowsIdentity` (to *synchronize*, using .NET jargon), it has to be configured to use `WindowsAuthenticationModule`. Otherwise, CLR and process threads will have two different identities.

As opposed to .NET's hierarchy, Java uses the word *Principal*, and the corresponding interface `java.security.Principal`, to represent user's identity. This user object carries only username information in it, not roles or any additional attributes about the logged-on user. This design reflects the focus on Code Access Security, prevailing in J2SE, since user access checks were not the main point of concern for Java designers initially. As for the identity synchronization with the OS thread, the J2EE specifications merely state that for a single sign-on capabilities a compliant J2EE product must be able to relate those identities.

Clearly, having only a username is not sufficient for any kind of serious application, so JAAS augmented it with additional information. JAAS groups multiple `Principal` objects,

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

representing the same user, into a single `Subject`, which also holds the user's credentials, such as password, certificate, or any kind of user-related information.

```
public final class Subject {
        public Set getPublicCredentials(); //not security-checked
        public Set getPrivateCredentials(); // security-checked
        public Set getPrincipals();
        ...
}
```

User credentials, obviously, have to be stored somewhere, for matching them later during the authentication process. Both platforms support multiple storage formats for user accounts: OS, disk-based files, database, and directory services. Necessary care must be taken to configure those storage areas properly, using file protection, secure hosts, communication encryption, etc.

ASP.NET applications have several options for storing user credentials:

- Through IIS, it can use whatever storage options are configured there (Windows account database, Active Directory Service).
- Passport accounts directory.
- For the simplest case of Forms mode, user credentials can be stored right in the application's configuration file. However, this approach has obvious maintenance drawbacks.

```
<credentials passwordFormat="SHA1">

  <user name="User1" password="3784AAB557DC76789FFA">

  <user name="User2" password="23933DCA564EE">

</credentials>
```

Neither Java nor J2EE specifications define any specific storage means for user accounts. The applications are capable of using Directory Services via `JNDI` mechanism, as well as other custom or vendor-provided solutions. Additionally, most commercial J2EE application servers provide some kind of mapping between the underlying OS' accounts and J2EE users and groups.

When a user logs in, a new session is created on the server and associated with that user. Servers typically terminate user sessions after some period of inactivity, as configured or set in the code.

ASP.NET applications map user requests to the `Session` objects, with their timeouts (in minutes) determined by `sessionState` tags in the `web.config` application file or through global setting in `machine.config`.

```
<sessionState timeout="20"/>
```

To track user sessions, Java Servlet engines use the `HttpSession` object. The sessions can be managed automatically and/or manually, providing expiration time to prevent session hijacking. The servlet container determines the default timeout for servlets sessions, which can be retrieved by calling `HttpSession.getMaxInactiveInterval`, and changed by calling

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

`HttpSession.setMaxInactiveInterval`. Specifying "-1" as the timeout interval means that the session never expires. An application can also override the default timeout by setting the desired value (in minutes) in its `web.xml` file:

```
<web-app>
  <session-config>
    <session-timeout>20</session-timeout>
  </session-config>
</web-app>
```

In EJB servers, Principals are associated with caller's requests in server-specific ways. J2EE specifications require that for all EJBs in a call chain within the same application the same identity must be returned for all calls to `EJBContext.getCallerPrincipal`, which should be the same identity as in `HttpServletRequest.getUserPrincipal` if that is not null. Whereas Servlet specifications do allow returning a null Principal, EJB specifications explicitly state that a non-null object should be returned at any time, even for representing an unauthenticated user.

EJB specification does not dictate any programmatic ways of propagating principals in the case of calling multiple beans or even multiple servers. Some EJB servers implement principal delegation mechanism akin to Java's `doPrivileged` privileged code execution. If desired, the application assemblers, via the deployment descriptor, may affect the choice of identities that execute their beans. There is `<security-identity>` element for that, which has two possible values:

- `<use-caller-identity>` — to force using caller's identity on any method of the bean by propagating it from the caller.
- `<run-as>` — to specify a particular role to run the bean.

```
<security-identity>
  <run-as>
    <role-name>Administrator</role-name>
  </run-as>
</security-identity>
```

User Authentication – Web Mechanisms

There is a standard set of Web-based authentication methods that may reasonably be expected by application developers on a particular platform. Generally supported authentication mechanisms include HTTP authentication (Basic and Digest), Forms/Cookies authentication, and Certificate authentication. The latter is usually coupled with mutual SSL/TLS authentication — this is the standard way of implementing it.

Forms authentication is normally performed with the help of a cookies mechanism. Two types of cookies are in use: temporary and persistent. The former last only during the current browser session, the latter are stored at the client's computer. Both types have an expiration time, to prevent identity theft, but persistent cookies are typically stored on the client's computer and remain valid for many days, so they pose a greater security threat.

Certificate authentication is significantly more secure, as it allows mutual authentication, so the client can be assured that it connects to the proper server. Configuring it, however, is more problematic, because it requires installing an X509 certificate on the client side.

ASP.NET heavily relies on IIS for its authentication needs. In fact, it uses ISS to implement all

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

of the above authentication modes, except for Forms, and merely consumes the results of the authentication process performed by IIS. Therefore, forms authentication is the only one that is going to be further discussed here for ASP.NET.

Authentication is handled by `FormsAuthenticationModule`, which handles all traffic, received from IIS via the extensions mechanism, as shown in Figure 4-1. It passes all authenticated requests through, while forwarding all unauthenticated ones to the specified logon page. The authentication, once performed, is sustained via the cookie mechanism, which can be made to expire after a timeout to prevent stealing user identity. Alternatively, another form-authentication scheme can be developed, for example, using hidden fields to store credentials in the form, and taking full control of authentication process by providing a `FormsAuthentication_OnAuthenticate` handler event in `Global.asax` file. It is possible (although not trivial) to create a completely cookieless authentication schema using this method.

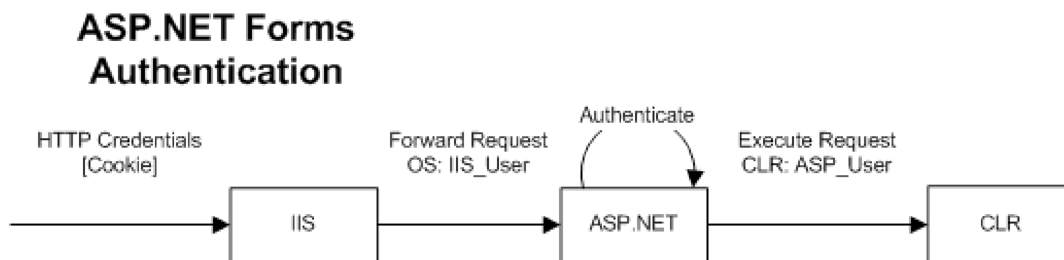

Figure 4-1. ASP.NET Forms Authentication

ASP.NET provides a helper class, `FormsAuthentication`, to help with common authentication tasks: authenticating username and password, issuing, encrypting and decrypting tickets, redirecting user request to the originally requested page after successful authentication, and signing out. An authenticated user is identified by the presence of an authentication cookie (either temporary or persistent), that is usually implemented by `FormsAuthenticationTicket` class. However, a custom cookie may be set in the code — this allows better control over its expiration property, as well as over the cookie's content.

```
string data = "Application data";

HttpCookie cookie = new HttpCookie(

    FormsAuthentication.FormsCookieName, data);

//expires in 10 minutes

cookie.Expires = DateTime.Now + new TimeSpan(0,0,10,0);

Response.Cookies.Add(cookie);
```

Custom content in a cookie may be protected by encrypting, via `FormsAuthentication.Encrypt` call, with user-provided or auto-generated 3DES key, which is then stored at the server's *Local Security Authority* (LSA). HMAC validation with a specified algorithm may also be requested.

```
<machineKey validationKey="AutoGenerate"

            decryptionKey="AutoGenerate"

            validation="SHA1"/>
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

The associated authentication cookie's name and expiration timeout may be configured in the top level application's `web.config` file. Cookies are renewed upon the next request after half of their expiration time has passed, thus keeping them from expiring. However, as was explained earlier, the server-side session associated with the user's identity expires after a certain period of inactivity, as determined by the `timeout` setting in the `sessionState` element. So, even if a request has a valid cookie, if the corresponding session has expired, the user will still be prompted to re-authenticate. Calling `FormsAuthentication.SignOut` will terminate any session association of the current user and remove cookies from the browser's cache.

