

CUADERNOS TÉCNICOS
dotNetManía



cuaderno técnico n 3

Programación segura con .NET Framework

Marino Posadas

3

SEGURIDAD EN .NET FRAMEWORK

Consideraciones de seguridad en el ciclo de vida de las aplicaciones.

Marino Posadas
MVP Visual C#
MCSD .NET, MCAD, MCT

MANUALES PRÁCTICOS PARA EL DESARROLLADOR

Colección Guías Prácticas Netaia

© Netaia Ediciones S.L.
C/ Robledal, 137 • Madrid
Teléfono (91) 555 2282 • Fax (91) 555 9199

Tabla de contenido

-pendiente

Capítulo 1: Introducción a los conceptos de seguridad

La seguridad es un problema de nuestros tiempos. Desde hace algunos años, todas las actividades cotidianas están teñidas de algún concepto asociado a la seguridad. Desde los simples juguetes, hasta los automóviles. El software, no iba a ser una excepción. Las aplicaciones de hoy no son elementos aislados, y no se trata siquiera de que nuestra aplicación funcione correctamente, sino de que otras aplicaciones no interfieran con la nuestra en forma no deseada. O que los usuarios no puedan realizar operaciones no previstas. Por otro lado, el software de hoy necesita estar interconectado, integrarse, emitir y recibir información fuera de los límites de su proceso de ejecución y muchas veces, fuera del sistema operativo sobre el que se ejecuta.

Los primeros virus informáticos aparecidos en los 90, causaban más curiosidad que miedo y tenían como medio de transmisión la inserción directa en un medio físico portátil (disquete, casi siempre). La situación hoy es muy diferente, debido a la aparición de Internet, que, si bien se ha convertido en imprescindible, al tiempo, ha abierto la Caja de Pandora donde todos los truenos se esconden dispuestos a impactar contra nuestros sistemas.

¿Estamos hablando de virus? No, realmente. Hablamos de seguridad, y eso, va más allá de la influencia de Internet. No es necesario en absoluto que exista infección vírica para que se violen los principios de la seguridad. Cualquier persona que acceda a información cuya naturaleza no le compete o que –simplemente– no haya sido expresamente autorizada, está violando la seguridad. Cualquier dato que salga de los límites para los que ha sido concebido, está excediendo su contexto de seguridad. Un empleado que se apropia de código fuente indebidamente, puede atentar contra la seguridad una vez abandonada su empresa (o incluso mientras permanece allí).

En general, cualquier proceso utilizado para fines para los que no ha sido diseñado, o sometido a cargas de trabajo para las que no está preparado (como los ataques **DoS**, o **Denial of Service**) puede convertirse en un peligro para la seguridad. En la práctica, cualquier entrada de datos puede ser potencialmente dañina, si el usuario conoce (o es capaz de inferir) el funcionamiento interno del programa: de hecho, dos de los tipos de ataque más conocidos se basan en esto: "**Buffer overrun**" y "**SQL-Injection**".

Incluso existen factores no intencionados en absoluto que pueden provocar un mal funcionamiento de un sistema (o dispositivo) simplemente, por que el programador no ha previsto que la capacidad humana de error (casi infinita) puede provocar dicha situación.

Un ejemplo: Pongamos un caso en extremo inocente: un programa solicita en una caja de diálogo de impresión cuántas copias queremos de un documento, y, por cualquier razón, le indicamos 9999 ó 32767 (valores máximos autorizados para la mayoría de impresoras) en lugar de 9. O que la impresora permite una configuración de tipo de letra, a la que asignamos un tipo vectorial de i500 puntos! Estaremos forzando al sistema más allá de los límites entre los que su funcionamiento es adecuado. No son -en sí mismos- agujeros de seguridad intencionados (por ninguna de las partes, muchas veces), pero pueden provocar efectos totalmente indeseados.

Como vemos, la naturaleza de los ataques a la seguridad es muy variada, y, de hecho, también lo es la defensa contra dichos ataques. Buena parte de esa defensa, tiene que ver con la administración del propio sistema operativo, pero nuestro código puede, en ocasiones, permitir lo que el propio sistema impide y eso supone un planteamiento serio de todo el modelo de amenazas.

Esta obra va a centrarse en la seguridad del código fuente y utilizará como modelo la plataforma .NET Framework 2.0, que nos servirá de laboratorio de pruebas, en muchas ocasiones. La mayor parte de lo indicado aquí es perfectamente extrapolable a las versiones anteriores y, cuando no sea así, lo indicaremos de forma explícita.

Proponemos un recorrido por el ciclo de vida de las aplicaciones (haciendo especial hincapié en la fase de desarrollo), recomendando qué aspectos de seguridad debieran de tenerse en cuenta, cómo mitigar los efectos indeseados y qué medidas preventivas nos

parecen imprescindibles para ofrecer el adecuado nivel de calidad en las aplicaciones que queremos para nuestros clientes.

Aviso: Debemos de tener en cuenta que es un error pensar que, una vez que la aplicación esté terminada, si observamos un fallo de seguridad, será sencillo poner un parche para arreglarlo. A veces, puede significar un recorrido por el ciclo completo.

De hecho, cuanto más tarde se arreglen las cosas (en cualquier aspecto, en general), peor. De ahí que, este paseo rápido por la seguridad, recorra todo el ciclo de vida y no sólo la fase de codificación, de tal forma que el lector pueda tener una guía básica pero global de los problemas de seguridad y sus soluciones potenciales.

Tampoco es nuestro objetivo revisar los mecanismos de seguridad perimetral propios de los sistemas operativos. En los apartados adecuados, indicaremos qué parte de la arquitectura de software es la responsable de la seguridad, y en algunos casos, cómo modificarla sin recurrir al administrador del sistema. Mucho y bueno hay escrito al respecto, y solamente asumiremos que, en lo fundamental, se conocen los mecanismos básicos o sabemos la forma de solicitar su implantación por el encargado de la administración del sistema.

La iniciativa de Informática Fiable (Trustworthy Computing)

Para abordar el problema de la seguridad con fundamento, un grupo de grandes empresas, entre las que se encontraba desde sus inicios Microsoft, propusieron una iniciativa de "Informática fiable", que, - como puede leerse en su documento oficial¹- establece que se debe perseguir la meta de que los sistemas cumplan con estos cuatro principios:

- **Seguros:** ningún virus atentará contra nuestros sistemas o los volverá inutilizables.
- **Privados:** La información personal no será expuesta de ninguna forma, ni podrá ser utilizada en manera alguna que no sea la explícitamente indicada en el momento de su exposición.
- **Fiables:** cuando instalemos un programa, no tendremos que preocuparnos por los posibles efectos colaterales que pueda tener sobre otro software instalado.

¹ Disponible en http://www.microsoft.com/mscorp/twc/twc_whitepaper.msp

- **Integridad de negocio:** Mi proveedor de servicios responderá rápida y efectivamente cuando informo de un problema.

Uno de los frutos de esta iniciativa ha sido la colaboración y el acuerdo, respecto a la distribución de software, de conseguir una nueva generación de productos basándose en estos patrones.

Dentro de esta propuesta, hay que destacar, en lo que concierne al ciclo de vida de desarrollo de software, el documento oficial "*The Trustworthy Computing Security Development Lifecycle*" (a partir de ahora, SDL), que pretende establecer las guías a seguir en un desarrollo seguro y definir patrones repetibles que puedan servir de medida de la seguridad del software producido. Adicionalmente, existe un sitio Web (<http://www.microsoft.com/mscorp/twc/default.mspx>) que explica con detalle el conjunto de medidas tomadas por Microsoft para cumplimentar los principios definidos por la iniciativa. El número de acciones para llevarlo a cabo es bastante grande (como es de esperar de una compañía de tal volumen), y sobrepasan con mucho el ámbito de los objetivos de esta obra, por lo que remitimos al lector interesado a la Web antes citada.

Los 3 factores clave de la propuesta SDL son:

- Procesos repetibles (en camino hacia un enfoque más científico del ciclo de vida, los procesos en que podemos descomponer el desarrollo de una aplicación, deberían de poderse repetir siempre, incluyendo aquellos que atañen a la seguridad de la misma)
- Educación en Ingeniería de Software, haciendo especial énfasis en la seguridad, lo que implica la creación de un comité de seguridad dentro de la empresa y de unos responsables de seguridad en el equipo de desarrollo. Naturalmente, tales compromisos implican un conocimiento previo de la materia y una voluntad de continuidad en el esfuerzo.
- Métrica y auditoria: Conjunto de técnicas tendentes a conseguir una medida del avance del proceso en términos de cantidad de proceso realizado y tiempo restante respecto al plazo de finalización programado.

El principio SD³

La forma en que estos principios se formalizan en metodologías, recibe el nombre de **SD³: Seguro por Diseño, Seguro por Defecto y Seguro en el Despliegue**. A lo que hay que añadir un cuarto factor fundamental: Seguridad en las comunicaciones, con lo que la fórmula final adopta la forma **SD³ + C**.

Las dos primeras, son las que mayores beneficios ofrecen. En general, la introducción de estas ideas dentro del ciclo de vida, conducen a un gráfico de proceso similar al siguiente:

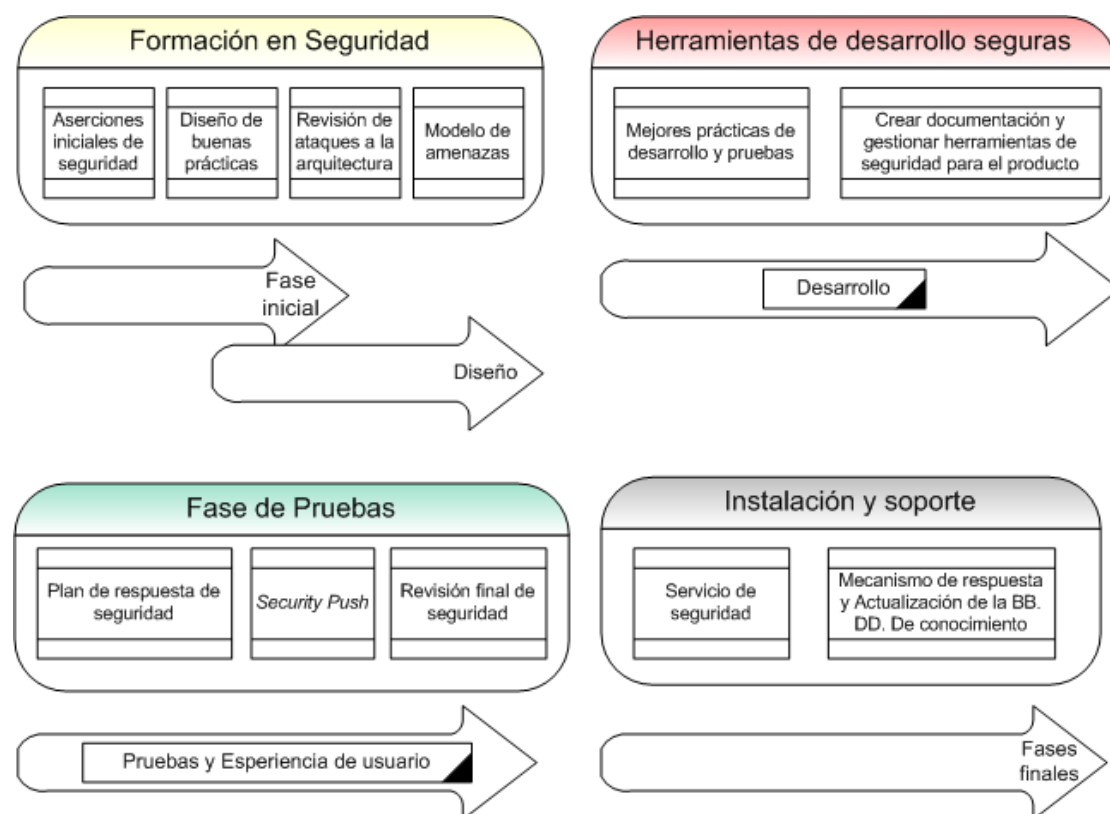


Fig.: Secuencia de acciones de seguridad a lo largo del ciclo de vida

Seguro por Diseño. Al menos 3 puntos a tener en cuenta:

- Arquitectura y código seguros
- Análisis de riesgos
- Reducción de vulnerabilidades

Seguro por Defecto. Se puede dividir en otros 3 aspectos:

- Reducción del posible área de ataque
- Las características no usadas del producto deben deshabilitarse por defecto
- Requerir privilegios mínimos

Seguro en el despliegue. Implica:

- Proteger, detectar, defender, recuperar, administrar...todos estos verbos, asociados con la gestión de la seguridad.
- En la Gestión de Procesos: elaborar guías "Como hacer...?", y guías de arquitectura
- Formación

¿Parece excesivo? Pues no lo es. Basta con recordar lo que **Howard/Leblanc** llaman:

La ventaja del atacante y el dilema del defensor

Como el problema, se plantea de forma global -en su sentido más amplio-, requiere de una estrategia, un análisis y unas acciones de la misma naturaleza. Naturaleza, que, por otra parte, es desventajosa, y que -los citados autores- en su excelente obra² sobre seguridad, denominan "La ventaja del atacante y el dilema del defensor", que se articula en 4 principios fundamentales:

1. El que defiende tiene que defender todos los puntos; el atacante puede seleccionar el más débil
2. El defensor sólo puede defenderse de ataques conocidos; el atacante puede probar nuevas formas de ataque.
3. El defensor debe de estar en constante estado de vigilancia; el atacante puede golpear a voluntad.
4. El defensor debe jugar según las reglas; el atacante puede jugar sucio.

Y para acabar de empeorar las cosas, ni siquiera conocemos la identidad del atacante, que en el peor de los casos podrían estar dentro de nuestra propia "casa": empleados descontentos, espías tecnológicos, o simple dejadez pueden suponer una vía de escape a importante información corporativa.

Naturalmente, nunca estaremos totalmente a salvo, pero sí podremos poner las cosas difíciles a los atacantes y -sobre todo- definir cuáles son los baremos de seguridad que consideramos suficientes en nuestras aplicaciones.

² "Writing secure code" 2ª Edición. Ed. Microsoft Press. Michael Howard y David LeBlanc.

Creación de una cultura de seguridad en la empresa

Los citados autores, van un paso más allá y nos sugieren algunas ideas para instaurar en nuestros equipos de desarrollo una “cultura sobre la seguridad”, tales como: nominar un *especialista de seguridad*, que esté al tanto de los últimos parches y sistemas que ofrezca el mercado, seleccione un *equipo de seguridad* dentro de la empresa, suministre la información o la formación adecuada sobre el tema a los que la necesitan, establezca –incluso– recompensas al código más seguro, o al mejor parche, y finalmente, compruebe las amenazas de seguridad y su severidad para ofrecer consejos sobre la mejor forma de solucionarlas.

Sería el Gestor Principal de Seguridad de la empresa, y, en grandes organizaciones, el encargado de designar y mantener el *Equipo de Seguridad* (*Security Team*), propuesto por las iniciativas de buenas prácticas.

Recomendación: Lo adecuado es que esa cultura forme parte de todos y cada uno de los integrantes de un proyecto de software, y de acuerdo con ese principio, cada rol del equipo de trabajo debería también ser responsable de su “parcela” de seguridad: la que compete ese rol.

Áreas de Influencia

En un documento publicado por Microsoft en 2003³, se establecían un conjunto de principios que regirían a toda la empresa y que serían aplicables a cualquier otra que quisiera abordar con un mínimo de seriedad los problemas de seguridad. En forma esquemática, se dividían en 4 áreas de influencia: la Administrativa, la de Usuarios y Datos, la puramente vinculada al mundo del Desarrollo de Software, y la correspondiente a Operaciones y Mantenimiento. Describimos someramente las competencias de cada una.

Administrativa: Debería asumir un compromiso con la gestión del riesgo y la concienciación de la seguridad. Sus acciones relacionadas serían la gestión del riesgo en función de los objetivos empresariales, la gestión y administración de roles vinculados a la seguridad y el compromiso de trabajo en modo seguro.

³ El documento todavía estaba disponible en el momento de la escritura de este capítulo en la dirección:
http://download.microsoft.com/download/c/3/e/c3e59c9e-9fe9-4cca-b555-dd059e34ac28/La_seguridad_en_Microsoft.doc

Usuarios y Datos: se relaciona con 2 activos fundamentales a nivel corporativo. Para el primero de ellos, se trata de vigilar y garantizar que las personas acceden a la información necesaria para la correcta realización de sus funciones, preservando el resto de la información y tratando de establecer el difícil equilibrio entre seguridad y eficacia laboral. Hay muchos factores que intervienen aquí, pero, comprobar la garantía de identidad y basar las decisiones en la clasificación y el uso razonable de los datos, son dos de las más importantes a este nivel.

Desarrollo de software: Respecto al desarrollo, el objetivo debería ser conseguir la producción de software seguro y de calidad, y para ello, hay que establecer las estrategias de una defensa en profundidad, reduciendo al mínimo la superficie de ataque (recuerde el lector ni siquiera Aquiles se libró de los ataques, aunque su superficie de exposición era mínima)...

Operaciones y Mantenimiento: Por último, integrar sistemas y personas, y lograr establecer los mecanismos a través de los cuales éstas puedan garantizar el funcionamiento de esos sistemas con un mínimo de garantías de seguridad.

Aunque son muchas las posibilidades en este campo, algunas de las acciones típicas podrían basarse en la planificación del mantenimiento, el respeto a las políticas de seguridad, la supervisión y auditoria de los mismos, y, por supuesto, la respuesta adecuada a los incidentes.

En la siguiente tabla, aparecen estos principios expuestos de forma esquemática, y detallados para cada una de las 4 áreas de influencia:

TABLA 1. PRINCIPIOS DE SEGURIDAD

Categoría	Principio de seguridad
Administrativa: Dirigido al compromiso de la administración con la gestión del riesgo y la concienciación sobre seguridad	Gestionar el riesgo de acuerdo con los objetivos empresariales. Definir roles y responsabilidades administrativos. Invertir en el diseño de la seguridad. Comprometerse a trabajar de modo seguro
Usuarios y datos: Incluye la autenticación, la	Practicar el menor número de privilegios.

privacidad del usuario y la autorización de datos	Basar las decisiones en la clasificación y el uso razonable de los datos. Hacer cumplir la protección de la Privacidad y de la Información Personal Identificable. Asegurar la integridad de los datos. Controlar la garantía de identidad. Integrar la disponibilidad.
Desarrollo de aplicaciones y sistemas: Dedicado a diseñar y desarrollar sistemas seguros	Integrar la seguridad en el ciclo de vida. Diseñar una defensa en profundidad. Reducir la superficie de ataque. Mantener la simplicidad.
Operaciones y mantenimiento: Integra las personas, los procesos y la tecnología para construir, mantener y poner en marcha sistemas seguros	Planificar el mantenimiento del sistema. Hacer respetar la configuración de seguridad y su endurecimiento. Supervisar y auditar. Responder ante los incidentes. Verificar la recuperación tras un desastre.

En el citado documento, se concluye con este apartado denominado “las cinco Garantías de Confianza”:

1. Mi identidad no se ve comprometida.
2. Los recursos que necesito son seguros y están disponibles, entendiendo seguro y disponible de este modo:
 - Seguro. Libre de la manipulación y del acceso no autorizado.
 - Disponible. Libre de vulnerabilidades de seguridad y disponible a través de acuerdos de servicio.
3. Mis datos y comunicaciones son privados
4. Entiendo mi papel y soy responsable de mis deberes para garantizar un entorno seguro.

5. Recibo respuestas adecuadas a los riesgos que me afectan.

Y todo esto, sin que abandonemos nunca nuestra mejor arma: el sentido común. Si una buena práctica, una recomendación del *gurú*, o cualquier otra consideración al respecto no encajan con el **conjunto coherente** de las medidas diseñadas, es que probablemente no es aplicable en nuestro caso. No se preocupe, no pasa nada.

Capítulo 2: Seguridad en el Ciclo de Vida de un proyecto de software

Dado que existe una iniciativa en Microsoft vinculada al marco de trabajo sobre el que construir un proyecto de software y que ésta aborda todo su ciclo de vida, utilizaremos los principios recomendados por ese marco de trabajo: **Microsoft Solutions Framework (MSF)**, que establece las buenas prácticas a seguir en la construcción de un proyecto. No se trata de ninguna herramienta, sino de un conjunto de normas tendentes a conducir por el camino correcto el proceso de construcción de un proyecto de software.

Los principios básicos de MSF

Para los no conocedores del tema, los resumimos muy brevemente: MSF establece 3 modelos necesarios que deben seguirse en el ciclo de vida de una aplicación:

Modelo de roles de aplicación (*Team Model*): que establece quiénes, con qué funciones y en qué condiciones serán los responsables de la ejecución del proyecto. Por supuesto, también establece cuáles serán las responsabilidades de cada rol (fíjese el lector que una sola persona puede asumir más de un rol, y que esos roles pueden escalar hasta extremos como los que llevan a la propia Microsoft a formar equipos de desarrollo cercanos a las 3000 personas (Visual Studio 2005) o incluso el doble (como en Windows Vista).

Modelo de Riesgos (*Risk Model*): Establece cuáles son los peligros a los que potencialmente se puede someter todo el proceso. Aquí, se asumen como tales, desde la deserción de un programador crucial en el proyecto, hasta la falta de un determinado "Service Pack" en los equipos de distribución final. Cada riesgo es ponderado mediante un par de valores que se multiplican, y es publicado (idealmente, en una Intranet), de forma que todos los integrantes del equipo puedan efectuar el seguimiento apropiado. Cada riesgo tiene un propietario (vinculado a los roles), y la responsabilidad de ese propietario es mantener el valor de riesgo lo más bajo posible, tomando las acciones adecuadas.

- La fórmula utilizada en la evaluación de los riesgos es el resultado de multiplicar el impacto de cada riesgo potencial (efectos que tendrían lugar si el riesgo se convierte en realidad, que valoramos mediante un número entero acotado entre 1 y 10, por ejemplo) por la probabilidad (su concepto matemático, ponderado entre 0 y 1):

$\text{Riesgo Potencial} = \text{Impacto} \times \text{Probabilidad}$

Las acciones a realizar en cada caso son de dos tipos:

- **Mitigación** (acción preactiva, para evitar que suceda)
 - Este será uno de nuestros principales focos de trabajo, ya que la seguridad del código consiste precisamente en escribir software que establezca esas medidas en prevención de posibles ataques.
 - Las técnicas para conseguirlo son muy variadas, pero pueden agruparse en una serie de directivas y recomendaciones bien determinadas que examinaremos con detalle más adelante.
- **Contingencia** (acción reactiva, en caso de que el riesgo deje de serlo y se convierta en realidad). También aquí es labor del software prever en lo posible esta circunstancia, especialmente, teniendo en cuenta que, una vez instalado, es labor de la propia aplicación dotar al usuario de los mecanismos necesarios para una recuperación y puesta en marcha de sus servicios.

Los riesgos deben de estar claramente definidos

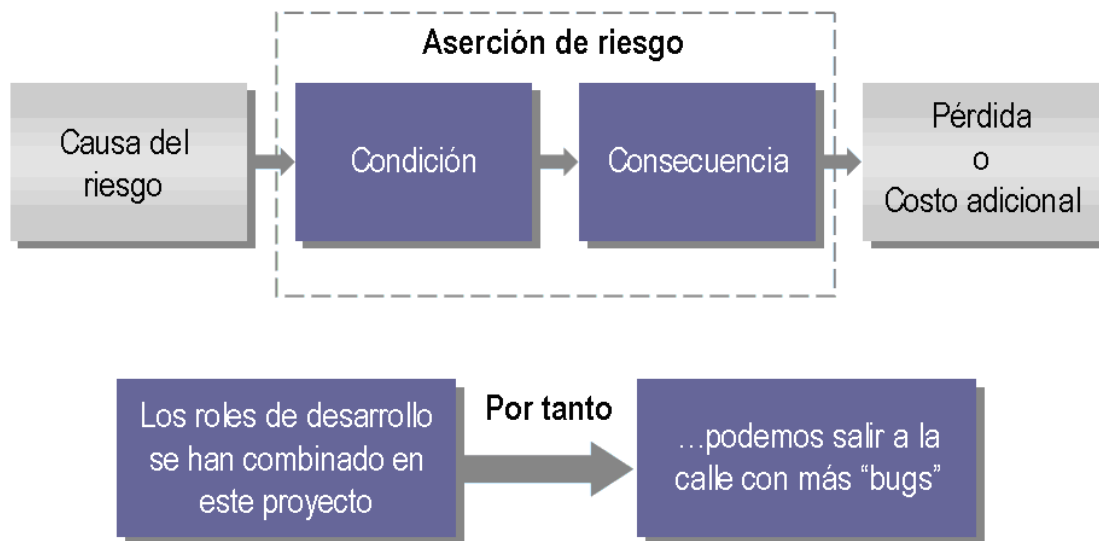


Fig. xx: Proceso de evaluación de un riesgo

El Modelo de Amenazas: Establece cuáles de esos riesgos están vinculados con la seguridad, y cuál es la forma adecuada de tratarlos. En las primeras versiones de MSF, este aspecto no se contemplaba como requisito obligatorio (si bien se recomendaba), pero la situación ha variado mucho, aunque éste punto merecerá un tratamiento aparte.

El Modelo de Procesos: Es el modelo principal, y engloba de cierta forma a los anteriores. Consta de 5 fases:

- Previsión (*Envisioning*)
- Análisis y diseño (*Design*)
- Implementación (*Development*)
- Depuración y Pruebas (*Testing*) y,
- Distribución y Mantenimiento (*Deployment*).

Como es lógico, los dos primeros modelos (Equipo de trabajo y Gestión de Riesgos), son resueltos en una fase inicial dentro de la etapa de Previsión, al establecer el equipo de desarrollo, sus roles y responsabilidades, junto a la primera edición del Modelo de Riesgos y Amenazas.

Podríamos decir que todas las fases tienen una naturaleza dinámica, de forma que se encuentran constantemente en evolución a lo largo del ciclo de vida de la aplicación. Así pues, tras una primera fase, en la que se constituyen los documentos iniciales de trabajo y se configura el equipo de desarrollo y la primera versión del Modelo de Riesgos, ya podríamos empezar a tomar cartas en el asunto, desde el punto de vista de la seguridad.

El gráfico siguiente es una explicación de esta situación en forma esquemática:

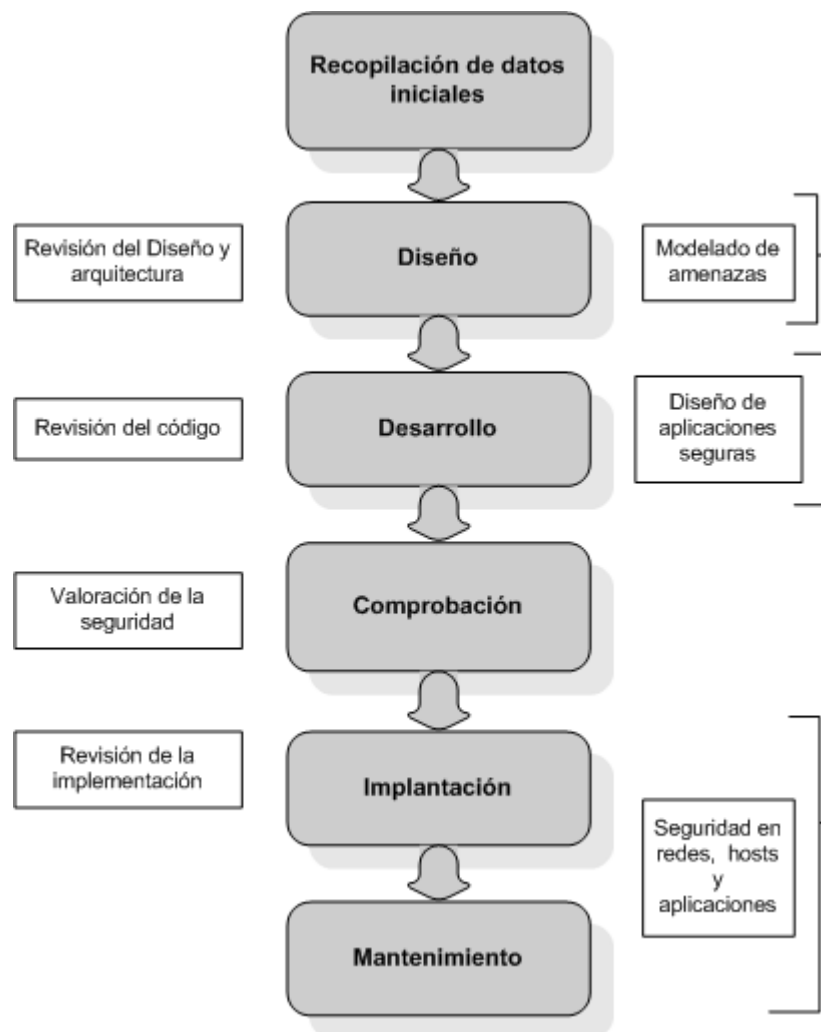


Fig. xx: El ciclo de vida de una aplicación y los modelos de seguridad

Recomendaciones generales para cada fase del ciclo de vida

A la vista del ciclo completo de desarrollo, hay un par de ideas previas a tener en cuenta: la primera, es que no existen reglas absolutas para todo tipo de aplicaciones: deberemos de aplicar los principios de seguridad de forma individual –al igual que lo hacemos con los análisis y diseños- y, aunque muchas ideas coincidan, la implantación final será específica en cada caso. La segunda es que algunas soluciones de seguridad se propagarán a lo largo de todo el ciclo de vida, mientras que otras pueden estar circunscritas a una fase concreta. No pensemos pues, que por haber tocado un aspecto de seguridad en la fase inicial, podremos olvidarnos de él.

Seguridad en la confección del Equipo de Trabajo

Aunque el sentido común nos dictará ciertas normas adicionales cuando formemos el equipo, la fase de Previsión requiere de la designación del Gestor de Seguridad mencionado antes. Si tal figura ya existiese, sería él, junto a los responsables del proyecto, quienes deberían establecer las pautas de seguridad a seguir en el ciclo. Incluso si –por el momento- no pensamos hacerlo a nivel empresarial, sí resulta imprescindible hacerlo para el proyecto, aunque suponga la asignación de un rol múltiple, asignado a una persona ya integrada en el equipo de trabajo.

Este responsable de seguridad estaría encargado de proporcionar los recursos y el soporte adecuado de seguridad a lo largo de todo el ciclo de vida. Sus principales tareas, tras los supuestos iniciales son:

- Asesoramiento sobre las características específicas de seguridad en la aplicación (en función de la complejidad, tamaño y riesgos evaluados)
 - Dicho asesoramiento puede adoptar varias formas, pero uno de los pasos iniciales lo establecen las entrevistas sobre seguridad entre los responsables generales de seguridad y el equipo de trabajo. En esas entrevistas, se manejarán conceptos como nivel medio de los usuarios de la aplicación, la existencia de conectividad y/o cortafuegos asociados, y todas las implicaciones de compromiso, ya fuese en acceso no autorizado, problemas ante posibles usuarios con elevación de privilegios, infraestructura de servicios del sistema que requiere la aplicación, y –evidentemente- en qué forma debe el usuario ser defendido de los problemas potenciales derivados. En la literatura existente en Internet pueden encontrarse catálogos de preguntas típicas de esta fase.
- Consejo: Una buena práctica de seguridad puede consistir en desarrollar con un nivel de privilegios igual que el del usuario final, y siempre –salvo en caso de necesidad- lo más cercano posible a ese nivel y no al del administrador del sistema.
- No es necesario decir que conocer los mecanismos asociados con la seguridad requiere de una serie de conocimientos y entrenamiento. Un buen candidato es aquel que se ha interesado en características de

“hacking” con anterioridad o que investiga a fondo los sistemas.

- Respecto a la formación, es más que probable que encontremos reticencias iniciales (“No hay tiempo para esas cosas”, etc.). Habrá que sopesar cuál es prioritario: entregar al cliente un sistema suficientemente robusto, o afrontar las consecuencias.
- La revisión de viejos defectos, solucionados en forma de parches temporales, que hayamos podido “heredar” de aplicaciones anteriores.
- Establecimiento de hitos de seguridad, de la misma forma en que se establecen hitos como separadores de las fases y etapas que configuran MSF (los hitos son puntos de inflexión, que señalan la consecución de un conjunto de objetivos del Modelo de Procesos, o, en general, cualquier modelo concebido en fases).

- Consejo: Los hitos de seguridad deberían de ser revisados por todo el equipo. Tras la consecución de un hito, no estaría demás, una breve reunión *post-mortem*, para establecer cuáles han sido los problemas de seguridad resueltos en esa fase, y qué enseñanzas se obtienen de ese trabajo de cara al futuro del proyecto y de otros venideros.

- Desde un punto de vista práctico, esto supone, la identificación clara de los objetivos de seguridad previstos para el proyecto, (y los momentos en que tales objetivos serán evaluados)
- Establecer la forma en que la seguridad se integrará con el software que utilicemos en su construcción, o que deba convivir con la aplicación como requisito.
- Un buen punto de partida es el conocimiento exacto de las cotas de seguridad que ofrecerá el sistema de destino: qué políticas de grupos se seguirán (o podrán seguirse después de la implantación), configuración de cortafuegos, configuración de IIS, etc. El conocimiento adecuado de la seguridad en el perímetro establece el marco de sistema en el que nuestra aplicación deberá de ejecutarse finalmente.

De este análisis podría derivarse qué sugerencias de seguridad deberían de ser propuestas al cliente antes de

concluir la fase de Requisitos o Previsión, ya que todo el funcionamiento de la aplicación dependerá de esa estructura.

- Todos estos aspectos, pueden ser adecuadamente recogidos en los documentos llamados entregables, que dan forma impresa a las ideas que organizan esta fase.

Aunque los planes estén sujetos a cambios debido al carácter dinámico del marco de trabajo, una exposición de todos estos factores de forma inicial, junto a un compromiso de cumplimiento por las partes, evitarán, al menos, sorpresas de última hora y la aparición del fantasma de *Murphy* (ya sabe el lector, el temido “cuando todo va bien, es que Vd. ha olvidado algo...”).

El equipo de desarrollo, debe de considerar estos requisitos como parte de la fase inicial, y aunque muchos otros serán identificados en el Modelo de Amenazas, las solicitudes de los usuarios finales probablemente añadirán características no previstas en esta etapa. Y eso, sin contar con las restricciones impuestas por el cumplimiento de estándares empresariales o de la industria, u otros vinculados a obligaciones paralelas, como la convivencia con software existente (sea este seguro o no), o los procesos de obtención de algunas certificaciones, como las vinculadas a las normativas de calidad.

Seguridad en la fase de Previsión (o Análisis)

En esta fase, se analizan en profundidad los requisitos y estructura del software. Desde el punto de vista de la seguridad los puntos clave a considerar son los siguientes:

Definición de la arquitectura de seguridad y patrones de diseño.

Resulta crucial identificar aquellos factores críticos en la seguridad desde el punto de vista arquitectónico. Esto supone la identificación de la seguridad dentro de las técnicas de diseño, tales como:

- La división en capas de la aplicación (*layering*): Las técnicas de división en capas establecen los criterios de organización del software en un conjunto de componentes bien definidos y no interdependientes. El diseño seguro tiene en cuenta a todo el conjunto como una unidad en riesgo.

<u>Consejo</u>: Un típico problema de seguridad que plantea la división en capas, se deriva de la propia arquitectura y la forma en que
--

las capas se comunican entre sí. No olvide incluir en estas fases el principio de ejecución con los privilegios mínimos.

- La utilización de lenguajes fuertemente tipados
- La aplicación del principio del privilegio mínimo (que establece que las aplicaciones ideales desde el punto de vista de la seguridad podrían permitir su explotación haciendo uso del mínimo conjunto de permisos posible. Más adelante explicaremos con más detalle cómo aplicarlo)
- Minimización de la superficie de ataque (entendemos por tal, el conjunto de elementos de que consta la aplicación susceptible de contener algún tipo de vulnerabilidad).
- Documentar claramente qué elementos del software constituyen la superficie de ataque antes citada.
- Dado que es imposible obtener una seguridad absoluta, sí constituye una buena práctica exponer a los usuarios de forma predeterminada sólo aquellos elementos que van a ser usados por la gran mayoría, dejando el resto de manera opcional y configurable. Todas estas características, deben ser expuestas haciendo el uso del principio del mínimo privilegio.
- Si bien es cierto que algunas de estas características van a extender la superficie de ataque, resultarán necesarias debido a cuestiones de funcionalidad y requisitos. No obstante, todas ellas debieran de ser analizadas adecuadamente, evitando exposiciones innecesarias.

El modelo de Riesgos bajo el prisma de la seguridad

Consideraciones sobre el Modelo de Amenazas

Al hablar de seguridad en la fase de desarrollo, una parte crucial es la confección del Modelo de Amenazas. Es aquí donde el equipo de trabajo debe de realizar sus esfuerzos más decididos para conseguir una estructura segura basándose en los posibles tipos de ataque que su aplicación, debido a su estructura, puede sufrir.

Vamos a intentar resumir de forma puntual, -como hasta ahora-, los puntos principales del modelo, de forma que puedan servir de pauta para posteriores análisis, algo así como una lista de deberes de seguridad que debieran de revisarse al abordar esta fase.

- El equipo de desarrollo deberá de dirigir la creación del modelo según un análisis elemento a elemento.
- Utilizando una metodología estructurada, se identificarán los elementos funcionales y sus interfaces de usuario (posibles puntos de entrada).
- En la confección del modelo, se establecerán valores ponderados de riesgo para cada amenaza, y se seguirán los principios conocidos del modelo de riesgos: publicación de la amenaza ponderada, y primer esbozo de su tratamiento en mitigación (preventivo) y en contingencia (reactivo)
- Las contramedidas pueden adoptar un sinnúmero de formas (hablaremos de ellas en siguientes capítulos), y cada una será específica de la amenaza detectada, si bien, existen un paquete de medidas que podemos agrupar por conceptos que suelen aplicarse de forma genérica.
- Otro tipo de contramedidas, consiste, simplemente en la realización de revisiones detalladas del código fuente en busca de posibles vulnerabilidades, o áreas que requieran un tratamiento especial.
- Todo el modelo debería de almacenarse en forma fácilmente modificable mediante técnicas informáticas (esto parece obvio, pero podría no serlo tanto en ciertas sociedades o entornos en los que las fases de Análisis y Diseño siempre han ido vinculadas a la elaboración de documentación impresa, casi de forma exclusiva).

A este respecto la propuesta de Microsoft ha ido evolucionando con el tiempo hasta llegar a la completa solución actual que la compañía propone para la gestión del ciclo de vida de las aplicaciones: **Visual Studio Team System**. Es una herramienta separada de Visual Studio –aunque algunas de sus características, como el Diseñador de Clases, han sido incorporadas en las versiones profesionales, y ofrece un control total del ciclo.

En caso de no disponer de esa herramienta, la oferta de la compañía incluye **Microsoft Project** y **Microsoft Visio for Enterprise Architects**, que permiten un control adecuado de estos aspectos, y que –en el caso de Visio- incluyen un soporte completo del estándar **UML** (Lenguaje Unificado de Modelado). Otras herramientas populares de soporte del ciclo vital son **Rational Rose** (de IBM) o **ERwin**.

- Los encargados del proyecto de administración de riesgos deben comunicar claramente al resto del equipo que dichos miembros del personal están autorizados para implementar los controles de seguridad. Si no se realiza esta declaración explícita, algunos empleados pueden poner objeciones o incluso resistirse a los esfuerzos de implementar los nuevos controles.
- Al personal encargado de la implementación de los controles de seguridad se le debe permitir cambiar las prioridades de sus responsabilidades existentes, dependiendo de la aparición de riesgos de alto nivel durante cualquiera de las fases. Este trabajo tiene una alta prioridad: si no se presupuestan los recursos y el tiempo adecuados, es posible que los controles no se implementen de forma eficaz. Asimismo, una asignación no adecuada de los recursos puede derivar en problemas injustamente atribuidos a la tecnología o al control.
- Al personal encargado de implementar los controles se le debe ofrecer apoyo financiero, formación, equipos y otros recursos necesarios para implementar cada control de forma eficaz.

Definir criterios de distribución adicionales, en función de la seguridad.

- Aunque exista un paquete de criterios corporativos de seguridad, algunas aplicaciones puntuales requerirán de un tratamiento especial.
- En general deberíamos asumir que –a pesar de la creación de políticas de implementación segura- cada aplicación individual va a tener sus peculiaridades, y que esta política deberá adaptarse a las necesidades puntuales que cada solución requiera.