```
<authentication mode="Forms">
  <forms forms="DemoApp"
         Loginurl="https://www.DemoApp.com/login.aspx"
         Name=".DEMOAPPCOOKIE"
         protection="All" Timeout="30" Path="/">
  </forms>
</authentication>
```

The authentication sequence works in the following way: after a request comes in, it is forwarded to the `OnAuthenticate` handler, if present. Here, any additional information may be extracted from the custom cookie or URL, and additional roles assigned. If a user identity is associated with the request after finishing rhe handler's execution, no further actions are taken. Otherwise, the request is checked by name for the presence of authentication cookie. If such a cookie is found, it is used to construct the appropriate `Principal` and associate it with the current request; otherwise, the request is forwarded to the logon page.

The Java Servlet specification requires support for the Basic HTTP authorization mechanism, and encourages (but does not require!) support for digest authentication, because it is a fairly rare mechanism. Additionally, two other forms of authentication are required for J2EE compatibility: Form-based and mutual Certificate (HTTPS Client). Basic, Digest, and Certificate authentication is carried out transparently, between Web Server and the connecting client, and does not require writing additional code.

```
<auth-method>BASIC|DIGEST|FORM|CLIENT-CERT
    </auth-method>
```

The form-based authentication schema is probably the most common option in use today. The specifications require the presence of the following names on the logon form: a `j_security_check` action, andthe fields `j_username`, and `j_password`. These indicate to the servlet engine that this is the logon information to process.

```
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password">
</form>
```

A container creates a persistent or temporary cookie named `JSESSIONID` for the user request, sets its expiration policy via a call to `Cookie.setMaxAge` before adding the cookie to the current session, and then keeps sending it back to the client with each response. The client returns it with each request, which allows mapping the connectionless requests to the user's

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

session. Alternatively, a container may use a technique called *URL Rewriting* to support those clients who do not accept cookies:

```
http://www.SomeHost.com/servlets/index.html;jsessionid=0124343
```

Note that using a `GET` operation in form-based login is a bad idea — it puts the entire request, with all its fields, into the URL, thus making it available when browsing the server log, for instance:

```
http://www.DemoServer.com/login?j_username=MyName&j_password=MyPassword
```

Using the `POST` operation, the URL will be as shown below, so the user information will not show up in the server's log:

```
http://www.DemoServer.com/login
```

When a request comes for one of the protected resources, the engine checks the user's authentication, and if he is has not been authenticated yet, forwards him to the login page associated with the resource. The Servlet engine is then responsible for redirecting the user back to the originally requested resource (or error page, in case of a failure).

```
<web-app>
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-error-page>/Error.html
      </form-error-page>
      <form-login-page>/SignOn.html
      </form-login-page>
    </form-login-config>
</login-config>
</web-app>
```

Certificate authentication is configured declaratively on the server side, but support for mutual HTTPS communication is required on both sides. The client's request should contain a certificate which can be mapped to a server's defined principal, which is going to be associated with this and further requests. Note that HTTPS, as opposed to HTTP, is a stateful protocol, and cookies are not needed to maintain session association.

```
<web-app>
  <login-config>
    <auth-method>CLIENT-CERT</auth-method>
  </login-config>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</web-app>
```

From inside of a servlet, client certificate information can be retrieved by accessing the `getAttribute` method of `javax.servlet.http.HttpServletRequest` class, requesting the following attribute:

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
X509Certificate cert =
  (X509Certificate)request.getAttribute
     ("javax.servlet.request.X509Certificate");
```

In the case of a HTTPS connection, the Servlet engine sets this attribute, as required by the Servlet specifications, before invoking the target servlet. Using attributes of the returned `X509Certificate` object, the servlet can perform any additional programmatic authentication of the caller.

Note: .NET delegates all types of user authentication, except for Forms, to IIS, and barely consumes the results. J2EE requires support for all standard authentication mechanisms from the compliant servers.

User Authentication: Other Mechanisms

Besides the standard Web-based mechanisms, both platforms provide other means for authentication.

In .NET, Windows and Passport authentication are incorporated as separate entities via the corresponding modules. They both are used together with IIS, and require very little configuration in the application's configuration file.

`WindowsAuthenticationModule` relies on IIS to authenticate the caller, as shown in Figure 4-2, and attaches `WindowsPrincipal` object to the application context. This is the default provider for ASP.NET, and the easiest to use in pure Microsoft network environment, as it requires no additional application code.

```
<authentication mode="Windows">
</authentication>
```

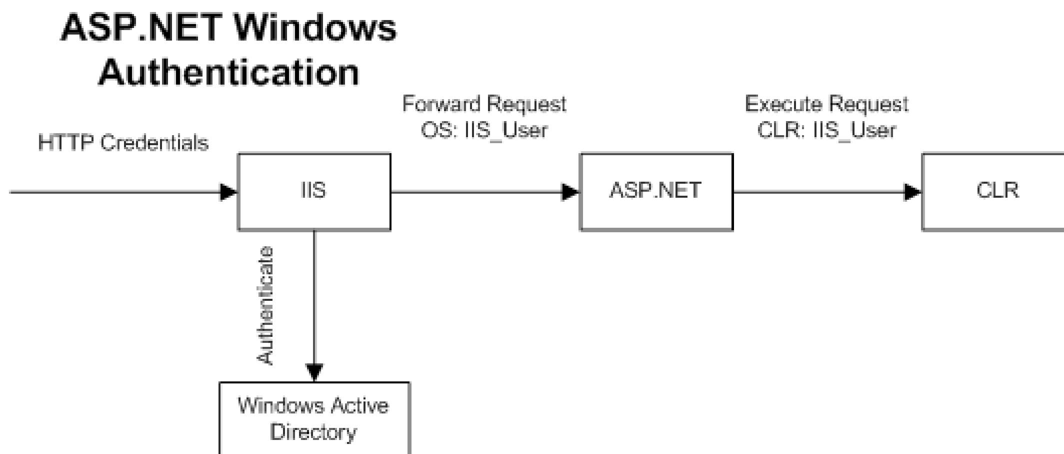

Figure 4-2. ASP.NET Windows Authentication

`PassportAuthenticationModule` is a wrapper around the Passport SDK that creates a `PassportIdentity` object using Passport service and profile, as shown in Figure 4-3. This *identity* object provides access to the Passport profile and allows the encrypting/decrypting authentication tickets. Most of the authentication details are handled by the ASP.NET framework; the developer can control the process by overloading `OnAuthentication` handler.

```
<authentication mode="Passport">
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
</authentication>
```

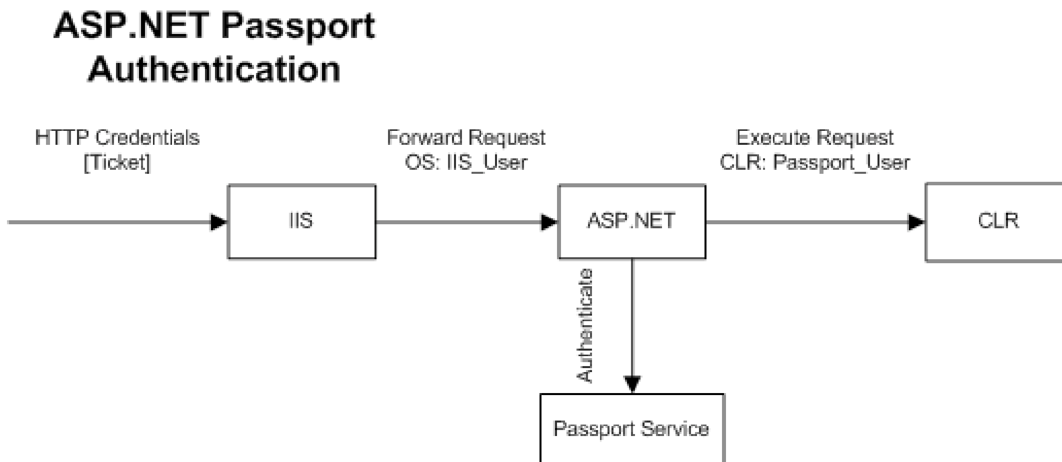

Figure 4-3. ASP.NET Passport Authentication

In Java, JAAS servers as a general abstraction for providing authentication services to applications. JAAS relies on *Pluggable Authentication Modules* (PAM) in its operation to provide a flexible authentication framework. Administrators can add various implementations to the environment and modify its behavior and authentication method. The default PAM is username/password based; however, it is possible to use alternative schemas. In JDK 1.4, Sun provides implementations for the following login schemas via its `LoginModule` implementations: `UnixLoginModule`, `NTLoginModule`, `JNDILoginModule`, `KeyStoreLoginModule`, `Krb5LoginModule`. Additionally, there exist implementations of SmartCard login modules by independent vendors, for instance — GemPlus.

The sample Java application Java.IV.JAASAuthentication.zip demonstrates some of the discussed JAAS authentication techniques.

In this section, only the authentication part of JAAS will be reviewed. Its operation is controlled by the `LoginContext`, which uses the `Configuration` class to retrieve the specified `LoginModules`. Those modules retrieve credentials with the help of provided `Callbacks`, although it is possible to use other means as well. There might be several `LoginModules` configured, and during the login process all of them are queried in turn, which is shown in Figure 4-4.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

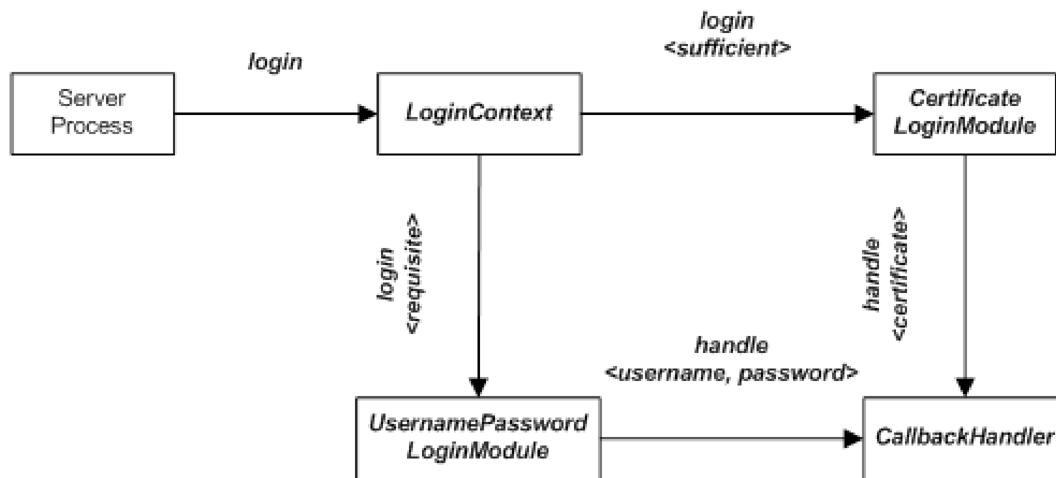

Figure 4-4. JAAS Login

The relationship among `LoginModules` is determined by the strategy configuration settings, which tells the system how to treat login failures in individual modules:

- `required` — it must succeed for the overall login to succeed. However, `LoginContext` finishes querying other modules before aborting the process.
- `requisite` — same as `required`, but the login process stops if this module fails.
- `sufficient` — success of this module means that the overall process succeeds (provided that no `required, requisite` modules failed) and the login process stops.
- `optional` — just what it says, pretty much does not affect the login process.

JAAS introduces a couple of new permissions, `javax.security.auth.PrivateCredentialPermission` and `javax.security.auth.AuthPermission`, to guard access to the `Subject`, `LoginContext`, and `Configuration` classes. The code that works with JAAS classes will need to have them (especially `AuthPermission`) granted in `java.policy`. See the online javadoc for details about their usage.

The following classes are used as part of JAAS authentication process:

- `javax.security.auth.login.LoginContext` — represents the PAM framework. This class is used by the server to access `Configuration` and use the specified `LoginModule`(s) to validate the passed user credentials. Once the login operation successfully finishes on all configured `LoginModule`(s) (using two-phase commit process), the `Subject` is attached to the call context and is available for the application code.

```
public final class LoginContext {
  public LoginContext(String name);
  public void login(); // two phase process
  public void logout();
  public Subject getSubject();//get the authenticated Subject
}
```

- `javax.security.auth.login.Configuration` — tells JAAS' `LoginContext` which `LoginModule`(s) are configured, and what the login strategy is. An alternative configuration provider implementation can be specified in `java.security` file by setting `login.configuration.provider` property.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
MyJAASConfig {
  UsernamePasswordLoginModule   requisite;
  CertificateLoginModule        sufficient;
};
```

The location of the configuration file is specified using the command-line option:

```
java -Djava.security.auth.login.config==jaas.config MyJAASApp
```

- `javax.security.auth.spi.LoginModule` — represents a particular authentication type in an application — for instance, password-based or RSA. During login, each of the configured modules is requested by the `LoginContext` object to authenticate the user credentials. If the authentication succeeds (as configured in the JAAS), the login is committed, and a `Subject` is created, otherwise the `abort` method is called.

```
public interface LoginModule {
  boolean login(); // 1st authentication phase
  boolean commit(); // 2nd authentication phase
  boolean abort();
  boolean logout();
  void initialize(Subject subject, CallbackHandler handler,
      Map sharedState, Map options);
}
```

- `javax.security.auth.callback.Callback, javax.security.auth.callback.CallbackHandler` — are used to gather all necessary credentials and report them back to the requesting module. Sun provides default implementations for several callbacks with the package, most importantly: `NameCallback`, and `PasswordCallback`. Varying handlers may be used to gather and return the requested information; the ones supplied by default include `DialogCallbackHandler` for dialog-based, and `TextCallbackHandler` — for command-line prompts.

Different user identities and roles are represented via `Principals`, added to the resulting `Subject` by the configured `LoginModules` during the `commit` phase, and removed during `logout`. Their credentials may also be added to that object, as well as any additional identification information.

```
public abstract class DemoLoginModule implements LoginModule {
  protected Subject m_subject;
  protected ArrayList m_principals = null;
  public boolean commit() throws LoginException {
    // Login succeeded,
    // add demo Principals to the Subject.
    if (!(m_subject.getPrincipals().containsAll(
                                m_principals))) {
        m_subject.getPrincipals().addAll(
                                   m_principals);
    }
    return ret;
  }
  public boolean logout() throws LoginException {
    // Need to remove our
    // principals from the Subject.
    if (null != m_principals) {
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
            m_subject.getPrincipals().removeAll(
                                            m_principals);
            m_principals = null;
        }
        return true;
    }
}
```

The interaction of the various JAAS classes during initialization process is shown in Figure 4-5 below.

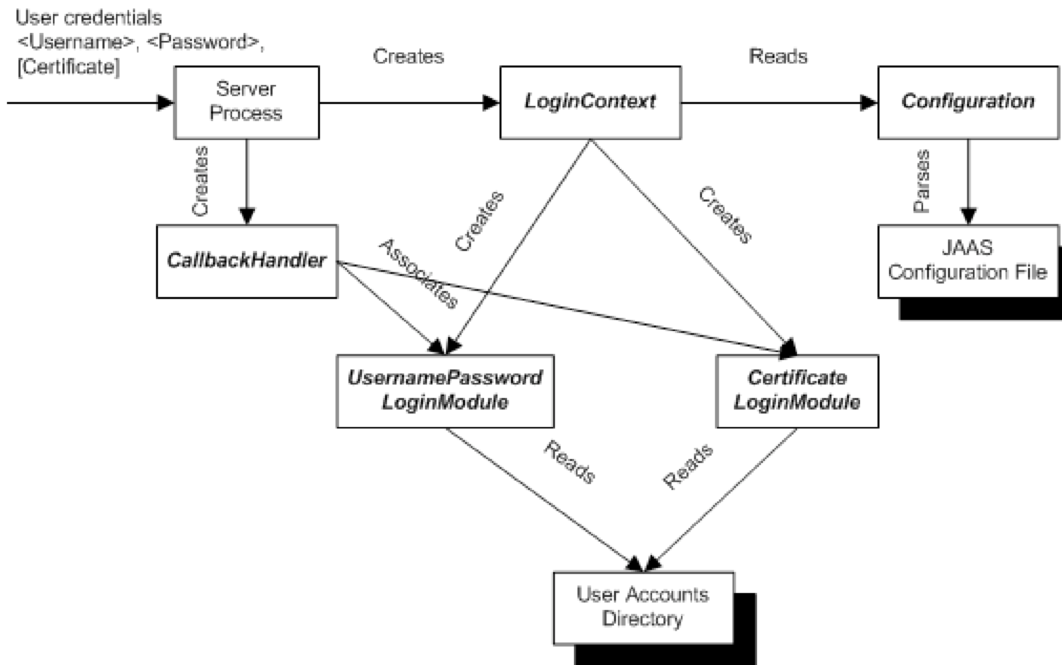

Figure 4-5. JAAS Initialization

Note: JAAS provides a flexible and versatile mechanism for adding authentication to any type of Java application.

User Authentication – Impersonation

In addition to the direct login, an application can impersonate the logged-in user while performing sensitive operations locally, or delegate his identity when communicating with remote servers. This way, the execution will happen in the context of the logged-in user, with privileges granted to the user's identity, and not that of the application.

.NET's impersonation has a distinctive feature of being in a close relationship with the Windows process' identity. Turning on impersonation in ASP.NET configuration will result in ASP.NET's Windows execution thread borrowing the security token of the calling IIS process. So, as far as Windows security system is concerned, ASP.NET Windows thread's identity will be the same as that of the IIS thread, although CLR identity may be completely different, as has been explained before.

Sample ASP.NET configurations are listed below:

- **Impersonating the IIS calling identity in ASP.NET process, and synchronizing CLR identity.** This will result in all three threads (IIS and ASP.NET Windows, and CLR

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

managed) to share the same Windows identity, authenticated by IIS, or *IUSR_<MACHINE>* for anonymous users.