El control de la seguridad en las capas lógicas de la aplicación

En su “Guía de la implementación de riesgos de seguridad”⁴, Microsoft recuerda las consideraciones de riesgo pasan por un conjunto de capas (Física, de Red, *Host*, Aplicación y Datos) cuyo tiene mucho que ver con la tecnología implantada y el grado de control deseado (o requerido) por la aplicación, lo que queda esquematizado en el gráfico adjunto:

⁴ <http://www.microsoft.com/latam/technet/articulos/adminriesgos/srsgch06.msp>

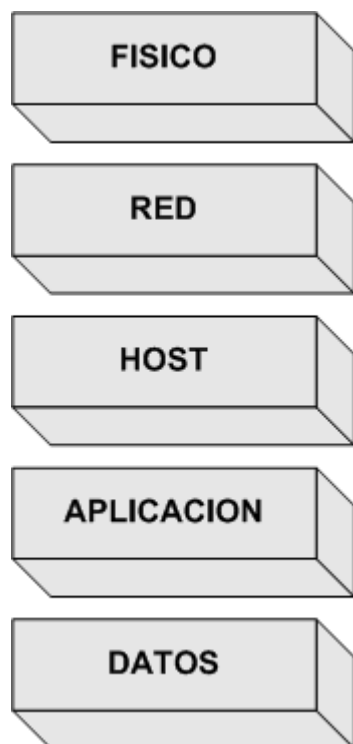


Fig.. Pila de implementaciones de seguridad recomendada por MSF

Seguridad en el nivel físico

Respecto al nivel Físico, estaríamos hablando de seguridad embebida en el propio hardware. Microsoft no se manifiesta a este respecto, pero sí es cierto que existen varias iniciativas importantes a este respecto.

Por un lado, la compañía ha llegado a acuerdos con Intel y AMD, y el nuevo subsistema de seguridad de Windows Vista (de nombre en código previo *Palladium* y posteriormente *Next Generation Secure Computing Base (NGSCB)*), se apoya en funcionalidades ofrecidas por el propio hardware para establecer condiciones y restricciones únicas.

No obstante, para muchas de estas características, se requiere la presencia de un chip especial, *Intel Trusted Platform Module (TPM)*, que interactuará con el sistema para ofrecer al usuario –de forma opcional– esas características adicionales de seguridad. Insistimos en el término **opcional**: la mayor parte de las máquinas no implementan todavía ese chip, y transcurrirán años hasta que su inclusión sea moneda corriente en el hardware habitual. De forma que se tratará –todavía– de seguridad en el propio software.

A título de curiosidad, las propuestas de este subsistema desde el punto de vista operativo son las siguientes:

NGSCB persigue la consecución de los siguientes objetivos establecidos por la compañía de Redmond:

- Informará con quién estas tratando "online", y qué está haciendo. Identificará tu PC como único y podrá limitar lo que llega y ejecuta cada PC. La información que viene de Internet será verificada antes de que se pueda acceder a ella.
- Protege la información por medio de encriptación para la protección de datos. el sistema podrá mantener la integridad de los documentos para que no puedan ser alterados sin el consentimiento del usuario.
- Controla la aparición de virus y gusanos. El sistema no podrá ejecutar programas no autorizados, evitando que los virus destruyan el sistema.
- Aumenta la privacidad. Además de proteger los datos del PC, controla los que se envían a través de Internet, usando agentes de software que asegura que los datos llegan sólo a la gente adecuada.
- Ejerce un control detallado sobre la información que sale del PC. Usando la tecnología DRM (Administración de Derechos Digitales) *NGSCB* evitará la distribución de música, películas, y otro tipo de propiedades intelectuales a través de Internet. Las productoras de películas y la industria de la música podrán usar esta tecnología para permitir a sus clientes usar los derechos contra el copiado de CDs o películas, por ejemplo. Se asegura que *NGSCB* podrá proteger el correo electrónico de tal forma que no podrá ser reenviado o copiado a otra gente. Se podrán crear documentos de texto sólo accesibles en fechas determinadas.
- *Inicio Seguro*: Seguridad en los ataques por "fuerza bruta": evita que se acceda al equipo mediante la carga del sistema a través de otros dispositivos (CD's, USB's, etc.) Está vinculado a la presencia del chip TPM.
- *Bit Locker* (antes llamado *Full Volume Encryption*): Protege el sistema impidiendo que se pueda acceder al contenido del disco duro removiéndolo y accediendo desde otro equipo. Igualmente vinculado al chip TPM.

- Mejoras del control parental de contenidos: filtros, niveles y horarios de restricción, control del uso de juegos basándose en las calificaciones de ESRB⁵.
- Mejoras en el cortafuegos de Windows. A primera vista, no parece haber muchos cambios, pero internamente ha sufrido varias modificaciones: soporte de filtro bidireccional, posibilidad de configuración corporativa de IPSec.
- Nuevo sistema de Windows Update: ahora accesible desde el Panel de control (ver imagen adjunta), combina las características de actualizaciones automáticas con Windows Update en una sola aplicación. (Ver gráfico adjunto)

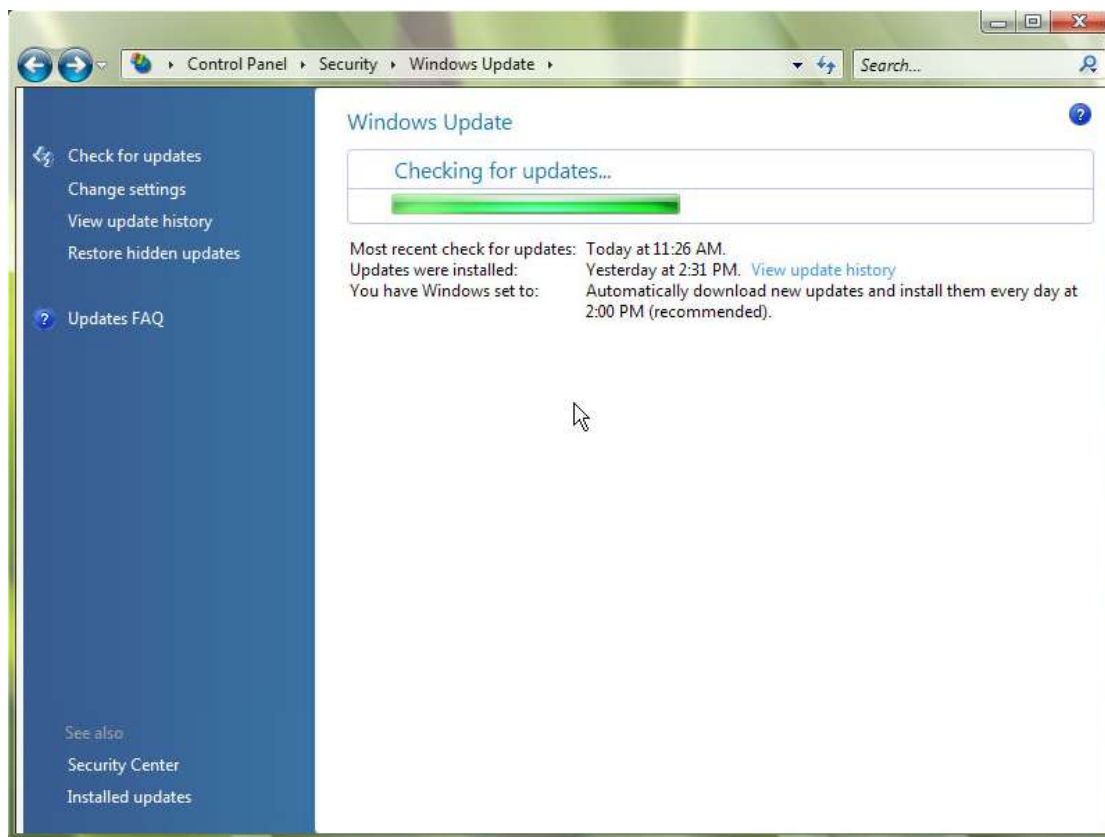


Fig. xx: Ventana de Windows Vista mostrando el control de actualizaciones (Windows Vista CTP, de Febrero/2006).

Hay muchos rumores que corren por la red que están siendo desmentidos, como por ejemplo que Microsoft introducirá cambios propietarios en el protocolo TCP/IP o que NGSCB no funcionará con Linux. Todos inciertos. El mensaje oficial es que los usuarios han de

⁵ Para más datos sobre este mecanismo de control, ver la página en castellano:
<http://www.gdgames.com.mx/ratings.html>

estar tranquilos, ya que NGSCB hace todas estas protecciones del PC de forma clara y transparente.

Además de estas iniciativas basadas en chips, cabe mencionar de pasada algunas otras que tienen que ver con la implantación de sistemas de comunicación y/o almacenamiento seguro basándose en aspectos relacionados con la transferencia de información basada en los principios de la Mecánica Cuántica, ya probados en redes experimentales a partir de 2004 en el MIT (Instituto Tecnológico de Massachussets), los descubrimientos sobre nanocircuitos publicados por Hewlett-Packard, las investigaciones de la U. de Stanford e IBM sobre "spintrónica", y los nuevos chips de memoria holográfica propuestos por la compañía japonesa NTT Docomo, son algunos ejemplos de estas tecnologías emergentes. Pero, por el momento, centrémonos en lo existente hoy en día.

Seguridad en la Red

Pero, sigamos con las recomendaciones "oficiales" a propósito de las capas de aplicación. Desde el sitio antes citado, se nos recuerda, a este respecto, que "Una arquitectura de red bien diseñada e implementada correctamente proporciona servicios de alta disponibilidad, seguros, escalables, fáciles de administrar y confiables. Puede disponer de varias redes en su organización y debe evaluar cada una individualmente para asegurarse de que tienen una protección correcta o de que las redes de alto valor están protegidas de las redes inseguras. La implementación de defensas de red internas incluye la consideración del diseño de red adecuado, la seguridad de red inalámbrica y, posiblemente, el uso de Seguridad del protocolo de Internet (*IPSec*) para garantizar que sólo los equipos de confianza tengan acceso a los recursos de red críticos."

Seguridad en el Host

Los hosts son de dos tipos: clientes y servidores. La protección de ambos requiere conseguir un equilibrio entre el grado de protección y el nivel de capacidad de uso. Aunque existen excepciones, la seguridad de un equipo normalmente aumenta a medida que se reduce su capacidad de uso. Las defensas de *host* pueden incluir deshabilitar servicios, quitar derechos de usuario específicos, mantener actualizado el sistema operativo, así como utilizar antivirus y productos de servidor de seguridad distribuidos.

Seguridad en la Aplicación

Es en lo que nos vamos a centrar principalmente en el resto del libro. Al existir en el contexto global del sistema, deben tener en cuenta la seguridad de todo el entorno al evaluar su propia seguridad. Se debe probar exhaustivamente el cumplimiento de seguridad de cada aplicación antes de ejecutarla en un entorno de producción. La implementación de las defensas de aplicación incluye toda la normativa antes expuesta.

Seguridad en los datos

Los datos constituyen el recurso más valioso de la mayoría de las organizaciones. En el nivel de cliente, los datos normalmente se almacenan localmente y pueden ser muy vulnerables a los ataques. Los datos se pueden proteger de diferentes maneras, incluido el uso del servicio de archivos de cifrado (EFS) y las copias de seguridad frecuentes y seguras.

Seguridad en la Fase de Implementación

En esta fase se codifica, testea e integra el software con los elementos y sistemas necesarios para su funcionamiento. Todos los pasos que se den en esta fase tendentes a eliminar potenciales agujeros de seguridad, resultan críticos y reducen significativamente la probabilidad de que tales riesgos lleguen más adelante en la fase de desarrollo (recuérdese el principio de que cuanto más temprana sea la detección del problema potencial, más segura será la solución).

Los resultados del modelo de amenazas se convierten en esta fase en una guía imprescindible para evitar la progresión hacia delante del problema potencial. Se pondrá especial énfasis en las entradas de datos y –en general– en cualquier superficie de ataque. Los siguientes elementos son particularmente aplicables en esta fase:

- Aplicar los estándares de codificación y pruebas. Dichos estándares son de gran ayuda para evitar la producción de agujeros no previstos en las fases anteriores. Por ejemplo, la utilización de sistemas de manipulación de cadenas seguros y más consistentes, y la manipulación adecuada de los búferes de entrada, pueden evitar la introducción de vulnerabilidades del tipo “*buffer overrun*”.

- Los estándares de pruebas y testeo se centrarán en el análisis de problemas de seguridad, y no solamente en el correcto funcionamiento del programa.
- Hacer uso de herramientas especiales, tales como las que testean API's, cajas de diálogo y similares (*fuzzy tools*), o herramientas de análisis estático para supervisión de código, capaces de detectar fallos del tipo *buffer overrun*, *integer overrun* o variables no inicializadas. Entre ellas, probablemente **FXCop** sea la que ha obtenido una mayor aceptación, si bien hay un buen número, y buena parte de ellas de carácter gratuito.
- Hacerse cargo de las revisiones de código. Esas revisiones las realizan los encargados de seguridad conjuntamente con el equipo de desarrollo, y es una fase posterior a la de la utilización de las herramientas mencionadas.

Seguridad en la Fase de Pruebas

Se entiende que el programa llega a esta fase de desarrollo, cuando toda la parte funcional está cubierta, aunque se conocen un número de bugs que están siendo tratados –o van a serlo– en esta fase final, aparte de los que aparezcan como consecuencia de las modificaciones planteadas por la operativa de la fase en sí. También se incluyen en esta fase revisiones de código del estilo de las anteriores, pero focalizando más específicamente en problemas concretos, y en posibles “bugs” que aparezcan al arreglar otros de tipo funcional.

Microsoft suele realizar en esta fase una acción denominada “*Security Push*”, que, aparte de lo anterior, viene a ser una revisión general de seguridad ante el equipo de desarrollo, donde se incluyen partes preprogramadas y otras del tipo “tormenta de ideas”, a medida que el proceso va avanzando.

Aparte de resultar un momento apropiado para realizar estas pruebas, la revisión de código y las pruebas se hacen ya sobre el conjunto funcional que verá la luz en la fase de implementación, proporcionando una oportunidad de revisar, tanto el código de las fases anteriores, como el código “heredado” de otras aplicaciones (*legacy code*).

Consejo: Debido a su naturaleza anterior en el tiempo, este código estará –normalmente– más propenso a la explotación de vulnerabilidades. Tenga cuidado, y si es preciso, revise las partes críticas que pudieran afectar al desarrollo de su aplicación.

Seguridad en la Fase de Distribución

Durante esta fase, el software debe de estar sujeto a lo que se denomina una Revisión Final de Seguridad (*Final Security Review* ó *FSR*), cuyo objetivo principal es responder a la pregunta de si el software está listo realmente para ser instalado en las máquinas cliente. El software debe de encontrarse en un estado estable, sólo a falta de mínimas modificaciones antes de su salida final.

Esta prueba debe de ser llevada a cabo por el equipo de seguridad de la empresa, y no por el equipo de desarrollo. Estos, informarán a Desarrollo en qué forma y con qué condiciones van a conducir la Revisión Final y solicitarán todos los recursos necesarios para su compleción.

Cualquier Revisión Final requiere una revisión de los *bugs* inicialmente identificados como *bugs* de seguridad, pero que un análisis posterior postergó como no relevantes desde el punto de vista de la seguridad, si queremos garantizar que el análisis se realizó correctamente. Otra de las pruebas incluidas en el FSR es la revisión de las capacidades de resistencia ante nuevas vulnerabilidades aparecidas desde el momento del análisis de la aplicación.

No se trata de encontrar todas las vulnerabilidades que todavía puedan existir en el software, sino de obtener un dibujo global de la seguridad y las probabilidades de que la aplicación resista ante ataques de seguridad una vez instalado en el cliente final.

Aviso: En caso de encontrarse nuevas o antiguas vulnerabilidades, el paso final no es solamente su arreglo, sino una revisión de las partes del análisis y diseño de la aplicación respecto a ellas, al objeto de detectar si esos fallos pueden conducir a nuevos agujeros aún no descubiertos.

Además, durante la fase de mantenimiento (el resto del ciclo de vida de la aplicación, hasta su retirada definitiva), con mucha probabilidad aparecerán nuevas vulnerabilidades que necesitarán de un tratamiento, a veces puntual (en función de la gravedad), a veces global (un parche que recoja y actualice un grupo de vulnerabilidades de nivel medio). El objetivo de seguridad, se mantiene, por tanto, activo y las acciones incluirán –en lo posible– reuniones “post-mortem” para aprender de las lecciones asimiladas como consecuencia de su tratamiento y solución.

Todo el conjunto de recomendaciones anterior debe de conducirnos inexorablemente, a la implantación de estrategias de defensa.

Capítulo 3: Estrategias de defensa: SD3 + C

El conjunto de guías y recomendaciones que hemos revisado hasta aquí, se traduce finalmente en estrategias de defensa que cada rol del equipo debe de tener en mente al analizar la seguridad, comenzando por la aplicación de los principios SD3 + C y el análisis de los tipos de amenazas que los expertos han reducido a 6 categorías, distinguidas por las iniciales de cada tipo de ataque en ingles (STRIDE), que comentaremos más adelante en este capítulo.

Seguro por diseño

Como ya anticipamos en el capítulo anterior, mediante este principio, garantizamos que todo el proceso queda cubierto bajo el cumplimiento de las directivas de seguridad. En concreto, las recomendaciones que se suelen hacer, van desde la asignación de un responsable de seguridad en el equipo de desarrollo, hasta un *test* final, en el que se pide a alguien del equipo (aunque mejor sería a algún especialista en seguridad), que intente romper la aplicación.

Otras recomendaciones clásicas de esta fase son: Publicación y Revisión periódica del modelo de amenazas (por ejemplo en una Intranet, que pueda ser actualizada con las medidas que se toman para minimizar los riesgos, o con la aparición de nuevas amenazas), tratar de seguir principios seguros en la escritura de código, corregir los “*bugs*” que se desvían de esos principios lo antes posible, y evitar el fenómeno denominado “*code rot*” (código podrido, o estropeado), mediante la revisión y simplificación sistemática del código fuente, con la seguridad como principio rector.

Seguro por defecto

Se trata de que, ya desde el momento posterior a la instalación de la aplicación, ésta, tenga un comportamiento suficientemente seguro. (Desde luego, será labor de los analistas evaluar qué significa en nuestra aplicación el término *suficientemente*). Algunos principios aplicables a esta directiva, incluyen: no instalar todas las características de la aplicación de forma predeterminada (suministrando una forma cómoda de instalar el resto de opciones),

no requerir que el código sea utilizado por miembros del equipo de administradores del sistema, sino hacer que se ejecute con el menor nivel de privilegios posible, y proteger los recursos críticos adecuadamente.

Seguro en la distribución

Trata este principio de garantizar la facilidad de mantenimiento de una aplicación una vez instalada. Cuanto más compleja sea de administrar y mantener, más difícil resultará hacer frente a nuevas amenazas. Podemos articular este apartado en tres consejos fundamentales: Suministrar una forma adecuada de modificación de las características de seguridad en la aplicación, crear “parches” de seguridad tan pronto como una vulnerabilidad sea detectada, e informar adecuadamente al usuario final sobre las características de seguridad de la aplicación.

Creación del Modelo de amenazas

Todo esto no es trivial, y la forma más recomendable para su abordaje es dividir el problema en fases paralelas al propio ciclo de vida. En el esquema de la figura adjunta, puede verse cómo los problemas se van presentando a lo largo del ciclo de vida de la aplicación, y cuáles serían algunas de las acciones a realizar.

Es significativo que Visual Studio 2005 incorpore estas características en diferente grado de profundidad, dependiendo de que se trate de sus versiones básicas o de una versión que incorpore VSTS (*Visual Studio Team System*), la solución de gestión del ciclo de vida de las aplicaciones.

El modelo de amenazas (Threat Model)

La puesta en práctica de la creación de un modelo de amenazas puede ser un tanto difícil al principio, cuando no se tiene experiencia, pero, aunque el resultado del trabajo no nos ofrezca nunca una garantía segura al 100%, sí que, al menos, conseguirá que nuestra aplicación cumpla con un nivel suficiente de calidad en la seguridad.

Como ya se ha comentado, el modelo de amenazas de seguridad se integra como parte del modelo de riesgos, y consecuentemente, se mantiene vivo y activo durante todo el ciclo. Al objeto de establecer una metodología que permita el análisis más detallado del modelo, se recomiendan las siguientes acciones, como proceso de definición del modelo de amenazas:

- Identificar activos de la aplicación
- Crear información general sobre la arquitectura
- Descomponer la aplicación
- Identificar, documentar y clasificar las amenazas.

¿Qué significan concretamente estos pasos? Veamos someramente cada uno de ellos:

Identificar los activos y crear información general sobre la arquitectura

Supone crear una lista de elementos que requieren protección, como por ejemplo: Datos confidenciales (como BB.DD. de clientes, etc.), páginas Web, y en general, cualquier elemento que –de verse comprometido en su seguridad- pudiese atentar contra el correcto funcionamiento de la aplicación.

Podríamos decir que el objetivo de esta segunda fase, tiene 3 partes principales: identificar lo que hace exactamente la aplicación, crear el diagrama de la arquitectura, e identificar las tecnologías implicadas.

Descomponer la aplicación

Entendemos por tal, el proceso de analizar el funcionamiento de la aplicación en todas sus partes, y a partir de ello, identificar los límites de confianza que rodean a cada uno de los activos de la aplicación: flujos de datos, puntos de entrada, código privilegiado (con más permisos que el resto), y con todo ello, documentar el perfil de seguridad.

Identificar, documentar y clasificar las amenazas

Como primer paso, podríamos definir un grupo de trabajo especializado en estos aspectos, estableciendo unos responsables y portavoces, al igual que se hace en el modelo de riesgos de MSF. Una vez hecho esto, podemos distinguir, a priori, 3 tipos de amenazas: De red, de *Host*, y de la aplicación.

En este punto, es donde interviene **STRIDE**. Se trata de un modelo de identificación de amenazas que ya anticipábamos al comienzo del artículo, y que cataloga las amenazas en categorías generales, dependiendo de la forma que adoptan y los efectos de las mismas. Veamos aunque sea por encima los seis elementos que constituyen el modelo:

El Modelo STRIDE

STRIDE agrupa 6 tipos de amenazas, por sus siglas:

- **Spoofing Identity** (suplantación de personalidad)
- **Tampering** (alteración de la información)
- **Repudiation** (eliminar un archivo esencial y denegar el hecho, o negar la realización de una acción que supone un compromiso, tal como el pago de un bien adquirido)
- **Information disclosure** (publicación o acceso a información no autorizada, incluyendo código fuente, *scripts*, etc.)
- **Denial of service** (Denegación de servicio, como resultado de una acción que compromete el rendimiento del sistema, como las que apuntábamos al inicio de este artículo)
- **Elevation of privilege** (Obtención de permisos de usuario no autorizados, de forma no legítima, como la saturación de búfer (*buffer overrun*), u otras.

A partir de aquí, procederemos a documentar las amenazas identificadas, y a crear una estrategia de mitigación. Podría tomar la forma de una plantilla o sitio Web con páginas que recogieran información similar a esta:

Descripción de la amenaza	Inserción de comandos SQL
Destino de la amenaza	Componente de acceso a datos
Riesgo	(en blanco, hasta su definición)
Técnicas de ataque	El intruso anexa comandos SQL al nombre de usuario, que se utiliza para crear una consulta SQL.
Medidas preventivas	Utilizar una expresión regular para validar el nombre de usuario y utilizar un procedimiento almacenado con parámetros para tener acceso a la base de datos

Por fin, en el sexto paso, clasificaríamos las amenazas para facilitar su mitigación y tratamiento posterior. Para ello, podemos ponderar las amenazas utilizando una fórmula del tipo:

$\text{Riesgo} = \text{Probabilidad} * \text{Daños potenciales}$
--

Al igual que con los riesgos del modelo MSF, esa clasificación sería permanentemente revisada y tenida en cuenta cuando aparezcan

elementos que puedan hacer cambiar la valoración de su nivel de riesgo, y cualquier aparición de una nueva amenaza, supondrá la actualización de la plantilla, y probablemente, la creación de una estrategia de tratamiento.

Uno de los problemas que se presentan con asiduidad a la hora de valorar con la mayor exactitud posible los riesgos y establecer el impacto potencial es el desacuerdo a la hora de ponderar los factores que intervienen.

La directiva DREAD

Con objeto de fijar con más claridad las medidas de un riesgo, se ha presentado con frecuencia un submodelo denominado **DREAD**. Consta de cinco puntos que pueden servir de hilo conductor para establecer cuán peligrosa puede resultar cualquier amenaza. Los cinco puntos recogidos por la directiva son:

- **D**año Potencial (cuántos activos, de qué clase y en qué forma pueden verse afectados por la amenaza).
- Facilidad de **R**eproducción (lo fácil que resulta volver a reproducir la amenaza, con objeto de aislarla y tratarla adecuadamente).
- Capacidad de **E**xplotación (la facilidad o dificultad que implica lanzar un ataque)
- Usuarios **A**fectados (en ocasiones, distinguiendo entre propios y ajenos y los distintos tipos de usuarios, en función de los roles).
- Dificultad para su **D**escubrimiento: lo difícil que nos puede resultar detectar la fuente exacta del problema en cualquiera de las fases del ciclo de vida.

A esta directiva, naturalmente, podríamos añadirle cualquier otra consideración que juzgásemos de interés que pueden ir, desde las meramente técnicas a las puramente comerciales, como la reputación. Lo fundamental es alcanzar un grado de valoración aproximado y tomar unas medidas adecuadas a nuestras circunstancias y posibilidades.

Una vez que hemos obtenido el valor podemos utilizar una tabla como la adjunta, para establecer con claridad la estrategia a seguir respecto a ella.

Descripción	El atacante obtiene credenciales de autenticación monitoreando la red
Objetivo del ataque	Proceso de autenticación de una aplicación Web

Baremo	Alto
Técnicas de ataque	Utilización de software de monitorización de red
Contramedidas	Usar SSL para suministrar un canal cifrado

Breve explicación de las amenazas del modelo STRIDE

Pero, antes de seguir, hagamos un repaso a cada uno de los 6 tipos de amenazas que dan origen al modelo STRIDE, por sus siglas, para poder reconocer su posible presencia durante un análisis:

- **Spoofing Identity** (suplantación de personalidad) Permite a los usuarios malintencionados (o simplemente curiosos), “engañar” al sistema, haciéndole creer que está tratando con alguien con más permisos de los reales, y otorgándole control sobre el sistema en cuestión. Es uno de los más peligrosos por las implicaciones potenciales. La forma de producirlos depende del sistema, por supuesto.
- **Tampering** (alteración de la información) Capacita al atacante para modificar la información que se transmite entre dos puntos, pudiendo, en algunos casos desembocar en una situación de *Spoofing*, o simplemente de alteración de información confidencial. No hay que confundirlo con **Información Disclosure** (ver más abajo)
- **Repudiation** (eliminar un archivo esencial y denegar el hecho, o negar la realización de una acción que supone un compromiso, tal como el pago de un bien adquirido) Un caso típico es la compra por Internet, mediante sistemas de pago digital y la denegación posterior (o previa) del pago mediante notificaciones a la entidad de encargada de satisfacer la deuda. Con los sistemas actuales, es la que está teniendo una solución más clara en una buena parte de los casos. Recordemos que las entidades de certificación como **VeriSign** o la **Fábrica Nacional de Moneda y Timbre**, pueden emitir certificados digitales que tienen el mismo valor legal que los físicos.
- **Information disclosure** (publicación o acceso a información no autorizada, incluyendo código fuente, *scripts*, etc.) Los modos de llegar aquí pueden ser variados e incluir alguna de las técnicas anteriores. En general se trata de que cualquier información sobrepase los límites de publicación (audiencia potencial), para los que ha sido creada. Pueden utilizar

técnicas de Ingeniería Social, que comentaremos brevemente más adelante.

- **Denial of service** (Denegación de servicio, como resultado de una acción que compromete el rendimiento del sistema, como las que apuntábamos en el capítulo de Introducción) Normalmente, se trata de solicitar al sistema un mayor conjunto de servicios de los que está preparado para ofrecer. La mayor parte de las veces, el atacante no obtiene beneficio alguno. Sólo perjudica al sistema.
- **Elevation of privilege** (Obtención de permisos de usuario no autorizados, de forma no legítima, como la saturación de búfer (*buffer overrun*), u otras. Es similar a la primera, y, en ocasiones, el camino natural para llegar a ella. Casi indefectiblemente, se consigue mediante la introducción de información adicional a las peticiones de identificación y/o proceso dentro del sistema.

Aunque los tipos de amenazas sean solamente 6, la forma de producir cada una de ellas es de lo más variada. Y, por si eso fuera poco, constantemente aparecen nuevas formas de explotación de esas vulnerabilidades de los que no está a salvo ningún sistema (todo depende del sistema y de la aplicación). Pero algunos de ellos, han tomado carta de naturaleza, debido a su eficacia y el número de veces que se repiten con diferentes versiones y modos de actuación.

No es nuestra pretensión analizar aquí las mil y una formas en que el sistema puede ser explotado en busca de vulnerabilidades. Pero sí que podemos hacer recomendaciones para que el código fuente que construimos sea más seguro y -en un gran porcentaje-, eso tiene que ver con las entradas de datos.

Todos los libros de seguridad en el software abundan en un principio básico: *toda entrada de datos es potencialmente peligrosa*. Eso nos conduce hacia la validación de las entradas. De hecho, las flaquezas definidas por STRIDE necesitan para su explotación que las entradas de usuario admitan alguna forma extendida que permita la introducción de órdenes adicionales (o que no debieran de ser conocidas a priori). Otro problema típico aparece con los resultados devueltos por las peticiones o entradas de los usuarios, cuando el sistema devuelve información añadida que puede ser utilizada para continuar un ataque (información de nombres de tablas o campos de bases de datos, nombres y tipos de servidores, ubicaciones, etc.). Dentro de los muchos principios que se recomiendan seguir en la implantación segura, de cara a la defensa

e implantación de la estrategia, Microsoft publicó⁶ hace algún tiempo las siguientes recomendaciones:

Adoptar el principio de privilegios mínimos

Los procesos que ejecutan código o secuencias de comandos deberán ejecutarse con una cuenta con privilegios mínimos para limitar los posibles daños que podrían producirse si se ve comprometido el proceso. Si un usuario malintencionado consigue insertar código en un proceso de servidor, los privilegios concedidos al proceso determinan en gran medida los tipos de operaciones que puede realizar el usuario. El código que requiere confianza adicional (y privilegios elevados) deberá aislarse en procesos independientes.

El equipo de ASP.NET decidió deliberadamente ejecutar la cuenta de ASP.NET con los mínimos privilegios (mediante la cuenta ASPNET). Durante el lanzamiento de la versión beta de .NET Framework, ASP.NET se ejecutó como SISTEMA, una configuración intrínsecamente menos segura.

Usar las defensas a fondo

Colocar puntos de control en cada una de las capas y subsistemas de la aplicación. Los puntos de control son los guardianes que garantizan que sólo los usuarios autenticados y autorizados puedan obtener acceso a la siguiente capa inferior.

Consejo: La ubicación de puntos de control puede servir de apoyo a la comprobación de los hitos de seguridad sugeridos más arriba.

No confiar en los datos introducidos por el usuario

Las aplicaciones deberán validar al completo todos los datos introducidos por el usuario antes de ejecutar operaciones con esos datos. La validación podría incluir el filtrado de caracteres especiales. Esta medida preventiva protege a la aplicación de usos incorrectos accidentales o ataques deliberados por parte de usuarios que intentan insertar comandos dañinos en el sistema. Entre los ejemplos más habituales, figuran los ataques de

⁶ http://www.microsoft.com/spanish/msdn/arquitectura/BuildSecNetApps/html/SecurityGuide_Chapter01.asp

inyección SQL, la inserción de secuencias de comandos y el desbordamiento del búfer.

Consecuencia: Además del principio de no permitir que una entrada de usuario que represente a un campo de una tabla, tenga como longitud de entrada máxima un valor mayor que el de la longitud del campo, la utilización de algoritmos de validación sería más que recomendable en la mayor parte de los casos.

Utilizar opciones predeterminadas seguras

Una práctica habitual entre los desarrolladores es la de utilizar opciones de configuración de seguridad reducida, simplemente para que funcione la aplicación. Si la aplicación requiere características que le obligan a reducir o cambiar la configuración de seguridad predeterminada, Pruébalo primero para comprobar sus efectos y consecuencias antes de realizar el cambio.

La ejecución con privilegios mínimos puede ser de gran ayuda, a este respecto.

No depender de la seguridad por medio de la oscuridad.

Si intenta ocultar secretos mediante el uso de nombres de variables engañosos o los guarda en ubicaciones de archivos no habituales, no estará mejorando la seguridad. En el juego del escondite, es mejor utilizar características de la plataforma o técnicas de probada eficacia para proteger los datos. (Lo que para nosotros es oscuro, a un hacker puede parecerle diáfano).

Validar todo acceso al sistema.

No siempre es necesario transmitir el contexto de seguridad del usuario hasta el servidor para realizar comprobaciones de autorización. A menudo, ésta no suele ser la mejor opción en un sistema distribuido. El control de clientes en la puerta se refiere a autorizar al usuario en el primer punto de autenticación (por ejemplo, en la aplicación Web del servidor Web) y a determinar los recursos y operaciones (que pueden proporcionar los servicios de capas inferiores) a los que se permite obtener acceso al usuario.

Si diseña estrategias de autenticación y autorización sólidas de tipo perimetral, podrá evitar la necesidad de delegar el contexto de seguridad del llamador original hasta el nivel de datos de la aplicación. Más adelante, hablaremos sobre la autenticación y autorización en la fase de desarrollo.

Asumir que los sistemas externos no son seguros

Si no es de su propiedad, no presuponga que alguien se esté ocupando de la seguridad por usted. (Más bien, presuponga lo contrario).

Reducir el área de exposición

Evite exponer información no necesaria. Si lo hace, posiblemente estará abriendo puertas que pueden traer consigo vulnerabilidades adicionales. También deberá tener cuidado con los errores; no exponga más información de la necesaria al devolver un mensaje de error al usuario final.

Nota: Un ejemplo claro de esta práctica recientemente publicado por Microsoft, es la herramienta de SQL Server 2005 denominada "Configuración de la superficie de exposición", donde se permite la habilitación (por defecto el conjunto de valores está deshabilitado), de varias características de ejecución que podrían dar cabida a puntos vulnerables
--

Cometer errores de forma segura

Si se produce un error en la aplicación, asegúrese de no dejar al descubierto ninguna información confidencial. Tampoco deberá proporcionar demasiados detalles en los mensajes de error; es decir, no deberá incluir información que pueda ayudar a un intruso a explotar una vulnerabilidad de la aplicación. Escriba información detallada de errores en el registro de sucesos de Windows.

No olvidar que el alcance de la seguridad lo define su punto más débil

La seguridad es un motivo de preocupación en todos los niveles de la aplicación. De poco sirve una infraestructura supersólida, cuando nos olvidamos de un punto de entrada crítico.

Si no lo utiliza, deshabilítelo

Puede eliminar posibles puntos de ataque si deshabilita módulos y componentes innecesarios para la aplicación. Por ejemplo, si la aplicación no utiliza el almacenamiento en caché de resultados, debería deshabilitar el módulo de caché de resultados de ASP.NET. Si más adelante se detecta una vulnerabilidad de seguridad en el módulo, su aplicación no se verá amenazada.

Un ejemplo de escenarios típicos de ataque: la inyección SQL

Casi todos los especialistas de seguridad están de acuerdo en que el número uno de las amenazas, es el denominado "*Buffer Overrun*". Se basa en un principio simple: si una entrada de datos no está debidamente acotada (o sea, si permitimos la inclusión de datos más allá de la longitud esperada por la aplicación o la Base de Datos que usamos), los resultados pueden ser impredecibles, como consecuencia de los datos adicionales que el usuario pueda añadir en la entrada.

Aunque los ataques originales basados en ésta técnica consistían simplemente en escribir en el búfer temporal de memoria más información de la esperada, de forma que ocupasen posiciones asignadas a otros valores (como por ejemplo, contraseñas), en la actualidad, se entiende que cualquier dato añadido al esperado en una entrada de datos de usuario que tenga por objeto un ataque al sistema, está utilizándolo.

Basado en el principio de la entrada no acotada de usuario (aunque sin llegar a ser un desbordamiento de búfer), la inyección SQL se basa en incluir sentencias SQL donde el programa espera simples literales de cadena. Si no se controla esta circunstancia, el atacante puede incluir a continuación del literal comandos SQL como "DROP TABLE XXX", "CREATE USER YYY", "GRANT USER YYY ON ZZZ", y así sucesivamente.

En el peor de los casos puede llegar a lanzar la ejecución de comandos del sistema mediante el procedimiento almacenado xp_cmdShell, y llegar a hacerse con el control de la máquina. Para más datos sobre la inyección SQL, véase el capítulo dedicado a la seguridad en la fase de desarrollo.

El lector tendrá que extender este principio a sus propios sistemas, a la luz de las recomendaciones indicadas más arriba, si desea diseñar e implementar un sistema seguro. E incluso tendrá que prever aspectos que quizá no tienen que ver directamente con el desarrollo, sino con la forma en la que nos relacionamos las personas. Algunos de los "hackers" más conocidos son auténticos maestros en lo que se ha venido en llamar "Ingeniería Social".

Se trata de la utilización de técnicas psicológicas para obtener información (directa o indirectamente) de un usuario y suplantarle para actuar en su cuenta. O completar la información necesaria para un ataque remoto, investigando datos del sitio que se desea atacar para –posteriormente– hacerse pasar por empleado enfermo o cualquier otra personalidad y recabar información acerca de los datos que le faltan al atacante.

En la actualidad, probablemente la mayoría de lectores habrá recibido solicitudes de confirmación de información privada por parte de supuestos correos electrónicos "auténticos" provenientes de e-Bay o de alguna entidad bancaria, que –aduciendo, además su deseo de prevenir nuestra propia seguridad– nos solicitan confirmación de usuario y contraseña para corroborarla con sus sistemas. A ese tipo de ataque, conocido como "*phishing*", la mayoría de la gente avisada no responderá, pero –al tratarse de ataques masivos– basta con que un ínfimo porcentaje de correos obtenga una respuesta positiva, para que el ataque resulte un éxito. El atacante, además, reproducirá con fidelidad los *logos* y estilo utilizados por las comunicaciones reales de las entidades para parecer más auténticos, y adornará sus solicitudes con varias diatribas en contra de los "hackers" y a favor de la seguridad informática.

Implantación de las estrategias de defensa

Resulta una buena práctica mantener bajo un prisma reducido el número de elementos a tener en cuenta. Para ello, se establecen técnicas vinculadas al conjunto de prácticas propuesto. Citemos las más importantes: **Autenticación, Autorización, Flujo de Identidad y Auditoria.**

Comencemos por las dos fundamentales en cualquier sistema, basadas en la siguiente premisa “si no puedes entrar, no puedes hacer nada”. E, incluso, una vez dentro de alguna forma, ¿qué acciones están permitidas bajo el mecanismo de acceso que un usuario dado tiene? Esa segunda parte, crítica, es, por cierto, una de las más mejoradas en la versión de ASP.NET 2.0.

Autenticación⁷

También conocida como “control en la puerta”. Sus efectos son varios, pero el más evidente es evitar la primera de las amenazas del modelo: suplantación de personalidad. Cuando el usuario introduce sus credenciales a la entrada de un sistema o aplicación, Windows puede utilizar **NTLM** o **Kerberos** para autenticar al usuario y garantizar que es quien dice ser. De ahí la importancia de una política de claves coherente.

Podemos diferenciar dos clases de autenticación: basada en Windows o no. La selección de uno de ellos puede depender de muchos factores, como el navegador del usuario, la existencia de grupos y roles en la máquina de operaciones, etc.

⁷ El lector puede descargarse gratuitamente la obra “Designing Application Management Authorization”, con una explicación pormenorizada de este principio de seguridad en la dirección “<http://www.microsoft.com/downloads/details.aspx?FamilyId=40A58453-EC1B-4627-874B-F83437DBE00C&displaylang=en>”. Muchas de las indicaciones que siguen se basan en lo allí expuesto.