```
<authentication mode="Windows">
      </authentication>
<identity impersonate="true"/>
```

- **Impersonating the IIS calling identity in ASP.NET process, and using separate CLR identity (or synchronizing in code)**: note that the code should have a proper CAS permission to modify the principal. This will result in IIS and ASP.NET Windows threads sharing the same Windows identity, and CLR — having an empty one initially.

```
<authentication mode="None">
      </authentication>
<identity impersonate="true"/>

// Providing a generic identity
<%
Identity clrIdentity =
      new GenericIdentity("CLRUser");
String[] roles = {"PowerUser"};
GenericPrincipal clrPrincipal =
       new GenericPrincipal(clrIdentity, roles);
Thread.CurrentPrincipal = clrPrincipal;
%>
```

When knowing account's credentials, it is possible to impersonate not only the currently logged in user, but also an arbitrary user. `WindowsIdentity` has an overloaded constructor which accepts a Windows account token. That token can be retrieved by making an unmanaged call to Windows function `LogonUser` (permissions permitting, of course).

```
// Impersonating a logged-in Windows user
<%
WindowsIdentity id =
        new WindowsIdentity(userTokenHandle);
WindowsImpersonationContext ctxt =
                            id.Impersonate();
...
ctxt.Undo();
%>
```

An important point to remember about Windows impersonation is that it was designed for use with trusted code only — any unmanaged DLL down the call chain can call `RevertToSelf`, and start using IIS process identity, which will most likely be *System*. Although managed code is a subject to CAS permission checks, which generally deny the corresponding security permissions to most applications, it does not apply to the locally installed code, which has `FullTrust` under the default policy.

Among .NET's authentication options, delegation is currently supported only by the Kerberos protocol, which is used only with `WindowsIdentity` for now. In order to use a Windows identity for delegation, the impersonated user's Windows account should be granted the right *"Act as a part of OS"* by the administrator, which is not given on a general basis.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Java uses JAAS to implement impersonation on the application level. JAAS' `Subject` class allows executing a particular sensitive operation (marked so by implementing `java.security.PrivilegedAction` interface) as another user's identity.

```
public final class Subject {
  ...
  // associate the subject with the current
  // AccessControlContext and
  // execute the Privileged action
  public static Object doAs(Subject s,
       java.security.PrivilegedAction action);
}
```

When this operation is run, the `doAs` method associates the impersonated subject with the current `AccessControlContext`, and then executes the action. At the end of the `doAs` call, the subject is removed from the `AccessControlContext`:

```
//class representing a protected operation
class ProtectedOperation
                 implements PrivilegedAction {
  //do something veeeery sensitive here...
  public Object run();
}

public class ImpersonationExample {
  public static void main(String args[]) {
    ...
    //carry out the authentication process
    Subject subject = loginContext.getSubject();
    //run as the impersonated user
    Subject.doAs(subject,
                 new ProtectedOperation());
  }
}
```

In the default JDK implementation, Java impersonation is limited to application level only — specifications do not define any relationship to user accounts on the underlying OS. Specific vendor implementations can implement functionality that maps the logged-in user to the OS domain names. For instance, WebLogic, if configured, can use NT PAM to authenticate users against Windows account names.

However, GSS-API, in combination with JAAS, can handle both impersonation and delegation, as shown in Figure 4-6.

- Impersonation is handled by creating a new Subject, using the name obtained from the call context, and associating it with the current thread. No credentials are obtained this way, so it will not be possible to delegate the user identity to some other service.

```
GSSName name = context.getSrcName();
  Subject newSubject =
    com.sun.security.jgss.GSSUtil.createSubject(
                                   name,null);
  //set the execution subject and call doAs
  ...
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

- Delegation requires obtaining user credentials, in addition to the username, which requires cooperation from the client — he should authorize credentials delegation to enable this mode. This would be usually done together with mutual authentication, to verify the server's identity as well. Once the server obtains client's Kerberos Ticket (TGT), it can represent the client in calls to remote services, in addition to performing local operations. The server's actions are controlled by two Kerberos-specific permissions: `ServicePermission` and `DelegationPermission`.

```
// client allows using its credentials
// with mutual authentication
GSSManager manager = GSSManager.getInstance();
GSSContext contextClient =
                manager.createContext(serverName,krb5Oid,
null,GSSContext.DEFAUL_LIFETIME);
contextClient.requestMutualAuth(true);
contextClient.requestCredDeleg(true);
...

//server obtains client's credentials
if (contextServer.getCredDelegState()) {
  GSSCredential credClient =
                contextServer.getDelegCr();
  //use the credentials to act as a client
  GSSContext contextDelegate =
    manager.createContext(backendName,krb5Oid,
  credClient,GSSContext.DEFAUL_LIFETIME);
}
```

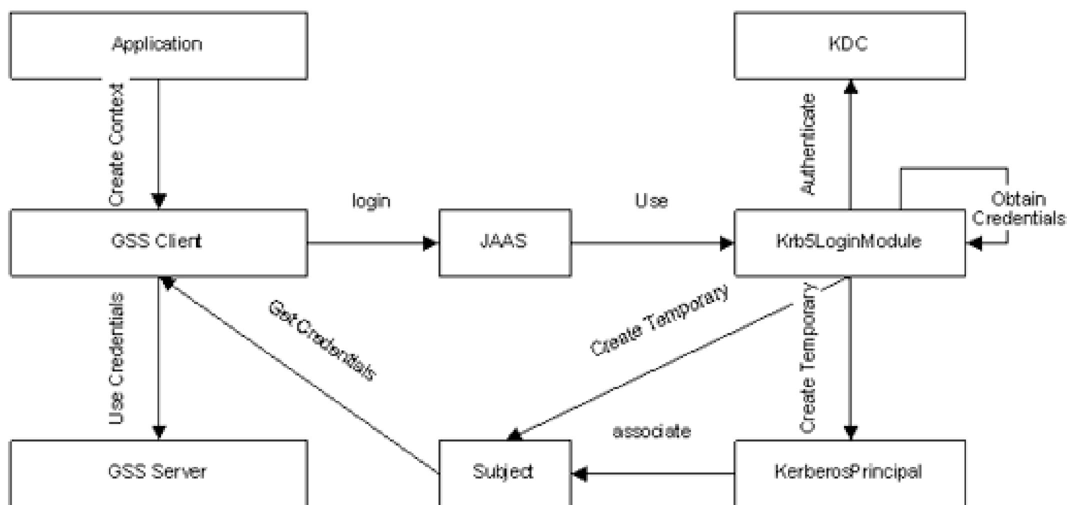

Figure 4-6. GSS/JAAS Authentication

It is possible to configure the Kerberos provider to use an existing credentials cache so that the login happens completely transparently.

```
// client JAAS configuration for GSS-API
com.sun.security.jgss.initiate {
  com.sun.security.auth.module.Krb5LoginModule
                                    required;
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
};

// server JAAS configuration for GSS-API
com.sun.security.jgss.accept {
  com.sun.security.auth.module.Krb5LoginModule
                                        required
                  useKeyTab=true storeKey=true
                          principal="nfs/host";
};

// default configuration for GSS-API
// if the above is not present
other {
...
}
```

Note: .NET provides good support for impersonation on Windows-only networks, but delegation across the Internet is not possible. Java can do application-level impersonation and is capable of supporting delegation across the Internet.

User Access Security – Basic

Once a distinguished *principal* has been identified as a result of the authentication step and attached to the call context (usually associated with threads), it can be used in determining resource access rights. In role-based systems, application code may operate not only with specific principals, but also with their abstract roles, which results in more flexible system configuration. So after establishing a *principal*, the server goes through an additional step of mapping it to the possible application roles.

Each executing .NET thread has an associated `CallContext`, which carries around the *Principal* and his *Identity* — they are either copied from the creating thread, or created anew by CLR when code tries to access them for the first time.

```
WindowsPrincipal principal =
  (WindowsPrincipal) Thread.CurrentPrincipal;
WindowsIdentity identity =
  WindowsIdentity.GetCurrent();
```

A configurable policy governs the type of *principal* created by default: `NoPrincipal`, `UnauthenticatedPrincipal`, `WindowsPrincipal`. An application, which is granted the appropriate `SecurityPermission`, can set this policy imperatively:

```
AppDomain.CurrentDomain.SetPrincipalPolicy(
  PrincipalPolicy.WindowsPrincipal);
```

An application that possesses a proper `SecurityPermission` to control the *principal* can replace the current thread's *principal*. However, this permission is not required for normal role-based checks:

```
GenericIdentity id = new GenericIdentity("user");
String[] roles = {"Manager","User"};
GenericPrincipal pr = new GenericPrincipal(
                                      id,roles);
Thread.CurrentPrincipal = pr;
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

To provide more consistent security architecture, .NET incorporates role-based security into code access hierarchy by providing a `PrincipalPermission`, available for both declarative and imperative checks. Checks can be performed by name, role, or combination of both.

```
[PrincipalPermissionAttribute(
    SecurityAction.Demand, Role = "PowerUser")]
```

Optionally, principal permission objects can be combined in code (but not declaratively!) to support checks several identity/roles at once:

```
PrincipalPermission perm1 =
   new PrincipalPermission("John","Admin");
PrincipalPermission perm2 =
   new PrincipalPermission("PowerUser");
(perm2.Union(perm1)).Demand();
```

Finally, ordinary checks for user names and role can be performed in code by directly accessing the `IPrincipal` object:

```
Principal principal = Thread.CurrentPrincipal;
if (principal.IsInRole("Admin"))
{
  //do something for Admin
} else if (principal.Identity.Name == "John")
{
  //do something for John
}
```

A sample application, demonstrating the user access checks in .NET, is provided as NET.IV.CodeAuthorization.zip.

It is important to realize that .NET policy can not extend the final permission set granted to the assembly, based on user's identity. In other words, if an assembly *A* is granted, as a result of policy evaluation, permission set *PA*, the same will be granted happen for any user executing this assembly. This set can be further restricted based on the results of role and user checks. This is in contrast to the way most modern operating systems, including Windows, work: a user is granted certain additional privileges based on his identity or group membership.

In Java, JAAS grants permissions based on user identity, as defined by name, as opposed to the pure policy-based model, which grants the permissions based on the code's origin. Declarative security is set through the `java.policy` file — JAAS adds `Principal` entries to the Java policy. As an important difference from .NET model, JAAS Principal-based model can extend the permission set granted to a module. In the example below, the code, signed by "MyPublisher", is granted write permissions to "C:\" only if it is executed by "user":

```
grant Signedby "MyPublisher" {
        permission java.io.FilePermission "c:\","read";
}

//an example of grant by username
grant Principal com.comp.PrincipalClass "user" {
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
        permission java.io.FilePermission "c:\","write";
}
```

Principal-based access policy enforcement is performed using `PrivilegedAction` and impersonation. As has been explained in Part 3, running this class effectively asserts all privileges granted to the code, including those based on the current `Principal`. Technically, after `doAs` has been called with an impersonated `Subject`, `java.lang.SecurityManager` updates current `AccessControlContex` from the policy file, adding permissions for the impersonated user. An internal JAAS implementation of `java.security.DomainCombiner` is responsible for instructing the installed `SecurityManager` to query JAAS policy and update the `AccessControlContext`. In the server environment, which concurrently handles multiple calls, it is important to use `doAsPrivileged` and pass it null `AccessControlContex` to force policy re-evaluation by the *Combiner* and to create a new context customized for the user, instead of borrowing the server's existing one. At the security checkpoints during the execution, the total granted permission set now includes code-based, as well as `Principal`-based application permissions.

The Java.IV.JAASAuthorization.zip application demonstrates the effects of dynamic policy evaluation in JAAS.

There is no notion of roles in the JAAS hierarchy; everything is determined by usernames. Although not very convenient, roles and groups may be treated as named principals, and access control may be imposed on them in the same way. Moreover, since a `Subject` may contain any number of `Principals`, objects representing role(s) can be added to its `Principal` collection. Later the `Subject`'s roles may be retrieved by requesting principals of only a particular class, which denotes a particular role. To build application name-based role hierarchies, JAAS defines `com.sun.security.auth.PrincipalComparator` interface, which may be implemented by the `Principal` classes specified in the policy's `"grant"` entries. `PrincipalComparator.implies` method should return `true` when the specified `Subject` is in a particular role:

```
// an example of role-based entry
grant Principal com.MySite.AppRole "PowerUser" {
  permission java.io.FilePermission "c:\","read";
}

// this class is used for building role hierarchy
public class AppRole implements
                      PrincipalComparator {
  // the role this object represents
  public AppRole(String role) {...}

  //this method checks the Subject
  //for being in the role
  public boolean implies(Subject currSubject) {
    ...
  }
}
```

`Subjects` are assigned by JAAS to the current thread's execution context, and are available for examining directly from the code — therefore, programmatic security checks can also be based on principal names, as obtained from the current execution context.

```
AccessControlContext ctxt =
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
                    AccessController.getContext();
Subject subj = Subject.getContext(ctxt);
if (subj == null) {
  //no authenticated user
} else {
  Set principalsSet = subj.getPrincipals();
  Iterator iter = principalsSet.iterator();
  while(iter.hasNext()) {
    MyPrincipalClass princ =
      (MyPrincipalClass)iter.next();
    if (princ.getName().equals("MyUser")) {
      // have an authenticated user
    }
  }
}
```

Note: .NET has a very convenient, permission-based user access system. However, it can only restrict the total permission set for an assembly, never extend it. JAAS makes use of dynamic policies in Java to extend granted permission set with user-specific permissions.

User Access Security: Extended

In addition to the basic facilities for user access checks, extension packages on both platforms define their own mechanisms.

ASP.NET provides security checks, which work on the top of regular CLR security facilities:

- `FileAuthorizationModule` — performs ACL checks on accessed `.aspx` and `.asmx` files. It is active when `Windows` authentication is enabled, and is used to determine whether the user passes Windows ACL checks.
- HTTP handlers — there are several of these specified in `machine.config` to prevent disclosure of certain types of files. Note that this mechanism works separately from the IIS-defined one, and only for the file extensions registered to ASP.NET, so separate IIS configuration is needed to ensure full protection.

```
<httpHandlers>
  <add verb="*" path="*.vjsproj"
         type="System.Web.HttpForbiddenHandler"/>
  <add verb="*" path="*.java"

         type="System.Web.HttpForbiddenHandler"/>
  ...
  <add verb="*" path="*"
  type="System.Web.HttpMethodNotAllowedHandler"/>
</httpHandlers>
```

- `URLAuthorizationModule` — performs URL authorization by providing declarative hierarchical mapping of users and roles to URI namespace. This mode and allows for positive and negative assertions on the protected resources, and accepts wildcards "*" for all users and "?" for anonymous users. There is a global configuration file, and each subdirectory may have its own version of it, overwriting some attributes. The hierarchy is parsed starting from the lowest level, and the first match wins. This is certainly the quickest way to enable access control, but not necessarily the best, because it scales poorly, and is not easily manageable, especially for multi-server applications.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
<authorization>
  <allow users="Don, MyDomain\Don" roles="Admin"
                verbs="GET, POST">
  <deny users="?" roles="Guest" >
</authorization>
```

Java Servlets and JSPs use role-based access control checks, which can be specified programmatically or declaratively, similar to ASP.NET. The mapping between authenticated users and security roles is not specified; it happens in a vendor-specific way. However, the Servlet specification does standardize ACL declarations by security roles in the `web.xml` deployment descriptor, which can protect web resources defined as HTTP methods applied to URL-patterns. Also, the `transport-guarantee` element is considered during requests evaluation. A side effect of this approach is that the resulting declarative access control mechanism is rather coarse, on the file/operation level:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Restricted Servlets
       </web-resource-name>
    <url-pattern>/myserver/AccountingServlets/*
       </url-pattern>
    <url-pattern>/myserver/FinanceServlets/*
       </url-pattern>
    <http-method>POST</http-method >
    <http-method>GET</http-method >
  </web-resource-collection>
  <auth-constraint>
    <role-name>owner</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>INTEGRAL
      </transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Violation of the `auth` constraint will result in either in HTTP 401 (if unauthenticated) or HTTP 403 (if authenticated, but ACL-rejected) status code being returned to the caller. For cases of anonymous web users, the Web application's deployment descriptor may contain a `<run-as>` element, which will specify the identity that will be used to process the request. If it is specified, the Servlet container is required to propagate this security identity in calls to the EJB layer, whether in the same or different J2EE application, as was explained in the `Identities` section.