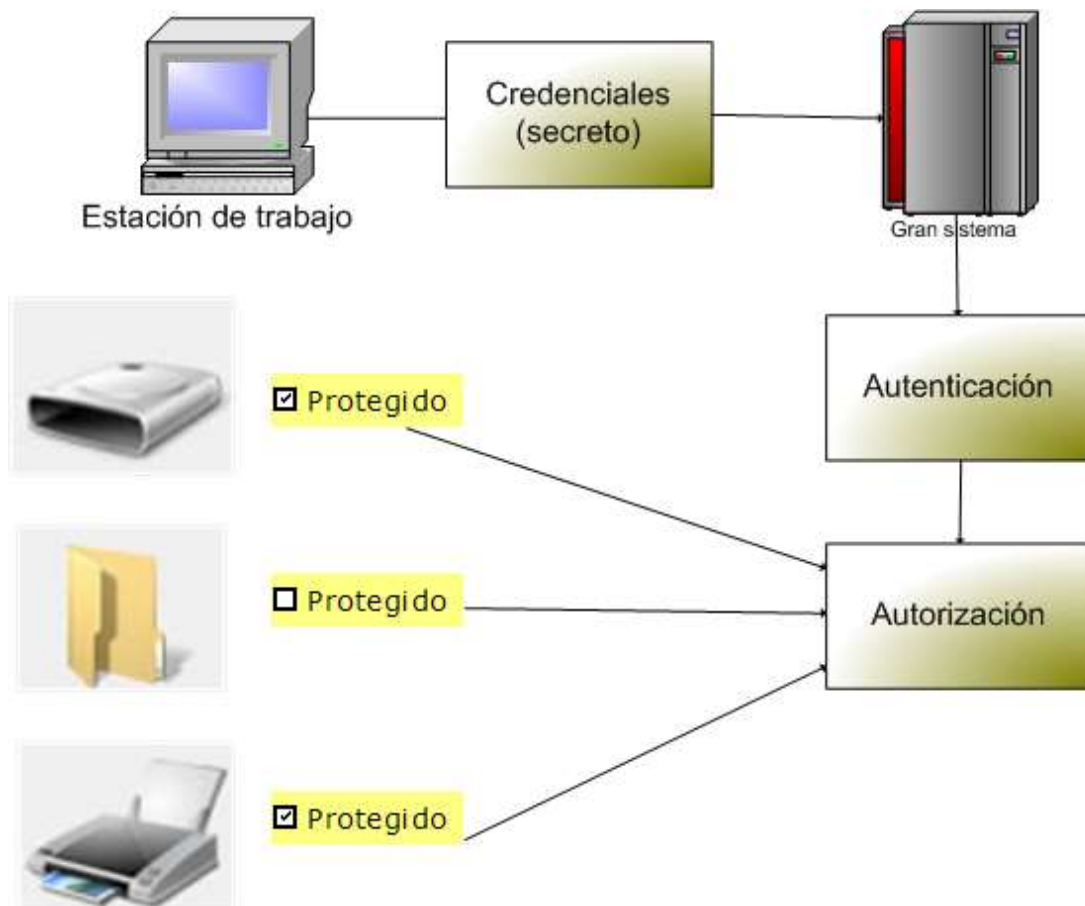


Fig.. Proceso de concesión de permisos sobre recursos del sistema a un usuario, basado en valores credenciales

La autenticación se debe implementar en la capa de la interfaz de usuario para proporcionar funciones de autorización, auditoría y personalización. Esto generalmente requiere que el usuario escriba sus credenciales (como por ejemplo, nombre y contraseña) para demostrar su identidad. Entre otros tipos de credenciales se incluyen las lecturas biométricas, tarjetas inteligentes, claves físicas, certificados digitales, etc.

Si una aplicación se expone como servicio, probablemente se deberán autenticar también en ciertas interfaces de servicio para asegurarse de que se compromete en un intercambio con otra identidad conocida y de confianza, así como que otros servicios externos no simulan su aplicación para hacer creer que quien llama es otra persona.

Autenticación basada en Windows

La pregunta a realizarse por el profano en temas de autenticación sería ¿bajo que identificación se ejecuta un programa que yo no he hecho? (o que sí he hecho, pero en el que no me he preocupado en

absoluto por estas cuestiones). La respuesta es que todo programa Windows se ejecuta con la cuenta de usuario conocida como Identidad Windows (*Windows Identity*). La selección específica de una cuenta para un usuario tiene lugar en el momento del acceso al sistema en dos posibles formas:

- 1) Mediante los datos suministrados por el usuario al conectarse.
- 2) Utilizando una cuenta de servicio especialmente creada para la aplicación, si es un programa el que accede de forma anónima al sistema.

Una vez autenticado, el usuario es "personalizado" (proceso llamado *Impersonation*) hacia una cuenta mantenida por el sistema, que otorga ciertos derechos sobre el mismo.

¿Qué derechos? Ahí entra la segunda parte, la Autorización. La respuesta sería que Windows mantiene una Lista de Control de Accesos (ACL ó Access Control Lists), que especifican en cada caso lo que está permitido o no. Esto es así en Windows en general, sin indicar nada sobre si el tipo de aplicación es .NET o de otra clase.

Autorización

La autorización garantiza que –una vez autenticado– un usuario no accede a información o realiza labores que están más allá de los límites previstos por los diseñadores y administradores de las mismas. Veremos que existen todo un conjunto de técnicas distintas de autorización.

Para determinar la directiva de autorización, deberá tener en cuenta dos factores principales:

- Los permisos y derechos de usuario.
- La seguridad de acceso al código.

Una vez más, si volvemos a las fuentes de recomendación del fabricante, podremos leer a este respecto que "Los permisos y derechos de usuario determinan lo que se permite hacer en una cuenta de usuario en el contexto de la aplicación. Técnicamente, el término "permisos" se refiere a las acciones permitidas en un recurso (como por ejemplo, un archivo o una tabla de base de datos), mientras que los "derechos" hacen referencia a las tareas del sistema que se permite realizar al usuario (como por ejemplo, configurar la hora del sistema o apagar el equipo). Los permisos y derechos de usuario se pueden asignar de forma individual para

cada usuario, si bien resultan más fáciles de administrar cuando los usuarios se organizan de una manera lógica en grupos o roles. La mayor parte de los recursos tienen algún tipo de lista de permisos relacionada, en la que se indican los permisos asignados a los usuarios para ese determinado recurso”.

Mecanismos de Autorización

Básicamente, podemos distinguir 3 formas de autorización:

- Autorización del sistema: De tipo si o no. La decide el administrador y se basa en las listas de control de acceso, ya citadas anteriormente.
- Autorización de seguridad del entorno .NET: Basado en el propio código y el control del CLR sobre la ejecución, de acuerdo con políticas de seguridad establecidas y la evidencia obtenida del propio ensamblado. Un ejemplo podría ser la forma en que el CLR impedirá que cualquier ensamblado obtenido de la zona Internet acceda directamente al Registro del Sistema. Analizaremos esto con más detalle en el próximo capítulo.
- Autorización específica de una aplicación: tiene que ver con los permisos de alto nivel en la lógica de la aplicación, y no sobre recursos concretos del sistema. Se implementa mediante código, y muchas veces afecta a la implementación de capas de negocio (Por ejemplo: si el usuario de la aplicación es empleado de la empresa, debe de tener acceso al estado de las pólizas de seguros)

Flujo de identidad (*Identity Flow*)

Cuando las aplicaciones son cliente/servidor o multicapa, con frecuencia residen en máquinas distintas, y es preciso transferir la identidad del usuario junto con la del proceso que solicita el servicio, para su aprobación por el proceso receptor. El diseño correcto puede evitar alteración de identidad en la transferencia de información entre distintas capas de una aplicación.

Hemos de tener en cuenta que cuanto más lejos del usuario se encuentra una parte de la funcionalidad, menos significativa se vuelve la identidad de éste. En una solución basada en servicios, puede que algunas actividades ni siquiera las inicie un usuario. El objetivo de su diseño debe ser reducir la relevancia del usuario que

hace la llamada, en función de la lejanía de esta llamada a la propia interfaz.

La propia Microsoft recomienda que “El llamador y la aplicación a la que se llama comparten un mecanismo de autenticación que no pertenece a Windows como, por ejemplo, una solución con un inicio de sesión único o un servicio Web centralizado que autentica a los usuarios. En este escenario, establece el flujo de símbolos o *tokens* que proporciona el servicio de autenticación. Debe pasar estos símbolos en mecanismos fuera de banda (no en parámetros de funciones) como, por ejemplo, en encabezados SOAP. El mecanismo de autenticación debe autenticar al usuario cuando se le presenta un símbolo válido; lo que implica que los símbolos que autentica no tienen afinidad con el equipo de origen. Asimismo, se debe asegurar de que los símbolos se pueden autenticar en una ventana de tiempo suficientemente amplia, especialmente en transacciones de ejecución larga”.

El concepto de SID o número de identificación de usuario único

Un SID es el acrónimo de *Security Identifier* (Identificador de seguridad). El SID es la forma que tiene el sistema operativo de hacer un seguimiento de las cuentas. Por ejemplo, se puede renombrar la cuenta Administrador en un ordenador y seguir utilizando esa cuenta para actuar como administrador, puesto que Windows no se preocupa realmente del nombre; Windows todavía es consciente de que esa cuenta es la del administrador porque el SID sigue siendo el mismo, independientemente del nombre de la cuenta.

Hay ocasiones en que resulta útil conocer el SID asociado a una cuenta de usuario. Las clases de seguridad de WMI, por ejemplo, tienen en cuenta los SID; asimismo, el registro de Windows rastrea los perfiles de usuario por el SID, y no por el nombre (basta echar un vistazo a HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList para ver lo que esto significa). De la misma forma, como veremos a continuación, los servidores de datos identifican a cada usuario mediante un SID, siendo el *token*, o cadena de caracteres que nosotros utilizamos una mera descripción de cara a los usuarios.

Existen varias formas de averiguar el SID de un usuario, en función del contexto en el que nos encontremos. Por ejemplo, sin necesidad de utilizar modelos complejos de programación podríamos usar el siguiente código en VBScript, para leer el SID del usuario por defecto en una máquina dada:

```
strEquipo = "."
Set ServicioWMI = GetObject("winmgmts:\\\" & strEquipo & "\root\cimv2")

Set Cuenta = ServicioWMI.Get _
    ("Win32_UserAccount.Name='MARINO',Domain='EQUIPO1'")
Wscript.Echo Cuenta.SID
```

Nota: para una cuenta de usuario de una máquina local, el código funciona igualmente; solo hay que cambiar el nombre del dominio en el parámetro Domain por el nombre de la máquina local.

También es posible ir en la otra dirección; es decir, disponer de un SID y tener la necesidad de conocer la cuenta a la que pertenece ese SID:

```
strEquipo = "."
Set ServicioWMI = GetObject("winmgmts:\\\" & strEquipo & "\root\cimv2")

Set Cuenta = ServicioWMI.Get _
    ("Win32_SID.SID='S-1-5-2-1454475-100348-1600848-5505'")
Wscript.Echo Cuenta.AccountName
Wscript.Echo Cuenta.ReferencedDomainName
```

Este concepto puede ser de utilidad al programador que desee investigar algo más a fondo en la forma en que los distintos sistemas montan y controlan entornos de ejecución, y de hecho, aparecerá en los análisis de formas de ataque que –tanto en este libro, como en la red- podamos analizar (muy especialmente, en ataques basados en Script de Directorio Activo, e inyección SQL).

SID's y el contexto de seguridad en SQL Server 2005

Estos conceptos pueden encontrarse en muchas situaciones, y de cara a las pruebas de desarrollo, es digna de mención la inclusión (para el caso de SQL Server 2005) de la sentencia EXECUTE AS que permite llamar a un procedimiento almacenado a través de una identidad distinta a la del propio contexto, permitiendo, mediante la sentencia REVERT retornar al estado original en cualquier momento. Podríamos decir que se trata de un comando similar al, anteriormente existente, SETUSER.

Nota: Comprobar el contexto de ejecución en seguridad dentro de la base de datos o desde fuera de ella (por ejemplo, llamando a

procedimientos almacenados mediante esta sentencia), es una prueba muy útil para los controles de seguridad, tanto en la fase de desarrollo como en las de pruebas e implantación. Podríamos considerarlo un equivalente al comando del sistema RUNAS que ya comentamos en capítulos anteriores.

Veamos un ejemplo, que la propia MSDN sugiere en su página antes citada. Para ver la información sobre símbolos de usuario creados para un usuario de la base de datos **master**, se ejecuta esta instrucción:

```
SELECT principal_id, sid, name, type, usage FROM sys.user_token;
GO
```

Lo que dependiendo de la identidad y los permisos establecidos en SQL Server 2005, producirá una salida del tipo:

271	0x010500000000 00051500000015 25AF4707E9942D 43170A32EB0300 00	EQUIPO1\MARINO	WINDOWS LOGIN	GRANT OR DENY
2	0x02	public	SERVER ROLE	GRANT OR DENY
3	0x03	sysadmin	SERVER ROLE	GRANT OR DENY
257	0x010200000000 00052000000020 020000	BUILTIN\Administradores	WINDOWS GROUP	GRANT OR DENY
0	0x010500000000 00051500000015 25AF4707E9942D 43170A32010200 00	EQUIPO1\Ninguno	WINDOWS GROUP	GRANT OR DENY
0	0x010100000000 000100000000	\Todos	WINDOWS GROUP	GRANT OR DENY
0	0x010500000000 00051500000015 25AF4707E9942D 43170A32ED0300 00	EQUIPO1\Usuarios del depurador	WINDOWS GROUP	GRANT OR DENY
257	0x010200000000 00052000000020 020000	BUILTIN\Administradores	WINDOWS GROUP	GRANT OR DENY
0	0x010200000000 00052000000021 020000	BUILTIN\Usuarios	WINDOWS GROUP	GRANT OR DENY
0	0x010100000000 000504000000	NT AUTHORITY\INTERACTIVE	WINDOWS GROUP	GRANT OR DENY

0	0x010100000000 00050B000000	NT AUTHORITY\Usuarios autenticados	WINDOWS GROUP	GRANT OR DENY
0	0x010300000000 00050500000000 0000004DFE0000	S-1-5-5-0-65101	WINDOWS GROUP	GRANT OR DENY
0	0x010100000000 000200000000	\LOCAL	WINDOWS GROUP	GRANT OR DENY

Note el lector que la segunda columna (SID) es el identificador único de usuario (UUID): un valor creado y manejado de forma interna y automática por el sistema que para llevar un control adecuado de las identidades autorizadas para su manejo. El SID es, pues, una cadena única (por ejemplo, en la salida anterior el SID de la cuenta de Administradores es 0x01020000000000052000000020020000), que se asigna a cada una de las cuentas creadas en un dominio, sistema u ordenador local.

Cabe recordar, además, a este respecto que, un símbolo de inicio de sesión tiene validez en toda la instancia de SQL Server, mientras que un símbolo de usuario sólo es válido para una base de datos específica. Los miembros de la función fija de servidor *sysadmin* siempre utilizan *dbo* como identidad primaria de su símbolo de usuario. El listado anterior muestra todos los símbolos definidos en la máquina de trabajo para la instancia concreta de SQL Server 2005, tanto los de Inicio de Sesión, como los de usuario.

Estrategias de diseño en el Flujo de Identidad

Cuando se trata de transferir la identidad a otros procesos o capas de la aplicación, disponemos de dos formas básicas de hacerlo: automática y manual.

Flujo automático de identidad

Tiene lugar cuando es el CLR el que suministra ese flujo de identidad ante los requisitos de una aplicación, siempre que el objeto que efectúa la llamada se encuentre en el mismo contexto de ejecución (subproceso o *thread*) que el objeto llamado. En ese caso, .NET utiliza el mismo objeto *Thread.CurrentPrincipal* para ambos. Contexto de ejecución es un flujo de control de un programa, esto es, una secuencia de instrucciones en ejecución *vinculada a un SID concreto*. El concepto que debemos acuñar como subproceso es similar al de proceso, excepto que múltiples subprocesos se pueden ejecutar dentro del mismo proceso, compartiendo los datos y el código de programa.

Este concepto es válido para cualquier sistema operativo, pero, dentro del entorno en el que nos movemos, también conviene recordar que la propia Microsoft define el contexto de ejecución desde el punto de vista de los objetos de software implicados, de la siguiente forma⁸:

"El contexto de ejecución está determinado por el usuario o el inicio de sesión que está conectado a la sesión o que está ejecutando (llamando) un módulo. Establece la identidad para la que se comprueban los permisos para ejecutar instrucciones o realizar acciones. El contexto de ejecución está representado por un par de símbolos de seguridad: un símbolo de inicio de sesión y un símbolo de usuario. Los símbolos identifican las entidades de seguridad primaria y secundaria para las que se comprueban los permisos y el origen utilizado para autenticar el símbolo. Un inicio de sesión que se conecta a una instancia de SQL Server tiene un símbolo de inicio de sesión y uno o más símbolos de usuario en función del número de bases de datos a la que tiene acceso la cuenta."

Un símbolo de seguridad para un usuario o un inicio de sesión está formado por:

- Una entidad de seguridad de servidor o base de datos como identidad primaria*
- Una o varias entidades de seguridad como identidades secundarias*
- Cero o más mecanismos de autenticación*
- Los privilegios y permisos de las identidades primaria y secundaria"*

Flujo Manual de Identidad

Tiene lugar cuando el objeto llamado reside en una capa remota y no es posible la autorización debido a restricciones de infraestructura de la aplicación. Por ejemplo, en una llamada mediante .NET Remoting a través de un canal TCP.

Para implementar de forma manual el Flujo de Identidad, se requiere pasar los datos de identidad de forma distinta a como se pasan los datos normales entre llamadas.

⁸ Para una explicación más en detalle de este concepto, sugerimos la visita al sitio oficial de MSDN:
<http://msdn2.microsoft.com/es-es/library/ms187096.aspx>

Por ejemplo, si se están realizando llamadas a un servicio Web mediante SOAP, no debemos incluir la información de identificación en el cuerpo del mensaje SOAP, sino en su cabecera (de hecho, esta característica es la forma recomendada, aparte de la posible utilización de las Extensiones de los Servicios Web (tales como *WS-Security*), en este caso.

También se debe de tener en cuenta que el Flujo de Identidad no significa volver a pasar las credenciales (entre otras cosas, por que deberían de pasar cifradas, lo que supone otro tipo de problemas de comunicación y reconocimiento de identidades).

Algunas prácticas adecuadas para la implantación del Flujo de Identidad

La experiencia y la implantación de sistemas cada vez más complejos, aconseja como buenas prácticas de implantación, las siguientes:

- Utilizar el mecanismo más potente de autenticación disponible, dentro de los requisitos y restricciones impuestos por la aplicación.
- Si las técnicas utilizadas suponen pasar o almacenar información secreta de algún tipo, debe cifrarse esa información utilizando un canal seguro. A este respecto, en al versiones 1.0/1.1 de .NET ya disponíamos de el espacio de nombres *System.Security.Cryptography*, que implementa mecanismos bien comprobados de cifrado.
- Respecto al almacenamiento, puede considerarse el uso de DPAPI (*Data Protection API*), como lugar seguro de almacenamiento de datos en el propio sistema. Otra forma posible de almacenamiento seguro es la, ya mencionada clase *SecureString*, que utiliza internamente los algoritmos de cifrado incluidos en el espacio de nombres de soporte criptográfico.
- Garantice que la implementación permita el uso posterior de técnicas de auditoría, hasta la profundidad que los sistemas de seguridad necesiten.

Auditoria

La fase de medición de la efectividad del programa permite que el equipo de administración de riesgos de seguridad documente formalmente el estado actual de los riesgos de la organización. A medida que la empresa continúa por el ciclo de administración de riesgos, esta fase también contribuye a demostrar el progreso de administrar el riesgo con un nivel aceptable a lo largo del tiempo. Para facilitar la comunicación del progreso, en esta sección se presenta el concepto de un cálculo de riesgo de seguridad como un indicador de alto nivel del riesgo en una organización. El cálculo ayuda a demostrar que la administración de riesgos está realmente integrada en las operaciones de TI.

Hay muchos aspectos que pueden auditarse para ayudar en el control de la seguridad: un registro de las operaciones aceptadas y rechazadas, reduce el problema del repudio, lo que es especialmente crítico en el comercio digital. La consulta regular de los resultados de las auditorias permite mejoras constantes en la seguridad y la detección temprana de problemas potenciales.

Estas técnicas de auditoria pueden complementarse con las reuniones “post-mortem” ya comentadas en las fases de análisis y diseño, para extraer conclusiones de cara a futuros desarrollos.

Las posibles acciones a este respecto son muchas y van desde la implementación de soluciones personalizadas, pasando por el uso combinado de herramientas del sistema, tales como el seguimiento de los Registros de Eventos del Sistema, el uso de las listas SACL — *System Access Control List* – que llevan una auditoría de la utilización de ciertos recursos previamente definidos o establecidos de antemano, hasta la utilización de herramientas de terceros, o el control adicional mediante código por la propia aplicación.

Auditoría y Medición

Un estudio llevado a cabo por el *Government Accountability Office* de EE.UU. acerca de la administración de la seguridad de la información en las principales organizaciones no federales (GAO/AIMD-98-68), las pruebas directas constituyen el método más habitual para comprobar de forma eficaz el nivel de reducción de riesgo logrado por los controles de seguridad. Existen varios enfoques para llevar a cabo estos tipos de pruebas, entre los que se incluyen las herramientas automatizadas de evaluación de vulnerabilidades, evaluaciones manuales y pruebas de penetración.

En el documento antes citado sobre "Guías de Implantación de seguridad", se nos dan las pautas para este tipo de pruebas:

"En la evaluación manual, un miembro del equipo de TI comprueba que se ha aplicado cada control y que parece funcionar correctamente. Ésta puede ser una tarea lenta, tediosa y propensa a errores cuando se están comprobando numerosos sistemas. Microsoft ha publicado una herramienta de evaluación de vulnerabilidades gratuita y automatizada denominada Microsoft Baseline Security Analyzer (MBSA). MBSA puede analizar los sistemas locales y remotos para determinar las revisiones de seguridad críticas que faltan, así como otras configuraciones de seguridad importantes. En www.microsoft.com/technet/security/tools/mbsahome.mspx hay disponible más información acerca de MBSA. Aunque MBSA es gratuito y resulta muy útil, también hay disponibles otras herramientas de evaluación automatizadas de otros proveedores.

El otro enfoque mencionado anteriormente es el de las pruebas de penetración. En una prueba de penetración, uno o varios usuarios tienen permiso para realizar pruebas automatizadas y manuales para comprobar si pueden tener acceso a la red de una organización de diferentes formas. Algunas organizaciones realizan las pruebas de penetración con expertos de seguridad propios, mientras que otras contratan a expertos externos que están especializados en realizar estas pruebas.

Independientemente de quién efectúe las pruebas de penetración, el grupo de seguridad de información debe ser el encargado de administrar el proceso y de realizar el seguimiento de los resultados. Aunque las pruebas de penetración constituyen un enfoque eficaz, normalmente no detectan una amplia variedad de vulnerabilidades ya que no son tan exhaustivas como una evaluación de vulnerabilidades implementada correctamente. Por lo tanto, se recomienda complementar las pruebas de penetración con otros métodos."

Y todo esto, independientemente de lo que el sentido común, la experiencia en implantaciones anteriores y la posible adopción de software especial para las mediciones, pueda aportar a ese respecto.

La ejecución de código en .NET

En el caso de .NET, como ya sabe el lector, todo el código es administrado por el *Common Language Runtime* (CLR), lo que significa que es éste el responsable de intercalar una capa de

seguridad adicional al código en ejecución. La forma de realizarlo, que explicaremos en detalle en el siguiente capítulo, se basa en dos aspectos fundamentales: el concepto de evidencia (que el CLR obtiene de la ubicación desde donde se ejecuta el ensamblado, y de los metadatos del mismo) y el conjunto de Políticas de Seguridad que se administran en la configuración del sistema, y que permite o no la ejecución basándose en las evidencias obtenidas pudiéndose personalizar para que una aplicación utilice una política dada.

Dichas políticas de seguridad son mantenidas por el entorno de ejecución e independientes de las existentes en el sistema, de forma que un administrador ejecutando un programa con permisos restringidos no podrá soslayar las políticas establecidas por el programa o su evidencia.

El subsistema de seguridad de .NET se fundamenta en varios conceptos que explicaremos más adelante: la seguridad en el acceso del código (CAS o *Code Access Security*), que utiliza ese sistema de evidencias, y la seguridad dentro del propio código, que un desarrollador puede codificar adecuadamente para adaptar la aplicación a un entorno dado. De la aplicación de este conjunto de sistemas se deriva la posibilidad o denegación del permiso para realizar una acción por parte de una identidad.

En el lado de la autorización del sistema, se trata de un mecanismo de todo o nada. Se tiene permiso para acceder a la impresora de red o no se tiene, se puede o no acceder al sistema de archivos, etc.

Capítulo 4: Programación de la seguridad en .NET Framework

.NET como plataforma de ejecución segura

Cuando los ingenieros de Microsoft recibieron la aprobación final para abordar la construcción de la plataforma, una de las directrices principales era la de conseguir una infraestructura segura respecto a la ejecución basada en cómo el código era manejado por un administrador de ejecución: el **Common Language Runtime (CLR)**.

De hay que .NET se define como “**type safe**” desde sus orígenes, en el sentido de que la plataforma de ejecución garantiza que el código no va a realizar operaciones no permitidas o peligrosas, o intentar acceder a zonas de memoria directamente, o acceder a miembros de un tipo de forma incorrecta. También se garantiza que sólo aquellos usuarios con permiso podrán lanzar un programa dado.

Nota: Naturalmente, algunas de esas operaciones (típicas en lenguajes como C++), tenían que estar soportadas por el sistema, y de ahí la posibilidad de llamar a código no administrado desde código .NET. Pero sólo el mecanismo de seguridad del sistema operativo puede controlar las acciones realizadas por código no administrado.

Para obtener esto, se precisaba una estructura nueva del modelo de los propios ficheros que albergan el código fuente, así como el abandono de las dependencias del registro de Windows. En .NET, esos ficheros reciben en nombre de ensamblados, y pueden tener dos extensiones .EXE ó .DLL, dependiendo de si generan su propio proceso de ejecución, o se ejecutan en el proceso de otra aplicación que los llama.

Arquitectura de un ensamblado

La arquitectura de un ensamblado consta genéricamente de 3 partes, que –como si se tratase de un documento- podemos denominar cabecera, cuerpo y pie. En la cabecera, se sitúa la meta-información, esto es, la información sobre la información de las otras partes del ensamblado.

Cómo ésta hecho el propio ensamblado, de qué tipos⁹ consta, los recursos que se guardan en el pie, etc. En otra documentación, también podremos ver cómo se refieren al conjunto de elementos de cabecera como al *Manifest* del ensamblado, y al resto de él como a su *Body* o cuerpo.

Cuando el usuario solicita la ejecución de un programa (ensamblado del tipo EXE), interviene el administrador, que vigilará la correcta ejecución del contenido. En concreto, la librería MSCOREE.DLL es la encargada de realizar esas labores iniciales y establecer el contexto de ejecución. Además de esta librería principal, en .NET existe otra librería base (MSCORLIB.DLL) donde residen los tipos básicos de la jerarquía de clases del sistema. Una de las características que marca la seguridad de estos ensamblados es su ubicación y las condiciones necesarias para que un elemento pueda residir allí.

Estas librerías se guardan en una zona (un conjunto de directorios del sistema), especialmente protegida, llamada GAC (**Global Assembly Cache**). Ningún elemento nuevo puede ser ubicado allí si no se firma primero digitalmente. (Dentro del SDK, la herramienta SN.EXE permite generar firmas digitales y GACUTIL, fundir un ensamblado con una firma digital, y llevarla al GAC).

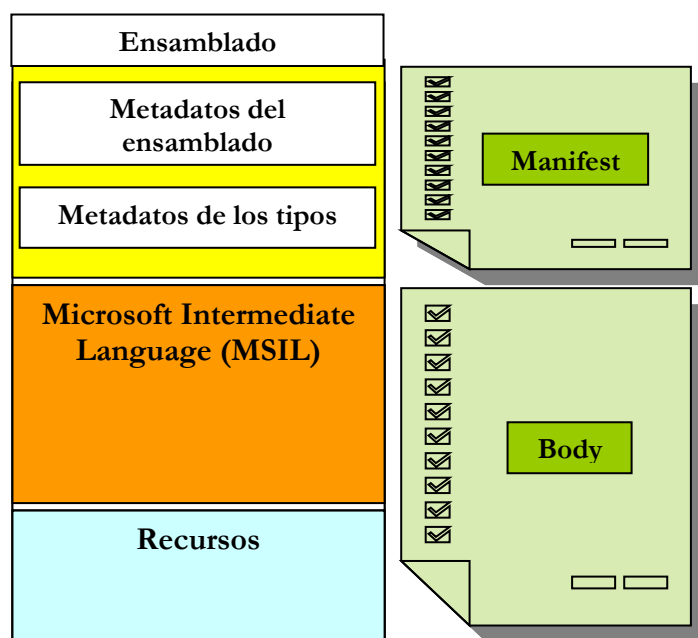


Fig.: Vista esquemática de un ensamblado .NET

⁹ En su obra de referencia “Essential .NET”, Don Box, con su proverbial magisterio define un tipo como “una abstracción reutilizable con nombre”. En la misma obra se comenta que el concepto de contrato de un tipo (utilizado también en XML y SOAP) es “su conjunto de miembros accesibles”. Es ese contrato el que define la información del ensamblado.

Ahora bien, cuando se establece ese contexto de ejecución, existe una información vital que decide (en tiempo de ejecución) si ésta es posible e incluso qué partes de ella estarán o no disponibles. En eso radica la seguridad del código inicialmente. El mecanismo mediante el cual se evalúan las características de ejecución se denomina CAS (**Code Access Security**) y su funcionamiento es como sigue:

Seguridad en el acceso al código (CAS)

La seguridad de acceso al código se basa en dos conceptos básicos: **impersonation** (que hemos estado traduciendo por personalización): característica mediante la cual un programa opera en nombre de una persona, y **confianza (trust)**: garantía previa de que a una determinada persona le corresponde el derecho de ejecución de un cierto fragmento de código.

Para conseguir esto, los dos elementos tienen que estar representados de alguna manera en el proceso de ejecución administrado, y esto se consigue en el primer caso mediante una política de creación de usuarios y grupos, que -en principio- no tiene por qué coincidir en absoluto con las establecidas en Windows por las propias ACL (Listas de control de Acceso). De hecho, en el Panel de Control, la instalación de .NET SDK introduce dos nuevos elementos en el Panel de Control, uno de los cuales (*Microsoft .NET Framework 2.0 Configuration*) permite la administración independiente de este conjunto de personalidades y facilita la asignación de permisos.

Roles y usuarios

La forma más usual de controlar lo que distintos grupos de usuarios con el mismo tipo de permisos tienen habilitado en el sistema, es mediante el concepto de rol. Un rol puede agrupar conjuntos de usuarios similares y facilitar la tarea de cambiar los permisos de los usuarios.

.NET soporta ambos conceptos y permite de forma sencilla solicitar información adicional o comprobar si un usuario pertenece o no a un determinado rol.

Un rol es una abstracción que encapsula un conjunto de capacidades operativas sobre el sistema. Por ejemplo, Administrador de Bases de Datos, Operador de Copias, o Lector del Registro. Un usuario se vincula con una personalidad concreta (que

no necesariamente con una persona, ya que podríamos tener varios usuarios en el mismo equipo).

El sistema de asignación de permisos, permite ubicar a un usuario dentro de un rol, o de varios, dependiendo de las funciones que se espere que el usuario tenga que realizar. La costumbre, hace que – inicialmente- el código se ejecute con la identidad, roles y permisos de la persona que lo ejecuta.

.NET introduce una novedad importante en este escenario, conocida como **CAS** (Seguridad en el acceso al código), que suministra un control fino sobre las operaciones y recursos a los cuales el código administrado tiene acceso. Con CAS **el acceso se basa en la identidad del código, y no en la del usuario que lo ejecuta**.

Algunos ejemplos de operaciones protegidas por CAS son:

- Creación, modificación y borrado de ficheros y directorios
- Lectura y escritura en el Registro de Windows
- Utilización de “sockets” para aceptar o iniciar una conexión de red
- Creación de nuevos Dominios de aplicación
- Ejecución de código no administrado.

Obviamente, deben de existir procesos iniciales que establezcan con claridad esa identidad del código, especialmente en un mundo distribuido y conectado, en el que su origen puede ser desconocido. Para obtener esa identidad, .NET introduce un concepto novedoso: la evidencia (**Evidence**).

El soporte de seguridad en .NET es muy amplio, e incluso se ha visto incrementado notablemente en la versión 2.0. La mayor parte de las características de seguridad (aunque no todas), se encuentran representadas en el espacio de nombres *System.Security*, que –a su vez- se subdivide en los espacios: *System.Security.AccessControl*, *System.Security.Authentication*, *System.Security.Cryptography*, *System.Security.Permissions*, *System.Security.Policy*, y *System.Security.Principal*.

Evidencias, permisos y políticas de seguridad

Pero, centrémonos primero en el escenario: Los 3 elementos clave de CAS son: la **evidencia**, los **permisos** y las **políticas de seguridad**.

Cuando el CLR carga un ensamblado, inspecciona algunas de sus características para determinar la identidad del código: su evidencia. Una vez obtenida esa información, el CLR la utiliza para pasarla como datos de entrada a un proceso conocido como "**Policy Resolution**" (Resolución de la Política de Seguridad). El resultado es un conjunto de operaciones y recursos protegidos a los cuales el código tiene acceso (permisos).

A lo largo de la ejecución, el código intentará ejecutar un conjunto variado de operaciones. Si se intenta ejecutar una operación protegida, el CLR examinará los permisos otorgados al ensamblado y determinará si el permiso es suficiente para que la acción solicitada continúe. El gráfico adjunto ilustra brevemente este proceso.

Veamos en qué consiste cada uno de estos elementos y cuál es el soporte de que disponemos para su manejo.

Evidencias

Se denomina así al conjunto de características utilizadas por el CLR para establecer la identidad de un ensamblado. El CLR incluye por defecto, siete tipos de evidencias. Cuatro de ellas, tienen que ver con la ubicación de dónde se ha obtenido el código (**Site**, **URL**, **Zone** y **Application Directory**). Otras dos tienen que ver con quién escribió el código (**Strong Name** y **Publisher**). Finalmente, la séptima clase de evidencia, **Hash**, se basa en los contenidos globales del ensamblado, y permite la detección de una compilación concreta del código, independientemente del número de versión.

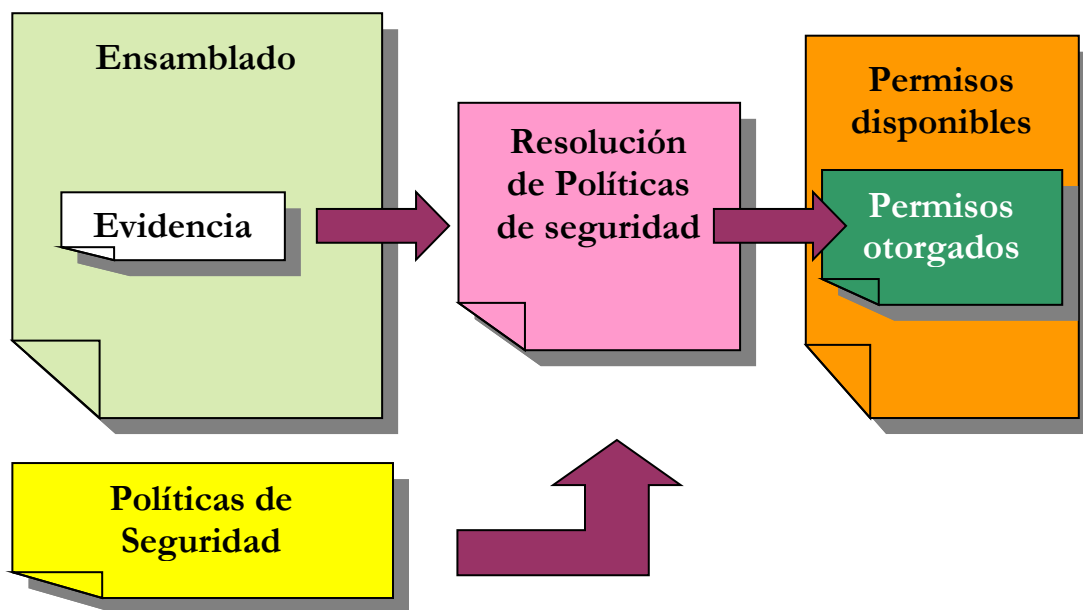


Fig.: Esquema operativo de la Resolución de Políticas de Seguridad en .NET

También es posible crear manualmente nuevos tipos de evidencia, denominados **Evidencias de Ensamblados** (en oposición a los citados, llamados **Evidencias de Host**, que son implementados por el entorno de ejecución). En el apartado dedicado a este elemento veremos su funcionamiento con más detalle. De momento baste saber que para poder utilizar evidencias personalizadas, es necesario extender el mecanismo de políticas de seguridad para que lo reconozca.

La librería de clases de .NET suministra acceso mediante programa a estos elementos a través de un conjunto de clases, vinculadas al espacio de nombres **System.Security**. En una aplicación que se responsabilice sobre su comportamiento en seguridad, los programadores pueden acceder a los datos de evidencias (aunque esa sea labor principal del CLR), de forma sencilla. El siguiente código permite acceder a las evidencias tipo **zone** de un ensamblado cualquiera.

```

using System.Reflection;
using System.Security.Policy;
using System.Collections;
using System.Collections.Generic;
... //otro código

public sealed class ClaseEvidencias
{

```

```
public static Zone LeerZonas(object obj) {  
    Evidence ev = obj.GetType().Module.Assembly.Evidence;  
    IEnumerator i = ev.GetHostEnumerator();  
    while (i.MoveNext()) {  
        Zone zona = i.Current as Zone;  
        if (zona != null) return zona;  
    }  
    return null; // si no hay zona accesible  
}
```

El código anterior incluido en una aplicación de escritorio, devolvería un objeto **Zone**, cuya propiedad **SecurityZone** vale "MyComputer".

Además del acceso a los objetos vinculados con la seguridad a través del espacio de nombres **System.Security**, en este caso, se utiliza **Reflection** para acceder al propio ensamblado.

Políticas de Seguridad

Se trata de un conjunto configurable de reglas que el entorno de ejecución utiliza durante el proceso de **Resolución de Políticas de Seguridad**. Podríamos establecer, por ejemplo, que cualquier código descargado del sitio Web <http://www.elavefenix.net> tenga permiso para escribir el fichero *C:\TextoPlano.txt* en el disco duro local, o que sólo el código firmado por Microsoft pueda ejecutar código no administrado.

Como ya indicamos antes, un nuevo elemento del Panel de Control permite configurar adecuadamente los niveles que podemos establecer para estas políticas de seguridad, estableciendo 3 zonas predeterminadas o ámbitos de actuación, que van del más general al más específico: **Enterprise**, **Machine** y **User**. La figura adjunta muestra la ventana de Administración de Políticas de Seguridad para la versión 1.1 del .NET Framework.

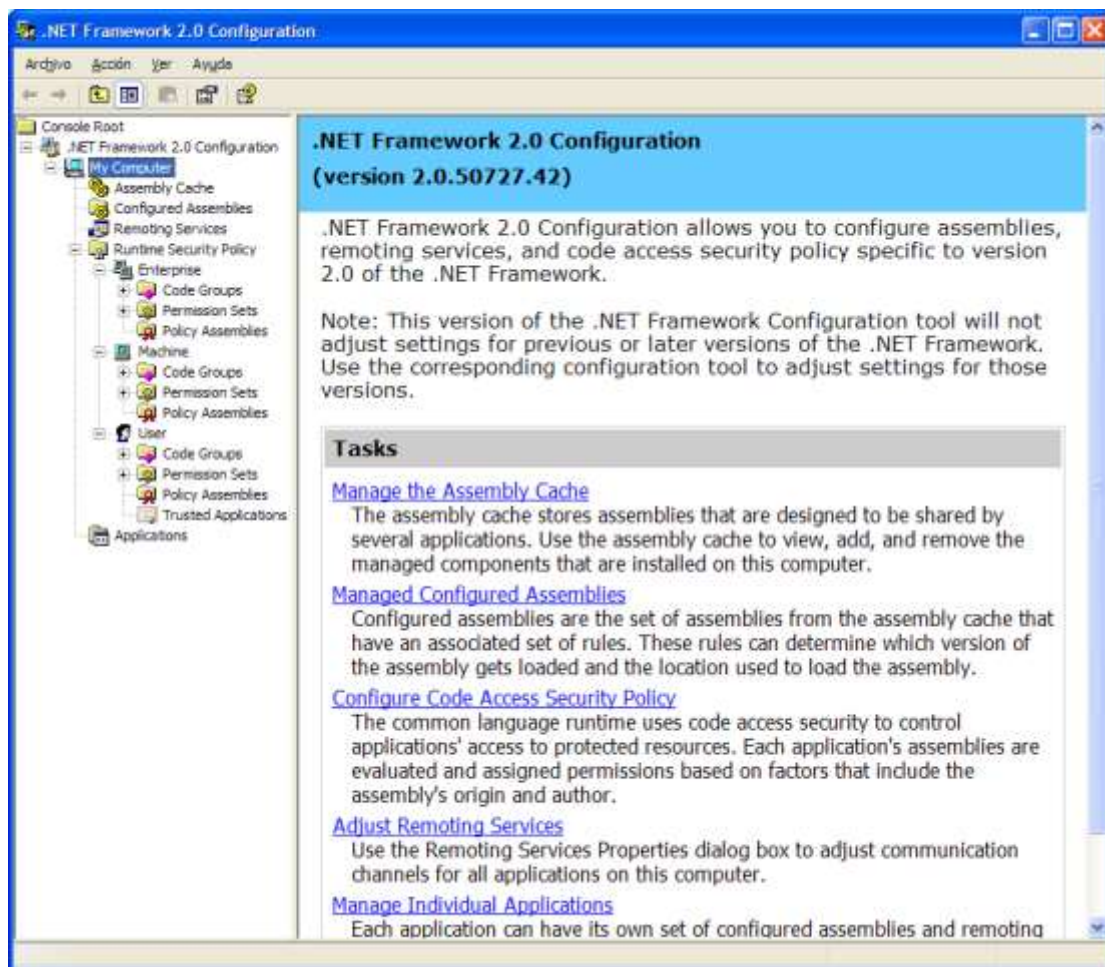


Fig x: Consola de Administración de Políticas de Seguridad en .NET Framework 2.0

(La herramienta de administración de .NET Framework 2.0 tiene también otras utilidades de interés, como la visualización y gestión de los ensamblados del Global Assembly Cache, o la configuración individual de aplicaciones, y –como puede verse en su pantalla principal- es específica de la versión de la plataforma).