Principal checks may be performed imperatively, using one of the methods exposed by `HttpServletRequest`: `getUserPrincipal, getRemoteUser, isUserInRole`. They can be used to provide finer-grained checks than declarative security allows for:

```
public void doGet(HttpServletRequest request,
         HttpServletResponse response) {
  java.security.Principal principal =
           request.getUserPrincipal();
  String user = request.getRemoteUser();
  if (user != null) {
    //have an authenticated user, check his name
  }
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
  if (request.isUserInRole("owner")) {
    //owner of the account
  }
}
```

EJB role-based security is similar to that of Servlets, and can be declarative or programmatic. However, the declarative variant is finer-grained, as it allows access control up to methods-level. Mapping of principals to roles is vendor-specific, but the EJB specification dictates role-based ACL format in the bean deployment descriptor, with `*` as a wildcard for all permissions:

```
<assembly-descriptor>
  <security-role>
    <description>Role description</description>
    <role-name>UserRole</role-name>
  </security-role>
</assembly-descriptor>

<method-permission>
  <role-name>UserRole</role-name>
  <method>
    <ejb-name>UserAccess</ejb-name>
    <method-name>*</method-name>
  </method>
  <method>
    <ejb-name>OwnerAccess</ejb-name>
    <method-name>getUserInfo</method-name>
  </method>
</method-permission>
```

Using the `<unchecked>` element in the bean's descriptor will bypass any authorization, even if the `<role-name>` element is also specified.

Some methods may even be excluded from being called at deployment time by specifying exclude list. This list provides directive from the application assembler to the Deployer that these methods should be configured to deny any access:

```
<exclude-list>
  <method>
    <ejb-name>SomeBean</ejb-name>
    <method-name>problematicMethod</method-name>
  </method>
</exclude-list>
```

Alternatively, the principal's attributes can be accessed from the bean's code, using methods exposed by `EJBContext` class. Note, that those methods may be invoked only in the EJB business methods with security context present — otherwise, a `java.land.IllegalStateException` will be thrown. Also, both the `getCallerPrincipal` and `isCallerInRole` methods from `EJBContext` always operate on the caller identity, even if `<run-as>` attribute was specified.

```
public class UserAccessBean
                 implements SessionBean {
  EJBContext beanContext;

  public void getUserInfo() {
```

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

```
    java.security.Principal principal =
            beanContext.getCallerPrincipal();
    if (beanContext.isCallerInRole("UserRole")) {
      //authenticated user
    }
  }
}
```

Note: For extended access checks, both systems provide an adequate level of declarative support.

Chapter 4 – Conclusions

This section addressed the user authentication and authorization features of Java and .NET platforms. .NET suffers from tight integration with IIS, without which it is not really capable of performing authentication. In terms of access control, it does provide a convenient mechanism that meshes nicely with its CAS features. Java, in addition to the standard authentication types, offers the powerful JAAS mechanism as its primary vehicle for adding authentication and `Principal`-based authorization to Java applications, which adds a lot of flexibility to the design choices.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Conclusion and Summary

The following table summarizes the items about both platforms which were highlighted previously, and assigns a crude score for each reviewed feature. The points, following each category, are present for helping to identify some issues or referring to strong points of each platform, but at the end they are not simply summarized to arrive at the final score. Many items, especially when both platforms provide similar solutions or face similar issues, are not listed here — it is necessary to consult the appropriate section to see the complete picture and understand the reasoning behind these scores.

Category	.NET	Java
Configuration	*Fair*	*Excellent*
Multiple installs	- Single shared configuration for each installed version	+ Configurable multiple installations
Command-line	- No command-line overrides	+ Multiple command-line overrides
Code Containment	*Very Good*	*Good*
Verification	+ All code is verified	- Local code is not verified
Runtime checks	- Combination of static analysis and added verification code	+ Bytecode stack preserved for checks
Isolation model	+ Well-structured and comprehensible	
Languages	+ An additional runtime constant modifier	- Compile-time constants only
Cryptography	*Good*	*Good*
Structure	- Heavily relies on Windows	- All providers have to be signed by a trusted CA, architecture dictated by the obsolete US export law
Algorithms		+ Significant community support
Instantiation	+ Creation by of algorithms by names, strong defaults	+ Allows choice by specifying additional parameters
Secure Communication	*Fair*	*Very Good*
Platform	- No support besides IIS, some samples available	+ JSSE as a standard component of JDK
Application	- No support	+ Standard GSSAPI implementation
Web Services	+ Up to date support of WSA	- Only supported by external vendors
Code Protection	*Good*	*Good*
Certificates	- Poor default functionality (WSE corrects the problem)	+ Solid and easy API

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Code signing	+ Choice of `strong names` and publisher signing	+ JAR allows multiple signers
Access control		+ Configurable protection policy
Code Access Security	*Excellent*	*Good*
Permissions	+ A rich permission set	
Policy	+ Very sophisticated policy structure	+ Unlimited number of policies, command-line overrides
	- Machine-wide policy may cause conflicts between applications	
Access control	+ Very fine-grained and powerful mechanism	- Privileged code grants all-or-nothing permissions
	- `FullTrust` is granted to all local code by default	
User Authentication	*Good*	*Very Good*
Structure	- Heavily relies on Windows and IIS	+ Part of J2SE/J2EE specifications
Authentication modes	+ Multiple modes out of the box	+ Very flexible mechanism
Impresonation	+ Integration with Windows	- Application-level only
		+ GSS enables application-level delegation
User Access Security	*Good*	*Excellent*
Types of checks	+ Explicit and permission-based — in code, declarative in ASP.NET	+ Declarative checks in J2EE, explicit — in code
Dynamic policy	- Static policy evaluation	+ Dynamic policy evaluation and user-based permission grants

Table 1.*Platform feature summary*

Closing Comments

There are several specific points that can be taken from the above table, as well as from reading the previous chapters. Java usually provides a much more configurable and flexible solution, while .NET designers in many cases were able to simplify the subsystem's structure and API. A well-known downside of .NET is its tight bounding to the underlying system and reliance on its services — `CryptoAPI` and IIS are examples of this.

Overall, the picture seems to be quite spotty, because each platform has its strengths and weaknesses: when issues like communication security and user access control start coming into play, Java seems to be a good choice, while .NET provides a far more superior mechanism for doing code access security and did the right thing with introducing *strong names* (which also helped to address versioning).

Despite few problematic areas, both .NET and Java have been consistently earning high marks on various independent security comparisons, which should not be too surprising,

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

considering the long history of Java in the enterprise and the amount of effort that Microsoft is putting into making .NET a premier Windows development platform. Traditionally, and .NET is not an exception, Microsoft products have done best in the closed homogenous environment of all-Windows networks, while Enterprise Java performs quite well in heterogeneous environments. If we consider an all-Microsoft network, its services allow system integration and utilization of .NET's security features to their fullest potential. In the case of a mixed environment, Java's platform-independent security features may be more useful than those of .NET.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Epilogue – Upcoming Security Features

Both platforms are expecting significant new releases (J2SE 1.5 and .NET "Whidbey") in the current year, and a brief preview of the upcoming security features is attempted in this epilogue. As before, we will focus on the security features of the platforms themselves, avoiding more broad discussion of additional products and services. The security features will be reviewed by protection categories, to ensure that similar items are compared in each case and to see how they augment the existing functionality.