Permisos

Para poder hacernos una idea previa de qué términos se manejan exactamente en relación con los conjuntos de permisos disponibles de forma predeterminada, basta con seleccionar uno cualquiera, y podremos observar en la ventana de detalles, de qué permisos consta cada conjunto. Por ejemplo, el conjunto de permisos **LocalIntranet** consta de los permisos **Environment Variables**, **File Dialog**, **Isolated Storage File**, **Reflection**, **Security**, **User Interface**, **DNS**, **Printing** e **Event Log**.

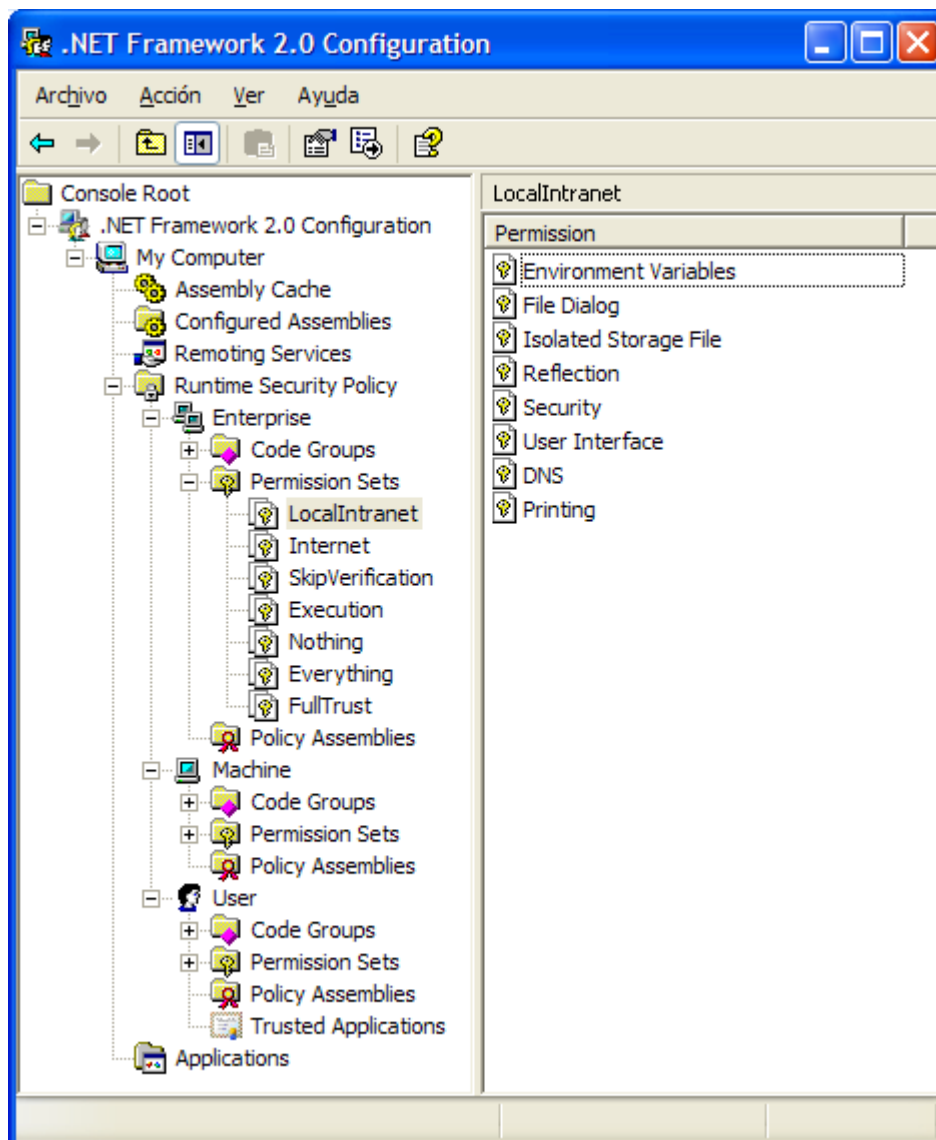


Fig.: Consola de Configuración de .NET 2.0, mostrando el conjunto de permisos para el área de LocalIntranet

A su vez, cada permiso específico, puede ser analizado y configurado, para conseguir un ajuste fino que se adapte a los principios expuestos en los capítulos anteriores, como el de la minimización de la superficie de exposición al riesgo.

Cualquier de los permisos otorgados a una zona puede ser comprobado con la opción del Visor de Permisos incluido en la herramienta (*Permission Viewer*, en gráfico adjunto)

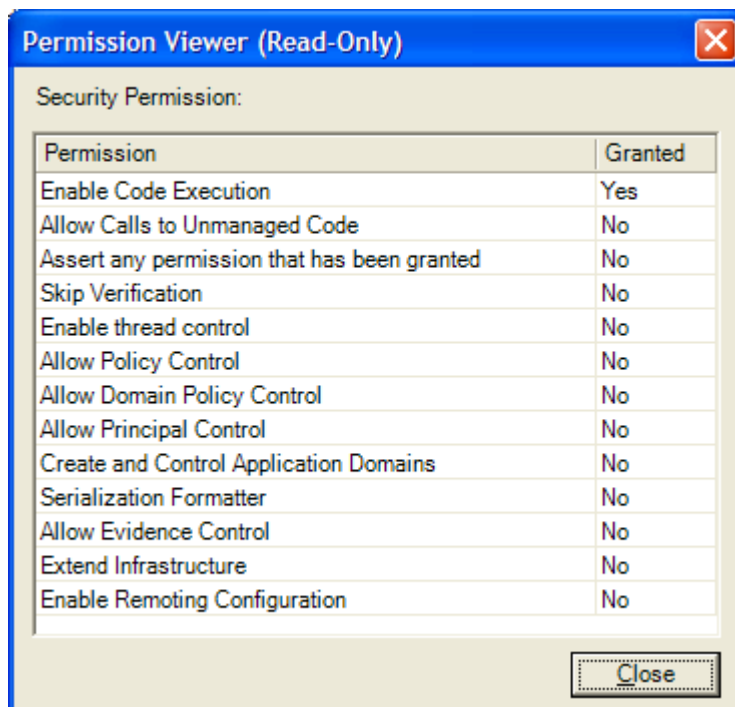


Fig.: Ventana del Visor de Permisos, mostrando los permisos otorgados al permiso "Code Execution"

Un ejemplo muy claro de esto lo constituye el acceso a ciertas zonas de disco. Dentro del Conjunto de Permisos **Everything**, encontramos el permiso individual **File IO**. Por defecto, este permiso concede la posibilidad de escribir, leer o borrar en cualquier zona del disco. Si queremos personalizar la ejecución, sólo tendremos que cambiar este valor, tal y como se aprecia en la figura adjunta, donde limitamos a una las acciones posibles (lectura), y las ubicaciones a un único directorio (C:\Temp). Esto sería un ejemplo -y más que recomendable, por otro lado- de la aplicación del principio de mínima exposición.

Uno de los usos más interesantes de esta configuración es la posibilidad de crear nuestras propias políticas de seguridad, configurando conjuntos de permisos personalizados donde se puedan expresar todos los requisitos analizados durante las fases iniciales del ciclo de vida. La herramienta de administración hace que la construcción de un conjunto sea una labor sencilla. Basta con seleccionar uno de los niveles sobre el que queremos actuar (por ejemplo, **Machine**) y bajo el epígrafe **Permission Sets** seleccionar "Añadir nuevo permiso". Una vez asignados un nombre y una descripción, pasamos a seleccionar dentro del conjunto de permisos individuales aquellos que necesitemos, y a configurarlos a nuestra medida. Cuando finalicemos el proceso, contaremos con un nuevo conjunto de permisos que aparecerá en nuestra lista, y que podremos visualizar o editar.

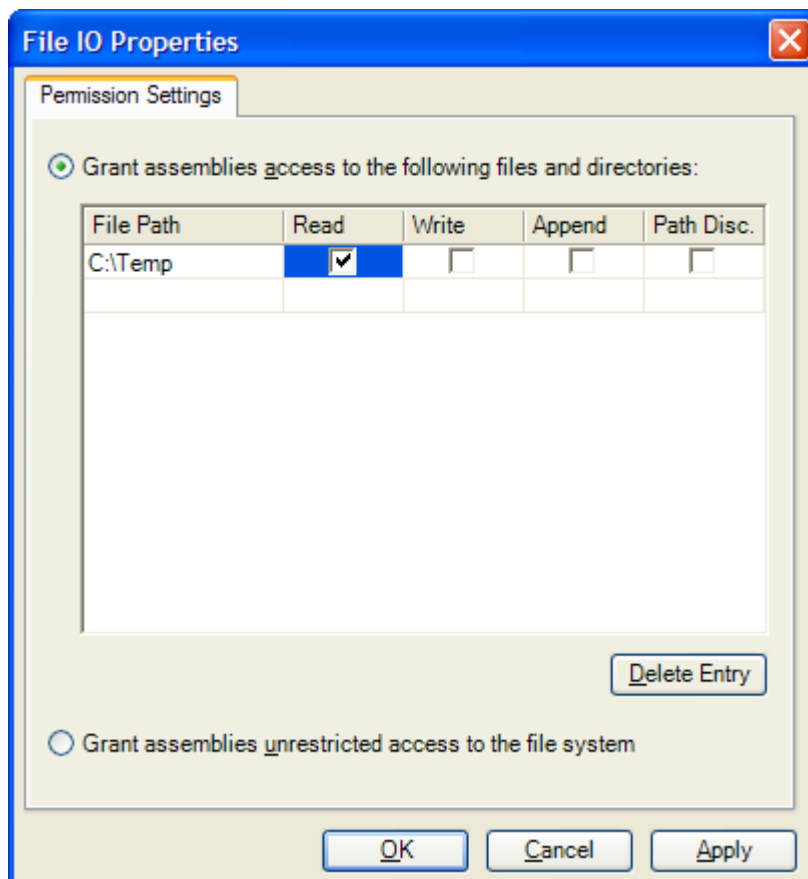


Fig. xx: Ventana de configuración de permisos individuales.

Todavía existe una cuarta zona o ámbito de actuación llamada Dominio de Aplicación (***ApplicationDomain***), que es aplicable a una aplicación concreta ejecutándose en un proceso dado. Esta característica se establece siempre mediante código.

La información establecida mediante el Administrador de Políticas de Seguridad (la de los 3 primeros niveles), es almacenada en varios ficheros XML dentro del sistema ¹⁰, mientras que –como hemos dicho– la cuarta, es especial, y puede configurarse para una aplicación en concreto mediante código llamando al método **System.AppDomain.SetAppDomainPolicy()**. Además de las herramientas mencionadas, los ficheros pueden ser manejados con la herramienta **caspol.exe**, o incluso con un editor cualquiera de texto plano, al ser de formato XML.

La combinación de las cuatro políticas de seguridad se denomina también **Jerarquía de Políticas de Seguridad**, y cada una garantiza un conjunto de permisos basados en las evidencias obtenidas. El conjunto resultante de la intersección de estas cuatro

¹⁰ En concreto en los directorios C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG, para los niveles *Machine* (*security.config*) y *Enterprise* (*enterprisesec.config*), mientras que el de usuario reside en ..\Datos de programa\Microsoft\NET Framework Config\v2.0.0.0\security.config bajo el perfil de cada usuario.

políticas es el conjunto permitido efectivo, como ilustra la figura adjunta:

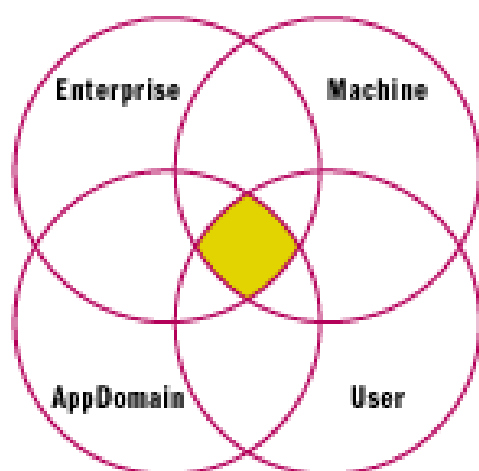


Fig. xx: Proceso de intersección de políticas de seguridad

Por otra parte el proceso de la propia resolución de políticas de seguridad puede ser monitorizado o incluso programado mediante código, permitiéndonos, por ejemplo la captura de excepciones donde preveamos que es posible la aparición de alguna incompatibilidad respecto a la seguridad. A las distintas formas de implementar la seguridad en el código se les denomina declarativa e imperativa, dependiendo de si se establece como un atributo de un tipo o como una sentencia de código.

Hay que recordar, además, que la aplicación de las políticas de seguridad de CAS no contraviene en forma alguna a aquellas establecidas por el sistema, suponiendo un filtro añadido a los permisos de ejecución: por ejemplo, puede que el código esté autorizado para una acción sobre el sistema de ficheros, pero una política vinculada a la administración del Directorio Activo, evite que el código realice su función. Es importante, por tanto, al encontrarnos con un mensaje de error, evaluar primero cuál de los dos posibles orígenes es el causante. A partir de ahí, seguiremos el hilo de secuencia de error hasta encontrar la causa.

Por último, existe la posibilidad de configurar una aplicación en concreto, bajo la opción "*Applications*", de la configuración de .NET Framework 2.0. Si seleccionamos esta opción se nos ofrece una ventana de selección (ver gráficos adjuntos), que nos permitirá seleccionar una aplicación concreta, con la posibilidad de examinar las dependencias que tenga esa aplicación de otros ensamblados del GAC o bien externos, establecer características personalizadas de comportamiento, distribución, etc.

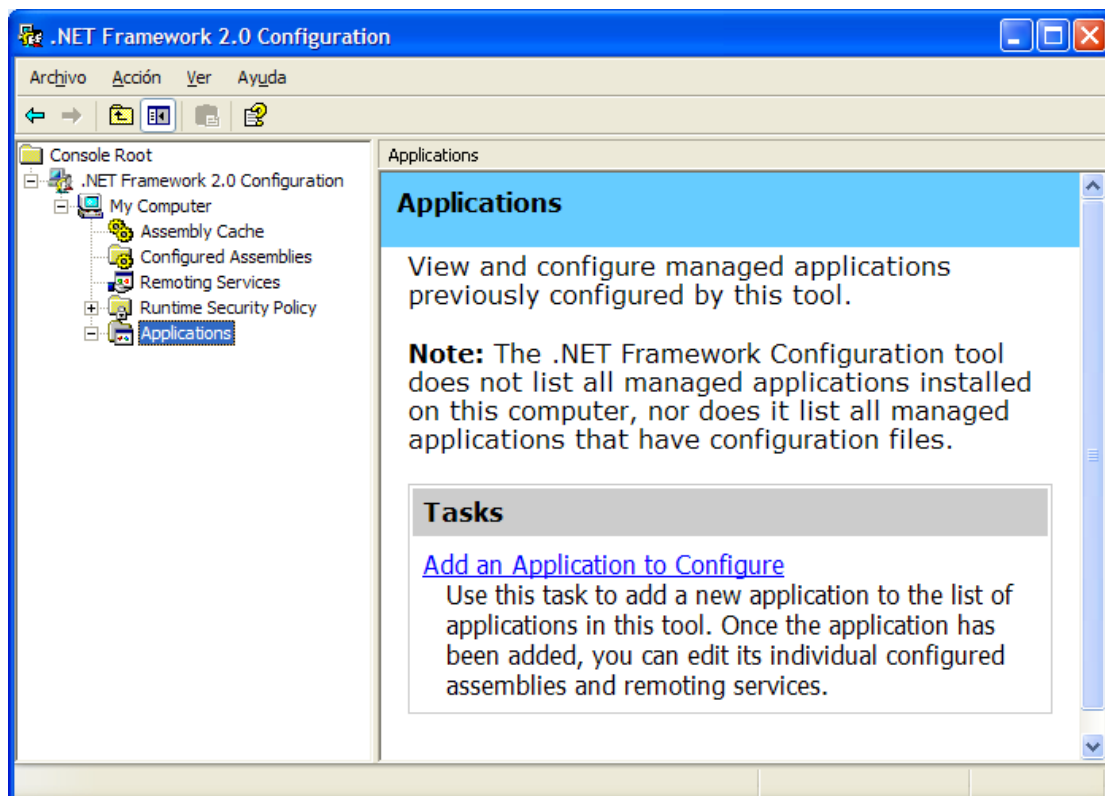


Fig.: Selección de configuración de aplicaciones individuales

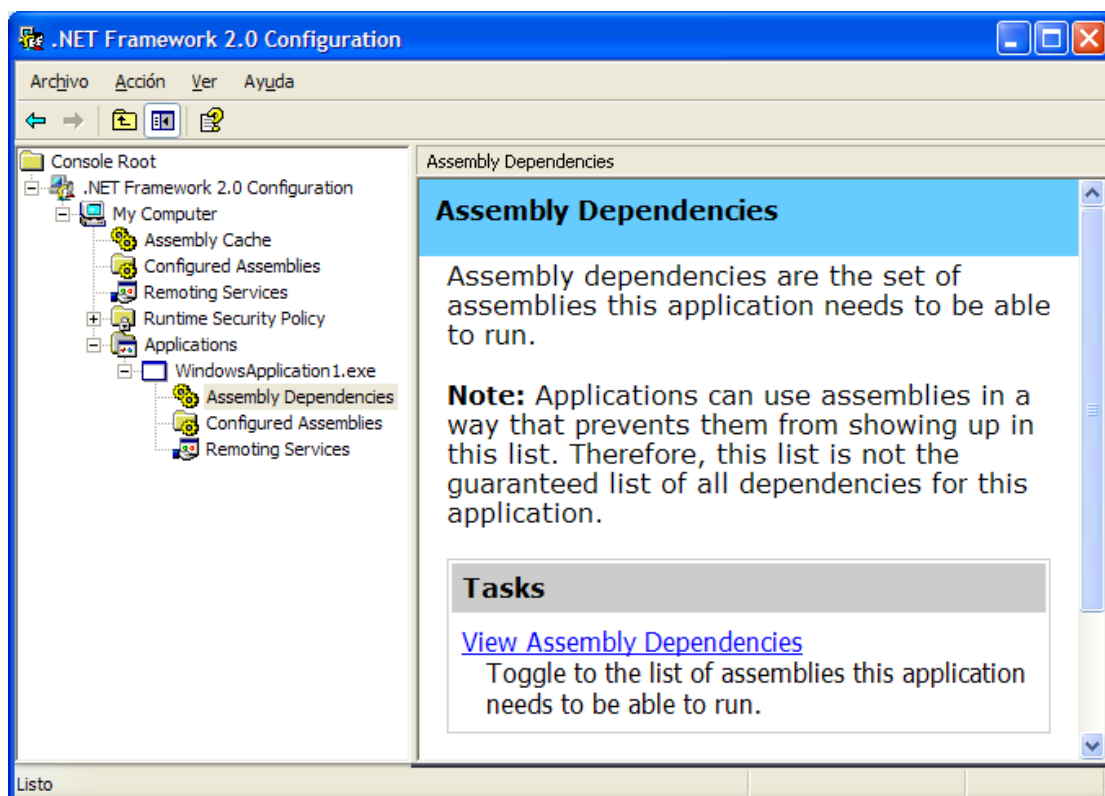


Fig.: Vista de dependencias de un ensamblado en la ventana anterior

Gestión de seguridad mediante código

Como ya se ha comentado, todos los objetos representados en el Modelo de Seguridad de .NET tienen una contrapartida programática en la jerarquía de clases, típicamente, dentro del espacio de nombres **System.Security**.

Acceso a Evidencias y Políticas mediante código

Desde el punto de vista del desarrollo, el CLR expone un acceso a estas políticas a través de la clase **System.Security.SecurityManager**, en su método **PolicyHierarchy()**. El siguiente código, lee los nombres y las ubicaciones del conjunto de políticas aplicables a un ensamblado cualquiera (recordemos que no residen en el ensamblado ni dependen de su contenido).

```
IEnumerator i = SecurityManager.PolicyHierarchy();
while (i.MoveNext())
{
    PolicyLevel level = (PolicyLevel) i.Current;
    Console.WriteLine("{0, 10}: {1}", level.Label,
        level.StoreLocation);
}
```

El sencillo código anterior produce (en la máquina donde lo estamos ejecutando) la siguiente salida:

```
Enterprise:
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\config\enterprisesec.
config
Machine:
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\config\security.confi
g
User:
C:\Documents and Settings\Marino\Datos de programa\Microsoft\CLR
Security Config\ v2.0.50727.42\security.config
```

Donde cabe destacar el hecho de que –de forma predeterminada– no existen políticas específicas de Dominio de Aplicación.

Niveles de Políticas de Seguridad

Cada Nivel de Política de Seguridad (NPS) consta de 3 componentes. Un NPS contiene una **lista de conjuntos de permisos con nombre**, (que podemos leer mediante la propiedad *PolicyLevel.NamedPermissionsSet*), cada uno de los cuales incluye cero o más privilegios.

También encontramos una **Jerarquía de Grupos de Código** (accesible mediante la propiedad *PolicyLevel.RootCodeGroup*), que se usa para determinar qué conjuntos de permisos aplicar ante una determinada evidencia. Finalmente, observaremos la existencia de una lista de **ensamblados de confianza** (*full-trust assemblies*), a la que podemos acceder mediante la propiedad *PolicyLevel.FullTrustAssemblies*), que –expresamente– contiene la lista de los ensamblados del sistema que contienen tipos que requieren de una política dada.

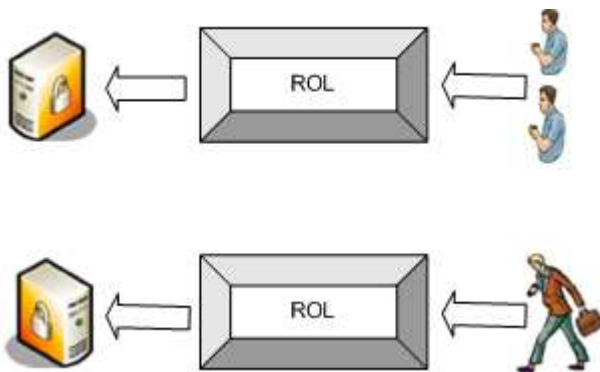


Fig xx: (Autorización) Cada grupo de usuarios se vincula con un rol, y éste con un conjunto de permisos

Podemos comprobar estos extremos modificando ligeramente el código fuente anterior, para solicitar a la clase *PolicyLevel* información sobre todos los conjuntos de permisos disponibles (que siempre tendrán que coincidir con los visibles a través de la consola de administración del Panel de Control). En nuestro caso, como la lista es bastante larga, hemos incluido en un formulario básico un control *RichTextBox* y vamos añadiendo a su contenido la información que devuelve la clase:

```
using System.Security.Permissions;
/* Otro código
...
*/
IEnumerator i = SecurityManager.PolicyHierarchy();
while (i.MoveNext())
```

```

{
    PolicyLevel level = (PolicyLevel) i.Current;
    foreach (PermissionSet NPS in level.NamedPermissionSets)
    {
        rtb.Text += level.Label + Keys.Enter + NPS.ToXml();
    }
}

```

Nótese que el acceso se sigue realizando a través de la clase *PolicyLevel*, pero, en este caso, recogemos el valor colectivo devuelto por la propiedad *NamedPermissionSets*, que devuelve un conjunto de objetos *PermissionSet*, a su vez, definidos en el espacio de nombres *System.Security.Permissions*.

El resultado de la ejecución es una larga lista de conjuntos de permisos en formato XML, del cual resumimos aquí el primero de ellos:

```

EnterpriseEnter<PermissionSet
class="System.Security.NamedPermissionSet"
    version="1"
    Unrestricted="true"
    Name="FullTrust"
    Description="Allows full access to all resources"/>

```

... y la lista continúa hasta el último:

```

UserEnter<PermissionSet class="System.Security.NamedPermissionSet"
    version="1"
    Name="Execution"
    Description="Permits execution">

```

... a continuación se listan todos objetos de la clase *PermissionClass*:

```

UserEnter<PermissionSet class="System.Security.NamedPermissionSet"
    version="1"
    Name="Execution"
    Description="Permits execution">

```

... y así sucesivamente.

Disponemos, pues, de mecanismos de acceso a toda la información de seguridad relacionada con CAS desde nuestro código, e incluso podemos comprobar en todo momento cuál es la situación para cualquier ensamblado.

Las notables mejoras en la seguridad respecto a versiones anteriores y las nuevas capacidades de la versión 2.0 de .NET

Framework, ofrecen además, un valor añadido en seguridad, como sucede con la clase **SecureString**, que utiliza internamente un algoritmo de cifrado para permitir el almacenamiento de cadenas de forma segura.

Objetos de soporte de autenticación e identidad

Para .NET, el usuario de Windows se encuentra representado por la clase **WindowsPrincipal**, que, en su propiedad **Identity**, permite leer las características de conexión o acceso de ese usuario.

Nota: Para implantar la práctica de la ejecución y pruebas con privilegios mínimos, una práctica interesante consiste en ejecutar procesos de la aplicación bajo la identidad de otro usuario diferente mediante el comando del sistema *runas*.

Autenticación: clases que implementan las interfaces **IIdentity** e **IPrincipal**

Dentro del espacio de nombres *System.Security.Principal* existen algunas clases especiales relacionadas con la autenticación de usuarios. La interfaz *IIdentity* define 3 propiedades clave que deben de soportar los objetos que manejen identidades: *AuthenticationType*, *IsAuthenticated* y *Name*. Por su parte, *IPrincipal*, requiere una propiedad de tipo *IIdentity* más el método *IsInrole()*, para comprobar la pertenencia de una identidad a un rol definido en el sistema.

Para determinar con qué identidad se ejecuta nuestro programa, y obtener toda la información pertinente, el código puede hacer uso de las clases *WindowsPrincipal*, *WindowsIdentity*, *GenericPrincipal* y *GenericIdentity*, que implementan las interfaces antes citadas. Veamos algunos ejemplos de utilización de estas clases:

```
//WindowsIdentity representa a un usuario del sistema
WindowsIdentity wi = WindowsIdentity.GetCurrent();
    MessageBox.Show("Tipo de Autenticación: " +
wi.AuthenticationType);
    MessageBox.Show("Nivel de Personalización: " +
wi.ImpersonationLevel.ToString());
    MessageBox.Show("Está autenticado: " +
wi.IsAuthenticated.ToString());
    MessageBox.Show("Windows Identity: " + wi.Name);
```

Por su parte, *WindowsPrincipal*, permite al usuario comprobar la pertenencia a un rol o grupo de usuarios de un usuario dado. Por ejemplo, el código siguiente comprueba si la identidad definida en el código anterior pertenece al rol de Administradores:

```
WindowsPrincipal wp = new WindowsPrincipal(wi);  
MessageBox.Show(wp.IsInRole("Administrador").ToString());
```

En el caso de las clases genéricas, permiten crear una identidad en el sistema a partir de los datos introducidos por un usuario externo. Supongamos que ese usuario introduce sus datos en las cajas de texto *txtUsuarioLogin* y *txtUsuarioPWD* de una página Web y que, con esos datos, realizamos una validación contra un sistema de base de datos. Si ese usuario debe de poder operar ciertas opciones de la aplicación podríamos crear una identidad *GenericPrincipal* mediante el siguiente código:

```
GenericIdentity Identidad = new GenericIdentity(txtUsuarioLogin.Text);  
string[] arrayRoles = Roles.GetAllRoles();  
  
GenericPrincipal principal = new GenericPrincipal(Identidad, arrayRoles);  
  
// Por último, vinculamos el usuario al subproceso de ejecución  
Thread.CurrentPrincipal = principal;
```

Donde *roles* es una clase definida en el espacio de nombres *System.Web.Security*, que además dispone de otros interesantes métodos, como *GetAllRoles()*, *GetRolesForUser()*, *GetUsersIsInRole()*, *CreateRole()* y muchos otros, y la propiedad de lectura/escritura *CurrentPrincipal* asociada al subproceso activo (*Thread*) permite leer o establecer una identidad definida por código al sistema en ejecución. El gráfico adjunto refleja la relación existente entre ambas interfaces.

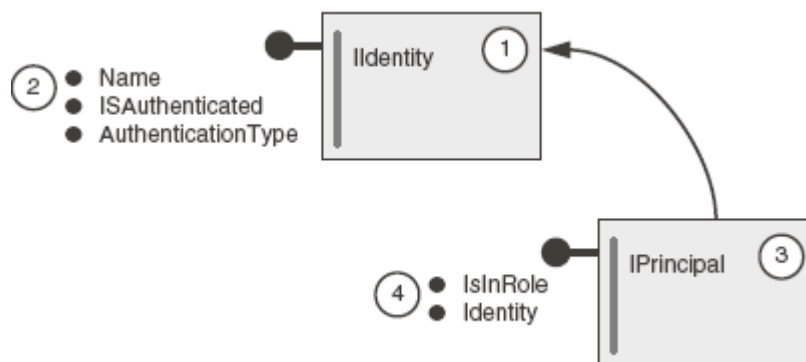


Fig. Relación entre las interfaces IIdentity e IPrincipal

Otra forma de establecer la identificación –para aplicaciones que no sean ASP.NET– es mediante la llamada al método *SetPrincipalPolicy()* de la clase *AppDomain* que representa al dominio actual de la aplicación. Permite vincular tipos específicos de identidades *IPrincipal* con este dominio:

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

Se puede llamar al principio de la aplicación, para asegurar que todas las hebras de ejecución se vinculan con el objeto Principal deseado, si bien debemos de tener en cuenta dos cosas:

- La llamada a *SetPrincipalPolicy()* sólo será efectiva si no existe un objeto Principal predeterminado para el dominio de la aplicación.
- En aplicaciones ASP.NET, la asignación del objeto Principal se produce en Global.ASAX, dentro del evento *AuthenticationRequest*, por lo que no debemos asignarlo manualmente.

El trabajo con Identidades en Autorización

.NET ofrece 4 objetos principales para trabajar con identidades: *WindowsIdentity*, *GenericIdentity*, *PassportIdentity* y *FormsIdentity*, de los que hemos visto el uso de los dos primeros. Los otros dos están directamente vinculados con aplicaciones Web, y les comentaremos más adelante.

También es posible la creación de clases personalizadas de manejo de identidad, mediante la implementación de la interfaz adecuada en una clase de usuario.

Para establecer la autorización mediante código podemos utilizar la técnica de asignar la identidad deseada a la propiedad

CurrentPrincipal del objeto *Thread*, tal como hacíamos más arriba en el ejemplo de la clase *GenericPrincipal*. Si tuviéramos que comprobar si una clase *Principal* está en un rol necesario para permitir la ejecución de un fragmento de código, podríamos hacerlo mediante:

```
bool EstaEnRol = Thread.CurrentPrincipal.IsInRole("Administradores");
```

Identidades en ASP.NET

En el caso de ASP.NET, el funcionamiento es diferente al del resto de aplicaciones .NET. El usuario está representado a nivel de sesión, mediante la propiedad *User*, que pertenece al objeto *HttpContext*. Después de producirse el evento *OnAuthenticate* en la clase representada por *Global.ASAX*, el CLR actualiza el valor de *Thread.CurrentPrincipal* con el valor de *Context.User*.

En caso de que quisiéramos comprobar una autorización similar a la anterior desde ASP.NET, usaríamos:

```
bool EstaEnRol = Context.User.IsInRole("Administradores");
```

Nota: El comportamiento de *Context.User* respecto a *Thread.CurrentPrincipal*, no es transitivo. Esto es, un cambio de *Context.User* producirá un cambio automático del otro, pero lo contrario no es cierto. Este tipo de modificaciones dinámicas del entorno de seguridad en ejecución requieren explícitamente de permisos del tipo *Code Access Security*. Por defecto todas las aplicaciones instaladas en el sistema local disponen del permiso, ya que se ejecutan bajo el conjunto de permisos *Full Trust*.

Es posible asignar manualmente el permiso requerido mediante la utilidad *Caspol*, otorgando el permiso *SecurityPermissionAttribute.ControlPrincipal* a la aplicación deseada.

Además, en este caso, las aplicaciones ASP.NET se ejecutan, de forma predeterminada, bajo la cuenta de Windows llamada *ASPNET*, y podemos considerar que, desde el punto de vista de la seguridad es como si hubiese dos personalidades. De cara al contexto de ejecución de ASP.NET, la especificada mediante el mecanismo de autenticación (el nombre e información asociada con el usuario concreto). De cara al sistema, la personificada por la cuenta *ASPNET*.

¿En qué ocasiones se puede producir el salto de una a otra durante la ejecución? Cuando ASP.NET solicite del sistema servicios que presuponen la utilización de un permiso. Por ejemplo, el acceso al sistema de archivos, o la inserción de valores en una base de datos. Mientras para ASP.NET la identidad es la de `Thread.CurrentPrincipal`, para el sistema es la cuenta ASP.NET.

Personalización (*Impersonation*)

La solución al problema que se presenta en estos casos viene dada por una técnica denominada ***Impersonation*** (podríamos traducirla por personalización o suplantación controlada de identidad).

Podemos hacer que un proceso asuma la identidad de otro objeto válido del sistema (de cara a una operación dada) mediante el método *Impersonate()* del objeto *WindowsPrincipal*. Requiere tener credenciales sobre el servidor, para poder llamar al API `LogonUser`, y en ocasiones puede entrañar cierta complejidad. Por ello, debe de utilizarse con cuidado.

Además una vez concluida la tarea, debemos de revertir la situación al estado previo en que se encontraba antes de la llamada, lo que puede conseguirse mediante el objeto *WindowsImpersonationContext*, a través de su método *Undo()*. A continuación mostramos una plantilla de código que realizaría algunas acciones utilizando este mecanismo:

```
// Asumir otra identidad para realizar operaciones protegidas
WindowsImpersonationContext impersonationCtx =
    WindowsIdentity.Impersonate(CuentaDeUsuario);
// Realizar operaciones bajo esa identidad
// Revertir a la identidad real del proceso
impersonationCtx.Undo();
```

Almacenamiento seguro de información crítica

En toda aplicación segura existe información crítica que debemos proteger contra un acceso no autorizado. Claves de cifrado, claves de firma digital, contraseñas, e incluso, en ocasiones, ficheros completos de información de configuración, o de otro tipo (aunque esta no sea, de entrada la mejor aproximación al problema).

Hay que añadir que la seguridad total de esta información no es posible con el hardware disponible hoy en día, así que el truco consiste en poner las cosas lo más difíciles posibles para los

posibles atacantes. Y nunca olvidemos que un secreto muy compartido no es tal.

También tenemos que tener en consideración la necesidad real de almacenamiento de información secreta. Muchas veces, tal cosa no es necesaria, bastando con el almacenamiento de una clave o *hash* que establezca un mecanismo de verificación. Esto es suficiente muchas veces para verificar que un usuario es quien dice ser sin necesidad de guardar la clave misma. Los algoritmos de *hash* son unidireccionales, sirven para comprobar la veracidad del contenido pero no tienen un mecanismo reversible.

No obstante, la primera pregunta que debemos hacernos se refiere el destino de la información protegida. No siempre va a ser un fichero; de hecho, podría tratarse de la memoria (por estar trabajando con un objeto *MemoryStream*, por ejemplo). A continuación vamos a revisar algunas técnicas de almacenamiento protegido que van desde la protección de datos almacenados en memoria, hasta la salvaguarda de información en una base de datos protegida por el propio sistema y accesible solamente mediante un API especial llamada *Data Protection API* (DPAPI).

Creación de valores *hash* mediante Criptografía

Resulta sencillo crear un valor *hash* utilizando las librerías criptográficas de .NET 1.x y 2.0¹¹. Todas las funciones *hash* toman como entrada una matriz de bytes (como veremos, las nuevas funciones de protección de la versión 2.0, también lo hacen).

El trabajo de codificación lo realiza la clase *MD5CryptoServiceProvider*, una de las clases de soporte de algoritmos de encriptación en el espacio de nombres *System.Security.Cryptography*. Este conjunto de clases soporta los algoritmos de *hashing* más reconocidos de la industria: MD5, SHA-1, SHA-256, SHA-384 y SHA-512. A estos, hay que añadir una versión especial de algoritmos con clave (llamada clave MAC), HMACSHA (Algoritmo *hash* SHA con clave) y MACTripleDES (Algoritmo *hash TripleDES* con clave) Resulta de una complejidad astronómica averiguar la clave original a partir únicamente del código *hash*, así como crear datos que concuerden con un *hash* determinado¹².

¹¹ En el apartado dedicado a Criptografía estudiaremos el soporte .NET de algoritmos simétricos y de clave pública.

¹² Parece haber una excepción: en 2005, una universidad de la República Popular China, manifestó haber conseguido “romper” el código SHA-1 utilizando técnicas de computación distribuida. Este hecho ha movido a más de una compañía a rehacer su software utilizando el algoritmo SHA-512, que requeriría un trabajo de fuerza bruta no al alcance los ordenadores actuales).

MSDN¹³ ilustra gráficamente el soporte de dichos algoritmos en .NET mediante el siguiente gráfico:

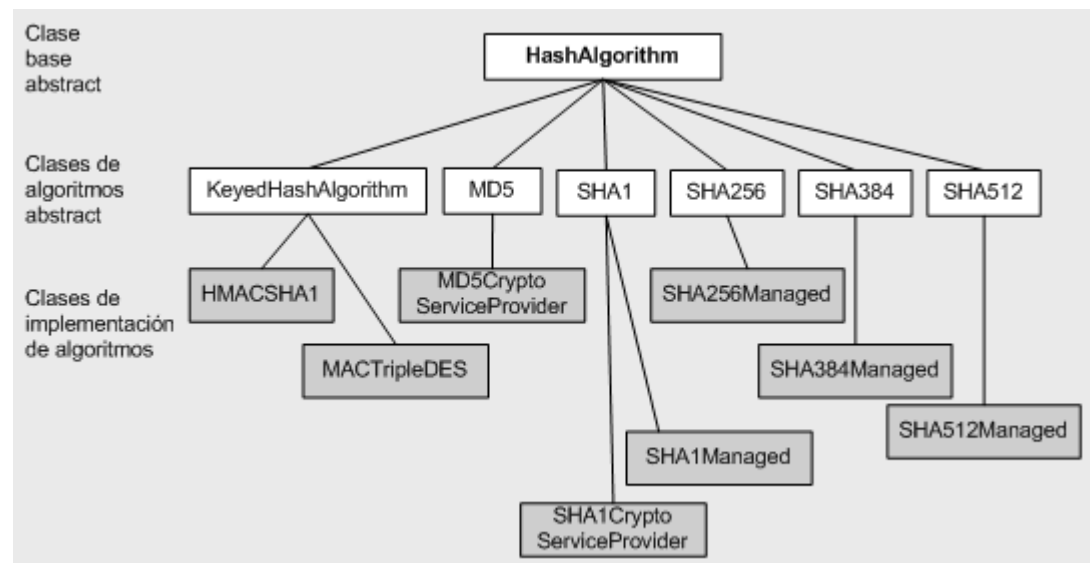


Fig.: Grafo del soporte de algoritmos de *hash* en .NET

Veamos un ejemplo de funcionamiento en que creamos primero un valor *hash* y después lo comparamos con otro, para su validación, utilizando el algoritmo MD5:

```

string Datos = "Datos originales sin codificar";
//Crea una matriz de bytes a partir del origen
byte[] tmpOrigen = ASCIIEncoding.ASCII.GetBytes(Datos);
//Calcular el hash basado en datos de origen.
byte[] tmpHash = new
MD5CryptoServiceProvider().ComputeHash(tmpOrigen);
//Mostramos el hash producido como valor hexadecimal
richTextBox1.Text = "Valor hash\n\n";
int i;
StringBuilder CadSalida = new StringBuilder(tmpHash.Length);
for (i = 0; i < tmpHash.Length; i++)
{
    CadSalida.Append(tmpHash[i].ToString("X2"));
}
//Mostramos la cadena en un RichTextbox
richTextBox1.Text = CadSalida.ToString() + "\n\n";
  
```

Para realizar la comprobación deberemos de proceder recorriendo cada uno de los *bytes* del primer *hash* y comparándolos con el

¹³ http://www.microsoft.com/spanish/msdn/arquitectura/BuildSecNetApps/html/Appendix_04.asp

segundo. Sólo si todos son iguales coincidirá el *hash*. El código es sencillo:

```
string Datos2 = "Datos modificados sin codificar";
//Operamos como anteriormente
byte[] tmpOrigen2 = ASCIIEncoding.ASCII.GetBytes(Datos2);
//Calcular nuevo hash
byte[] tmpHash2 = new
MD5CryptoServiceProvider().ComputeHash(tmpOrigen2);
//Para compararlos recorreremos los valores hexadecimales
//en busca de diferencias.
bool bIguales = false;
if (tmpHash2.Length == tmpHash.Length)
{
    int i = 0;
    while ((i < tmpHash2.Length) && (tmpHash2[i] == tmpHash[i]))
    {
        i += 1;
    }
    if (i == tmpHash2.Length)
    {
        bIguales = true;
    }
}

if (bIguales)
    richTextBox1.Text += "Son iguales";
else
    richTextBox1.Text += "El valor hash no coincide";
```

La ventaja estriba en que no necesitamos conocer la cadena original para saber si ha sido alterada, si coincide su valor *hash* es que está intacta. Este método se utiliza muchas veces de forma combinada, junto a los algoritmos simétricos y/o de clave pública, para formar un conjunto, una infraestructura completa de transporte seguro. (Ver más adelante Infraestructura de Clave Pública, o PKI).

Mejorando el *hash* con un poco de *Salt*

El problema potencial que podría presentarse con esta aproximación es que dos usuarios con la misma contraseña podrían dar pistas a un hacker potencial. Naturalmente, podríamos complicar la contraseña añadiendo información propia del usuario, como iniciales, etc., pero esa solución seguiría estando sujeta a problemas similares. Sin embargo, se puede conseguir que el *hash* así creado tenga un nivel de resistencia a los ataques superior si añadimos un elemento aleatorio en el momento de crear el *hash*, si

bien tenemos que tener en cuenta que dicho valor también deberá de ser guardado de alguna forma. Ese valor se denomina *salt*.

Tenga en cuenta el lector que los algoritmos utilizados en criptografía y que están soportados por el espacio de nombres *Cryptography*, han sido sobradamente comprobados, como se comenta en el apartado de Criptografía.

Uno de los proveedores para Criptografía suministra ese tipo de servicios. Se trata de la clase *wrapper* RNGCryptoServiceProvider, cuyas 3 letras iniciales responden a las siglas de Random Number Generador (Una clase *wrapper* es una clase .NET que permite acceso directo a API's del sistema, y en este caso, a CryptoAPI).

Una vez conseguido ese valor aleatorio se le añade a la cadena a calcular para completar el proceso y dificultar aún más su descifrado mediante ataques. Un código que podríamos anteponer al utilizado para el *hashing* anterior podría ser, sencillamente, el siguiente:

```
//ADICION DE SALT A UN HASH
byte[] Sal = new byte[8];
RNGCryptoServiceProvider rng= new RNGCryptoServiceProvider();
rng.GetBytes(Sal);
// ...y ahora usamos el valor de Sal junto a nuestra cadena original
```

Data Protection API (DPAPI)

En otras ocasiones, no se trata de cifrar el contenido, sino de almacenar éste en un lugar seguro. Naturalmente, una base de datos puede ser un sitio adecuado, pero –a veces- las aplicaciones necesitan guardar datos simples sin depender de sistemas externos de almacenamiento. Lo idóneo sería poder hacerlo dentro del propio sistema operativo. Para dar una solución a este problema, a partir de Windows 2000 y en todas las versiones posteriores, se incluyó un mecanismo de almacenamiento interno del sistema que no es visible ni accesible más que a través de las llamadas a las funciones que suministra el API mencionada.

Se trata de una arquitectura basada en el acceso mediante el suministro de una contraseña, y, por tanto, tiene el inconveniente de que esa contraseña debe ser a su vez guardada en alguna parte para su uso posterior. No requiere de ningún artificio criptográfico: de su seguridad se encarga el propio sistema, y el propio Login sirve de fuente básica para la codificación del almacenamiento.

Protección de datos en memoria

Una de las nuevas características de seguridad que aporta la versión 2.0 de .NET Framework es la clase estática (de hecho no dispone de constructor, y está marcada como *sealed*) *ProtectedMemory*. Solo dispone de dos métodos *Protect()* y *Unprotect()* y permite proteger/desproteger una matriz de bytes utilizando el propio array como origen y destino de las operaciones. El array pasado debe de ser múltiplo de 16.

En el ejemplo siguiente, un sencillo formulario con dos botones y dos cajas de texto es usado para llamar a los dos métodos, después de haber hecho referencia expresa a la librería *System.Security*, y en el código, al espacio de nombres *System.Security.Cryptography*.

El código es el siguiente:

```
// Datos originales cifrar (debe ser múltiplo de 16).
// Se definen a nivel de formulario
byte[] clave = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };

private void btnCifrar_Click(object sender, EventArgs e)
{
    // Encripta y almacena en el mismo array
    ProtectedMemory.Protect(clave, MemoryProtectionScope.SameLogon);
    txtClave.Text = "";
    foreach (byte b in clave)
    {
        txtClave.Text += b.ToString() + "-";
    }
}

private void btnDescifrar_Click(object sender, EventArgs e)
{
    // Descifra y lo deja en el mismo array
    ProtectedMemory.Unprotect(clave, MemoryProtectionScope.SameLogon);
    txtDescifrado.Text = "";
    foreach (byte b in clave)
    {
        txtDescifrado.Text += b.ToString() + "-";
    }
}
```

El segundo argumento pasado a los métodos de cifrado especifica quien tiene los permisos para descifrar la clave: *CrossProcess* (cualquier proceso), *SameLogon* (solo código en el mismo contexto

de uso que lo inició) y *SameProcess* (solo el mismo proceso que lo inició).

La salida, con las claves cifradas y descifradas en sendas cajas de texto, tiene el siguiente aspecto:

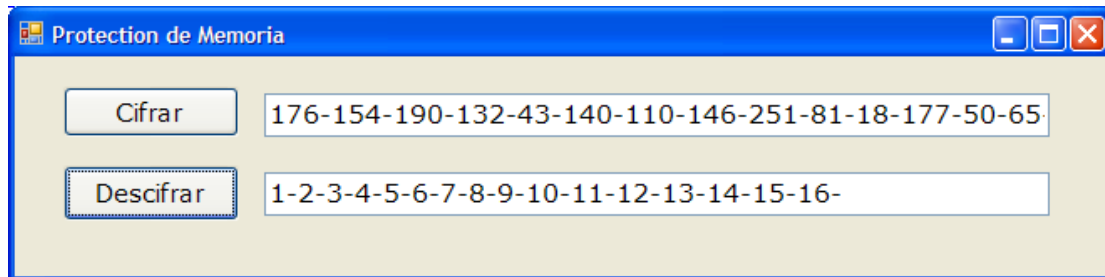


Fig. xx: Salida del programa anterior

(Los guiones se usan como separador de cada byte, por razones de claridad en la lectura)

Protección de datos en escritura a ficheros

Pero, ¿qué sucede si lo que lo queremos usar esta técnica para grabar una secuencia de caracteres cifrada a un fichero? El mismo espacio de nombres nos ofrece una solución a través de la clase *RNGCryptoServiceProvider*, que permite la creación una matriz de entropía ¹⁴ que será utilizada posteriormente por la clase *ProtectedData* para desordenar el contenido cifrándolo de acuerdo con los valores de esa entropía. El código sería el siguiente:

```
//Por sencillez en el código, se omiten las comprobaciones
byte[] ACifrar;
// Asignamos datos originales a cifrar
ACifrar = UnicodeEncoding.ASCII.GetBytes("Datos a cifrar de cualquier
longitud.");
// Crea el fichero
FileStream fStream = new FileStream("Data.dat",
 FileMode.OpenOrCreate);
// Crea un nivel de desorden (entropía), expresado como un array
// que se pasa como argumento a la clase RNGCryptoServiceProvider
byte[] entropia = new byte[16];
// Crea (pero no vuelve a usar) una instancia de
RNGCryptoServiceProvider y rellena el array con un valor aleatorio.
new RNGCryptoServiceProvider().GetBytes(entropia);
// Cifrar datos en memoria, sobre el mismo array
byte[] DatosCifrados = ProtectedData.Protect(ACifrar, entropia,
 DataProtectionScope.CurrentUser);
// Escribir datos cifrados al stream.
```

¹⁴ En Física, la Entropía mide el nivel de desorden de un sistema

```
if (fStream.CanWrite && DatosCifrados != null)
{
    fStream.Write(DatosCifrados, 0, DatosCifrados.Length);
    fStream.Close();
}
```

La única diferencia estriba en que llamamos a la clase *ProtectedData*, que no tiene limitaciones en la longitud de la matriz de bytes a cifrar, y que permite ese segundo parámetro para indicar el grado de desorden. Respecto a la clase *RNGCryptoServiceProvider*, solo necesitamos la operación que realiza sobre el *array* que se pasa como parámetro al constructor, por lo que no es necesario recoger el valor devuelto.

Almacenamiento Aislado (*Isolated Storage*)

Finalmente, disponemos de la posibilidad de vincular un conjunto de información almacenada en un fichero a un usuario concreto, pudiendo, de esa forma, evitar que otras aplicaciones lleguen a ver, modificar o borrar su contenido. A esa técnica, -presente desde la primera versión de .NET Framework- se le denomina Almacenamiento Aislado (AA), y se basa en varios conceptos.

Por una parte, los datos se aíslan por usuario y ensamblado (aunque también puede hacerse por Dominio de Aplicación). La identidad del ensamblado se determina por su nombre seguro o sus credenciales. Los administradores del sistema tienen control sobre la cantidad de espacio en disco utilizada por el AA, que -en sistemas Windows XP y Windows 2003, estará asociada con los directorios *Documents and Settings\<usuario>\Application Data*, y *Documents and Settings\<usuario>\Local Settings\Application Data*-. Además, para poder crear una zona de AA, el código debe de disponer del permiso correspondiente, del tipo *IsolatedStorageFilePermission*.

Las 3 clases principales suministradas en .NET para el manejo de AA son *IsolatedStorageFile*, *IsolatedStorageFileStream* y *IsolatedStorageScope*. También se dispone de una utilidad del SDK, *Storeadm.exe*, que permite listar o borrar los almacenamientos asociados a un usuario concreto.

Veamos su funcionamiento a través de un pequeño ejemplo que crea, almacena, recupera y, finalmente, borra información de un AA.

```
IsolatedStorageFile MiAA = IsolatedStorageFile.GetStore(
```

```

        IsolatedStorageScope.User |
IsolatedStorageScope.Assembly, null, null);
//Creamos unos cuantos directorios
MiAA.CreateDirectory("DirectorioPrincipal1");
MiAA.CreateDirectory("DirectorioPrincipal1/Secundario");
// Creamos dos de forma directa.
MiAA.CreateDirectory("DirectorioPrincipal2/Interno");
// Creamos un par de ficheros en los directorios así creados
IsolatedStorageFileStream AASstream1 = new IsolatedStorageFileStream(
        "FicheroEnRaiz.txt", FileMode.Create, MiAA);
//Como todo stream, debe de cerrarse después de usarlo
AASstream1.Close();

// Creamos otro en el directorio interno.
IsolatedStorageFileStream AASstream2 = new IsolatedStorageFileStream(
        "DirectorioPrincipal2/Interno/FicheroEnInterno.txt",
FileMode.Create, MiAA);
AASstream2.Close();

```

Donde, como vemos puede crearse una estructura de directorios aislados tan compleja como se desee. De forma similar, podemos enumerar los ficheros contenidos en el AA, con un código similar al siguiente:

```

IEnumerator Enumerador =
IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.User);
while (Enumerador.MoveNext())
{
    IsolatedStorageFile store = (IsolatedStorageFile)Enumerador.Current;
    string[] Ficheros = store.GetFilesNames("*");
    foreach(string s in Ficheros)
    {
        MessageBox.Show(s);
    }
}

```

Donde, como vemos, nuevamente es la clase IsolatedStorageFile la encargada de casi toda la labor.

Criptografía

Para comprender, siquiera de forma básica las técnicas utilizadas en la protección de datos en la Informática de hoy en día es imprescindible revisar, aunque sea muy básicamente, los conceptos en los que se basan todas ellas, derivados de la ciencia criptográfica.

La criptografía es el arte y la ciencia de transformar la información para asegurar su secreto, su autenticidad o ambos, y prevenir acciones no autorizadas o ilegales sobre la información, los recursos de red y los servicios, es decir, es la encargada del diseño de procedimientos, controlados por una clave, para cifrar una determinada información de carácter confidencial.

A través de la criptografía la información puede ser protegida contra el acceso no autorizado, la fusión, la modificación y la inserción (Como hemos visto en el capítulo dedicado a la protección de la información, todas las técnicas utilizan mecanismos criptográficos). También puede ser usada para prevenir el acceso y uso no autorizado de los recursos de la red y para prevenir a los usuarios contra la negación de servicios a los que deben tener acceso.

Así mismo, es la metodología para proveer la seguridad de la red, y, por ello, trata con todos los aspectos de la seguridad en un ambiente seguro de red, incluyendo la identificación de entidades y autenticación, el control de acceso a los recursos, la confidencialidad del mensaje, la integridad del mensaje y el no repudio. (Como verá el lector, la criptografía puede intervenir en la defensa contra la mayoría de las técnicas de ataque estudiadas aquí).

Objetivos de la Criptografía Informática

Igualmente, es la base de la mayor parte de los mecanismos asociados con la seguridad informática, y se apoya en la implantación de un conjunto de estándares de la industria, -a su vez- fundamentados en algoritmos matemáticos de sobrada reputación.

La criptografía intenta garantizar:

- Discreción: solamente los usuarios autorizados tienen acceso a la información.

- Integridad de la información: garantía ofrecida a los usuarios de que la información original, no será alterada, ni intencional ni accidentalmente.
- Autenticación de usuario: es un proceso que permite al sistema verificar si el usuario que pretende acceder o hacer uso del sistema es quien dice ser.
- Autenticación de remitente: es el proceso que permite a un usuario certificar que el mensaje recibido fue de hecho enviado por el remitente.
- Autenticación del destinatario: es el proceso que permite garantizar que la información enviada solo podrá ser leída por el destinatario.
- Autenticación de actualidad: consiste en probar que el mensaje es actual, y que no se trata de un mensaje antiguo reenviado.

En la criptografía, los mensajes originales se conocen como texto en claro o texto fuente y mediante la codificación, se obtiene un mensaje cifrado que se conoce como texto cifrado o criptograma, que puede ser almacenado o transmitido por un canal público, y a este conjunto de elementos se le denomina sistema criptográfico. Mantener en secreto la información es la meta histórica de la criptografía y ésta se alinea con las metas de la seguridad informática: la confidencialidad y la integridad. La autenticidad implica que el receptor de un mensaje debe ser capaz de determinar:

- a) Autenticidad del remitente
- b) Integridad del contenido

La seguridad de un sistema criptográfico no se basa únicamente en los algoritmos de cifrado, sino también en la clave utilizada. Por ello, las técnicas de cifrado deben tener las siguientes propiedades:

- Para los usuarios autorizados, debe ser relativamente sencillo cifrar y descifrar el mensaje.
- El esquema de cifrado no depende de mantener en secreto el algoritmo, sino de un parámetro del algoritmo llamado clave de cifrado.
- Para un intruso (criptoanalista) debe ser muy difícil determinar cuál es la clave de cifrado.

Existen multitud de sistemas criptográficos, pero puede agrupárseles por el objetivo que persiguen y la técnica que utilizan para conseguirlo. Desde el punto de vista de las claves que utilizan cabe su división en dos clases: los simétricos y los asimétricos.

Desde el punto de vista de los objetivos que persiguen podríamos establecer el siguiente cuadro explicativo

- Creación de claves *hash* (revisada en el capítulo anterior)
- Criptografía de algoritmo simétrico
- Criptografía de algoritmo asimétrico o de doble clave
- Criptografía de generación de certificados digitales
- Criptografía de firma digital en documentos XML

Soporte criptográfico en .NET Framework

Como ya hemos comentado, el soporte criptográfico en .NET es amplio, y podemos agrupar y relacionar la gran cantidad de clases disponible en *System.Security.Cryptography*, mediante el siguiente cuadro.

Esto no significa que todas las clases criptográficas que manejamos en .NET sean administradas. Algunas de ellas, son simplemente un envoltorio (*wrapper*) del API no administrado *CryptoAPI*, que incluye el propio sistema operativo. En general, las clases de soporte del API nativo contienen en su nombre el sufijo *ServiceProvider*, (como, por ejemplo *SHA1CryptoServiceProvider*, o *MD5CryptoServiceProvider*, utilizada en los ejemplos de *hash*) mientras que las que se encuentran hechas 100% en .NET, contienen el sufijo *managed*, como *RijndaelManaged* o *SHA512Managed*.

Componentes de Criptografía en .NET Framework

Propósito	Descripción
Algoritmos de cifrado	Conjunto de clases para implementar, tanto los algoritmos simétricos, como los asimétricos
Clases de apoyo	Clases utilizadas para generar números aleatorios, realizar conversiones, interactuar con el almacén <i>CryptoAPI</i> y en la actualidad para realizar cifrados basados en modelo de corrientes de información.
Certificados X.509	Clases definidas en el espacio de nombres <i>System.Security.Cryptography.X509Certificates</i> usada para representar certificados digitales

Firmas digitales XML	Clases definidas en el espacio de nombres <i>System.Cryptography.Xml</i> usadas para representar firmas digitales en documentos XML
----------------------	---

Criptografía simétrica o de clave secreta

Los métodos simétricos son aquéllos en los que la clave de cifrado es la misma que la clave de descifrado. Para ello, es necesario que la clave únicamente sea conocida por el emisor y el receptor. Dado que la misma clave es usada para cifrar y descifrar el mensaje, a este método de criptografía se le llamó “secreto compartido”, y el término más formal para este método es la criptografía simétrica. Un esquema de cifrado convencional cuenta con cinco elementos principales:

1. Texto en claro: es el mensaje original o los datos que son alimentados como entrada al algoritmo.
2. Algoritmo de cifrado: el algoritmo ejecuta varias sustituciones y transformaciones sobre el texto en claro.
3. Clave secreta: es también una entrada al algoritmo. Las sustituciones y transformaciones exactas dependen de la clave.
4. Texto cifrado o criptograma: es el mensaje que se produce como salida. Depende del texto y de la clave secreta. Para un mensaje dado, dos claves diferentes producen dos textos cifrados diferentes.
5. Algoritmo de descifrado: se trata del algoritmo de cifrado ejecutado en sentido inverso. A partir del texto cifrado y la misma clave secreta se reproduce el texto original.

Existen dos requisitos para el uso seguro del cifrado convencional:

- a) Un algoritmo fuerte de cifrado. Como mínimo, debe ser tal que, un oponente que lo conozca y que tenga acceso a uno o más textos cifrados sea incapaz de descifrar el texto o encontrar la clave.
- b) La clave secreta debe de compartirse entre el emisor y el receptor.

Es importante notar que la seguridad del cifrado convencional depende de lo secreta que sea la clave y no de qué tan secreto sea el algoritmo, es decir, el algoritmo no debe mantenerse en secreto, pero la clave sí pues no se puede descifrar un texto sólo partiendo

de la base de que se conoce el algoritmo. La clave utilizada en el cifrado convencional es llamada clave secreta.

Algoritmos de clave simétrica

Existen 4 algoritmos populares de clave simétrica: **DES**, **Triple DES**, **RC2** y **Rijndael**, con diferentes implementaciones y longitudes de clave. Todos ellos están representados por las siguientes clases en .NET:

Algoritmo	Tamaño de clave válida	Tamaño de clave predeterminada	Clase de Implementación
DES	64	64	DESCryptoServiceProvider
TripleDES	128, 192	192	TripleDESCryptoServiceProvider
RC2	40-128	128	RC2CryptoServiceProvider
Rijndael	128, 192, 256	256	RijndaelManaged

Tabla xx: Algoritmos simétricos y su implementación en .NET

La Norma de Cifrado de Datos (*DES – Data Encryption Standard*) es un algoritmo para cifrar desarrollado por IBM, e implantado el año 1977, por entidades oficiales norteamericanas. El DES es un algoritmo de cifrado de bloque, donde la longitud de bloque es de 64 bits y la longitud de la clave es de 56 bits. Si el texto es más grande entonces se procesa en bloques de 64 bits. La propia norma fue pensada para su utilización informática por lo que requiere obligatoriamente que DES se implemente por medios digitales.

El algoritmo TripleDES se hizo como un avance respecto a DES para proporcionar una longitud mayor de clave, RC2 está en desuso debido a su debilidad ante ataques de fuerza bruta, y por último, el algoritmo Rijndael (también llamado AES: *Advanced Encryption Standard*), ha sido adoptado por un sinnúmero de entidades y ofrece una buena resistencia a los ataques tradicionales, especialmente en su versión de 256 bits de longitud de clave.

El mismo espacio de nombres, soporta este algoritmo mediante las clases mencionadas en la tabla anterior. Todas estas clases, derivan de la clase abstracta *SymmetricAlgorithm*, que no dispone de constructor, pero sí de un método *static* (*Create()*), que permite crear una clase representativa del algoritmo deseado pasándole el identificador del algoritmo.

El gráfico adjunto, representa el modelo de clases de soporte en .NET para la criptografía simétrica (tomado de MSDN:

http://www.microsoft.com/spanish/msdn/arquitectura/BuildSecNetApps/html/Appendix_04.asp)

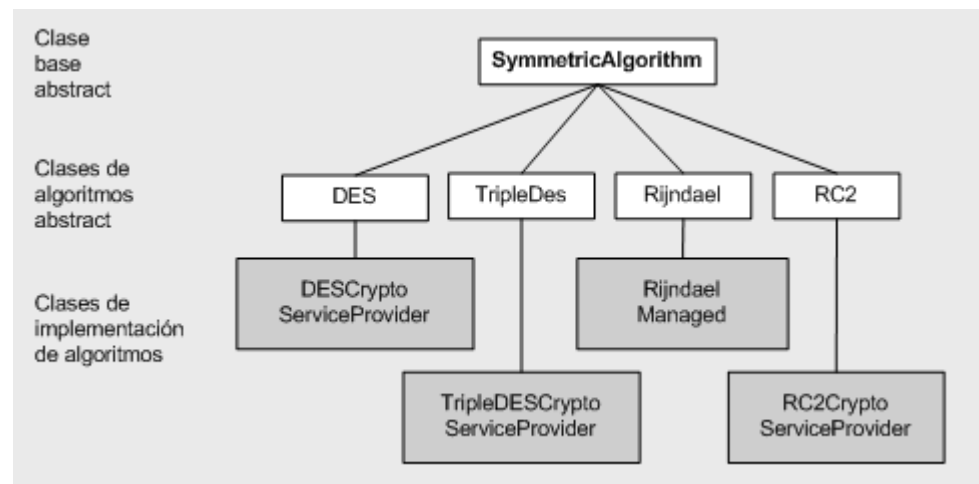


Fig. xx: Arbol de clases para criptografía simétrica en .NET

Donde podemos apreciar que la única implementada directamente en código administrado, es Rijndael, siendo el resto *wrappers* del soporte CryptoAPI del sistema operativo.

Implementación básica de un cifrado simétrico

Por ejemplo, para crear una clase de tipo DES en .NET podríamos hacer lo siguiente:

```
DES algDES = SymmetricAlgorithm.Create("DES");
ICryptoTransform Transf = algDES.CreateEncryptor();
```

A partir de aquí, depende de la labor criptográfica que queramos realizar. En realidad, depende del tipo de *Stream* que necesitemos como mecanismo almacenamiento (*FileStream*, *MemoryStream* o *NetworkStream*) y del algoritmo a usar. Imaginemos que lo que se desea es crear un fichero físico cifrado. El constructor de la clase *CryptoStream*, admite como argumentos un objeto de tipo *Stream*, un objeto que implemente la interfaz *ICryptoTransform* y un modo de acceso.

Continuando con el código anterior, sólo tendríamos que añadir la sentencia:

```
DES algDES = DES.Create();
ICryptoTransform Transf = algDES.CreateEncryptor();
FileStream f = new FileStream("Fichero.dat", FileMode.OpenOrCreate);
CryptoStream CS = new CryptoStream( f, Transf,
    CryptoStreamMode.Write);
```

```
//Como todos los algoritmos de cifrado reciben una matriz de bytes
//como entrada necesitamos convertir la cadena.
byte[] bytesACifrar = System.Text.ASCIIEncoding.ASCII.GetBytes(
    "Texto inicial cifrado mediante algoritmo DES");
CS.Write(bytesACifrar,0,25);
CS.Close();
```

Lo que produce una salida binaria pura y cifrada, no legible mediante los editores hexadecimales tal y como podemos ver en la figura adjunta:

Fig.: Salida cifrada del algoritmo anterior.

Para el proceso de descifrado, podemos operar de la misma forma, pero esta vez creando un objeto específico para descifrar ese algoritmo, llamando al método *CreateDecryptor*:

```
ICryptoTransform Descifrador = algDES.CreateDecryptor();
```

Siguiendo el resto de las operaciones un patrón muy similar al expuesto.

De cualquier forma, existen varias maneras de instanciar clases criptográficas dependiendo de la aproximación que utilicemos. El siguiente código crea 4 clases Rijndael de 4 formas distintas:

```
RijndaelManaged r, r1, r2, r3;
// Creación calificada (por nombre)
r3 = (RijndaelManaged) CryptoConfig.CreateFromName("Rijndael");
// Instancia por defecto de algoritmo simétrico
r2 = (RijndaelManaged) SymmetricAlgorithm.Create();
// por llamada al método estático de creación
r1 = (RijndaelManaged) Rijndael.Create();
// por llamada al constructor
r = new RijndaelManaged();
```

donde hay que recalcar que, en el primer caso *CryptoConfig*, admite la creación a partir del nombre del algoritmo, en el segundo la creación tiene lugar debido a que la llamada a *SymmetricAlgorithm.Create()*, genera una clase *Rijndael*, por ser ésta la predeterminada de los 4 algoritmos simétricos, y sólo en los dos últimos casos utilizamos una referencia directa al algoritmo en la creación.

Criptografía asimétrica de doble clave (o clave pública)

Para comprender mejor la forma en que funciona la criptografía asimétrica, vamos a operar de forma totalmente distinta de lo que hemos hecho hasta ahora. Sin usar más matemáticas que la "cuenta de la vieja" vamos a poner un ejemplo reduciendo al máximo la complejidad, de manera que podamos seguir el proceso de cifrado/descifrado de una sola palabra, operando sobre un sistema también reducido, de manera que un sencillo ejercicio mental de sumas y restas nos explique todo el proceso.

A partir de ahí, comprenderá el lector el porqué del funcionamiento actual de este modelo, y podremos profundizar algo más en su implementación en .NET y en los sistemas actuales.

Supongamos que utilizamos un alfabeto castellano de 27 letras (seguimos la norma DRAE de 1994) para escribir nuestros mensajes y damos a cada una un valor numérico asociado con su orden alfabético ($a=1$, $b=2$, $c=3$, $d=4$, etc.). Vamos a cifrar el mensaje "hola" (usamos sólo minúsculas a efectos de simplificación).

A continuación escogemos como clave pública (o privada, da lo mismo), una cualquiera de las letras del alfabeto, por ejemplo la *c*, que este sistema tiene el valor 3. Si el cifrado se realiza mediante la operación de suma, el proceso de cifrado completo, queda descrito en la tabla adjunta:

Letras (Texto)	h	o	l	a
Valor numérico	8	16	12	1
Clave Privada	3	3	3	3
Valor cifrado (suma)	11	19	15	4

Para descifrarlo, añadimos la clave pública, que será $+24$ ($27-3$). Si el resultado sobrepasa el límite de rango (27), restamos o añadimos esta cantidad hasta que resulte válida. O, por decirlo de otro modo, para descifrar el mensaje, añadimos la clave pública y tomamos el resultado del módulo de 27, y volvemos a decodificar los valores numéricos para obtener el texto en claro:

Valor cifrado	11	19	15	4
---------------	----	----	----	---

Clave Pública	24	24	24	24
1ª fase de descifrado	35	43	39	28
Valor normalizado	8	16	12	1
Texto en claro	h	o	l	a

¿Y cómo romperíamos esta clave de cifrado? Tendríamos que ir probando una a una, hasta que en una de las 27 combinaciones resultantes obtuviésemos un mensaje inteligible.

Algoritmos criptográficos asimétricos

Los algoritmos criptográficos asimétricos funcionan según este principio, pero, para impedir que sea tan sencillo el mecanismo de reconstrucción del mensaje original, en lugar de la operación de suma se utiliza la exponenciación y los números son de cientos (o miles) de dígitos de longitud. Los grados de complejidad resultantes son de una magnitud astronómica, que superan la capacidad computacional de un ordenador de hoy (aunque, como ya comentamos en el apartado de la creación de valores *hash* ya ha habido un caso de reconstrucción del algoritmo SHA1, utilizando computación distribuida).

Los algoritmos criptográficos asimétricos, denominados también algoritmos de clave pública, requieren que tanto el remitente como el receptor mantengan un par de claves relacionadas: se utilizan dos claves: una pública y una privada, que se encuentran relacionadas matemáticamente. Para diferenciarlo del cifrado de claves simétricas, en ocasiones el cifrado de claves públicas también se denomina cifrado de claves asimétricas.

De esta forma, la clave pública puede intercambiarse libremente entre las partes o incluso publicarse sin restricciones. Sin embargo, la clave privada seguirá siendo privada. Los datos cifrados con la clave pública sólo pueden descifrarse con la clave privada. Los datos cifrados con la clave privada sólo pueden descifrarse con la clave pública.

En la figura adjunta, el remitente tiene la clave pública del destinatario y la utiliza para cifrar un mensaje, pero sólo el destinatario tiene la clave privada relacionada que se utiliza para descifrar el mensaje.



Fig.: El usuario de la izquierda dispone de ambas claves y cifra un mensaje con la clave privada. El de la derecha sólo necesita la clave pública para descifrar.

El quid de la cuestión está en que el mecanismo de cifrado/descifrado requiere de ambas claves **como una unidad indivisible**, sin la cuál no es posible el proceso. Esto significa que si el emisor sospecha que alguien ha accedido a la clave privada de uno de sus receptores, basta con que cree un nuevo par público/privado y lo distribuya adecuadamente para restaurar la integridad de la comunicación.

Esto permite una flexibilidad enorme, ya que la clave pública puede ser enviada a listas de correo de aquellos que queremos que nos manden mensajes cifrados, o incluso publicarla en Internet, sin miedo a que ningún tercero pueda interferir.

Microsoft Technet

(<http://www.microsoft.com/spain/technet/recursos/articulos/2006.a.spx>), nos recuerda los pasos involucrados en el intercambio de claves basado en claves públicas:

- El remitente obtiene la clave pública del destinatario
- El remitente crea una clave secreta aleatoria (la única clave utilizada en el cifrado de claves simétricas). En Windows, se utiliza CryptoAPI para crear la clave secreta. El remitente utiliza la clave secreta con un algoritmo simétrico para convertir el texto sin formato en texto cifrado
- El remitente utiliza la clave pública del destinatario para transformar la clave secreta en una clave secreta de texto cifrado
- El remitente envía al destinatario el texto cifrado y la clave secreta de texto cifrado

- Con su clave privada, el destinatario convierte la clave secreta cifrada en clave de texto sin formato
- Finalmente, con la clave secreta de texto sin formato, el destinatario convierte el texto cifrado en texto en claro. De nuevo, estos pasos los realizan aplicaciones compatibles con PKI como Microsoft Outlook y son transparentes para el usuario.

La única pega de esta tecnología es que resulta relativamente lenta respecto de sus correspondientes simétricos, por lo que, la mayoría de las veces, se utiliza una mezcla adecuada de ambos sistemas combinando ambos en un sistema de firmas digitales (comentaremos algo más a este respecto a continuación).

Conceptos de infraestructura de claves públicas

El término infraestructura de claves públicas (PKI) se utiliza para describir las directivas, los estándares y el software que regulan o manipulan los certificados, y las claves públicas y privadas. En la práctica, PKI hace referencia a un sistema de certificados digitales, entidades emisoras de certificados (CA) y otras entidades de registro que comprueban y autentican la validez de cada parte implicada en una transacción electrónica. Los estándares de la infraestructura de claves públicas siguen evolucionando, aunque se estén implementando de forma generalizada como elemento necesario del comercio electrónico.

En Windows XP y Windows 2003, el soporte de PKI se ha extendido notablemente. En la primera plataforma, con mejoras en los servicios de Directorio Activo, Servicios de Certificados, Servicios de Administración de Intercambio de Claves, y el soporte de aplicaciones como EFS, Microsoft Internet Explorer, Microsoft Money, Internet Information Server, servicio de acceso remoto, Microsoft Outlook, y Microsoft Outlook Express. En la segunda, además de las anteriores mejoras, con otras propias de un servidor seguro avanzado, como mejoras en el transporte SSL, en la creación y gestión de políticas de seguridad, en archivado y recuperación de claves, en las API's de cifrado y muchos otros.

Para más datos sobre el soporte detallado de PKI en ambos sistemas, remitimos al lector a la dirección: <http://www.microsoft.com/technet/prodtechnol/winxppro/plan/pkie.nh.mspx>, que contiene además varios enlaces adicionales de soporte.

Algoritmos criptográficos y su soporte en .NET

Algunos de los algoritmos criptográficos más populares son: RSA, DSA, Diffie-Hellman, MD5, SHA-1, SHA-256, SHA-384 y SHA-512. Pero no todos pueden utilizarse en doble sentido y valen como mecanismos de clave pública/privada: sólo RSA (acróstico de las siglas de sus autores, Rivest-Shamir-Adleman), DSA (para firmas digitales) y Diffie-Hellman (para intercambio de claves), se pueden utilizar en infraestructura de clave pública, siendo el RSA el más popularmente implementado, debido a que soporta en un solo algoritmo, las características que ofrecen individualmente DSA y Diffie-Hellman, esto es, firma digital e intercambio de claves.

Todos estos algoritmos están soportados de diferentes maneras en el espacio de nombres que estamos utilizando. RSA y DSA están implementados por clases que derivan de la clase abstracta *AsymmetricAlgorithm*. Debido a su complejidad, existen muchas clases de soporte que complementan la funcionalidad de esta clase base, como, por ejemplo, *AsymmetricKeyExchangeFormatter* y *AsymmetricSignatureFormatter*. Diffie-Hellman, lo está mediante una clase tipo "wrapper", que establece un puente con CryptoAPI para su utilización. El resto, suele utilizarse en operaciones de cálculo de valores *hash*, y heredan de la clase *HashAlgorithm*. El gráfico MSDN (http://www.microsoft.com/spanish/msdn/arquitectura/BuildSecNetApps/html/Appendix_04.asp) explica perfectamente esta estructura de clases:

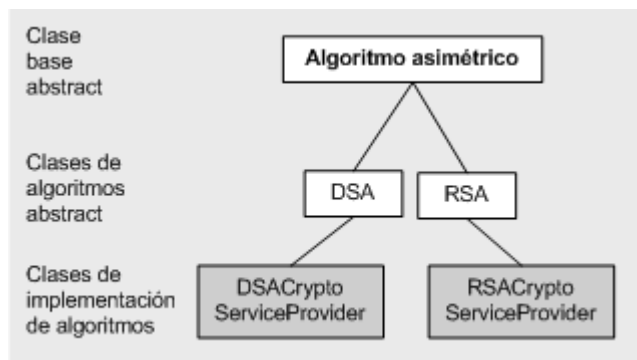


Fig.: Clases de soporte de criptografía asimétrica en .NET

Veamos un ejemplo de uso del algoritmo RSA para el cifrado/descifrado de una cadena, utilizando la clase *RSACryptoServiceProvider*, que hereda de *AsymmetricAlgorithm*. Se requiere el espacio de nombres *System.Text*, además de los propios de servicios criptográficos.

```
// define el texto claro
byte[] TextoClaro= Encoding.Default.GetBytes("Texto sin cifrar");
// crea una instancia de RSA
RSACryptoServiceProvider algRSA = new RSACryptoServiceProvider();
// cifrado usando relleno de tipo OAEP
byte[] TextoCifrado = algRSA.Encrypt(TextoClaro, true);
// descifra el texto codificado
byte[] NuevoTextoClaro = algRSA.Decrypt(TextoCifrado, true);
// reconstruimos la cadena texto claro en otra variable
// e imprimimos a consola.
string Salida = Encoding.Default.GetString(NuevoTextoClaro);
Console.WriteLine(Salida);
```

Al igual que en ocasiones anteriores trabajamos convirtiendo el texto en una matriz de bytes, para usarlo como entrada de los métodos *Encrypt* y *Decrypt* de la clase *algRSA* que implementa el algoritmo RSA. El segundo argumento de *Encrypt*, utiliza una característica de *padding* (relleno de caracteres) propia únicamente de sistemas operativos Windows XP y posteriores.

Lógicamente, a mayor longitud del texto, mayor es el tiempo que tarda el algoritmo de cifrado. El lector deberá de evaluar las técnicas más convenientes a cada escenario y cada solución.

Certificados Digitales

Estos algoritmos son frecuentemente utilizados para más propósitos, como sucede por ejemplo con los llamados certificados digitales. Un certificado digital, es una declaración firmada digitalmente que enlaza el valor de una clave pública con la identidad de una persona, dispositivo o servicio que posee la clave privada correspondiente.

Al igual que sucede con los algoritmos citados hasta aquí, existen estándares de creación y emisión de certificados. La mayor parte de los certificados de uso común se basan en el estándar de certificados **X.509 v3**. Se trata del formato de certificados estándar que utilizan en Windows los procesos basados en certificados.

Un certificado X.509 contiene la clave pública e información acerca de la persona o entidad para la que se emite el certificado, información acerca del certificado, además de información opcional acerca de la entidad emisora (CA) del mismo. Los certificados se pueden emitir con varios fines, como la autenticación de usuarios en la Web, autenticación de servidores Web, correo electrónico seguro (mediante Extensiones seguras multipropósito de correo Internet llamadas (S/MIME)), Seguridad de Protocolo Internet (IPSec), Seguridad en el nivel de transporte (TLS) y firma de código.

Podríamos asumirlo como una especie de carnet de identidad de carácter digital, hecho de tal forma que no puede repetirse bajo ninguna circunstancia, y –mediante un contrato entre la entidad emisora y el sujeto receptor– permite garantizar la identidad del poseedor del certificado. Además, los certificados tienen una caducidad, por lo que deben de ser renovados en el tiempo.

Normalmente, los certificados contienen la información siguiente:

- Valor de la clave pública del sujeto
- Información del identificador del sujeto, como el nombre y la dirección de correo electrónico.
- Período de validez (el intervalo de tiempo durante el que el certificado se considera válido).
- Información del identificador del emisor.
- La firma digital del emisor, que da fe de la validez del enlace entre la clave pública del sujeto y la información del identificador de sujeto.

En el modelo asimétrico que hemos estudiado, el servidor se ve requerido a mantener información de claves de los usuarios con los que se quiere establecer una comunicación segura. Ahora, imaginemos que pudiéramos establecer –simplemente– que el servidor considere todos los certificados emitidos por una cierta entidad, como de confianza. Si así lo hacemos, el servidor sólo tendrá que definir la Entidad de Emisión como entidad de confianza y comprobar el valor del emisor del certificado para saber que el usuario está autorizado en el sistema. La entidad emisora, hace las veces de notario digital, que garantiza la identidad del portador del certificado.

Pero, ¿un certificado debe vincularse siempre con una persona? No, realmente. Además de una persona física, también puede ser un servicio (como IPSec) o un equipo informático. Internet Explorer, como la mayor parte de los exploradores y servidores Web actuales, utiliza el nivel de sockets seguros (SSL) para cifrar mensajes y transmitirlos de forma segura a través de Internet.

El Centro de soporte técnico de Microsoft, define en estos pocos pasos la utilización de un certificado en una comunicación por canal seguro que tiene por objeto una transacción bancaria:

- *Se inicia la comunicación con una entidad bancaria. El servidor Web del banco envía un certificado de servidor a su explorador Web.*
- *Para autenticar el servidor, el explorador comprueba el almacén de certificados de equipo local. Si la entidad emisora de certificados que emitió el certificado a su banco es de confianza, la transacción*

puede proseguir, y el certificado del banco queda almacenado en su almacén de certificados.