Summary of Future Security Features: Java

In terms of Java security, this is an evolutionary, rather than a revolutionary release, because it does not bring any new, ground-breaking changes. The updates are mostly concentrated in the areas of cryptography and PKI.

Specifically, JSSE experiences a significant change:

- A new SSL/TLS abstraction layer is added to separate its logic from threading and I/O issues.
- JSSE will now use JCE providers exclusively.
- External provider pluggability will be allowed.
- By default, will use a X.509 PKIX-compliant `TrustManager` that is based on `CertPath` provider.
- Default *SunJSSE* provider will include support for Kerberos suites.
- AES_256 cipher suites will be enabled by default in the *SunJSSE* provider.

JCE is going to see some new functionality as well:

- PKCS#11 provider will be included, which adds support for hardware-based accelerators and smartcards.
- New APIs for ECC will be added.
- *SunJCE* provider will include RSA encryption and several additional algorithms.
- Several parameters will be added or enhanced to provide support for XML Encryption algorithms.
- On Solaris, better integration with the OS' cryptographic framework will result in significant performance improvements.

Java PKI is going to be updated as follows:

- Smartcard-based keystores are going to be available, thanks to the added PKCS#11 provider.
- Enhanced PKCS#12 implementation will be included.
- Client-side support for *On-Line Certificate Service Protocol* (OCSP) will be added, and APIs for indirect CRLs will be extended.
- `CertPath` implementation will be PKIX-compliant.

JAR files will benefit from the newly added support for *Time-Stamp Protocol* (TSP), which will enhance the verification of archives.

The *Simple Authentication and Security Layer* (SASL) will be supported through new APIs and a SASL provider for JCA.

Finally, the JAAS Kerberos module in 1.5 will have an option for TGT renewals, which should help avoid unnecessary service re-authentications.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Summary of Future Security Features: .NET

For .NET, more changes are in store in the upcoming release, which address existing shortcomings and add new types of functionality. These changes are quite broad in scope and make a number of significant new features available to .NET developers.

Application Identity Based Security is a .NET buzzword for providing a restricted execution environment, based on information found in application and deployment manifests. New tools will be bundled with the next release to help determine the required application permission set. This identity-based schema is designed to fit into the newly introduced `ClickOnce` programming model in Windows and allow deployment of semi-trusted applications.

CAS will be extended to include demand choices, which will allow for presenting several choices for satisfying demands, and `friend` assemblies will be introduced, similar to `friend` classes in C++.

PKI will be fully integrated in the upcoming release, so falling back on `CryptoAPI` or WSE will no longer be necessary.

In addition to XML Signature, the upcoming release will include support for XML Encryption, both being fully integrated with the new PKI.

File-based Windows access control will be incorporated into the framework, which will allow setting file ACLs from managed applications.

Many of .NET's communication protection pitfalls are going to be fully or partially resolved in the upcoming release:

- Allowing decoupling from IIS via adding server-side HTTP listeners.
- Providing client and server classes for SSL conversation, and means of identity propagation over streams.
- The new messaging framework, codenamed "Indigo", will incorporate all functionality formerly found in WSE into the core framework.

Last, but not least, ASP.NET 2.0 will include enhancements that take care of the drudgery of programming Forms-based authentication and authorization via its new server controls, as well as `Membership` and `Role Management` APIs.

Specifics: Cryptography

In its 1.5 release, Java extends its already quite rich offering in the cryptography space, described in Chapter 2, with several new features.

Probably the most important addition to JCE is a new PKCS#11 provider, which, in contrast to other existing JCE providers that contain cryptographic implementations themselves, simply serves as a bridge to the installed PKCS#11 v2.0 implementations, to enable support for hardware accelerators and smartcards in Java applications. Its introduction caused a number of updates/enhancements to the existing core JCA/JCE and PKI classes, as well as creating new ones, which will allow communication with hardware tokens and smartcard-based keystores. Tools like *Keytool* and *jarsigner* have been updated as well to utilize the extended functionality available with the new provider. Additionally, only on Solaris 10 platform, JCE will take advantage of the *Solaris Cryptographic Framework's* PKCS#11 provider, which will result in significant (i.e., orders of magnitude) performance improvements.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

JCE will be furnished with additional APIs to better support Elliptic Curves (ECC). Users, who previously have had to rely on external providers, can now use a number of standard ECC classes from the `java.security` namespace.

Several classes will be added or extended to support OAEP and PSS padding schemas, as defined in PKCS#1 v2.1 and W3C Recommendations for XML Encryption, enabling full support for RSA-OAEP Key Transport algorithm.

In `javax.crypto` namespace, existing classes are updated to facilitate key-related operations:

- `javax.crypto.EncryptedPrivateKeyInfo` is extended with additional overloads of `getKeySpec` method to enable easier retrieval of private key information.
- `javax.crypto.Cipher` class has new methods that allow retrieval of maximum values for key lengths and parameters.

SunJCE default provider will have several new algorithms, which makes it a more attractive candidate for use in development: HmacSHA (256-512), RSA and RC2 encryption, and additional PBE algorithms.

Java's PKI implementation benefits from the improved PKCS#12 keystore implementation, which will have additional protection algorithms and support keystore read/write operations. This enhancement will substantially facilitate key and certificate exchange, especially when it comes to browsers, which tend to use PKCS#12 format for these operations.

Client-side support for the *On-Line Certificate Status Protocol* (OCSP), conforming to RFC 2560, will be added to PKI. In case of problems with the OCSP operation Java applications will fail-over to the traditional CRL checking via *Certification Path API*, which now boasts full PKIX compliance after passing the *Public Key Interoperability Test Suite* (PKITS) .

.NET's story with PKI has been quite spotty up until now (see Chapter), to say the least. The upcoming release brings the long overdue integration of full PKI into the .NET framework, exposing managed implementations of Windows APIs for X509 and PKCS#7. Support for the former includes newly updated `X509CertificateEx`, which essentially brings the features of X.509 certificate class from WSE into the core framework, and allows access to all certificate properties, as well as validation and chaining. Added support for PKCS#7 means easier interfacing to cryptography applications, written in other systems, particularly in Java, which already supports PKCS#7.

Continuing with its general XML push, .NET adds a fully W3C compliant implementation of the XML Encryption recommendation. This implementation provides most popular symmetric and asymmetric algorithms, such as 3DES, multiple AES, RSA, and is flexible enough to allow encryption of multiple sections inside one document with different keys. Both the existing classes for XML Digital Signature and the new ones for XML Encryption take advantage of functions in the integrated .NET PKI to utilize X.509 certificates for their operations.

Specifics: Secure Communication

The already-thorough Java offerings for communication protection at the platform and application levels, described in Chapter 2, are updated by enriching the existing JSSE model and further extended by adding SASL support.

Java's platform-level mechanism, JSSE, undergoes a significant facelift with an addition of a `SSLEngine` (fully described in the JSSE Reference Guide Addendum), which nicely abstracts

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

the logic of SSL/TLS layer and allows advanced applications to take complete control of I/O and threading issues. At the same time, the simplicity of traditional socket-stream SSL programming has been preserved, and the `SSLSocket` class still implements that functionality.

An important change for JSSE is switching to using JCE providers exclusively — in version 1.4, it still contains internal cryptographic code, which will be gone in the new release. Thus, JSSE will be able to take advantage of any configured JCE provider, including hardware accelerators.

Additionally, as a result of relaxing US export restrictions, JSSE will now allow plugging in external providers, which should support a specific set of cipher suites. The JSSE Reference Guide Addendum contains the complete listing of required ciphers.

Kerberos suites have been included in the default *SunJSSE* provider, which provide support for Kerberos-based TLS communication, as described in RFC 2712.

At the application-level, the existing GSS provider is augmented by adding a SASL implementation, which provides a lightweight authentication and security services for network communication. SASL is utilized by several popular modern Internet protocols, among them LDAP v3 and IMAP v4. Its advantage over JSSE and GSS lies primarily in very lightweight infrastructure requirements, whereas those two require complex setup, like PKI and Kerberos. However, it is expected that JSSE, GSS, and SASL mechanisms will be layered on top of each other in many implementations.