- *Para cifrar todas las comunicaciones con el servidor del banco, el explorador Web crea una clave de sesión exclusiva. El explorador cifra la clave de sesión con el certificado del servidor Web del banco para que sólo el servidor Web del banco pueda leer los mensajes que envía desde su explorador. De esta forma se establece un código único y seguro de comunicación entre ambos. (Algunos de estos mensajes contendrán su nombre y contraseña de inicio de sesión, y más información sensible, de modo que este nivel de seguridad es necesario).*
- *Se establece la sesión segura, y se puede enviar información sensible entre su explorador Web y el servidor Web del banco de forma segura.*

Software y Almacenes de certificados

En general, cualquier software que quiera garantizarse la instalación sin problemas en un equipo cualquiera puede ir acompañado de un certificado digital. Si el certificado está intacto, el sistema sabe que el software que se está copiando al equipo es de confianza. Por ejemplo los drivers de los controladores de hardware del sistema que han pasado por la lista de compatibilidad establecida por el fabricante disponen siempre de un certificado que acredita esta circunstancia. De no ser así, el sistema interroga al usuario para que este –bajo su propio riesgo- instale dicho driver en el equipo. Es importante que todos los drivers instalados cumplan con la norma de compatibilidad de hardware (HCL).

De esta forma, todo equipo cuenta con un Almacén de Certificados, que cataloga a sus emisores, y puede provenir de varios orígenes:

- El certificado se incluye con su instalación de Windows Server 2003 y viene en el CD de Windows Server 2003.
- Utiliza una aplicación como Internet Explorer para entrar en una sesión SSL, durante la cual se almacenan certificados en su equipo después de establecer la confianza.
- Decide aceptar explícitamente un certificado, como cuando instala software o recibe correo electrónico cifrado o firmado digitalmente de otras personas.
- Solicita un certificado a una entidad emisora de certificados, como un certificado necesario para tener acceso a recursos organizativos específicos.

El almacén de certificados es visible directamente a través de las opciones de Contenido, de la configuración de Internet Explorer. Para el usuario que se adentre por primera vez en este tema, pensamos que puede ser de

interés una pequeña guía visual del acceso al almacén de certificados con las opciones disponibles (en este momento desde un equipo con Windows 2003 Server).

Opciones de manejo y visualización de certificados

Podemos ver los certificados instalados en el equipo organizado por categorías, accediendo a las Opciones de Internet Explorer, y en la pestaña Contenidos, seleccionando la opción certificados tal y como se muestra en la figura adjunta:

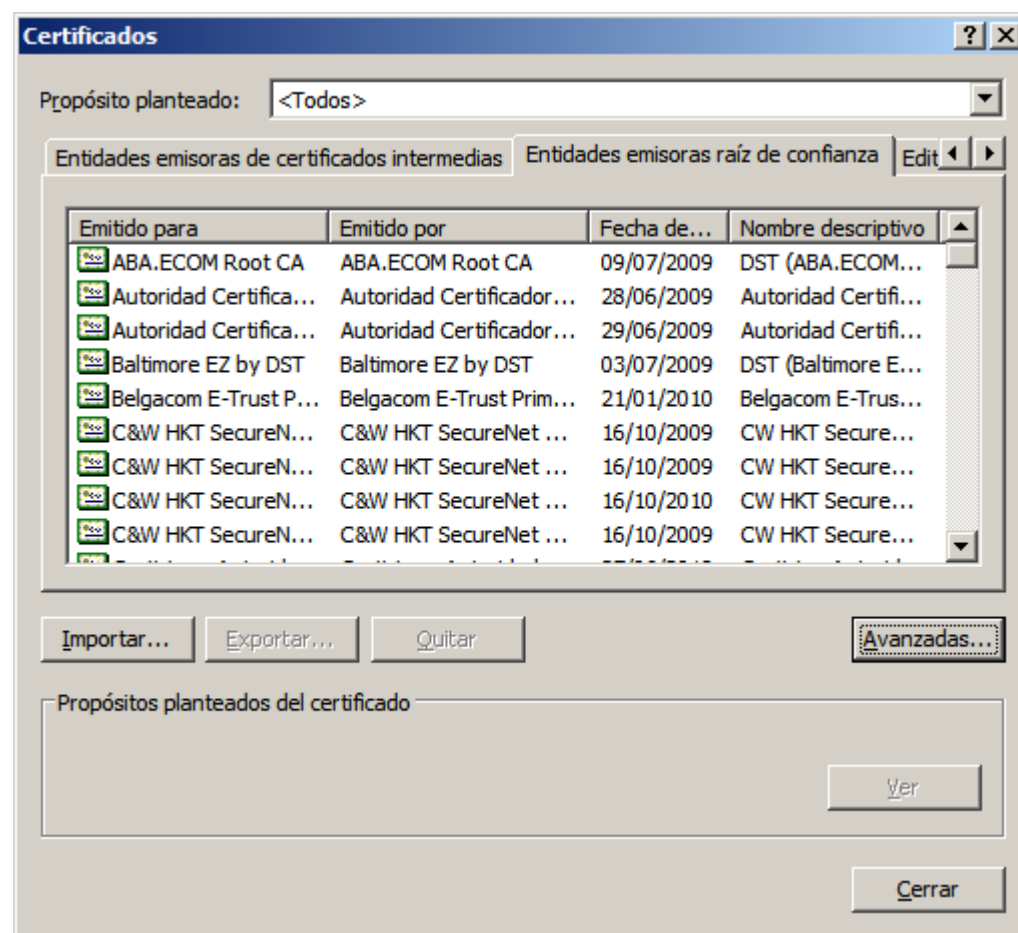


Figura xx: Certificados instalados por entidades de confianza

Como vemos, es posible importar nuevos certificados a partir de una ubicación, o analizar las características de un certificado en concreto, como por ejemplo, el instalado en el sistema por la Fábrica Nacional de Moneda y Timbre:

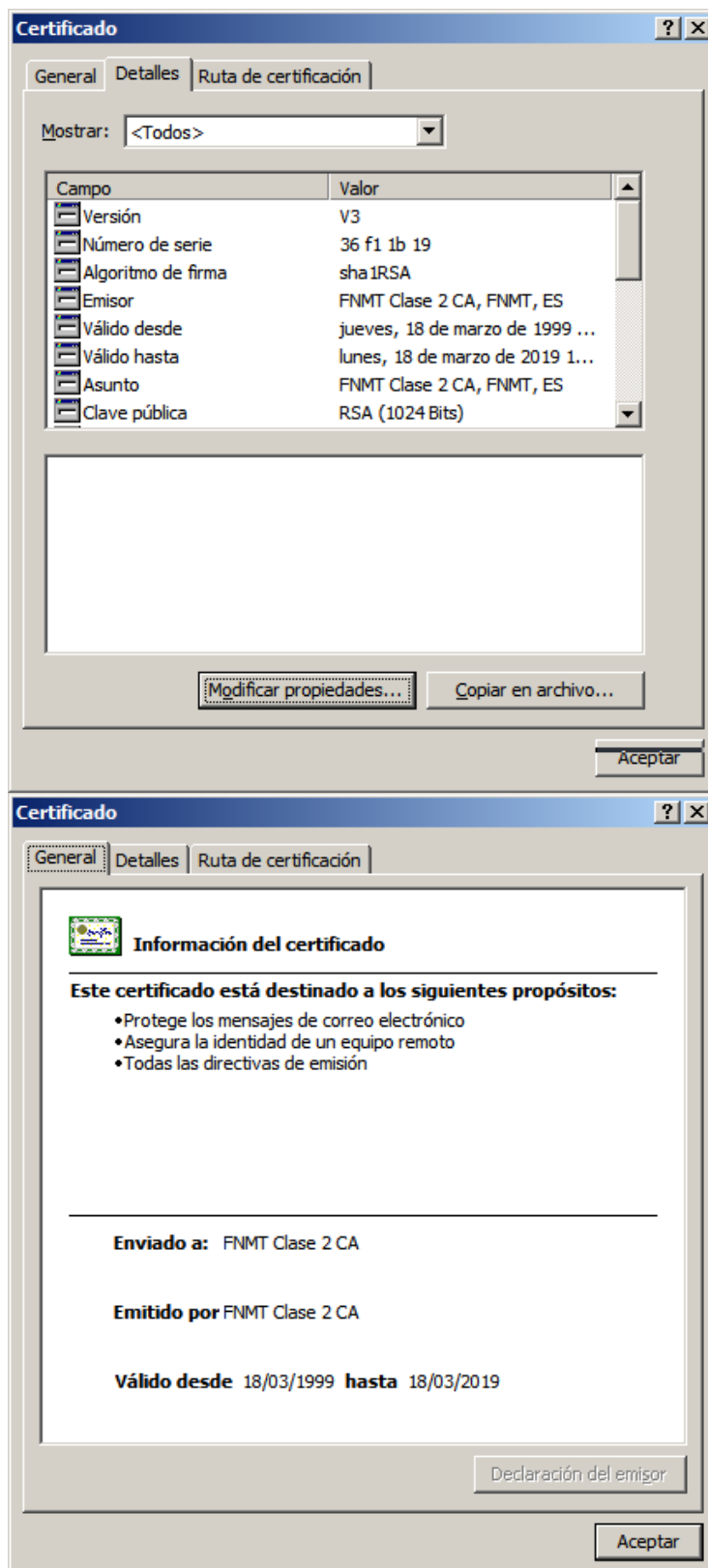


Figura xx: Contenido del certificado de la FNMT

De la misma forma, podemos modificar los propósitos para los que está configurado el certificado, accediendo desde la ventana anterior a la opción "Modificar propiedades", como se muestra a continuación:

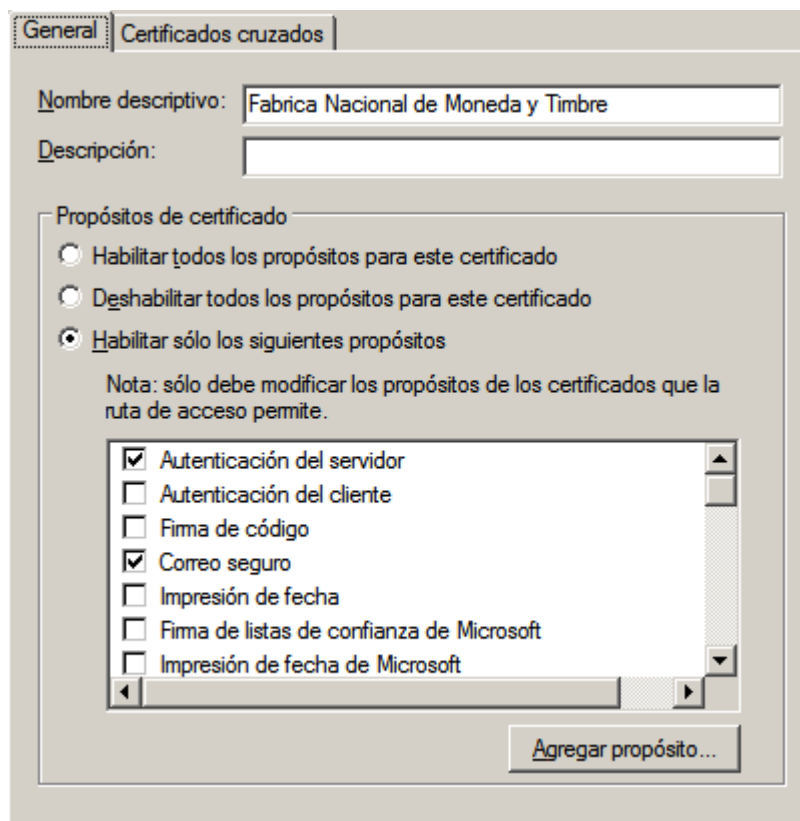


Fig. xx: Opciones de configuración del certificado anterior.

Por último, es posible exportar el certificado a otra ubicación, seleccionando la opción "Copiar en archivo" dentro de la vista de un certificado concreto.

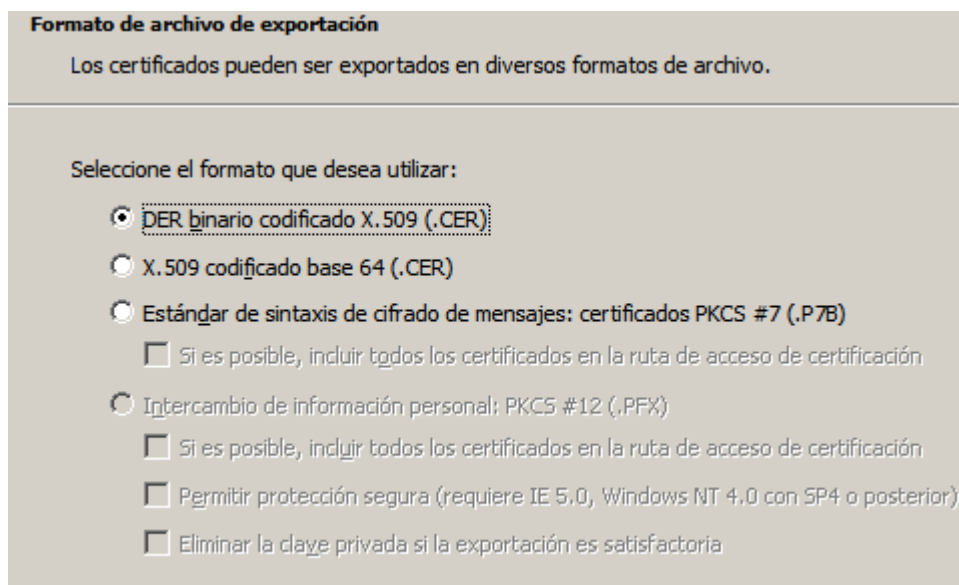


Fig.: Opciones de exportación de certificados

Por último cabe destacar la existencia de una consola de administración en la que se accede exclusivamente a la información relacionada con certificados a través del MMC llamado ***certmgr***.

Manejo de certificados en .NET Framework

En .NET Framework 2.0 (aunque ya estaba disponible también en versiones anteriores, pero no de una forma tan completa como en esta versión), disponemos de un buen conjunto de clases de soporte de certificados desde el espacio de nombres *System.Security.Cryptography.X509Certificates*, que nos permite visualizar, consultar contenidos, modificar, registrar y crear certificados a partir de ficheros de certificado, y –en general– realizar la mayor parte de las operaciones disponibles desde las opciones de administración que acabamos de ver.

Comencemos por la visualización de los almacenes disponibles en una máquina concreta. Cada almacén se corresponde con un valor enumerado del tipo *StoreName*:

```
// Asumimos que lstAlmacenes es un listbox de selección simple
string[] Almacenes = System.Enum.GetNames(typeof(StoreName));
foreach (string sn in Almacenes)
{
    lstAlmacenes.Items.Add(sn);
}

// A continuación, leemos un certificado, y aprovechando una sobrecarga
// del método ToString(), imprimimos algunos de los valores del contenido
string Certificado = "pmd.cer";
// Cargar certificado
X509Certificado cert = new X509Certificate(Certificado);
// Leer el valor
string LeerVerdadero = cert.ToString(true);
// Mostrar resultado.
Console.WriteLine(LeerVerdadero);
// Leer otros valores
string LeerFalso = cert.ToString(false);
// Mostrar resultado.
Console.WriteLine(LeerFalso);
```

La primera parte de la rutina, recupera un vector con todos los valores válidos definidos como almacén de certificados en el tipo enumerado *StoreName*. En la segunda parte de la rutina, aprovechamos la sobrecarga del método *ToString()* de la clase

X509Certificate, para obtener, respectivamente dos representaciones del certificado con distinto nivel de información.

La salida produce el resultado adjunto:

Almacenes disponibles ----- AddressBook AuthRoot CertificateAuthority Disallowed My Root TrustedPeople TrustedPublisher Certificado ----- [Subject] C=U.S.A., O=Adobe Systems Inc., CN=PaperMetaData Plugin [Issuer]	CN=Adobe Systems Inc. [Serial Number] 000001 [Not Before] 13/04/2004 22:24:42 [Not After] 11/04/2014 22:24:42 [Thumbprint] C9DB3C770246B6A4C1802F22 18881A66864EE70E System.Security.Cryptography. X509Certificates.X509Certificat e
--	--

Donde el valor de *Thumbprint* se refiere a la llamada "huella digital" del certificado. También resulta sencillo ver el contenido cifrado, o el de la clave pública, mediante los métodos *GetRawCertData()* y *GetPublicKeyString()*:

```
byte[] contenido = cert.GetRawCertData();

// Mostrar bytes
foreach (byte b in contenido)
{
    Console.Write(b);
}
string ClavePublica = cert.GetPublicKeyString();
// Mostrar clave pública
Console.WriteLine(ClavePublica);
-----
4813051944813051081603212210971322134000001254 ...
-----
30818902818100AAF12227B408EA70D6A67CBCB8B97E7 ...
```

El conjunto de servicios ofrecido por las clases de este espacio de nombres nos permitirá llevar una gestión adecuada de certificados desde cualquier aplicación .NET Framework 2.0.

Firma digital

Una firma digital actúa como una comprobación de integridad de datos y proporciona una prueba de posesión de la clave privada. Se puede utilizar tecnología de claves públicas junto con algoritmos *hash* para crear una firma digital. Los pasos son los siguientes:

- El remitente aplica un algoritmo *hash* a los datos y genera un valor *hash*.
- Con una clave privada, el remitente transforma el valor *hash* en una firma digital.
- A continuación, el remitente envía al destinatario los datos, la firma y el certificado del remitente.
- El destinatario aplica el algoritmo *hash* a los datos recibidos y genera un valor *hash*.
- El destinatario utiliza la clave pública del firmante y el valor *hash* recién generado para comprobar la firma.

Los algoritmos *hash* pueden procesar los datos más deprisa que los algoritmos de claves públicas. La codificación *hash* de datos también reduce el tamaño de los datos que se van a firmar y, por tanto, acelera el proceso de firma. Cuando se crea o se comprueba la firma, el algoritmo de claves públicas tiene que transformar únicamente el valor de *hash* (128 ó 160 bits de datos). Los detalles acerca de la creación y la comprobación de la firma dependen del algoritmo de claves públicas utilizado.

Según se explica en la propia página de **Microsoft Technet** respecto al intercambio de claves, se trata de combinar la clave simétrica con la clave pública, en forma rápida y de fácil utilización comercial:

“Los algoritmos de claves simétricas son excelentes para cifrar datos de manera rápida y segura. Sin embargo, su punto débil reside en que el remitente y el destinatario deben intercambiar una clave secreta antes de intercambiar datos. La combinación de algoritmos simétricos para cifrar datos con algoritmos de claves

públicas con el fin de intercambiar la clave secreta resulta ser una solución rápida y escalable.

Como ya se ha comentado más arriba, .NET permite el manejo de firmas digitales a través de las clases de soporte de algoritmos asimétricos *DSACryptoServiceProvider*, y *RSACryptoServiceProvider*.

Creación de una firma digital mediante código

La generación de una firma digital, normalmente acompaña a un mensaje de mucha mayor extensión, del que conocemos su valor *hash*. Para generar una firma digital desde .NET podemos usar las clases *RSACryptoServiceProvider*, junto a *RSAPKCS1SignatureFormatter*, que es la que –realmente- genera la firma) mediante una llamada al método *GenerateSignature()*. Para ello, el constructor recibe una instancia

Veamos un sencillo código de generación de una firma digital a partir de un valor *hash* dado:

```
//Valor hash que queremos firmar
byte[] ValorHash = { 59, 4, 248, 102, 77, 97, 142, 201, 210,
    12, 224, 93, 25, 41, 100, 197, 213, 134, 130, 135 };

//Resultado del proceso
byte[] ValorHashFirmado;

// Generamos el par de claves publica/privada, que se almacena
// en RSA
RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();

// Para poder llamar al método CreateSignature de un proveedor
// criptográfico creamos una instancia del formateador adecuado
// pasándole como argumento el proveedor, lo que le transfiere la
// clave privada
RSAPKCS1SignatureFormatter RSAFormatter =
    new RSAPKCS1SignatureFormatter(RSA);

//Seleccionamos el algoritmo a utilizar
RSAFormatter.SetHashAlgorithm("SHA1");

// Con la clave privada y el algoritmo a usar, creamos la firma y la
//mostramos
ValorHashFirmado = RSAFormatter.CreateSignature(ValorHash);
foreach (byte b in ValorHashFirmado)
    Console.Write(b + "-");
```

Lo que produce como salida el valor firmado:

17-123-68-181-234-115-236-53-100-21... etcétera.
--

RSASigner, posee, además, métodos de soporte, como *SetKey()* y *SetHashAlgorithm()*, todos ellos, debidamente documentados en las ayudas y el MSDN "on-line".

El proceso de comprobación de una firma sería muy similar pero, esta vez, utilizaríamos la clase *RSAPKCS1SignatureDeformatter*, que, mediante su método *VerifySignature()*, comprueba la validez de una firma y un *hash* concretos.

Firmado de archivos XML

En ocasiones, el archivo que se desea autenticar tiene el formato XML, por lo que el mecanismo de firmado tiene que ser diferente. La firma de XML es importante cuando se desea comprobar que el XML se origina en cierta fuente (entidades oficiales, organismos gubernamentales, etc.).

El espacio de nombres `System.Security.Cryptography.Xml` contiene clases para admitir la creación y la validación de las firmas digitales XML. Las clases en este espacio de nombres implementan la recomendación del Consorcio World Wide Web, "XML-Signature Syntax and Processing", que se recoge en <http://www.w3.org/TR/xmldsig-core/>. Su implementación y desarrollo, implica un nivel de complejidad muy superior al que pretendemos incluir en este breve manual.

Si el lector está interesado, recomendamos la visita al documento oficial de la entidad W3C, que se encuentra en la dirección: <http://www.w3c.es/Traducciones/es/TR/2002/REC-xmldsig-filter2-20021108/#ref-XML-DSig>.

Algunos tipos de ataques

La implantación de técnicas de defensa, consiste, muchas veces en cúmulo de técnicas combinadas a partir de las que hemos visto hasta ahora. En ocasiones, bastará con la seguridad perimetral, en otras con la seguridad basada en roles, y casi siempre, si queremos hacer un ajuste fino, nos vendrá bien utilizar CAS para garantizar el uso adecuado de recursos.

Vamos a revisar algunas de las recomendaciones de defensa que la propia Microsoft propone cuando se refiere a aplicaciones .NET. Primero, citaremos algunas de las amenazas típicas y a continuación veremos cómo pueden ser resueltos.

Una lista de amenazas

Esta es puede ser una lista (nada exhaustiva) de amenazas típicas:

- 1) Saturación de búfer (*buffer overrun*)
- 2) Secuencias de comandos entre sitios
- 3) Inyección SQL
- 4) Problemas de resolución de nombres canónicos
- 5) Debilidades de criptografía
- 6) Problemas en la manipulación de Unicode
- 7) Ataques de Denegación de Servicio (DoS)

No piense ni por un momento el lector que los tipos de ataque se reducen a estos. La lista podría completarse con muchos otros: Inyección CRLF, Inyección LDAP, Inyección de código SSI (*Server Side Includes*), Inyección XPath, Autenticación Insuficiente, Débil Validación en la Recuperación de Contraseñas, Predicción de Credenciales/Sesión, Autorización Insuficiente, Expiración de Sesión Insuficiente, Fijación de Sesión (se fuerza al ID de Sesión a adoptar un valor determinado), Suplantación de Contenido, Fuga de Información, Localización de Recursos Predecibles, Abuso de Funcionalidad, Anti-automatización Insuficiente (cuando un sitio Web permite a un atacante automatizar un proceso que sólo debe ser llevado a cabo manualmente), Validación de Proceso Insuficiente (produce cuando un sitio Web permite a un atacante evadir o engañar el flujo de control esperado por la aplicación), Comandos de Sistema Operativo (explota sitios Web mediante la ejecución de comandos de sistema operativo a través de la manipulación de las entradas a la aplicación) y un largo etcétera.

Los mencionados en primer lugar, sin embargo, pueden servir de modelo para el resto. Vamos a comentar por encima en qué se basa el funcionamiento de algunos de ellos.

Saturación (o desbordamiento) de búfer

Sucede cuando el tamaño de una variable no es suficiente para almacenar el dato introducido por el usuario, y los datos adicionales van a parar al búfer de memoria contigua, pudiendo sobrescribir información vital. Suelen darse en los lenguajes C/C++ debido al control directo que tienen sobre la memoria.

Hay distintos tipos, según se basen en saturar la pila, el montón, los controladores de excepción (o de interrupciones), los punteros a función, etc.

Defensa: No utilizar funciones que puedan sobrescribir el valor de retorno de la pila, o que no permitan comprobar la longitud del búfer, como *StrCopy* y estar alerta ante los riesgos que plantean funciones como *CopyMemory* o *MultiByteToWideChar* (ambas confían en que el programa pasará correctamente los tamaños de búfer y las direcciones de destino).

La utilización de la opción de compilación /GS también ayudará a impedir las saturaciones colocando un valor aleatorio entre los datos locales y la dirección de retorno, que será comprobado después.

Como el problema, aparece con mucha frecuencia en el tratamiento de cadenas, se recomienda, además, utilizar el fichero de inclusión (o cabecera) *Strsafe.h* que suministra un conjunto de funciones seguras para estos procesos.

En general, compruebe que todos los valores en los que se implican rangos (matrices, longitudes de directorios, etc.) no exceden lo esperado por el programa.

Secuencias de comandos entre sitios (*Cross-site scripting*)

Se trata de una técnica que permite al atacante ejecutar secuencias de comandos no esperadas en el navegador del cliente, con muy variados efectos, que van, desde robar información del equipo del cliente, hasta insertar etiquetas HTML que permiten la ejecución de secuencias más complejas o de objetos, tales como *<applet>*, *<script>*, *<object>*, *<form>* y *<embed>*.

Se trata, además, de un problema universal, presente en todas las plataformas y todos los navegadores, y contra el que no existe solución por el momento.

Inyección SQL

Dentro de las aplicaciones cliente/servidor tradicionales, uno de los principales ataques a la seguridad basados en esta técnica se denomina "SQL-Injection", y anticipábamos algo sobre su funcionamiento en la entrega del mes pasado. El efecto es consecuencia de la negligencia asociada a las validaciones (o de la extrema pericia de los atacantes). No solo no se validan las entradas de datos, sino que asumimos que –como el usuario nunca va a intentar nada prohibido– no es necesario comprobar el estado de las sentencias SQL resultantes de una construcción a base de fragmentos de una sentencia.

Si el usuario tiene conocimientos del funcionamiento del servidor de bases de datos, no es complicado adornar nuestra respuesta con sentencias adicionales que anulen parte de la construcción predeterminada, e incluyan modos de conseguir una elevación de privilegios o cualquier intromisión referenciada en STRIDE), a partir de lo cual, todo es posible. (En este caso las técnicas de ataque pueden ser mixtas, por así decirlo; se utiliza una sobrecarga de búfer para enviar más información de la esperada, pero una de las posibles consecuencias es la ganancia de permisos de administrador por parte del lector de datos).

Lo que puede obtenerse mediante SQL-Injection es muy variado. Por ejemplo, podemos atacar el sistema de ficheros apoyándonos en la sintaxis de comandos múltiple que permite a una petición concatenar más de una solicitud mediante un punto y coma. Pensemos en la inocente sentencia

```
SELECT * FROM CLIENTES WHERE CIUDAD = `
```

El sistema solicitará del usuario una entrada de una ciudad y completará la sentencia añadiendo otra comilla simple de cierre que delimita la constante de cadena. Ahora imagínese que el usuario introduce lo siguiente:

```
Madrid'; EXEC master..dbo.xp_cmdshell 'dir *.*' --
```

con doble guión final (comentario T-SQL), que anula cualquier valor que el programa introdujese a continuación. Además de la salida solicitada, se produciría otra con el listado de ficheros del directorio predeterminado (o del que se quisiese examinar). Dado que el argumento entre comillas simples de **xp_cmdShell** puede ser cualquier sentencia del **MS-DOS**, la posibilidad de copiar o borrar

ficheros, inspeccionar los contenidos del sistema de archivos o lo ficheros de configuración, etc., es evidente. (Baste pensar en otras posibilidades del tipo

```
master..xp_cmdshell 'dir c:\inetpub\wwwroot\' --
```

para información del sitio Web, ó

```
master..xp_cmdshell 'NET USER usuario password' --
```

para la creación de un usuario en Windows, etc.

Y ni siquiera es preciso utilizar procedimientos extendidos de la base de datos para comprometer el sistema. Otros procedimientos llamados mediante estas técnicas o similares, bastarían. Por ejemplo,

```
master..sp_addlogin usuario password' --
```

puede servir perfectamente para el mismo propósito. La técnica de *SQL-Injection* permite llegar mucho más allá y permitirnos incluso la inclusión de un "troyano" realizado por nosotros, con objeto de devolvernos información adicional del sistema a petición. Imaginemos que dicho troyano se llama **xp_tr.dll**. Una sentencia del tipo

```
master..sp_addextendedproc 'xp_webserver',  
'c:\temp\xp_tr.dll'--
```

registraría nuestro troyano como un procedimiento almacenado al que llamaríamos mediante **xp_webserver** a partir de este momento, obteniendo la puesta en marcha de la funcionalidad almacenada en él. Y esto es sólo el principio. Las técnicas citadas pueden combinarse con la inclusión de "scripts" y muchas otras posibilidades.

Nota: Como se ha comentado, en la versión actual de SQL Server 2005, existen herramientas para habilitar esa y otras capacidades del servidor de datos, que -ahora- se encuentran fuera de servicio de forma predeterminada.

En el evento Tech-Ed 2005, de Ámsterdam, **Jesper Johansson** presentaba una interesante ponencia titulada "*Anatomy of a Network Hack: How to have your Network hacked in 10 easy steps*", donde, utilizando éstas técnicas y otras combinaciones, conseguía

hacerse con el control absoluto de un sistema, incluyendo la inyección de troyanos, creación de directorios en el sistema remoto, creación de usuarios privilegiados, etc. Si el lector está interesado en profundizar sobre el trabajo de Johanssen, le recomiendo visitar su [página http://blogs.technet.com/jesper_johansson/archive/2005/11/07/413930.aspx](http://blogs.technet.com/jesper_johansson/archive/2005/11/07/413930.aspx), donde encontrará esa y otras excelentes presentaciones.

Defensa: Consiste, como tantas veces, en un sistema mixto. Por un lado, en controlar la información introducida por el usuario (en cantidad, no permitiendo más de un número de caracteres, y en calidad, admitiendo únicamente conjuntos válidos de estos). Una rutina de validación simple podría -en VBScript- ser algo por el estilo:

```
function validarpasword( input )
    caracteres_validos =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    validarpasword = true
    for i = 1 to len( input )
        c = mid( input, i, 1 )
        if ( InStr(caracteres_validos, c ) = 0 ) then
            validarpasword = false
            exit function
        end if
    next
end function
```

Donde -al menos- podríamos comprobar que el conjunto de caracteres introducido es el permitido por la operación.

Por otra parte una inyección SQL basada en **xp_cmdShell** fallaría si la identidad (*impersonation*) del proceso que hace la llamada se limitase a un usuario creado a tal efecto, sin más permisos que los de la consulta que tiene que realizar. Si el código realiza la consulta representando al Administrador (SA en SQL Server), el permiso de ejecución está habilitado por defecto.

En las aplicaciones .NET, otra técnica más que recomendable es el uso de expresiones regulares, para comprobar la presencia de secuencias o elementos sospechosos.

Otras técnicas adicionales pasan por habilitar el registro de usuarios, para poder monitorizar los intentos fallidos, limitar los mensajes de error del servidor, y evitar concatenaciones para la construcción de sentencias SQL dinámicas (Esto último puede conseguirse fácilmente utilizando las colecciones *Parameters* de los objetos *Command*).

Naturalmente, ni así estaremos seguros, pero al menos habremos reducido notablemente la superficie de ataque, uno de los principios básicos para la defensa. A veces la solución viene por parte de validaciones combinadas, en las que el defensor debe de ponerse en la piel del atacante para un diseño adecuado de la estrategia.

Problemas de resolución de nombres canónicos

Se basa en el proceso de **canonicalización**, por el cual distintas variantes de un nombre, se convierten en un solo nombre estándar resuelto por el servidor. Esta técnica, a su vez, puede adoptar distintas variantes, como la llamada "Indexación de directorio", basada en la respuesta que ofrece el servidor ante la petición de una página inexistente. En este caso, algunos servidores pueden devolver el índice completo de las páginas existentes en el sitio, ofreciendo pistas al atacante.

Otra forma popular de este tipo de ataque se basa en las distintas formas de escribir las secuencias de escape, o la escritura de direcciones IP alternativas. Una dirección IP alternativa, se deriva de la fórmula:

$(\text{Primer byte} * 224) + (\text{Segundo byte} * 216) + (\text{Tercer byte} * 28) + (\text{cuarto byte})$

Por lo que la dirección <http://172.43.122.12> se convierte, una vez aplicada la fórmula en <http://2888530444>

Defensa: Evitar la introducción manual de nombres de archivo. Si esto no es posible, asegurarse de que están bien formados, comprobar la codificación de datos de entrada, e –incluso, si no queda más remedio- configurar IIS para evitar que la aplicación tenga acceso a recursos principales.

Debilidades derivadas de la criptografía

Como hemos visto en este capítulo, la Criptografía ofrece el mecanismo más seguro para la ocultación de información, pero hay tres factores humanos que pueden hacer que los sistemas criptográficos no cumplan su función como debieran:

- a) La utilización de algoritmos propios en detrimento de los sobradamente comprobados.

- b) Mantenimiento inseguro de las claves.
- c) Revelado accidental

Defensa: Hay varias técnicas de defensa que podemos aplicar aquí, todas ellas basadas en una actitud atenta y previsoras respecto a la información crítica.

Reciclar las claves periódicamente, utilizar ACL's y SACL's para restringir y supervisar el acceso a las claves, almacenar éstas en un dispositivo externo, utilizar claves seguras (de mayor longitud y complejidad), utilizar DPAPI y no implementar nuestras propias rutinas criptográficas.

Problemas en la manipulación de Unicode

Como es sabido el estándar Unicode, permite representar un enorme número de caracteres distintos (a diferencia de lo que sucede con ASCII) debido a que utiliza 2 bytes en vez de uno para la representación de un carácter. Esto permite que los sistemas puedan utilizar todos los alfabetos existentes sin dificultad.

El problema surge con las operaciones de conversión entre los dos sistemas, y la mayoría de las manipulaciones incorrectas en este sentido podrían llevar a una saturación de búfer, con las consecuencias comentadas a propósito de este tipo de ataque.

Las causas pueden ser: Tratamiento de un carácter Unicode como un único byte, cálculo incorrecto de tamaños de búferes de destino en la conversión, y muchas veces, la ausencia de validación posterior a la conversión (cuando una cadena Unicode se convierte a ASCII, podrían estar presentes caracteres no admisibles).

Defensa: Calcular correctamente los tamaños de búfer mediante *SizeOf* o funciones similares, convertir las cadenas Unicode a ASCII antes de la validación, utilizar la función *IsNLSDefinedString* para saber si cada carácter de una cadena tiene un resultado definido en NLS (compatibilidad para el idioma local).

Ataques de Denegación de Servicio (DoS)

La mayor parte de ellos se basan en entradas malintencionadas que producen un efecto -llamado de inanición- en un recurso vital de un equipo local o servidor. Globalmente, podríamos dividir los tipos de inanición en cuatro grupos: de CPU, de Memoria, de recursos y de red.

Las formas de producirlos son tantas como la imaginación permita soñar, y algunas, pueden aparecer de una forma tan inocente como la señalada en el primer capítulo de esta obra, a propósito de un ataque basado en una caja de diálogo de impresión en el que no se validan las solicitudes de entrada.

Defensa: Dadas las múltiples formas de implementación de este tipo de ataques, la defensa en este sentido no es puntual, sino derivada de las consideraciones de análisis y diseño que ya hemos comentado aquí en otros apartados anteriores.

Como norma, la validación de todo tipo de datos de entrada, la evitación del uso de recursos costosos en caso de fallo y las pruebas de los mecanismos de seguridad serán en mejor aval (vea el lector el capítulo siguiente, dedicado a la seguridad en las pruebas para más información al respecto).

Capítulo 5: Seguridad

vinculada a arquitecturas

La propia arquitectura de una aplicación, implica, a menudo, riesgos debidos a su propia infraestructura. Un ejemplo típico de esta situación son las tecnologías de Internet, y las aplicaciones Web y los Servicios Web en particular.

En estas aplicaciones, el acceso es –por naturaleza- de tipo global, con lo que no sabemos, de antemano, cuántos o quiénes son los clientes de nuestra aplicación. En los apartados dedicados a la Autenticación y Autorización hemos visto los principios fundamentales en los que podemos basarnos para establecer lo que algunos llaman “seguridad en la puerta”: el control de acceso y el permiso asociado a ese control.

Pero existen muchos otros aspectos a tener en cuenta, y creemos que sería bueno comentar aquí algunos de los más importantes, vinculados a la aplicación misma, y no a la seguridad asociada con el *host* donde se aloje.

Seguridad en Aplicaciones Web ASP.NET

La construcción de aplicaciones Web es uno de los subsistemas que más cambios han experimentado entre las versiones 1.1 y 2.0 de .NET Framework 2.0. Y el apartado de seguridad no iba a estar ajeno a esto.

Para comenzar merece destacarse la nueva herramienta de administración de sitios Web (*Web Site Administration Tool*), que permite a los administradores crear nuevos usuarios y funciones, y controlar el acceso a las páginas de una aplicación Web. Esta herramienta, capacita la configuración administración y gestión de muchos aspectos que –en la versión anterior- debían de ser resueltos manualmente, mediante valores asignados al(los) fichero(s) de configuración Web.config.

Como puede ver el lector, es posible configurar, tanto aspectos de seguridad, como de la propia aplicación o del proveedor (Por cierto, la propia herramienta está configurada con un *timeout*, precisamente, por razones de seguridad).

Como vemos los gráficos adjuntos, en la solapa de seguridad se encuentran las funciones críticas control, como la de selección del tipo de autenticación, la habilitación de los roles o funciones de usuario, y la de gestión de reglas o políticas de acceso al sitio Web.

Desde esta ventana, también podemos recorrer el Asistente para la Configuración de Sitios Web, para optar por una configuración general, de una sola pasada, donde seleccionaremos si el acceso es

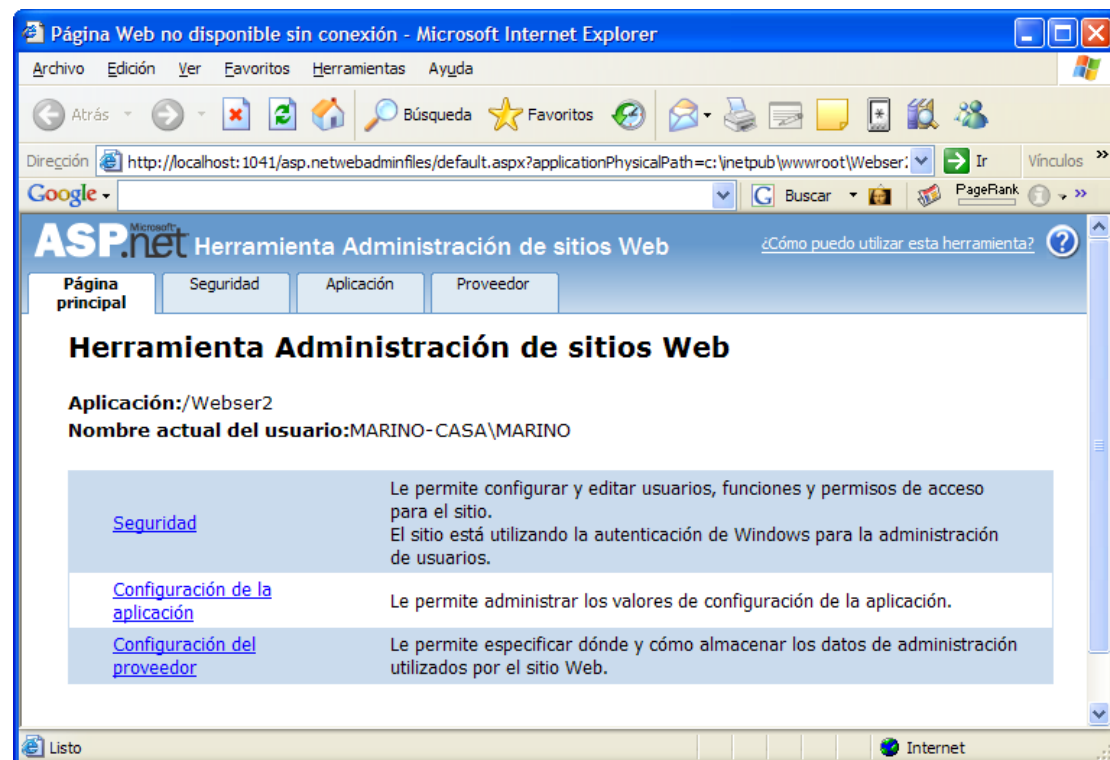


Fig. xx: La nueva herramienta de administración de seguridad

desde una red de área local o pública, las reglas de acceso que queremos establecer (por rol –si está habilitado- o por usuario), etc.

Todas estas opciones, serán más adelante, almacenadas en el fichero de configuración de la aplicación, y serán accedidas automáticamente por el código generado por ASP.NET 2.0, para la gestión de pertenencia a grupos y roles, el manejo de permisos y otro montón de opciones de configuración de la aplicación.

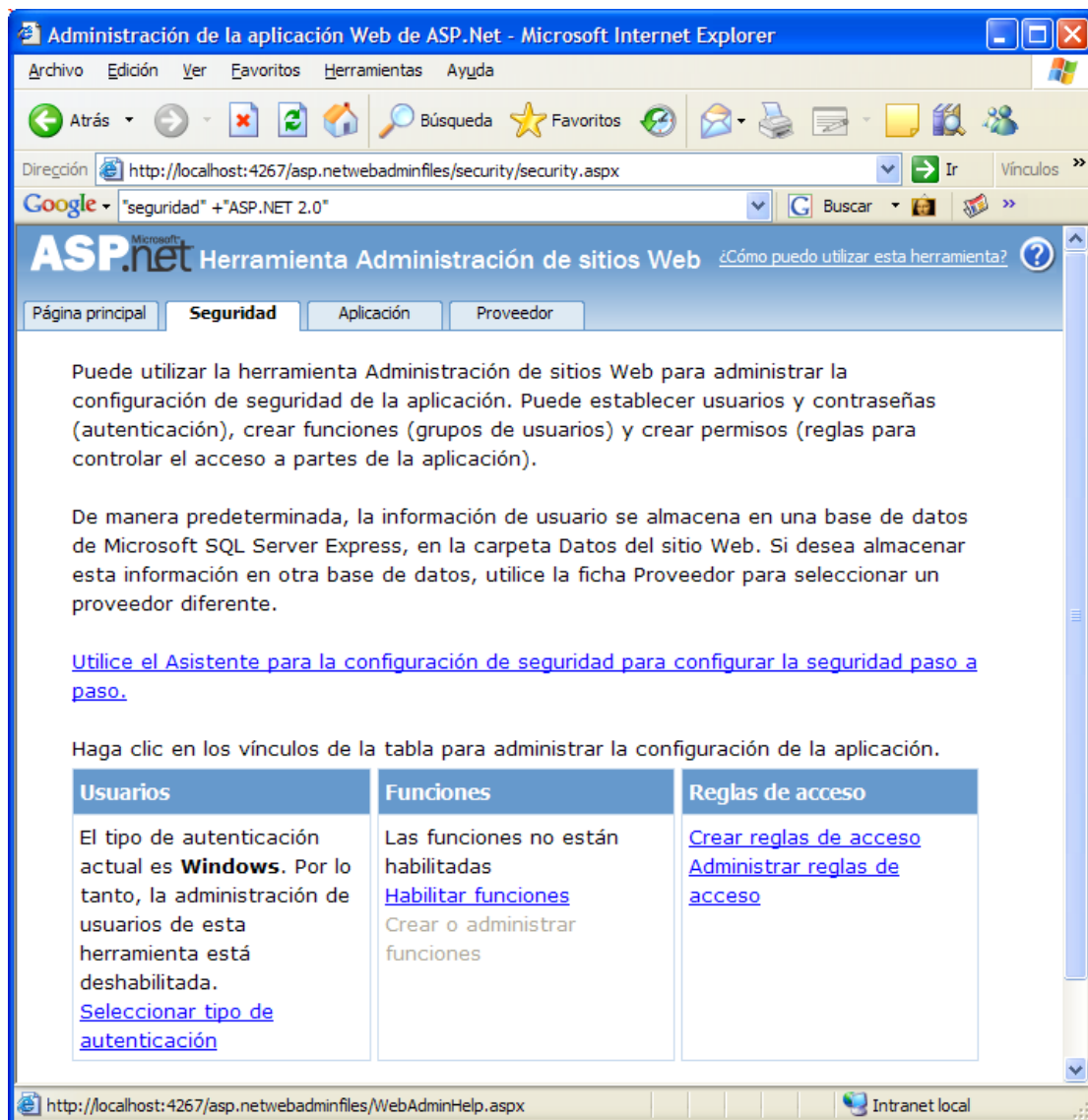


Fig. xx: Solapa de seguridad en la Herramienta de Admón. Web.

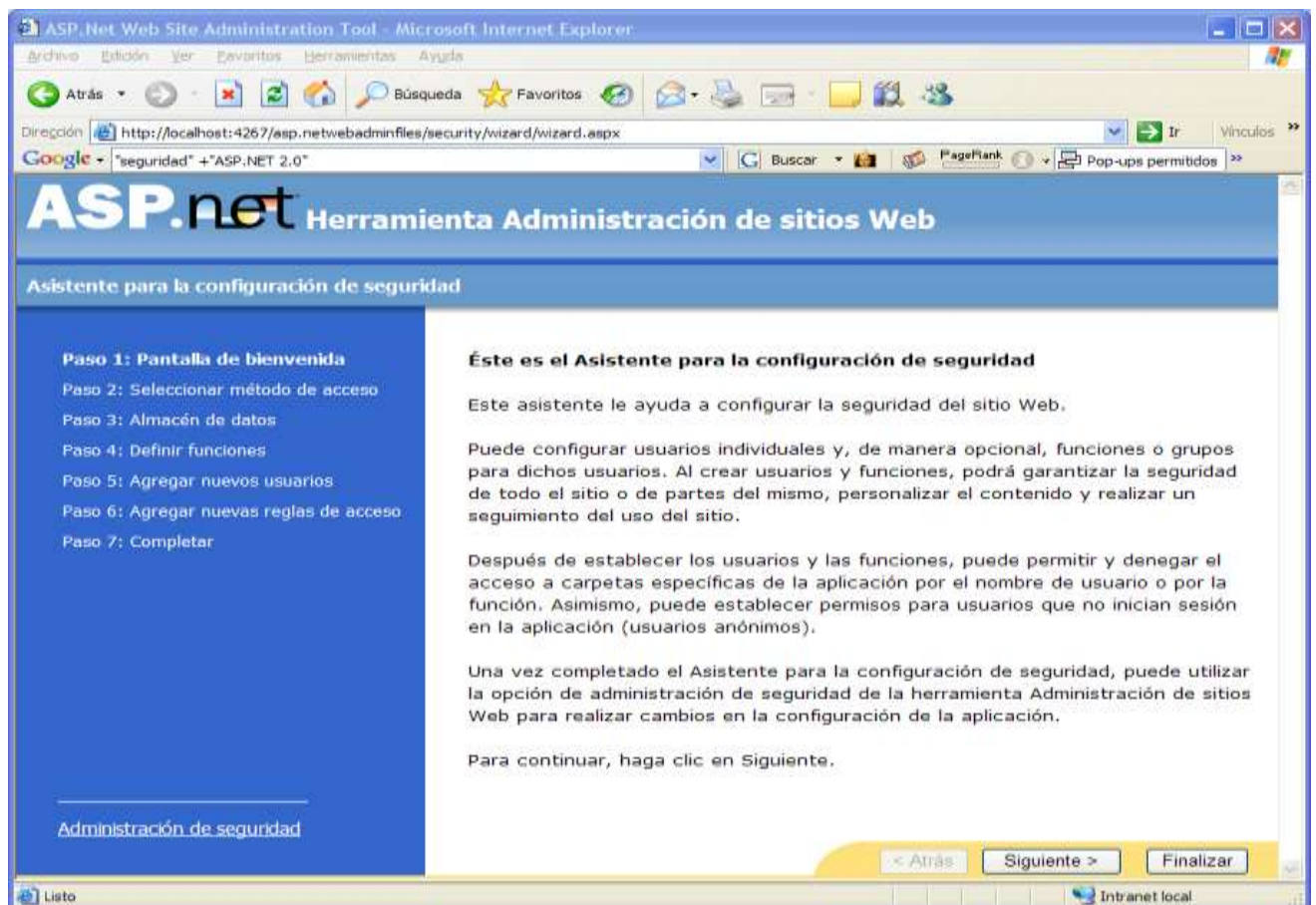


Fig.: Asistente de configuración de seguridad en ASP.NET 2.0

Agregar nuevas reglas de acceso

De manera opcional, puede agregar reglas de acceso para controlar el acceso a todo el sitio Web o a carpetas individuales. Las reglas pueden aplicarse a usuarios y funciones específicas, a todos los usuarios, a usuarios anónimos o a alguna combinación de éstos. Las reglas se aplican a las subcarpetas.

Para agregar una nueva regla de acceso, seleccione el nombre del sitio o una carpeta para la que establecer una regla. A continuación, seleccione los usuarios o las funciones a las que se aplica la regla y seleccione Permitir o Denegar.

Si define varias reglas, se aplicarán en el orden en el que se aparecen en la tabla. Se aplicará la primera regla que coincida. Podrá agregar, eliminar, modificar y reordenar las reglas más adelante.

Agregar nueva regla de acceso

Seleccione un directorio para esta regla:



DemoSeg1

App_Data

La regla se aplica a:

☐ Función

[funciones deshabilitadas]

☒ Usuario

☐ Todos los usuarios

☐ Usuarios anónimos

Permiso:

☐ Permitir

☒ Denegar

Agregar esta regla

[Buscar por usuarios](#)

Las reglas que aparecen atenuadas se heredan de la regla principal y no pueden cambiarse en este nivel.

Permiso	Usuarios y funciones	Eliminar
Permitir	 [todos]	Eliminar

Fig.: Opción de selección de roles y permisos y gestión de reglas

Novedades en la seguridad en ASP.NET 2.0

Aparte de esto, existe un amplio conjunto de novedades vinculadas con la seguridad en ASP.NET 2.0. Citemos las más importantes:

Controles de Inicio de Sesión: Los nuevos controles *Login*, permiten generar con rapidez mecanismos de autenticación. Se pueden crear tanto páginas de inicio de sesión como páginas de registro y recuperación de contraseñas de forma semiautomática,

Nuevas API's de pertenencia a grupo (Membership): son un conjunto de clases que contienen métodos para crear y recuperar información sobre los usuarios de las aplicaciones. También se

incorporan clases que simplifican el trabajo con las funciones de usuario personalizadas.

Gestor de roles (*RoleManager*): permite una gestión sencilla de los roles basada en 3 proveedores de almacenamiento: *SqlRoleProvider* para almacenes de rol SQL Server, *WindowsTokenRoleProvider* usado con la autenticación de Windows que utiliza como roles grupos de Windows, y *AuthorizationStoreRoleProvider*, que utiliza *Windows Server 2003 Authorization Manager* para gestionar roles en Active Directory o ADAM.

Fichero de configuración de seguridad a nivel de máquina: antes, estos valores se almacenaban en el fichero general de configuración Machine.config. Ahora, se ubican en un fichero separado en el directorio `\Windows\Microsoft.NET\Framework\{version}\CONFIG`, de nombre Web.config.

Cifrado de ficheros de configuración: ASP.NET versión 2.0 introduce una característica que permite cifrar secciones de los ficheros Machine.config y Web.config ya sea utilizando DPAPI o cifrado RSA. Especialmente útil en el cifrado de cadenas de conexión, y credenciales de usuario.

Monitorización de accesos indebidos: Otra novedad consiste en la posibilidad de hacer un seguimiento de aquellos intentos fallidos de acceder a la Web. Estos incluyen los fallos en la utilización del API de pertenencia a grupos, intentos de modificar o reutilizar los tickets de autenticación por formularios, o eventos de infraestructura, como los fallos en el acceso al sistema de ficheros. Permite incluso la creación de eventos personalizados para el seguimiento detallados de aspectos particulares de una aplicación.

Seguridad en el acceso al código: El proveedor administrado de acceso a datos de SQL Server no exige la ejecución con permisos *FullTrust*. Las aplicaciones con permisos *MediumTrust* pueden ahora acceder a datos de la misma forma. Así mismo, para permitir que las aplicaciones de niveles *FullTrust*, *HighTrust* y *MediumTrust* puedan enviar correo, *SmtpPermission* está disponible en todos estos niveles.

Cifrado y descifrado de tickets de autenticación: Ahora es posible utilizar criptografía con los tickets de autenticación, para añadir un nivel extra de seguridad, utilizando la clave `<machinekey>`, con un nuevo atributo *Decryption*, donde podemos establecer el tipo de algoritmo de cifrado a utilizar: AES (predeterminado), DES o TripleDES.

La seguridad y el modelo de proveedores

Si requerimos una gestión personalizada de la autenticación y/o autorización, podemos optar por la autenticación por formularios (tipo "Forms"). Inmediatamente después de habilitarla, sin necesidad de generar tablas de base de datos ni de escribir ningún código, se puede comenzar a registrar y autenticar usuarios comparando la información de la base de datos. Esto es posible porque el marco ASP.NET 2.0 utiliza un modelo de proveedores para la seguridad (*Security Provider Model*), soportado por una base de datos local que puede adoptar varios formatos, y que se almacena en el directorio App_Data, creado automáticamente por Visual Studio 2005.

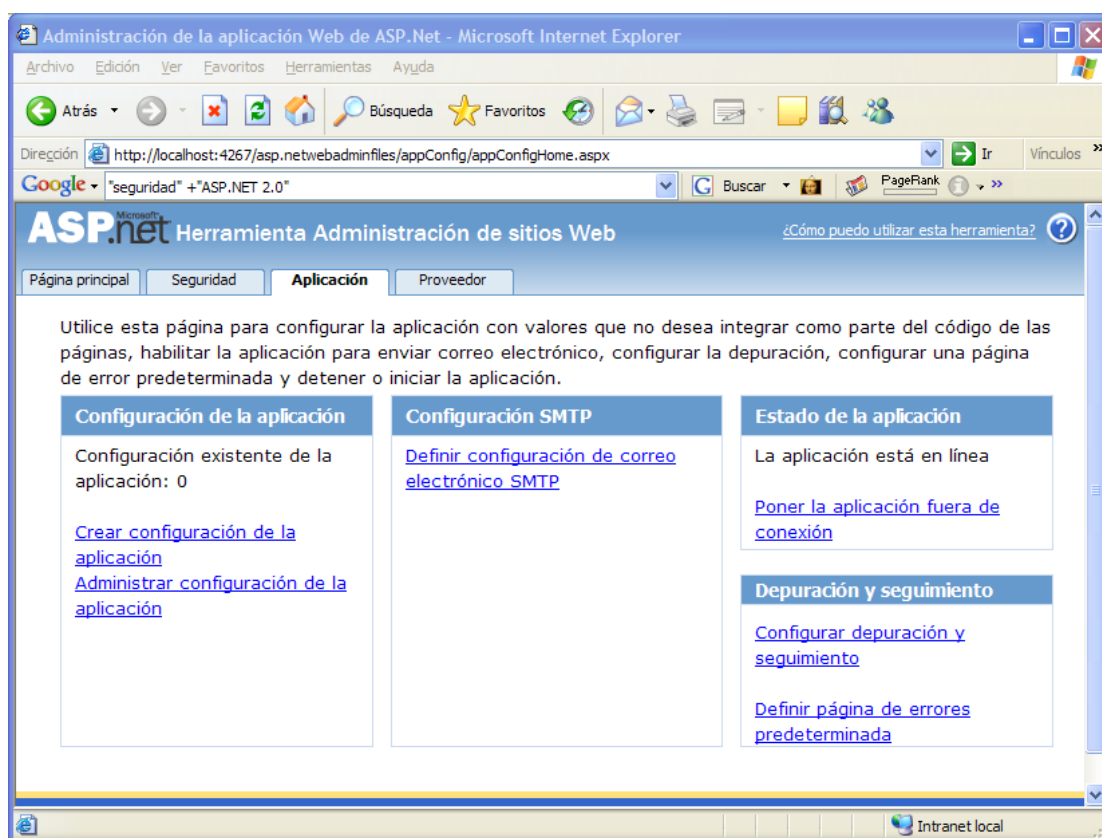
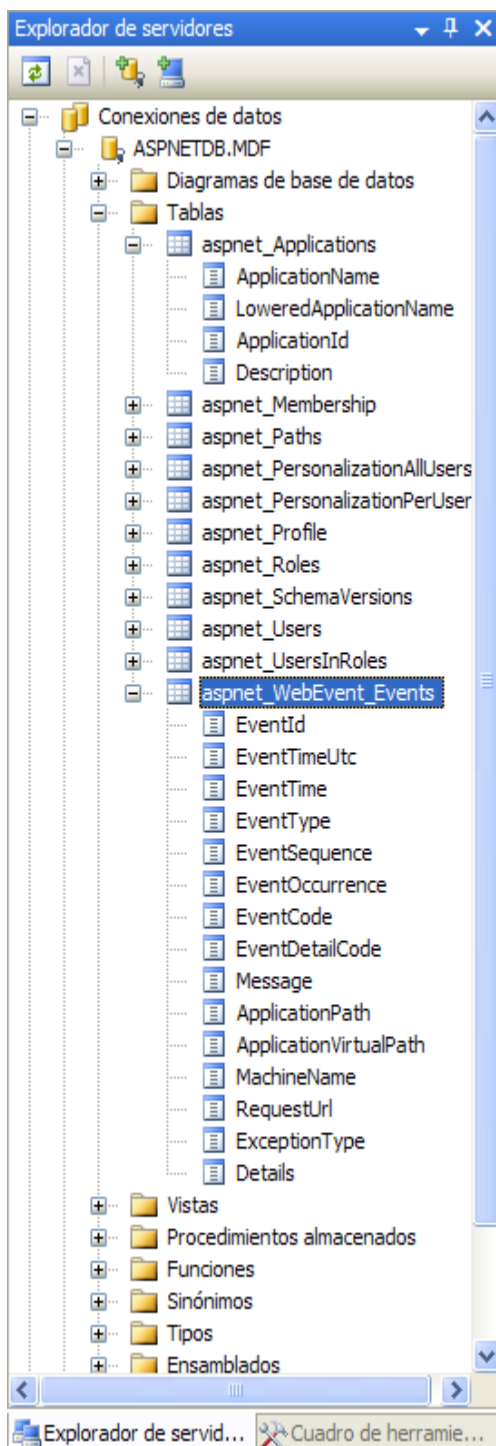


Fig.: Pantalla de configuración de la aplicación

El modelo de proveedores, ofrece un método estándar para conectar componentes, encargados de implementar los distintos servicios de la aplicación. ASP.NET 2.0 emplea dos tipos de proveedores para la seguridad: el proveedor de pertenencia a grupo y el proveedor de roles. El proveedor de pertenencia a grupo se utiliza para almacenar nombres de usuario y contraseñas, mientras

que el proveedor de roles (lógicamente) para guardar y recuperar información sobre roles de usuario.

Como hemos apuntado, toda esta información se almacena en una base de datos creada automáticamente por Visual Studio en el directorio *App_Data*, que puede ser de dos tipos: Access, si no hemos optado por la instalación predeterminada de una instancia de *SQL Server Express Edition*, (o si no tenemos instalada en la máquina ninguna versión de otro tipo de SQL Server), o SQL Server (archivos .MDF) si lo hemos hecho de forma predeterminada o ya disponemos de una instancia del servidor instalada. Además, después de habilitar los roles y crear usuarios, el fichero Web.Config, mostrará esta información en los apartados correspondientes, tal y como muestra el código adjunto.



```
<roleManager enabled="true"
defaultProvider="AspNetWindowsTokenRole
Provider" />
```

Uso de SqlMembershipProvider

Para almacenar la información de pertenencia a grupo en una base de datos de Microsoft SQL será necesario establecer el proveedor de pertenencia a grupo predeterminado de la aplicación. El siguiente archivo Web.Config (de ayuda de MSND) muestra una configuración prototipo de este estilo:

Fig.: Base de datos de ASP.NET predeterminada, creada por visual Studio 2005

```
<configuration>
  <connectionStrings>
    <add name="SqlServices" connectionString="Data
Source=localhost;Integrated Security=SSPI;Initial Catalog=aspnetdb;" />
  </connectionStrings>
  <system.web>
    <authentication mode="Forms" >
      <forms loginUrl="login.aspx"
        name=".ASPXFORMSAUTH" />
    </authentication>
    <authorization>
      <deny users="?" />
    </authorization>
    <membership defaultProvider="SqlProvider"
      userIsOnlineTimeWindow="15">
      <providers>
        <add
          name="SqlProvider"
          type="System.Web.Security.SqlMembershipProvider"
          connectionStringName="SqlServices"
          applicationName="MyApplication"
          enablePasswordRetrieval="false"
          enablePasswordReset="true"
          requiresQuestionAndAnswer="true"
          requiresUniqueEmail="false"
          passwordFormat="Hashed"
          maxInvalidPasswordAttempts="5"
          passwordAttemptWindow="10" />
      </providers>
    </membership>
  </system.web>
</configuration>
```

Para comprobar que la aplicación ASP.NET puede obtener acceso a ella, nos basta con utilizar la herramienta de configuración antes citada en la solapa Proveedor, en el enlace "Prueba":

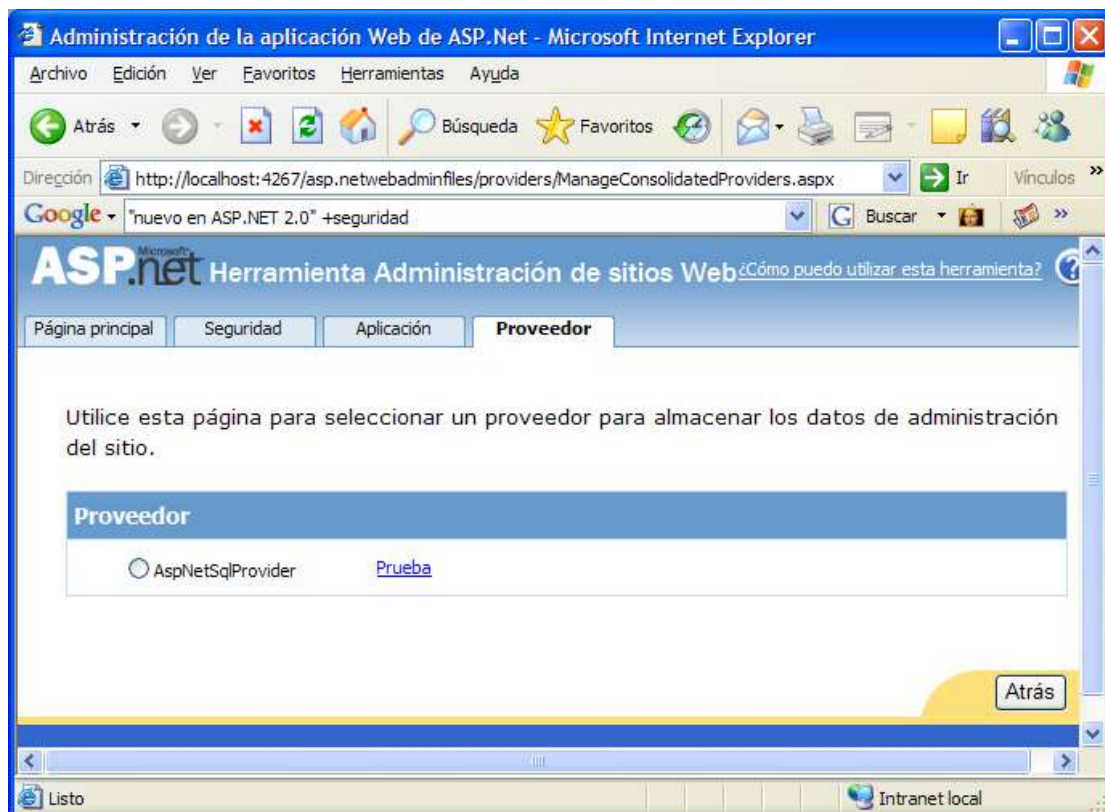


Fig. xx: Opción de comprobación de conexión con la base de datos de pertenencia y gestión de roles.

De forma predeterminada se utiliza una seguridad integrada. Por lo tanto, será preciso asegurarse de que la cuenta de ASP.NET (*NT AUTHORITY\NETWORK SERVICE* en el caso de Windows Server 2003 o ASPNET en el caso de Windows 2000 o XP) dispone de los permisos necesarios para obtener acceso a la base de datos *aspnetdb*.

Configuración de las propiedades del proveedor de pertenencia a grupo

Los proveedores como *SqlMembershipProvider* admiten varios atributos de proveedor específicos: *applicationName*, *connectionStringName*, *passwordFormat*, etc, que permiten una configuración detallada del comportamiento.

Sugerencia: *passwordFormat* determina cómo se almacenan las contraseñas en la base de datos. Se ofrece la opción de almacenar contraseñas de texto sin cifrar, contraseñas cifradas o valores *hash* de contraseñas. Por motivos de seguridad, puede que se deseen almacenar valores *hash* en la base de datos en lugar de las contraseñas reales para que, si resulta comprometida la seguridad de la aplicación Web, los usuarios malintencionados no puedan robar las contraseñas de usuario auténticas.

Si, por motivos de seguridad, no se desea permitir que los usuarios recuperen sus contraseñas, se puede impedir mediante el establecimiento del atributo *enablePasswordRetrieval*.

Ejemplos de implantación de seguridad en ASP.NET 2.0

Aunque el análisis de todas las posibles implementaciones relacionadas con la seguridad de ASP.NET 2.0 requeriría –por sí sola- un libro como este, hemos seleccionado algunos ejemplos para dar una idea al lector del conjunto de novedades disponible, de forma que pueda fácilmente añadir sus propios elementos de seguridad partiendo de estas sugerencias.

Gestión de la Autenticación y Autorización: controles Login y API's de pertenencia a grupos.

Controles de Inicio de Sesión

ASP.NET 2.0 contiene una serie de controles nuevos relacionados con la seguridad, conocidos como controles **Inicio de Sesión**. Con ellos, se pueden crear páginas estándar de registro, de inicio de sesión y de recuperación de contraseñas sin necesidad de escribir código.

También se emplean para mostrar distinta información a los usuarios dependiendo de sus funciones y de su estado de autenticación en ese momento. (*LoginView* permite definir las distintas plantillas que se muestran a los miembros de distintos roles). Además, el control *Login* aprovechará automáticamente toda la funcionalidad del modelo de proveedores, cualquiera que sea el elegido.

Antes de utilizar cualquiera de los controles de Inicio de Sesión, se debe habilitar la autenticación por formularios para la aplicación.

Uso del control de Inicio de Sesión

Basta con arrastrar un control Login a una página, para que automáticamente incluya un código como este:

```
<asp:Login ID="Login1" Runat="server" />
```

El control *Login* genera automáticamente una interfaz de inicio de sesión estándar. Tras aplicar un formato automático y algún aspecto de configuración, obtendremos un código de este tipo:

```
<asp:Login ID="Login1" runat="server" BackColor="#F7F7DE"
BorderColor="#CCCC99" BorderStyle="Solid"
    BorderWidth="1px" Font-Names="Verdana" Font-Size="10pt">
    <TitleTextStyle BackColor="#6B696B" Font-Bold="True"
ForeColor="White" />
</asp:Login>
```

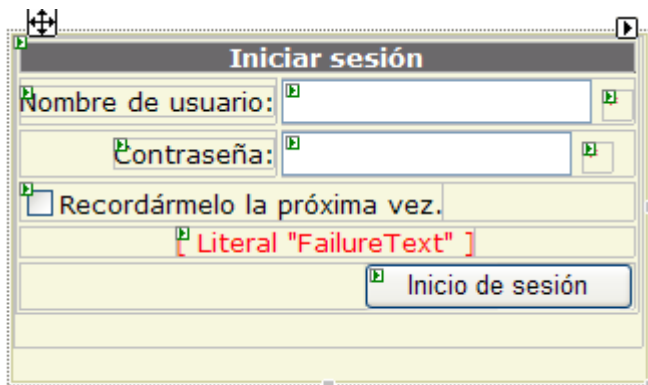


Figura xx. Creación de una interfaz de inicio de sesión con el control Login

El control *Login* funciona conjuntamente con los controles relacionados en el epígrafe "Inicio de Sesión", como *LoginView*, *PasswordRecovery* y *LoginName* permitiendo enviar el nombre de usuario y la contraseña, e implementando automáticamente un mecanismo de validación de credenciales.

Uso del control *CreateUserWizard*

Otro control de interés en la implantación de la seguridad en ASP.NET 2.0 es *CreateUserWizard*, que permite crear una página estándar de registro de usuarios. Basta agregar el control a la página en tiempo de diseño o escribir la siguiente etiqueta a la página, y los nuevos usuarios podrán registrarse en el sitio Web.

```
<asp:CreateUserWizard ID="CreateUserWizard1" Runat="server" />
```

Fig.: Asistente de creación de usuarios

Las opciones concretas de la interfaz generada por CreateUserWizard dependerán de la configuración del proveedor de pertenencia a grupo de la aplicación. Por ejemplo, los cuadros de texto de la pregunta y la respuesta para recuperar la contraseña sólo aparecen cuando está activado el atributo *requiresQuestionAndAnswer*. La página de registro que se muestra en la figura, se genera cuando se utiliza *CreateUserWizard* con el proveedor de pertenencia a grupo *AspNetSqlProvider*.

Este control, dispone, además de una interesante funcionalidad de correo electrónico que permite automatizar el envío de respuestas al usuario registrado.

Uso del control *PasswordRecovery*

Como su nombre indica, *PasswordRecovery* permite que los usuarios puedan solicitar un mensaje de correo electrónico de recordatorio de su contraseña. Al igual que el control *CreateUserWizard*, las propiedades del mensaje de correo electrónico enviado al usuario se definen mediante la propiedad *MailDefinition*.

Al igual que los anteriores, admite personalización de casi todos los aspectos funcionales y estéticos, mediante los asistentes personalizados que cada control tiene en esta versión de .NET.

Figura 5. El control PasswordRecovery

Si observamos el código ASP generado, veremos un código similar al siguiente:

```
<asp>PasswordRecovery ID="Recordatorio1" Runat="server">
  <MailDefinition
    From="Sitio@Dominio.com"
    Subject="Recordatorio de contraseña">
  </MailDefinition>
</asp>PasswordRecovery>
```

Siguiendo el patrón utilizado hasta aquí, dependiendo de la configuración del proveedor de pertenencia a grupo predeterminado, admite distintas opciones. Si se activa el atributo *enablePasswordRetrieval* y el atributo *passwordFormat* no tiene el valor *Hashed*, se envía la contraseña de texto sin cifrar original del usuario en el mensaje de correo electrónico. En cualquier otro caso, la contraseña se restablece en una secuencia de caracteres generada de forma aleatoria que se envía al usuario.

Uso del control *ChangePassword*

El control *ChangePassword*, como se puede suponer, permite a los usuarios cambiar sus contraseñas registradas. El control crea cuadros de texto en los que se puede facilitar la contraseña original y la nueva (ver figura adjunta).

Figura 6. Cambio de la contraseña con el control ChangePassword

Al igual que los controles *CreateUserWizard* y *PasswordRecovery*, el control *ChangePassword* incluye una propiedad *MailDefinition*. Cuando se asignan valores a la propiedad *MailDefinition*, el control *ChangePassword* envía automáticamente un mensaje de correo

electrónico al usuario cuando se cambia correctamente una contraseña.

Podemos definir una etiqueta que declara una instancia del control *ChangePassword* y define la propiedad *MailDefinition*, con el siguiente código:

```
<asp:ChangePassword ID="CambiarContraseña" Runat="server">
  <MailDefinition
    BodyFileName="~/CambiarContraseñaCorreo.txt"
    From=" Sitio@Dominio.com"
    Subject="Contraseña actualizada">
  </MailDefinition>
</asp:ChangePassword>
```

Donde podemos ver que las propiedades *BodyFileName*, *From* y *Subject* definen las características de envío. En el archivo de texto, se pueden utilizar expresiones especiales como `<% Username %>` y `<% Password %>` para mostrar el nombre de usuario y la contraseña modificada del usuario registrado en el mensaje.

LoginName y LoginStatus

Como indica su nombre, *LoginName* y *LoginStatus* permiten mostrar información sobre el estado actual de la autenticación de un usuario. Después de que un usuario ha iniciado sesión en la aplicación, el control *LoginName* muestra su nombre de usuario registrado. Si un usuario no se autentica con la autenticación por formularios, el control *LoginName* no muestra nada. Puede declararse en una página de la siguiente forma:

```
<asp:LoginName ID="LoginName1" Runat="server" />
```

Por su parte *LoginStatus*, permite al usuario iniciar o cerrar sesión. El control muestra uno de dos vínculos. Si el usuario no se ha autenticado, se muestra un vínculo a la página *Login.aspx*. Si ya se ha autenticado, se muestra un vínculo que le permite cerrar sesión. Podemos declararlo con el siguiente código:

```
<asp:LoginStatus ID="LoginStatus1" Runat="server" />
```

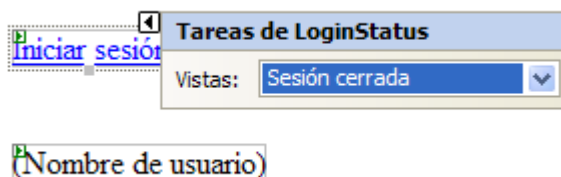


Fig. xx: Aspecto de los controles LoginStatus y LoginName.

El control *LoginView*

Finalmente, *LoginView*, permite mostrar contenido distinto dependiendo del rol del usuario. Por ejemplo, dependiendo de si el usuario es nuevo o ya está registrado.

A continuación se muestra cómo se puede utilizar el control *LoginView* para mostrar contenido distinto a un usuario anónimo y a un usuario autenticado.

```
<asp:LoginView ID="LoginView1" Runat="server">
  <LoggedInTemplate>
    Bienvenido<asp:LoginName ID="LoginName1" Runat="server" />
  </LoggedInTemplate>
  <AnonymousTemplate>
    ¡Bienvenido a nuestro sitio Web! Para ver toda la información
    disponible considere registrarse con nosotros.
  </AnonymousTemplate>
</asp:LoginView>
```

Un usuario anónimo ve todo el contenido incluido en la plantilla *AnonymousTemplate*, mientras que uno autenticado accede a todo lo que exista dentro de la plantilla *LoggedInTemplate*.

Si habilita las funciones predeterminadas para la aplicación Web, puede utilizar el control *LoginView* para mostrar contenido que sólo puedan ver miembros con funciones específicas. Por ejemplo, imagine que utilizó la herramienta de administración de sitios Web para crear una nueva función llamada Administradores. Se podría, entonces, usar *LoginView* para mostrar contenido exclusivo de un rol, añadiendo la etiqueta *<RoleGroups>* con una información de este tipo:

```
<RoleGroups>
  <asp:RoleGroup Roles="Administrador">
    <ContentTemplate>
      Solo para administradores
    </ContentTemplate>
  </asp:RoleGroup>
```

```
</RoleGroups>
```

El funcionamiento es exclusivo, esto es, cada miembro sólo puede acceder a aquello que pertenece a su platilla.

El API de Pertenencia a grupo: La Clase *Membership*

Es otra novedad de ASP.NET 2.0 que permite el control mediante programa de todos estos aspectos que hemos visto por diseño y permite realizar un ajuste fino de las condiciones de ejecución en la mayoría de las veces en las que lo generado por diseño no es suficiente.

La clase *Membership* incluye métodos que permiten crear nuevos usuarios, cambiar contraseñas y buscar usuarios que coincidan con criterios específicos. En un segundo plano, los controles Login utilizan estos métodos para interactuar con el proveedor de pertenencia a grupo configurado.

Esta clase dispone de métodos como *CreateUser*, *DeleteUser*, *FindUsersByEmail*, *FindUsersByName*, *GeneratePassword*, *GetAllUsers*, *GetNumberOfUsersOnline*, *GetUser*, *GetUsernameByEmail*, *UpdateUser*, y *ValidateUser*, todos ellos suficientemente explícitos como para no necesitar una explicación adicional.

Para manejar usuarios, Membership permite una gestión sencilla: desde crear un nuevo usuario con una única línea de código, hasta su validación:

```
Membership.CreateUser( "Onofre", "eRf0no" );  
// ...y para comprobar un usuario:  
if (Membership.Validate("Onofre", "eRf0no")) ....etc...
```

Esta instrucción crea un nuevo nombre de usuario llamado Onofre, con la contraseña "eRf0no".

Además podemos usar métodos como *GetAllUsers* y *FindUsersByName*, para obtener colecciones de objetos *MembershipUser*. La clase *MembershipUser* representa información sobre un usuario concreto, con montones de propiedades útiles. De hecho, podríamos ver toda la información de los usuarios registrados en un objeto GridView icon solo un par de líneas de código en el evento Load de la página! (una vez autenticados, y suponiendo que el usuario tenga los permisos adecuados). El código siguiente sería suficiente para ese propósito:

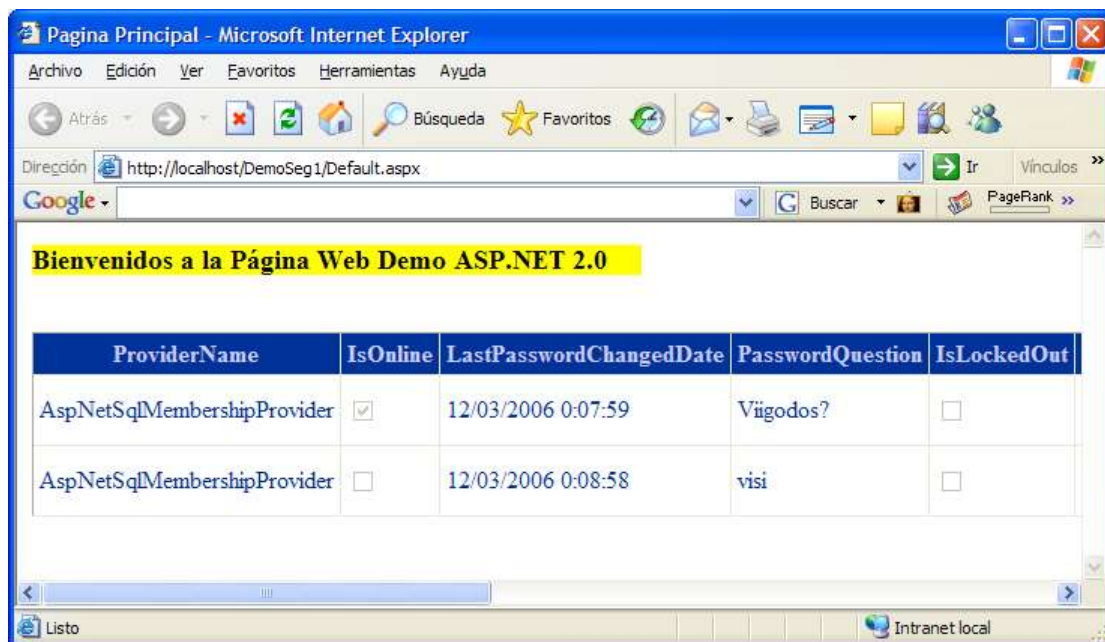


Fig._ Salida del código anterior mostrando propiedades de los usuarios registrados

Los usuarios están vinculados al control *GridView* dentro del método *Page_Load*. La lista de usuarios se recupera mediante el método *GetAllUsers* de la clase *Membership*:

```
this.GridView1.DataSource =
    System.Web.Security.Membership.GetAllUsers();
this.GridView1.DataBind();
```

Como vemos, sencillez en el diseño y la operación y un control sin precedentes hasta ahora.

Perfiles

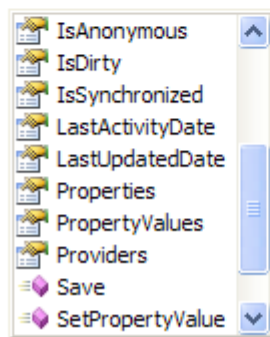
También resulta de extrema facilidad la creación de perfiles de usuario, definiéndolos en el fichero *Web.Config*, mediante la adición de una etiqueta *<Profile>* a la que podemos añadir los campos que necesitemos para describir adecuadamente el perfil. Además, el concepto de perfil se representa en ASP.NET 2.0 mediante una clase estática *Profile*, mediante la que podemos acceder a propiedades de los perfiles de forma automática:

```

using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Profile.
    }
}

```



Seguridad Declarativa

La implementación del ajuste fino de la seguridad, en ocasiones requiere llegar al mismo código fuente, e incluso a miembros individuales de una clase o control.

En .NET Framework 2.0, se siguen los principios ya establecidos en versiones anteriores, pudiéndose, mediante atributos, establecer un control de seguridad mediante aserciones que establecen quién puede acceder a qué partes de la aplicación. Este control, podemos realizarlo por rol o por usuario individual o una mezcla de ambos.