.NET, which has been clearly lagging in the communication space (see Chapter 2), finally catches up by providing a managed SSL/TLS implementation that is capable of protecting TCP-based socket-level communication. It operates via the `SslClientStream` and `SslServerStream` classes, and is very similar to the model used in Java. Fortunately, .NET does not stop at this, and adds support for a standalone HTTP listener (implemented in class `HTTPWebListener`), essentially removing mandatory application reliance on IIS for web interactions. Actually, a similar class has existed in .NET's Remoting infrastructure since v1.0 — it was used in standalone applications under the covers for exposing Remoting services on HTTP channels. Now its functionality will be upgraded to include support for authentication and SSL.

The new "Indigo" messaging framework will incorporate all security standards from *Web Services Architecture* (WSA) released to date, which were formerly found in WSE. It further generalizes the concept of secure messaging by applying similar message-based security protection to different types of messaging mechanisms, like `ASMX` and `EnterpriseServices`. The "Indigo" framework partitions security functionality in three security layers: *TurnKey*, *Custom*, and *Extensibility*. A majority of applications (80%) is expected to fall into the *TurnKey* category, which requires fully declarative support from the developers by adding declarative attributes and editing policy settings. The latter two categories are used for programmatical customization of the framework at different levels.

Specifics: Code Protection and Deployment

The signing process for Java's main code distribution vehicles, JAR files, has been described in Chapter 3. One of the shortcomings of the existing process is the inability to determine the validity of the archive relative to the certificate's expiration time. The addition of *Signature Timestamps* solves this problem by adding timestamps to the JAR signatures, thus allowing checking whether the signing certificate was valid at the time of signing. The `jarsigner` tool will be updated to include new signing options, and the JAR API in the `java.security` package will extended with new classes and methods to access the timestamp information.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

.NET extends its existing deployment model (XCOPY or MSI-based) by introducing the *ClickOnce* deployment and update mechanism for server-based installations. It is based on using signed manifests and deployment files, similar to the model J2EE has been using for EJB deployments.

However, there are significant improvements waiting ahead — for details of the manifest, see the preliminary MSDN Longhorn documentation. First, the application manifest will include security requirements of the application (not those of individual assemblies). These trust requests and the application's evidence are evaluated by the newly-introduced `TrustManager`, and presented for the user's consent if an application requires additional permissions outside of the *Secure Execution Environment* (SEE). The results of this evaluation and the user's decision are stored on per-user and per-machine levels and used later for CAS decisions (see CAS section). Secondly, deployment manifests specify update policy, which allows secure, XML Digital Signature-based application updates. An interesting detail about the manifest's structure is that it is the application manifest itself that is signed, and not the individual assemblies comprising the application — they are represented in the manifest by their digests. In this respect, structure of .NET's application manifest resembles signed JAR files in Java.

The designers of .NET's *Base Class Library* (BCL) introduced a new feature for limiting code exposure: `friend` assemblies. The premise is that the internal classes in a particular assembly are declared to be accessible from another assembly, referenced by its `PublicToken`, much like how the `friend` declaration works in C++ for classes. Following the general .NET approach, this extension is introduced via a new assembly-level attribute, `InternalsVisibleTo`.

Specifics: Code Access Security

.NET's CAS model, described in Chapter 3, offers an excellent framework for code access security. However, it is rendered completely unusable for locally installed applications, whcih are blankly granted `FullTrust` by the default policy, meaning that **any** CAS permission check will succeed. The upcoming release of the .NET framework includes the `ApplicationSecurityManager` and `TrustManager`, which will make the decision of granting application trust requests based on the machine policy in effect and the application manifests (see the Deployment section).

At the same time, developers are urged to develop their applications targeting the `Internet` permission set, as an application sandbox with low trust and a "safe" permission set (preliminary named *Secure Execution Environment*, or SEE) will be introduced in the Longhorn Windows OS (due in 2005) for executing applications. However, according to the preliminary MSDN Longhorn documentation, Longhorn's security system, as it is presently designed, does not attempt to verify trust of local `exe` files which do not have deployment manifests, and simply grants them the same `FullTrust` as before. Hopefully, this policy will change by the release time, because, with its present design, this setup presents an unfortunate way to bypass the system checks in local scenario.

To allow several choices for attribute-based CAS demands, CLR adds `DemandChoice` and `LinkDemandChoice` actions. Their logic is similar to the ordinary demands, but they allow specifying several attributes with different permission sets. Satisfying any of them is sufficient for the success of the overall check.

Specifics: Authentication and Authorization

Java's flexible Authentication and Authorization Service, JAAS, has been reviewed in detail in

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Chapter 4, which offers `Krb5LoginModule` among other so-called *Pluggable Authentication Modules* (PAM's). Presently, it does not include an option for TGT renewal, which causes their expiration in long-running services and requires either restarting the process or user intervention for re-authentication. Setting the newly introduced configuration option, `renewTGT`, to `true` will now result in automatic TGT renewals whenever expired tickets are retrieved from the ticket cache.

ASP.NET 2.0, which remains the primary Web development platform for .NET, brings a whole slew of improvements for its existing Forms-based authentication model (see Chapter 4). Most importantly, it introduces *Membership* and *Role Management* APIs (found in `System.Web.Security` namespace), which take care of tedious programming tasks by essentially eliminating, or significantly reducing the need for, writing security plumbing.

The *Membership* API takes care of the issues commonly present in password-based systems, like secure credential operations (CRUD), finding and authenticating users, and password management. *Role Management* API, based on ASP.NET Role model in 1.x, works together with the *Membership* API (although it can be accessed separately) to solve user-to-role mapping issue and can be used programmatically and declaratively, in `Web.config`. Both APIs use a Provider Model design pattern, and are highly extensible (providers for SQL Server and Access are included in the default installation). If these APIs eventually are made available outside of the ASP.NET umbrella (like WSE Pipeline), they will provide a great and flexible addition to many types of .NET applications besides Web-based ones.

New server-based GUI controls for ASP.NET take advantage of these new APIs, further reducing the amount of required programming, often making it as simple as dropping the controls on the form. The following controls will be made available, among others, in the new release: `Login`, `LoginName`, `LoginStatus`, `LoginView`, and `PasswordRecovery`.

Conclusions

Java does not offer any significant new features in the upcoming release, extending instead its offerings in existing categories.

.NET, on the other hand, aggressively pursues new security functionality. It will catch up with (or even pass) Java on several topics where Java currently holds an advantage, and extend its lead in the areas of its dominance. Most prominently it catches up in the categories of communication protection and PKI, and it goes one more step ahead by adding full support for the W3C XML Encryption recommendation.

However, it is worth noting that with incorporation of .NET into the core Windows OS (starting with Windows 2003 Server), it is becoming progressively harder to distinguish .NET-specific features from OS features, as in the upcoming versions they are often designed to complement each other. This confusion might stem from the fact that Microsoft authors and spokespeople often do not specifically distinguish between the two in their publications and presentations, thus muddying the overall picture.

Incidentally, adding support for the WSA family of standards into the core .NET libraries prior to submitting them for approval by OASIS (with the exception of WS-Security, which already goes through the technical committee review) means that Microsoft, probably does not expect any significant changes to the released set of specifications, or does not intend to submit them at all. Its intentions remain to be seen, but they certainly do not make life easier for Sun's developers, who have to shoot at a moving target in this case.

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Bibliography

- Dovydaitis, V. and Piliptchouk, D. (2002) Enterprise Java and .NET security side-by-side. In the *Computer Security Journal*, Vol XVIII, Numbers 3-4, 2002.
- LaMacchia, B. et al (2002) *.NET Framework Security*. Addison-Wesley, Boston, MA.
- Gough, J. (2001) *Compiling for CLR*. Prentice Hall
- Box, D. (2002) *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, Boston, MA.
- Garms, J. (2001) *Professional Java Security*. Wrox Press.
- Roman, E. (2001) *Mastering Enterprise JavaBeans (2nd Edition)*. John Wiley & Sons.
- Microsoft MSDN website
- Microsoft GotDotNet website
- Microsoft ASP.NET website
- .NET 247 website
- Sun's Java website
- IBM's developerWorks website
- IBM's alphaWorks website
- TheServerSide J2EE website
- JavaWorld website

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

About the Author

Denis Piliptchouk

Denis is a senior software architect with SunClinical Data Institute (a division of Eclipsys Corporation) with thirteen years of expertise architecting and securing Enterprise systems for healthcare, semiconductor, and avionics industries. He holds MS in Computer Science, has been involved in security research projects, has publications and presentations in the area of application security. He can be reached at dpiliptchouk@hotmail.com

Copyright © 2004 O'Reilly Media, Inc. All rights reserved.