Por ejemplo, para exigir que una clase sólo pueda ser accedida por los administradores del sistema, basta con añadir el atributo *PrincipalPermissionAttribute*, con los argumentos *SecurityAction.Demand* y *Role*.

```

[PrincipalPermissionAttribute(SecurityAction.Demand,
Role=@"DOMINIO\Administradores")]
public sealed class SoloAccesibleAdministradores
{
}

```

Seguridad Imperativa

Si la seguridad a nivel de clase/miembro, no fuese suficiente, podemos llegar a fragmentos concretos del código utilizando seguridad imperativa, mediante la creación *PrincipalPermission*, como por ejemplo:

```
PrincipalPermission permCheck = new PrincipalPermission(  
    null, @" DOMINIO\Administradores");  
permCheck.Demand();
```

también podemos evitar el uso de una variable local mediante:

```
new PrincipalPermission(null,  
@"DOMINIO\Administradores").Demand();
```

Esto crea un objeto *PrincipalPermission* con un usuario con el nombre en blanco y un rol especificado, y entonces llama al método *Demand()*. Esto hace que el CLR interroge al objeto *Principal* vinculado al proceso de ejecución activo para comprobar si la identidad asociada es un miembro del rol indicado. En este ejemplo se utiliza autenticación Windows, por lo que si estuviéramos usando la anterior autenticación por formularios la solicitud debería indicar alguno de los roles creados mediante la herramienta de Administración del Sitio Web.

Si, como resultado de la llamada el miembro indicado no pertenece al grupo, se genera una excepción que deberemos de controlar adecuadamente. El ejemplo que sigue muestra cómo restringir el acceso a un usuario concreto:

```
(new PrincipalPermission(@"DOMINIO\Invitado", null)).Demand();
```

Seguridad en los Servicios Web

Hasta la versión 1.1 (y también en esta nueva 2.0), la seguridad en los servicios Web puede aplicarse a 3 niveles distintos:

- Al nivel de transporte: utilizando un canal seguro.
- Al nivel de la aplicación
- Al nivel del mensaje

La selección de una u otra, depende de las características de la plataforma e infraestructura de la aplicación, así como de algunas otras consideraciones derivadas del análisis.

El primer nivel, puede ser adecuado, por ejemplo, en una Intranet, donde tenemos controlados todos los mecanismos de la infraestructura así como del canal, y resulta la opción más fácil de implantar.

En el segundo nivel, es la aplicación misma la que se encarga de la seguridad, lo que presupone varios aspectos concretos:

- La existencia de una información de credencial incluida en la cabecera del mensaje (SOAP Header)
- La aplicación puede generar su propio objeto que implemente la interfaz *IPrincipal*, conteniendo roles. Puede tratarse de una clase personalizada o la propia *GenericPrincipal* suministrada por .NET.
- La aplicación puede, de forma selectiva, cifrar el contenido, aunque eso supone el almacenamiento de claves.
- También pueden usarse SSL (*Secure Socket Layers*), de forma combinada con cabeceras SOAP, a efectos de autenticación.

Nota¹⁵: Se puede utilizar SSL para establecer conexiones seguras en redes que no son de confianza (como Internet). SSL cifra y descifra mensajes enviados entre el cliente y el servidor. Al cifrar los datos, se evita que los mensajes sean leídos por terceros durante la transmisión. SSL cifra el mensaje del cliente y, a continuación, lo envía al servidor. Una vez que el servidor lo recibe, SSL lo descifra y comprueba que procede del remitente correcto (proceso que se

¹⁵ <http://www.microsoft.com/spanish/msdn/articulos/archivo/030502/voices/xmlwssec.asp>

denomina autenticación). El servidor, o el servidor y el cliente, pueden tener certificados que se utilizan como parte del proceso de autenticación y que proporcionan capacidades de autenticación además del cifrado de la conexión. Aunque es un método muy eficaz para crear comunicaciones seguras, SSL presenta un coste en rendimiento que debe tenerse en cuenta. Los servicios Web XML de Microsoft admiten SSL integrado, tanto en clientes como en servidores.

Por último, si optamos por el nivel de mensaje, conseguimos el mayor nivel de flexibilidad, si bien deberemos de codificar su comportamiento, y utilizar el estándar *WS-Security*, para asegurar en un solo mensaje autenticación, integridad y confidencialidad.

Es perfecto para escenarios heterogéneos donde no podemos controlar ambos puntos de comunicación, ni los nodos intermedios. Sus características más sobresalientes son:

- Inclusión de secuencias de caracteres en la cabecera SOAP, a efectos de autenticación, que pueden ser de varios tipos: tickets Kerberos, certificados X509, o una secuencia binaria personalizada.

Autorización y características del protocolo

En resumen, podemos aprovechar las características de autenticación del protocolo que se utilice para intercambiar mensajes. Microsoft Internet Information Server (IIS) e ISA Server funcionan en conjunción con Windows 2000 Server y ofrecen soporte para varios mecanismos de autenticación en HTTP.

- Básica: Usada para identificación no segura o poco segura de clientes, ya que el nombre de usuario y la contraseña se envían como texto codificado en base64Binary, que puede ser fácilmente descodificado. IIS autorizará el acceso a los servicios Web si las credenciales coinciden con las de una cuenta de usuario válida.
- Básica sobre SSL: Como la autenticación básica, excepto que el canal de comunicación está cifrado y protege de ese modo el nombre de usuario y la contraseña. Una buena opción para entornos en Internet; sin embargo, el uso de SSL influye negativamente en el rendimiento.
- Implícita: Utiliza algoritmos *hash* para transmitir las credenciales del cliente de forma segura. Es posible que no

sea compatible con todas las herramientas de desarrollo. IIS autoriza el acceso a los servicios Web si las credenciales coinciden con las de una cuenta de usuario válida.

- Autenticación de Windows integrada: Util sobre todo en entornos en Intranet. Utiliza NTLM o Kerberos y el cliente debe pertenecer al mismo dominio que el servidor o a uno de confianza. IIS autorizará el acceso a los servicios Web si las credenciales coinciden con las de una cuenta de usuario válida.
- Certificados de cliente a través de SSL: Cada cliente obtiene un certificado. Los certificados se asignan a las cuentas de usuario, utilizadas por IIS para autorizar el acceso a los servicios Web XML. Es una solución viable para entornos en Internet. Es posible que no sea compatible con todas las herramientas de desarrollo para generar clientes de servicios Web. El rendimiento puede verse afectado, al estar sólo disponible en conexiones SSL

El siguiente ejemplo, de Don Smith¹⁶, muestra cómo implementar secuencias personalizadas (*UsernameToken*) y adjuntarlo a un mensaje de solicitud Web.

```
using Microsoft.Web.Services.Security;
...
// Crea secuencia Username
UsernameToken token = new UsernameToken( "Usuario", "Contraseña",
PasswordOption.SendPlainText );
// Crea una instancia del Proxy del servicio
WeblogProxy proxy = new WeblogProxy();
// Añade la secuencia al contexto de la petición
proxy.RequestSoapContext.Security.Tokens.Add( token );
// Llama al servicio
ConfirmedWeblogEntry confirmed = proxy.AddEntry( newEntry );
```

Este tipo de secuencias se adjuntan a los mensajes de respuesta y de solicitud SOAP para que el extremo receptor pueda comprobar la reclamación de identidad. La clase *SecurityToken* es una implementación de la interfaz *System.Xml.IXmlElement*, la cual representa un elemento XML en .NET Framework. Por lo tanto, no resulta extraño el efecto que causa esta inclusión en el mensaje SOAP, como puede verse en el código adjunto:

```
<?xml version="1.0" encoding="utf-8"?>
```

¹⁶ <http://www.microsoft.com/spanish/msdn/articulos/archivo/141103/voices/wssecdrill.asp>

```

<soap:Envelope xmlns:soap="..." xmlns:wsu="..." xmlns:wssse="...">
  <soap:Header>
    ...
    <wssse:Security soap:mustUnderstand="1">
      <wssse:UsernameToken
xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
wsu:Id="SecurityToken-c7ef4231-4397-4b4b-ab8d-0ea3fad1f79e">
        <wssse:Username>domain_name\Usuario</wssse:Username>
        <wssse:Password
Type="wssse:PasswordText">Contraseña</wssse:Password>
        <wssse:Nonce>QLSVRt9g3e19jJXJYhtBKA==</wssse:Nonce>
        <wsu:Created>2006-02-05T22:37:52Z</wsu:Created>
      </wssse:UsernameToken>
    </wssse:Security>
  </soap:Header>
  <soap:Body>
    <AddEntry xmlns="http://sitioweb/wse/ejemplos/2006/02">
      <entry>
        <title>¿Viste Terminator III anoche?</title>
        <author>Usuario</author>
        <issued>2006-02-05T18:37:52.0030368-04:00</issued>
        <content>Buenos efectos especiales</content>
      </entry>
    </AddEntry>
  </soap:Body>
</soap:Envelope>

```

Como se puede ver a continuación, para la implementación de la autenticación personalizada basta con reemplazar el método *AuthenticateToken* de un *UsernameTokenManager*.

```

public class WeblogAuthManager : UsernameTokenManager
{
  // Este método devuelve la contraseña para el nombre de usuario
  // WSE comprueba si coinciden
  protected override string AuthenticateToken( UsernameToken token )
  {
    string password = "";
    string username = token.Username;
    string CustomCreds =
"http://localhost/WeblogServiceSample/CustomCredentials.xml";

    // leer el documento XML
    XmlTextReader reader = new XmlTextReader( CustomCreds );

    // Buscar usuario
    while( reader.Read() )
    {
      if( (reader.LocalName == "Account") && (reader.GetAttribute(
"username" ) == username) )
      {

```

```

        password = reader.GetAttribute( "password" );
        break;
    }
}
// devolver contraseña
reader.Close();
return password;
}
}

```

En el archivo de configuración (Web.config), se utiliza el elemento *securityTokenManager*, que permite a WSE localizar al *UsernameTokenManager* que llevará a cabo la autenticación.

```

<security>
  <securityTokenManager
    type="WeblogService.WeblogAuthManager, WeblogService"
    xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
    qname="wsse:UsernameToken" />
</security>

```

Para más detalles de la implantación exacta de un ejemplo de este tipo, remitimos al lector al ejemplo (traducido a castellano) mencionado en la nota al pie anterior, o en el excelente ejemplo de **Matt Powell** (ms-help://MS.VSCC.v80/MS.MSDN.v80/MS.NETDEV.v10/en/dnwse/html/programwse2.htm) incluido en la ayuda de Visual Studio 2005.

Extensiones de Servicios Web, versión 3.0

En la actualidad, se encuentra disponible para descarga¹⁷ la versión 3.0 del paquete de extensiones de los servicios Web (*Web Services Enhancements*), que simplifica considerablemente la implantación de la seguridad en este entorno, y ofrece un paquete completo de nuevas opciones de seguridad en cumplimiento con los estándares establecidos por la <http://www.ws-i.org/> (*Web Services Interoperability Organization*).

Aunque su análisis –siquiera superficial- sobrepasaría con creces la extensión de esta obra, cabe destacar las siguientes novedades de cara al desarrollador:

- Seguridad basada en aserciones de Políticas de Configuración
- Soporte en tiempo de diseño en Visual Studio 2005 y .NET Framework 2.0

¹⁷ <http://www.microsoft.com/downloads/details.aspx?familyid=018a09fd-3a74-43c5-8ec1-8d789091255d&displaylang=en>

- Mejoras respecto a la versión anterior en el soporte del estándar *WS-SecureConversation* para establecer sesiones de conversación seguras entre un cliente y un servicio Web. Esto se consigue de dos formas: permitiendo que los servicios puedan instalarse en granjas de servidores, permitiendo que un cliente reestablezca la sesión con un servicio cuyo dominio ha sido reiniciado, y suministrando la capacidad de cancelación y reestablecimiento controlado por parte del servidor.
- Mejoras en la confirmación de firmas digitales (el servicio puede confirmar al cliente que su firma digital se ha verificado)
- Optimización el rendimiento de operaciones criptográficas cuando sólo está presente la clave pública de un par asimétrico, como sucede, por ejemplo, con los certificados.
- Hace innecesaria la actualización en el cliente de los “*security tokens*”, a diferencia de lo que sucedía con la versión anterior.

Capítulo 6: Seguridad en las fases finales

Fase de Pruebas

En esta fase, todo el código funcional se supone terminado. No debiera de haber ninguna parte del programa que estuviese pendiente de codificación, pero se asume que, al entrar en la fase de pruebas finales aparecerán un montón de aspectos a retocar y – con frecuencia- elementos pasados por alto, o pospuestos o solo esbozados.

La recomendación oficial de Microsoft en esta fase es que el grupo de pruebas se divida en dos, como vimos en el capítulo dedicado a MSF. Por un lado, las pruebas de usuario final, que –en ocasiones- desembocan en cambios a nivel de diseño, y las pruebas de carga, donde se proponen *camino*s a seguir recorriendo la aplicación desde un punto original, representado por un actor, hasta un punto final, en el que el actor ha logrado su objetivo mediante el uso del programa.

Las razones que esgrime la propia Microsoft para incluir una revisión de seguridad en esta fase son varias¹⁸:

- Comprobar la ausencia de amenazas de seguridad conocidas
- Verificar el funcionamiento del software a la luz de un usuario malintencionado.
- El software de calidad debe de ser seguro u confiable, y esto siempre ha incluido los aspectos de seguridad, aunque hasta ahora no se haya hecho un tratamiento aparte.

Para ello, es imprescindible crear un Plan de Pruebas de Seguridad.

Creación de un Plan de Pruebas de Seguridad

Dicho plan de pruebas debe de contemplar, entre muchos otros aspectos, al menos, los siguientes:

- Todos los elementos enumerados en el Modelo de Amenazas: especialmente los puntos de ataque.

¹⁸ Curso oficial de “Directrices de seguridad para desarrolladores II (2954^a)

- Posibilidad de que el sitio o aplicación pueda ser sometido a ataques específicos de Inyección SQL.
- Debilidad mostrada por productos de terceros: si conocemos que un producto similar muestra algún fallo de seguridad conocido, deberemos de probar el nuestro en las mismas, o similares circunstancias.
- Aprendizaje "post mortem". Si esta versión es la continuación de otras, o una migración, comprobaremos la presencia de fallas de seguridad mostradas en las anteriores.
- Modelo de usuario variable: hemos de ponernos en el caso de todos los usuarios posibles que tengan una acción significativa sobre el programa. Para ello, la ejecución mediante comandos como "RunAs" y otras técnicas de privilegio mínimo pueden ser de gran ayuda.
- Cuidado con la instalación. La propia Microsoft ha modificado radicalmente en las instalaciones de sus productos respecto a la seguridad: Desde los sistemas operativos (esto es especialmente visible en Windows 2003), hasta herramientas de producción como SQL Server 2005 vienen, ahora con muchos elementos deshabilitados por razones de seguridad.
- En caso de incluirse alguna demo o código de ejemplo, éste, también debe de ser seguro.

A la hora de ponernos manos a la obra con la creación de esta fase, tenemos que tener en cuenta (como siempre en seguridad), que el enemigo número uno son las entradas del usuario. No importa de dónde vengan o cuán inocentes parezcan, pueden constituir un peligro potencial.

En concreto, una lista que enumera la fuente oficial antes citada, incluye los siguientes posibles puntos de entrada de ataques a la seguridad:

- Sockets, canalizaciones (*pipes*), archivos abiertos de forma automática (podrían ser sustituidos por virus o similares), controladores de protocolo, controles ActiveX (que pueden alojarse en el Explorador), llamada a procedimientos remotos (RPC), solicitudes HTTP, que pueden incluir en su cabecera información destinada a obtener información o a atacar directamente a un servidor, correo electrónico y .NET

Remoting, debido a la capacidad que tiene de enviar datos entre dominios de aplicación.

Otros consejos de esta fase pasan por la elección cuidadosa de las herramientas de pruebas, para decidir que el conjunto de datos que se carga en la entrada es apropiado, usar segmentos de código publicados que expongan vulnerabilidades, ser cuidadoso con la calidad del código de prueba que se va a distribuir, y tener especial cuidado con la comunicación entre componentes.

Nota: Este cuidado especial en la comunicación entre componentes implica, muchas veces, un flujo de identidad. Uno de los puntos débiles puede encontrarse en los flujos operativos largos o muy largos, que implican una demora de –a veces– días o meses, hasta que la transacción subyacente puede considerarse concluida. Sea especialmente cuidadoso en este tipo de procesos.

En el momento en que todas las vulnerabilidades conocidas hayan sido solventadas, podemos considerar concluida esta fase. Naturalmente, es muy posible que aparezcan nuevos puntos débiles una vez que la aplicación este en funcionamiento, de ahí que se recomiende la creación complementaria de un plan de contingencia o respuesta a nuevos agujeros de seguridad, que será implementado en la fase siguiente.

Seguridad en la Implantación y Mantenimiento

En el capítulo anterior, concluíamos que el conjunto de pruebas de seguridad no estaría completo si no continuase también en la fase de Implantación y Mantenimiento, para poder hacer frente a las nuevas vulnerabilidades que puedan aparecer una vez instalada la aplicación.

El proceso suele ser el siguiente: el cliente detecta un problema que puede tener que ver con la seguridad, e informa a la empresa creadora responsable. La empresa recibe la alerta, genera una acción de solución del problema y publica nuevamente la solución habiendo llevado un control y abierto un expediente a dicho problema, para su almacenamiento y utilización en reuniones “post-mortem”.

El autor de ese fragmento de código es responsable de su solución, si esta depende de dicho código. El conjunto de acciones de este tipo descubiertas en la fase de implantación, o después de la instalación final, pasan a formar parte una Base de Datos de Conocimiento, que la empresa puede mantener, para aprender de sus propias experiencias pasadas, y tener en cuenta esos factores en otras aplicaciones.

Nota: En algunos casos, puede resultar una práctica interesante publicar los problemas resueltos en la Intranet de Desarrollo, de forma accesible a todos los empleados del departamento. En posteriores desarrollos, tales incidencias podrían ser factores a tener en cuenta en todas las fases, o alimentar con posibles soluciones otros errores.

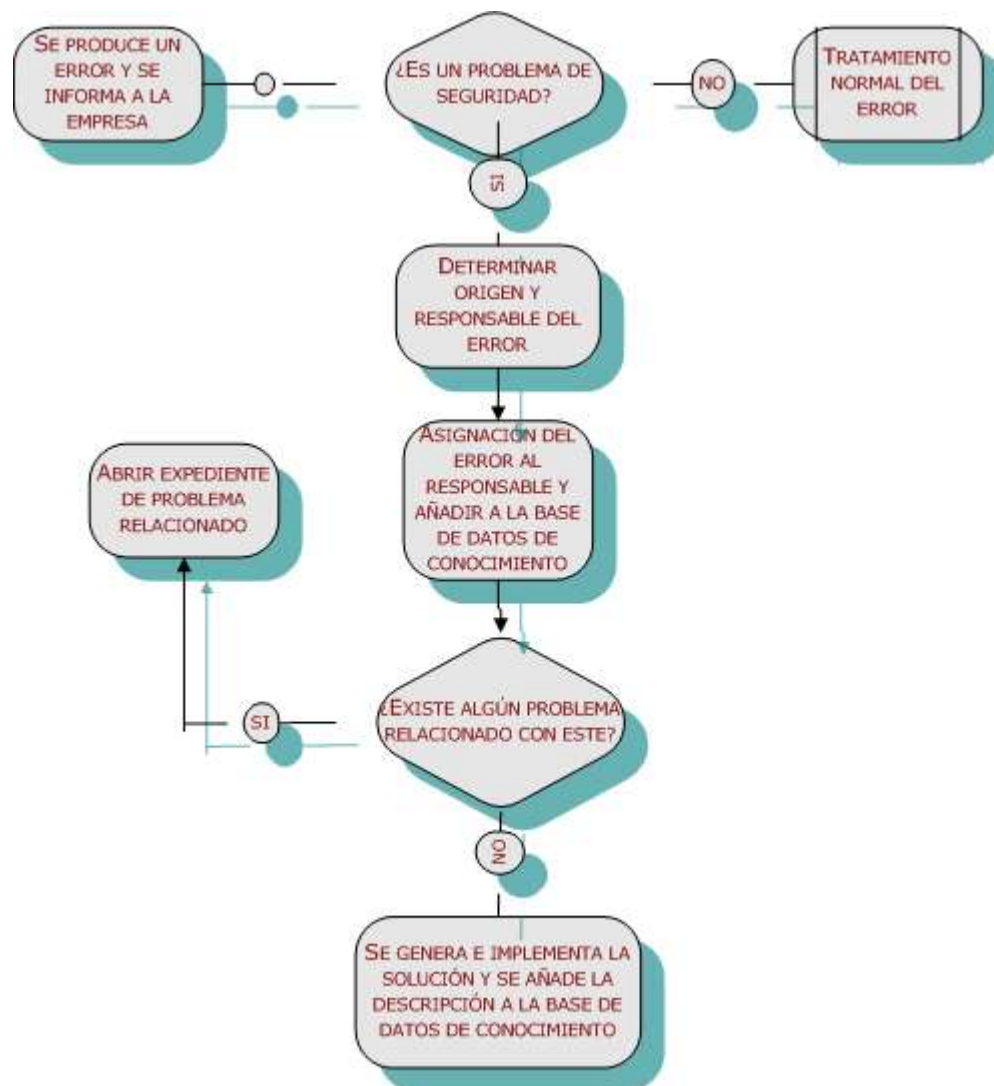


Fig. xx: Esquema de toma de decisiones ante alertas de seguridad

Una buena práctica resulta de la utilización del Registro de Eventos del sistema, como mecanismo de auditoría, no solo para el control de uso de la aplicación, sino como depósito adecuado (es un lugar seguro del sistema al que no todos los usuarios tienen acceso), de las incidencias producidas por su utilización.

Manejo del Registro de Eventos como mecanismo auditor

Aunque en versiones anteriores ya existía soporte de lectura/escritura en el Registro de Eventos del sistema, éste se ha visto mejorado en la versión actual, tanto desde el punto de vista funcional como en algunos aspectos que se refieren a la seguridad, y, en concreto, a los permisos necesarios.

De hecho, la manipulación de este objeto en la versión 2.0, requiere el permiso *EventLogPermission*, y la recomendación es que se cree una entrada específica para la aplicación, en el proceso de instalación, de forma que permitamos al sistema operativo que

refresque su información, antes de que sea escrita la primera entrada

La alternativa clásica consistía en situar el código para la creación de la entrada del Registro de Eventos, en una rutina inicial, de forma que se insertase la primera vez que se ejecutara la aplicación, o como una opción de la ventana de configuraciones de la aplicación. En éste último caso, podríamos utilizar los métodos *WriteEntry* y *WriteEvent*, para escribir entradas personalizadas, una vez detectada la presencia del origen de entradas inicial:

```
// Se crea al origen si no existe ya
if(!EventLog.SourceExists("OrigenDeEntradas"))
{
    EventLog.CreateEventSource("OrigenDeEntradas", "Entrada");
}

// Crea una instancia de EventLog y le asigna su origen.
EventLog RegEventos = new EventLog();
RegEventos.Source = "OrigenDeEntradas";

// Escribir información de soporte en el Registro.
RegEventos.WriteEntry("Se ha producido un intento de hack");
```

Donde si optamos por utilizar *WriteEvent*, disponemos de la opción de escritura múltiple, incluyendo información binaria.

Preparación del Registro de Eventos en la instalación

Otra opción recomendable, consiste en crear una entrada de Registro usando la clase *EventLogInstaller*, dentro de una clase separada, que registre fragmentos de instalación (objetos *Installer*), en un instalador personalizado que puede lanzarse como un proceso externo al programa, o como una opción separada, permitiendo en ambos casos la vuelta atrás de todo el proceso, o la desinstalación posterior.

Nota: Pasar un objeto *EventLog* como argumento a un código de confianza parcial, puede generar un problema de seguridad, por lo que se recomienda que, tanto éste, como sus objetos relacionados *EventLogEntry* e *EventLogEntryCollection*, no se pasen nunca a código de confianza inferior al de su creación.

Esto permite la preparación de un sistema de auditoría basado en la creación de una clase de apoyo para auditoría de excepciones de seguridad. Esto es, si se produce cualquier excepción que asociemos con la seguridad, se transfiere el control a un método estático de la clase, encargado de la escritura, y –opcionalmente– de tomar medidas alternativas en función del tipo de excepción producida.

Auditoría e instaladores personalizados

La creación de instaladores personalizados es una de las características más flexibles e interesantes de .NET respecto a los procesos de distribución final. Aparte de los proyectos de instalación, –de sobra conocidos– la clase *Installer*, permite crear tantas clases heredadas como necesitemos, cada una con uno o más propósitos de instalación personalizados (fragmentos de instalación, vinculados a procesos concretos).

Dichas clases, pueden formar parte del ensamblado final de la aplicación (o de uno separado de ella, que se distribuya conjuntamente), y pueden ser ejecutados en su conjunto cuando se llama a la utilidad del SDK ***InstallUtil.exe***. Tras la llamada, todas las instancias de clases derivadas de *Installer*, son buscadas y llamadas secuencialmente.

Si queremos automatizar todavía más estas posibilidades podemos hacer que el proyecto de instalación llame a la utilidad en alguna de las acciones programables personalizadas que podemos incluir en cualquier proyecto de instalación.

Por ejemplo, para preparar el registro con una nueva entrada de auditoría para el seguimiento de sucesos en una aplicación ficticia llamada Facturación, podríamos incluir el siguiente código:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Configuration.Install;
using System.Diagnostics;
namespace Facturacion
{
    [RunInstaller(true)]
    public partial class InstaladorFacturación : Installer
    {
        private EventLogInstaller instaladorRegEventos;
```

```

public InstaladorFacturación()
{
    // Crea una instancia de un EventLogInstaller.
    // Solo es válida en la máquina local
    instaladorRegEventos = new EventLogInstaller();

    // Establece el origen del EventLog
    instaladorRegEventos.Source = "EntradaFacturacion";

    // Asigna la entrada a utilizar por nuestra aplicación
    instaladorRegEventos.Log = "LogFacturacion";

    // Finalmente, lo añadimos a la colección de instaladores
    // predeterminada de esta clase
    this.Installers.Add(instaladorRegEventos);
}
}
}

```

El funcionamiento es el siguiente: Cuando creamos una instancia de una clase derivada de *Installer*, las acciones incluidas en el constructor se pueden invocar, bien por la utilidad *InstallUtil* antes citada, o por la clase *AssemblyInstaller* (del espacio de nombres *System.Configuration.Install*), ejecutando todos los objetos almacenados en la colección *Installers* (del tipo *InstallersCollection*), que la clase contiene de forma predeterminada.

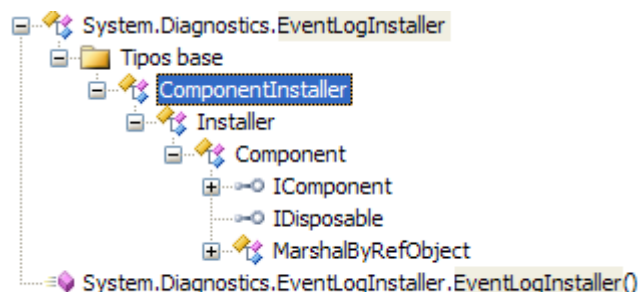


Fig xx: Jerarquía de herencia de la clase EventLogInstaller

Como vemos en el gráfico adjunto, la clase *EventLogInstaller*, a su vez, hereda de *Installer*, y suministra un método sencillo de manipulación del Registro de Eventos del sistema en la fase de instalación, o posteriormente, si decidiéramos lanzar el proceso mediante la clase *AssemblyInstaller*, a través de su método *Install*, que carga un ensamblado y lanza todos los instaladores contenidos en él.

En esta segunda opción, podemos realizar esa misma operación sin necesidad de la llamada a *InstallUtil*, solo mediante código con una secuencia similar a la siguiente (suponemos que *btnLlamarInstalador_Click*, es el procedimiento de evento Click de un botón ubicado en una opción de configuración de la aplicación):

```

private void btnLlamarInstalador_Click(object sender, EventArgs e)
{
    IDictionary Estado = new Hashtable();
    try
    {
        // Establecemos el argumento de línea de comandos
        // para el fichero de log
        string[] CadenaLog = new string[1];
        CadenaLog[0] = "/logfile=Seguimiento.log";

        // Crea un objeto de la clase 'AssemblyInstaller'
        AssemblyInstaller InstaladorDeEnsamblados = new
            AssemblyInstaller();

        // Establece las propiedades requeridas por el Installer
        InstaladorDeEnsamblados.Path = "Facturacion.exe";
        InstaladorDeEnsamblados.CommandLine = CadenaLog;
        InstaladorDeEnsamblados.UseNewContext = true;

        // Limpiar el objeto 'IDictionary'
        Estado.Clear();

        // Invocar las acciones del 'InstaladorDeEnsamblados'
        // llamando al método Install()
        InstaladorDeEnsamblados.Install(Estado);

        // Confirmar instalación mediante el método Commit
        // si todo va bien
        InstaladorDeEnsamblados.Commit(Estado);
    }

    catch (Exception Ex)
    {
        MessageBox.Show("Fallo en el proceso de instalación: " +
            Ex.Message);
        InstaladorDeEnsamblados.Rollback(Estado);
    }
}

```

Solo hay una condición especial en este proceso. El estado en que se encuentra la instalación se persiste mediante un objeto que implemente la interfaz *IDictionary*, que se ha inicializado mediante una tabla *hash* al comienzo. Ese estado, se utilizará como argumento posterior en las llamadas a *Install*, *Commit* (y *Rollback*, si fuera necesario).

Tras crear una matriz de cadenas para el almacenamiento de los nombres utilizados por la clase *Installer* para los ficheros Log de seguimiento de la instalación, se crea un objeto *AssemblyInstaller*, y se procede a la instalación mediante la invocación de los métodos *Install()* (y si nada falla

en el proceso), *Commit()* al objeto de dar los cambios realizados por buenos. Si algo falla, invocamos al método *Rollback*, pasándole su estado.

Instalar la configuración de seguridad

Las clases de soporte del espacio de nombres *System.Security*, permiten, además, realizar instalaciones o configuraciones personalizadas de aspectos de seguridad tales como las Políticas de Máquina, manejo de Grupos de Permisos, comprobación de validación de usuarios y muchas otras.

Lo siguiente es un mecanismo personalizado para modificar la configuración de seguridad de una máquina.

```
public class InstaladorSeguridad
{
    public InstaladorSeguridad() { }

    string NombreGrupoCodigo = "Grupo Facturacion";
    string DescGrupoCodigo = "Grupo para seguridad en Facturacion";

    public void InstalarSeguridad()
    {
        // Primero necesitamos la Política de la máquina
        // que es donde haremos los cambios
        PolicyLevel PoliticaDeMaquina =
            BuscarNivelDePolitica("Machine");

        if (null == PoliticaDeMaquina)
        {
            // Esto es por seguridad; no debería suceder nunca
            throw new ApplicationException(
                "No se pudo encontrar la política de la máquina");
        }

        // Necesitamos añadir un conjunto de permisos con nombre
        // que incluya cualquier permiso que queramos otorgar
        NamedPermissionSet cpn = new
            NamedPermissionSet("PermisosFacturacion",
                PermissionState.None);
        cpn.Description = "Conjunto de permisos para el programa de
            Facturación";

        // TODO: Añadir todos los permisos que necesite Facturación
        cpn.AddPermission( new DataProtectionPermission(
            PermissionState.Unrestricted));
        cpn.AddPermission(new SqlClientPermission(
            PermissionState.Unrestricted));
        cpn.AddPermission(new FileIOPermission(
            FileIOPermissionAccess.Read,@"C:\Facturacion"));
    }
}
```

```

// Añadir nuestro conjunto de permisos con nombre a nivel de
// Política de Máquina
// Todavía no se ha grabado nada (se hace al final)
try
{
    PoliticaDeMaquina.AddNamedPermissionSet(cpn);
}
catch
{
    // Nombre duplicado- actualizamos el existente
    PoliticaDeMaquina.ChangeNamedPermissionSet(cpn.Name,
        cpn);
}

// finalmente guardar todos los cambios de forma transaccional.
SecurityManager.SavePolicyLevel(PoliticaDeMaquina);
}

PolicyLevel BuscarNivelDePolitica(string NivelDeseado)
{
    IEnumerator EnumeradorNiveles =
        SecurityManager.PolicyHierarchy();
    PolicyLevel encontrado = null;
    while (EnumeradorNiveles.MoveNext())
    {
        PolicyLevel Nivel = (PolicyLevel)EnumeradorNiveles.Current;
        if (NivelDeseado == Nivel.Label)
        {
            encontrado = Nivel;
        }
    }
    return encontrado;
}

} //Fin de la clase

```

El código adjunto establece un conjunto de 3 permisos para el programa ficticio de Facturación, que hemos estado usando como ejemplo. Primero, busca la Política de la Máquina, (método *BuscarNivelDePolitica*, al que pasamos el nivel que deseamos). Crea un conjunto de permisos con nombre, y los añade al objeto encontrado.

Finalmente, se graba el objeto *PoliticaDeMaquina* en la configuración de la máquina mediante una llamada al método *SavePolicyLevel*, del objeto gestor de seguridad (*SecurityManager*), pasándole como argumento el propio objeto.

Si posteriormente, comprobamos el estado de la ejecución, veremos que se ha creado una nueva entrada con los 3 permisos solicitados:

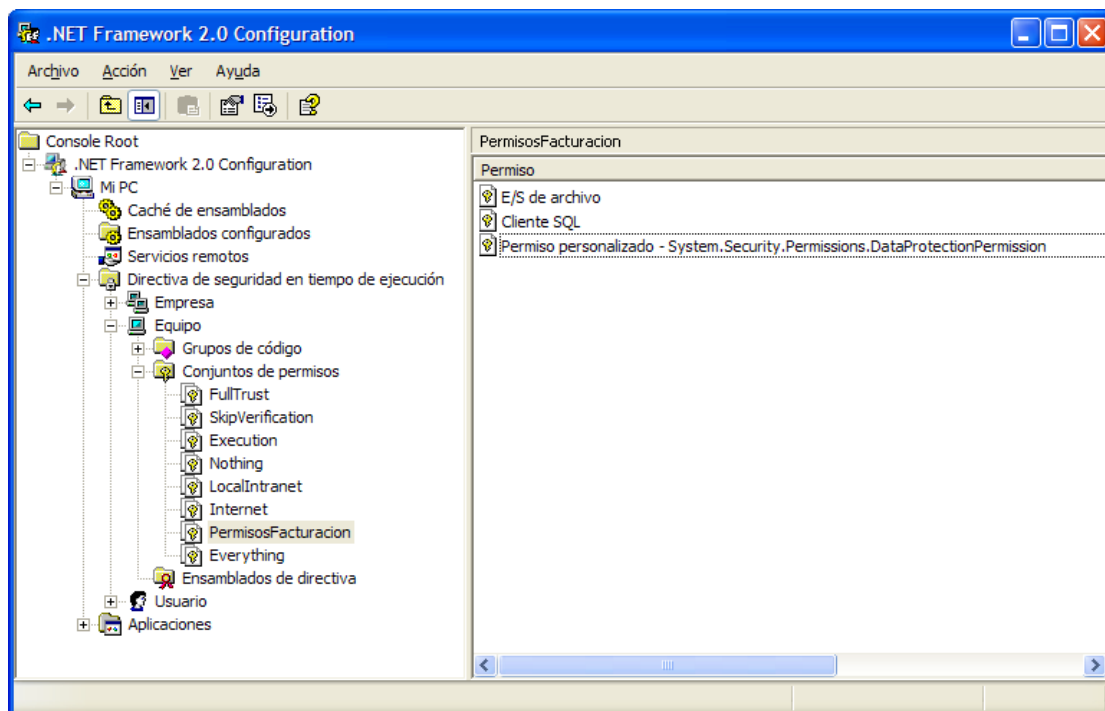


Fig: Visor de Configuración de seguridad, con los nuevos permisos instalados

De la misma forma, podemos comprobar, que los permisos establecidos corresponden con las especificaciones del código: por ejemplo, inspeccionando el permiso de E/S de archivos sobre el directorio C:\Facturacion.

Si fuera necesario, podríamos crear también grupos basados en claves públicas, vinculando éstos con el nombre seguro de un ensamblado, mediante la clase *StrongNameMembershipCondition*, así como otro sinfín de opciones de comprobación, borrado y manejo de permisos nuevos y existentes, que permitan facilitar y garantizar que -también en la fase de implantación y mantenimiento- el control de la seguridad se lleve a cabo de forma organizada y controlada.

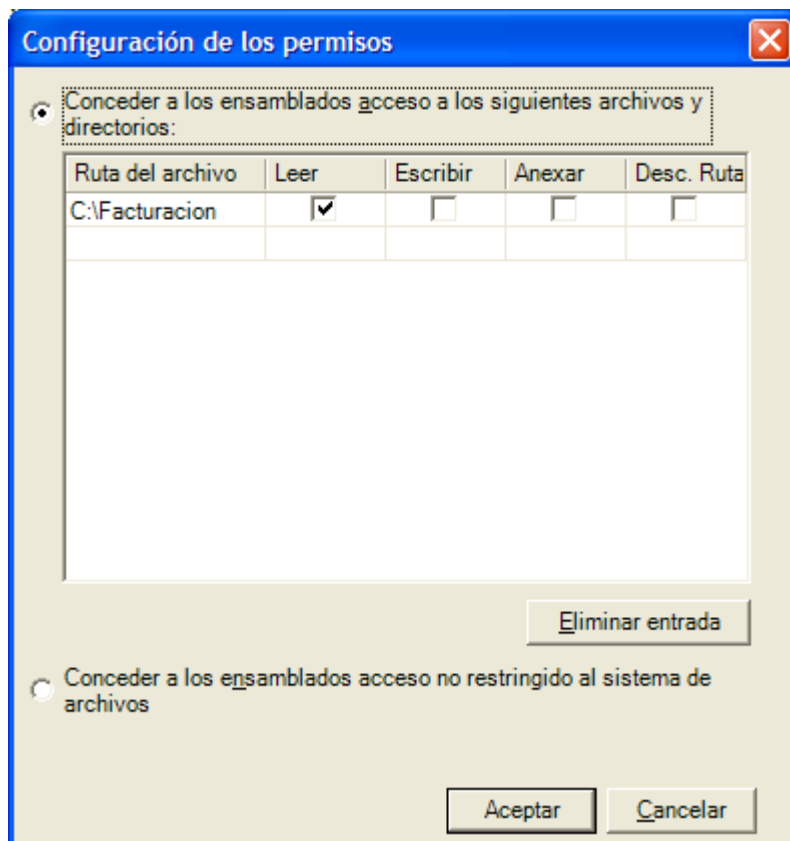


Fig.: Permisos de E/S para el permiso Facturación

Conclusión

La seguridad no es un "algo" a incluir al final de las aplicaciones, como un añadido molesto, pero necesario. Se trata de un requisito, una necesidad de muchas aplicaciones de hoy, si queremos decir que estamos trabajando de acuerdo con los niveles de calidad que los clientes exigen en el software actual. Y tampoco se trata de una preocupación que solo concierne a los grandes productores de software, sino de una labor que debería asumirse para cualquier programa que salga del ámbito de la mera utilidad personal.

Las recomendaciones y prácticas descritas en este cuaderno técnico, pretenden servir de introducción al desarrollador atento, y al Jefe de Proyecto responsable, para que, mediante las nuevas tecnologías aportadas por .NET Framework 2.0, le permitan, con coherencia, pero con sencillez, la implantación de esas técnicas de seguridad que garanticen que el software que sale de sus manos cumpla con unas normativas de calidad cada vez más exigentes.

Programación segura con .NET Framework

Marino Posadas

Se trata de una obra introductoria a los conceptos de seguridad en las aplicaciones y a los conceptos de codificación segura. Utilizando los principios de Microsoft Solutions Framework como marco de trabajo y las recomendaciones oficiales de buenas prácticas, se recorre el ciclo de vida de una aplicación, haciendo especial hincapié en las prácticas de seguridad, los modelos de ataques, y las formas correctas de prevención.

Los **Cuadernos Técnicos de dotNetManía** son una serie de pequeños libros enfocados a temas concretos para programadores y arquitectos de software de la plataforma .NET. Cubren el hueco existente entre artículos muy específicos en una revista especializada como **dotNetManía** o los grandes libros sobre temas genéricos.



Marino Posadas es Redactor Jefe de **dotNetManía**. Trabaja como formador y consultor en **Alhambra-Eidos** y es Microsoft Most Valuable Professional en Visual C#, además de titulado MCSD, MCAD, MCT y MSFP. Antes de esta obra, ha colaborado con varios compañeros de trabajo en la elaboración de otros cuatro títulos, y esta es su segunda obra en solitario. Conferenciante en diversos eventos organizados por **Alhambra-Eidos**, **Microsoft** y varias universidades españolas y extranjeras, se interesó por .NET Framework desde las primeras versiones "alfa" y mantiene una Web de soporte de sus actividades de formación en el sitio www.EIAveFenix.net.



Con la colaboración de:



3

Programación segura con .NET Framework • Marino Posadas